Architectural Metapatterns

The Pattern Language of Software Architecture

v 0.9 (12-2024)

Denys Poltorak

What this book is

- A compendium of one or two hundred architectural patterns.
- A classification (taxonomy) of architectural patterns.
- The first large generic *pattern language* since volume 4 of *Pattern-Oriented Software Architecture*.
- A step towards the *ubiquitous language* of software architecture.
- Knowledge should be free. (*Let's all love FOSS!*)

What this book is not

- A review of modern technologies.
- A cookbook (decision tree) of ready-made recipes.
- A guide through system design interviews.
- A detailed pattern catalogue.
- A polished commercial-grade textbook.

Opentowork

I am looking for a good job supportive of this book. If I don't find any, the book will be abandoned.

Short table of contents

Full table of contents

About this book

When I was learning programming, there was *GoF*. The book promised to teach software design, and it did to an extent with the example provided. However, the patterns described were mere random tools that had little in common. After several years, having reinvented *Hexagonal Architecture* by the way, I learned about *POSA*. The series had many more intriguing patterns, and promised to provide a *system of patterns* or a *pattern language*, but failed to build an intuitive whole. Then there were several specialized pattern books for *DDD* and microservices. There was the *Software Architecture Patterns* primer by Mark Richards. Its simplicity felt great, but it had only 5 architectural styles, while his next book, *Fundamentals of Software Architecture*, dived too deep into practical details and examples to be easy to grasp.

Now, having leisure thanks to the war, burnout, unemployment and depression I got a chance to collect architectural patterns from multiple sources and build a taxonomy of architectures. My goal was to write the very book I lacked in those early years: a shallow but intuitive overview of all the software and system architectures used in practice, their properties and relations. I hope that it will be of some help both to novice programmers as a kind of a primer on the principles of high-level software design and to adept architects by reminding them of the big picture outside of their areas of expertise.

The book is mostly technology-agnostic. It does not answer practical questions like "Which database should I use?" Instead it inclines towards the understanding of "When should I use a shared database?" Any specific technologies ~~are easy to google~~ can be found ~~over the Internet~~ somewhere in the Noosphere.

This book started as a rather small project to prove that patterns can be intuitively classified (*These nightmarish creatures* can *be felled! They* can *be beaten!*) but grew into a multifaceted compendium of a hundred or so architectures and architectural patterns. It is grounded in the idea that software and system architecture evolves naturally, as opposed to being scientifically planned. Thus, the architectures may exhibit fractal features, just like those in biology – just because the set of guidelines and forces remains the same for most systems ranging from low-end embedded devices to world-wide banking networks. Moreover, in some cases we can see the same approaches applied to hardware design.

The idea of unifying software and system architecture is heretical, I am aware of that. Still, the industry is in the stage of alchemy these days: the same things are sold under multitudes of names, being remarketed or reinvented every decade. If this book manages to provide a set of guidelines, similar to those of biology (a bat is a mammal, thus it should run on its four, while ostriches, as birds, must fly to Europe each spring), I will be happy with that. *Science makes progress funeral by funeral*.

The latest version of the book is available for free on github. As there is none who practiced all the known architectures, it should be full of mistakes. I rely on your goodwill to correct them and improve the text. Any critical reviews are warmly welcome, please email me or connect on LinkedIn. An early version of each chapter is available on Medium as well.

Structure of the book

The first chapter explains the main idea that makes this book different from others. The following chapters of the first part touch several general topics which are referenced throughout the book.

Further four parts iterate over clusters of closely related architectural patterns, starting with the simplest one, namely *Monolith*, then heading towards more complex systems that may be derived from *Monolith* by repeatedly dissecting it with interfaces. Each chapter there describes a cluster of related patterns with its benefits, drawbacks, known names and subtypes, adds in few references to books and websites, and summarizes the ways the patterns can be transformed into other architectures.

The sixth part of the book is analytics – the fruits of the classification of patterns from the earlier parts. It will probably grow in future releases.

Finally, there are appendices. Appendix B is the list of books referenced, Appendix E contains many detailed evolutions of patterns, and Appendix I is the index of patterns found in the book.

The architectural religions

There are several schools of software architecture:
1. The believers in SOLID.
2. The followers of eight qualities, five views and as-many-as-one-gets certifications.
3. The aspirants to the nameless way of patterns.

In my opinion:
1. SOLID is a silver bullet that tends to produce a DDD-layered kind of Hexagonal Architecture. It lacks the agility of pluralism found with evolutionary ecosystems.
2. Architectural frameworks are overcomplicated thus hard to understand and inflexible.
3. Patterns are a kind of toolbox a mechanician is often seen carrying around. A skilled craftsman knows best uses of his tools, and can invent new instruments if something is missing in the standard toolset. However, the toolset's size should be limited for the tools to be familiar to the practitioner and easily carried around.

It is likely that those approaches are best used with systems of various sizes: SOLID is aimed at stand-alone application design while the heavy frameworks and certifications suit distributed enterprise architectures. In such a worldview patterns span everything in between the poles.

What's wrong with patterns

Too much information is no information or, as they say, *what is not remembered never existed*. There are literally thousands of patterns described for software and system architecture. Nobody knows them all and nobody cares to know (if you say you do, you must have already read the Pattern Languages of Programs archives. Have you? Neither I). Hundreds of patterns are generated yearly in conferences alone, not to mention books and software engineering websites. Old patterns get rebranded or forgotten and reinvented. This is especially true for the discrepancy between the pattern names in software architecture and system architecture. The new *N-tier* is the old good *Layers* under the hood, isn't it?

This undermines the original ideas that brought in the patterns hype:

1. *Patterns as a ubiquitous language*. Nowadays similar, if not identical, patterns bear different names, and some of them are too obscure to be ever heard of (see [the PLoP archives](#)).
2. *Patterns as a vessel for knowledge transfer*. If an old pattern is reinvented or plagiarized, most of the old knowledge is lost. There is no continuity of experience.
3. *Pattern language as the ultimate architect's tool*. As patterns are re-invented, so are pattern languages. At best, we have domain-specific or architecture-limited (DDD, microservices) systems of patterns. There is no the unified vision which pattern enthusiasts of old promised to provide.

Have we been fooled?

TLDR

Compare *Firewall* and *Response Cache*. Both represent a system to its users and implement generic aspects of the system's behavior. Both are [*proxies*](#).

Take *Saga Execution Component* and *API Composer*. Both are high-level services that make series of calls into an underlying system – they *orchestrate* it. Both are [*orchestrators*](#).

It's that simple and stupid. We can classify architectural patterns.

Metapatterns

Is there a way to bring the patterns to order? They are way too many, some obscure, others overly specialized.

We can try. On a subset. And the subset should be:

- *Important* enough to matter for the majority of programmers.
- *Small* enough to fit in one's memory or in a book.
- *Complete* enough to assure that we don't miss anything crucial.

Is there such a set? I believe so.

Architectural patterns

[POSA1] defines three categories of patterns:

- *Architectural patterns* that deal with the overall structure of a system and functions of its components.
- *Design patterns* that describe relations between objects.
- *Idioms* that provide abstractions on top of a given programming language.

Architectural patterns are important by definition ("Architecture is about the important stuff. Whatever that is"). Point 1 (*importance*) – checked.

Any system has an internal structure. When its developers talk about "*architectural style*" [POSA1] or draw structural diagrams that usually boils down to a composition of two or three well-known architectural patterns. Choosing architectural patterns as the subject of our study lets us feed on a large body of books and articles that describe similar designs over and over again. Moreover, as soon as a system does not follow the latest fashion, it is widely advertised as a novelty (or its designers are labeled as old-fashioned and shortsighted), thus we may expect to have heard of nearly all of the architectures which are used in practice. Point 3 (*completeness*) – we have more than enough examples to analyze.

To organize a set of patterns we rely on the concept of

Design space

Design space [POSA1, POSA5] is a model that allocates a dimension for each choice made while architecting the system. Thus it contains all the possible ways for a system to be designed. The only trouble – it is multidimensional, probably infinite and the dimensions differ from system to system.

There is a workaround – we can use a projection from the design space into a 2- or 3-dimensional space which we are more comfortable with. However, projecting is a loss of information. Counterintuitively, that is good for us – similar architectures that differ in small details become identical as soon as the dimensions they differ in disappear. If we could only find 2 or 3 most important dimensions that apply equally well to each pattern in the set that we want to research, that is architectural patterns, which cover all the known system designs.

Structure defines architecture

Systems tend to have an internal structure. Those that don't are derogatively called "*Big Balls of Mud*" for their peculiar properties. Structure is all about modules, their roles and interactions. Many architectural styles, e.g. *Layers* or *Pipeline*, are named after their structures, while others, like *Event-Driven Architecture*, highlight some of its aspects, hinting that it is the structure that defines principal properties of a system.

I am not the first person to reach such a conclusion. *Metapatterns* – clusters of patterns of similar structure – were defined shortly after the first collections of design patterns had appeared but they never made a lasting impact on software engineering. I believe that the approach was applied prematurely to analyze the [GoF] patterns, which make quite a random and incomplete subset of design patterns, resulting in an overgeneralization. I intend to plot structures of the complete set of architectural patterns, group patterns of identical structure together (resulting in metapatterns), draw relations between the metapatterns and maybe show how a system's structure defines its properties. Quite an ambitious plan for a short book, isn't it?

Our set of architectural patterns is still not known to be complete, is not small and, moreover, the way structural diagrams are drawn differs from source to source – we cannot compare them unless we make up a universal

The system of coordinates

Inventing a generic coordinate system to fit any pattern's representation, from *Iterator* to *Half-Sync/Half-Async*, may be too hard, but we surely can find something for architectural patterns, as all of them share the scope, namely the system as a whole. Which dimensions an implementation of a system would usually be plotted along?

1. *Abstractness* – there is the high-level business logic and there are low-level details. A single highly abstract operation unrolls into many lower-level ones: Python scripts run on top of a C runtime and assembly drivers; orchestrators call API methods of services, which themselves run SQL queries towards their databases which are full of low-level computations and disk operations.
2. *Subdomain* – any complex system manages multiple subdomains. An OS needs to deal with a variety of peripheral devices and protocols: a video card driver has very little resemblance to an HDD driver or to the TCP/IP stack. An enterprise has multiple departments, each operating a software that fits its needs.
3. *Sharding* – if several instances of a module are deployed, and that fact is an integral part of the architecture, we should represent the multiple instances on the structural diagram.

We'll draw the abstractness axis vertically with higher-level modules positioned towards the upper side of the diagram, the subdomain axis horizontally, and sharding diagonally. Here is an (arbitrary) example of such a diagram:

Abstractness

Frontend
Frontend
Frontend

Reverse proxy

Backend

Domain Model

ORM

OLTP DB

Cache

OLAP DB

Sharding

Subdomain

CQRS

A component

A logical group

⟹ Asynchronous message

=⟩ Optional message

→ Method call or RPC

⟺ 2-way communication

Generic code is white

Business logic is blue

Data is grey
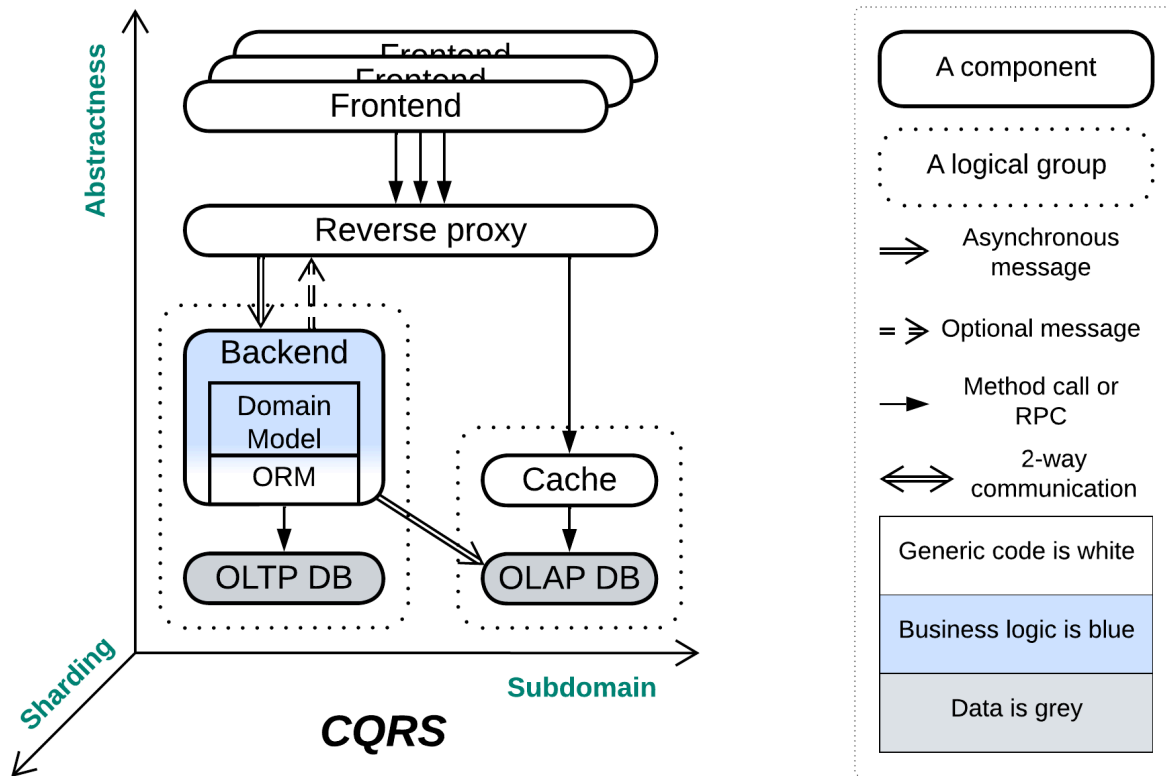
(A structural diagram for CQRS, adapted from Udi Dahan's article, to introduce the notation)

Map and reduce

Now that we have the generic coordinates which seem to fit any architectural pattern, we can start mapping our set of architectural patterns into that coordinate system – the process of reducing the multidimensional design space to the few dimensions of structural diagrams which we were looking for. Then, after filtering out minor details, our hundred-or-so of the published patterns should yield a score of clusters of geometrically equivalent diagrams – just because there are very few simple systems that one can draw on a plane before repeating oneself. Each of the clusters will represent an *architectural metapattern* – a generalization of architectural patterns of similar structure and function.

Let's return for a second to our requirements for classifying a set of patterns. The importance (point 1) of architectural patterns was proved before. The reasonable size of the resulting classification (point 2) is granted by the existence of only a few simple 2D or 3D diagrams (metapatterns). The completeness of the analysis (point 3) comes from, on one hand, the geometrical approach which makes any blank spaces (possible geometries with no known patterns) obvious, on the other – from the large sample of architectural patterns which we are classifying.

Godspeed!

An example of metapatterns

Let's consider the following structure:

There are several
high-level ...

... and one low-level
component

Each high-level
module
communicates with
the low-level module
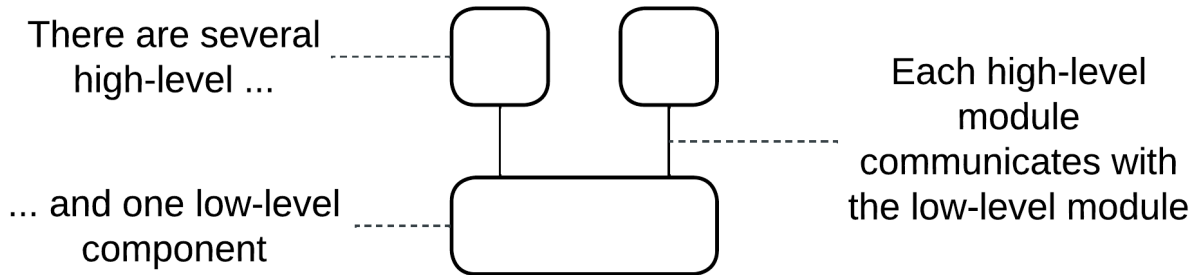
It features two (or more in real life) high-level modules that communicate with/via a lower-level module. Which patterns does it match?

- *Middleware* – a software that provides means of communication to other components.
- *Shared Database* – a space for other components to store and exchange data.
- *Model-View-Controller* – a platform-agnostic business logic with customized means of input and output.
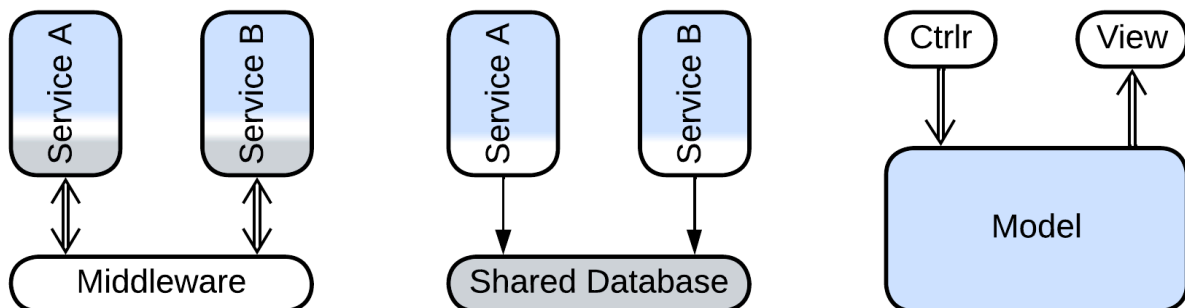
My idea of grouping patterns by structure seems to have backfired – we got three distinct patterns with similar structural diagrams. The first two of them are related – both implement indirect communication, and their distinction is fading as a *middleware* may feature a persistent storage for messages while a table in a *shared database* may be used to orchestrate services. The third one is very different – primarily because the bulk of its code, that is *business logic*, resides in the lower layer, leaving the upper-level components a minor role.

Notwithstanding, each of the patterns we found is a part of a distinct cluster:

- *Middleware* is also known as *(Message) Broker* [POSA1, POSA4, EIP, MP] and is an integral part of *Message Bus* [EIP], *Service Mesh* [FSA], *Event Mediator* [FSA], *Enterprise Service Bus* [FSA] and *Space-Based Architecture* [SAP, FSA].
- *Shared Database* is a kind of *Shared Repository* [POSA4] (*Shared Memory*, *Shared File System*) and the foundation for *Blackboard* [POSA1, POSA4], *Space-Based Architecture* [SAP, FSA] and *Service-Based Architecture* [FSA].
- *Model-View-Controller* [POSA1, POSA4] is a special kind of *Hexagonal Architecture* (aka *Ports and Adapters*, *Onion Architecture* and *Clean Architecture*) which itself is derived from *Plugins* [PEAA] (*Add-ons*, *Plug-In Architecture* [FSA] or *Microkernel Architecture* [SAP, FSA]).

Our tipping a single geometry of structural diagrams revealed a web of 20 or so pattern names that spreads all around. With such a pace there is a hope of exploring the whole fabric which is known as *pattern language* [GoF, POSA1, POSA2, POSA5].

There are three lessons to learn:

- The distribution of business logic is a crucial aspect of structural diagrams.

- Metapatterns are interrelated in multiple ways, forming a pattern language.
- Each metapattern combines several well-established patterns.

What does that mean

Chemistry got the [periodic table](). Biology got the [tree of life](). This book strives towards building something of the kind for software and system architecture. You can say "That makes no sense! Chemistry and biology are empirical sciences while software architecture isn't!" Is it?

Part 1. Foundations

This part defines some ideas which are used occasionally later in the book. Feel free to skip (through) it as you probably know most of them quite well.

Modules and complexity

This chapter is loosely based on [A Philosophy of Software Design](#) by *John Ousterhout* and [my article](#).

Any software system that we encounter is very likely to be too complex to comprehend all at once – the human mind is incapable of discerning a large number of entities and their relations. It tends to simplify reality by building abstractions: as soon as we define the many shiny pieces of metal, glass and rubber as a 'car' we can tell 'highways', 'parkings' and 'passengers' – we live in a world of abstractions which we create. In the same way the software we write is built of services, processes, files, classes, procedures – modules that conceal the swarm of bits and pieces we are powerless against. Let's reflect on that.

Concepts and complexity

Any system comprises *concepts* – notions defined in terms of other concepts. For example, if you are implementing a phonebook, you deal with *first* and *second names*, *numbers*, *sorting* and *search*, which one must always keep in mind for any phonebook-related development task – just because requirements for the phonebook are described in terms of those concepts and their relations.

In the code high-level concepts are embodied as services, modules or directories while lower-level concepts match to classes, API methods or source files.

Concepts are important because it is their number (or the number of the corresponding classes and methods) that defines the *complexity* of a system – the cognitive load developers of the system face. If programmers grasp in detail the behavior of a component they work on they tend to [become extremely productive](#) and are often able to find [simple solutions for seemingly complex tasks](#). Otherwise the development is slow and requires extensive testing because people are unsure of how their changes affect the system's behavior.
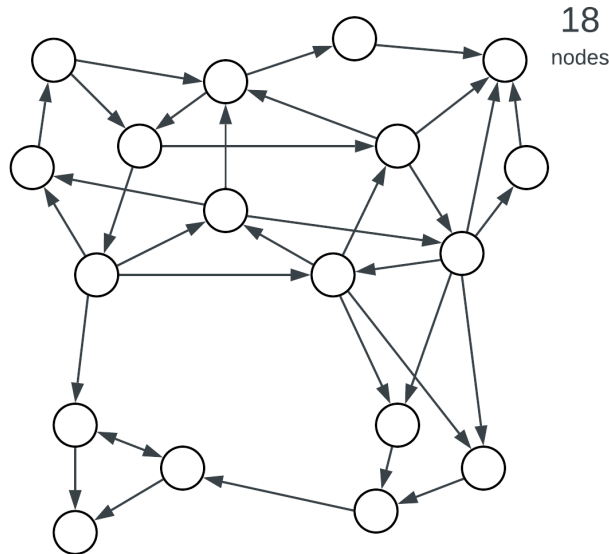
Figure 1: Complexity correlates with the number of entities.

Modules, encapsulation and bounded context

Let's return to our example. As you implement the phonebook you find out that sorting and search are way more complex than you originally thought. Once you prepare to enter the international market you are in deep trouble. Some telephony providers send 7-digit numbers, others use 10 digits, still others – 13 digits (with either "+" or "0" for the first character). German has "ß" which is identical to "ss" while Japanese uses two alphabets simultaneously. Once you start reading standards, implementing all the weird behavior and responding to user complaints you feel that your phonebook implementation is drowning in the unrelated logic of foreign alphabets full of special cases. You need *encapsulation*.

Enter *modules*. A module wraps several concepts, effectively hiding them from external users, and exposes a simplified view of its contents. Introducing modules splits a complex system into several, usually less complex, parts.
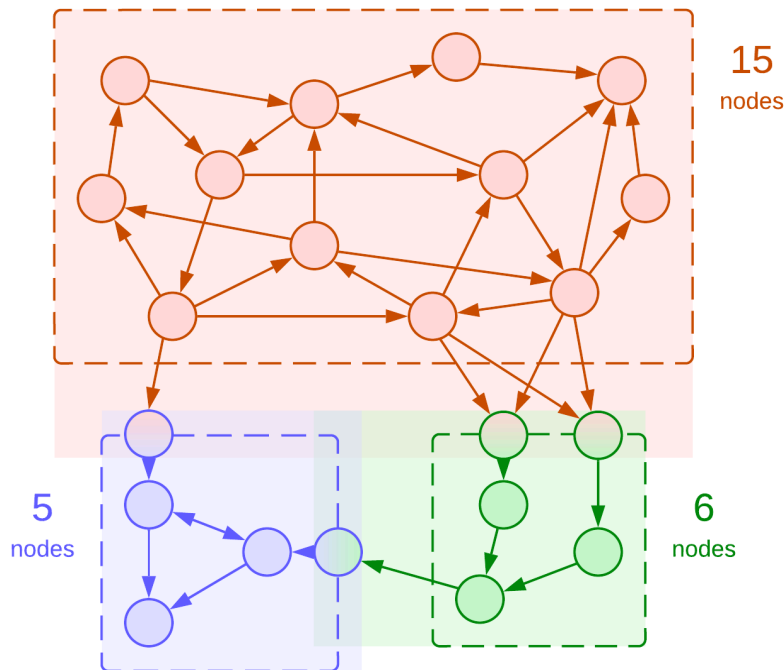
Figure 2: Dividing a system into modules, bounded contexts highlighted.

The diagram has several points of notice:
- Modules create new concepts for their *public APIs*.
- The API entry points add to the complexity of *both* the owner module and its clients.
- The total number of concepts in the system has increased (from 18 to 22) but the highest complexity in the system has dropped (from 18 to 15).

Here we see how introducing modularity applies the divide and conquer approach to lessen the cognitive load of working on any part of a system at the cost of a small increase in the total amount of work to be done.

In our phonebook example the peculiarities (including case sensitivity) of the locale-aware string comparison and alphabetical sorting of contact names should better be kept behind a simple string comparison interface to relieve the programmer of the phonebook engine of the complexity of supporting foreign languages.

Modules represent *bounded contexts* [DDD] – areas of the knowledge about a system that operate distinct sets of terms. In the case of phonebook the *collation* and *case sensitivity* do not matter for the phonebook engine – they are defined only in the context of language support. On the other hand, *matching a contact by number* is not defined in the language support module – that term exists only in the phonebook engine. It is the complexity of the current bounded context that a programmer struggles with.

Apart of dividing a problem into simpler subproblems modules open the path to a few extra benefits:
- *Code reuse.* A well-written module that implements something generic may be used in multiple projects.
- *Division of labor*. Once a system is split into modules and each module is assigned a programmer, development is efficiently parallelized.
- *High-level concepts*. Some cases allow for merging several concepts of the original problem into higher-level aggregates, further reducing the complexity:
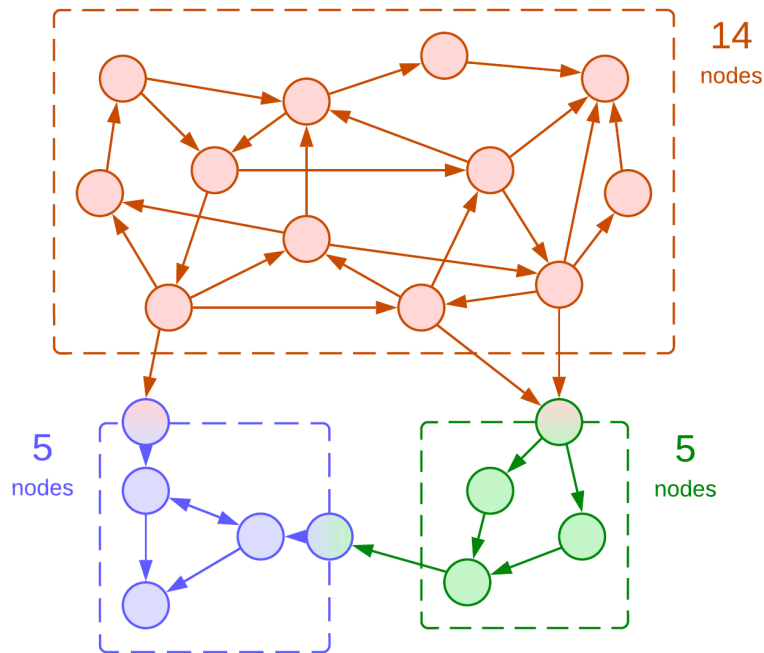
Figure 3: Merged two API concepts of the green module.

For example, the original definition of a phonebook contained *first name* and *second name*. Once we separate the language support into a dedicated module, we may find out that various locales differ in the way they represent contacts: some (USA) use 'first name + second name' while others (Japan) need 'second name + first name'. If we want to abstract ourselves from that detail, we should use a new concept of *full name* which conjoins first and second names in a locale-specific way. Such a change actually simplifies some of the phonebook's representation logic and code as it replaces two concepts with one.

Coupling and cohesion

We need to learn a couple of new concepts in order to use modules efficiently:

Coupling is a measure of the number (density) of connections between modules relative to the modules' sizes.

Cohesion is a measure of the number (density) of connections inside a module relative to the module's size.

The rule of thumb is to aim for **low coupling and high cohesion**, meaning that each module should encapsulate a cluster of related (intensely interacting) concepts. This is how we have split the system in figures 2 and 3. Now let's see what happens if we violate the rules:
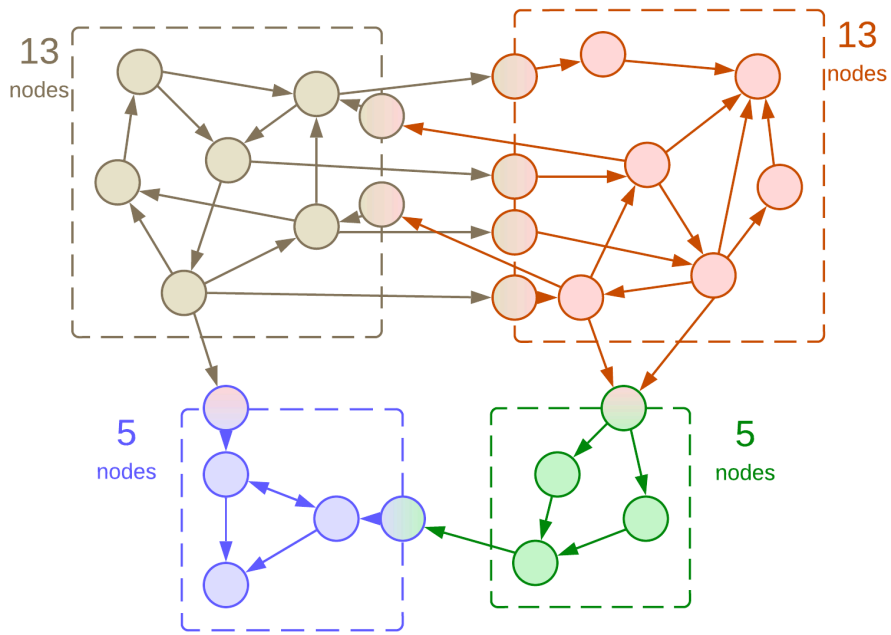
Figure 4: The upper modules are tightly coupled.

Splitting a cohesive module (a cluster of concepts that interact with each other) yields two strongly coupled modules. That's what we wanted, except that each of the new modules is nearly as complex as the original one. Meaning, that we now face two hard tasks instead of one. Also, the system's performance may be poor as communication between modules is rarely optimal, and we've got too much of that.
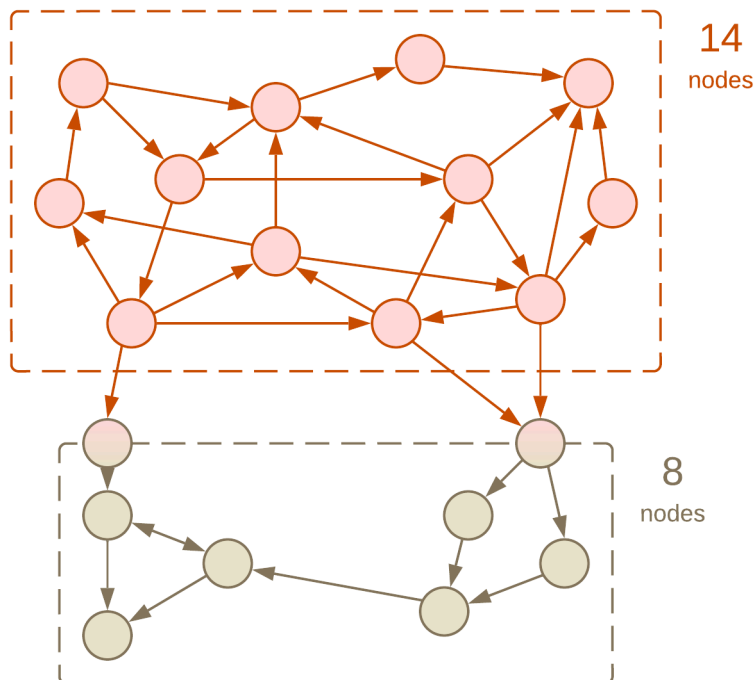
Figure 5: The lower module has low cohesion.

What happens if we put several clusters of concepts in the same module? Nothing too evil for small modules – the module gets higher complexity than each of its constituents, but

lower than their sum. In practice, multiple unrelated functions are often gathered in a 'utils' or 'tools' file or directory to alleviate *operational complexity*.

Development and operational complexity

What we discussed above is *structural* or *development complexity* – the number of concepts and rules inside a bounded context. However, we also need to understand operations and components of the system as a whole, leading to *operational* or *integration complexity*:

- Does this new requirement fit into an existing module or does it call for a dedicated one?
- Which libraries with known security vulnerabilities do we use?
- Is there any way to cut our cloud services cost?
- 1% of requests time out. Would you please investigate that?
- My team needs to implement this and that. Do we have something fit for reuse?
- What the **** is [that global variable](#) about?
- Do we really need this code in production?
- I need to change the behavior of that shared component a little bit. Any objections?

When there are hundreds or thousands of modules deployed nobody knows the answers. That's similar to the case of one needing to do something under Linux: hundreds of tools are pre-installed and thousands more are available as packages, but the only real way forward is first googling for your needs, then trying two or three recipes from the search results to see which one fits your setup. Unfortunately, Google does not index your company's code.

Composition of modules

A module may encapsulate not only individual concepts, but also other modules. That is not surprising as an OOP class is a kind of module – it has public methods and private members as well. Hiding a module inside another one removes it from the global scope, decreasing the operational complexity of the system – now it is not the system's architect but the maintainer of the outer module who must remember about the inner module. On one hand, that builds a manageable hierarchy in both the organization and the code. On the other hand, code reuse and many optimizations become nearly impossible as internal modules are hardly known organization-wide:
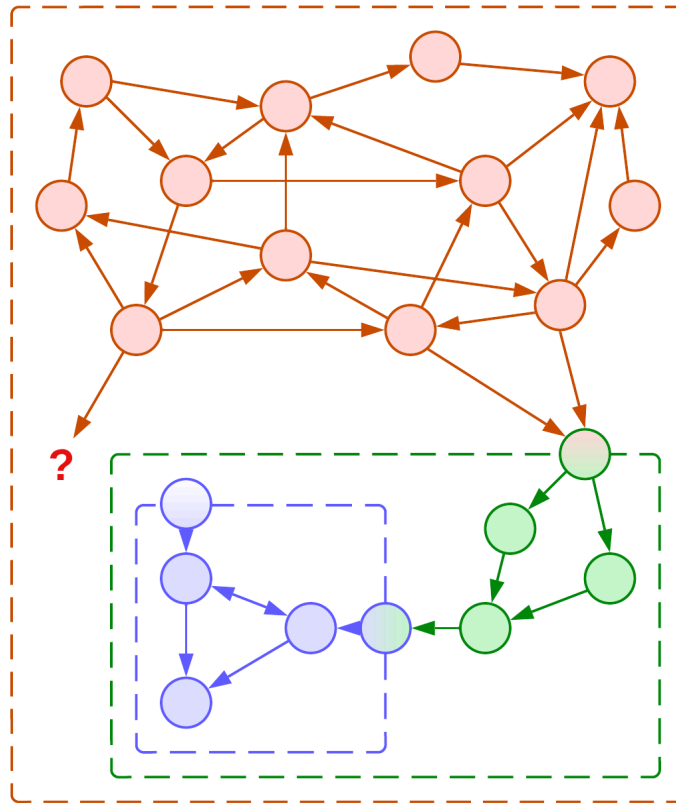
Figure 6: Composition of modules prevents reuse.

If the functionality of our internal module is needed by our clients, we have two bad options to choose from:

Forwarding and duplication
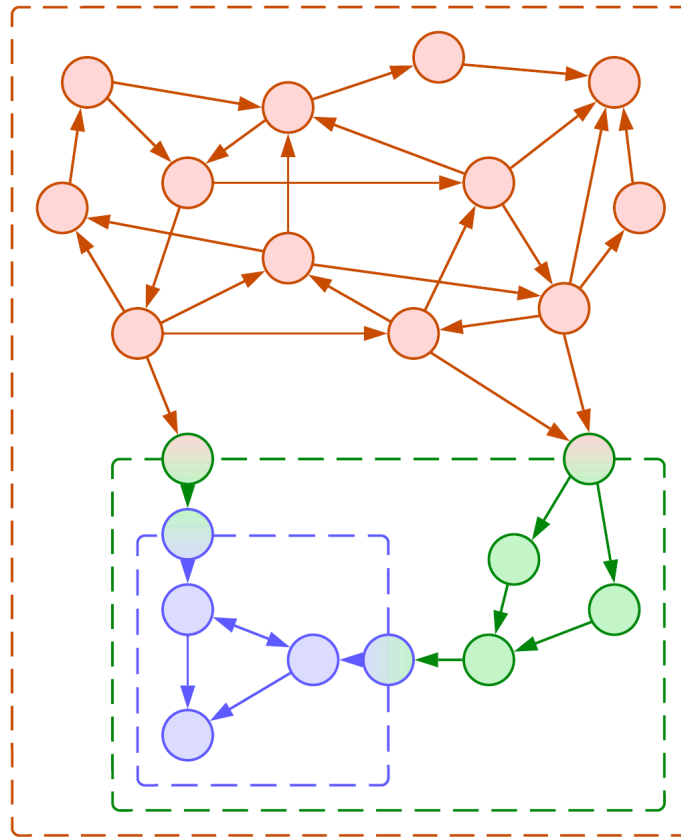
Figure 7: Forwarding the API of an internal module.

We can add the API of a module we encapsulate to our public API and forward its calls to the internal module. However, that increases the complexity and lowers the cohesion of our module – now each client of our module is also exposed to the details of the methods of the module we have encapsulated even if they are not interested in using it.
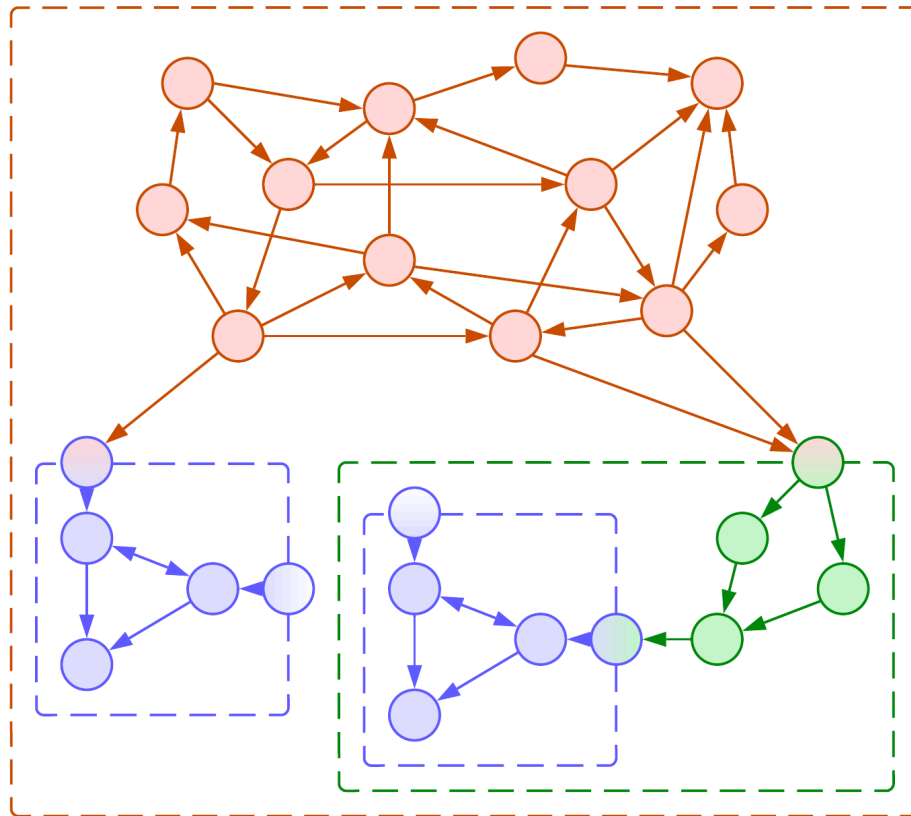
Figure 8: Duplicating an internal module.

Another bad option is to let the clients that need a module that we encapsulate duplicate it and own the copies as their own submodules. This relieves us of any shared responsibility, lets us modify and misuse our internals in any way we like, but violates a couple of rules of common sense.

Both approaches, namely keeping all the modules in the global scope and encapsulating utility modules through composition, found their place in history [FSA]. *Service-Oriented Architecture* was based on the idea of reuse but fell prey to the complexity of its *Enterprise Service Bus* which had to account for all the interactions (API methods) in the system. In reaction, the *Microservices* approach turned the tide in the opposite direction: its proponents disallowed sharing any resources or code between services to enforce their decoupling.

Summary

Complexity is the number of *concepts* and their relations that one should remember to work efficiently. A *module* hides some concepts from its users but creates new concepts (its *interface*). *Coupling* is the measure of dependencies between modules, while *cohesion* is the same for concepts inside a module. We prefer *low coupling and high cohesion* to group related things together.

Having too many modules is a trouble for the system's maintainers. A module may contain other modules. When a client wants to use a submodule, the wrapping module may extend its interface to forward client's requests to the submodule or the client may deploy a copy of the submodule for its use. Both approaches gave rise to prominent architectures.

Forces, asynchronicity and distribution

Many systems rely on asynchronous communication between their components or are distributed over a network. Why is dividing a system into modules not enough in real life?

Requirements and forces

Any system is built to meet a set of (explicit or implicit) *requirements*. As a bare minimum, you as a programmer must have a dim vision of how your software is expected to operate. As a maximum, business analysts bring you several volumes of incomprehensible documentation they wrote for the sole purpose of making you practice [DDD].

Some requirements are *functional*, others are *non-functional*.

Functional requirements describe what the system must do: a night vision device must be able to represent heat radiation as a video stream; a multiplayer game must create a shared virtual world for users to interact with over a network; a tool for formatting floppies … well, must format floppies.

Non-functional requirements, also known as *forces*, define expected properties of the system and it is said that they drive architectural decisions [POSA1, POSA5]. They may be formulated or implied: our game should be fast enough and stable enough. A medical application should be extremely well-tested. An online shop should provide an easy way to add new goods. Notice all those "fast enough", "stable enough", "well" and "easy". Sometimes they form an SLA with numbers: your service should be available 99.999% of the time.

Let's take an example.

A night vision surveillance camera may spend seconds compressing its video stream to limit the required network bandwidth – this kind of a system sacrifices low latency in favor of low traffic. The device will need a fast CPU (probably a DSP) and lots of RAM to store multiple frames for efficient compression.

A night vision camera of a drone should have moderately low latency as the drone (and probably its operator) uses the video stream for navigation. Thus it should send out every frame immediately, except that it may still spend some time compressing the frame to JPEG to achieve a balance between latency and bandwidth. Pushing for extremely low latency of the camera does not help much because the whole system is limited by the delay of the radio communication and the human in the loop.

Night vision goggles or helmets are stringent on latency to the extent which no ordinary digital system satisfies, thus expensive analog devices have to be used.

Here we see how non-functional requirements – namely, latency, bandwidth and cost – impact all the stuff down to the hardware. The same happens with multiplayer games: while a chess client is a simple web page, a fighting tournament or a first-person shooter is very likely to need a client-installed application that processes much of the game logic locally while relying on a highly customized network protocol to decrease the latency.

Another example is the choice of programming language: you can quickly write your system in Java or Python sacrificing its performance or you can spend much more time with

C or C++ and manual optimization to achieve top performance at the cost of the development speed.

Conflicting forces

We see that forces influence architecture. That becomes way more interesting when a system is shaped by conflicting forces – the ones that while opposing each other still need to be met by the architecture.

Remember how old Windows used to freeze on formatting a floppy or when it encountered one with a bad cluster? Let's see how such things could have happened (though the real cause was a bit different it also came from the modules' sharing a context).

User clicks the
"Format" button

The system UI is frozen	System UI
The window is frozen	Format floppy window
A long-running task	Floppy driver
Hardware	

Time

Formatting individual
sectors of the floppy
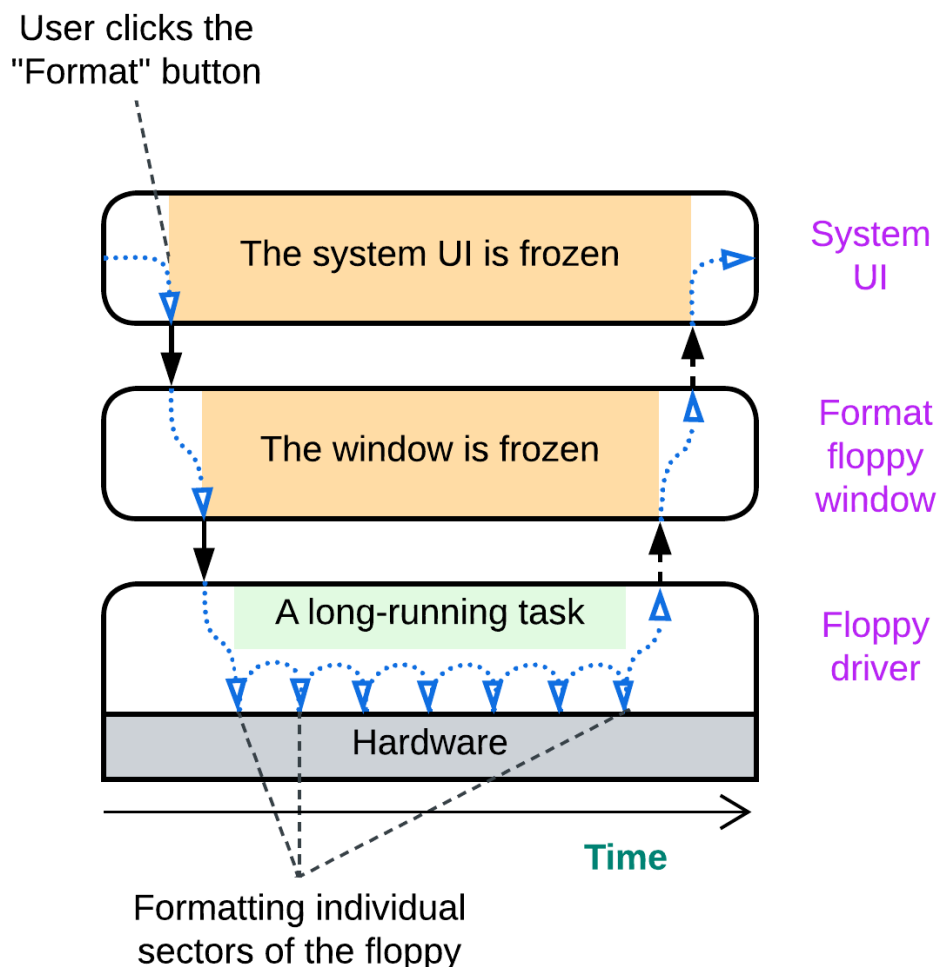
The system implements the function it was made for – it formats floppies. However, while the low-level module is busy interacting with the hardware, all the modules above it have no chance to run as they have called into the driver and are waiting for it to return. The modules are there, with the code separated into bounded contexts (the UI does not need to care about sectors and FATs) but all of them share non-functional properties – latency in this case. Either the UI is responsive or the floppy driver runs a long-running action. We need the UI and the driver to execute independently.

Asynchronous communication

If the modules cannot communicate directly (call each other and wait for the results returned) how should they interact? Through an intermediary where one of the modules leaves a message for another. Such an intermediary may be a message queue, a pub/sub channel or even a data record in a shared memory. The sender posts its message and continues its routine tasks. The receiver checks for incoming messages whenever it has a free time slot. Behold multithreading in action!
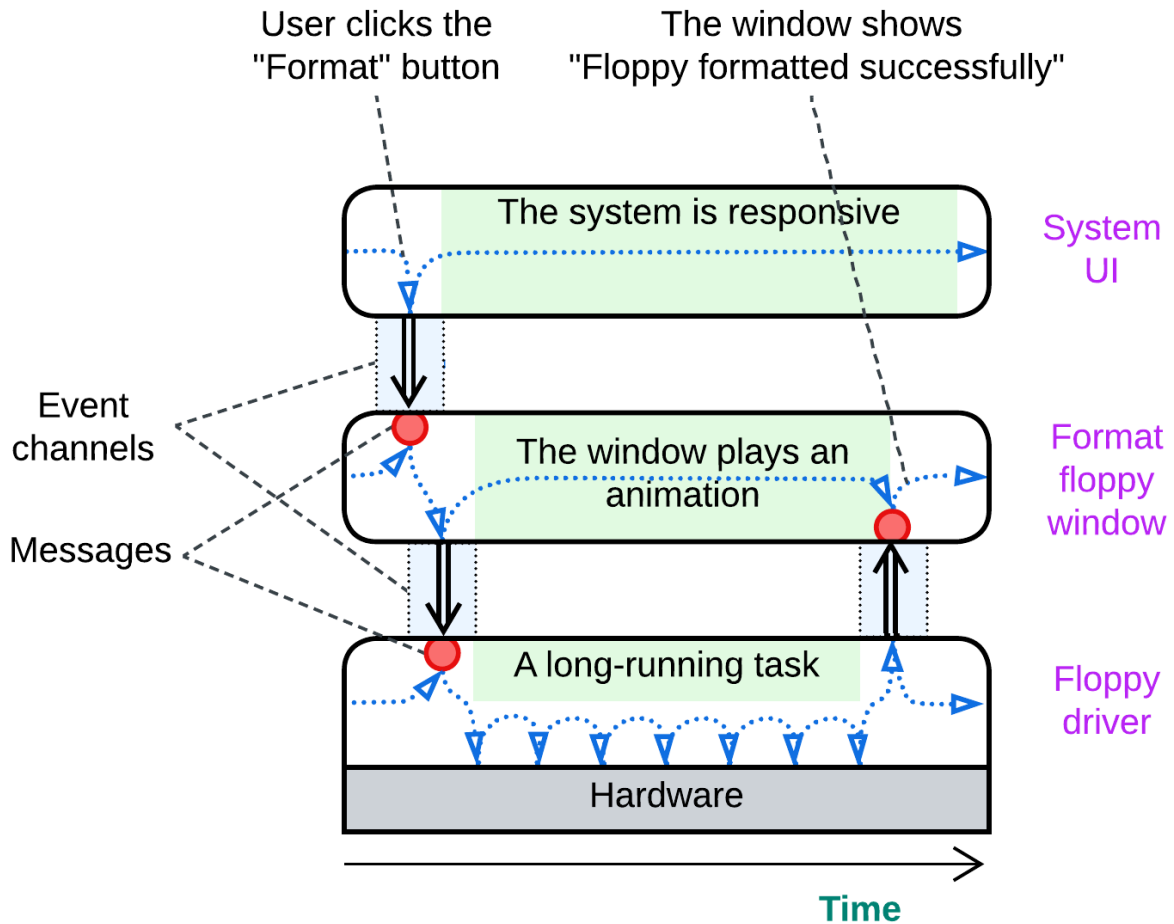
Distribution

Once modules run independently, we can separate them into processes and even distribute the processes over multiple computers. That is required to address fault tolerance and high availability and solve conflicts around scaling or locality.

Consider a web site. Most of them follow the 3-tier architecture:
- Frontend that runs in users' browsers.
- Backend that runs on the business owner's servers.
- Database that usually runs on a single powerful server.

This pervasive division makes quite a lot of sense.

Frontend (stacked)
Scalability: very high
Security: none
Cost: very low

Backend (stacked)
Scalability: high
Security: high
Cost: moderate

Database
Scalability: poor
Security: very high
Cost: high
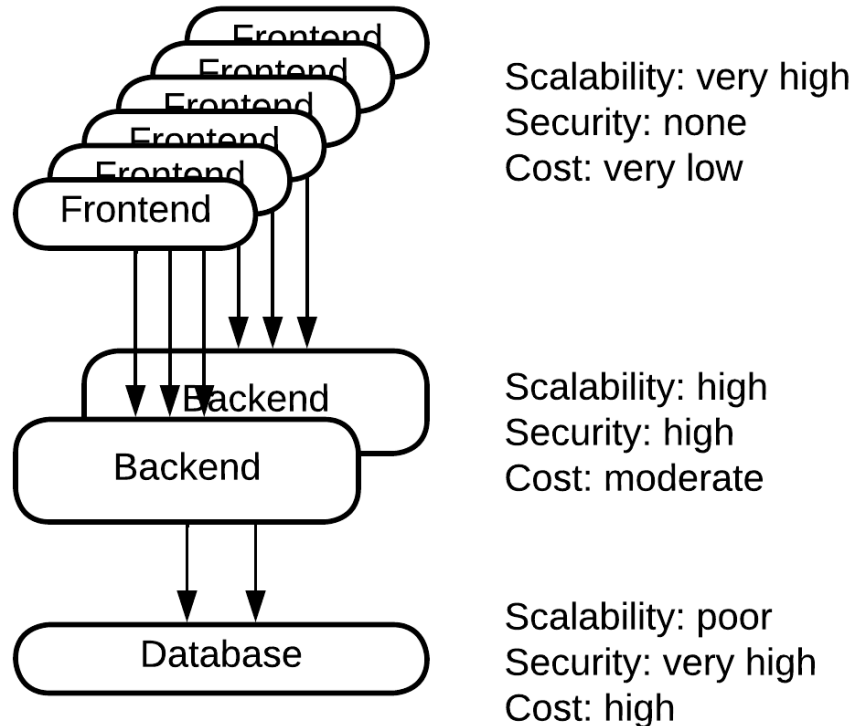
Websites are accessed by many users simultaneously. Any business owner wants to pay less for his servers, thus as much work as possible is offloaded to the users' web browsers which provide unlimited resources for free (from the business owners' viewpoint). Here we have a nearly perfect scalability – the business owners pay only for the traffic.

Other parts of the software are business-critical and should be protected from hacking. Such are kept on private servers or in a cloud. This means that the business owners pay for the servers while they may scale their application by flooding it with money.

The deepest layer – the database – is nontrivial to scale. Distributed databases are expensive and consume a lot of traffic. And they scale only to an extent. It often makes more sense to buy or rent top-tier hardware for a single database server than to switch over to a distributed database.

This is a good example of how the physical distribution of the system solves the scalability, security and cost conflict by choosing the best possible combination of the forces for each module. Whatever is not secure scales for free. Whatever does not scale gets expensive goods. Whatever remains is in between.

Another example comes from IoT – a fire alarm system. They tend to use 3 tiers as well:
● Sensors (smoke or fire detectors) and actuators (fire suppression, sirens, etc.).
● A field gateway – a kind of a router the sensors and actuators are connected to.
● A control panel – some place where operators drink their coffee.

Sensors and actuators are cheap and energy-efficient but dumb devices. They do not react to events unless explicitly commanded. The control panel is where all the magic happens, but it may be unreachable if the network is damaged or the wireless communication is jammed. Field gateways stand in between: they collect information from the sensors, aggregate it to save on traffic, communicate with the control panel and they can even activate the actuators if the control panel is unreachable. In this case a part of the business logic is installed in the dedicated devices which are located within the controlled building.

Here reliability conflicts with accuracy: a human operator makes an accurate estimate of the threat and chooses an appropriate action, but it is not granted that we can always reach the operator. Thus to be reliable we add an inaccurate but trustworthy fallback reaction.

Fire alarm system   *Normal operation*   *Autonomous operation*
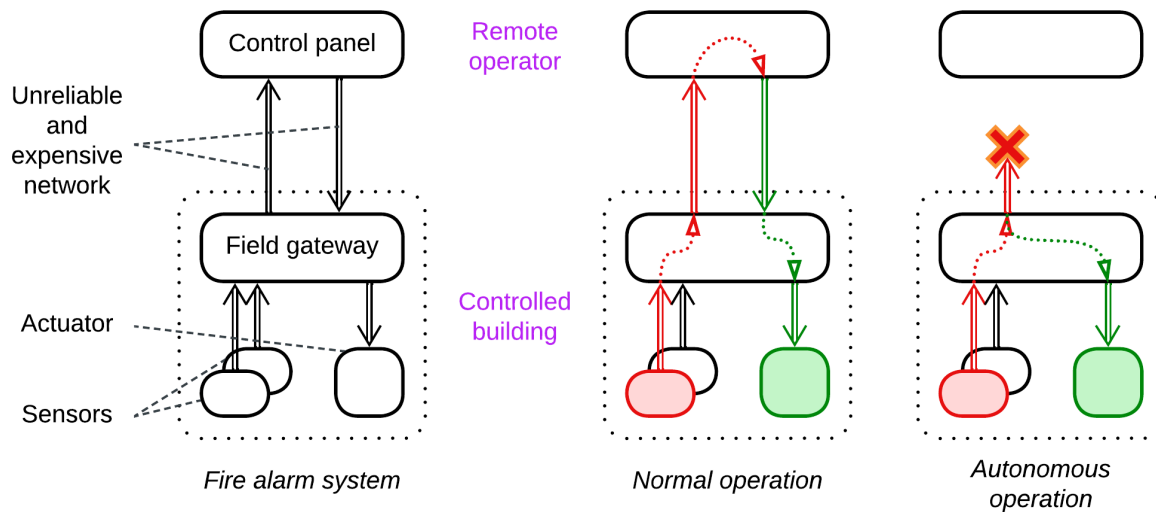
A similar pattern may be found with robotics, drones or even computer hardware (e.g. a HDD): dedicated peripheral controllers supervise their managed devices in real time while a more powerful but less interactive central processor drives the system as a whole.

The goods and the price

Let's recollect what we found out.

Modules make it easier to reason about the system, enable development by multiple teams in parallel and resolve some conflicts between forces. For example, development speed against performance or release frequency against stability are solved by choosing a programming language and release management style on a per module basis.

The cost is the loss of some options for performance optimization between modules and the extra cognitive load while debugging a module you are unfamiliar with.

Asynchronous communication is a step forward from modules that solves more conflicts of forces. It addresses latency and multitasking.

We pay for that with context switches and the need to copy and serialize data which is transferred in messages, which makes communication between the participating modules slower. Debugging asynchronous communication becomes non-trivial as one cannot single-step from the message sender into the message handler.

Distribution builds on asynchronous communication (as networks are asynchronous) and decouples the participant modules in such forces as scalability, security or locality. It separates release cycles of the modules involved and makes it possible for the system to recover from failures of some of its components.

The price? Even slower communication between the now distributed modules (networks are quite slow and unreliable) and extremely inconvenient debugging as you need to connect to multiple components over the network.

We see that the more isolated our modules become, the more forces are decoupled and the more flexible is the resulting system. But the very same decoupling devastates performance and makes debugging into a nightmare.

Any moral? There is one, even a few.

1. Do not overisolate. Go asynchronous or distributed only if you are *force*d to. Especially if you are actively evolving your system. Especially in an unfamiliar domain.
2. Cohesive logic goes together. If you split it among asynchronous or distributed components, it may be very hard to debug.
3. Modules that intercommunicate a lot go together. Distributing them may kill performance and even break consistency of the data.

Control and processing software

This chapter is too long and vague. It should be rewritten.

Software systems differ in many aspects and use a variety of styles. Still, not every approach fits every kind of system. There is a distinction which is probably more drastic than that between frontend and backend but is rarely if ever discussed. I mean the one between control and data processing.

That topic is elusive for a reason: software and mixed (hardware + software) systems fill the continuum of qualities between *real-time control* and *one-off data processing*. As we cannot analyze or even imagine all the existing kinds of programs, we'll look into a few diverse (and seemingly random) examples at each end of the spectrum and generalize the observations. Hopefully, that would improve our understanding of everything in between.

Control and interactive systems

Some systems exist to control the physical or digital reality: an autopilot makes sure that the aircraft is stable and on its course, gateways integrate distinct parts of the Internet, Tetris and Arkanoid were among the first computer games invented. Let's investigate each case.

Autopilot

The human pilot may adjust the desired state of the airplane which the autopilot must follow

The autopilot compares the current and desired states of the plane and adjusts actuators to compensate for the observed difference

Pilot

The autopilot updates the model of the current state of the airplane based on inputs from sensors

Target state

Calculate difference

Observed state

Autopilot

Speed

Compass

Inertial

Sensors

Jet

Wings

Tail

Actuators

The airplane

Figure 1: Autopilot.

A [basic autopilot](#) stabilizes a vehicle it controls. It receives inputs from multiple sensors, aggregates them to update the modeled state ([tilts](#), elevation and speed) of the vehicle, compares the observed and (pilot-defined) target states and sends requests to actuators to compensate for any difference. Rinse and repeat many times per second. Essentially, its job is to keep the observed state of a physical vehicle as close as possible to the values input by pilots.

There is a second, slower feedback loop in the system: the observed state is reported to the pilots (or a guidance system) who may change the target state via their controls, for example to start a climb or descent. The new target state becomes the one the autopilot follows.

Telephony
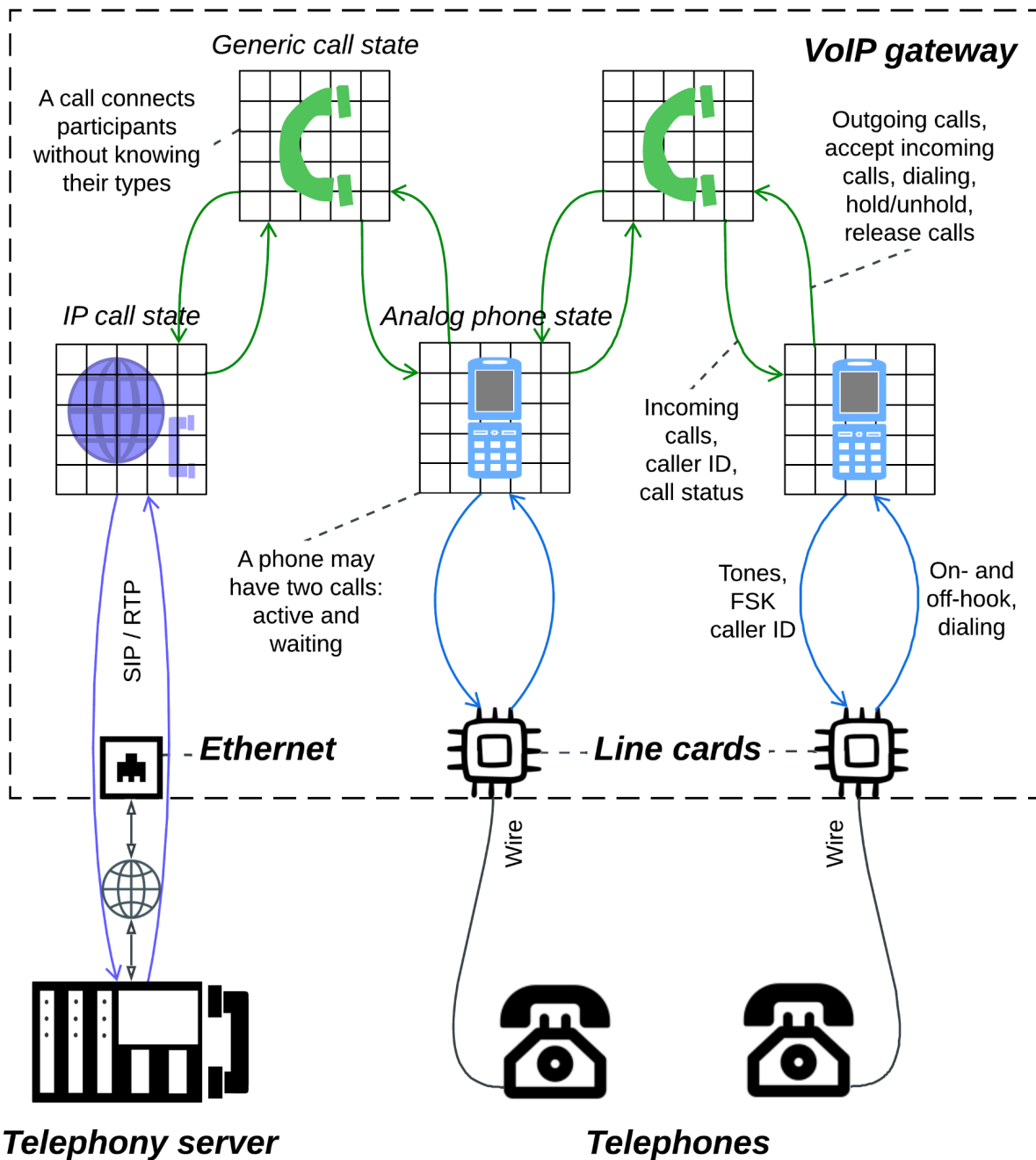
Figure 2: VoIP gateway.

A VoIP gateway connects several analog phones forming a telephony network and integrates the latter with IP telephony servers. It maintains multiple states: a model of phone hardware per connected phone, a model or two per call in the system and, probably, a model per session with an IP server. The models of the phone port hardware and of the server connections provide feedback loops to the parties they control while each model of a call connects two (or three for a conference call) lower-level models. Some of the models are created and destroyed as calls come and go, and there are no explicit target and observed states – an event that comes from a user of a local phone spreads a wave of changes through the corresponding analog phone, generic call and its other side (analog phone or IP

call) states to the hardware on the other side of the call. Thus all the states in a telephony system are intermediaries that adapt and connect hardware devices.

When a model receives an event, it chooses a reaction that matches its current state. For example, a model for an analog phone can react to an incoming call in the following ways:

- If the phone is idle (on-hook), a ringing pattern will be generated.
- If the phone is already in a single voice call, a call waiting tone will sound.
- If the phone is in a voice call and also has a waiting call, the incoming call will be rejected as there is no way for the user to discern between two waiting calls.
- If the phone is dialing an outgoing call, the incoming call fails as well.
- Otherwise the phone is in a transient state (connecting or disconnecting a call) and the incoming call should wait for a second and then retry.

In any case the model of analog phone responds to the call model with a call status (ringing, rejected or retry) and many branches initiate interactions with the line card's hardware. As we see, a single kind of event (an incoming call) causes various reactions depending on the current state of the component (phone model), some of them arranging for chains of further events (the incoming call will eventually be accepted or released).

Game

Tetris has a single model involved in two loops

The user feedback loop draws the current game state on the screen, causing the user to press keys

Game state

The timer loop updates the game state, making the user lose and retry
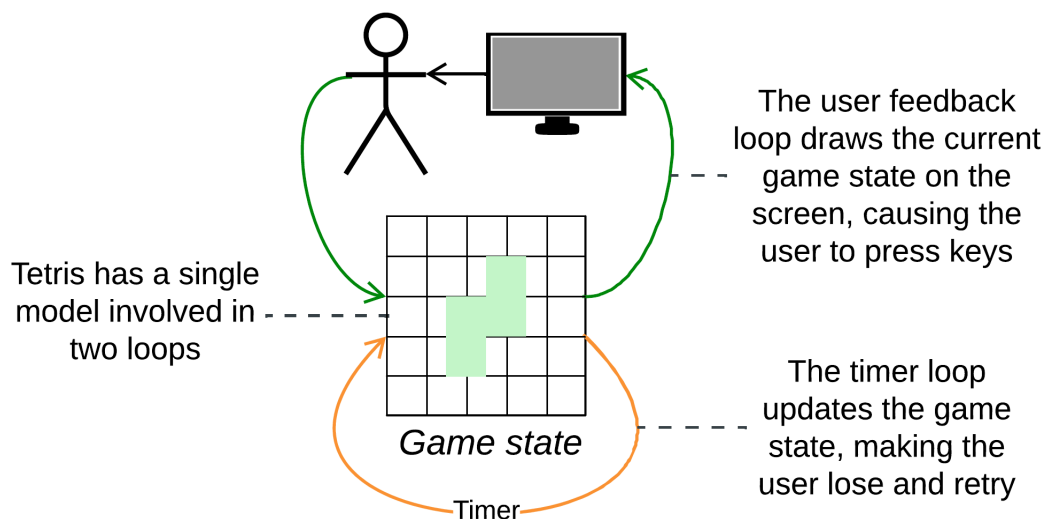
Timer

Figure 3: Tetris.

An action game maintains a model of the game world which changes on timer (physics and AI moves) and on user input (player's actions). The keys the user presses are meaningless without seeing the picture generated on the screen from the game or UI model: the reaction to a keypress is defined by the state of the game or the menu item highlighted on the screen.

Similarities and differences

Let's list shared features of the systems inspected:

- Each case is *real-time*. Latency is among the main driving forces: a delay in the stabilization of a plane may cost lives, a delay in accepting or releasing a call makes the user unhappy, a lagging game may not sell well.

- Each case runs one or more *feedback loops*. There are no predefined scenarios, the systems just *react* to events that change from run to run.
- The loops process *incomplete inputs*: the meaning and handling of each input event is defined by the current state of the system which emerged from the past events and is in some way updated with the current one.
- Each system relies on one or more *models* that store and control the system's state and constitute the core of its feedback loops and the system itself.

Other details differ:
- The number, roles and lifetimes of models and associated feedback loops.
- The number of parallel scenarios a system is involved in (multiple calls in telephony).
- Latency requirements (very stringent for autopilot).
- Nature of inputs (data from sensors comes in streams while other events are one-off).

Digging into the code

Other subtle details appear in the code. Let's dive into Tetris:

A basic implementation of the game contains 4 states:
- *Playing* the game, where the user controls pieces.
- *Demo*, where the game is played by the computer.
- *Game over*, which shows the high scores table.
- *Menu* that allows the player to start a new game, show help or quit the game.

Each state handles the 4 arrows and an action (enter/space/rotate) button. Each state uses a timer. However, the reaction to the events differs from state to state:
- In the *playing* state left and right arrows try to move the piece in the corresponding direction. If the move fails, the game beeps. The down arrow moves the piece down repeatedly till the move fails, then lands it. The up arrow just beeps as it is not possible to move the piece up. The action button rotates the piece. The handler of the periodic timer can do many things depending on the current state of the game:
  - If there is a piece on the board, it moves down and lands if the move is impossible.
  - If there is no piece, the game checks for full rows, deletes them and adds the number of deleted rows to the current score, which may increase the game speed if the score becomes higher than a threshold.
  - If there are no full rows, a new piece is generated and put to the board. If there is not enough free space for it, switch to the *game over* screen.
- The *demo* state aborts (goes to the *menu*) on any button. The timer handler first calls an AI to move the piece if there is one on the board, then follows the logic of the timer from the *playing* state, except that the game over condition leads to the *menu* state (closes the *demo*) instead of the *game over* state (high scores table).
- The *game over* state leads to the *menu* on any key or on timeout.
- The *menu* state uses keys to select and run items of the menu while on timeout the game enters the *demo* mode.
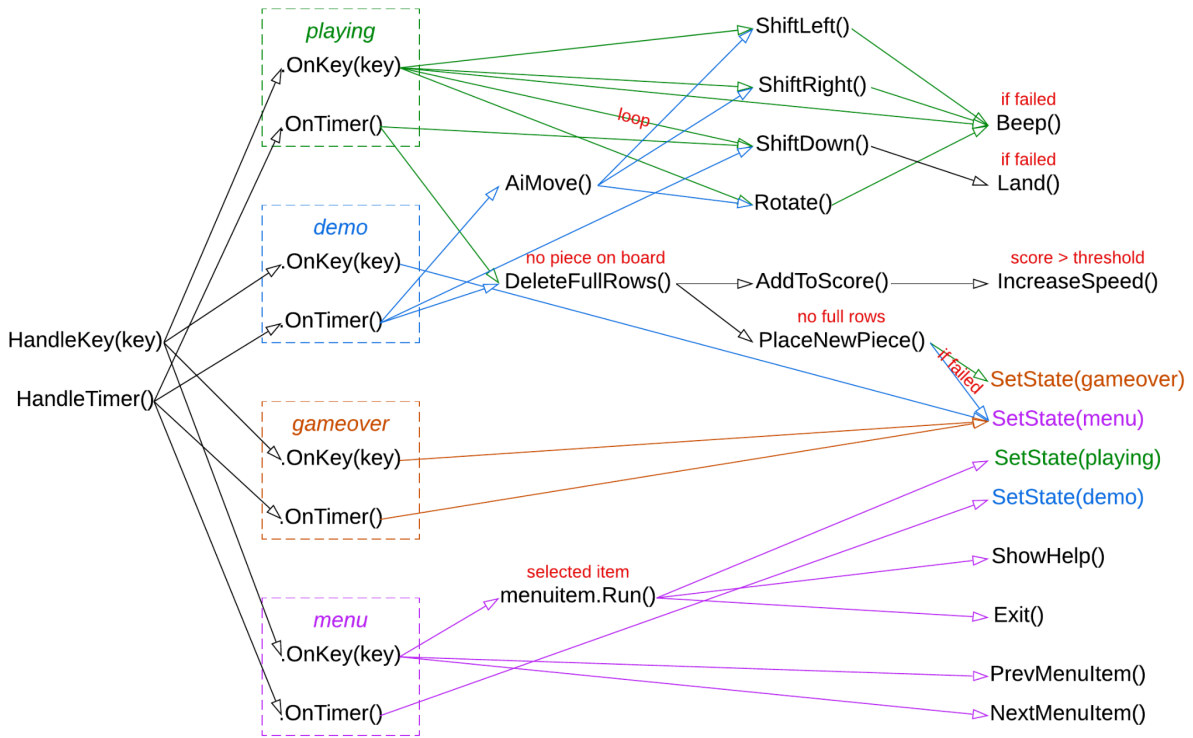
Figure 4: Call graph for Tetris.

The call graph is messy. 5 keys (arrows and spacebar) and a timer cause 10+ lower-level actions, and there is no correlation between an input and its result. Instead, the main decision comes from the current state of the application, as shown in color.

Such a code is probably the only place where the *State* [GoF] / *Objects for States* [POSA4] pattern naturally belongs. It is likely to have much branching via polymorphism (interfaces or dispatch tables) and conditionals (if, switch).

A telephony gateway is suitable for another insight:
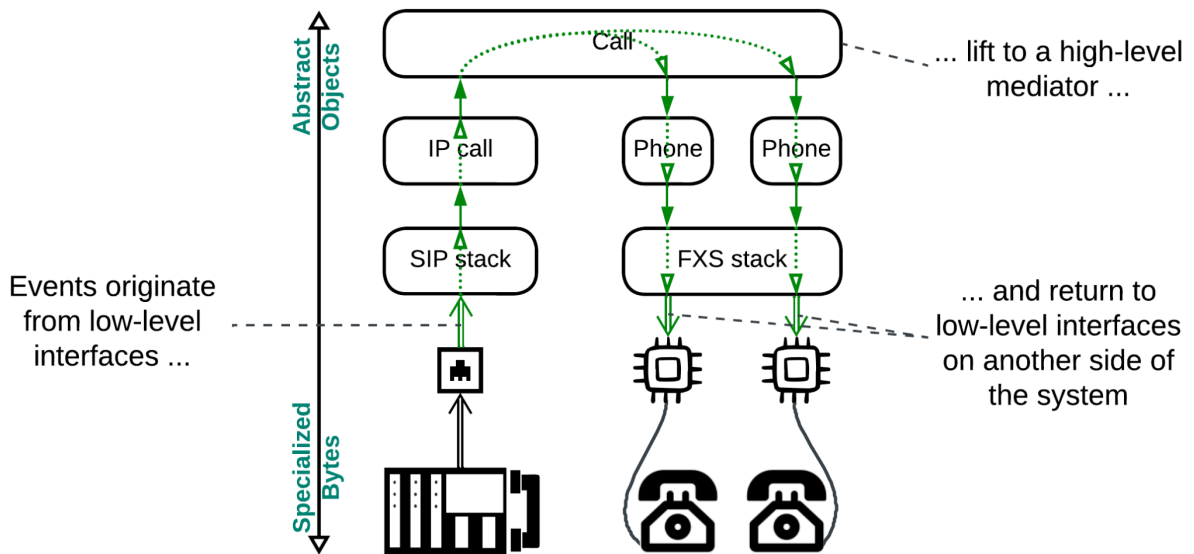
Figure 5: Control flow in a VoIP gateway.

The input comes from a low-level code (system interfaces or drivers), goes all the way up (becoming more abstract) to a high-level module which reacts to it by calling into other lower-level modules that return the transformed event back to the system level. This is a kind of *orchestration* (coordination of subsystems) which is called *Mediator* [GoF]: the call object organizes behavior of the modules it manages by interconnecting them and reinterpreting their messages, but it does not create tasks for them on its own.

In general, it is common to see control and interactive systems built of one or more *proactors* [POSA2] (often called *actors*) – single-threaded non-blocking stateful modules that communicate via events. Each *proactor* may contain one or more *models* synchronized among themselves. *Shared-nothing* (single-threaded logic) and *non-blocking interactions* with the OS and other modules leave few if any points that can delay processing of events (unless the system is overloaded) to assure that the latency is predictably low and the whole system is *real-time*.

Data processing systems

Data processing systems belong to the opposite side of the spectrum. A network video recorder or a digital video recorder receives, analyzes, compresses and stores video streams from multiple surveillance cameras. A single scientific calculation takes a week. A data analyst runs several hours-long queries and compares the results obtained to build a marketing strategy.

Video recorder
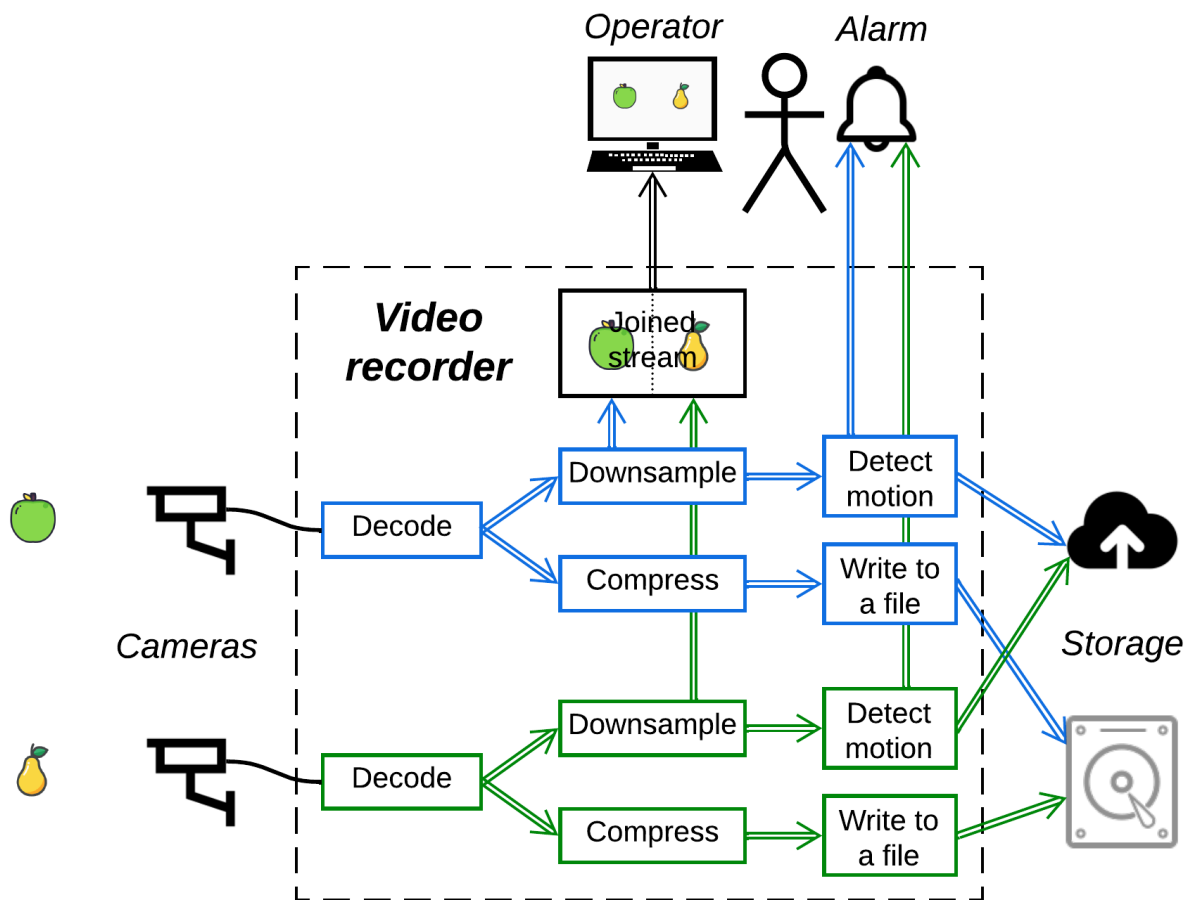
Figure 6: Video surveillance.

A surveillance video recorder receives video streams from multiple cameras, compresses each stream and writes it to a file in a permanent storage, at the same time composing all the streams into one for the security guard. It may also provide motion detection in pre-selected sectors of the camera views and upload pictures of detected intruders to a cloud. As video compression is computationally heavy, the system may employ dedicated (co)processors.

Scientific computation

Chemical computations model electrons through quantum mechanics

Move the nuclei

An iterative calculation looks for a stable geometry and distribution of electrons

H   H
   O

Input atom coordinates

Calculate electron density

Calculate forces for the nuclei

H — O — H

Output results
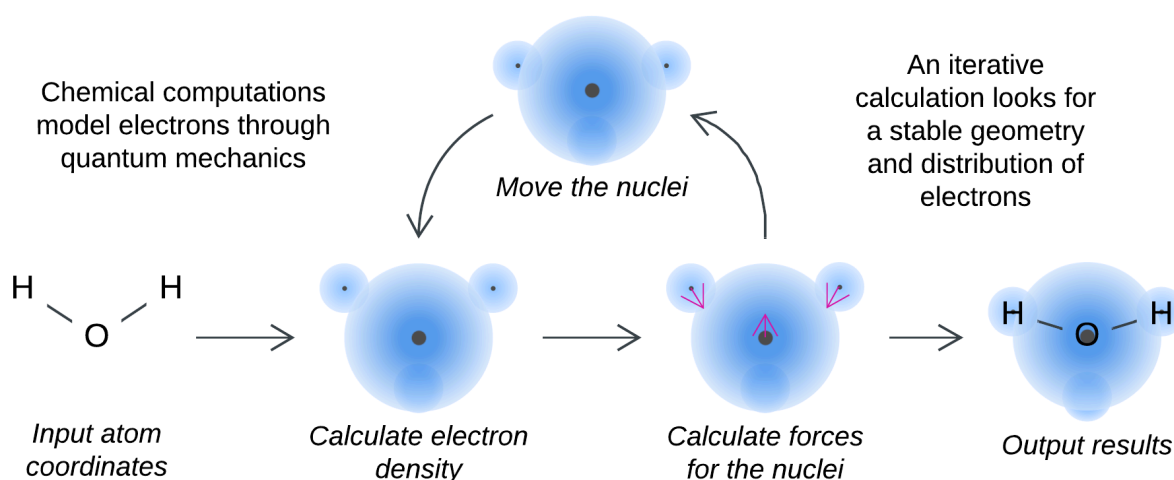
Figure 7: Scientific calculation.

A quantum chemistry computation takes a list of coordinates of atoms in a molecule as an input. It populates atomic orbitals with electrons and repeats the loop of: optimizing the density of electrons over the voxels of the system, calculating forces that act on the nuclei and moving the nuclei in the directions of the forces. As soon as the atom positions stabilize (the forces become small), a final more detailed calculation may be done to better estimate the density of the electron cloud. One iteration is hours-long while the entire process may take a week.

Database query
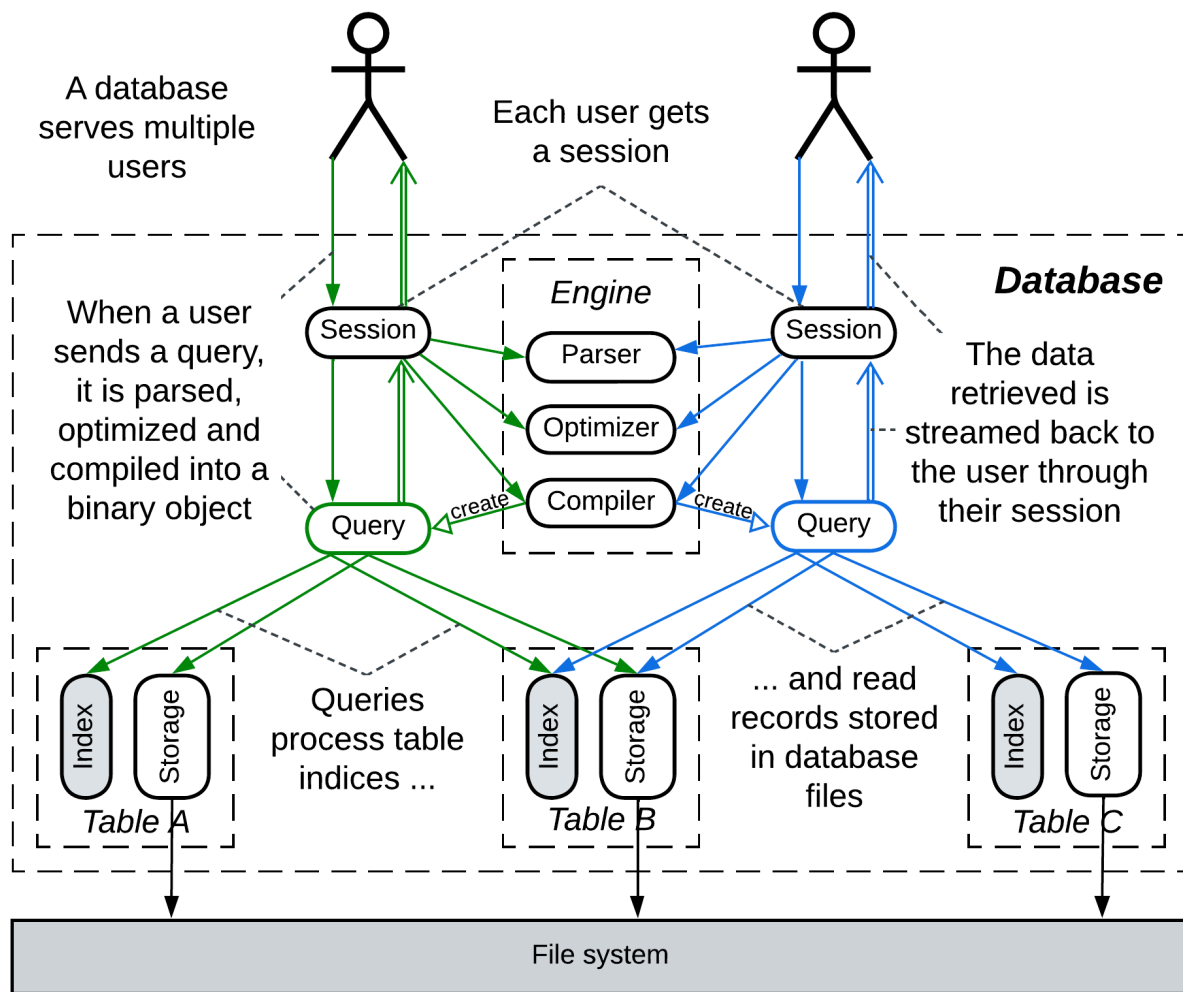
Figure 8: Database.

An analyst uses SQL queries that process sales records over the past couple of years to get insights into optimizing the business. Several queries may be run simultaneously if the database server has enough resources.

Each query goes through a pipeline of multiple preparation steps (e.g. parse, optimize, compile) that create a binary object which encapsulates an algorithm for execution of the user's query. When the query object is run, another pipeline comes to motion: indices of several database tables are joined, the filtered indices are streamed to tables' storages to read in the queried record fields from the file system, the streams of records flow to the query object to be merged and streamed to the client.

Common features

We can deduce the following shared tropes:
- Data processing is expected to be slow and resource-intensive. If we can purchase a more powerful system, we are likely to use the extra power to increase *throughput* (connect more cameras, calculate larger molecules, analyze sales data over the past decade) instead of just running the current tasks faster. Time is a secondary constraint.

- Each process is a pre-defined sequence of known steps – the ideal subject for *imperative* or *functional* programming.
- The *input defines the task*: the video frame bears the information to be compressed, the coordinates of the atoms in the molecule describe the substance of the calculation, the SQL query is the task for the database.
- The *state is temporary*, limited to the duration of query processing. Though codecs may need to cache a few video frames for efficient compression, that is not strictly necessary. A quantum calculation that uses enormous amounts of RAM produces a couple of small files. The usefulness of a SQL query is inverse of the length of its output.

What's in the code

A closer look at the code is likely to reveal *nested foreach* loops: the video compression needs to process every pixel in each block of the frame; the chemical computation iterates over voxels, atoms and orbitals; the SQL join merges table indices and reads fields of each record in the resulting dataset. As all the elements are processed uniformly and the size of the dataset is known beforehand, the calculations can often be parallelized over CPU cores, uploaded to a SIMD (TPU / GPU / AVX) or DSP processor or even to a distributed map/reduce framework.

The code is more or less linear, with *static branching* (the control flow does not change for the duration of the task) implemented through *Strategy* [GoF] or inheritance. *Pipes and Filters* [POSA1, POSA4] (aka *Pipeline*) pattern is almost exclusive to data processing as it is an abstraction for streaming.

High-level *orchestration* emerges as the *Facade* [GoF] pattern: the database sessions on the diagram above or an engine of a software for chemical calculations receives a task from a user, interprets it, calls into multiple components of the system it manages and returns the results to the user. The system components are passive and the whole process is driven by the *facade*.

Data processing applications rarely care about latency: *blocking* calls and *sharing data* between threads are abundant. It is common to have a *thread per task*, with some of the tasks coming from the system's clients while others being internal (maintenance) activities. Request processing usually builds around *Reactor* [POSA2] (blocking threads) or *Half-Sync/Half-Async* [POSA2] (coroutines).

The middle ground

Hitherto, we have looked into both ends of the control / processing dichotomy. How are the opposites mixed in the middle? To answer that we should find out why their specific aspects emerge.

The origin of the distinction

First and foremost, let's summarize the differences between control and processing systems:

	Control	Processing

Programming paradigm	Event-driven / reactive	Procedural / functional
The code is	Non-blocking	Blocking
Code has many	Branches	Foreach loops
Performance means	Latency	Throughput
The system runs	An infinite loop	A batch of tasks
Decisions rely on	State (models)	Input (task)
State is	Permanent (model)	Temporary (task's variables)
Inputs are	Updates for the model	Complete tasks
Synchronized unit is	Model	Task
Thread synchronization	Messages (shared nothing)	Mutexes (shared data)
A prominent pattern is	State	Pipeline
The central orchestrator is	Mediator	Facade

We can see that many aspects of control systems are complicated: event-driven and non-blocking code is hard to read, branches and state machines obscure the logic, keeping the system's state up-to-date and self-consistent is non-trivial. They have more than one root.

Some of the properties are requisite for *controlling* a system. Those include the main loop, state-based decisions with much branching and the central role of a *mediator*.

Others support *real-time* nature: messaging and non-blocking calls assure that the latency is predictable. The use of models to cache the last known state of components is often an optimization that improves latency (by removing redundant communication) and consistency (local decision-making does not care about CAP).

In contrast, the peculiarities of processing systems mostly simplify the code but fit only processing of a *large volume of uniform data*: a foreach loop treats all the elements of a container uniformly, temporary state means that tasks are unrelated, while pipelining relies on both preconditions.

Let's mix!

We have identified the following qualities that shape the software:
● Control nature – the system runs an eternal feedback loop.
● Real-time nature – the system must predictably quickly react to events.
● Data processing nature – the system operates a large volume of uniform data.
Let's make new combinations.

Nothing special

Most basic applications are neither interactive nor process a lot of data. A command line utility is likely to feature some branching, some loops, blocking calls and no multithreading.

Control but not real-time

In some cases tasks are slow and non-cancelable while the controlled (sub)system can execute one task at a time. In that case the code may employ branching, states and run a main loop, but is likely to block on calls to the hardware. Any incoming tasks wait in a queue

for the hardware to complete whatever it is busy with at the moment. Examples include writing to hard drives or flash, communication buses or even scheduling physical transportation (elevators).

Real-time but not control

Highly loaded services may apply optimizations similar to those of real-time systems. Non-blocking calls reduce the number of threads in the system and minimize the penalty for switching between tasks. *CQRS views* let a service cache subsets of data from other services to avoid calling them [MP].

Real-time data processing

A few cases, like computer vision and rendering in games, require real-time processing of data streams. They are likely to employ both pipelines and non-blocking calls, often with multithreading (a thread per CPU core) managed through custom schedulers and/or thread pools.

Composition

As we discussed in the previous chapter, conflicting forces are resolved by using modules. Modules may differ in coding practices and conventions, and – if they communicate asynchronously – in latency and throughput as well. Therefore a system may embrace both interactive and data processing components to get the benefits of each quality without reverting to complicated (if possible) techniques for achieving both in the same piece of code.

Online store as an example

Consider an online store:

The OLTP stack is mixed: the latency is important to an extent

The database needs to support both quick and long-running requests

All the frontends are interactive

The OLAP backend may block on a request to the database
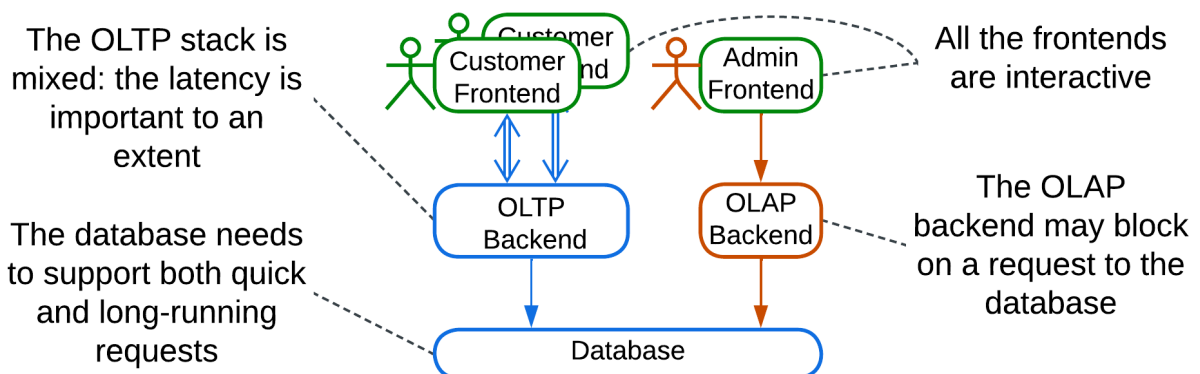
Figure 9: Online store.

The frontends are soft real-time interactive applications with main loops, event handlers and non-blocking interaction. The backends are likely to be of a mixed kind that utilizes much branching and blocking calls. They differ among themselves as the admin backend for analytics is supposed to be slow because of its complex database queries and its logic may be quite linear, while the backend for customers needs to respond quickly and check for many errors and corner cases. The database is the most complicated component as it must

support both quick and long-running queries simultaneously. The system is composed of distributed modules of every possible style and they flawlessly fit together, yielding their benefits to the users.

Revisiting a database

The database (figure 8) tries to follow the same approach. See those Query objects? They are separate modules that vary in latency. Each Query runs its pipeline on an isolated snapshot of the database [DDIA] to assure that no write may ever change the queried data. The isolation of both query state and the data the query operates on allows for the code of queries (which constitutes much of the codebase) to be written in a relatively straightforward way while whatever is around it (the data structures for snapshots, synchronization with the file system, scheduler) needs much attention to avoid race conditions and deadlocks while juggling tasks of wide range of latencies. Applicable hackarounds include fine-grained locking (or its extreme case – atomic data structures) and fibers / continuations / coroutines which are hard to write correctly and even harder to comprehend in a large codebase.

And in the silicon

The internals of software execution reveal the same pattern of separate components for control and processing: CPUs entail pipelines for instructions and state machines for cache coherence; compilers are pipelined but many of the steps involve large dispatch tables or visitors [GoF]; OS threads are stateful but are run on mostly stateless CPU cores.

Summary

We looked into several examples of real-time control and data processing systems, identified their common features and discussed ways to combine them: either within a single module or as separate components of an asynchronous system.

Arranging communication

As a project grows, it tends to become subdivided – into services, modules or whatever you call the components – by subdomains (or *bounded contexts*, if you prefer the [DDD] convention). However, there remain system-wide use cases which require collaboration from many or all of the modules – otherwise the components don't constitute a system. Let's consider the ways to coordinate them.
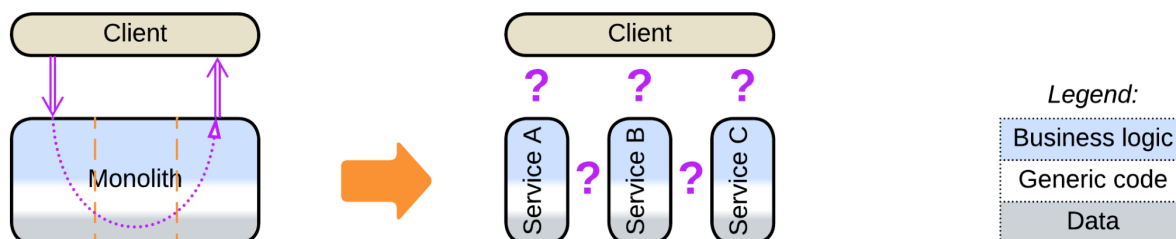
Orchestration

The straightforward way is through adding a coordinating layer on top of the services:

The orchestator implements use cases by calling into the services

The services don't need to communicate directly
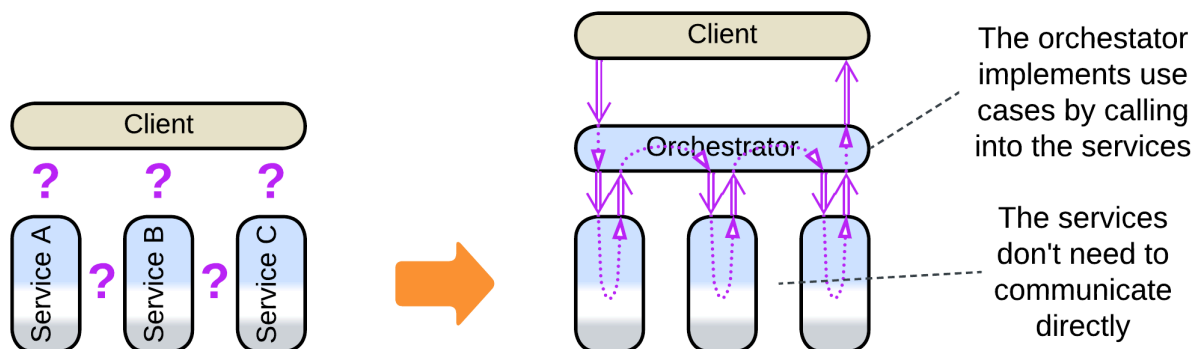
The good thing is that your *orchestrator* has explicit code for each use case it covers, and every running scenario will have an associated thread, coroutine or object, so that you should be able to attach to the orchestrator and debug any use case step by step. Neither you have to worry about keeping the state of the services consistent as they are passive and all the changes come from the orchestrator. It is also the default approach for desktop applications, where it is faster to call into an orchestrated module and return than to send it a message.

The bad thing about orchestration in distributed systems is that it doubles the communication overhead: two messages per service are involved as direct calls between processes are not possible.

Roles

In a backend that serves client requests an orchestrator takes the role of *facade* [GoF] – a module that provides and implements a high-level interface for a multicomponent system. It sends requests to the underlying services and waits for their confirmations – the mode of action that can be wrapped with RPC. The state of each scenario that the facade runs resides in the associated thread's or coroutine's call stack (for *Reactor* [POSA2] or

Half-Sync/Half-Async [POSA2] implementations, correspondingly) or in a dedicated object (for *Proactor* [POSA2]).

The facade receives
a high-level request
from a client ...

The interactions rely
on request/confirm
pairs characteristic
for RPC

... and executes it
through a series
of requests to the
services it
manages

The facade owns the
state of each task it
is processing

Client

Facade

Service A   Service B   Service C
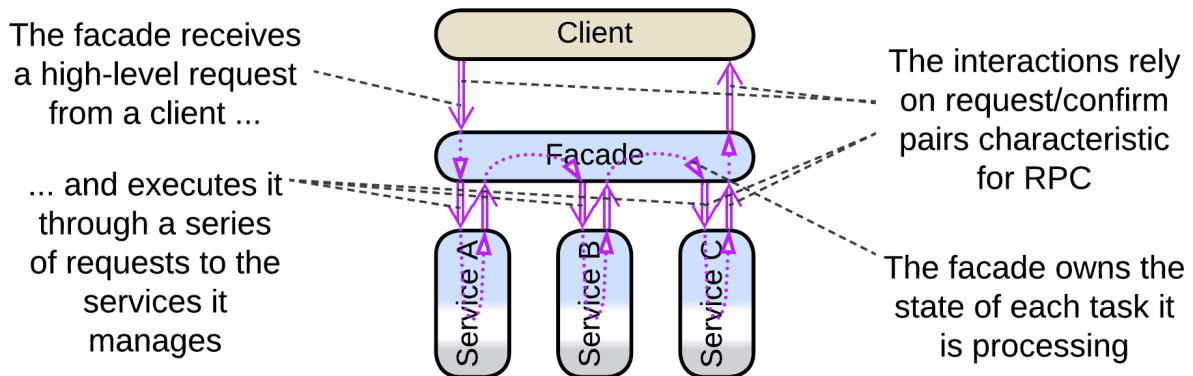
Facade also supports querying the services in parallel and collecting the data returned into a single message – the *Splitter* and *Aggregator* patterns of [EIP]. That reduces latency (and resource consumption as the whole task is completed faster) for scatter or gather requests when compared to sequential execution.

Facade can start
the subrequests
simultaneously ...

... and
aggregate
the results

Client

Facade

Service A   Service B   Service C

This
reduces
latency
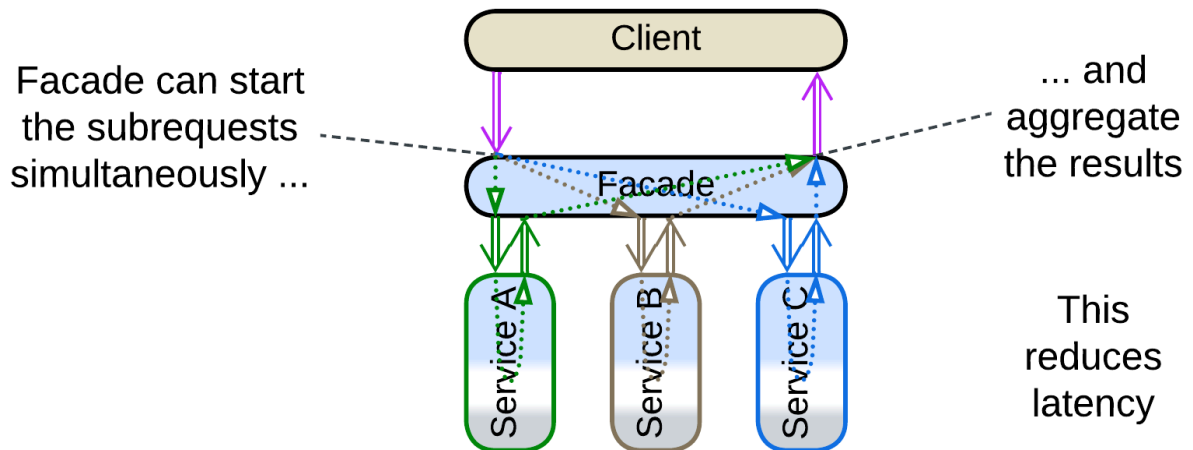
Embedded and system programming – the areas that deal with automating control of hardware or distributed software – employ orchestrators as *mediators* [GoF] – components that keep the state of the whole system (and, by implication, any hardware it may manage) consistent by enacting a system-wide reaction to any drastic change in a single component. A mediator operates in non-blocking fire-and-forget mode which is more characteristic of choreography, to be discussed in the next chapter. This also means that you will not be able to debug a use case as a thread – because there are no predefined scenarios in control software!

A mediator interconnectrs components of a control system

An event from one module cases reactions throughout the system
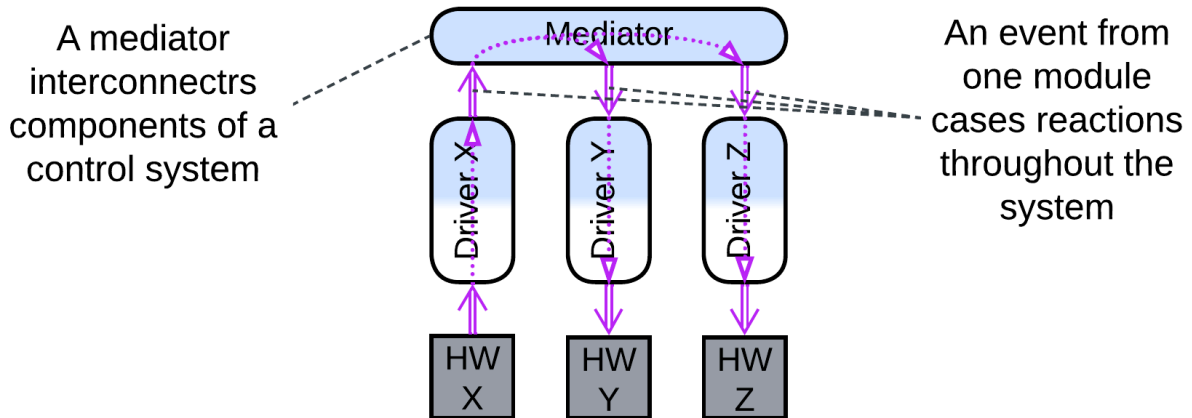
Such a difference may be rooted in the direction of the control and information flow: in backend it comes as a high-level command while control systems react to low-level events.

Dependencies

By default an orchestrator depends on each service it manages – this means that a change in a service's interface or contract – caused by fixing a bug, adding a feature or optimizing performance – requires corresponding changes in the orchestrator. That is acceptable as the orchestrator's client-facing high-level logic tends to evolve much faster than the business rules of the lower layer of *domain services* [DDD], so that the team behind the orchestrator, not restricted by other modules depending on it, will likely release way more often than any other team. However, as the number of the domain services and the lengths of their APIs increases, so does the amount of information that the orchestrator's team must remember and the inflow of changes they must integrate in the code. For a large project the amount of work to support the orchestration layer may paralyze the development – that was a major reason behind the decline of *enterprise SOA* [FSA] where *ESB* used to orchestrate all the interactions in the system, including those between domain services and utility layer components.

A typical orchestrator depends on the API of each service it integrates

The orchestrator itself is free to change as nothing depends on it
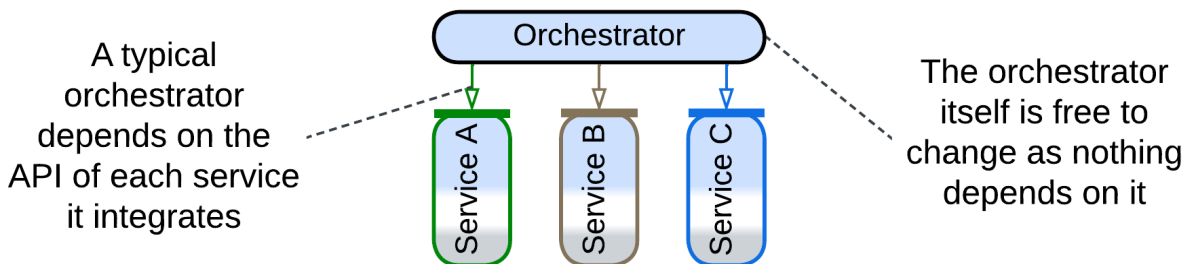
Another option, which appears in *Plugins* and develops in *Microkernel* and *Hexagonal Architecture*, stems from dependency inversion: the orchestrator defines an SPI for every service. That makes each service depend on the orchestrator so that the single orchestrator's team does not need to follow the updates of the multiple services' APIs – it initiates the changes at its own pace instead. However, with that approach the design of an SPI requires coordination from the teams on both sides of it and the once settled interface is hard to change. The most famous example of modules that implement SPIs are OS drivers.

With Microkernel each service depends on the SPI it implements

Microkernel

Driver A Driver B Driver C

Both the orchestrator and services are bound by the established interfaces
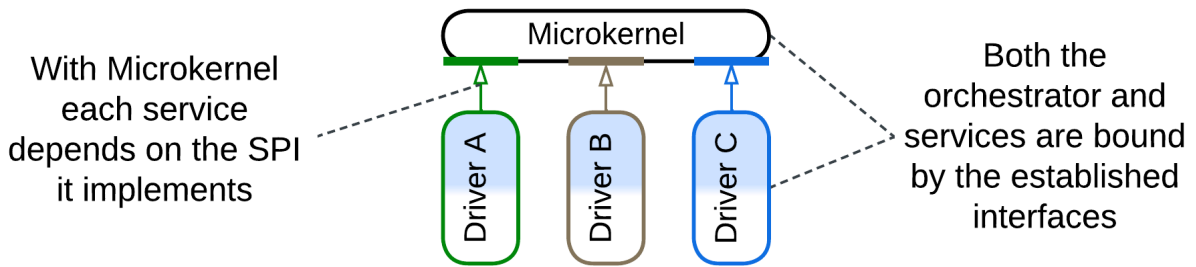
Furthermore, some domains develop the idea into *Hierarchy*: when the services implement related concepts, they may match a single SPI, making the orchestrator simpler (as there is no more need to remember multiple interfaces). That is the case with telecom or payment gateways and it may also be found with the tree of product categories in online marketplaces.

In Hierarchy each child implements its parent's SPI

Parent

Child A Child B Child C

The SPI is very hard to change once established

The child services are easy to add or remove
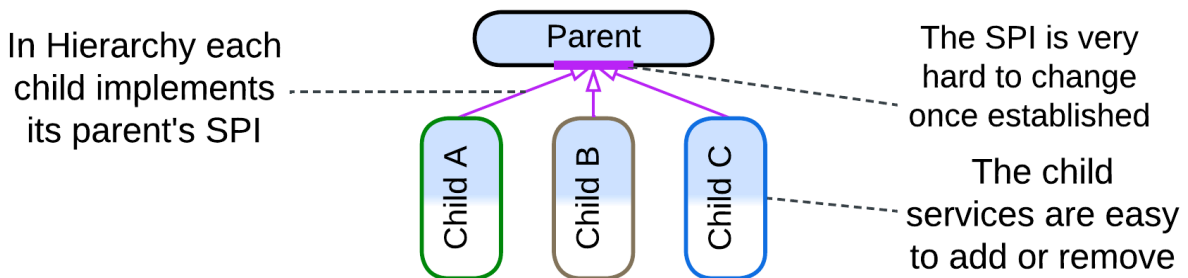
All kinds of orchestration allow for an easy addition of new use cases which may even involve new services as that changes nothing in the existing code. However, removing or restructuring (splitting or merging) already integrated services requires much work within the orchestrator, except for *Hierarchy* where all the services implement the same interface, which means that the code in the orchestrator does not depend (much) on any specific child.

Client

Orchestrator

Service A Service B Service C

Client

Service A Service B Service C Service X

It's easy to add a new use case ...

... even if it needs a new service
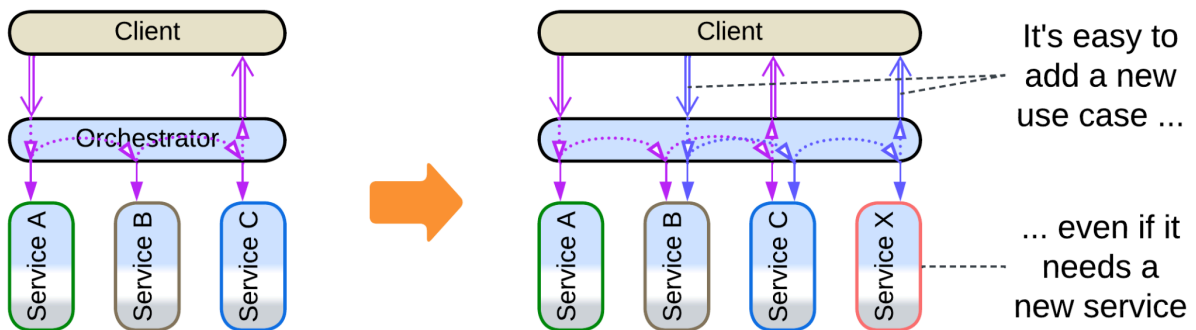
Mutual orchestration

In some systems there are several services that have their own kinds of clients (e.g. employees of different departments). Each of the services tries hard to process its clients' requests on its own but occasionally still needs help from other parts of the system. That creates a paradoxical case where several services orchestrate each other:

Services may orchestrate each other
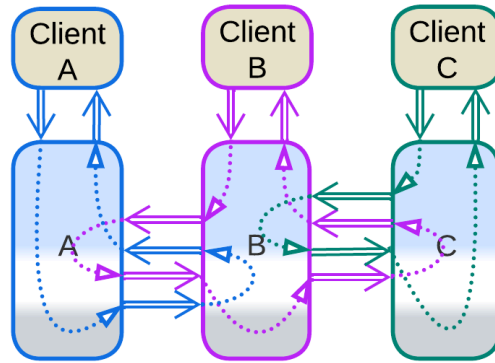
That makes them interdependent

As each of the services depends on APIs of the others, any change to any interface or composition of such a system requires consent and collaboration from all the teams as it impacts the code of all the services.

Services may orchestrate each other
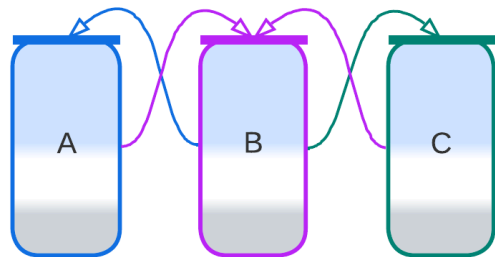
That makes them interdependent

In real life the services are likely layered, with their upper layers acting as both internal and external orchestrators. Layering isolates interdependencies to the relatively small application layer services and resolves to an extent the seemingly counterintuitive case of mutual orchestration as now there is an explicit, though fragmented, orchestration layer.

The application layer calls into both the domain layer of its service and applications of other services
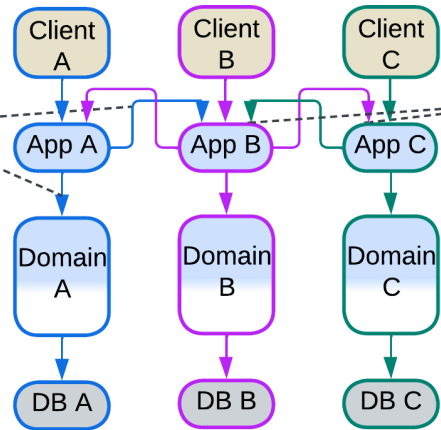
That makes only the applications interdependent

Layering limits the contamination with dependencies to the smaller application part
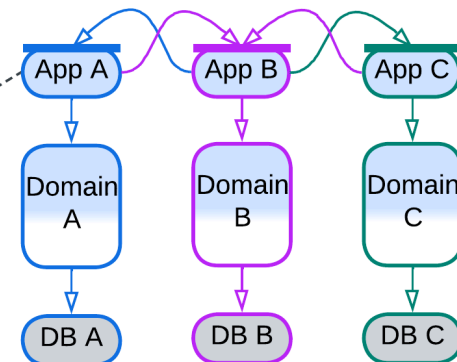
The larger domain services do not depend on each other

Summary

Orchestration represents use cases as a code which allows an orchestrated system to support many complex scenarios. Dealing with errors is as trivial as properly handling exceptions. The approach trades performance for clarity.

Choreography

Another option to integrate services is to build a pipeline which passes every client's request through the chain of modules:

Choreography is about services passing the client's request to each other
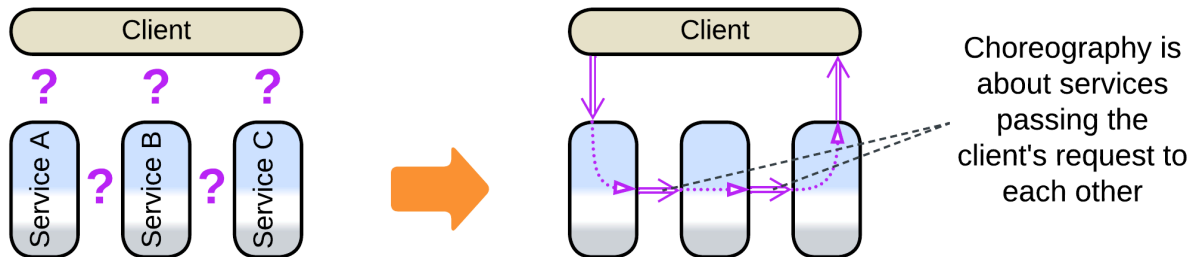
In that case there is no dedicated owner for workflows of the requests – the state of each request consists of its type, data and position in the pipeline. Debugging is mostly limited to reading logs as there is no module to connect to for stepwise request processing. Neither is there a single piece of code to define each of the system-wide use cases – their logic emerges from the structure of event channels between the services and from messages that each involved event handler sends. Consistency of the services' states is left for the services to take care of as there is no overseeing central component.

On the bright side, there is no communication overhead of response messages as there are no responses – the processing cost is single message per service, half of that with an orchestrated architecture. However, a message in a choreographed system is likely to be longer than the corresponding one with an orchestrator as it needs to carry the entire request's state as there is no orchestrator that distributes parts of the payload to the services involved.

A request will likely collect data from each service it passes through

Transfering long messages is bad for performance
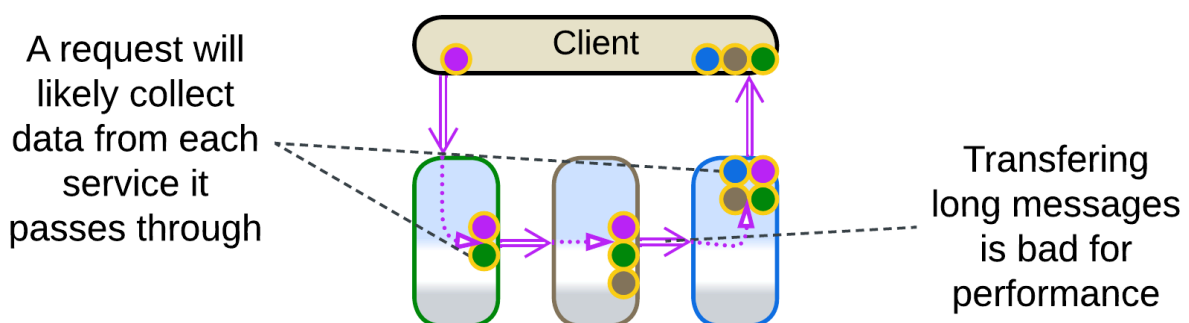
Latency may also be suboptimal as parallelizing execution of a request is easier said than done for there is no place (*Aggregator* [EIP]) to collect multiple related messages, which also means that there is no associated cost in resources (RAM and CPU time) for storing the fragments. Please note that an *aggregator*, when added, starts to orchestrate the system – it stands between the client and services and meddles with the traffic and logic. It spends resources to store the received messages for aggregation, and the messages start forming request/confirm pairs.

An orchestrated system benefits from running subrequests in parallel ...

... which is not an option for a pipeline
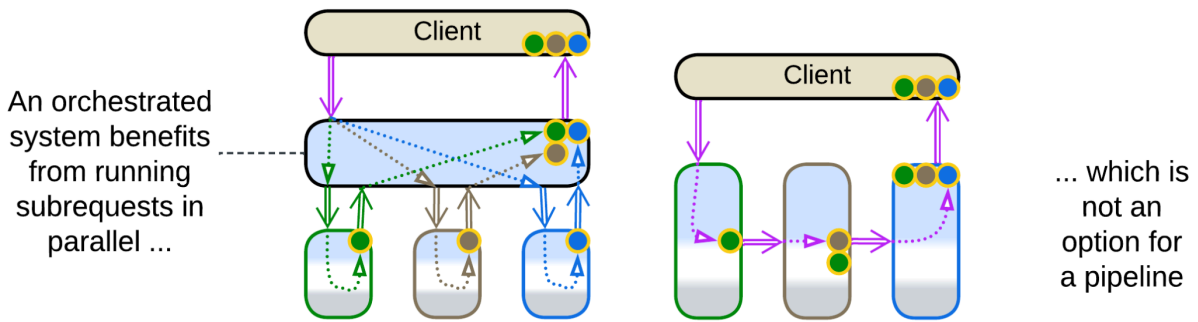
Still another trouble with choreography comes from its weakness in error processing. When a service in the middle of the request processing pipeline encounters an error, it cannot generate its normal output to be sent downstream. One option for it is to fill in a null (or error) value but then each receiver of the message should remember to check for null and know how to deal with the input error. Another way is to add a dedicated error channel for each service to push failed requests into, but that complicates the system's structure. Moreover, a failure in the middle of processing a request may cause the services to end up with inconsistent data if no special attention (a new kind of request to compensate the original one) is paid to roll back the partial change. Please note that all of that is conveniently handled by an orchestrator.

When there is a request that makes changes to several services ...

... and an error occurs ...

... there should be a reverse pipeline to roll back the changes
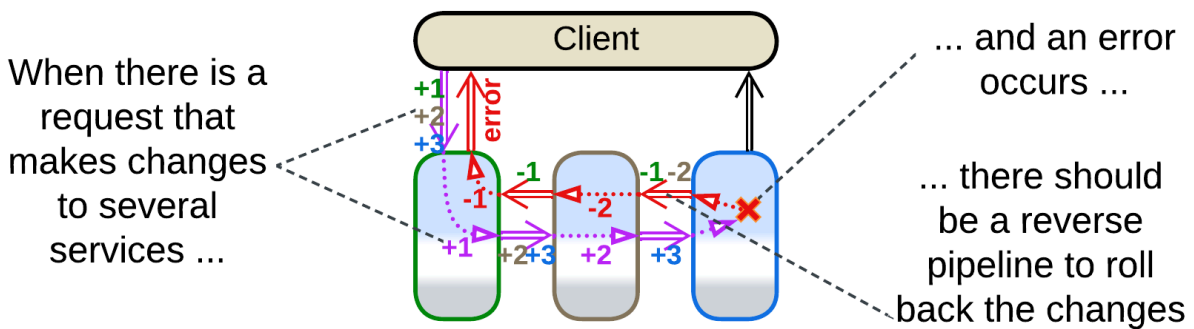
Early response

The ordinary mode of action for a pipeline – sending the final results of processing to the client – requires either for the tail of the pipeline to send data to its head or for existence of a stateful intermediate component – gateway – to receive the client's request, forward it to the head of the pipeline, wait on the pipeline's tail for the processing result and return it to the client. That is necessary because most clients would open a single connection which is impossible to share between multiple modules, namely the (receiving) head and (sending) tail of the pipeline.

It must be a single component that receives a client's request and sends back the response

We may use a dedicated gateway ...

... or loop the pipeline
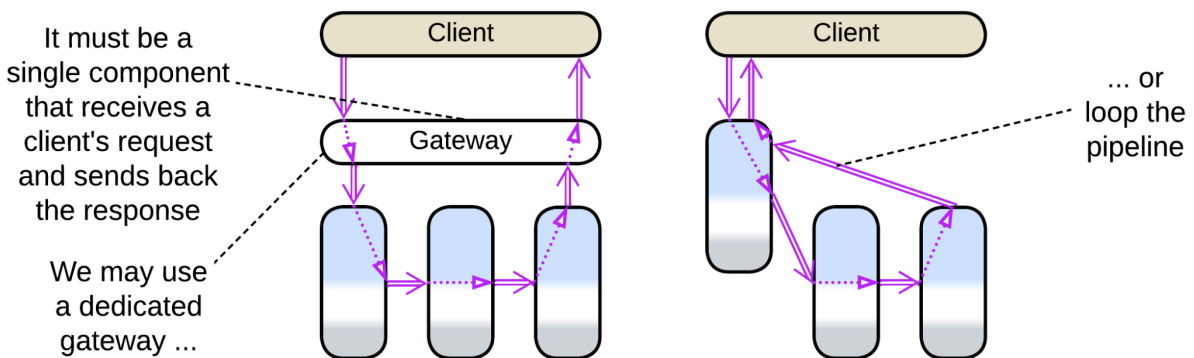
The gateway, if used, may parallelize processing of scatter or gather requests by turning into an *API gateway* which is a kind of orchestrator. Which means that the system changes its paradigm from choreography to orchestration.

The gateway is easily turned into an API gateway, a kind of orchestrator

That improves latency through running parts of the task in parallel
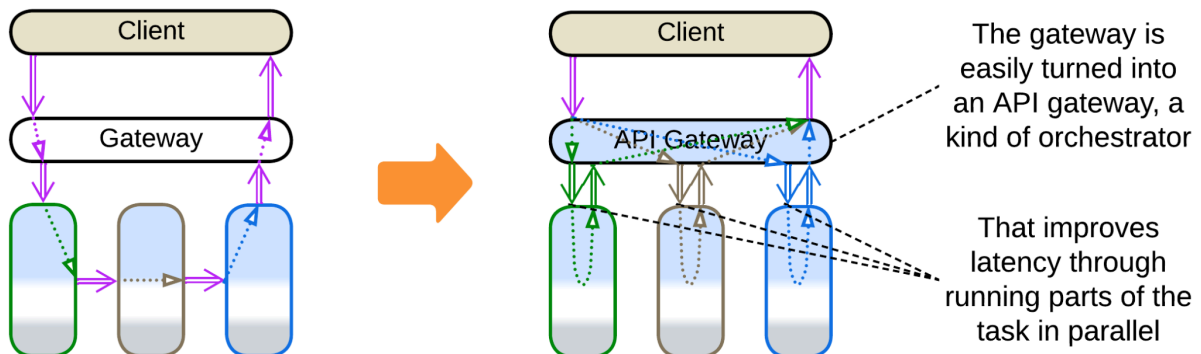
It is possible to avoid both adding the gateway and the cyclic dependency if the client does not immediately need the final results of processing their request. In such a case the service that receives the original request does its (first) step of processing, sends the response to the client, then notifies services down the pipeline. Though such a use case seems to be unlikely, it happens in real life, for example, with pizza delivery. As soon as a buyer fills in their contact details and pays for the food, the order can be confirmed and forwarded to the kitchen. When it is ready it's forwarded to the delivery, and finally the physical goods appear at the buyer's door.

It is much easier if only the first service deals with the client session

The rest of the system will eventually produce the result
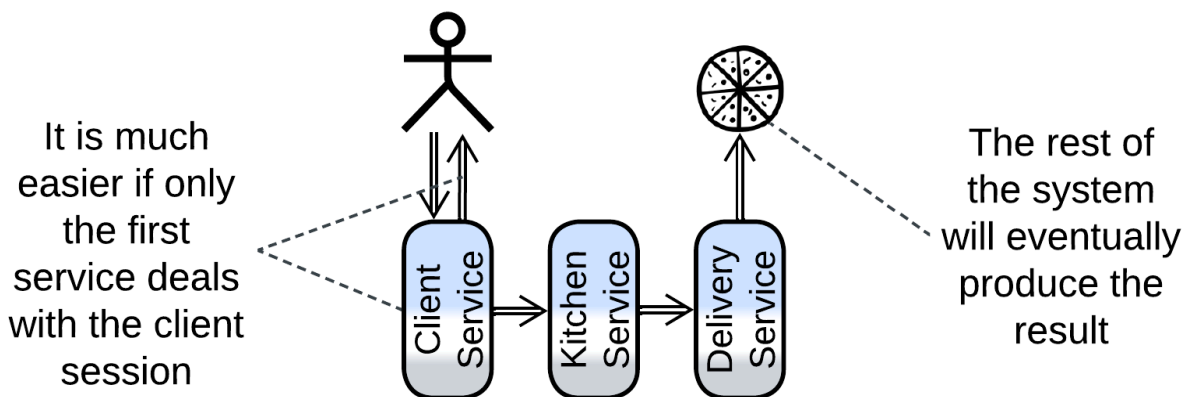
Early response allows for choreography in its purest form: with extensibility, high performance but also high latency. A similar approach may be used in *Service-Based Architecture* [FSA] (aka macroservices) for communication between the services (bounded contexts) as they may only need to notify each other of events without waiting for responses.

Dependencies

A pipeline may be built with downstream or upstream dependencies or with a shared schema.

If services communicate through commands, each service depends on all the direct destinations of its commands as it must know their APIs. This mode of communication is mostly used with actors that power embedded, telecom, messengers and some banking systems. Downstream dependencies make it easy to add input chains (upstream services that deal with various hardware or external components) while changing anything at the output end of the pipeline is going to break the input parts that send messages to the component changed.

Command-based communication makes each service depend on the service(s) it sends commands to
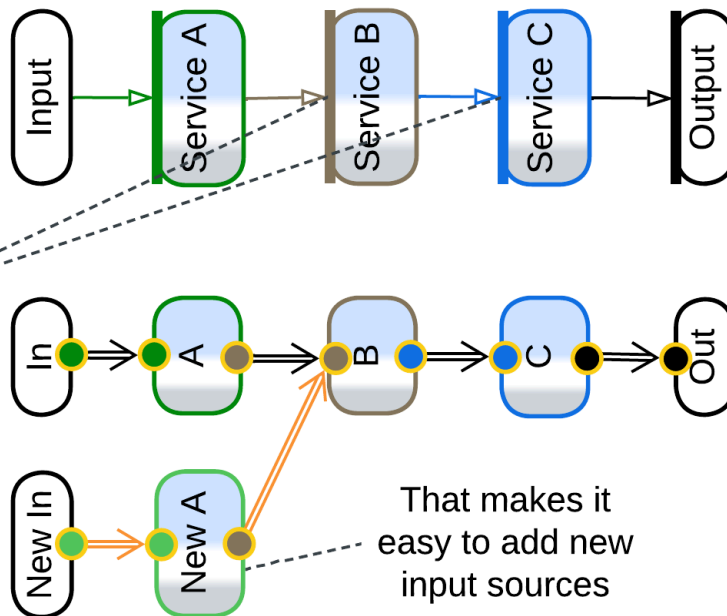
That makes it easy to add new input sources

Upstream dependencies come from the publish/subscribe model where each service spreads notifications about what it does to any interested subscriber. This way of building systems is common with *Event-Driven Architecture* [FSA] which is used in high-load backends. Extending or truncating the already implemented request processing tree is as easy as adding/removing subscribers to existing events but creation of a new event source will require changes in the downstream components. The addition of downstream branches supports new customer experience and analytical features that the business is hungry for.

With notifications (pub/sub) the dependencies point upstream
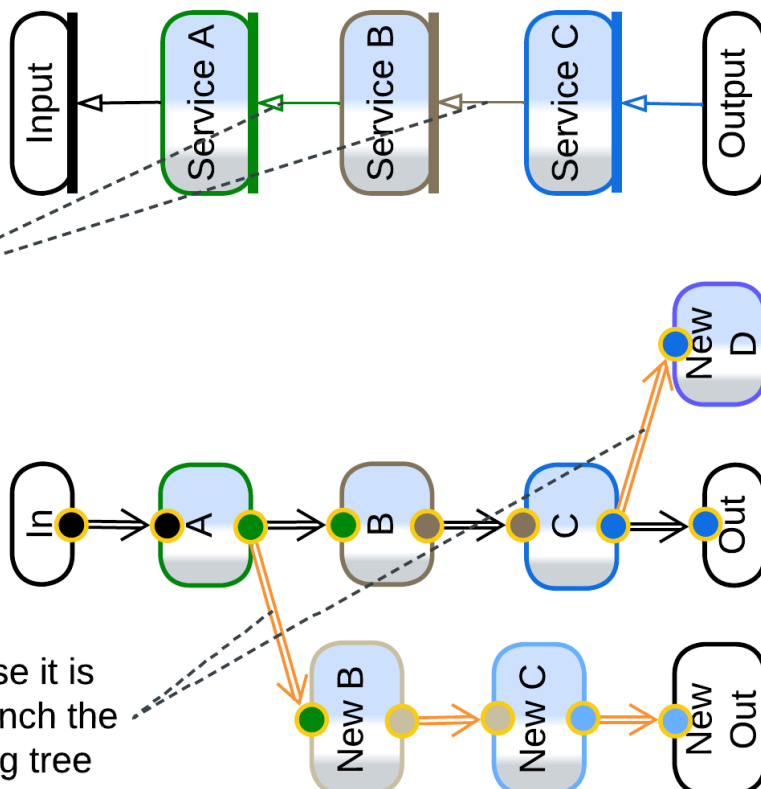
In that case it is easy to branch the processing tree

The final option is to have the entire pipeline use a single message format (*Stamp Coupling* [SAHP]), usually with a field dedicated to each service. This way a service depends

only on the message header (with the list of the fields and a record id) and the format of the single field it reads (stores data) from or writes (retrieves data as *Content Enricher* [EIP]) to. That works well for system-wide queries but binds all the services to the schema of the message, in a way similar to relying on a shared database (to be discussed in the next section). Such an architecture decouples the services to the extent that any of them can be freely added or removed, together with the message field(s) it fills or reads.

With a shared message format each service depends on the message header ...

... and on one or two fields it interacts with

Shared Message

We can remove and add services ...

... together with their data fields in the message
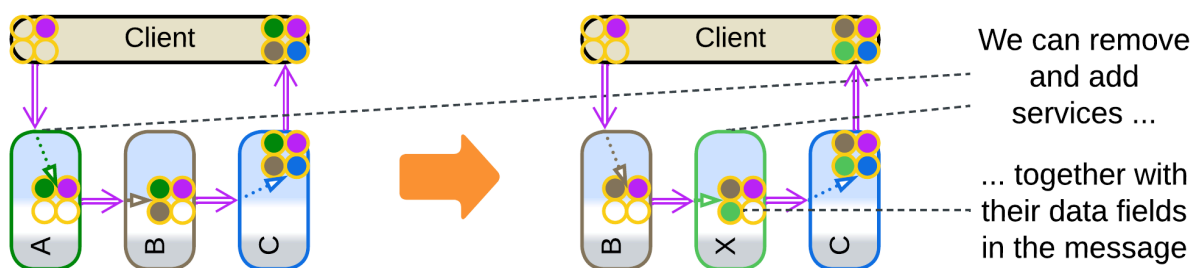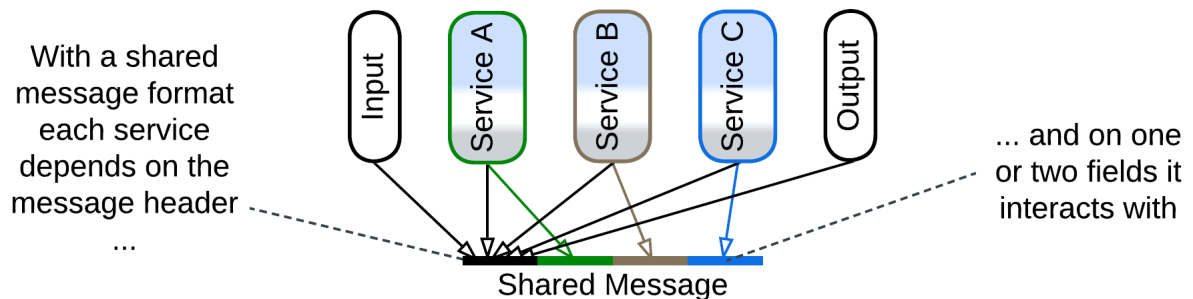
A peculiar feature of choreography is the ability to cut and cross-link pipelines with compatible interfaces by changing a single service (or even system config if you rely on communication channels). That gives much flexibility – as long as you can comprehend all the dependencies (and channels) in the system, which becomes non-trivial as it grows.
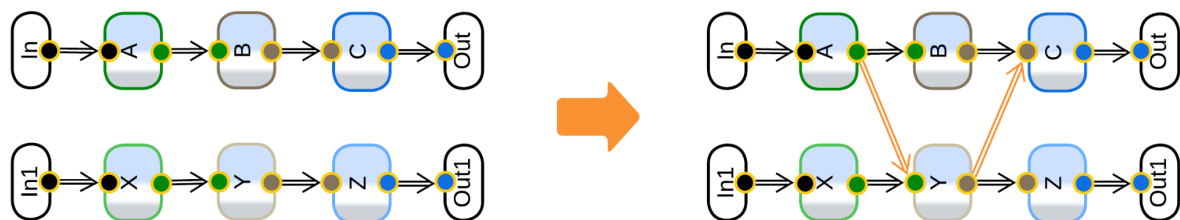
Multi-choreography

It is very common for a service to participate in several pipelines, especially if it owns a database – as there should be a use case that fills in the data and at least one other use case that reads from the database. This means that such a service depends on multiple interfaces that often belong to multiple services, coupling components of the system and making it resist future structural changes.

A service may take part in multiple pipelines

Each pipeline creates dependencies between services

Opposite direction of pipelines introduces cyclic dependencies
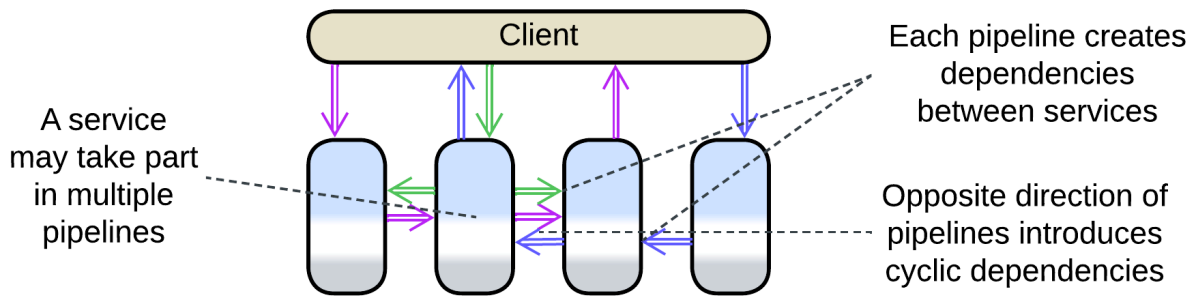
Summary

Overall, choreography seems to be a lightweight approach that prioritizes throughput over latency and is suitable for highly-loaded scenarios of limited complexity. A choreographed system will likely become unintelligible if it is made to support more than a few use cases.

There is a decent overview from Microsoft.

Shared data

The final option is integration through shared data:

Each service processes the data which is shared among them

The services don't know of each other
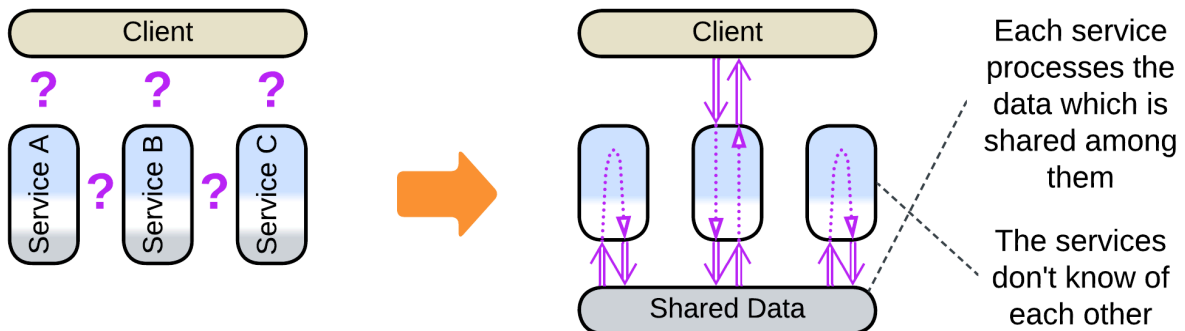
The shared data is a "blackboard" available for each service to read from and write to. It is passive (is used by the services) and does not contain any logic except for the data schema, which represents a part of domain knowledge. That makes communication through shared data an antipode of orchestration, which also features a shared component, but an orchestrator is active (uses services) and contains business logic, not data.

Shared data can be used for storage, messaging or both:

Storage

The most common case of shared data is storage (usually a database, sometimes a file system) for a (sub)domain that has functionally independent services that operate on a common dataset. For example, a ticket purchase service and a ticket return service share a database of ticket details. The ticket purchase service reads in free seats and fills in ticket data for purchases. The ticket return should be able to find all tickets bought by a user and delete the user data from seats returned. The only communication between the purchase and return services is the shared database of tickets or seats, so that one of them sees the changes made by the other the next time it reads the data.

The purchase service reads empty seats, shows them to the user and fills the user's details into the chosen seat

The return service finds the seat the user has bought and resets it to an empty state

Each service sees all the seats
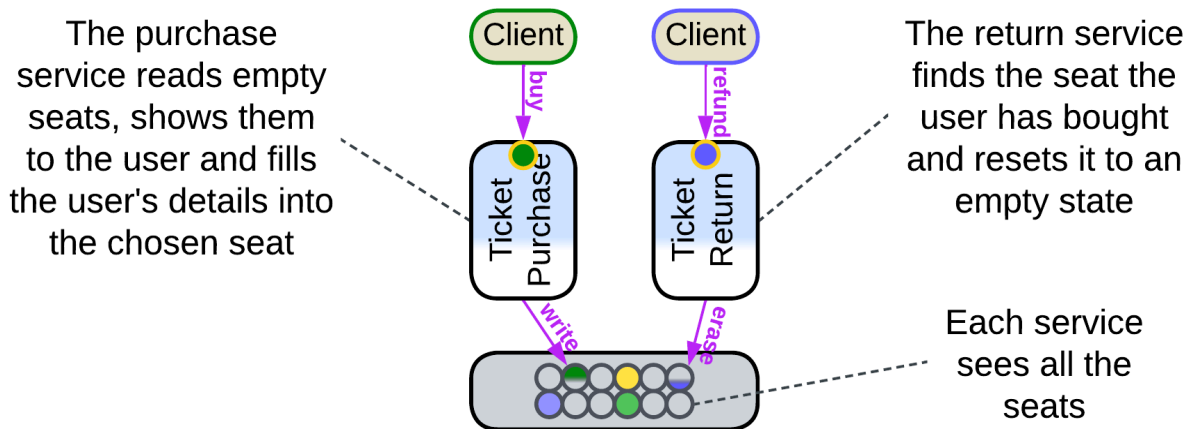
With this model the services don't depend on each other – instead, they depend on the shared (domain) data format and the database technology. Thus, it is easy to add, modify or remove services but hard to change the data structure or database vendor.

Each service depends on the database schema and technology

The services are independent and easy to replace

Adding a service requires no changes in the existing components
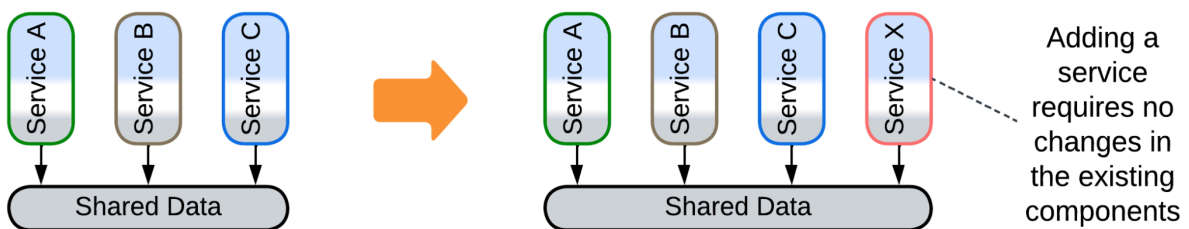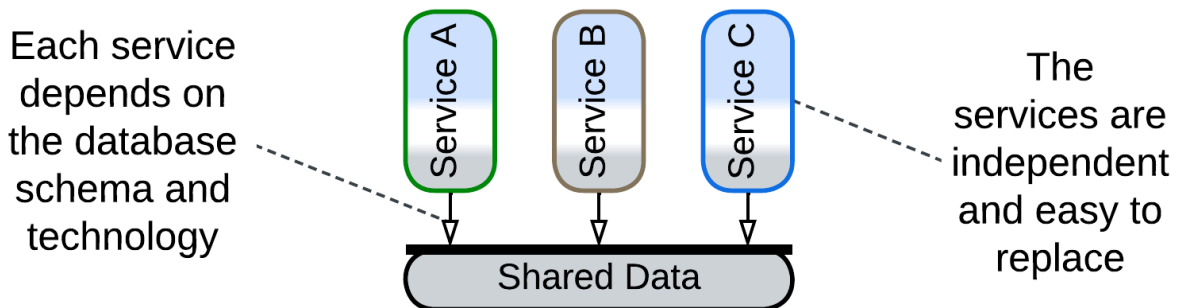
Services often need to coordinate their actions.

Most commonly, services with a shared database rely on a messaging *middleware* to notify each other about their actions. Users of our ticketing system want to be notified (through email, SMS or an instant message) when a free seat that they are interested in appears. We're not going to complicate either of the existing services with integration with instant messengers, so we create a new notification service, which must track each returned ticket to see if any user wants to buy it. This is easily implemented by the return service publishing and the notification service subscribing to a ticket return event.

Services may message each other when they change the data

However, messaging makes the services interdependent
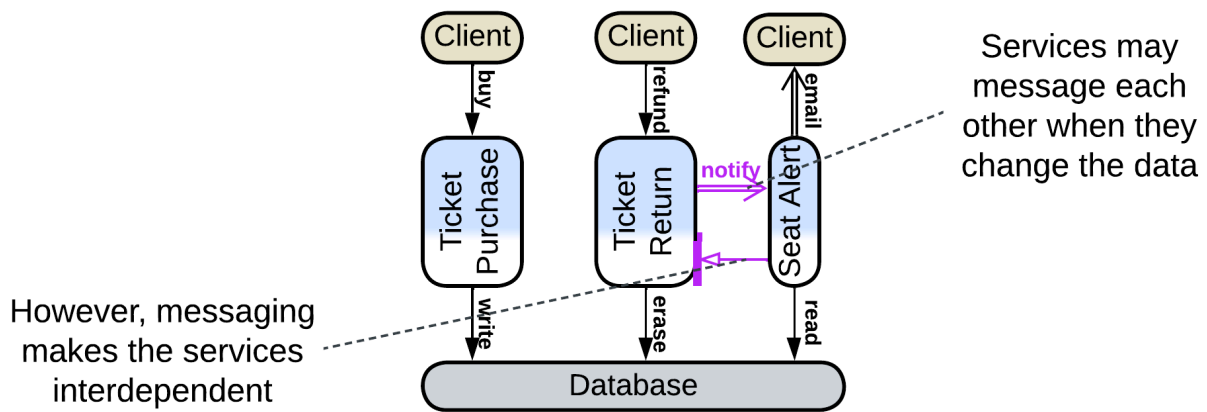
Another case is found with data processing pipelines where an element may periodically read new files from a folder or new records from a database table to avoid implementing notifications. This increases latency and may cause a little CPU load when the system is idle, but is perfectly ok for long-running calculations.

Each step of the algorithm reads a file, processes it and writes a new file

As soon as a file has been written the next step takes it for input

Which finally produces a batch of output files that have passed all the steps of processing ...

The input is provided as a batch of files
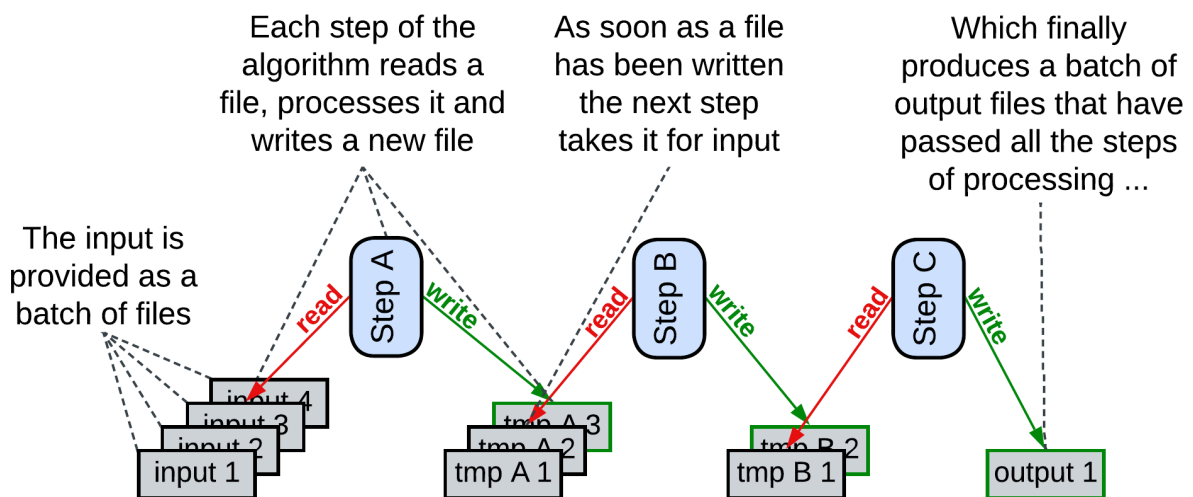
Finally, there is a rarely used option of an external *scheduler* that selects which services should run based on the data available. This is known as the *Blackboard* pattern and something similar happens in 3D game engines. The scheduler (which is an *orchestrator*, by the way) is needed when CPU (or GPU or RAM) resources are much lower than what the services would consume if all of them run in parallel, thus they must be given priorities, and the priorities change based on the context which is regularly estimated from the data.

Various algorithms work together towards finding a solution

The scheduler chooses the best algorithm to run next ...

The shared data: inputs and outputs of calculations

... based on the latest data
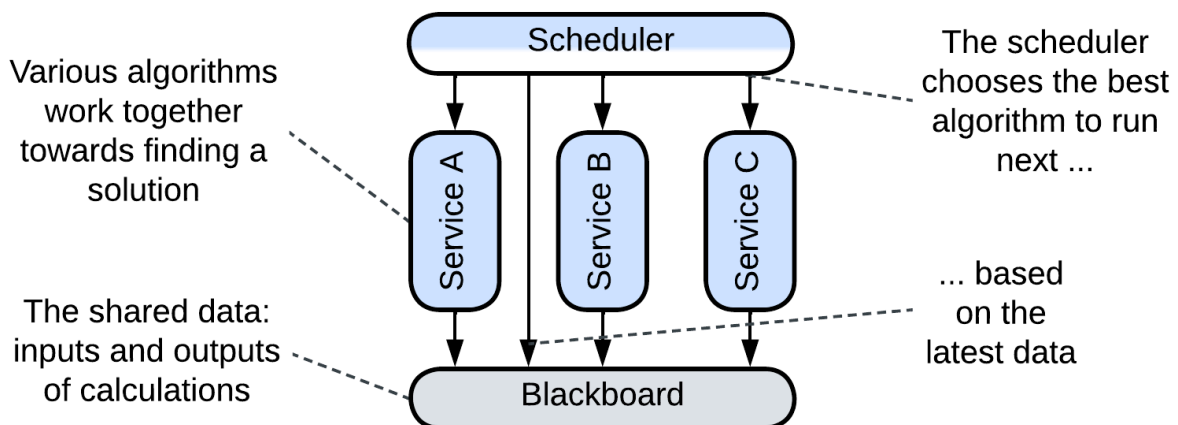
Messaging

The other, not as obvious, use case for shared data is messaging, which is implemented by the sender writing to a (shared) queue (or log) while the recipient waits to read from it. Queues can be used for any kind of messages: request/confirm pairs, commands or notifications. Each service may have a dedicated queue (either input for commands mode, or output for notifications), a pair of queues (messages from the service's output are duplicated by an underlying distributed middleware to input queues of their destinations), there may be a queue per communication channel or a single queue for the entire system, with each message carrying destination id (for commands) or topic (for notifications).
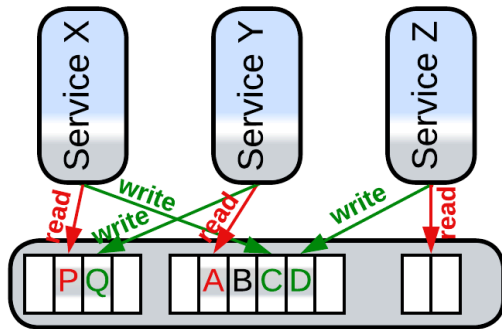
A Queue per Service
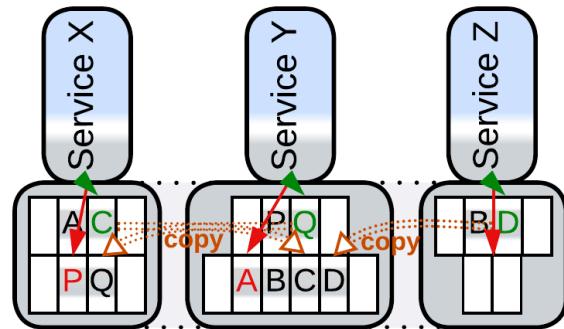
Input and Output Queues
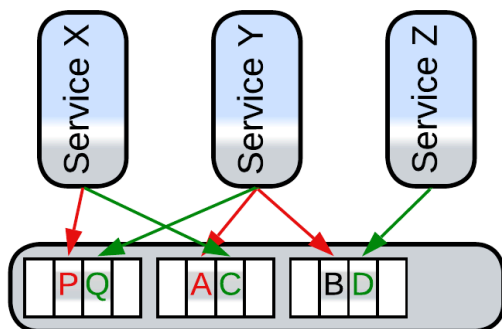
A Queue per Channel
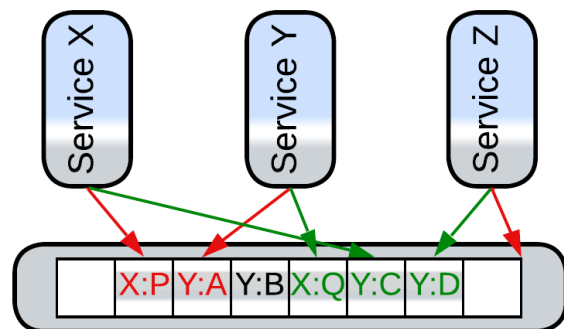
A Single System Queue

The use of shared data for messaging turns the datastore into a _middleware_. The dependencies are identical to those in choreography – each service depends on APIs of its destinations for commands or its sources for notifications.

There should be a means for the recipient of a message to know about its arrival so that it starts processing the input. Usually a messaging middleware implements a method to receive a message for the service to block on. However, very low latency applications, like HFT, may busy loop reading the shared memory so that the service starts processing the incoming data immediately on its arrival, skipping the OS scheduler. This is the fastest means of communication available in software.

Full-featured

Finally, some (usually distributed) datastores implement data change notifications. That allows for the services to communicate through the datastore in real-time, removing both the need for an additional middleware and interdependencies of the services. Such a system follows the _Shared Repository_ pattern of [POSA4] which was rectified as _Space-Based_

Architecture [SAP, FSA]. In our example the free seats notification service subscribes to changes in the seats data in the database, this way it does not need to be aware of the existence of other services at all. We can also split the email part of the ticket purchase server to a separate entity which would track purchases in the database and send a printable version of each newly purchased ticket to the buyer's mail address which is found in the ticket details in the database.

The datastore may let the services subscribe for data change notifications

The system is now data-centric with the data layer used for both storage and communication
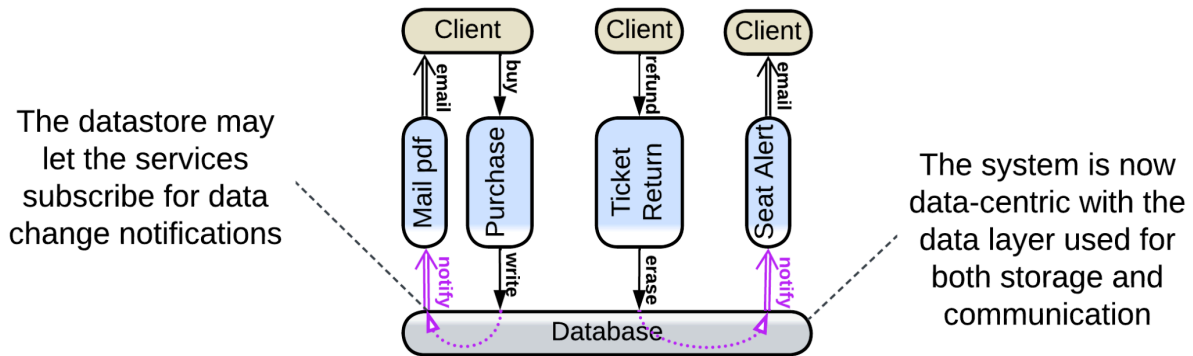
Summary

Communication through shared data is best suited for data-centered domains (for example, ticket purchase). It allows for the services to be unaware of each other's existence, just as they are with orchestration, but the structure of the domain data becomes hard to change as it is in use throughout the code. Shared data may also be used to implement messaging.

Comparison of the options

We have briefly discussed three communication metapatterns: orchestration, choreography and shared data. Let's see when it makes sense to use each of them.

Orchestration is built around use cases. They are easy to program and add, no matter how complex they become. Thus, if your (sub)domain is coupled, or your understanding of it is still evolving, this is the way to go, as you will be able to change the high-level logic in any imaginable way.

Shared data is all about… er… domain data. If you really (believe that you) know your domain, and it deals with coupled data – this is your case. You may even add an orchestrator if there are use cases that involve multiple subdomains. The business logic is going to be easy to extend while changes in the data schema will likely break much of the code.

Choreography pays off with weakly coupled domains and few simple use cases. It has good performance and flexibility, but lacks the expressive power of orchestration and becomes very messy as the number of tasks and components grows. It works best with independent teams and delayed processing – when the user does not wait for an immediate result of their action.

There is advice from Microsoft which makes perfect sense: use choreography for communication between _bounded contexts_ (subdomains) but revert to orchestration (or maybe shared data) inside each context. Indeed, subdomains are likely to be loosely coupled while most user requests don't traverse subdomain boundaries – which kindles hope that their interactions are few and not time-critical. If we follow the advice, we get

Cell-Based Architecture ([WSO2 definition](#)), which collects the best of two worlds: orchestration and/or shared data for strongly coupled parts and choreography between them.
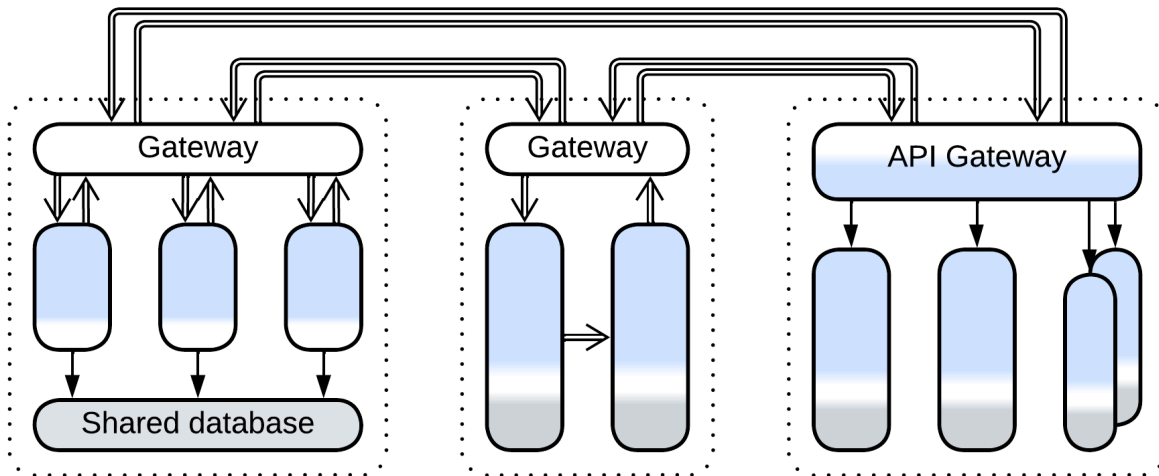
By the way, you could have noticed the odd cases:
- An orchestrator in a control system does not run scenarios and its mode of action resembles choreography.
- A choreographed system may use a shared message format, which makes it resemble a system with shared data, even though no shared database is present.
- A shared database may be used to implement messaging for an orchestrated or choreographed system.

Those probably mean that our distinction between the modes of communication is a bit artificial and there is some deeper model to look for.

Part 2. Basic Metapatterns

Basic metapatterns are both widespread stand-alone architectures and building blocks for more complex systems. They comprise the single-component monolithic architecture and the results of its division along each axis of coordinates (abstractness, subdomain and sharding):

Monolith

Monolith is a (sub)system the internals of which we prefer not to see
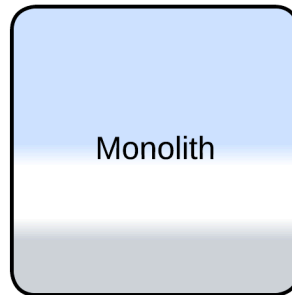
Monolith

It may lack an internal structure or its components may be coupled

Monolith is a single-component system, the simplest possible architecture. It is easy to write but hard to evolve and maintain.
Includes: Reactor, Proactor, Half-Sync/Half-Async.

Shards

Shards are multiple instances of a subsystem
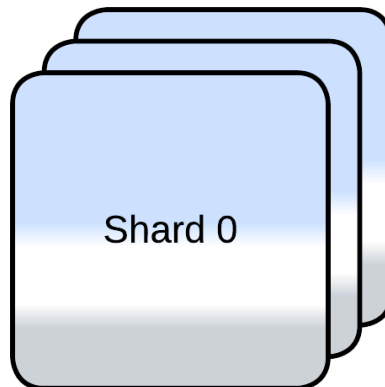
Shard 0

A shard may be stateless or stateful

Shards are multiple instances of a monolith. They scale but usually require an external component for coordination.
Includes: Shards and Amazon Cells, Replicas, pool of stateless instances, create on demand.

Layers

Layers divide the system by the level of abstractness
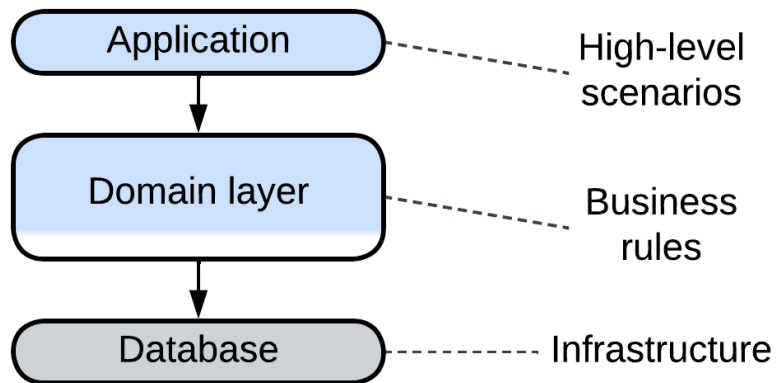
Layers contain a component per level of abstractness. The layers may vary in technologies and forces and scale individually.

Includes: Layers and Tiers.

Services

Each service implements a subdomain
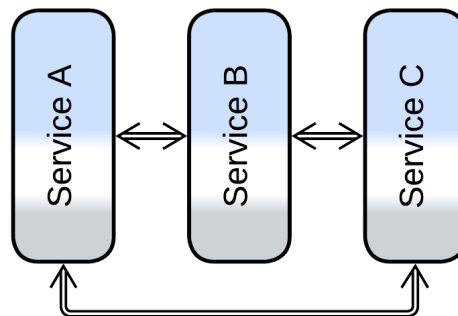
The subdomains should be loosely coupled

Services divide the system by subdomain, often resulting in parts of comparable size to be assigned to dedicated teams. However, a system of services is hard to synchronize or debug.

Includes: Service-Based Architecture, Modular Monolith (Modulith), Microservices, device drivers, actors.

Pipeline

The system processes data in a sequence of steps
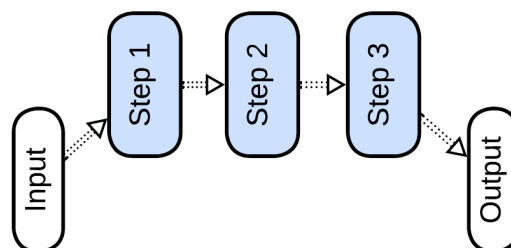
Each step knows nothing of its neighbors

Pipeline is a kind of Services with unidirectional flow. Each service implements a single step of request processing. The system is flexible but may grow out of control.

Includes: Pipes and Filters, Choreographed Event-Driven Architecture, Nanoservices, Data Mesh.

Monolith

Let's take a look at the simplest possible metapattern – *Monolith* – and see what it can teach us about.
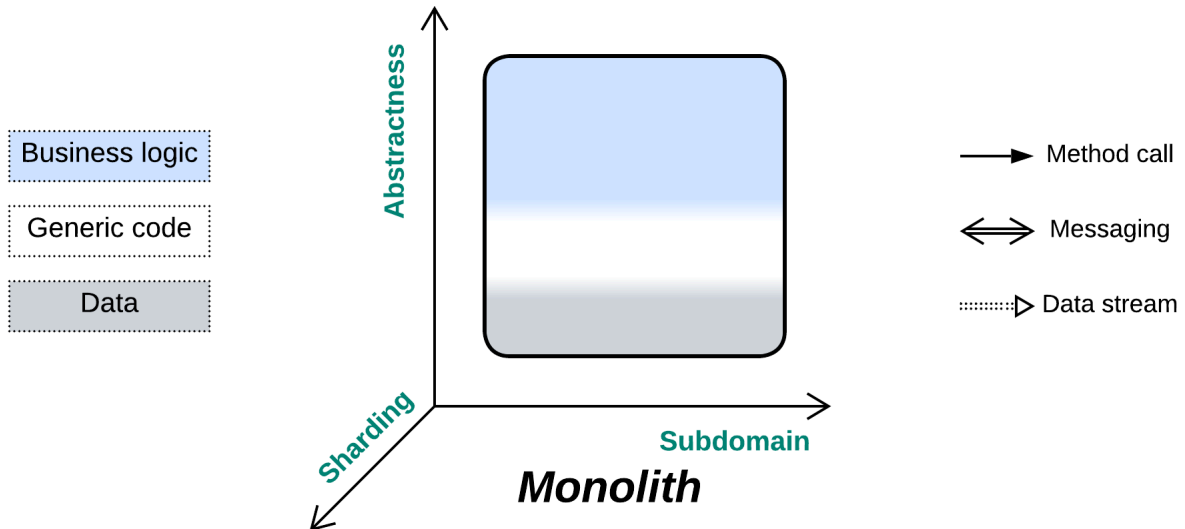
Monolith

Keep it simple, stupid! If you don't need a modular design, why bother?

Known as: Monolith.

Variants:

By the internal structure:

● True Monolith / Big Ball of Mud,
● (misapplied) Layered Monolith [FSA],
● (misapplied) Modular Monolith [FSA] (Modulith),
● (inexact) Plugins [FSA] and Hexagonal Architecture.

By the mode of action:

● Reactor [POSA2],
● Proactor [POSA2],
● (inexact) Half-Sync/Half-Async [POSA2],
● (inexact) (Re)Actor-with-Extractors.

Structure: A monoblock with no strong internal modularity.

Type: Main, root of the hierarchy of metapatterns.

Benefits	Drawbacks
Rapid start of development	Quickly deteriorates with project growth
Easy debugging	Hard to develop with multiple teams
Best latency	Does not scale
Low resource consumption	Lacks support for conflicting forces
The system's state is self-consistent	Any failure crashes the entire system

References: Big Ball of Mud for a philosophical discussion, my article and [POSA2] for the subtypes of *Monolith*, Martin Fowler's discussion on starting development with monolith, [MP] for the definition of monolithic hell and a post describing the first-hand experience of it.

We distantiate ourselves from the system architecture's definition of *Monolith* as a single unit of deployment as our main focus lies with the internal structure of systems. Instead, we

will use the old definition of monolithic application as a cohesive lump of code containing no discernible components.

Monolith is non-modular (not divided by interfaces) along all the structural dimensions. Its thorough cohesiveness is both its blessing (single-stepping debugging, system-wide optimizations) and its curse (messy code, no scalability of development and deployment, zero flexibility).

Performance

On one hand, monolithic applications provide perfect opportunities for performance optimizations as every piece of code is readily accessible from any other. On the other hand, if the application is stateful, the state must be protected from race conditions, thus the performance benefit of using multiple CPU cores is limited. Furthermore, large monoliths may become too messy for the programmers to identify and too complicated and fragile for them to implement any drastic measures towards better performance.

Overall, tiny monoliths provide the best latency and throughput per CPU core. Larger performance-critical projects may need to partition the code into *layers* or *services* so that any manually optimized part remains small enough to be manageable. Higher throughput is attainable through distributing the software over multiple computers: *sharding* employs multiple copies of the whole system while a *pipeline* may run each step of data processing on a separate server.

Dependencies

Even though *Monolith* is a single module, thus there are no dependencies among its parts (actually, everything depends on everything), it may depend on external components or services which it uses. Those dependencies tend to cause *vendor lock-in* or make the software OS- or hardware-dependent. *Hexagonal Architecture* (or *MVC* as its variant) is the way to decouple a monolithic implementation from its dependencies.

Applicability

Monolith is <u>good</u> for for the cases where the introduction of modularity causes more trouble than help:

- *Tiny projects.* The project is relatively small (below 10 000 lines) and the requirements will never change (e.g. you need to implement a library for running a specific mathematical calculation or interfacing a well-established communication protocol).
- *Ultra optimization.* You already have a working and thoroughly optimized system, but you still need that extra 5% performance improvement achievable through merging all the components together.
- *Low latency.* If you need ultra low latency for the entire application, any asynchronous communication between its modules is not a viable option. Example: high-frequency trading.
- *Prototyping.* You are writing a prototype in a domain that you are not familiar with, gathering requirements in the process. The chances for a correct initial identification of weakly coupled subdomains to be converted to modules are <u>quite low</u>, while it is worse to have wrong module boundaries than to have no modules at all. At the later stages of the project, when you will know the domain much better, and your users will

have approved the initial implementation, you will be able to split the system into modules in a correct way, if and when that will be needed. Nevertheless, you may already know enough to apply *Layers* or *Hexagonal Architecture* which keep the business logic monolithic while isolating it from periphery and 3rd party libraries.

- *Quick and dirty.* You are out of time and money, and need to show your customers something right now. No time to think, no money to perfect the code, no day after tomorrow.

Monolith should be avoided when we need modules:

- *Incompatible forces.* There are conflicting forces (non-functional requirements) for different subsets of functionality. They require splitting the system into (usually asynchronous) modules with each of the modules specifically designed to satisfy its own subset of the forces. Your main tool is the careful selection of appropriate technologies and architecture on a per module basis. That may allow the project to satisfy all the non-functional requirements even if the task looks impossible on the initial consideration.
- *Long-running projects.* The project is going to evolve over time, while you think you can predict the general direction of the future changes. Modularity brings flexibility, and the flexibility you will need.
- *Larger code bases.* The project grows above average (100 000 lines of code). If you don't split it into smaller modules now, it will get into monolithic hell, with development and debugging slowing down year after year, till it reaches the terminal stage. Slow development is a waste of money, both in salary and in time to market.
- *Multiple teams.* You have multiple teams to work on the project. Inter-team communication is hard and error-prone, while merging several teams together is known to greatly reduce the programmers' productivity (it peaks for teams of 5 or less members). Explicit interfaces between modules will formalize the interdependencies of the teams, lowering the communication overhead.
- *Fault tolerance.* Your domain requires fault tolerance which is really hard if not impossible for large monolithic applications.
- *Resource-limited.* Your project is too resource-hungry for commodity hardware. Even if you buy the best server for its needs right now, it is going to crave for more tomorrow (or on the next Black Friday).
- *Distributed setup.* Your project needs to use multiple hardware devices. One of common examples is a web service containing frontend and backend.
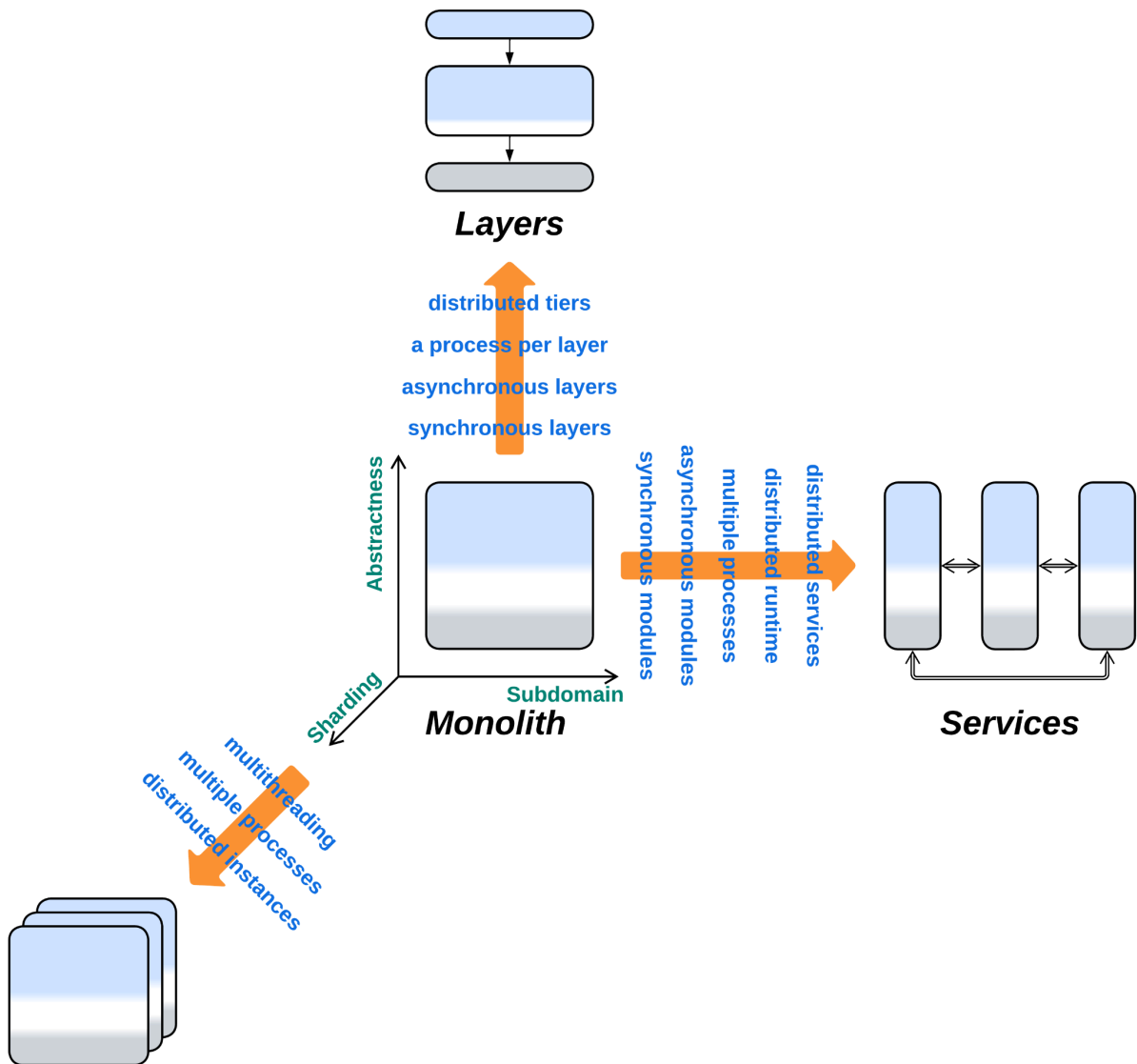
Relations

Monolith:
- Can be extended with *Proxy*, *Hexagonal Architecture* or *Plugins*.
- Yields *Layers*, *Services* or *Shards* if divided along the *abstractness*, *subdomain* or *sharding* dimensions, correspondingly. All the known architectures are combinations of those three metapatterns.
- Is the bird's-eye view of any architecture.

Variants by the internal structure

Monoliths are the atoms to create more complex architectures from, the opaque building blocks, each of which satisfies a consistent set of forces. Any individual component of a more complex architecture either is monolithic or encapsulates another architectural pattern, and any architecture looks monolithic to its clients.
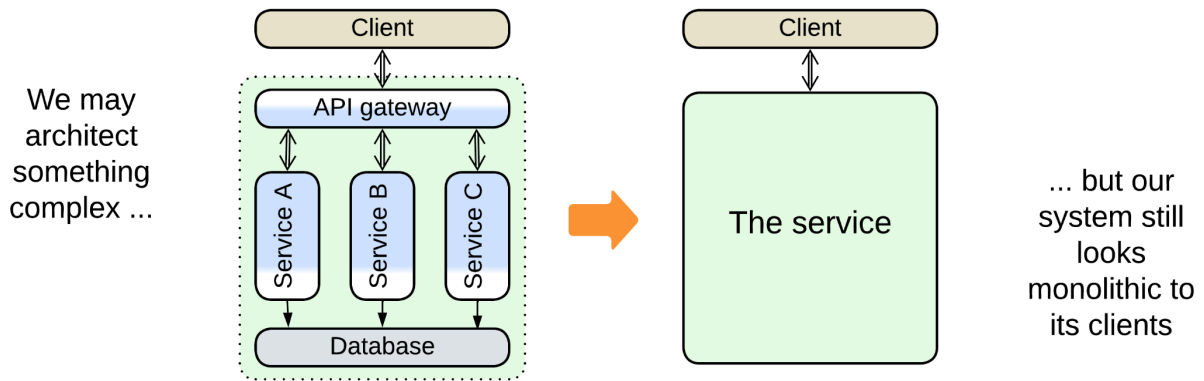
Thus, there exists a misunderstanding because *software architecture* inspects internals of *applications* at the level of *modules* or even classes while *system architecture* deals with *distributed systems* and operates *deployment units* which tend to contain multiple modules or even applications. Each of the branches of architecture calls its atomic unit "monolith", leading to the term sticking both to a *module that cannot be subdivided*, as in [GoF] and [POSA1], and to a (sub)system that must be deployed together, as in modern literature.

As we aspire to build a unified classification for both distributed and local systems, we must treat components of both kinds in the same way, whether they are distributed services, co-located actors or in-process modules. Thus, for the scope of the current book, we follow the definition from [GoF]: "Tight coupling leads to *monolithic* systems, where you can't change or remove a class without understanding and changing many other classes". Still, we need to account for a couple of misnomers from system architecture:

True Monolith, Big Ball of Mud

A true monolith has no clear internal structure. If it has any components, they are so tightly coupled that the entire thing behaves as a single cohesive module. This is what we explore in the current chapter.

(misapplied) Layered Monolith
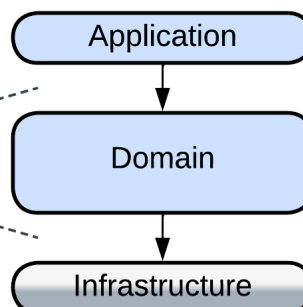
When they say "layered monolith" [FSA], that means a non-distributed application with layered structure, which is a proper _Layers_ architecture, and will be discussed in the corresponding chapter. It is called "monolith" for the sole reason that it is not distributed. Nevertheless, _Layers_ resemble _Monolith_ in many aspects, including easy debugging and the danger to outgrow the zone of comfort for developers.

(misapplied) Modular Monolith (Modulith)

A "modular monolith" has subdomain modules

Just like the Services architecture
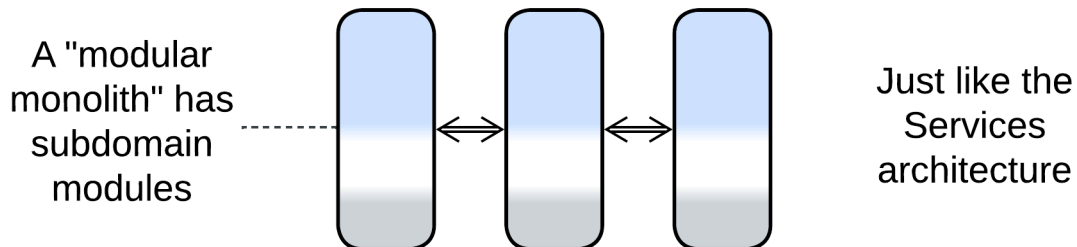
A "modular monolith" [FSA] is a single-process application subdivided into modules that correspond to subdomains. If the modules communicate via in-process messaging, the architecture is nearly identical to coarse-grained actors, thus it is a monolith only in name. _Modulith_ is a kind of _Services_ ("If it walks like a duck and it quacks like a duck, then it must be a duck") – it supports development by multiple teams, and the asynchronous variant is hard to debug. The relation to _Monolith_ is mostly restricted to the inability to scale individual parts of the system.

(inexact) Plugins and Hexagonal Architecture

Hexagonal Architecture extends its core with external modules

Input   Output

Core

DB   LibX   LibY

As the core is monolithic, the entire system behaves as a monolith
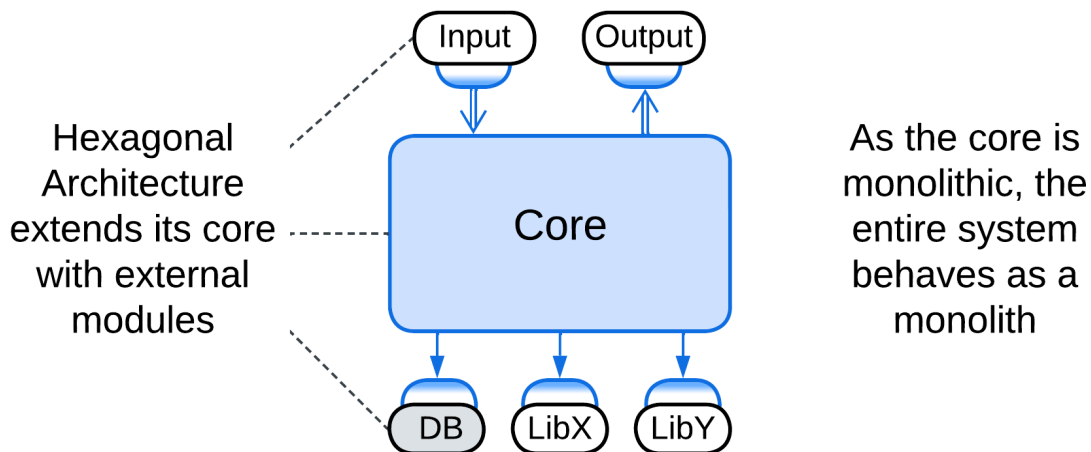
Plugins [FSA] and _Hexagonal Architecture_ extend a (sub)system with external modules. They can be applied to a monolith without drastically changing its properties – it still remains relatively easy to write and debug but hard to support when overgrown. Therefore, we will not currently discuss these modifications, mainly because each of them got a dedicated chapter.

Variants by the mode of action

Let's take a look inside a monolith.

Any software module reacts to incoming events or data and produces outgoing events or data. There are several basic ways to implement that cycle:
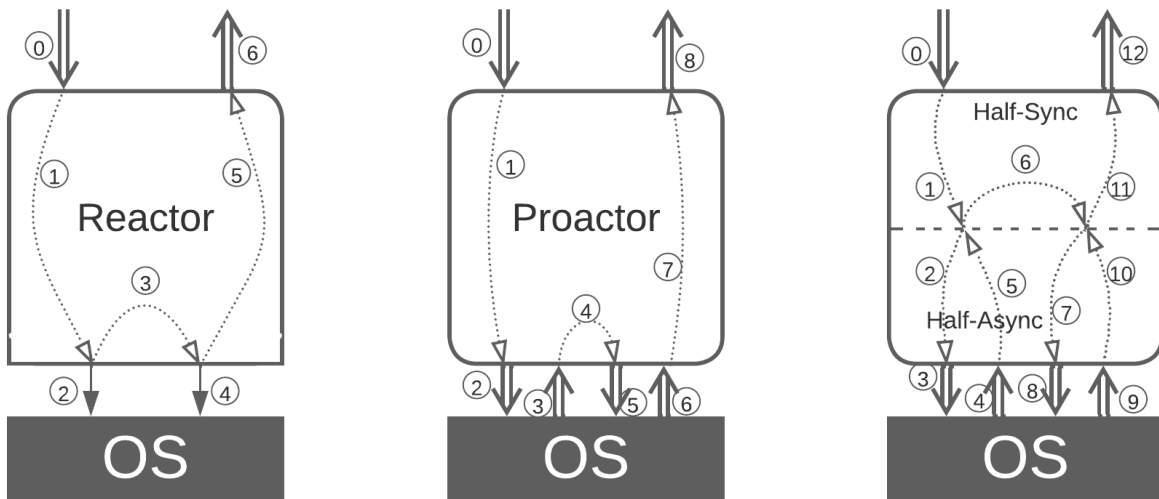
Single-threaded Reactor (one thread, one task)
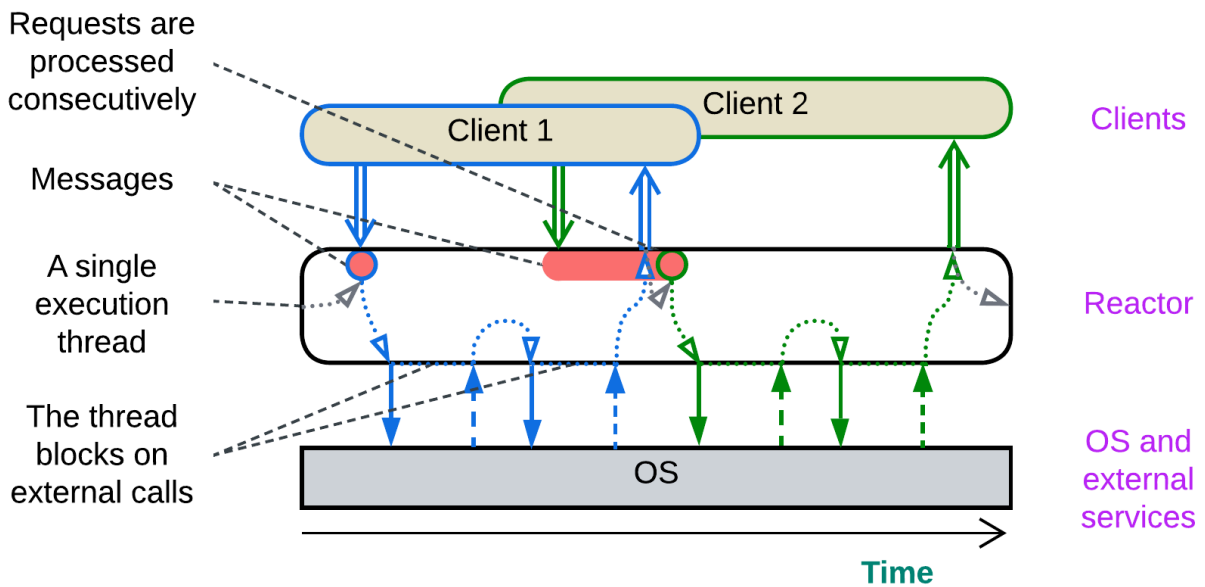
In *Reactor* [POSA2], a single thread waits for an incoming event or data packet, processes it with blocking calls to the underlying OS, hardware and external dependencies and returns the result, rinse and repeat.

That makes sense when the module owns and provides access to a hardware component which cannot do several actions at once, for example, a communication bus or a HDD firmware which does a single read or write at any given moment.

Multi-threaded Reactor (a thread per task)

Requests are processed in parallel

There are multiple threads

The threads blocks on external calls

Clients

Client 2

Client 1

Reactor

OS

OS and external services

Time
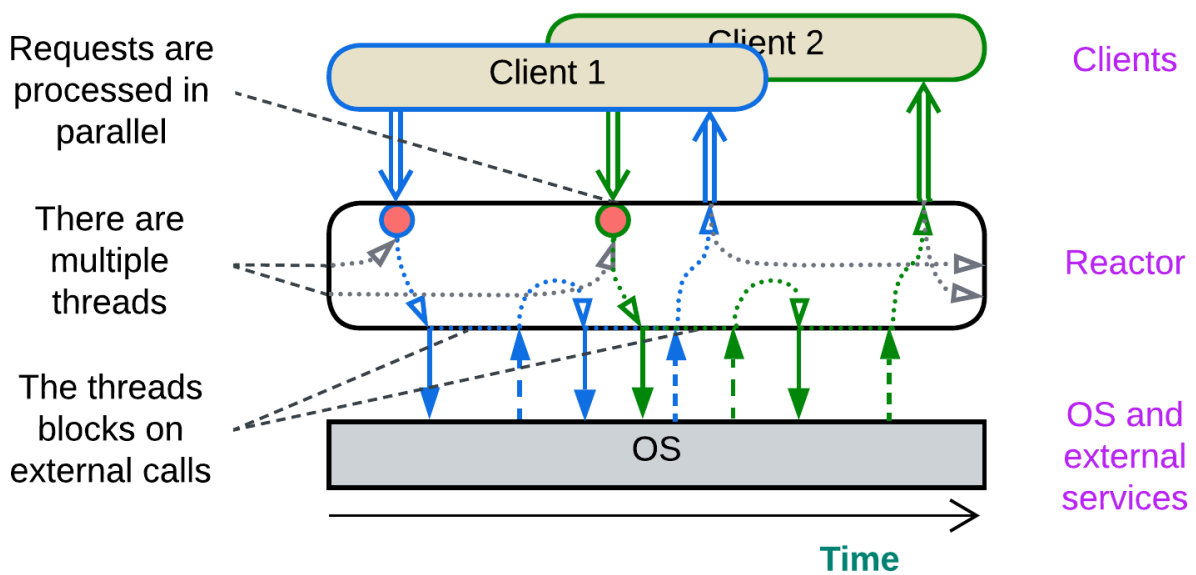
A *reactor* [POSA2] may employ multiple threads by having a *pool* of them waiting for a request or data to come. The event activates one thread, which becomes dedicated to processing it by doing several blocking calls and, finally, sending back a response. When the request processing is completed, the thread returns to the *pool* of idle threads to wait for the next event to process.

This is the default simple&stupid implementation of backend services. Its pitfalls include contention on shared resources, deadlocks and high resource consumption by the OS-level threads.

Proactor (one thread, many tasks)

Requests are processed in parallel

A single thread serves all the requests

The thread never blocks

Clients

Client 2

Client 1

Proactor

OS

OS and external services

Time
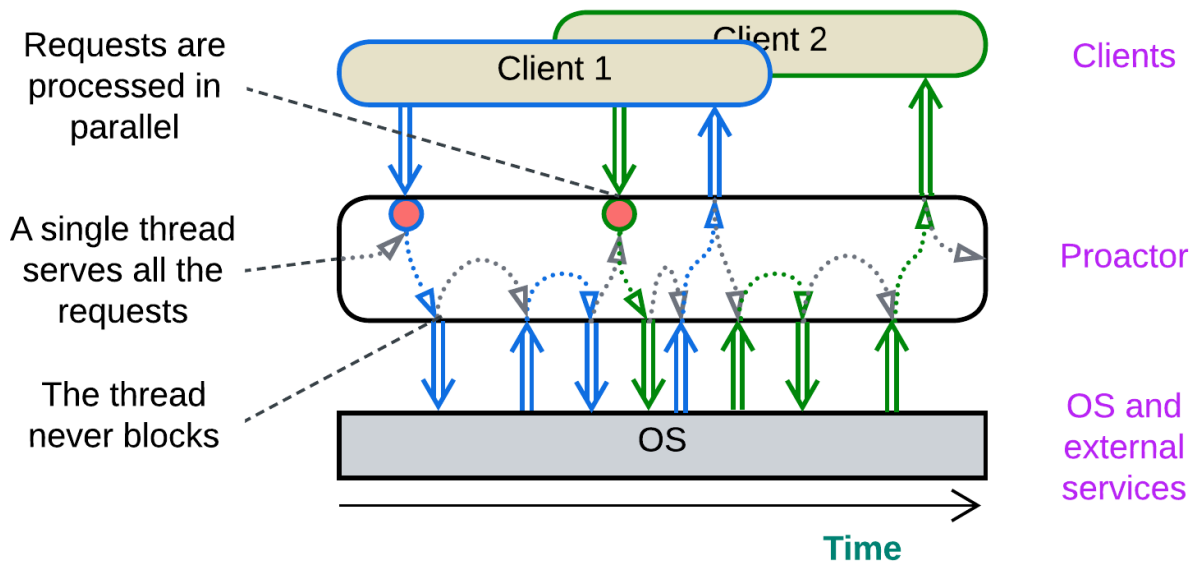
In *Proactor* [POSA2], a single thread processes all the incoming events, both from the module's clients and from the hardware or dependencies it manages. When an event is received, the thread goes through a short piece of corresponding business logic (*event handler*) and usually initiates one or more non-blocking actions, such as sending messages to other components, writing to registers of the managed hardware or initiating an async I/O.

Then the thread is ready to process any further events. As the thread never blocks, it is able to serve multiple interleaved tasks.

This approach is good for real-time systems where thread synchronization is largely forbidden because of the associated delays and for reactive control applications which mostly adapt to the environment instead of running pre-programmed scenarios. The drawback is very poor structure of the code and debuggability as any complex behavior is broken into many independent event handlers.

(inexact) Half-Sync/Half-Async (coroutines or fibers)
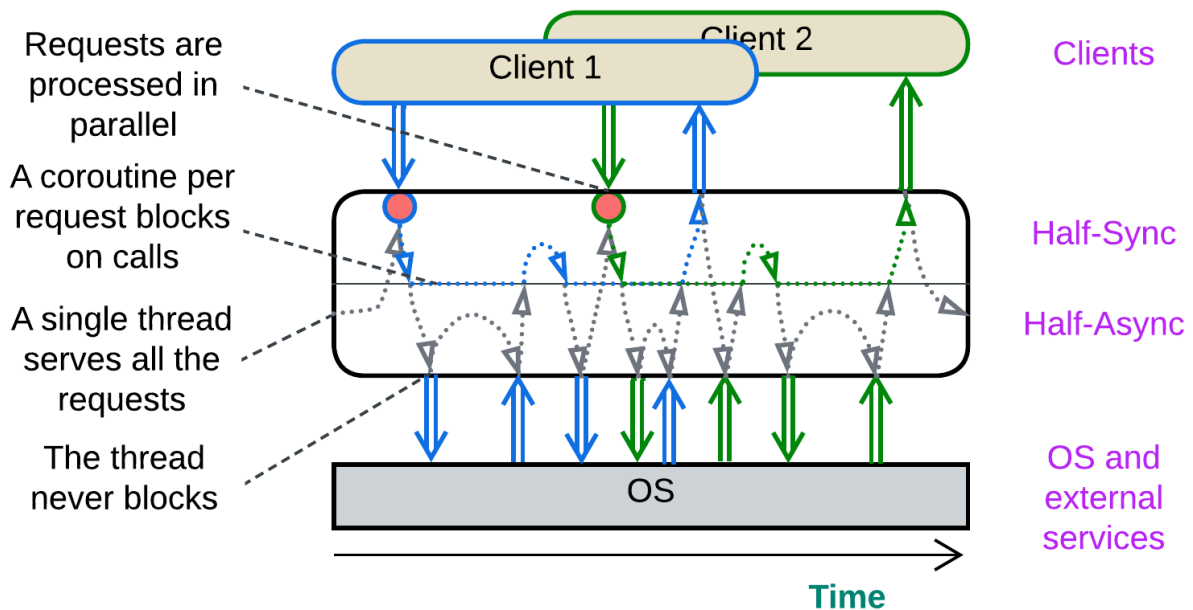
Half-Sync/Half-Async [POSA2] originally described the interaction between user space and kernel threads in operating systems and was (later?) rebranded as coroutines and fibers. A single thread or a thread pool with a thread per CPU core processes all the incoming events, with the execution thread(s) switching call stacks. Every incoming request is allocated a call stack which stores the state (local variables and methods called) of processing the request. When it needs to access an external component, the runtime saves the request's stack, does a non-blocking call, and the execution thread returns to its original stack to wait for an event to come while the request processing stack remains frozen till the current action completes asynchronously. Then the runtime switches the execution thread back to the stored request's stack and continues processing the request till it is completed and the stack is deleted.

This makes programming and debugging tasks as easy as with *Reactor* (procedural style) while retaining the low resource consumption and high performance of *Proactor*. Coroutines and fibers are used in highly efficient game engines and databases. Though *Half-Sync/Half-Async* contains two layers (is not truly monolithic), I believe it belongs next to *Reactor* and *Proactor* which make up its upper and lower layers, correspondingly.

The state of the art

These patterns are not widely known, and programmers tend to mix them together, for better or for worse. One is likely to encounter a heavily multithreaded big ball of mud where some threads serve user requests while others are dedicated to periodic service routines.

Moreover, people tend to call any event-driven service *Reactor*, causing confusion among those who discern between the three patterns above.

(inexact) (Re)Actor-with-Extractors (phased processing)

During extract phases each object queries its neighbors

It plans actions and puts them on its message queue

| Scheduler | Scheduler |

Extract Phase   **React Phase**

In react phases objects execute the actions they planned

Extract and react phases alternate

——Extract— React —Extract— React —Extract— React ——→ **Time**
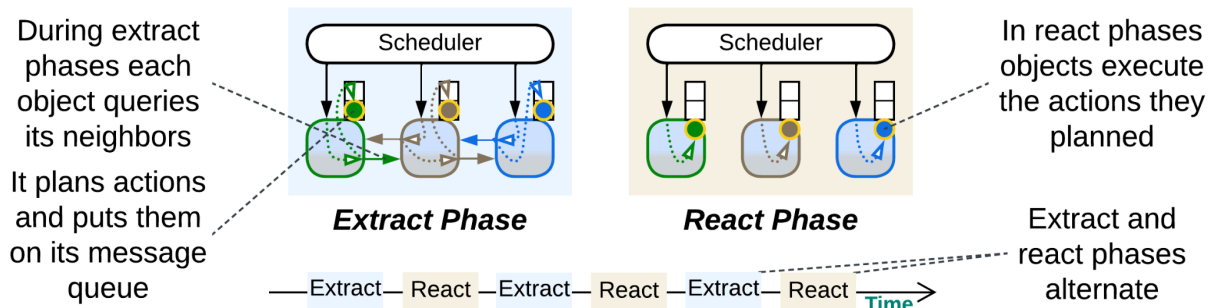
As a bonus, let's review an unconventional execution model that fits gamedev or other kinds of simulation with many interacting objects.

We have a long-running system where each simulated object has complex behavior that depends on objects around it. The common wisdom proposes two ways to implement it:

- *Actors* (asynchronous messaging, reactive programming) – each *actor* (simulated object) runs single-threaded and wakes up only to process incoming messages. While processing a message, an *actor* may change its state and/or send messages to other *actors*. The entire *actor*'s data is private and there are no synchronous calls between the *actors*. The good thing is that *actors* are very efficient in highly parallel tasks as they use no locks in their code. The bad thing is that *actors* have no way to synchronize their states: you can only request another actor to tell you about its state, and its response may become outdated even before you start processing it. Also, any complex logic that involves multiple *actors* is dispersed over several event handlers.
- The opposite option is to have the simulated objects access each other synchronously. This allows for complex logic that depends on states of multiple objects, but gets in trouble with changing the objects' states from multiple threads: you need to protect them with those inefficient locks and you get those dreadful deadlocks as the outcome.

Here we have two bad options to choose from. However, it is the simulated nature of the system that saves the day: we can *stop the world to get off*. The objects' querying each other and their changing their states needs neither happen at the same time nor obey the same rules!

The simulation runs in steps. Each step consists of two phases:

- *Query phase* (*extraction*), when the object states are immutable, thus the objects can communicate synchronously with no need for locks. In this phase each object collects information from its surroundings (other objects), plans its actions and posts them as commands to its own message queue. I suppose that objects may also send commands to each other in this phase.
- *Command phase* (*reaction*), when each object executes its planned (queued) actions that change its state, but cannot access other objects.

Each phase lasts till every object in the system completes its tasks scheduled for the phase. The phase toggle is supervised by a *scheduler* which runs the objects on all the

available CPU cores. The entire process resembles the [game of Mafia](#) with its public daily conversations and covert nightly actions.

(Re)Actor-with-Extractors is a perfect example of earning benefits of two architectures without paying their drawbacks. It utilizes both the lockless parallelism of *actors*-style *[shared-nothing](#)* and the simplicity of synchronous access in *[shared-memory](#)* by alternating between those modes through application of the *[CQRS principle](#)* to the time dimension.

Evolutions

Every architecture has its drawbacks, and tends to evolve in a variety of ways to address them. Below is a brief summary with more information available in [Appendix E](#).

Evolutions to Shards

One of the main drawbacks of monolithic architecture is its lack of scalability – a single running instance of your system may not be enough to serve all its clients no matter how much resources you add in. If that is the case, you should consider *[Shards](#)* – **multiple instances** of a monolith. There are following options:

- Self-managed *[shards](#)* – each instance owns a part of the system's data and may communicate with all the other instances (forming a *[mesh](#)*).

Each client should know their shard

Splitting to stateful (true) shards scales the system but takes much work and requires the data of users to be mostly independent

- *[Shards](#)* with a *[load balancer](#)* – each instance owns a part of the system's data, with an external component to select a shard for a client.

The load balancer knows which shard has the client's data

Slicing to independent shards scales the system perfectly but needs a load balancer which increases latency

- A *pool of stateless instances* with a *load balancer* and a *shared database* – any instance can process any request, but the database limits the throughput.

The load balancer chooses any available instance

Stateless instances scale dynamically but the shared database may become a bottleneck

- A *stateful instance per client* with an external persistent storage – each instance owns the data related to its client and runs in a virtual environment (i.e. web browser or an actor framework).

Every user gets a dedicated instance for the duration of their session

Each instance loads its user's data on creation and persists any changes to the database

Evolutions to Layers

Another drawback of *Monolith* is its… monolithism. The entire application exposes a single set of qualities and all its parts (if they ever emerge) are deployed together. However, life awards flexibility: parts of a system may benefit from being written in varying languages and styles, deployed with different frequency and amount of testing, sometimes to specific hardware or end users' devices. They may need to vary in security and scalability as well. Enter *Layers* – a subdivision by the **level of abstractness**:

- Most *monoliths* can be divided into 3 or 4 *layers* of different abstractness.

Layers are intuitive and they may differ in qualities and technologies and scale independently

- It is common to see the database separated from the main application.

Stateful Application → Stateless Application → Database

Your application becomes stateless, thus multiple instances of it can be deployed

The database persists and queries the data

- *Proxies* (e.g. *Firewall*, *Cache*, *Reverse Proxy*) are usual additions to the system.

Client → Entire Application (you write it) → Client → Proxy (3rd party) → Main Application (you write it)

A proxy provides a generic functionality on behalf of your system

That saves you time for implementing business logic

- An *orchestrator* adds a layer of indirection to simplify the system's API for its clients.

The orchestrator lets clients do many things with one request

The internal connection is much faster than the connection to clients

Evolutions to Services

The final major drawback of *Monolith* is the cohesiveness of its code. The rapid start of development begets a major obstacle as the project grows: every developer needs to know the entire codebase to be productive, the changes made by individual developers overlap and may break each other. Such a distress is usually solved by dividing the project into modules along **subdomain boundaries** (which tend to match bounded contexts). However, that requires much work, and good boundaries and APIs are hard to design. Thus many organizations prefer a slower iterative transition.

- A *monolith* can be split to *services* right away.

Services divide the codebase into parts of comparable size that can be developed almost independently
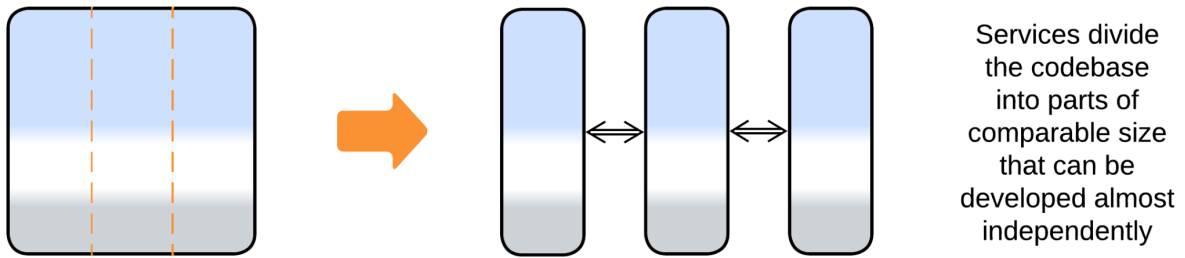
- A new feature may be added or a weakly coupled part separated as a *service*.

Splitting a service allows you to modify its functionality with little danger of breaking unrelated things

- Some domains allow for sequential data processing best described by *pipelines*.

Pipelines use small replaceable components to process streams of data
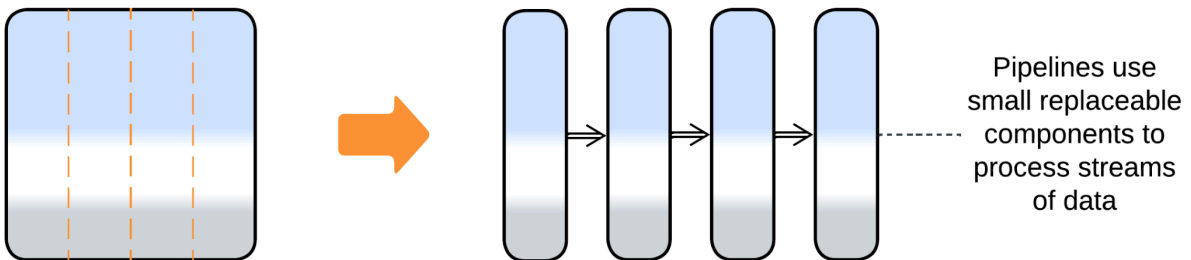
Evolutions with Plugins

The last group of evolutions does not really change the monolithic nature of the application. Instead, its goal is to improve the customizability of the monolith:

- Vanilla *Plugins* is the most direct approach which relies on replaceable bits of logic.

Each plugin customizes a single aspect of the application's behavior

- *Hexagonal Architecture* is a subtype of *Plugins* that is all about isolating the main code from any 3rd party components it uses.

All the external
dependencies
are isolated
with adapters

- *Scripts* is a kind of *Microkernel* – yet another subtype of *Plugins* – which gives users of the system full control over its behavior.

The high-level
business logic
moves to completely
customizable scripts

The implementation
details remain in the
library for use by the
scripts
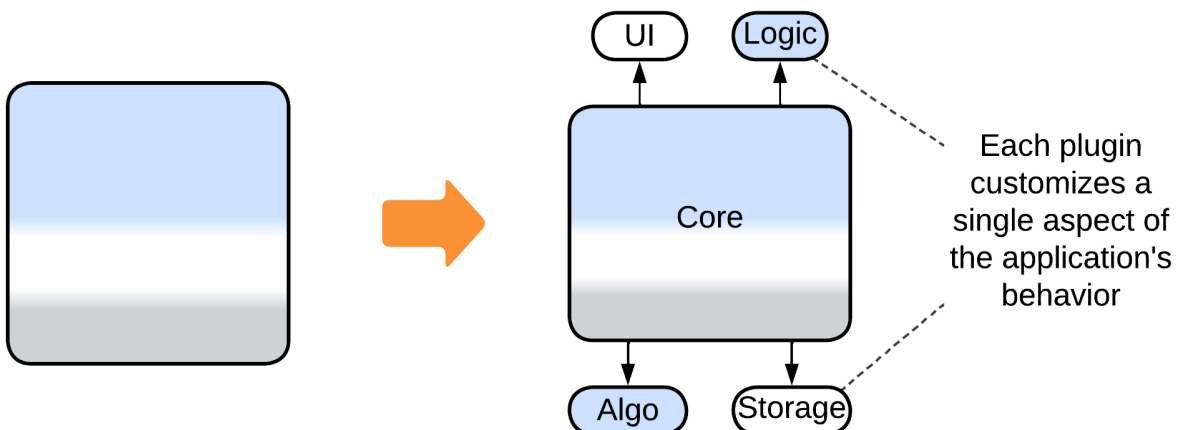
Summary

Monolith is the best architecture for rapid prototyping by a small team and it usually grants the best performance to costs ratio. However, it does not scale, lacks any flexibility and becomes intolerable as the amount of code grows.

Shards
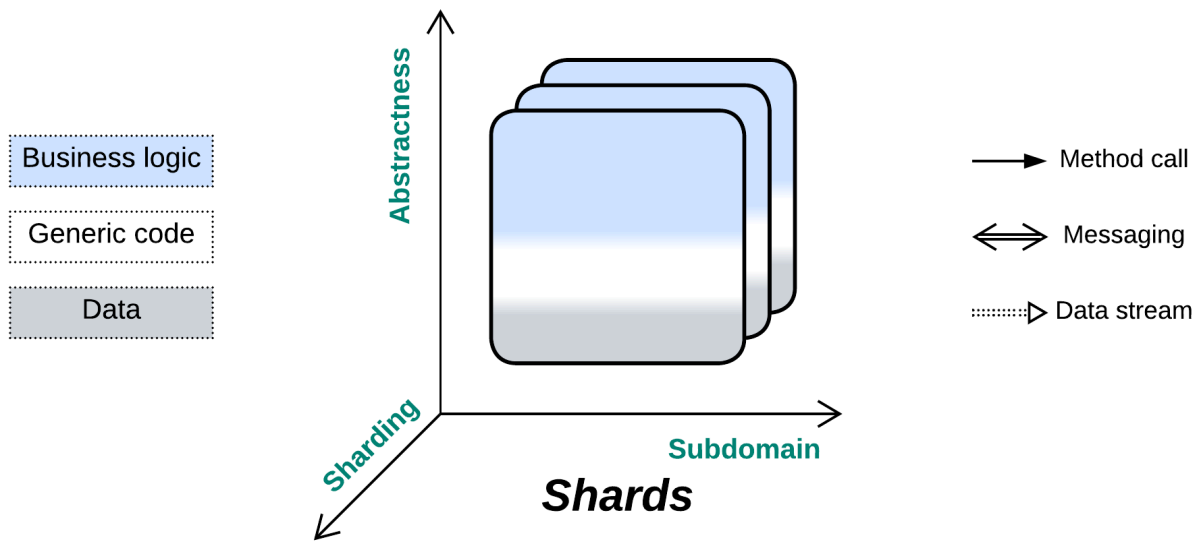
Shards

Attack of the clones. Solve scalability in the most straightforward manner.

Known as: Shards, Instances, Replicas [DDS].

Variants:

By isolation:

- Multithreading,
- Multiple processes,
- Distributed instances.

By state:

- Persistent slice: Sharding / Shards [DDS] / Cells (Amazon definition) / Partitions [DDIA],
- Persistent copy: Replica [DDS],
- Stateless: Pool [POSA3] / Instances / Replicated stateless services [DDS] / Work Queue [DDS],
- Temporary state: Create on Demand.

Structure: A set of functionally identical subsystems which usually don't intercommunicate.

Type: Implementation.

Benefits	Drawbacks
Good scalability	It's hard to synchronize the system's state
Good performance	

References: [POSA3] is dedicated to pooling and resource management; [DDS] reviews shards, replicas and stateless instances; [DDIA] covers sharding and synchronization of replicas in depth; Amazon promotes full-system sharding as *Cell-Based Architecture*.

Shards are multiple and in most cases independent instances of a component (or subsystem) that the pattern is applied to. They provide scalability, often redundancy and sometimes locality at the cost of slicing or duplicating the component's state (writable data), which obviously does not affect inherently stateless components. Most of this pattern's specific evolutions look for a way to combine several shards at the logic or data level.

Performance

Shards retain the performance of the original subsystem they scale (a *monolith* in the simplest case) as long as they run independently. Any task that involves intershard communication has its performance degraded by data serialization and network communication. And if multiple shards need to synchronize their states you are on the horns of a dilemma: damage data consistency through write conflicts or kill performance with distributed transactions [FSA].

| *Fast if the requested data belongs to the current shard* | *Slower if the request concerns the data owned by another shard* | *The slowest when the data of multiple shards must be synchronized* |
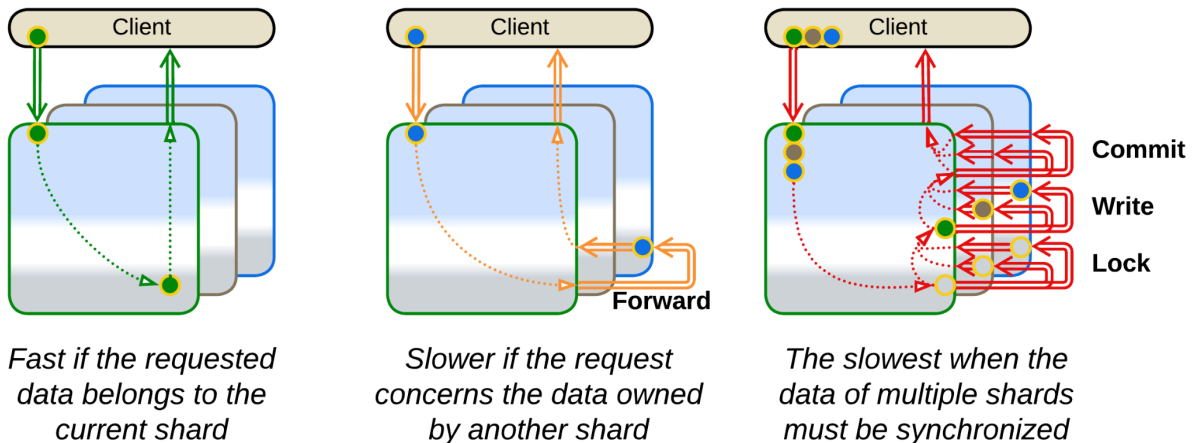
Shared Repository is a common solution to let multiple shards access the same dataset. However, it does not solve the performance vs consistency issue (rooted in the CAP theorem) but only encapsulates its complexity inside a third-party component. Which is not bad at all.

Dependencies

You may need to take care that all the shards are instances of the same version of your software or at least that their interfaces are backward- and forward-compatible.

Applicability

Shards share the properties of the pattern they are applied to (*Monolith* for a single component, *Layers* for an application or *Cell* for a system of services). The peculiarities, owing to the *Shards'* scalability, are listed below:

Shards are good for:

- *High or variable load.* You need to scale your service up (and sometimes down). With *Shards* you are not limited to a single server's CPU and memory.
- *Survival of hardware failures.* A bad HDD or failing RAM does not affect your business if there is another running instance of your application. Still, make sure that your load balancer and internet connection are backed up.
- *Improving worst case latency*. If your service suffers from latency spikes, you can run a few replicas of it in parallel, broadcasting every user request to all of them and returning the fastest response. Adding a single replica turns your p90 into p99.
- *Improving locality.* A world-wide Internet service wins in latency and costs by deploying an instance of its system to a local data center in each region it operates in. And the most responsive code runs in your client's browser!
- *Canary Release*. It is possible to deploy one instance of your application with updated code along with stable old instances. That tests the update in production.

Shards' <u>weak point</u> is:
- *Shared data.* If multiple instances of your application need to modify the same data, no instance properly owns the data, which means that you need help from an external component (a data layer, implying *Layers* and *Space-Based Architecture* or other kind of *Shared Repository*).

Relations

We can have multiple instances of select components ...

... or of the entire system
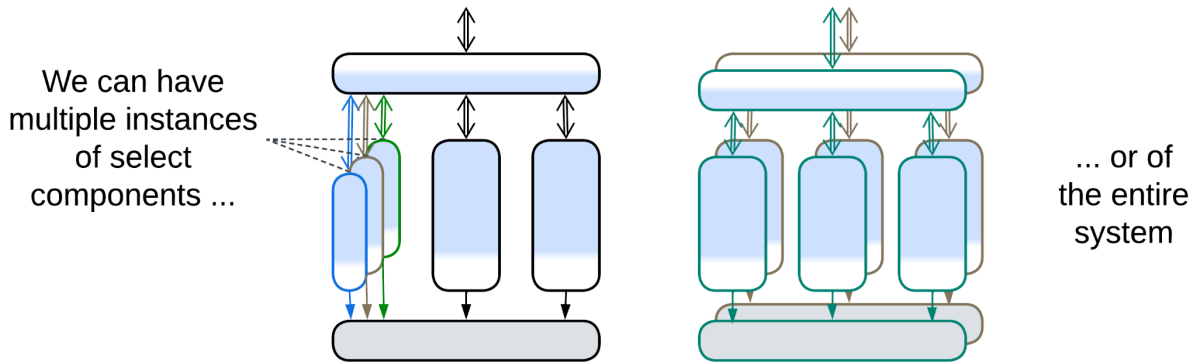
Shards:
- Modifies a *Monolith* or any kind of component or a subsystem.
- Can be extended with *Middleware*, *Shared Repository*, *Proxies* or *Orchestrator*.
- Is the foundation for *Mesh*.

Variants by isolation

There are intermediate steps between a single-threaded component and distributed *Shards*, which gradually augment the pros and cons of having multiple instances of a subsystem:

Multithreading

The first and very common advance towards scaling a component is running multiple execution threads. That helps to utilize all the available CPU cores or memory bandwidth but requires protecting the data from simultaneous access by several threads, which in turn may cause deadlocks.

Benefits	Drawbacks
Limited scalability	More complex data access

Multiple processes

The next stage is running several (usually single-threaded) instances of the component on the same system. If one of them crashes, others survive. However, sharing data among them and debugging multi-instance scenarios becomes untrivial.

Benefits	Drawbacks
Limited scalability	Untrivial shared data access
Software fault isolation	Troublesome multi-instance debugging

Distributed instances

Finally, the instances of the subsystem may be distributed over a network to achieve nearly unlimited scalability and fault tolerance by sacrificing consistency of the whole system's state.

Benefits	Drawbacks
Full scalability	No shared data access
Full fault isolation	Hard multi-instance debugging
	No good way to synchronize state of the instances

Variants by state

Sharding can often be transparently applied to individual components of a data processing system. That does not hold for control systems where decisions are made based on the current state, which must be accessible as a whole, thus the main business logic that owns the model (last known state of the system) cannot be sharded.

Many kinds of *shards* require an external coordinating module (*Load Balancer*) to assign tasks to the individual instances. In some cases the coordinator may be implicit, e.g. an OS socket or scheduler. In others it may be replicated and co-located with each client (as an *ambassador* [DDS]).

Shards usually don't communicate with each other directly. The common exception is *Mesh* (including distributed databases and actor systems) which explicitly relies on communication between the instances.

There are several subtypes of sharding that differ in the ways they handle state:

Persistent slice: Sharding, Shards, Cells (Amazon definition), Partitions

Clients reach the system through a sharding proxy ...

The shards are persistent, each stores a part of the system's state

... which forwards their requests to the shard that owns the first letter of their name
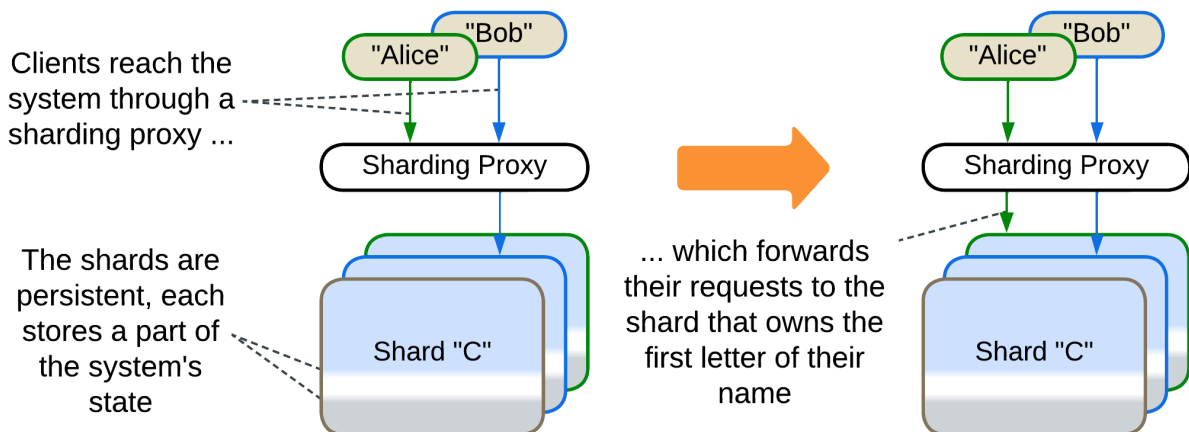
Shards [DDS] own non-overlapping parts of the system's state. For example, a sharded phonebook (or DNS) would use one shard for all contacts with initial "A", another shard for contacts with initial "B" and so on. A large wiki or forum may run several servers, each storing a subset of the articles. This is the proper *sharding*, which is also called *partitioning* [DDIA] in the world of databases.

Cells, according to the [Amazon terminology](#), are shards of a whole system deployed to several data centers, each serving local users. The locality improves latency and saves on internet traffic while having multiple instances of the system up and running improves availability. The downside of the approach is its complexity and amount of traffic needed to keep the *cells* in sync.

It usually takes a stand-alone *[sharding proxy](#)* [DDS] – a kind of *load balancer* – to route client's requests to the shard that owns its data. However, there are other options [DDIA]:

- The *sharding proxy* may be deployed as a client-side *[ambassador](#)* [DDS] to avoid the extra network hop. This approach requires a means for keeping the ambassadors up-to-date with your system's code.
- You can publish your *sharding function* [DDS] and the number of shards in your public API to let your clients choose the shard to access without your help. That may work for internal clients implemented by your or neighbor team.
- Finally, each shard may be able to forward client requests to any other shard – making each shard into a *sharding proxy* and an entry point to the resulting *[mesh](#)*. If your client accesses a wrong shard, their request is still served, though a little slower. This may improve customer experience for sharded databases [DDIA].

Sharding solves scaling of an application both in regard to the number of its clients and to the size of its data. However, it works well only if each client's data is independent from other clients. Moreover, if one of the shards crashes, the information it owns becomes unavailable unless *replication* has been set up as well.

Persistent copy: Replica
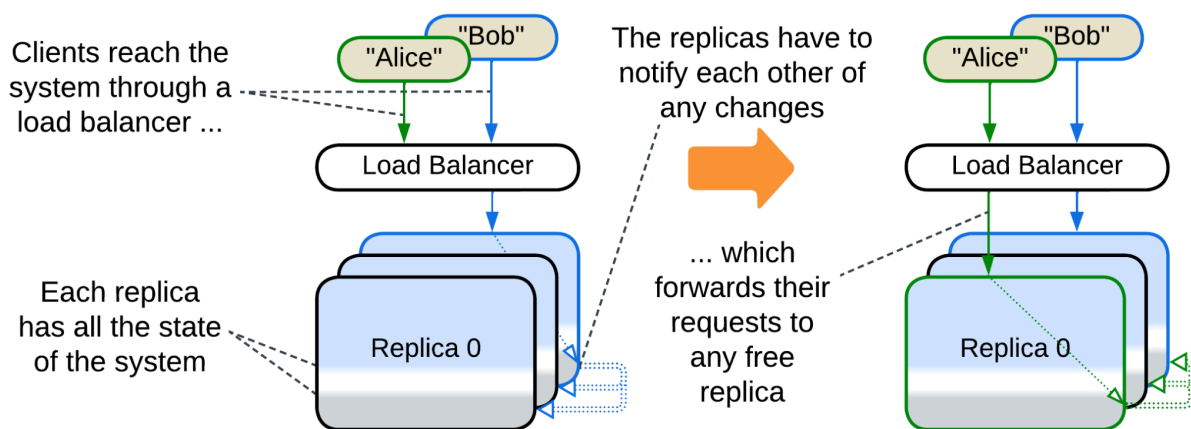
Replicas [DDS] are identical copies of a stateful (sub)system. Replication improves the system's throughput (as each replica serves client requests) and its stability (as a fault in one replica does not affect others which may quickly take up its clients). Replicas may also be used to improve tail latency, if you have enough money to send each request to all the replicas – and use the first response that you receive. Mission-critical hardware [runs in three copies](#) and relies on majority voting for computation results.

The hard part comes from the need to keep the replicas' data in sync. The ordinary way is to let the replicas talk to each other on each data update. If the communication is synchronous, it may greatly slow down the processing of the request, while if it is asynchronous the system suffers data conflicts when multiple clients change the same value simultaneously. The synchronization code is quite complex, thus you will likely use a ready-made *[Space-Based Architecture](#)* framework instead of writing one on your own.

Another option found in the field is to keep the replicas only loosely identical. That happens with *caches*, where each replica is a stand-alone cache for its subset of clients [DDS]. As clients often send similar requests, the content of the cache replicas is mostly, though not completely, identical, thus they are interchangeable.

And if your traffic is read-heavy, you may turn to *Polyglot Persistence* by discerning between derived read-only *followers* and a single fully-functional *leader* [DDIA]. The followers serve only read requests while the leader both updates its data and broadcasts the changes to all the followers. And if the leader dies, one of its followers is elected for the new leader. In some cases the code of the service itself may be separated into write (command) and read (query) services (*CQRS*).

Finally, you can mix *sharding* and *replication* so that the data of each *shard* is replicated in whole to identical components [DDS] or intermixed piecemeal with copies of data from other shards all over the system [DDIA]. That achieves fault tolerance for huge volumes of data which are impossible to store unsharded.

Stateless: Pool, Instances, Replicated stateless services, Work Queue
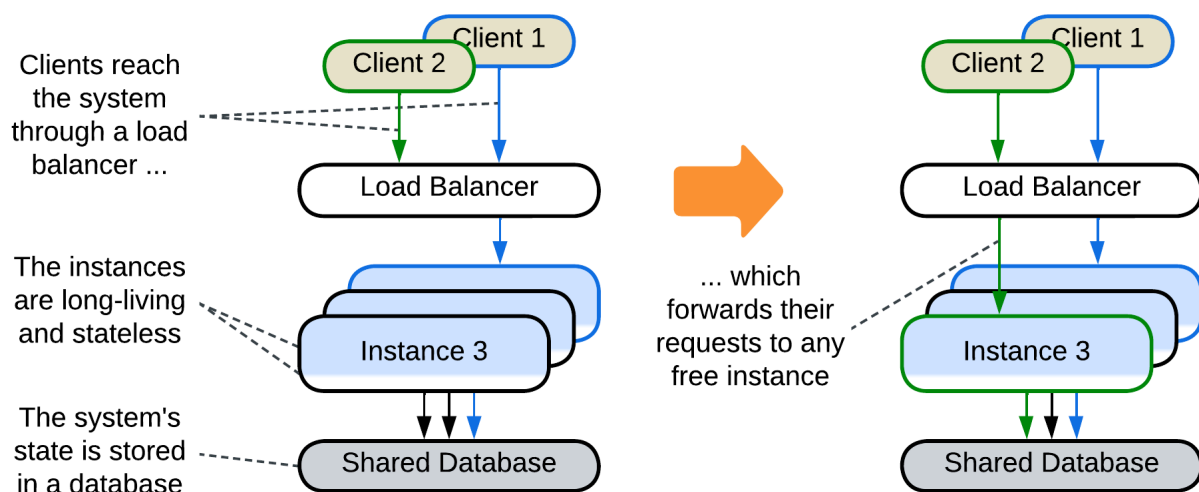
A predefined number (*pool* [POSA3]) of *instances* (*workers*) is created during the initialization of the system (*Work Queue* [DDS]). When the system receives a task, a *load balancer* assigns to one of the idle instances from the pool. As soon as the instance finishes its task it returns to the pool. The instances don't store any state while idle, thus they are also called *Replicated Stateless Services* [DDS]. A well-known example is FastCGI.

The approach allows for rapid allocation of a worker to any incoming task, but it uses a lot of resources even when there are no incoming requests to serve, and the system still may be overwhelmed by a peak load. Moreover, a *shared database* is usually required for persistent storage, limiting the pattern's scalability.

Many cloud services implement dynamic pools, the number of instances in which changes with the load: if all the current instances are busy serving user requests, new instances are created and added to the pool. If some of the instances are idle for a while, they are destroyed. Dynamic pooling often comes with *Mesh*, as in *Microservices* or *Space-Based Architecture*.

Temporary state: Create on demand
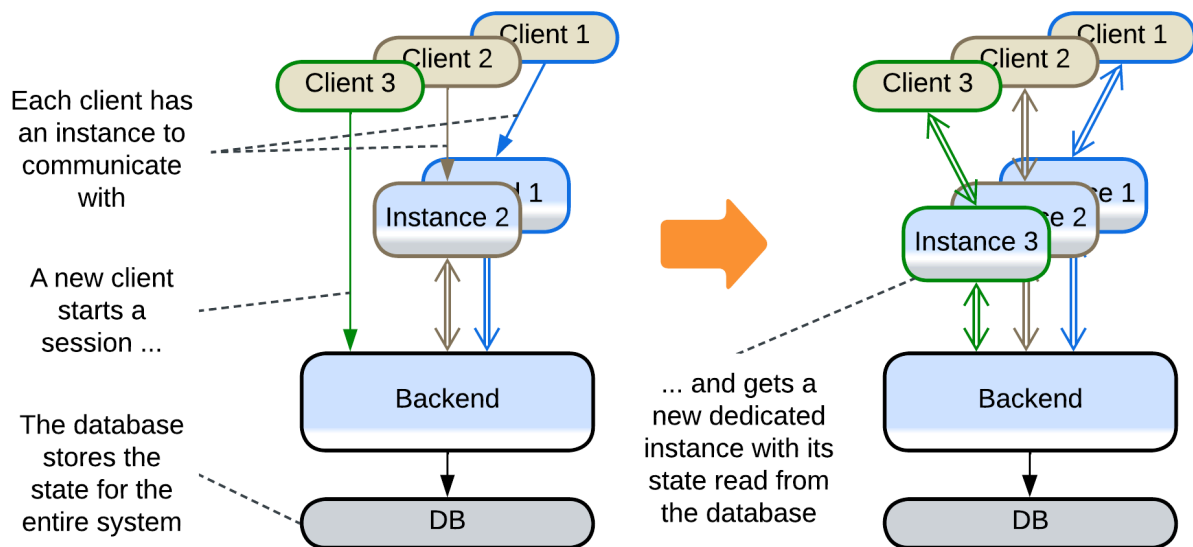
An instance is created for serving an incoming request and is destroyed when the request processing is finished. On creation it is initialized with all the client-related data to be able to interact with the client without much help from the backend. Examples include running web pages in the clients' browsers or client-dedicated actors in backends of instant messengers.

This provides perfect elasticity and flexibility of deployment at the cost of slower session establishment but usually relies on an external shared layer for persistence: Instances of a frontend are initialized from and send changes to a backend which itself uses a database.

Evolutions

There are two kinds of evolutions for *Shards*: those intrinsic to the component sharded and those specific to *Shards* pattern. All are summarized below while Appendix E provides more details on the second kind.

Evolutions of a sharded monolith

When *Shards* are applied to a single component, which is a monolith, the resulting (sub)system follows most of the evolutions of *Monolith*:

- *Layers* allow for the parts of the system to differ in *qualities* (*forces*) and deployment, 3rd party components to be integrated and the code to become better structured.
- *Services* or *Pipeline* help to distribute the work among teams and may decrease the project's complexity if the division results in loosely coupled components.
- *Plugins* and its subtypes, namely *Hexagonal Architecture* and *Scripts*, make the system more adaptable.
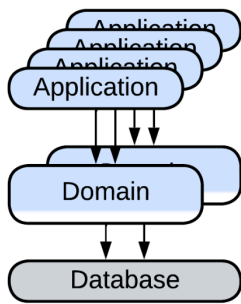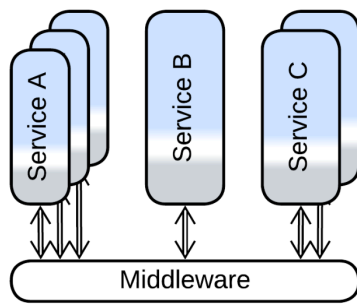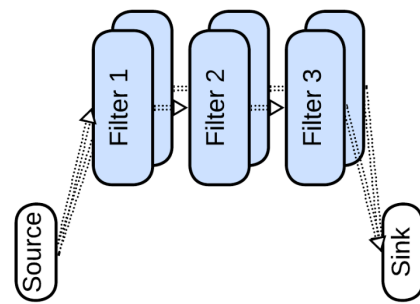
Layers **Services with a Middleware** **Pipeline**
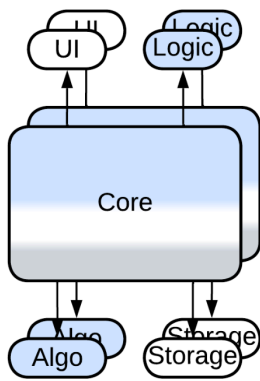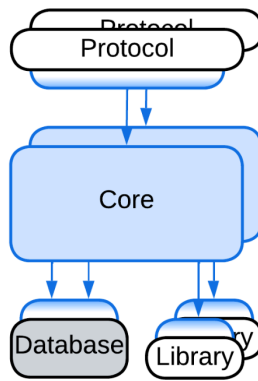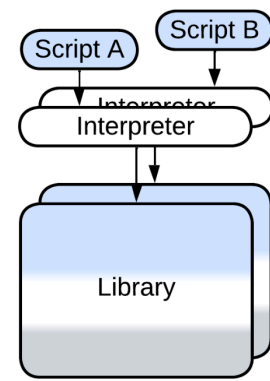
Plugins **Hexagonal Architecture** **Scripts**

There is a benefit of such transformations which is important in the context of *Shards*: in many cases the resulting modules can be scaled independently, arranging for a better resource utilization by the system (when compared to scaling a *monolith*). However, scaling individual modules often requires *load balancers* or a *middleware* to distribute internal requests among the scaled instances.

Evolutions that share data

The issue peculiar to *Shards* is that of coordinating the deployed instances, especially if their data becomes coupled. The most direct solution is to let the instances operate a shared data:

- If the whole dataset needs to be shared, it can be split to a *shared repository* layer.

- If data collisions are tolerated, *Space-Based Architecture* promises low latency and dynamic scalability.

The data grid provides any instance with fast access to all the data

The data is backed up to the persistent database

- If a part of the system's data becomes coupled, only that part can be moved to a *shared repository*, causing each instance to manage two stores of data: private and shared.

Each shard retains its identity and data

The coupled subset of data moves to a shared DB

- Another option is to split a *service* that owns the coupled data and is always deployed as a single instance. The remaining parts of the system become coupled to that service, not each other.

Each shard retains its identity and data

The coupled data and logic moves to a shared service

Evolutions that share logic

Other cases are better solved by extracting the logic that manipulates multiple *shards*:

- Splitting a *service* (as discussed above) yields a component that represents both shared data and shared logic.
- Adding a *middleware* lets the shards communicate to each other without keeping direct connections. It also may do housekeeping: error recovery, replication and scaling.

The middleware manages the shards and provides them with a means of communication

- A *load balancer* decouples clients from the knowledge about the existence of the shards.

The load balancer knows which shard has the client's data

- An *orchestrator* calls multiple shards to serve a user request. That relieves the shards of the need to coordinate their states and actions by themselves.

The orchestrator integrates information owned by multipls shards

Summary

Sharding makes any kind of component scalable and more fault tolerant. It does not change the nature of the component it applies to and usually relies on a *load balancer* or *middleware* to manage the multiple instances it generates. Its main weakness appears when the *shards* need to intercommunicate.

Layers
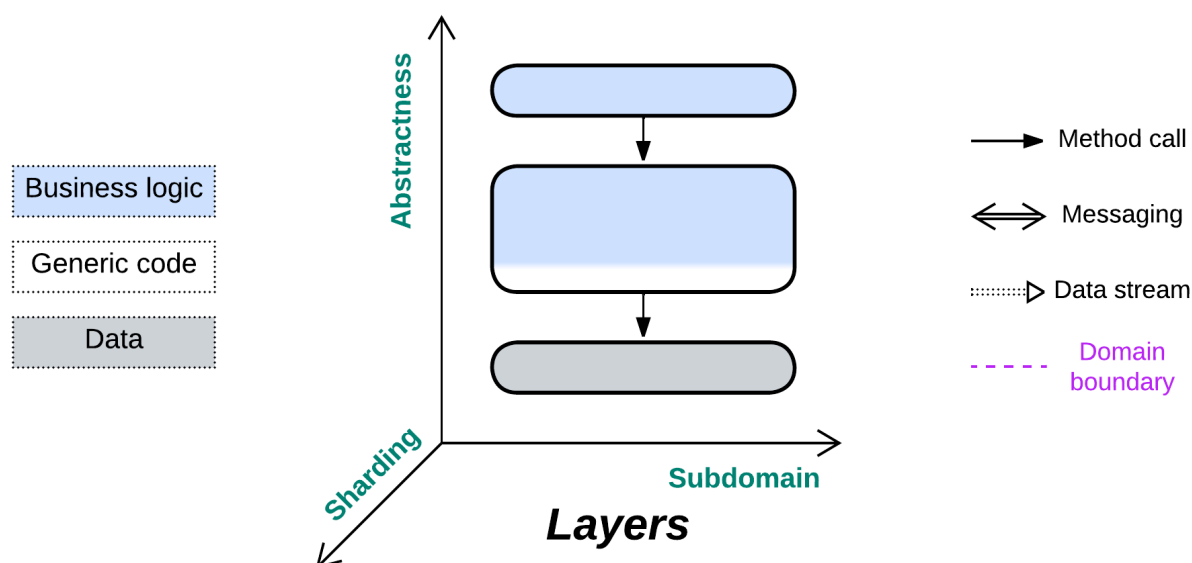
Layers

Yet another layer of indirection. Don't mix the business logic and implementation details.

Known as: Layers [POSA1, POSA4], Layered Architecture [SAP, FSA, LDDD], Multitier Architecture, N-tier Architecture [LDDD].

Variants: Co-located or distributed, synchronous or asynchronous communication, open or closed, the number of layers.

By isolation:
- Synchronous layers / Layered Monolith [FSA],
- Asynchronous layers,
- A process per layer,
- Distributed tiers.

Examples:
- Domain-Driven Design (DDD) Layers [DDD],
- Three-Tier Architecture,
- Embedded Systems.

Structure: A module per level of abstractness.

Type: Main, implementation.

Benefits	Drawbacks
Rapid start of development	Quickly deteriorates with project growth
Easy debugging	Hard to develop with more than few teams
Good performance	No solution for force conflicts between subdomains
Development teams may specialize	
Encapsulation of business logic	
Allows to resolve conflicting forces	
Deployment to a dedicated hardware	
Layers with no business logic are reusable	

References: [POSA1] and [FSA] provide in-depth discussions of layered applications; [DDD] promotes the layered style and most of the architectures in Herberto Graça's Software Architecture Chronicles are layered. The Wiki has a reasonably good article.

Layering a module creates interfaces between its various levels of abstractness (high-level logic, low-level logic, infrastructure) while retaining the monolithic cohesiveness inside each of the levels. That both allows for easy debugging inside each individual level (no need to jump into another programming language or re-attach the debugger to a remote server) and provides enough flexibility to assign development teams, tools, deployment and scaling policies on a per level basis. Though the code is slightly better than for *Monolith*, thanks to the separation of concerns, still one of the upper layers may grow too large for efficient development.

Splitting a system into layers resolves conflicts of forces between its abstract and optimized parts: the top-level business logic changes rapidly and does not require much optimization (as its methods are called infrequently), thus it can be written in a high-level programming language. Contrariwise, infrastructure, being called thousands of times every second, has stable workflows but must be highly optimized and extremely well tested.

Many patterns have one or more of their layers split by subdomain, resulting in a layer of *services*. That causes no penalties as long as the services are completely independent (the original layer had zero coupling between its subdomains), which happens if each of them deals with a separate subset of requests (as in *Backends for Frontends*) or is choreographed by an upper layer (as in *Polyglot Persistence*, *Hexagonal Architecture* or *Hierarchy*) – which boils down to the same "separate subset of subrequests" under the hood. However, if the services that form a layer need to intercommunicate, you immediately get the whole set of troubles with debugging, sharing data and performance.

The BFFs are completely independent for they handle distinct requests from distinct clients

Backends for Frontends   **Polyglot Persistence**

The databases are independent as they are reached through separate subrequests
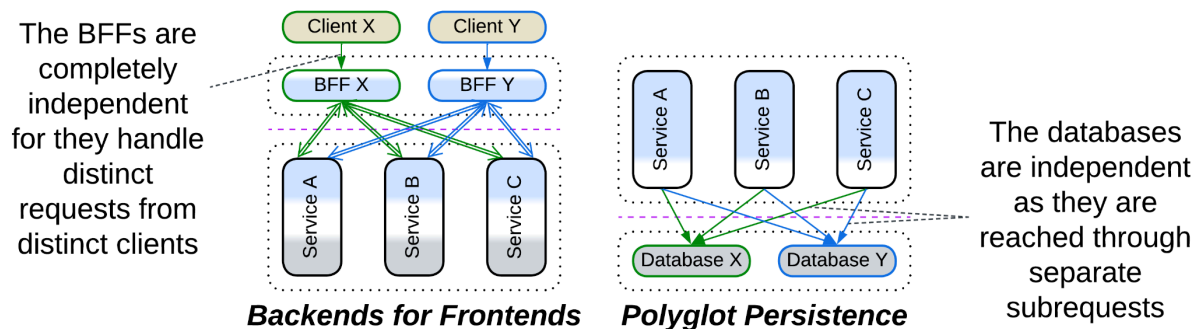
Performance

The performance of a layered system is shaped by two factors:
- Communication between layers is slower than that inside a layer. Components of a layer may access each other's data directly, while accessing another layer involves data transformation (as interfaces tend to operate generic data structures), serialization and often IPC or networking.
- The frequency and granularity of events or actions increases as we move from the upper more abstract layers to the low-level components that interface an OS or hardware.

That give rise to the following optimizations that decrease the number of interlayer calls:

Asking components about their state is slow

State of a low-level component

But we can remember and use the last known state of the components
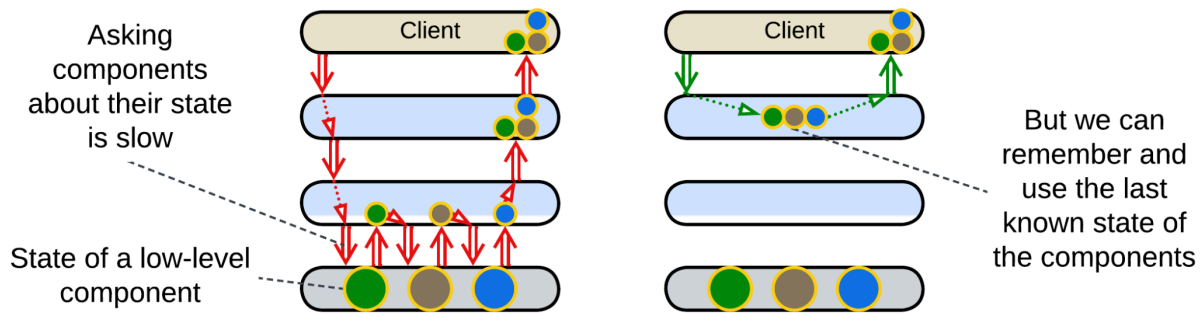
Caching: an upper layer tends to *model* (cache last known state of) the layers below it. This way it can behave as if it knew the state of the whole system without querying the actual state which is stored in the hardware below all the layers. Such an approach is universal with *control software* – the systems that operate physical devices. For example, a network monitoring suite shows you the last known state of all the components it observes without actually querying them – it is subscribed to notifications and remembers what each device has reported.

Notifyng high-level components of every event loads the system

However, we can aggregate multiple events into a single notification
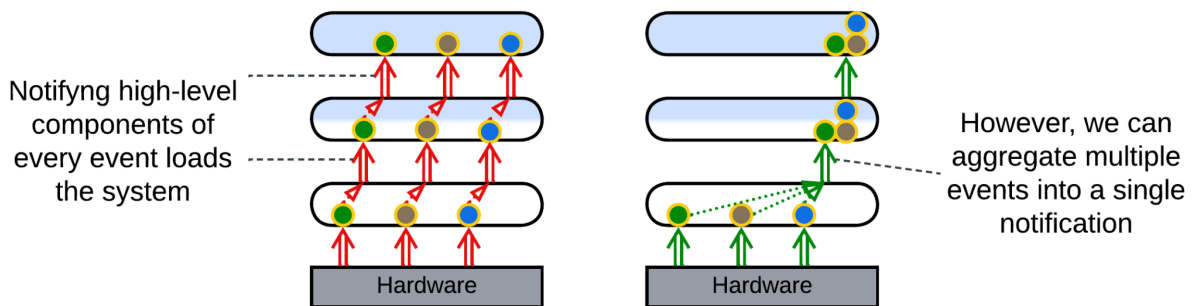
Aggregation: a lower layer collects multiple events before notifying the layer above it to avoid being overly chatty. An example would be an IIoT field gateway that collects data from all the sensors in the building and sends a single report to the server once in a while. Or take a data transfer over a network where a low-level driver collects multiple data packets coming from the hardware and sends an acknowledgement for each of them while waiting for a datagram or file transfer to complete. It notifies its user only once when all the data has been collected and its integrity confirmed.

Fine-grained commands are bad for performance

We can optimize by sending them as a batch
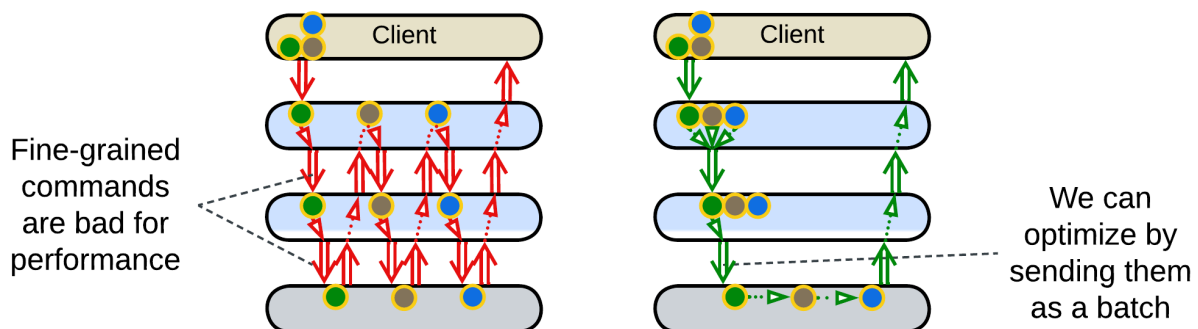
Batching: an upper layer forms a queue of commands and sends it as a single job to a layer below it. This takes place in drivers for complex low-level hardware, like printers, or in database access as stored procedures. [POSA4] describes the approach as *Combined Method*, *Enumeration Method* and *Batch Method* patterns. Programming languages and frameworks may implement *foreach* and *map/reduce* which allow for a single command to operate on multiple pieces of data.

Basic event filtering and processing

If your system is too slow to process events in time ...

... move an autonomous part of your event processing logic closer to the event source

Hardware

Hardware
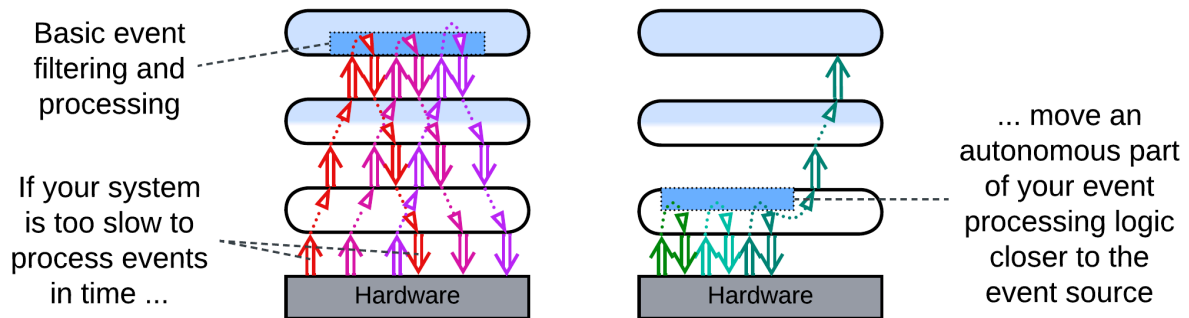
Strategy injection: an upper layer installs an event handler (hook) into the lower layer. The goal is for the hook to do basic pre-processing, filtering, aggregation and decision making to deal with the majority of events autonomously while escalating to the upper layer in exceptional or important cases. That may help in such time-critical domains as high-frequency trading.

Layers can be scaled independently, as exemplified by the common web applications that comprise a highly scalable and resource-consuming frontend, somewhat scalable backend and unscalable data layer, or by an OS (lower layer) that runs multiple user applications (upper layer).

Dependencies

Usually an upper layer depends on the *API* (application programming interface) of the layer directly below it. That makes sense as the lower the layer is, the more stable it tends to be: a user-facing software gets updated on a daily or weekly basis while system drivers may not change for years. As every update of a component may destabilize other components that depend on it, it is preferable for a quickly evolving component to depend on others, not be dependent upon.

Some domains, such as hardware or telecom, require polymorphism of their lower layers. In that case an upper layer (e.g. OS kernel) defines an *SPI* (service provider interface) which is implemented by every variant of the lower layer (e.g. a device driver). That allows for a single implementation of the upper layer to be interoperable with any subclass of the lower layer. Such an approach engines *Plugins*, *Microkernel* and *Hexagonal Architecture*.

There may also be an adapter layer between your system's SPI and an external API which is called *Anticorruption Layer* [DDD], *Database Abstraction Layer* / *Database Access Layer* [POSA4] / *Data Mapper* [PEAA], *OS Abstraction Layer* or *Hardware Abstraction Layer*, depending on what it adapts.

Application	Cloud Application	Application
APIs		SPI
Domain	Field Gateway	Adapter
SPIs		API
Infrastructure	IoT Devices	Library

Dependency on APIs · **Dependency on SPIs** · **Abstraction Layer**
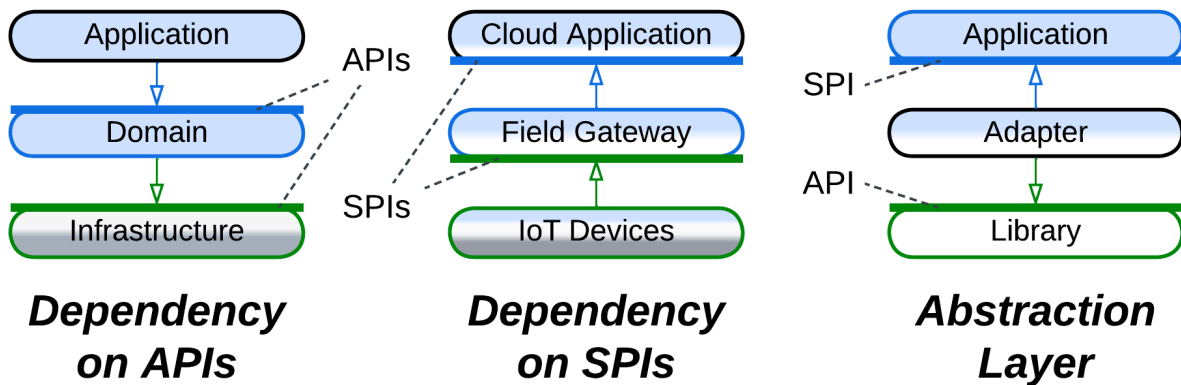
A layer can be *closed* (*strict*) or *open* (*relaxed*). A layer above a closed layer depends only on the closed layer right below it – it does not see through it. Contrariwise, a layer above an open layer depends on both the open layer and the layer below it. The open layer is transparent. That helps smaller layers that encapsulate one or two subdomains: if such a layer were closed, it would have to copy much of the interface of the layer below it just to forward there the incoming requests which it does not handle. The optimization of the open layer has its cost: the team that works on the layer above an open layer needs to learn two APIs, which may have incompatible terminology.

With a closed layer the application code always uses ORM

Application	Application
ORM ∇ API	ORM ∇ API
ORM	ORM
∇ SQL	SQL
Database	Database

Closed Layer · **Open Layer**

If the ORM is an open layer, the application may bypass it
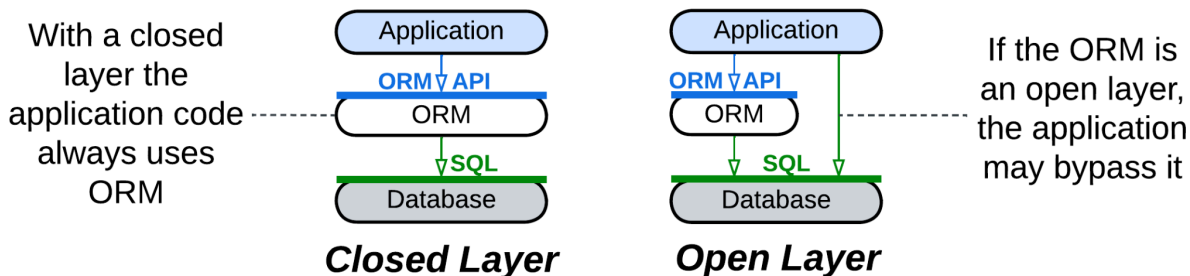
If you ever need to *scale* (run multiple instances of) a layer, you may notice that a layer which sends requests naturally supports multiple instances, with the instance address being appended to each request so that its destination layer knows where to send the response. On the other hand, if you need multiple instances of an interior layer, you'll need a *load balancer* to dispatch requests among its instances.

Frontend
Frontend
Frontend

Load Balancer

Backend

Database

When you access a layer that has many instances, you need a load balancer in between

Multiple instances of a layer can access a single instance directly
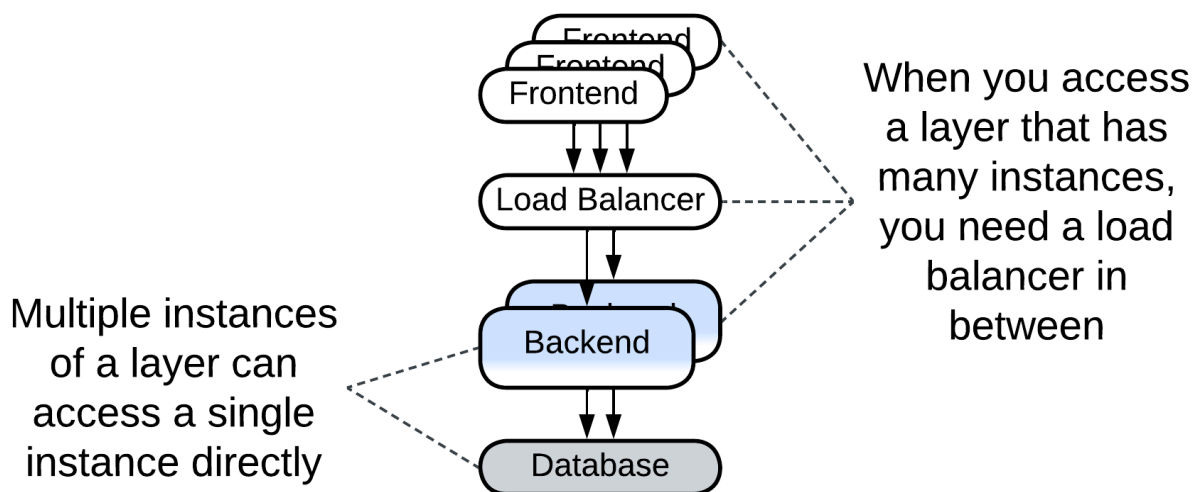
Applicability

Layers are <u>good</u> for:
- *Small and medium-sized projects.* Separating the business logic from the low-level code should be enough to work comfortably on codebases below 100 000 lines in size.
- *Specialized teams*. You can have a team per layer: some people, who are proficient in optimization, work on the high load infrastructure, while others talk to the customers and write the business logic.
- *Deployment over a specific hardware.* The frontend instances run on the client devices, the backend needs RAM, the data layer needs HDD and security. There is no way to unite them into a single generic module.
- *Flexible scaling.* It is common to have hundreds or thousands of frontend instances being served by several backend processes that use a single database.
- *Real-time systems*. Hardware components and network events often need the software to respond quickly. This is achievable by splitting the time-critical code from normal priority calculations. See *Hexagonal Architecture* and *Microkernel* for improved solutions.

Layers are <u>bad</u> for:
- *Large projects.* You are still going to enter the monolithic hell if you reach 1 000 000 lines of code.
- *Low-latency decision making*. If your business logic needs to be applied in real time, you cannot tolerate the extra latency caused by the interlayer communication.

Relations

It is possible to group independent components into a layer

Layers can be applied to internals of a component

However, the grouping in itself does not improve intralayer communication
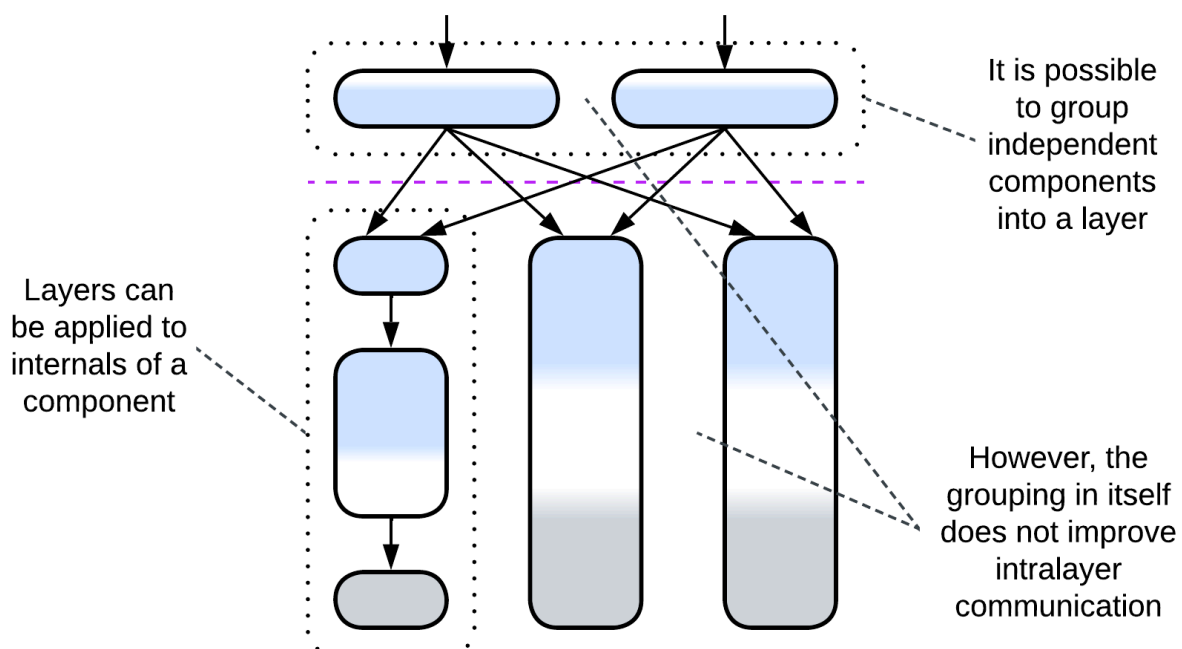
Layers:
- Can be applied to the internals of any module, for example, layering *Services* results in *Layered Services*.
- Can be altered by *Plugins* or extended with an *orchestrator*, *proxy* and/or *shared repository* to form an extra layer.

- Can be implemented by *Services* resulting in layers of services in *Service-Oriented Architecture*, *Backends for Frontends* or *Polyglot Persistence*.

Variants by isolation

There are several grades of layer isolation between unstructured *Monolith* and distributed *Tiers*. All of them are widely used in practice: each step adds its specific benefits and drawbacks to those of the previous stages, and at some point it makes sense to reject the next deal.

Synchronous layers, Layered Monolith

First you separate the high-level logic from low-level implementation details. You draw interfaces between them. The layers still call each other directly, but at least the code has been sorted out into some kind of structure, and you can now employ two or three dedicated teams, one per layer. The cost is quite low – the interfaces stand in the way of aggressive performance optimizations.

Benefits	Drawbacks
Structured code	Lost opportunities for optimization
Two or three teams	

Asynchronous layers

The next step you may decide to take isolates the layers' execution threads and data. The layers will communicate only with in-process messages, which is slower than direct calls, and is harder to debug, but now each layer can run at its own pace – which is a must for interactive systems.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Some troubles with debugging
The layers may differ in latency	

A process per layer

Next, you may run each layer as a separate process. You'll have to devise an efficient means of communication between them, but now the layers may differ in technologies, security, frequency of deployment and even stability – a crash of one layer does not affect the others. Moreover, you may scale a layer to saturate the available CPU cores. You pay with even harder debugging, lower performance and you'll have to think of error recovery, as if one of the components crashes, then others are likely to remain in an inconsistent state.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Troublesome debugging
The layers may differ in latency	Some performance penalty
The layers may differ in technologies	Need error recovery
The layers are deployed independently	
Software security isolation	
Software fault isolation	
Limited scalability	

Distributed tiers

Finally, you may separate the hardware the processes run on – going all out for distribution. That allows for fine-tuning system configuration, running parts of the system close to its clients, physically isolating the most secure components and scalability limited only by your budget. The price is paid in performance and debugging experience.

Benefits	Drawbacks
Structured code	No opportunities for optimization
Two or three teams	Even worse debugging
The layers may differ in latency	Definite performance penalty
The layers may differ in technologies	Need error recovery
The layers are deployed independently	
Full security isolation	
Full fault isolation	
Full scalability	
Layers vary in hardware setup	
Deployment close to clients	

Examples

The notion of layering seems to be so natural to our minds that most of the known architectures are layered. Not surprisingly, there are multiple approaches to assigning functionality to and naming the layers:

Domain-Driven Design (DDD) Layers

User interface (frontend)	**Presentation**	Interacts with the user. Latency is crucial.
Scenarios or scripts	**Application**	Drives use cases. Must be easy to write and deploy.
Classes that do the job	**Domain**	Implements complex behavior. Needs stability and speed.
Generic libraries and databases	**Infrastructure**	Tools that we bought and wrote. Should be fast and well-tested.
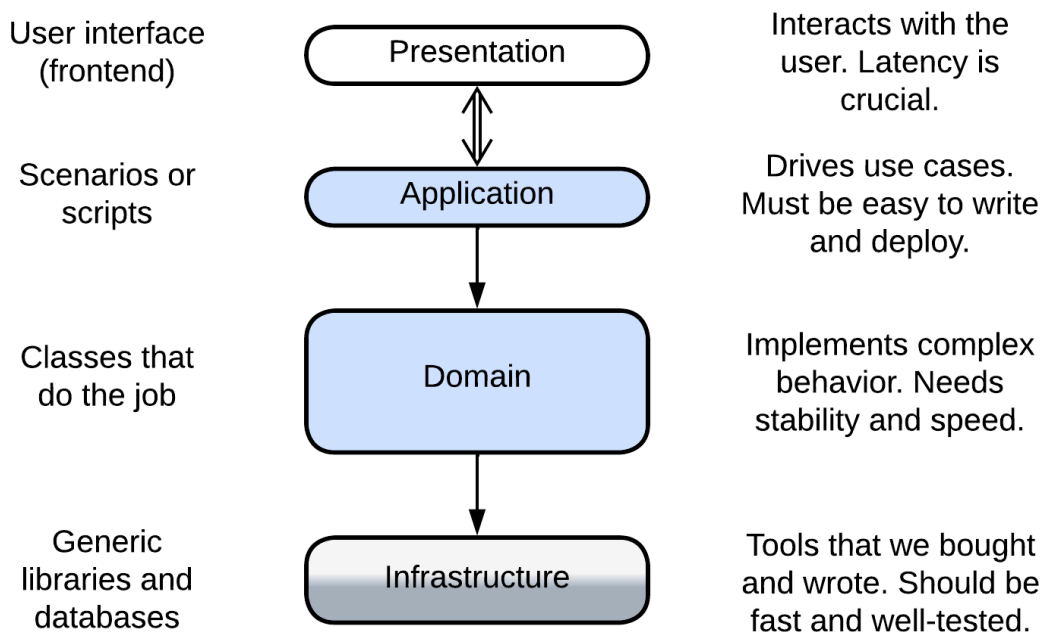
[DDD] recognizes four layers with the upper layers closer to the user:
- Presentation (User Interface) – the user-facing component (frontend, UI). It should be highly responsive to the user's input.
- Application (Integration, Service) – the high-level scenarios that build up on the API of the domain layer. It should be easy to change and deploy. See _Orchestrator_.

- Domain (Model, Business Logic) – the bulk of the mid- and low-level business logic. It should usually be well-tested and performant.
- Infrastructure (Utility, Data Access) – the utility components devoid of business logic. Their stability and performance is business-critical but updates are rare.

For example, in an online banking system comprises:
- the presentation layer is the frontend;
- the application layer knows the sequences of steps for payment, card to card transfer and viewing the client's history of transactions;
- the domain layer contains the classes for various kinds of cards and accounts;
- the infrastructure layer with the database and libraries for encryption and interbank communication.

We will often use this naming convention while describing more complex architectures, some of which have all the four layers, while others merge several layers together or omit the frontend and concentrate on the components that contain the business logic.

In practice you are much more likely to meet the derived *DDD-style Hexagonal Architecture* than the original *DDD Layers*.

Three-Tier Architecture
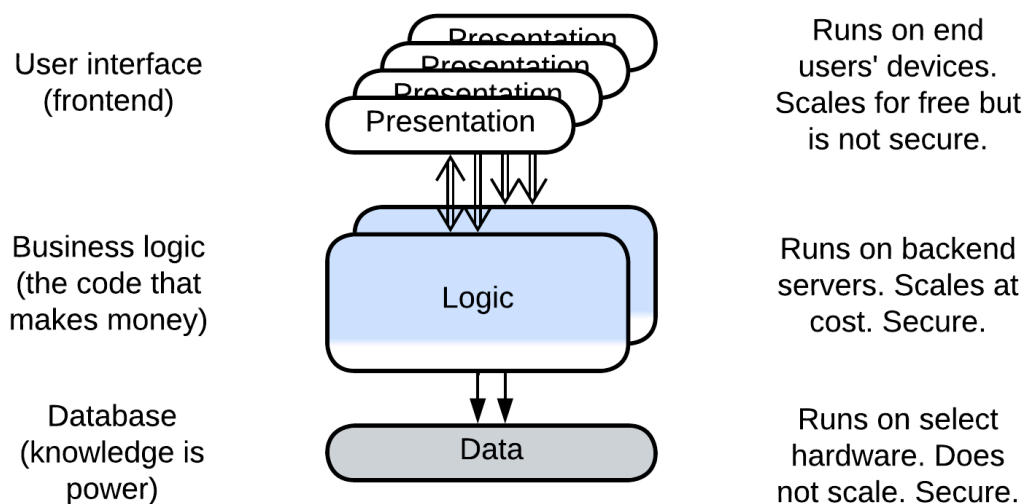
Here the focus lies with distribution of the components over heterogeneous hardware (*tiers*):
- Presentation (Frontend) tier – the user-facing application that runs on a user's hardware. It is very scalable, responsive but insecure.
- Logic (Backend) tier – the business logic which is deployed on the service provider's side. Its scalability is limited mostly by the funding committed, security is good and latency is high.
- Data (Database) tier – the service provider's database that runs on a dedicated server. It is not scalable but is very secure.

In this case the division into layers resolves the conflict between scalability, latency, security and cost as discussed in detail in the chapter on distribution.

As we can see, the tiers don't directly map to the DDD layers: for example, encryption libraries belong to the (lowest) infrastructure layer of DDD but to the (middle) application tier.

Embedded systems

User interface and behavior of the device	**UI** / **HMI**	Implemented by the device manufacturer. Updated frequently.
Drivers, standards, protocols	**SDK** / **HAL**	Comes from the OEM. Updated from time to time.
The black box	**FW** / **HW**	Comes from component suppliers. Updates are very rare.
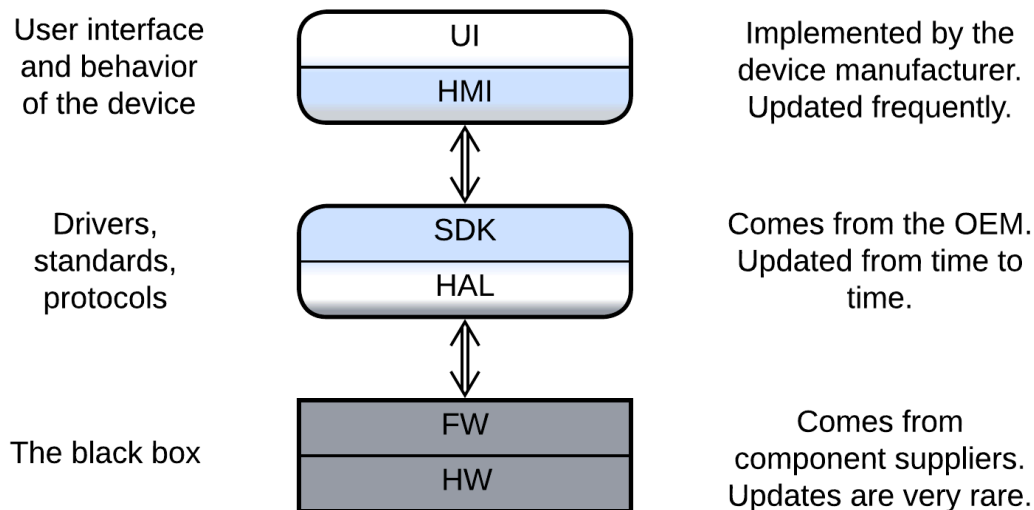
Bare metal and micro-OS systems that run on low-end chips use a different terminology, which is not unified among domains. A generic example involves:

- Presentation – the UI engine used by the HMI. It may be a 3rd party library or come as a part of the SDK.
- Human-Machine Interface (HMI aka MMI) – the UI and high-level business logic for user scenarios, written by a value-added reseller.
- Software Development Kit (SDK) – the mid-level business logic and device drivers, written by the original equipment manufacturer.
- Hardware Abstraction Layer (HAL) – the low-level code that abstracts the hardware interface to enable code reuse between hardware platforms.
- Firmware of Hardware Components – usually closed-source binary pre-programmed into chips by chipmakers.
- Hardware.

It is of note that in this approach the layers form strongly coupled pairs. Each pair is implemented by a separate party of the supply chain, which is an extra force that shapes the system into layers.

An example of such a system can be found in an old phone or digital camera.

Evolutions

Layers are not without drawbacks, which may cause the system to evolve. A summary of such evolutions is below while the details are reserved for Appendix E.

Evolutions that make more layers

Not all the layered architectures are equally layered. A *monolith* with a *proxy* or database has already stepped into the realm of *Layers* but is far away from reaping all its benefits. Such a kind of system may continue its journey in a few ways that were earlier discussed for *Monolith*:

- Employing a *database* (if you don't use one) lets you rely on a thoroughly optimized state-of-the-art subsystem for data processing and storage.
- *Proxies* are similarly reusable generic modules to be added at will.

- Implementing an *orchestrator* on top of your system may improve programming experience and runtime performance for your clients.

A proxy represents the system to its clients and takes care of one or more aspects of the communication

An orchestrator turns high-level requests into series of low-level steps

The domain layer implements business rules

The database provides persistence and analytics

It is also common to:
- Have the business logic divided in two layers.
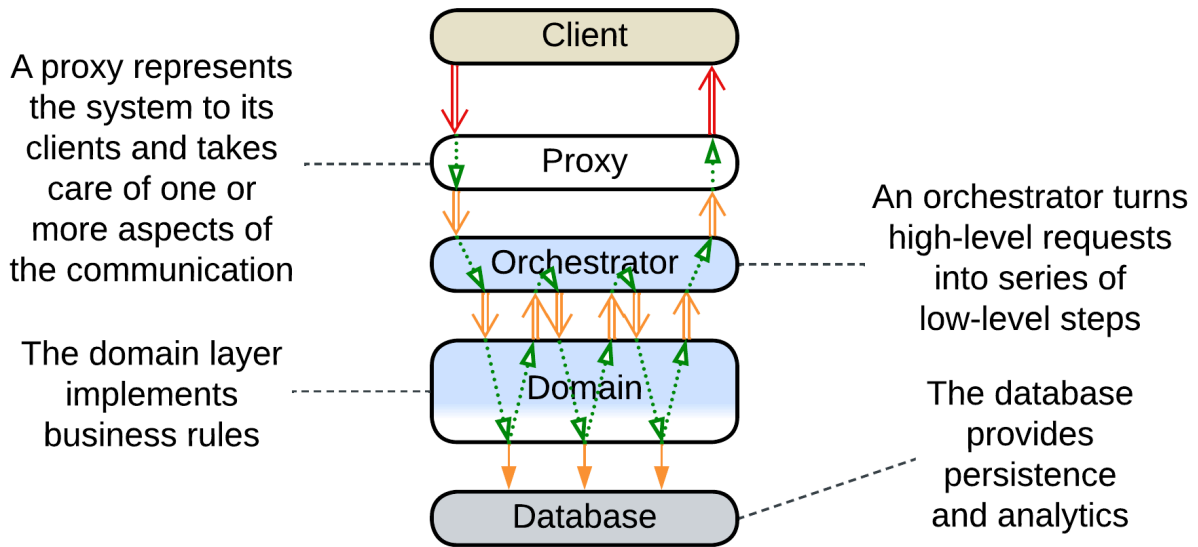
Separating the high-level and low-level logic lets them vary in technologies and qualities

Evolutions that help large projects

The main drawback (and benefit as well) of *Layers* is that much or all of the business logic is kept together in one or two components. That allows for easy debugging and fast development in the initial stages of the project but slows down and complicates work as the project grows in size. The only way for a growing project to survive and continue evolving at a reasonable speed is to divide its business logic into several smaller, thus less complex modules that match subdomains (bounded contexts) [DDD]. There are several options for such a change, with their applicability depending on the domain:
- The middle layer with the main business logic can be divided into *services* leaving the upper *orchestrator* and lower *database* layers intact for future evolutions.

The domain layer, the largest component, is usually easy to split by subdomains

- Sometimes the business logic can be represented as a set of directed graphs which is known as *Event-Driven Architecture*.

Each kind of request passes through a dedicated pipeline

- If you are lucky, your domain makes a *Top-Down Hierarchy*.

Hierarchy distributes decision-making over multiple layers and subdomains
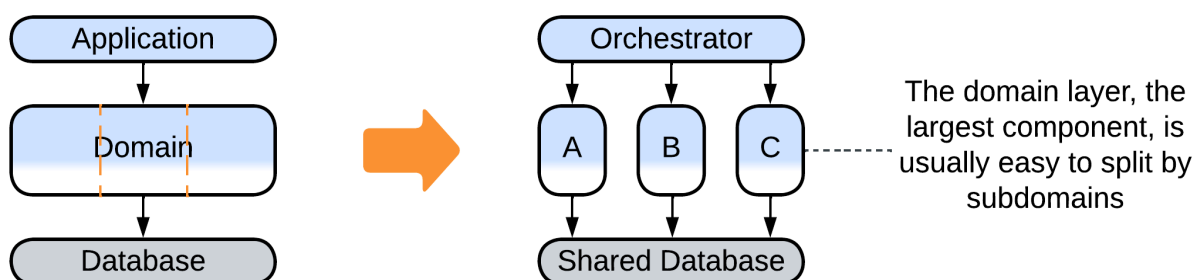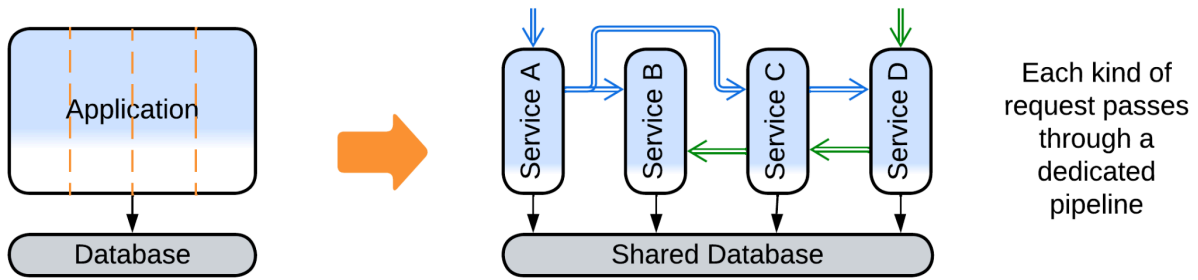
Evolutions that improve performance

There are several ways to improve performance of a layered system. One we have already discussed for *Shards*:

- *Space-Based Architecture* co-locates the database and business logic and scales both dynamically.

The data grid is a highly-scalable in-memory database

The data is backed up to the persistent database

Others are new:
- Merging several layers improves latency by eliminating the communication overhead.

A monolithic application provides more opportunities for optimization

- Scaling some of the layers may improve throughput.

Layers can be sharded independently of each other

- *Polyglot Persistence* is the pattern for using multiple specialized databases.

Each kind of use case or data type may call for a dedicated database
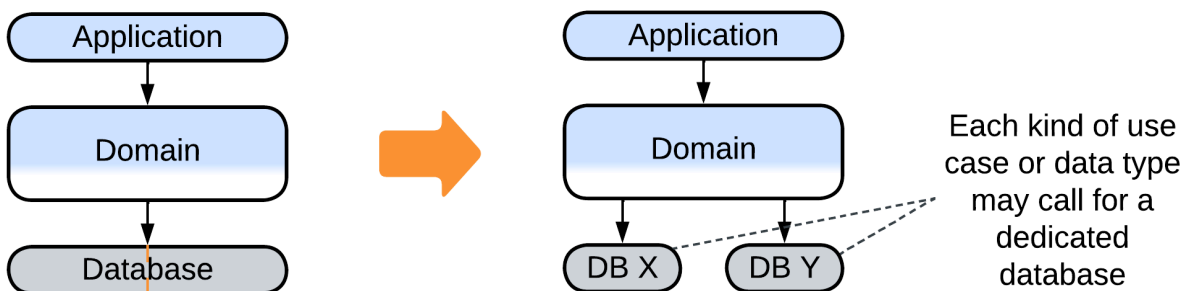
Evolutions to gain flexibility

The last group of evolutions to consider is about making the system more adaptable. We have already discussed the following evolutions for *Monolith*:

- The behavior of the system may be modified with *plugins*.
- *Hexagonal Architecture* allows for abstracting the business logic from the technologies used on the project.
- *Scripts* allow for customization of the system's logic on a per client basis.

Layers with Plugins

Layers with Scripts

Layered Hexagonal Architecture

There is also a new evolution that modifies the upper (orchestration) layer:

- The orchestration layer may be split into *backends for frontends* to match the needs of several kinds of clients.

Each kind of a client may have a dedicated application
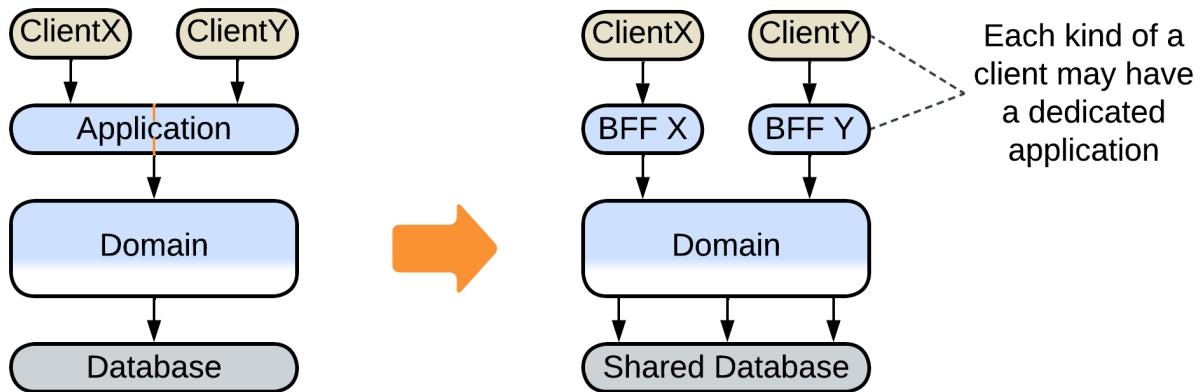
Summary

Layered architecture is superior for medium-sized projects. It supports rapid development by two or three teams, is flexible enough to resolve conflicting forces and provides many options for further evolution, which will likely be needed when the project grows in size and complexity.

Services
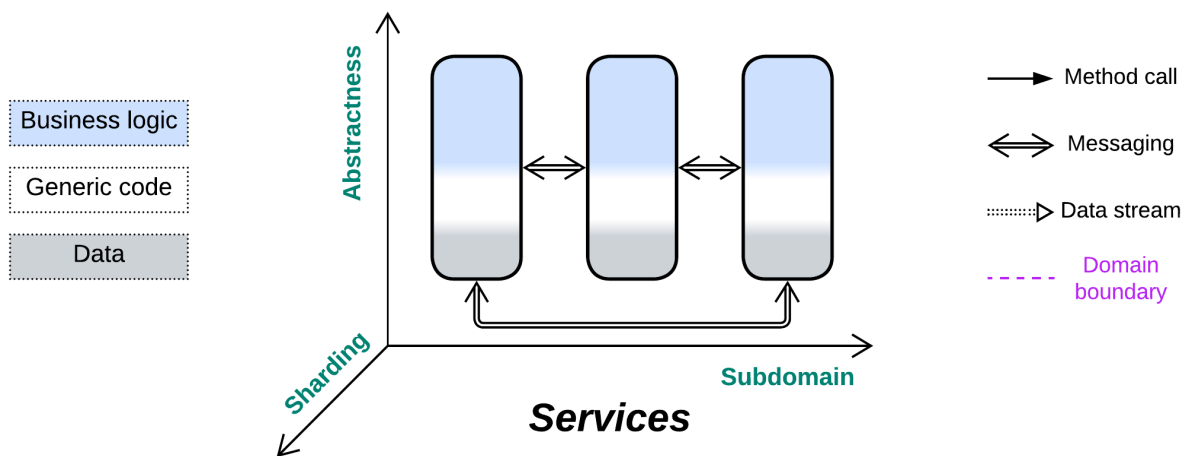
Services

Divide and conquer. Gain flexibility through decoupling subdomains.

Known as: Services, Domain Services [FSA and SAHP, but not DDD].

Variants: *Pipeline* got a dedicated chapter. Many modifications are listed under Evolution.

By isolation:
- Synchronous modules: Modular Monolith [FSA] (Modulith),
- Asynchronous modules: Modular Monolith (Modulith) / Embedded Actors,
- Multiple processes,
- Distributed runtime: Function as a Service (FaaS) (including Nanoservices) / Backend Actors,
- Distributed services: Service-Based Architecture [FSA] / Space-Based Architecture [FSA] / Microservices.

By communication:
- Direct method calls,
- RPCs and commands (request/confirm pairs),
- Notifications (pub/sub) and shared data,
- (inexact) No communication.

By size:
- Whole subdomain: Domain Services [FSA],
- Part of a subdomain: Microservices,
- Class-like: Actors,
- Single function: FaaS [DDS] / Nanoservices.

By internal structure:
- Monolithic service,
- Layered service,
- Hexagonal service,
- Scaled service,
- Cell (WSO2 definition) (service of services).

Examples:
- Service-Based Architecture [FSA],
- Microservices [MP, FSA],
- Actors,
- (inexact) Nanoservices (API layer),

- (inexact) Device Drivers.

Structure: A module per subdomain.

Type: Main.

Benefits	Drawbacks
Supports large codebases	Global use cases are hard to debug
Multiple development teams and technologies	Poor latency in global use cases
Forces may vary by subdomain	No good way to share state between services
	The domain structure should never change
	Operational complexity

References: [FSA] has a chapter on Service-Based Architecture; [MP] is dedicated to Microservices.

Splitting a monolith by *subdomain* allows for mostly independent development, deployment and properties of the resulting components. However, the subdomains must be loosely coupled, and if they are of comparable size, the partitioning can greatly reduce the total complexity of the project's code by cutting accidental dependencies between the subdomains. Moreover, if one of the services grows to an unmanageable size, it can often be further partitioned by sub-subdomains to form a *cell*. This flexibility is paid for by the complexity and performance of use cases that need collaboration from multiple subdomains. Another issue to remember is that the boundaries between services are nearly impossible to move at later project stages, as the services grow to rely on different technologies and implementation styles, thus this metapattern assumes perfect practical knowledge of the domain and relatively stable requirements.

Performance

Interservice communication is relatively slow and resource-consuming, therefore it should be kept to a minimum.

In the best case a single service processes an incoming request

The processing is slower if several services are involved in the chain

In the worst case there is a need to synchronize states of multiple services
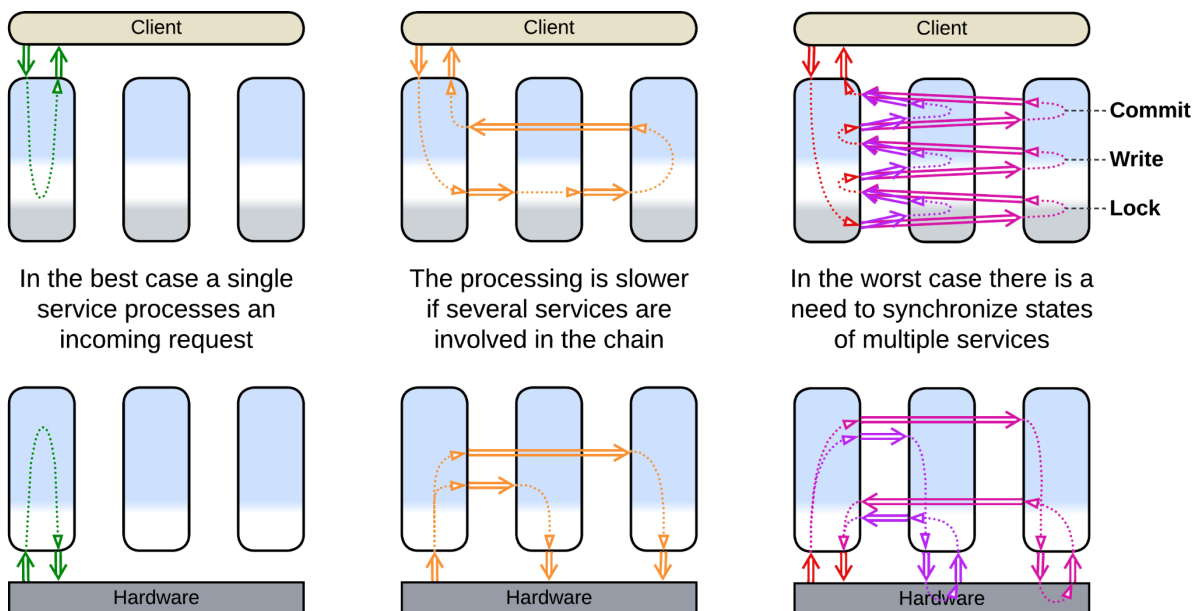
The perfect case is when a single service has enough authority to answer a client's request or process an event. That case should not be that rare as a service covers a whole subdomain while subdomains are expected to be loosely coupled (by definition).

Worse is the one when an event starts a chain reaction over a system, likely looping back a response to the original service or changing the target state of another controlled subsystem.

In the slowest scenario a service needs to synchronize its state with multiple other services, usually via locks and distributed transactions.

Multiple instances of an individual service may be deployed to improve throughput of the system. However, such a case will likely need a *middleware* or *load balancer* to distribute interservice requests among the instances and a *shared repository* to store and synchronize any non-shardable (accessed by several instances) state.

Dependencies

When we see a service to *request* help from other services and receive the results (*confirmation*), that service *orchestrates* the services it uses. Services often orchestrate each other because the subdomain a service is dedicated to is not independent of other subdomains. A service depends on any services that it uses.

A service depends on whatever it uses
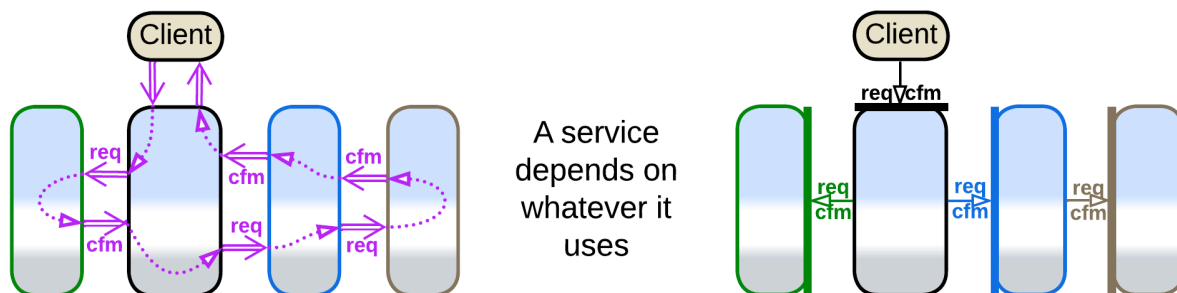
Another way for services to communicate is *choreography* – when a service sends a *command* or publishes a *notification* and does not expect any response. This is characteristic of *pipelines*, covered in the next chapter. Right now we should note that orchestration and choreography may be intermixed, in which case a service depends on all the services it uses or subscribes to.

A service depends on sources of notifications
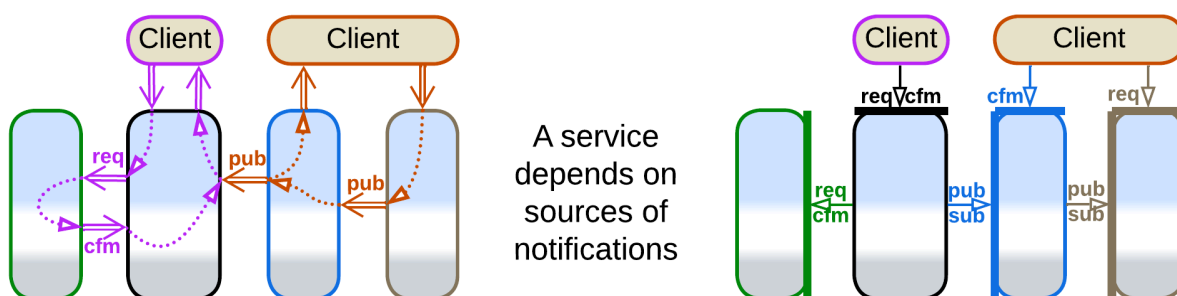
If the system relies on notifications (services publish *domain events*), it is possible to avoid interservice queries (pairs of a read request and confirmation with the data retrieved) by storing the data received in notifications to *CQRS (or materialized) views* [MP], which can reside *in memory* or in a database. The views can be planted inside every service that needs data which belongs to other services or can make a dedicated *query service* [MP]. Though the main goal of materialized views is to resolve distributed joins, they also remove dependencies in the code of services and optimize out interservice queries, simplifying APIs and improving performance. Further examples will be discussed in the *chapter on Polyglot Persistence*.

The CQRS view encapsulates the dependency
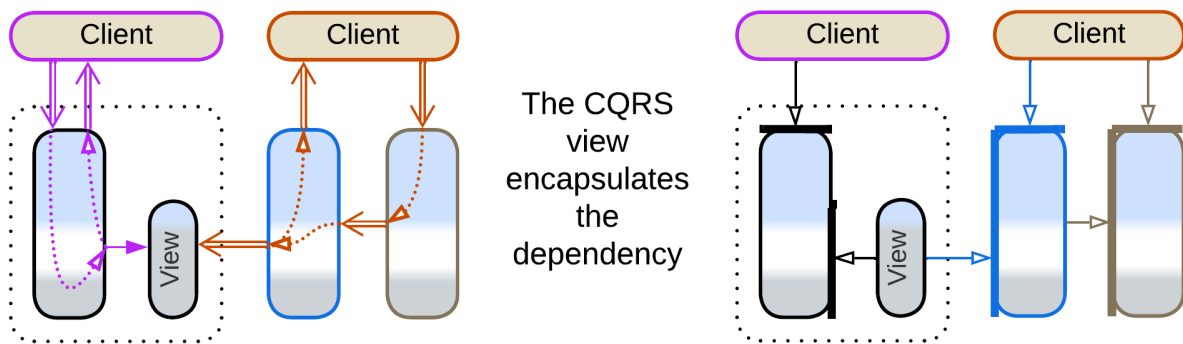
In general, a large service should wrap its dependencies with an *anticorruption layer* [DDD], following *Hexagonal Architecture*. The layer consists of adapters between the internal domain model of the service and the APIs of the components it uses, which isolate the business logic from the external environment, granting that no change in the interface of an external service or library may ever cause much work to the team that writes our business logic.

Adapters encapsulate dependencies
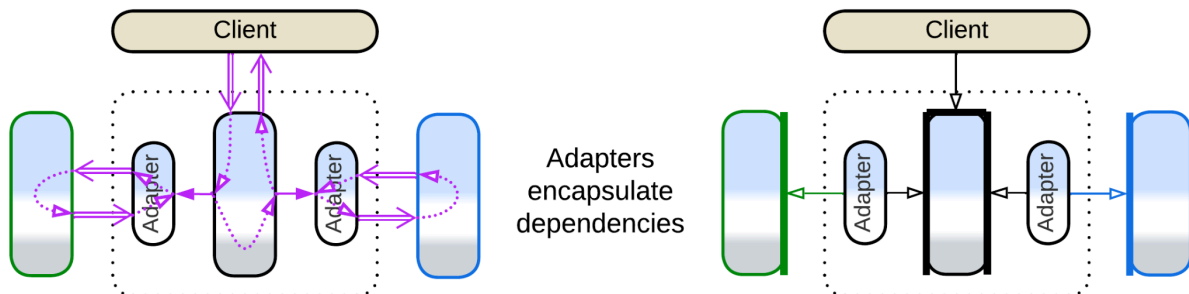
Applicability

Services are good for:
- *Large projects.* With multiple services independently developed, the project may grow well above 1 000 000 lines of code and still be comfortable to work on as no single service grows unreasonably large or complex.
- *Specialized teams.* Each service would often be developed by a dedicated team that invests its time in learning its subdomain. This way none needs to have a detailed knowledge of the full set of requirements, which is important for large domains.
- *Varied forces.* In system and embedded programming multiple components of wildly varying behaviors need to be managed. Each of them is controlled by a dedicated service (called "driver") which adapts to the specifics of the managed subsystem.
- *Flexible scaling.* Some services may be under more load than others. It makes sense to deploy multiple instances of heavily loaded services.

Services should be avoided for:
- *Cohesive domains.* If everything strongly depends on everything, any attempt to cut the knot with interfaces is going to make things worse. Unless the project is already dying because of its huge codebase, in which case you have nothing to lose.
- *Unfamiliar domains.* If you don't understand the intricacies of the system you are going to build, you may misalign the interfaces, and when the mistake will come to light, the architecture will be too hard to change [LDDD]. Coupled services may be worse than a monolith.

- *Quick start*. It takes effort to design good interfaces and contracts for the services, and managing multiple deployment units is not free of trouble. Debugging will be an issue.
- *Low latency*. If the system as a whole should react to events in real time, services should be avoided. However, individual services can provide low latency for local use cases (when a single service has enough authority to react to the incoming event).

Relations

A system component can be split into services
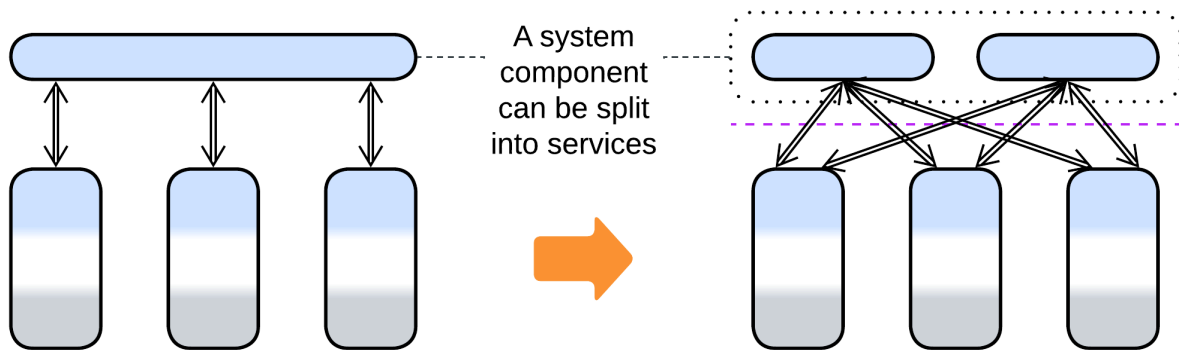
- The division by subdomain can be applied to *Layers* to form *Service-Oriented Architecture* (layers of services); to *Proxy*, *Orchestrator* or *API Gateway* to make *Backends for Frontends*; to a (shared) database to yield *Polyglot Persistence*.
- Services can be extended with a *proxy*, *orchestrator*, *middleware* or *shared repository*.
- Each service can be implemented by *layers* (resulting in a *layered service*), *hexagonal architecture* or a *cell*.

Variants by isolation

Division by subdomain is so commonplace and varied that no universal terminology emerged over the years. Below is my summary, in no way complete, of several dimensions of freedom for such systems. Each section lists well-known architectures it applies to.

First and foremost, there are multiple grades between cohesive *Monolith* and distributed *Services*. You should choose where to stop as the benefits of the next steps (color-coded below) may not outweigh their drawbacks for your project. *I list the most common options while a few more esoteric architectures can be found in Volodymyr Pavlyshyn's overview*.

Synchronous modules: Modular Monolith (Modulith)

The first step to take when designing a large project is division of the codebase into loosely coupled modules that match subdomains (*bounded contexts* [DDD]). If successful, that opens up development by a team per module. The entire application still runs as a single process, thus it is easy to debug, the modules can share data, and any crash kills the whole system. You pay by drawing the boundaries which will be hard to move in the future.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen

Asynchronous modules: Modular Monolith (Modulith), Embedded Actors

The next step is separating the modules' execution threads and data. Each module becomes a kind of *actor* that communicates with other components through messaging. Now your modules don't block each other's execution and you can replay events at the cost of nightmarish debugging and no clean way to share data between or synchronize the state of the components.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Some independence of module qualities	Hard to debug

Multiple processes

There is also an option of running the system components as separate binaries, which lets the components vary in technologies, allows for granular updates and addresses stability (a web browser does not stop when one of its tabs crashes) but adds the whole dimension of error recovery and half-committed transactions.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Independence of component qualities and technologies	Hard to debug
Single-component updates	Needs error recovery routines
Software fault isolation	Data inconsistencies after partial crashes
Limited granular scalability	

Distributed runtime: Function as a Service (FaaS) (including Nanoservices), Backend Actors

Modern distributed runtimes create a virtual namespace that may be deployed on a single machine or over a network. They may redistribute the running components over servers in a way to minimize network communication and may offer distributed debugging. With actors, if one of them crashes, that generates a message to another actor which may decide on how to handle the error. The convenience of using a runtime has the dark side of vendor lock-in.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No good way to share data or synchronize state
Independence of component qualities ~~and technologies~~	Hard to debug
Single-component updates	Needs error recovery routines
Full fault isolation	~~Data inconsistencies after partial crashes~~
Full dynamic granular scalability	Vendor lock-in

Moderate communication overhead
Moderate performance overhead caused by
the framework

Distributed services: Service-Based Architecture, Space-Based Architecture, Microservices

Fully autonomous services run on dedicated servers or virtual machines. This way you employ resources of multiple servers, but the communication between them is both unstable (leading to several kinds of artifacts [MP]) and slow, and debugging tends to be very hard. *Mesh*-based (*Microservices* and *Space-Based*) architectures provide dynamic scaling under load.

Benefits	Drawbacks
Multi-team development	Subdomain boundaries are frozen
Event replay	No ~~good~~ way to share data or synchronize state
Independence of component qualities and technologies	Very hard to debug
Single-component updates	Needs error recovery routines
Full fault isolation	Data inconsistencies after partial crashes
Full dynamic granular scalability	High communication overhead

Variants by communication

Services also differ in the way they communicate, which influences some of their properties:

Direct method calls

When components run in the same process and share execution threads, one component can call others. This is blazingly fast and efficient, but you should take care to protect the module's state from simultaneous access by multiple threads (and yes, deadlocks do happen). Moreover, it is hard to know what the module which you call is going to call in its turn – thus no matter how much you optimize your code, its performance depends on that of other components, often in subtle ways.

RPCs and commands (request/confirm pairs)

If a service calls into another service or requests it to act and return results (this is how method calls are implemented in distributed systems) it has to store the state of the scenario it is executing for the duration of the call (till the confirmation message is received). That takes resources: the stored state is kept in RAM, and the interruption and resumption of the execution wastes CPU cycles on context switch and the resulting cache misses. Blocked threads are especially heavy, while coroutines or fibers are more lightweight but still not free.

Another trouble for distributed systems comes about with error recovery: if your component did not receive a timely response, you don't know if your request was executed by its target – and you need to be really careful about possible data corruption if it is executed twice [MP].

On the bright side, orchestration tends to be human- and debugger-friendly as it keeps consecutive actions together in the code. Thus, synchronous interaction is the default mode of communication in many projects.

Notifications (pub/sub) and shared data

A service may do something, publish a notification or write results to a shared datastore for other services to process, and forget about the task it has completed its role in. Choreography is resource-efficient, but you need to look into multiple pieces of code spread over several services to see or debug the whole scenario.

(inexact) No communication

Finally, some kinds of services (*device drivers* and *Nanoservices*) never communicate with each other. Strictly speaking, such services don't make a system – instead, they are isolated monoliths which are managed by a higher-level component (OS kernel for drivers, client for nanoservices).

Nevertheless, it is a fun fact, that as soon as the services don't communicate, the main drawbacks of the *Services* architecture disappear:
- There is no slow and error-prone interservice communication (they never communicate!).
- It's not hard to debug multi-service use cases (there are no such scenarios!).
- The services don't corrupt data on crash (there are no distributed transactions).

Variants by size

Last but not least, the simplest classification of subdomain-separated components is by their size:

Whole subdomain: Domain Services

Each *domain service* [FSA] of *Service-Based Architecture* [FSA] implements a whole subdomain. It is the product of full-time work of a dedicated team. A project is unlikely to have more than 10 of such services (in part because the number of top-level subdomains is limited).

Part of a subdomain: Microservices

Microservice enthusiasts estimate the best size of a component of their architecture to be below a month of development by a single team. Such a size allows for a complete rewrite instead of refactoring if the requirements change. When a team completes one microservice it can start working on another, probably related, one, while still maintaining its previous work. A project is likely to contain from tens to few hundreds of microservices.

Class-like: Actors

An *actor* is an object with a message-based interface. They are used correspondingly. Though the size of an actor may vary, as does the size of an OOP class, it is still very likely to be written by a single programmer.

Single function: FaaS, Nanoservices

A *nanoservice* is a single function ([FaaS](#) [[DDS](#)]), usually deployed to a serverless provider. Nanoservices are used as API method handlers or as building blocks for *pipelines*.

Variants by internal structure

A service is not necessarily monolithic inside. As a service is encapsulated from its users by its interface, it can have any kind of internal structure. The most common cases are:

| **Monolithic Service** | **Layered Service** | **Hexagonal Service** | **Scaled Service** | **Cell** |
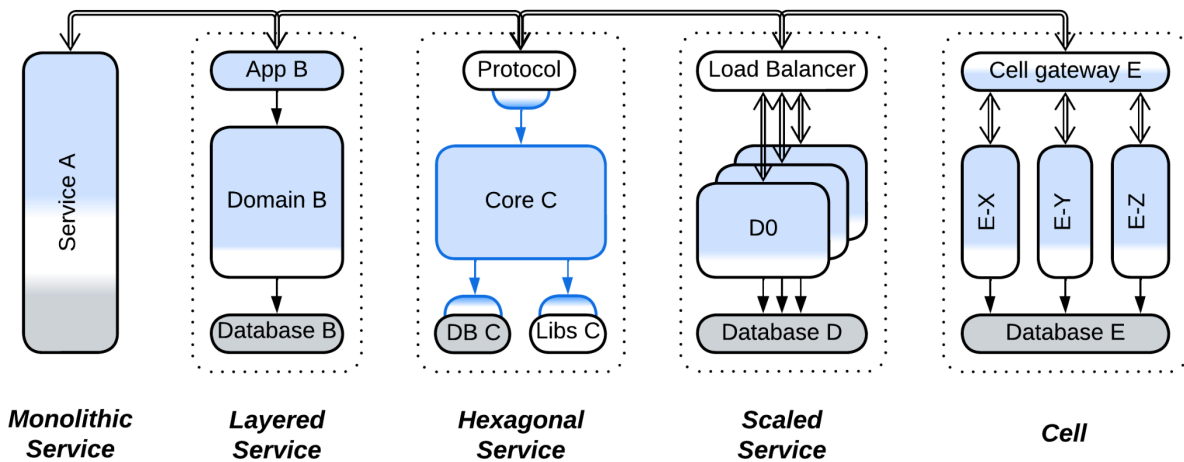
All of them can be intermixed in a single system.

Monolithic service
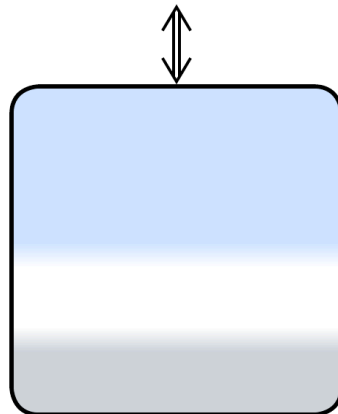
A service with no definite internal structure, probably small enough to allow for complete rewrite instead of refactoring – the ideal of proponents of *Microservices*. It's simple&stupid to implement but relies on external sources of persistent data. For example, *device drivers* and *actors* usually get their (persisted) config during initialization. A monolithic backend service may receive all the data it needs in incoming requests, query another service or read it from a *shared database*.

Layered service

Use cases	**Integration**	Orchestrates core methods and external services
The bulk of logic	**Core**	Implements complex behavior
Persistence	**Database**	Stores the data
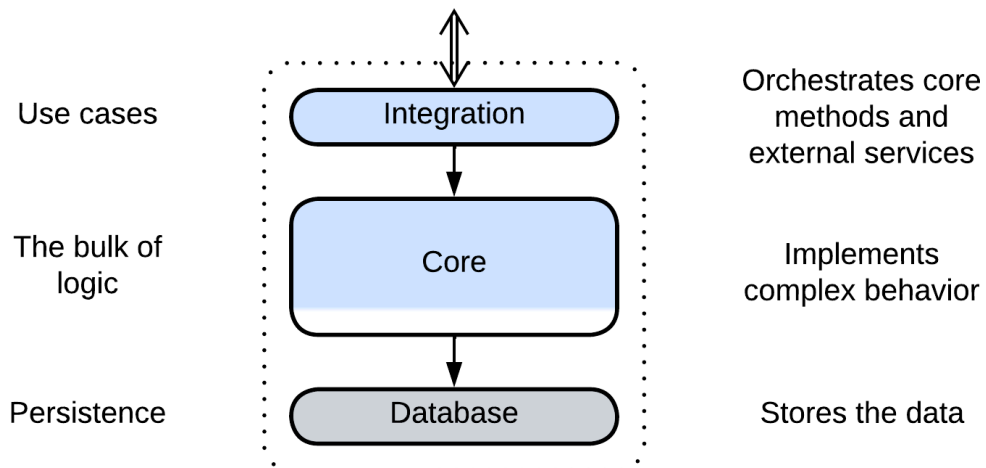
The service is divided into *layers*. The approach is very common both with backend (micro-)services, where at least the database is separated from the business logic, and with *device drivers* in system programming, where hardware-specific low-level interrupt handlers and register access are separated from the main logic and from the high-level OS interface.

Layering provides all the benefits of the *Layers* pattern, including support for conflicting forces, which may manifest, for example, as the ability to deploy the database to a dedicated server in backend or as a very low latency in the hardware-facing layer of a device driver.

Another benefit comes from the existence of the upper integration layer which may orchestrate interactions with other services.

Hexagonal service

The business logic (core) does not depend on any external component

That protects the service from vendor lock-in

All the external modules get adapters that translate between the module's generic API and the core's API

The core logic uses only the interfaces it defines
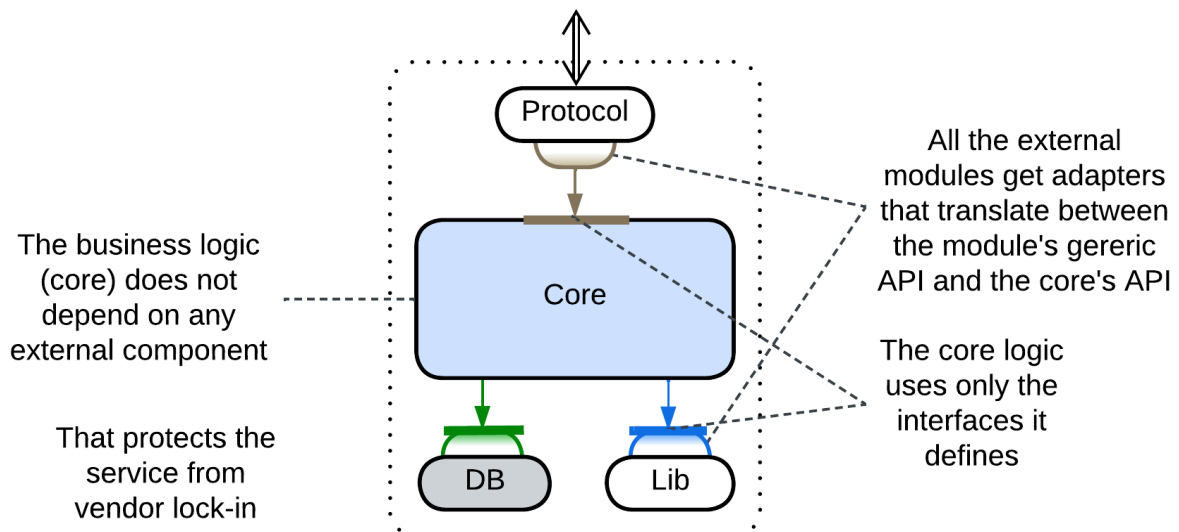
All the external dependencies of the service are isolated behind generic interfaces.

This is the application of *Hexagonal Architecture* which both ensures that the business logic does not depend on technologies selected and protects from vendor lock-in. Highly recommended for long-living projects.

Scaled service

If the instances are stateless, the load balancer forwards requests to any idle instance

The shared database stores the system state

Load Balancer

Database

Every shard may own a part of the system's state

Then the load balancer forwards each requests to the shard with matching data
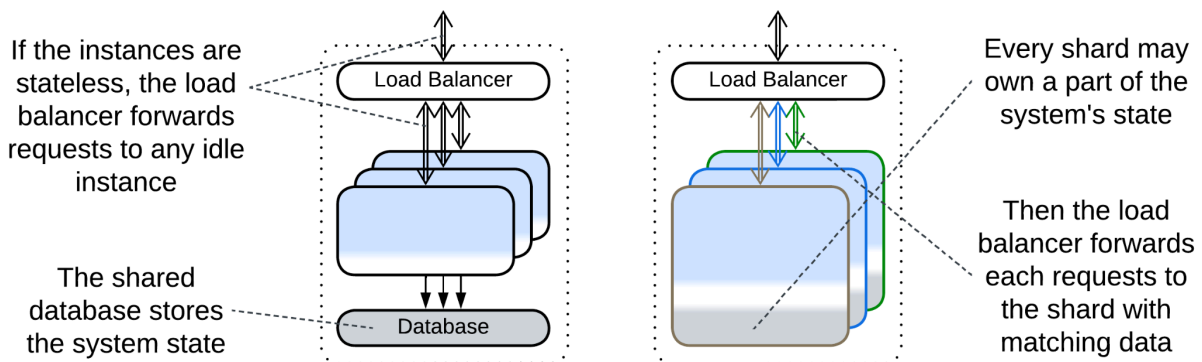
Load Balancer

There are multiple instances of a service. In most cases they *share a database* (though sometimes the database may be *sharded together with the service that uses it*) and get their requests through a *load balancer*.

Cell (WSO2 definition) (service of services)

The cell encapsulates several subservices and is deployed as a whole

Some of the subservices may share a database ...

Cell gateway

Subservice A   Subservice B   Subservice C

DB1   DB 2

The cell gateway hides the internals of the cell from its environment

Usually there is a single instance of each subservice

... while others are more independent
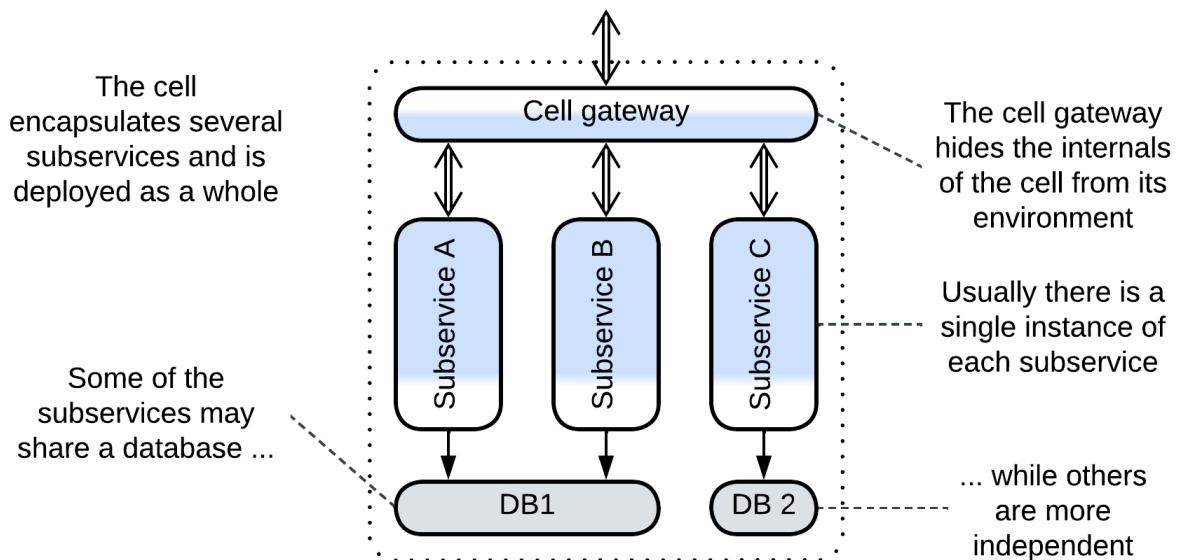
The service is split into a set of subservices, and all the incoming and outgoing communication goes through a *cell gateway* which encapsulates the *cell* from its environment. A cell may deploy its own *middleware* and/or *share a database* among its components.

Cell-Based Architecture (according to WSO2, as opposed to Amazon's alias for *Shards*) appears when there is a need to recursively split a service, either because it grew too large or because it makes sense to use several incompatible technologies for its parts. It may also be applied to group services if there are too many of them.

Examples

Services pervade advanced architectures which either build around a layer of services that contains the bulk of the business logic (*Proxy*, *Orchestrator*, *Middleware* and *Shared Repository*) or use small services as an extension of the main monolithic component (*PlugIns* and *Hexagonal Architecture*). *Polyglot Persistence*, *Backends for Frontends* and

Service-Oriented Architecture go all out partitioning the system into interconnected layers of services. *Hierarchy* and *Mesh* require the services to implement or use a polymorphic interface to simplify the components that manage them.

Examples of *Services* (a system divided by subdomain) include:

Service-Based Architecture

Usually there is a single instance of each service

Some of the services may share a database ...

... while others are more independent
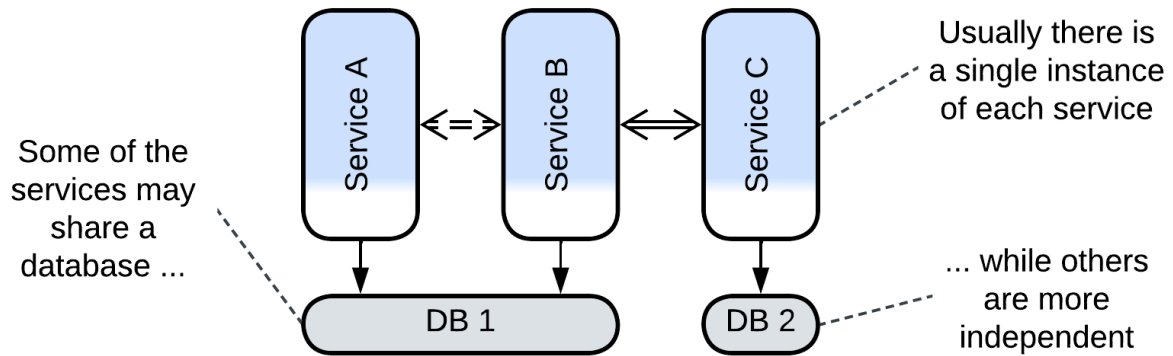
This is the simplest use of services where each subdomain gets a dedicated component. A *Service-Based Architecture* [FSA] tends to consist of few coarse-grained services, some of which may *share a database* and have little direct communication. An *API gateway* is often present as well.

Microservices

Services are scaled automagically

Each kind of a service owns a database

Each instance of a service gets a sidecar with common libraries

The mesh interconnects and manages the service instances via their sidecars
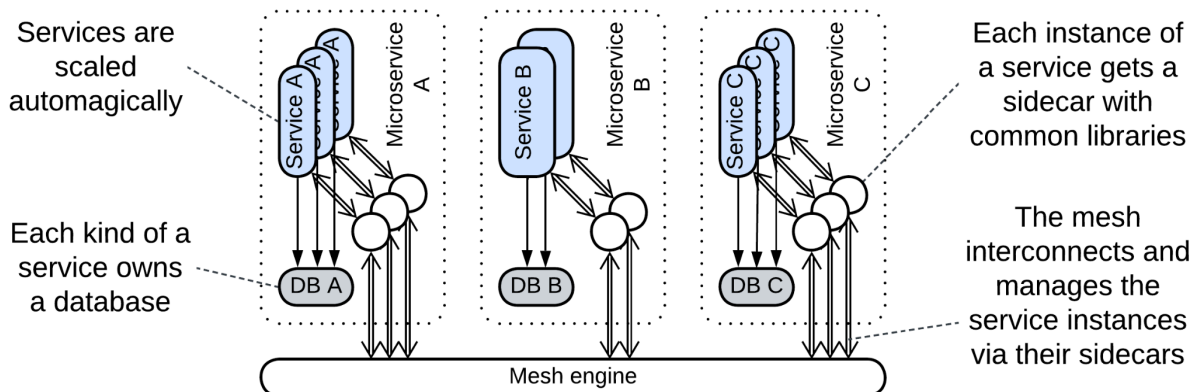
Microservices [MP, FSA] are usually smaller than *Service-Based Architecture*, with their multiple services per subdomain and strict decoupling: no *shared database*, independent (and often dynamic) scaling and deployment. Even *orchestration* and distributed transactions (*sagas*) are considered to be a smell of bad design.

Microservices fit loosely coupled domains with parts that drastically vary in forces and technologies. Any attempt to use them for an unfamiliar domain is calling for trouble. Some authors insist that the "micro-" means that a *microservice* should not be larger in scope than a couple of weeks of a team's programming. That allows for rewriting one from scratch instead of refactoring. Others oppose that too high granularity makes everything overcomplicated. Such a diversity of opinions may mean that the applicability and the very definition of *Microservices* varies from domain to domain.

This architecture usually relies on a *service mesh* for *middleware* where common functionality, like logging, is implemented in co-located *sidecars* [DDS]. A layer of *orchestrators* (called *integration microservices*) may be present, resulting in *Cell-Based Architecture* or *Backends for Frontends*.

Dynamically scaled pools of service instances are common thanks to the elasticity of hosting in a cloud. Extreme elasticity requires *Space-Based Architecture*, which puts a node of a distributed in-memory database in each *sidecar*.

Actors

There are often many instances of an actor ...

The underlying framework may feature a shared database
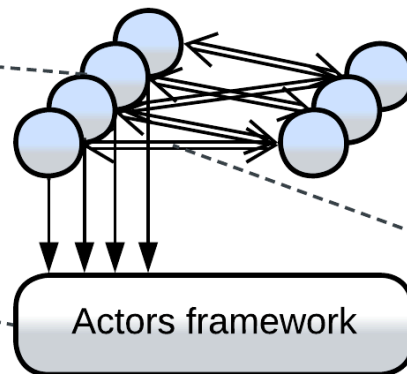
... that communicate with other actors by sending messages

Actors framework

An *actor* is an entity with private data and a public message queue. They are like objects with the difference that actors communicate only by sending each other asynchronous messages. The fact that a single execution thread may serve thousands of actors makes actor systems an extremely lightweight approach to asynchronous programming. As an actor is usually single-threaded, there is no place for mutexes and deadlocks in the code, and it is possible to replay events. Non-blocking *Proactors* are very very good for real-time systems.

Actors have long been used in telephony (which is the domain where real-time communication meets complex logic and low resources) and with the invention of distributed runtimes (e.g. Erlang/OTP or Akka) they found their place in messengers and banking, where there is a need to interconnect millions of users while providing personalized experience and history for everyone. Every user gets an actor that represents them in the system by communicating both with the other actors (forming a kind of *Mesh*) and with the user's client application(s).

If we apply a bit of generalization, we can deduce that any server or a backend service is an actor as its data cannot be accessed from outside while asynchronous IP packets are its only means of communication. Services of *Event-Driven Architecture* closely match the definition of actors.

(inexact) Nanoservices (API layer)

Client

Each nanoservice is a single API method

Get X Post X Get Y Post Y

The nanoservices don't interact

Shared Datastore

They share a database
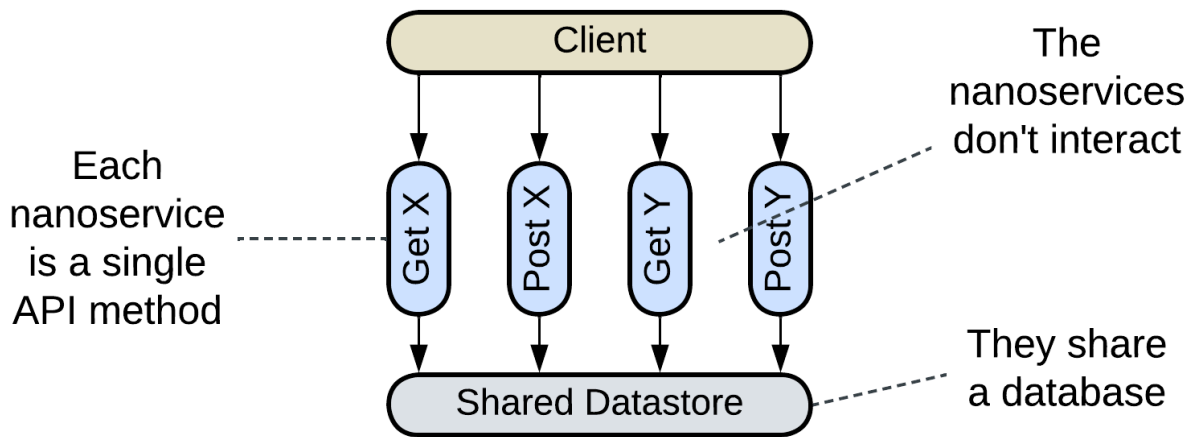
Though *Nanoservices* are defined by their size (a single function), not system topology, I want to mention their specific application from Diego Zanon's book *Building Serverless Web Applications*. That example is interesting because it comprises a single layer of isolated functions (each handling a single API method) which may share functionality by including code from a common repository. As nanoservices of this kind never interact, the common drawbacks of *Services* (poor debugging and high latency) don't apply to them.

(inexact) Device Drivers

User applications interact with the hardware through a unified OS interface

App A App B

User space

System call interface

The OS selects an appropriate driver based on the call arguments

Kernel space

Each hardware component gets a dedicated driver

Driver X Driver Y Driver Z

HW X HW Y HW Z
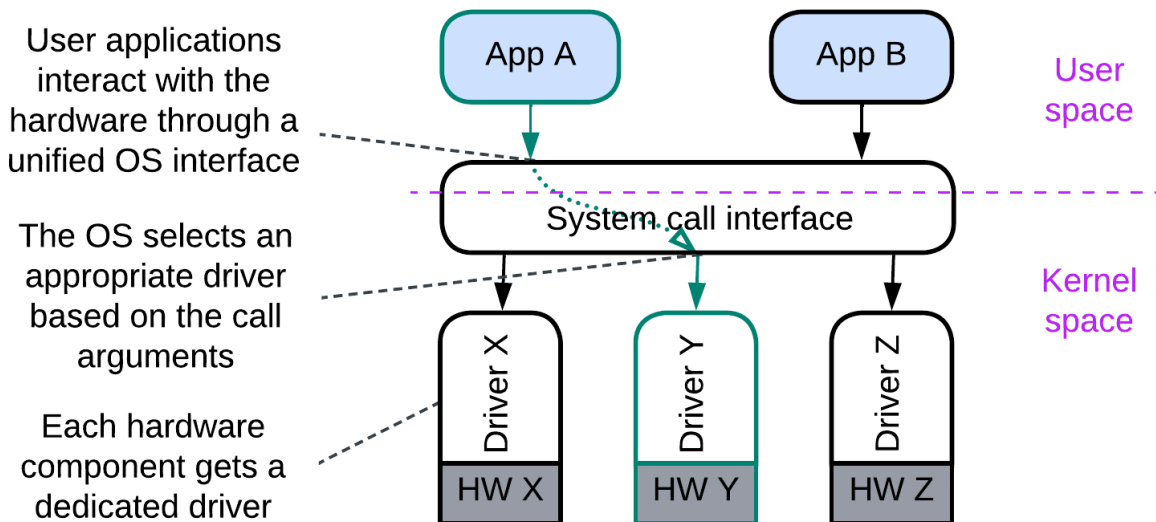
An operating system must run efficiently with any set of hardware components that come from different manufacturers. It is impossible to predict all the combinations beforehand. Thus it employs a service (called *driver*) per hardware device. A driver adapts a manufacturer- and model-specific hardware interface to the generic interface of the OS *kernel*, thus allowing for the kernel to operate the hardware it controls without the detailed knowledge of its model. Internally a driver is usually layered:

- The lowest layer, called *hardware abstraction layer* (*HAL*), provides a model-independent interface for a whole family of devices from a manufacturer.
- The next layer of a driver is likely to contain manufacturer-specific algorithms for efficient use of the hardware.

- The third layer, if present, is probably busy with high-level tasks which are common for all devices of the given type and may be implemented by the kernel programmers.

The whole system of kernel, drivers and user applications comprises the *Microkernel* pattern which is the common architecture for bridging resource consumers and resource providers. As the drivers don't need to coordinate themselves (this is done by the kernel), they don't really make a system of *Services* and thus don't have the corresponding drawbacks.

Evolutions

Services, just like the other basic metapatterns, are subject to a wide array of evolutions, which are summarized below and detailed in Appendix E.

Evolutions that add or remove services

Services work well when each service matches a subdomain and is developed by a single team. If those premises change, you'll need to restructure the services:

- A new feature request may emerge outside of any of the existing subdomains, creating a new *service* or a *service* may grow too large to be developed by a single team, calling for division.

Overgrown services can be split by subdomain
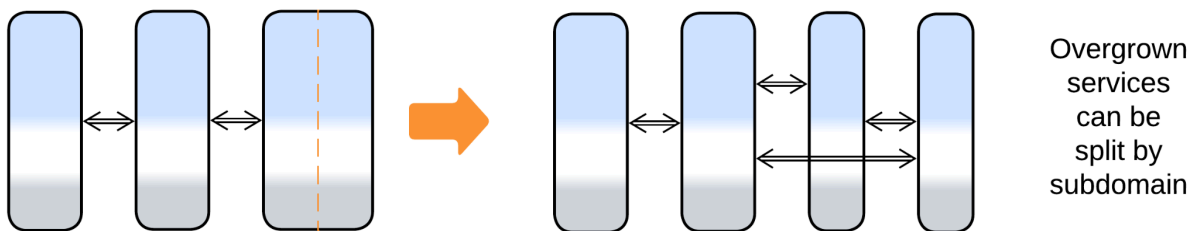
- Two services may become so strongly coupled that they fare better if merged together, or the entire system may need to be glued back into a *monolith* if the domain knowledge changes or if interservice communication strongly degrades performance.

Coupled services can be merged

Evolutions that add layers

The most common modifications of a system of services involve supplementary system-wide layers which compensate for the inability of the services to share anything among themselves:

- A *middleware* knows of all the deployed service instances. It mediates the communication between them and may manage their scaling and failure recovery.

A middleware manages service instances and provides them with a means of communication

- *Sidecars* [DDS] of a *service mesh* make a virtual layer of shared libraries for the *microservices* it hosts.

Service mesh is a distributed middleware for microservices

There is a virtual layer of sidecars - libraries colocated with each service instance

The mesh engine transfers messages between and scales the microservices

- A *shared database* simplifies the initial phases of development and interservice communication.

It is often easier to start a project with a single shared database

- *Proxies* stand between the system and its clients and take care of shared aspects that otherwise would need to be implemented by every service.

A proxy stands between the system and its clients and does something useful

- An *orchestrator* is the single place where the high-level logic of all use cases resides.

The orchestrator uses the services to execute client requests

The services don't need to intercommunicate

Those layers may also be combined into *combined components*:
- *Message Bus* is a *middleware* that supports multiple protocols.
- *API Gateway* combines *Gateway* (a kind of *Proxy*) and *Orchestrator*.
- *Event Mediator* is an *orchestrating middleware*.
- *Enterprise Service Bus (ESB)* is an *orchestrating message bus*.
- *Space-Based Architecture* employs all the four layers: *Gateway*, *Orchestrator*, *Shared Repository* and *Middleware*.

Evolutions of individual services

Each service starts as a *monolith* or *layers* and may undergo corresponding evolutions:
- *Layers* help to reuse 3rd party components (e.g. a database), organize the code, support conflicting forces and the upper layer of the service may orchestrate other services.
- A *cell* is a subdivision of a service into several services that share an *API gateway* and may share a database or a *middleware*. All the components of a *cell* are usually deployed together. That helps to deal with overgrown services without increasing the operational complexity of the system – but only if the cell's components are loosely coupled.
- A service may use a *load balancer* or a load balancing *middleware* to scale. The instances usually persist to a *shared database*.
- *Polyglot Persistence* or *CQRS* may be used inside a service to improve the performance of its data layer.
- *Materialized views* [DDIA] or a *query service* [MP] help reconstruct the state of other services from *event sourcing*.
- *Hexagonal Architecture* isolates the business logic of the service from external dependencies.
- In rare cases *Plugins* or *Scripts* help to vary the behavior of a service.

Summary

Services are mostly dedicated to subdividing a large codebase into modules of manageable size, each assigned to a dedicated team. They may vary in technologies and quality attributes. However, services have a hard time cooperating in anything, from sharing data to debugging, and come with an innate performance penalty. There are options between *Monolith* and *distributed Services* with milder benefits and drawbacks.

Pipeline

Business logic

Generic code

Data

Method call
Messaging
Data stream
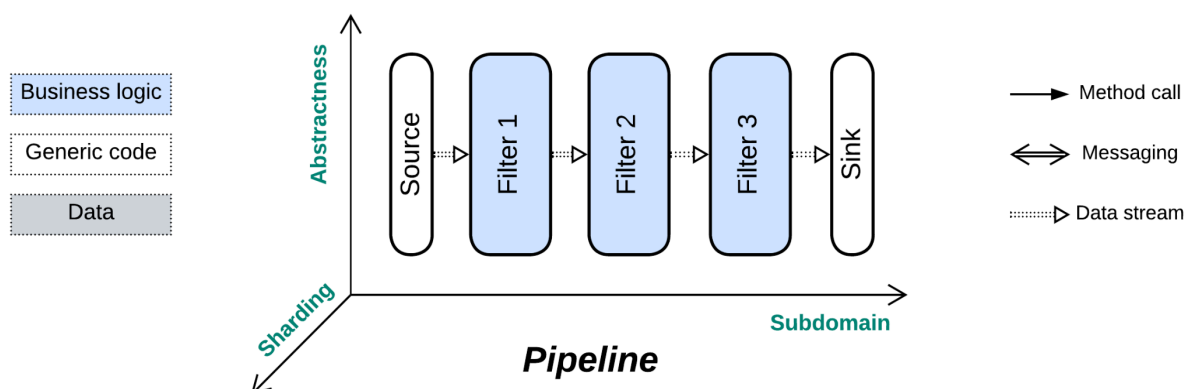
Pipeline

Never return. Push your data through a chain of processors.

Known as: Pipeline [DDS].

Variants:

By scheduling:

- Stream processing [DDIA] / Neartime system [DDIA],
- Batch processing [DDIA, DDS] / Offline system [DDIA].

Examples:

- Pipes and Filters [POSA1, POSA4] / Workflow System [DDS, DDIA],
- Choreographed (Broker Topology) Event-Driven Architecture (EDA) [SAP, FSA, DDS],
- Data Mesh [LDDD],
- Function as a Service (FaaS) [DDS] / Nanoservices (pipelined).

Structure: A module per step of data processing.

Type: Main.

Benefits	Drawbacks
It is very easy to add or replace components	Deteriorates as the number of scenarios grows
Multiple development teams and technologies	Poor latency
Good scalability	Significant communication overhead
The components may be reused	
The components may be tested in isolation	

References: *Pipes and Filters* is defined in [POSA1] and is the foundation for part 3 ("Derived Data") of [DDIA]. [DDS] overviews all kinds of pipelines in general while [FSA] has a very deep chapter on *Event-Driven Architecture*.

Pipeline is a variation of *Services* with no user sessions [DDS], unidirectional data flow and quite often a single message type per communication channel (which thus becomes a *data stream*). As processed data does not return to the module that requested the processing, there is no common notion of request ownership or high-level (supervisor, application) and low-level (worker, domain) business logic. On one hand, as all the components are equal and know nothing about each other (the interfaces are often limited to a single entry point), it is very easy to reshape the overall algorithm. On the other hand, the system lacks the abstractness dimension, thus any new use case builds a separate pipeline

which may easily turn the architecture into a mess of thousands of intrinsically interrelated pieces as the number of scenarios grows.

Performance

As any task for a *pipeline* is likely to involve all (or most of, if branched) its steps, there is no way to optimize away the communication. Thus, latency tends to be high. However, as its components are often stateless, multiple instances of individual services or entire pipelines can be run in parallel, making *Pipeline* a highly scalable architecture.

Another point of notice is that a local *pipeline* naturally spreads the load among available CPU cores without any explicit locks or thread synchronization.

Dependencies

There are three ways to build communication in a *pipeline*, with different dependencies:
- *Commands* make each service depend on the services it sends messages to. It is easy to add a new input to such a pipeline.
- With *publish/subscribe* each service depends on the services it subscribes to. That case favors downstream branching with multiple sinks.
- The services may share the *message schema* in which case all of them depend on it, not on each other. That allows for reshuffling the services.
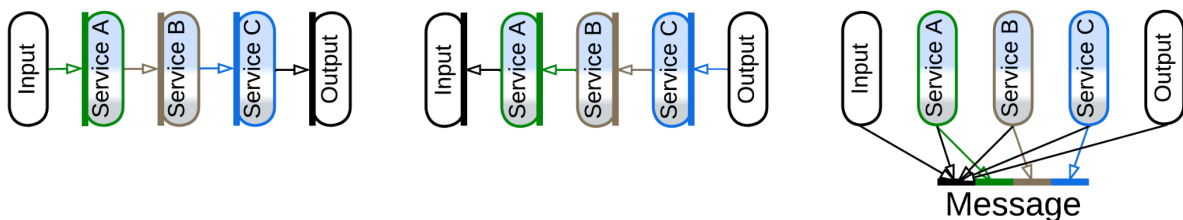
Dependencies with Commands

Dependencies with Notifications

Dependencies with Shared Schema

See the [*Choreography* chapter](#) for more detailed discussion.

Applicability

Pipeline is good for:
- *Experimental algorithms.* This architecture allows for the data processing steps both to be tested in isolation and connected into complex systems with no changes in the code.
- *Easy scaling.* Pipelines tend to evenly saturate all the available CPU cores without any need for custom schedulers. Stateless services can run distributed, thus the pipeline's scalability is limited only by the data channels.
- *Tailoring projects*. Many pipeline components are abstract enough to be easily reused, greatly lowering the cost of serial development of customized projects once the company builds a collection of common reusable services.

Pipeline does not work for:
- *High number of use cases.* The number of components and their agglomerates is going to be roughly proportional to the number of supported use cases and will easily overwhelm any developer or architect if the number of scenarios grows with time.

- *Low latency*. Every transfer of a data packet between services takes time, not in the least measure because of data serialization. Moreover, the next service is likely to be busy processing its previous data packets or has to wait on the OS scheduler.

Relations

Pipeline:
- Is a kind of *Services* with unidirectional communication and often single input type.
- Is involved in *CQRS*, *Polyglot Persistence* with derived databases and *MVC*.
- Can be extended with a *proxy*, *middleware* or *shared repository*.

Variants by scheduling

A pipeline may be always active or run once in a while:

Stream processing, Neartime system

Stream processing [DDIA] is when the pipeline components (*jobs, processors, filters, services* – whatever you prefer to call them) are actively waiting for input and process each incoming item immediately. This results in *near-real-time* (*neartime*) latency [DDIA] but the pipeline uses resources even when there is nothing to process.

Batch processing, Offline system

Batch processing [DDIA, DDS] happens when a batch of input items is collected in a storage, then the pipeline is run (often by a scheduler) to process it. Such a mode of action is called *offline* [DDIA] as the processing results are delayed. However, the company saves on resources as the pipeline is active only for brief periods.

Examples

Pipelines may be local or distributed, linear or branched (usually trees, but cycles happen in practice), they may utilize a feedback engine to keep throughput of all their components uniform by slowing down faster steps or scaling out slower ones. In some systems *pipes* (channels) or *filters* (services) persist data. Pipes may store the processed data in files or databases to enable error recovery and event sourcing. Filters may need to read or write to an (often shared) database if the data processing relies on historical data. Moreover, transferring data through a pipe may be implemented as anything ranging from a method call on the next filter to a pub/sub framework.

Such a variety of options enables the use of pipelines in a wide range of domains. Notwithstanding, there are few mainstream types of pipeline architectures:

Pipes and Filters, Workflow System

A filter is a single step of processing the data stream ...

... which does one thing and is often polymorpfic

The source inputs the data

Filter 1 → Filter 2 → Filter 3

Source

Sink

The sink outputs the results
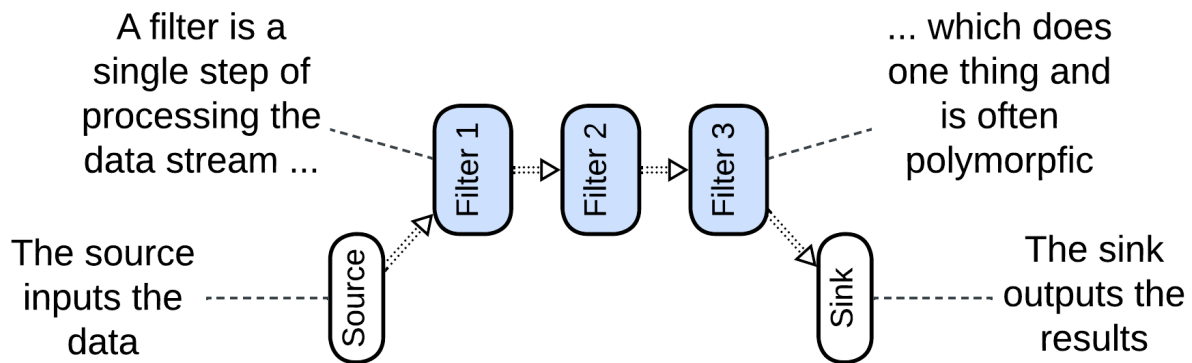
Pipes and Filters [POSA1, POSA4] usually names a linear local system, which often runs in a single process to avoid the overhead of data serialization. It may range from a Unix shell script that passes file contents through a series of utilities to a hardware pipeline for image processing in a video camera. The *filters* tend to be single-purpose (handle one type of payload) and stateless. In some cases a filter may use a dedicated hardware. The entire pipeline often operates on a single data format (*Stamp Coupling* [SAHP]).

Though most often a filter waits for data to appear in its input pipe, processes it and pushes to its output pipe, allowing for multiple filters to run in parallel, some implementations may let the source push the data through the entire pipeline, with each filter directly calling the next filter in the line, or the sink pull the data by making direct upstream calls [POSA1]. That removes the need for pipes but is limited to a single CPU core.

Workflow [DDIA, DDS] is sometimes used as a modern name for a similar stepwise processing, which often stores intermediate results in a file or database and may run distributed. However, the same word generally describes the sequence of high-level steps in a use case [PEAA], which is also called *application* or *integration* logic.

Examples: Unix shell pipes, processing of video streams, many types of hardware.

Choreographed (Broker Topology) Event-Driven Architecture (EDA)

The services subscribe for each other's notifications

The processing graph is branched

A service may process multiple inputs
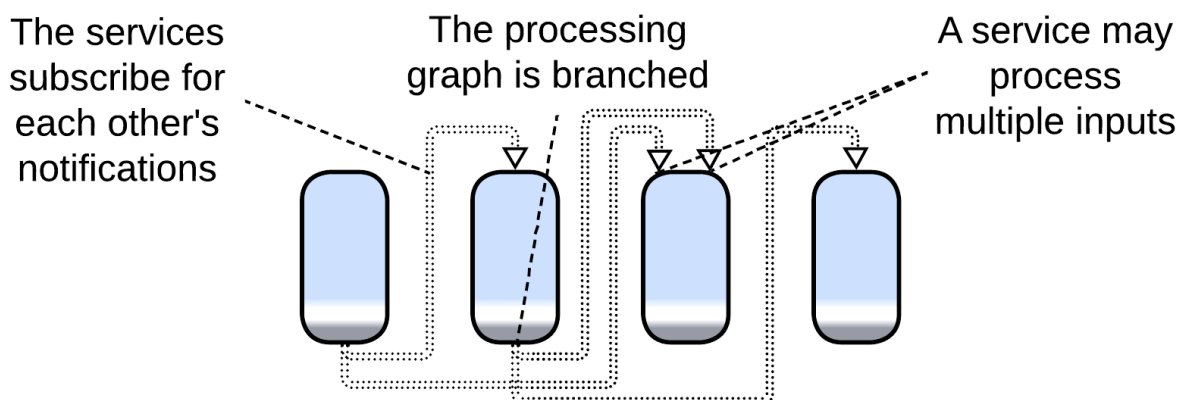
Event-Driven Architecture (*EDA*) means that the system is built of services which use events to communicate in a non-blocking way. The idea is similar to the actor model of telecom and embedded programming. Thus, *EDA* itself does not define anything about the structure of the system (except that it is not monolithic).

In practice, there are two kinds of *Event-Driven Architectures*:

- *Choreography / broker topology* – the events are notifications (usually via pub/sub) and the services involved form tree-like structures, which matches our definition of *Pipeline*.
- *Orchestration / mediator topology* – the events are request/confirm pairs, and usually there is a single entity that drives a use case by sending requests and receiving confirmations, which corresponds to our *Services* metapattern with the supervisor being an *orchestrator*, discussed in the chapter of its own.

An ordinary *choreographed Event-Driven Architecture* [SAP, FSA, DDS] is built as a set of subdomain services (similar to the parent *Services* metapattern). Each of the services subscribes to notifications from other services which it uses as action/data inputs and produces notifications that other services may rely on. For example, an email service may subscribe for error notifications from other services in the system to let the users know about troubles that occur while processing their orders. It will also subscribe to the user data service's add/edit/delete notifications to keep its user contacts database updated.

The example shows several differences from a typical *Pipes and Filters* implementation:
- The system supports multiple use cases (e.g. user registration and order processing).
- Services have multiple entry points (e.g. a mail service may expose an order error handler and user created handler).
- A notification that a service produces may have many subscribers or no subscribers (nobody needs to act on our sending a mail to a user).

Those points translate to the difference in structures: while *Pipes and Filters* is usually a linear chain of subdomain services, *EDA* entails multiple branched (and sometimes looped) event flow graphs over a set of subdomain services.

Pipelined *Event-Driven Architecture* (often boosted with event sourcing) works well for highly loaded systems of moderate size as larger projects are likely to grow forbiddingly complex graphs of event flows and service dependencies. The architecture's scalability is limited by the services' databases and the pub/sub framework.

Event-Driven Architecture may use a *gateway* as a user-facing event source and sink and a *middleware* for an application-wise pub/sub engine. *Front Controller* [SAHP] or *Stamp Coupling* [SAHP] can be used if it is important to know the state of the requests that are being processed by the pipeline.

Examples: high performance web services, IoT.

Data Mesh

Data Mesh is a graph for streaming analytical data

It is built on top of a system of services

Each service outputs its data in a convenient format

There may be data aggregation nodes
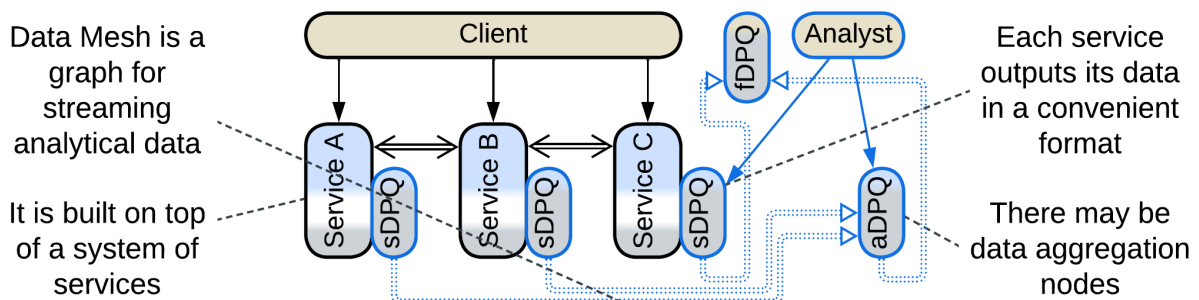
First and foremost, *Data Mesh* [LDDD, SAHP] is not a *Mesh*, but rather a *Pipeline*. This architecture applies *CQRS* on the system level: it separates the interfaces and channels through which the services change their state (matching *commands* or *OLTP* in *CQRS*) and

the ones used to retrieve their data (similar to *queries* or *OLAP*). That results in two overlapping systems, *operational* and *analytical*, that share most of their nodes.

The *operational system* is an ordinary *Microservices* or *Event-Driven Architecture*.

The *analytical system* contains *Data Product Quanta* (*DPQ*) – services that provide convenient access (streaming, replaying and possibly querying) to parts of the system's data. The *DPQs* are assembled into a graph akin to *Event-Driven Architecture*. There are three kinds of *DPQs* [SAHP]:

- A *source-aligned (native) DPQ* (*sDPQ*) coupled to an *operational service* and streams (or provides queries into) the service's data. It is likely to be implemented as a *reporting database*.
- An *aggregate DPQ* (*aDPQ*) merges and transforms inputs from several sources (*sDPQs* or *aDPQs*). It is similar to *Query Service*.
- A *fit-for-purpose (custom-made) DPQ* (*fDPQ*) is an end-user (leaf application) of the *data mesh*'s data. It may collect a dataset for machine learning or let a business-analyst do their research. *fDPQs* tend to be short-lived one-off projects.

There is a pragmatic option to allow *operational services* to resort to the *analytical system*'s *DPQs* instead of duplicating the functionality already present in the *DPQs* to query methods and *CQRS views* in the *operational services*.

Function as a Service (FaaS), Nanoservices (pipelined)

CRUD in nanoservices
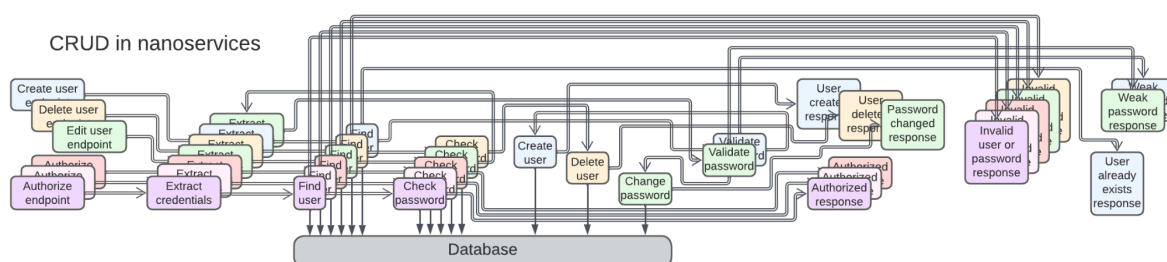
A *nanoservice* is, literally, a *function as a service* (*FaaS*) [DDS] – a stateless (thus perfectly scalable) component with a single input. They can run in a proprietary cloud *middleware* over a *shared database* and are *chained into pipelines*, one per use case. The code complexity stays low, but as the project grows, the integration will quickly turn into a nightmare, with hundreds or thousands of interconnected services.

Nanoservices are good for rapid development of small elastic/scalable applications. The supported load is limited by their shared database, the project evolvability – by the complexity of scenarios. As any use case is going to involve many asynchronous steps, latency is not a strong side of *Nanoservices*.

Evolutions

Pipeline inherits its set of evolutions from *Services*. Filters can be added, split in two, merged or replaced. Many systems employ a *middleware* (a pub/sub or pipeline framework), a *shared repository* (which may be a database or a file system) or *proxies*.

There are a couple of pipeline-specific evolutions, with more details available in Appendix E:

- The first service of the *pipeline* can be promoted to *Front Controller* [SAHP] which tracks the status updates for every request it handles.

- Adding an _orchestrator_ turns *Pipeline* into _Services_. As the high-level business logic moves to the orchestration layer, the filters don't need to interact directly, the interfilter communication channels disappear and the system becomes identical to *Orchestrated Services*.

The front controller subscribes to notifications to know the state of each running request

An orchestrator encapsulates the logic of use cases

The services don't need to communicate to each other directly
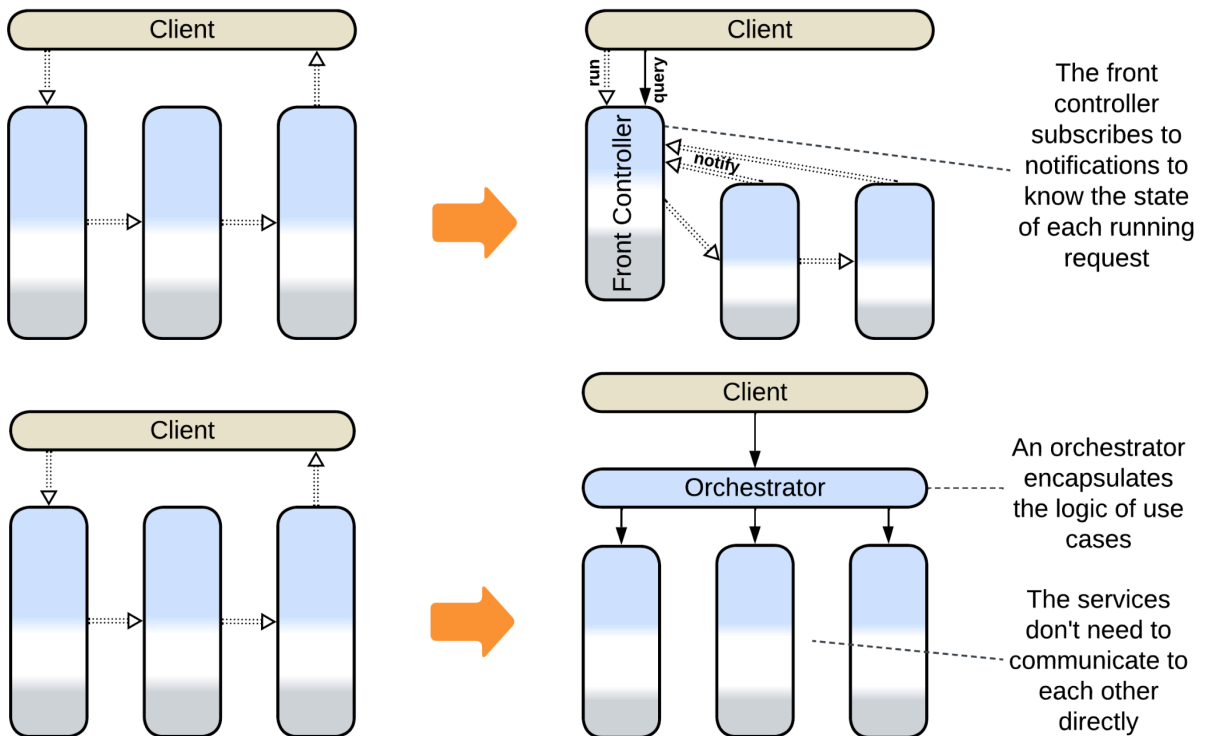
Summary

Pipeline not only divides the code into smaller modules but is also very flexible: its components are easy to add, remove or replace. Multiple use cases can be built over the same set of services. Scalability is good. Event replay helps debugging. However, operational complexity restricts the architecture to smaller domains with a limited number of use cases.

Part 3. Extension Metapatterns

These patterns extend services, shards or even a monolith with a layer that provides an aspect or two of the system's behavior and often glues other components together.

Middleware

The middleware provides communication for the services
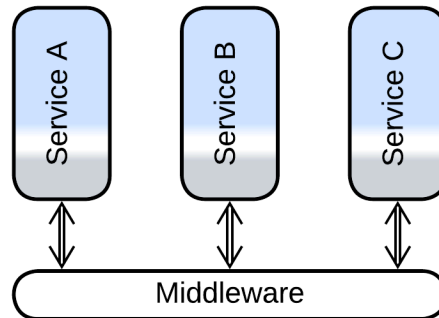
It may also manage their instances

A middleware is a layer that implements communication between instances of the system's components and may also manage the instances. This way each instance is relieved of the need to track other instances it deals with.

Includes: (Message) Broker, Deployment Manager.

Shared Repository

The shared repository owns the system's data
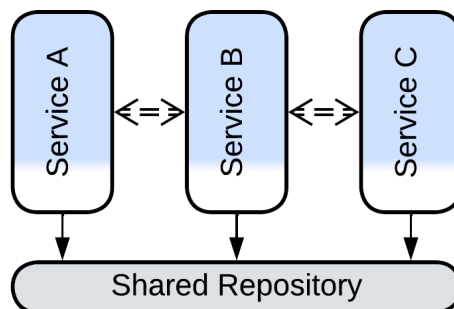
The services communicate directly or through the repository

A shared repository stores the system's data, maintains its integrity through transactions and may support subscriptions to changes in subsets of the data. That lets other system components concentrate on implementing the business logic.

Includes: Shared Database, Blackboard, Data Grid of Space-Based Architecture, shared memory, shared file system.

Proxy

A proxy stands between a (sub)system and its clients
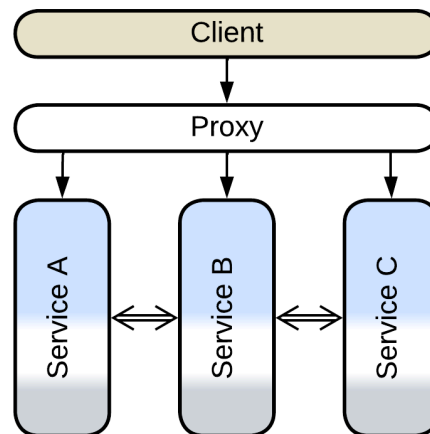
It implements a generic aspect of the system's behavior

A proxy mediates between the system and its clients, transparently taking care of some generic functionality.

Includes: Full Proxy and Half-Proxy; Sidecar and Ambassador; Firewall, Response Cache, Load Balancer, Reverse Proxy, various adapters, including Anticorruption Layer, Open Host Service, XXX Abstraction Layers and Repository.

Orchestrator

The orchestrator runs high-level scenarios by using the services
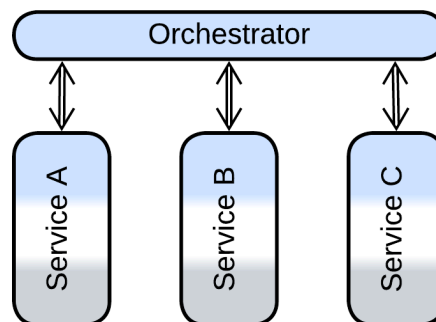
It also keeps the data of the services consistent

An orchestrator implements use cases as sequences of calls to the underlying components, which are usually left unaware of each other's existence.

Includes: Workflow Owner, Application Layer, Facade, Mediator; API Composer, Scatter-Gather, MapReduce, Process Manager, Saga Execution Component, Integration (Micro-)Service.

Combined Component

Several patterns combine functionality of two or more extension layers.

Includes: Message Bus, API Gateway, Event Mediator, Enterprise Service Bus, Service Mesh, Middleware of Space-Based Architecture.

Middleware

Services with a Middleware
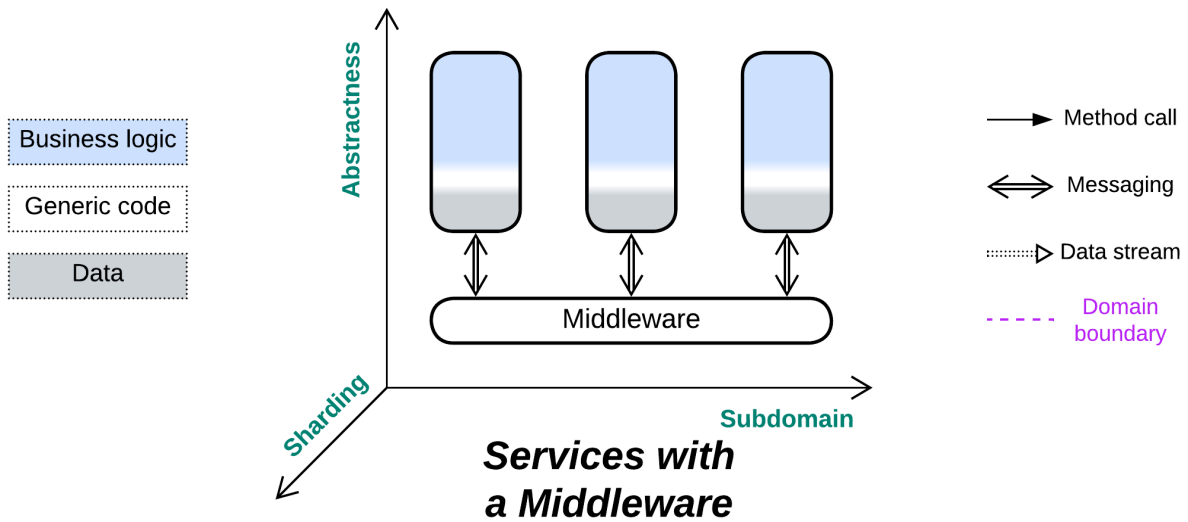
The line between disorder and order lies in logistics. Use a shared transport.

Known as: (Distributed) Middleware.

Aspects:

- Message Broker [POSA1, POSA4, EIP, MP],
- Deployment Manager [SAP, FSA].

Variants: Implementations differ in many dimensions.

The following combined patterns include Middleware:

- (with *Adapters*) Message Bus [EIP],
- (with *Proxies*) Service Mesh [FSA, MP],
- (with *Orchestrator*) Event Mediator [FSA],
- (with *Orchestrator* and *Adapters*) Enterprise Service Bus (ESB) [FSA].

Structure: A low-level layer that provides connectivity.

Type: Extension.

Benefits	Drawbacks
Separates connectivity concerns from the services	May become a single point of failure
Transparent sharding of components	May increase latency
Available off the shelf	A generic middleware may not fit specific communication needs

References: [EIP] is all about it. [POSA4] has a chapter on *Middleware*. However, those books are old, while technologies change every year. There is a Wikipedia article as well.

Extracting transport to a separate layer relieves the components that implement business logic of the need to know addresses and statuses of each other's instances. An industry-grade 3rd party *middleware* is likely to be more stable and provide better error recovery than anything your company can afford implementing on its own.

Middleware may function:

- As *Message Broker* [POSA1, POSA4, EIP, MP] – provide a unified means for communication and implement some cross-cutting concerns like persisting or logging messages.

- As *Deployment Manager* [SAP, FSA] – collect telemetry and manage service instances for error recovery and dynamic scaling.

As *Middleware* is ubiquitous and does not affect business logic, it is usually omitted from structural diagrams.

Performance

Middleware may negatively affect performance, compared to direct communication between services. Old implementations (star topology) relied on a *broker* [POSA1, POSA4, EIP] that used to add an extra network hop for each message and was limiting scalability. Newer *mesh*-based variants may not have that drawback but are very complex and may have consistency issues (according to the CAP theorem).

A more subtle drawback is that the transports supported or recommended by the middleware may be suboptimal for some of the interactions in the system, causing the programmers to hack around the limitations or build higher-level protocols on top of the middleware.

Both cases can be alleviated by adding means of direct communication between the services to bypass the middleware, or by using multiple specialized kinds of middleware. However, that adds to the complexity of the system – the very issue the middleware promised to help with.

Dependencies

Each service depends on the middleware. However, each service also depends on the APIs of the services it communicates with.

A service depends both on the middleware and the services it uses

You may decide to use an *anticorruption layer* [DDD] against your middleware just in case you may need to change its vendor in the future.

Applicability

Middleware helps:
- *Multi-component systems.* When several deployed instances of services need to intercommunicate, they must know addresses (or have channels to) each other. As the number of services grows, so does the amount of information that each service needs to track. Even worse, services sometimes crash or are being redeployed, requiring complicated algorithms that queue messages to deliver them in order once the service is alive again. It makes all the sense to use a dedicated component that takes care of that.

- *Dynamic scaling.* It is good to have a single component that manages routes for interservice messaging to account for newly deployed (or destroyed) service instances.
- *Blue-green deployment, canary release, dark launching* or *traffic mirroring*. It is easier to switch, whether fully or in part, to a new version of a service when the communication is centralized.
- *System stability.* Most implementations of middleware guarantee message delivery with unstable networks and failing components. Many persist messages to recover from failures of the middleware itself.

Middleware hurts:
- *Critical real-time paths.* An extra layer of message processing is bad for latency. Such messages may need to bypass the middleware.

Relations

Middleware for services

Middleware for shards

Middleware for a SOA
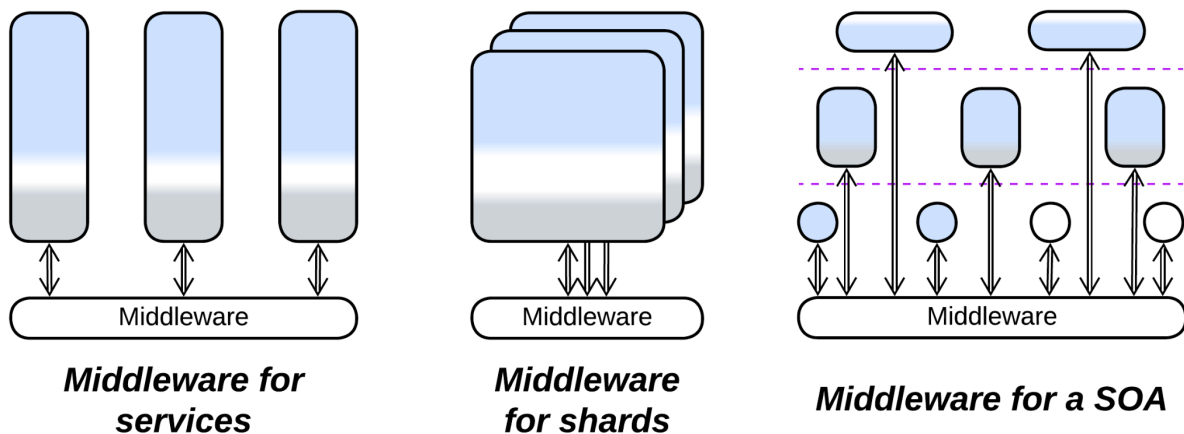
Middleware:
- Extends *Services*, *Service-Oriented Architecture* or in rare cases *Shards* or *Layers*.
- Can be built into a *hierarchy* or merged with other *extension metapatterns* into several kinds of *combined components*.
- Is closely related to *Shared Repository*. A persistent *middleware* employs a *shared repository*.
- Is usually implemented by a *mesh* or *microkernel* (which is often based on a *mesh*).

Variants by functionality

There are several dimensions of freedom for a middleware, some of them may be configurable:

By addressing (channels, actors, pub/sub)

Systems vary in the way their components address each other:
- *Channels* (one to one) – components that need to communicate establish a message channel. Once established a channel accepts messages on its input side and transparently delivers them to its output side. Examples: Linux pipes and private chats.
- *Actors* (many to one) – each component has a public mailbox. Any other component that knows its address or name may push a message into it. Examples: e-mail.

- *Publish/subscribe* (one to many) – a component can publish events to topics. Other components subscribe to the topics and one or all of the subscribers receive copies of a published event. Examples: IP multicast and subscriptions for e-mail notifications.

By flow (notifications, request/confirm, RPC)

Control flow may employ one or more of the following approaches:
- *Notifications* – a component sends a message about an event that occurred and does not really care of the consequences or wait for a response.
- *Request/confirm* – a component sends a message that requests another component to do something and send back the result. The sender may execute other tasks meanwhile. A request must include a unique id which will be sent back in the confirm so that the middleware knows which of the ongoing requests the confirm matches.
- *Remote procedure call* (RPC) is usually built on top of a request/confirm protocol. The difference is that the sender blocks in the middleware while waiting for the confirm, so that for the application code sending the request and receiving the confirm looks like an ordinary method call.

By delivery guarantee

If the transport (network) or the destination fails, a message may not be processed, or may be processed twice. A *Middleware* may promise to deliver messages:
- *Exactly once*. This is the slowest case which is implemented with a distributed transaction. If the network, middleware or the message handler fails, there is no side effect, and the whole process of delivering and executing the message is repeated. The *exactly once* contract is used for financial systems where money should never disappear or duplicate during a transfer.
- *At least once*. On failure the message is redelivered, but the previous message could have been processed (and only the confirmation was lost), thus there is a chance for a message to be processed twice. If the message is *idempotent* [MP], meaning that it sets a value (x = 42) instead of incrementing or decrementing it (x = x + 2), then we can more or less safely process it multiple times (x = 42; x = 42; x = 42;) and use the relatively fast *exactly once* guarantee.
- *At most once*. If something fails, the message is lost and never reprocessed. This is the fastest of the three guarantees, which fits monitoring applications, like weather sensors – it is not too bad if a single temperature measurement is lost as you receive hundreds of them every day.
- *At will* (no guarantee). As with the bare UDP transport, a message may disappear, become duplicated or come out of order. That fits real-time streaming protocols (video or audio calls) where it is acceptable to skip a frame while a frame coming too late is of no use at all. Each frame contains its sequential number, and it is up to the application to reorder and deduplicate the frames it receives.

By persistence

A middleware with a delivery guarantee needs to store messages whose delivery has not been confirmed. They may be:
- Written to the database of the *broker*.

- Persisted in a distributed database in brokerless (*Mesh*) systems.
- Replicated over an in-memory *mesh* storage.

The last cases involve a sibling metapattern, *Shared Repository*.

By structure (Microkernel, Mesh, Broker)

In a classic middleware there is a proxy per service

The proxies register to and communicate through a broker

Modern middleware is a mesh with direct communication between nodes
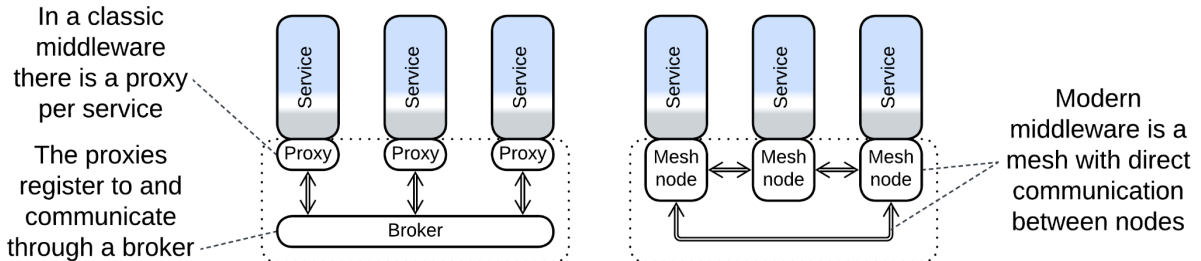
A middleware may be:
- Implemented by an underlying operating or virtualization system (see *Microkernel*).
- Run as a *mesh* of identical modules co-deployed with the distributed components.
- Rely on a single *broker* [POSA1, POSA4, EIP] for coordination.

The last configuration is simpler but features a single point of failure unless multiple instances of the *broker* are deployed and kept synchronized.

Examples of merging Middleware and other metapatterns

There are several patterns that extend *Middleware* with other functions:

Message Bus

Message Bus employs an adapter per service ...

... to translate between protocols the services use
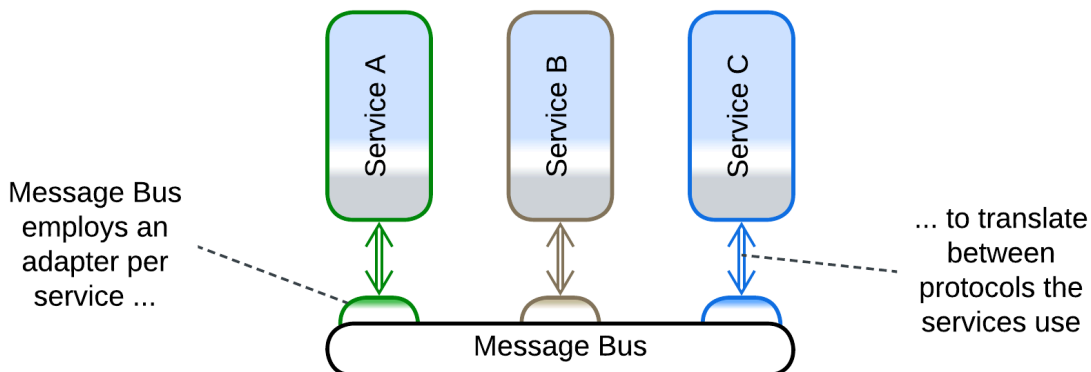
Message Bus [EIP] employs an *adapter* per service to let the services intercommunicate even if they differ in protocols. That helps to integrate legacy services without much changes to their code but degrades performance as up to two protocol translations per message are involved.

Service Mesh

Services are scaled automagically

Each kind of a service owns a database

Each instance of a service gets a sidecar with common libraries

The mesh interconnects and manages the service instances via their sidecars
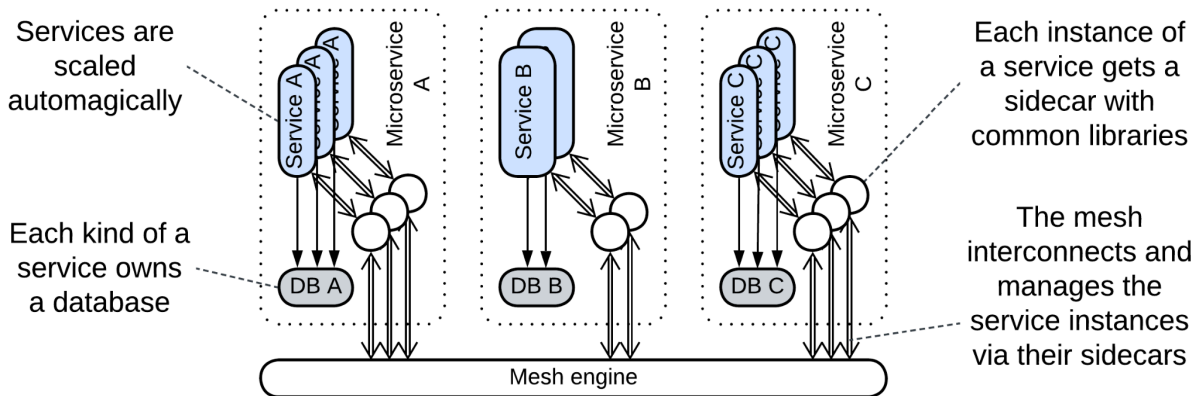
Service Mesh [FSA, MP] is a smart *mesh*-based *middleware* that manages service instances and employs a co-located *proxy* (called *sidecar* [DDS]) per deployed service. The sidecars may provide protocol translation and cover cross-cutting concerns such as encryption or logging. They make a good place to deploy shared libraries.

The internals of *Service Mesh* are discussed in the *Mesh* chapter.

Event Mediator

The event mediator builds a pipeline by transfering client requests to event processors ...

Each event processor contains business logic but does not know the event flow

... and forwarding their responses down the processing graph

The event mediator also provides message channels and scales the event processors
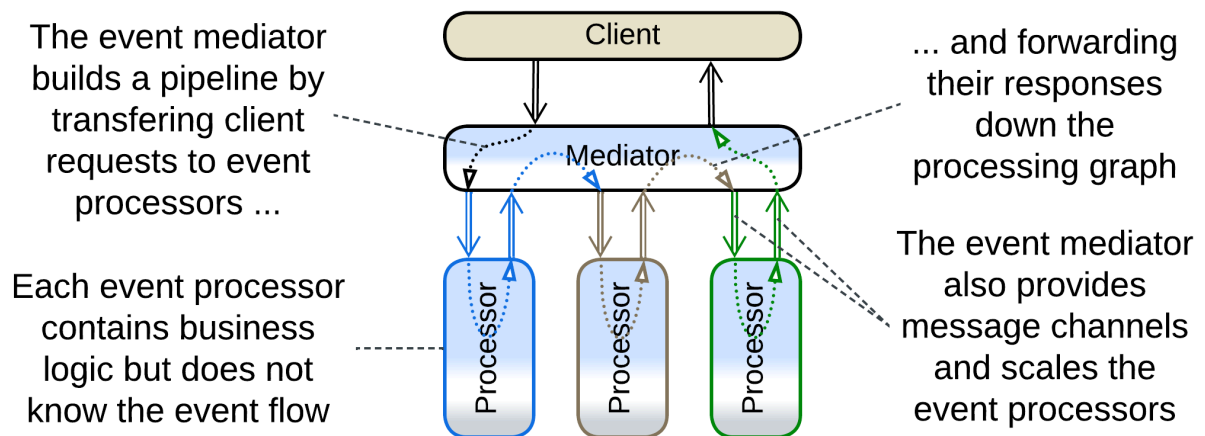
Event Mediator [FSA], pervading *Event-Driven Architectures* and *Nanoservices*, melds *Middleware* (used for delivery of messages) and *Orchestrator* (coordinating high-level use cases). Messages arrive to services and are responded to without any explicit component on the other side – they appear "out of thin *middleware*" which has incorporated the bulk of high-level business logic.

Slightly more details on *Event Mediator* are provided in the *Orchestrator* chapter.

Enterprise Service Bus (ESB)

An enterprise service bus mediates all calls and messages between services

The ESB also provides a transport for messaging and implements cross-cutting concerns like logging

A service posts its request to the ESB which knows how to handle it and which services are involved

It employs an adapter per service so that the services may vary in protocols they use

All that makes ESB a complex component which may become a development bottleneck

Enterprise Service Bus (*ESB*) [FSA] is a mixture of *Message Bus* and *Event Mediator*, *Enterprise Service Bus* blends *Middleware* and Orchestrator and adds an adapter per service as a topping. It was used to connect components that originated in incompatible networks of organizations that had been acquired by a corporation.

See the chapter about *Service-Oriented Architecture*.

Evolutions

A *middleware* is unlikely to be removed (though it may be replaced) once it is built into a system. There are few evolutions as a middleware is a 3rd party product and is unlikely to be messed with:

- If the middleware in use does not fit the preferred mode of communication between some of your services, there is an option to deploy a second specialized *middleware*.

Multiple specialized middlewares may be employed
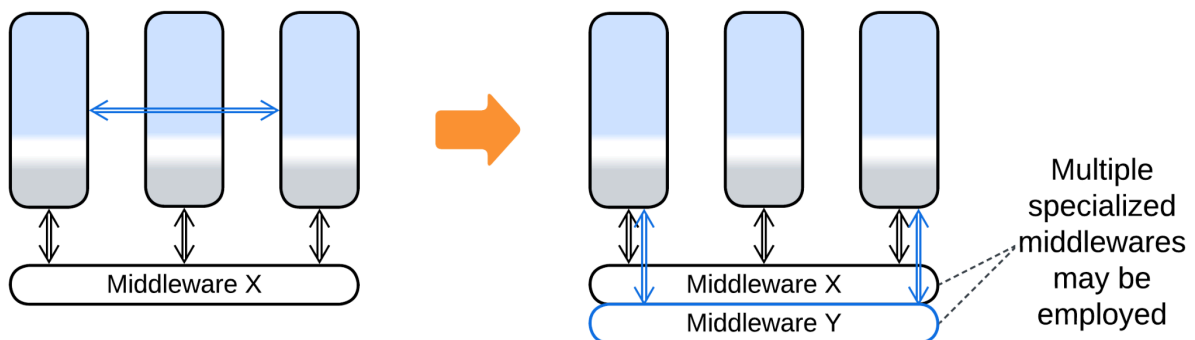
- If several existing systems need to be merged, that is accomplished by adding yet another layer of *middleware*, resulting in a bottom-up hierarchy *(bus of buses)*.
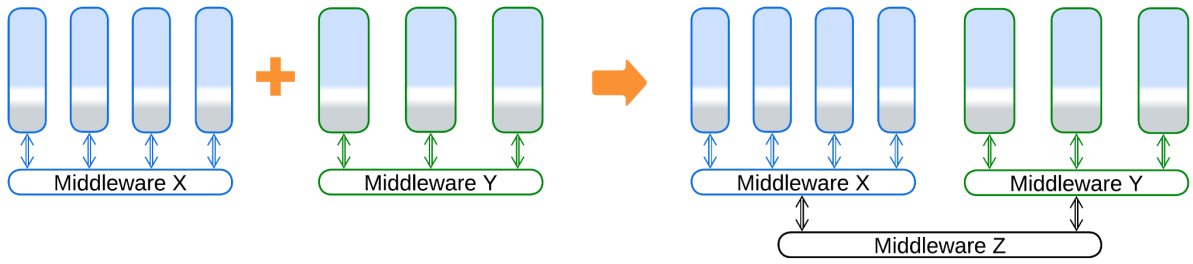
Summary

Middleware is a ready-to-use component that provides a system of services with means of communication, scalability and error recovery. It is very common in backend systems.

Shared Repository

Services with a Shared Repository
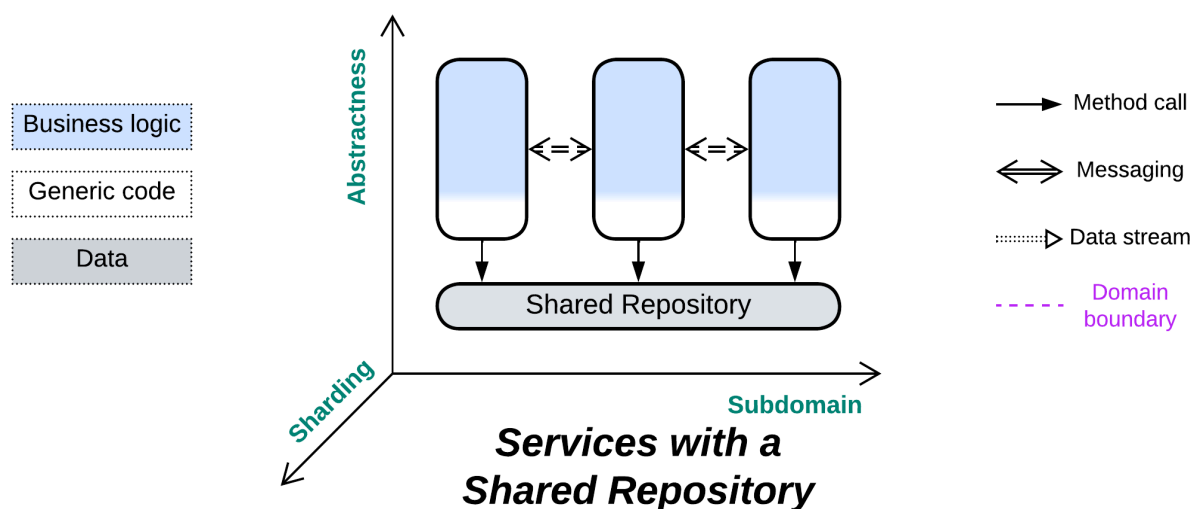
Knowledge itself is power. Sharing data is simple (& stupid).

Known as: Shared Repository [POSA4].

Aspects:
- Data storage,
- Data consistency,
- Data change notifications.

Variants:
- Shared Database [EIP] / Data Domain [SAHP] / Service-Based Architecture [FSA],
- Blackboard [POSA1, POSA4],
- Data Grid of Space-Based Architecture [SAP, FSA] / Replicated Cache [SAHP],
- Shared memory,
- Shared file system,
- (inexact) Stamp Coupling [SAHP].

Structure: A layer of data shared among higher-level components.

Type: Extension for *Services* or *Shards*.

Benefits	Drawbacks
Implements data access and synchronization concerns, thus eliminates data views and sagas	A single point of failure
Helps saving on hardware, licenses, traffic and administration	All the services depend on the schema of the shared database
Quick start of a project	Limits scalability
	A single database may not fit the needs of all the services equally well

References: [DDIA] is all about databases; [FSA] chapters on *Service-Based Architecture* and *Space-Based Architecture*.

Shared Repository builds communication in the system around its data. This is natural for data-centric domains and multiple instances of a stateless service, but may also greatly simplify development of ordinary services that need to exchange data. It covers the following concerns:

- Storing the entire domain data.
- Keeping the data self-consistent by providing atomic transactions for use by the application code.
- Communication between the services (if the repository supports notifications on data change).

The drawbacks are the extensive coupling (it's hard to alter a thing which is used throughout the entire system) and limited scalability (even a distributed database strives against distributed locks and the need to keep its nodes' data in sync).

Performance

A shared database with consistency guarantees (ACID) is likely to lower the total resource consumption compared to a database per service (as the services don't need to implement and keep updated materialized views [DDIA, MP] of other services' data) but increase latency and it may become the system's performance bottleneck. Moreover, the services lose the ability to use database technologies which best fit their tasks and data.

Another danger lies with locking records inside the database. Different services may use different order of tables in transactions, hitting deadlocks in the database engine which show up as transaction timeouts.

Non-transactional distributed databases may be very fast when colocated with the services (see *Space-Based Architecture*) but the resource consumption becomes very high because of the data duplication (each instance of each service gets a copy of the entire dataset) and simultaneous writes may corrupt the data (cause inconsistencies or merge conflicts).

Dependencies

Normally, each service depends on the repository. If the repository does not provide notifications on data changes, the services may need to communicate directly, in which case they also depend on each other through choreography or mutual orchestration.
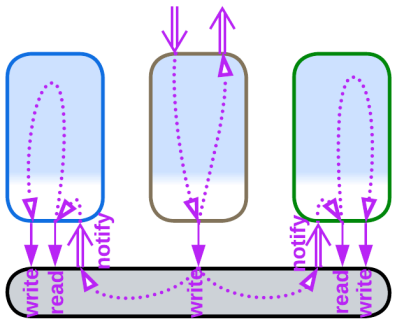
If the repository supports notifications, the services are independent
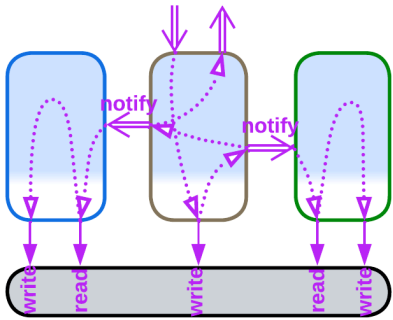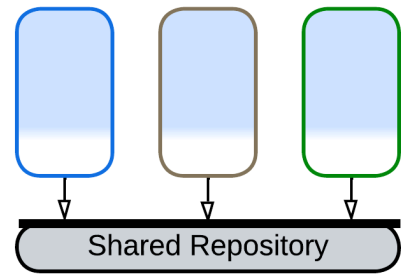
Otherwise a service depends on the services it subscribes to
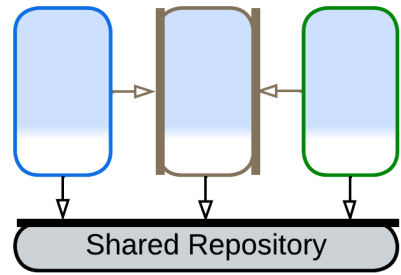
The dependency on the repository technology and data schema is dangerous for long-running projects as both of them may need to change sooner or later. Decoupling of the code from the data storage is done with yet another layer of indirection which is called *Database Abstraction Layer* (DAL) / *Database Access Layer* [POSA4] / *Data Mapper* [PEAA]. The DAL, which translates between the data schema and database's API on one side and the business logic's SPI on the other side, may reside inside each service or wrap the database:
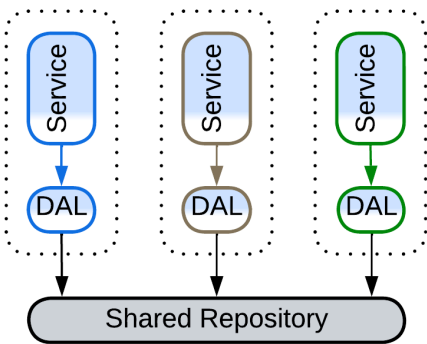
Each service may have its own database abstraction layer (DAL)
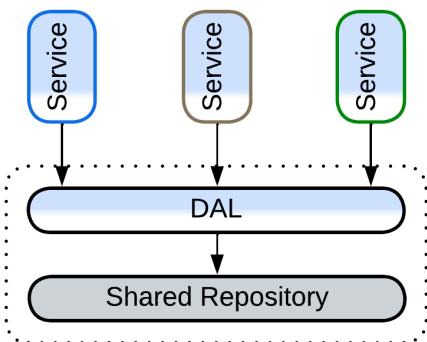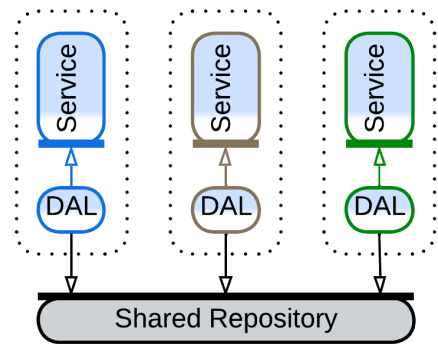
Or there may be a system-wide DAL co-located with the database
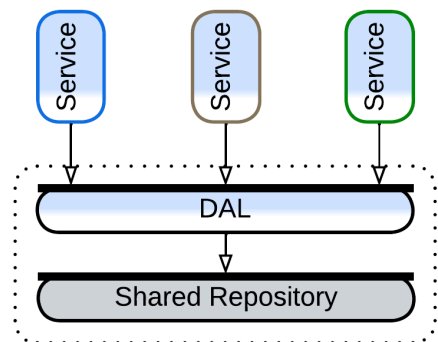
However, the DAL does not remove the shared dependencies, it only adds some flexibility. It seems that there is a peculiar kind of coupling through a shared component: in

our case one of the services may need to change the database schema or technology to better suit its needs, but is unable to do that because other components rely on (and profit from) the old schema and technology. Even deploying a second database, private to the service, is often not an option, as there is no convenient way to keep the databases in sync.

Applicability

Shared Repository is <u>good</u> for:
- *Data-centric domains.* If most of your domain's data is used in every subdomain, keeping any part of it private to a single subdomain service will be a pain in the system design. Examples include ticket reservation and even the minesweeper game.
- *A scalable service.* When you run several *instances* of a service, like in *Microservices*, the instances are likely to be identical and stateless, with the service's data being pushed out to a database which is accessed by all the instances.
- *Huge datasets*. Sometimes you may need to store a lot of data. It is unwise (meaning expensive) to stream and replicate it between your services just for the sake of ensuring their isolation. Share it. If the data does not fit in an ordinary database, some kind of *Space-Based Architecture* (which was invented to this end) may become your friend of choice.
- *Quick simple projects.* Don't over-engineer if you won't live long enough to need the flexibility. You may also save a buck or two on the storage.

Shared Repository is <u>bad</u> for:
- *Quickly evolving complex projects.* As everything flows, you just cannot devise a stable schema, while changing the database schema breaks all the services.
- *Varied forces and algorithms*. Different services may require different kinds of databases to work efficiently.
- *Big data with random writes*. Your data does not fit on a single server. If you want to avoid write conflicts, you must keep all the database nodes synchronized, which kills performance. If you let them loose async, you get collisions. You may want to first decouple and shard the data as much as possible, then turn your attention to esoteric databases, specialized caches and even tailor-made *middleware* to get out of the trouble.

Relations

Shared repository for services

Shared repository for shards

Shared repository for a SOA
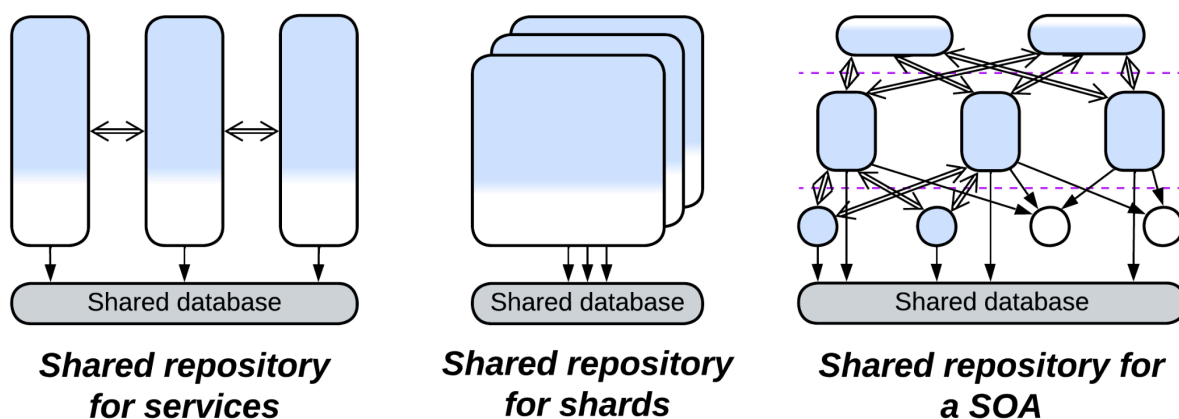
Shared Repository:

- Extends *Services*, *Service-Oriented Architecture*, *Shards* or occasionally *Layers*.
- Is a part of a persistent *middleware* or *Nanoservices*.
- Is closely related to *Middleware*.
- May be implemented by a *mesh*.

Variants

Shared Repository is a sibling of *Middleware*. While a *middleware* assists direct communication among services, a *shared repository* grants them indirect communication through access to an external state which usually stores all the data of the domain.

A *shared repository* may provide a generic interface (e.g. SQL) or a custom API (with a domain-aware *adapter* / ORM over the database). The *repository* can be anything ranging from a trivial OS file system or a memory block accessible from all the components to an ordinary database to a *mesh*-based distributed tuple space:

Shared Database, Data Domain, Service-Based Architecture

The services share data through the database

They may also communicate directly
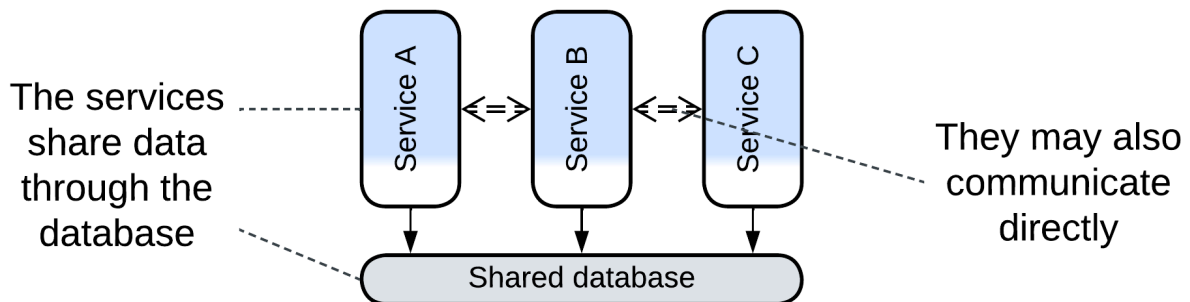
Shared Database [EIP] or *Data Domain* [SAHP] is a single database available to several *services*. The services may subscribe to data change triggers in the database or directly notify each other about domain events. The latter is often the case with *Service-Based Architecture* [FSA] which consists of large services dedicated to subdomains.

Blackboard

Individual algorithms that work towards finding a solution

The scheduler selects the best algorithm to run ...

A storage with shared knowledge: inputs and outputs of calculations

... based on the currently known results
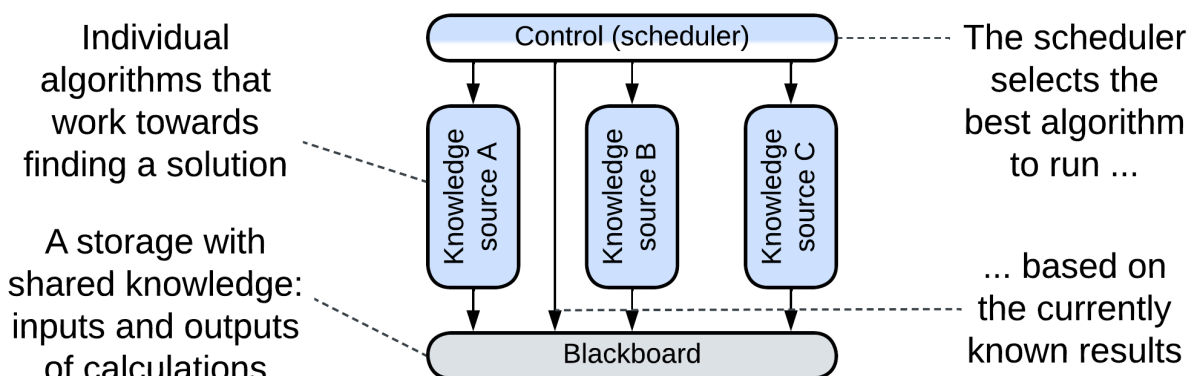
Blackboard [POSA1, POSA4] was used for non-deterministic calculations where several algorithms were concurring and collaborating to gradually build a solution from incomplete inputs. The *control* (*orchestrator*) component schedules the work of several *knowledge sources* (*services*) which encapsulate algorithms for processing the data stored in the

blackboard (*shared repository*). The approach has likely been superseded by convolutional neural networks.

Example: several use cases are mentioned on Wikipedia.

Data Grid of Space-Based Architecture (SBA), Replicated Cache
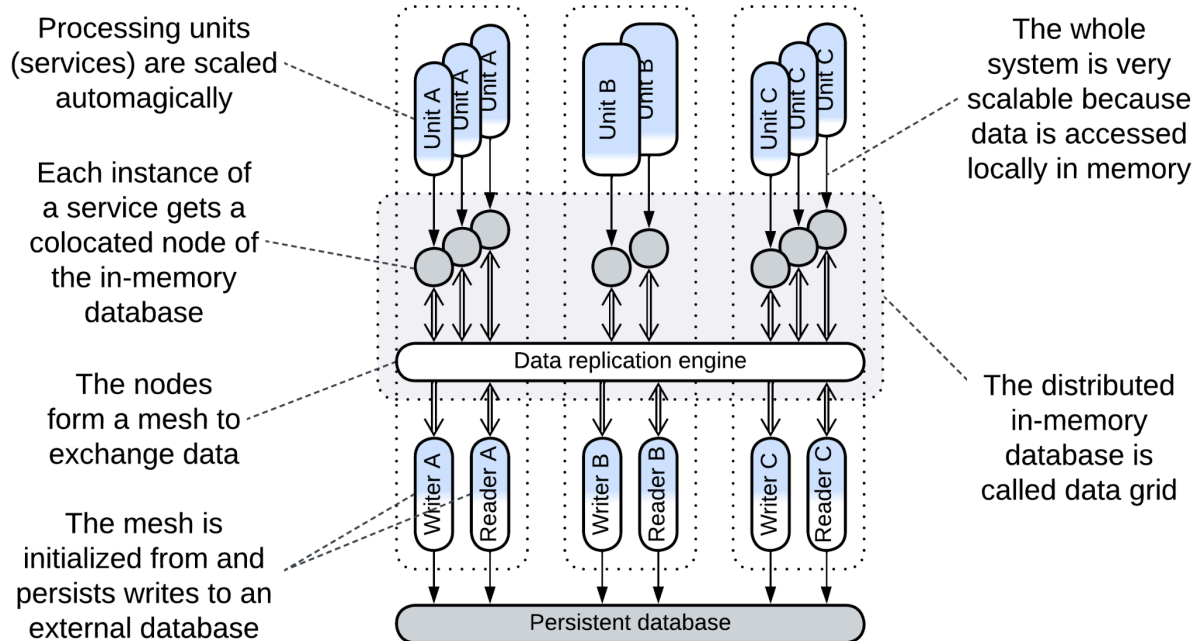
Space-Based Architecture [SAP, FSA] is a *service mesh* (*mesh*-based *middleware* with a *proxy* per service instance) that also implements a tuple space (shared dictionary). It may not provide a full-featured database interface, but has very good performance, elasticity and fault tolerance, while some implementations may allow for dealing with datasets which are much larger than anything digestible by ordinary databases. Its drawbacks include write collisions and costs (huge traffic for data replication and lots of RAM to store the replicas).

The main components of the architecture are:

- *Processing Units* – the services that contain the business logic. There may be one class of *processing units*, making *SBA* look like *Shards* or multiple classes in which case *SBA* becomes similar to *Microservices* with a shared database.
- *Data Grid* (*Replicated Cache* [SAHP]) – a *mesh*-based in-memory database. Each node of the *data grid* is co-located with a single instance of a *processing unit*, providing it with very fast access to the data stored. Changes to the data are replicated across the *grid* by its virtual *data replication engine* which is usually implemented by every node of the grid.
- *Persistent Database* – an external database which the data grid replicates. Its schema is encapsulated in the *readers* and *writers*.
- *Data Readers* – components that query the persistent database for any data which is not currently available in the *data grid*. Most cases see *readers* employed on starting the system to read the entire contents of the database into the memory of the nodes.
- *Data Writers* – components that replicate changes done in the *data grid* to the persistent storage to assure that no updates are lost if the system is shut down. There can be a pair of *reader* and *writer* per class of *processing units* (subdomain) or a global pair that processes all read and write requests.

This architecture provides nearly perfect scalability (high read and write traffic) and elasticity (new instances of processing units are created and initialized quickly as they replicate the data from already running units with no need to access the external database). For smaller datasets the entire database is replicated to every node of the grid. Still, *Space-Based Architecture* also allows to process datasets that don't fit in memory of a single node by assigning each node a part of the dataset.

The drawbacks of this architecture are:

- Structural and operational complexity.
- Somewhat inconvenient interface of the tuple space (no joins or other complex operations).
- High traffic for data replication among the nodes.
- Unavoidable data collisions when multiple users change the same piece of data simultaneously.

Shared memory

The sender appends (pushes) its messages to a shared memory buffer

The receiver reads and deletes (pops) the messages

If we abstract a little bit we see that ...

... each buffer makes a communication channel

Shared memory (ring buffers)
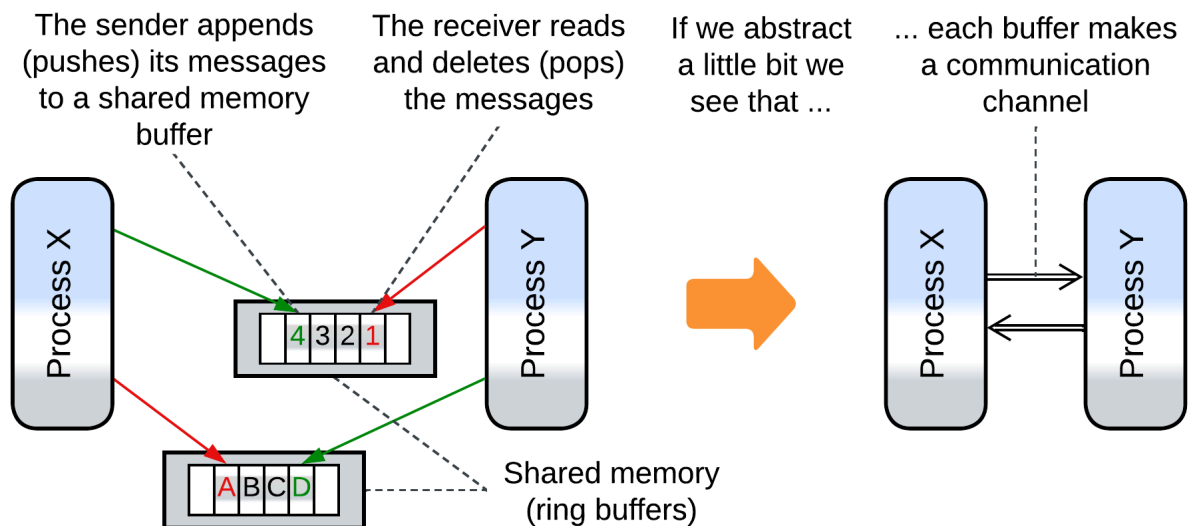
Several actors (processes, modules, device drivers) communicate through one or more mutually accessible data structures (arrays, trees or dictionaries). Accessing a shared object may require some kind of synchronization (e.g. taking a mutex) if it is not based on atomic variables. Notwithstanding that communication via shared memory is the archenemy of (shared-nothing) messaging it is actually used to implement messaging: high-load multi-process systems (e.g. web browsers) rely on shared memory *mailboxes* for messaging between their constituent processes.

Shared file system

The input is provided as a batch of files

Each step of the algorithm reads a file, processes it and writes a new file

As soon as a file has been written the next step takes it for input

In the end producing a batch of output files that have passed through all the steps ...
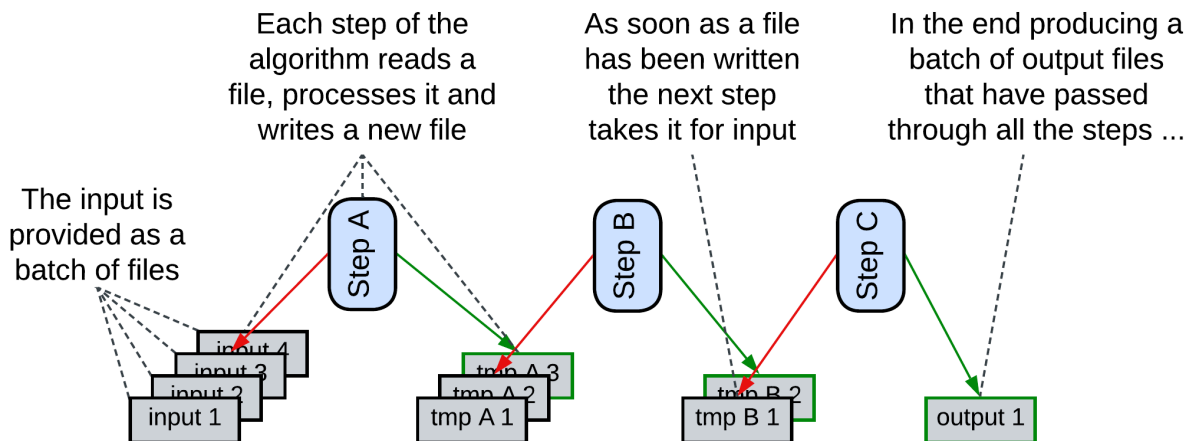
... which make the pipeline
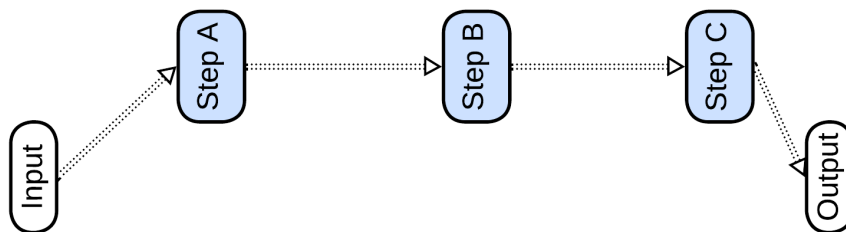
As a file system is a kind of shared dictionary, writing and reading files can be used to transfer data between applications. A data processing *pipeline* that stores intermediate results in files benefits from the ability to restart its calculation from the last successful step as the files are persistent [DDIA].

(inexact) Stamp Coupling

A message collects data from the services it passes

This looks like if the services were writing to a shared object
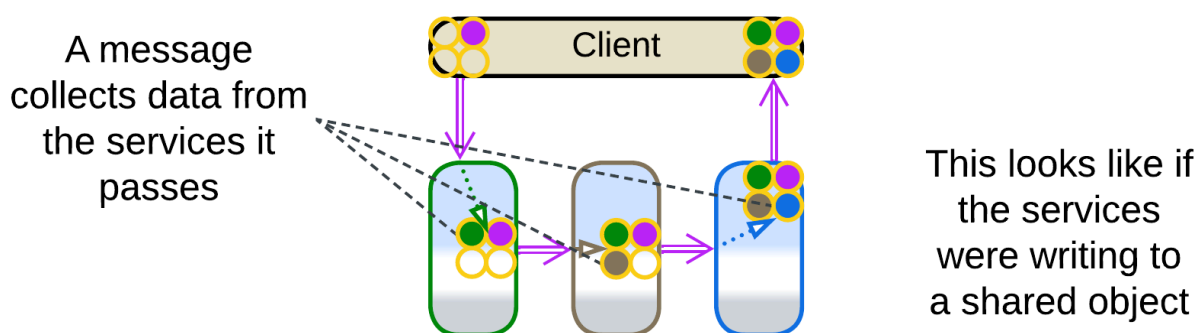
Stamp Coupling [SAHP] happens when a single data structure passes through the entire *pipeline*, with separate fields of the data structure targeting individual processing steps.

A *choreographed* system with no shared databases does not provide any way to aggregate related data spread over multiple services. If we need to collect everything known about a user or a purchase, we pass a query message through the system, and every service appends to it whatever it knows about the subject (just like post offices add their *stamps* to a letter). The unified message becomes a kind of *virtual shared repository* the

services (*Content Enrichers* according to [EIP]) write to. This also manifests in the dependencies: all the services depend on the format of the query message, as they would on the schema of a shared repository, instead of depending on each other, as is usual with *pipelines*.

Evolutions

Once a database appears, it is unlikely to go away. I see the following evolutions to improve performance of the data layer:

● Shard the database.

Sometimes data can be sharded to multiple instances of the database

● Use *Space-Based Architecture* for dynamic scalability.

Each service is co-located with an instance of a shared in-memory database

The instances intercommunicate, forming a mesh

● Divide the data into a private database per service.

Services are more independent when each of them owns its data

● Deploy specialized databases (*Polyglot Persistence*).

Service A ⟷ Service B ⟷ Service C

Shared Database

Service A ⟷ Service B ⟷ Service C

DB X   DB Y

Specialized databases have good performance if used appropriately
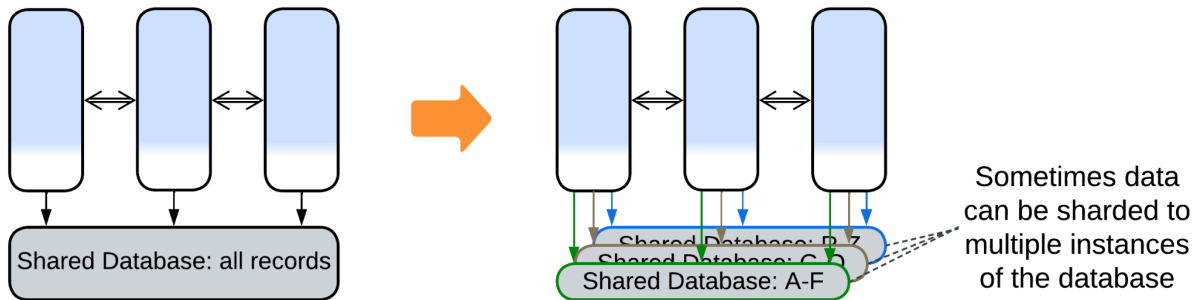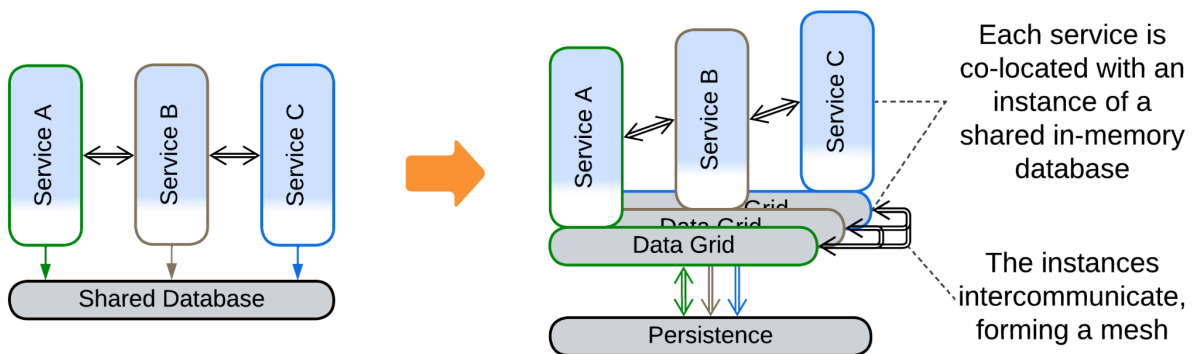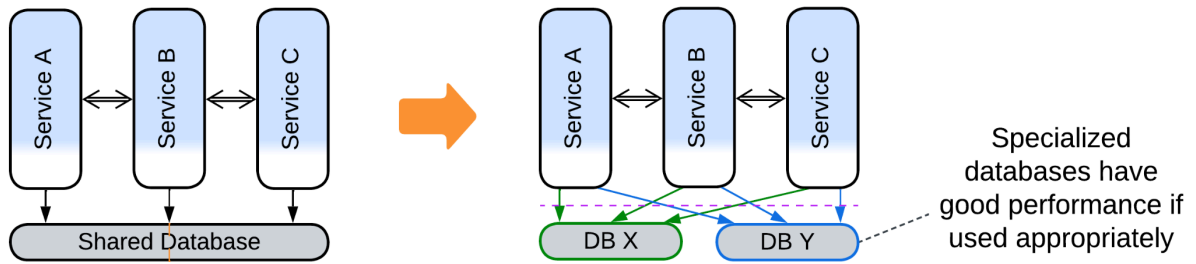
Summary

Shared Repository aids to implement a system of services quickly by simplifying their interactions at the cost of freezing its data model and possibly limiting its performance.

Proxy

Services with a Proxy
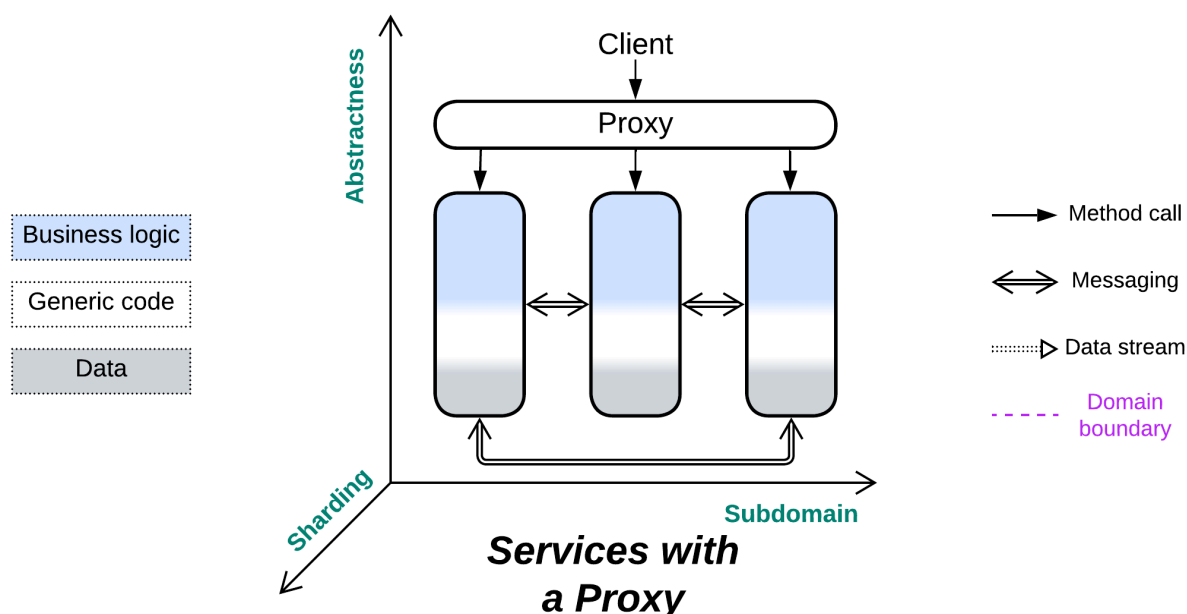
Should I build the wall? A layer of indirection between your system and its clients.
Known as: Proxy [GoF].
Aspects:
● Routing,
● Offloading.
Variants:
By transparency:
● Full Proxy,
● Half-Proxy.
By placement:
● Separate deployment,
● On the system side: Sidecar [DDS],
● On the client side: Ambassador [DDS].
By function:
● Firewall,
● Response Cache / Read-Through Cache / Write-Through Cache / Write-Behind Cache / Cache [DDS],
● Load Balancer [DDS] / Sharding Proxy [DDS] / Cell Router / Messaging Grid [FSA] / Scheduler,
● Dispatcher [POSA1] / Reverse Proxy / Ingress Controller / Edge Service / Microgateway,
● Adapter [GoF, DDS] / Anticorruption Layer [DDD] / Open Host Service [DDD] / Gateway [PEAA] / Message Translator [EIP, POSA4] / API Service / Cell Gateway / (inexact) Backend for Frontend / Hardware Abstraction Layer (HAL) / Operating System Abstraction Layer (OSAL) / Platform Abstraction Layer (PAL) / Database abstraction layer (DBAL or DAL) / Repository [PEAA, DDD].
● (with *Orchestrator*) API Gateway [MP].
See also *Backends for Frontends* (multiple *API Gateways*).

Structure: A layer that pre-processes and routes user requests.
Type: Extension.

Benefits	Drawbacks
Separates cross-cutting concerns (such as auth, logging, client protocols) from the services	A single point of failure
Decouples the system from its clients	Most proxies degrade latency
Low attack surface	
Available off the shelf	

References: Half of [DDS] is about the use of *proxies*. See also: [POSA4] on *Proxy*; Chris Richardson and Microsoft on *API Gateway*; Martin Fowler on *Gateway*, *Facade* and *API Gateway*.

A *proxy* stands between a (sub)system's implementation and its users. It receives a request from a client, does some pre-processing, then forwards the request to a lower-level component. In other words, a proxy takes care of some aspects of the system's communication with its clients by serving as yet another layer of indirection that collects common concerns which otherwise would have been intermixed with the business logic in the underlying components. It may also decouple the system internals from changes in the public protocol. The main functions of a proxy include:

- *Routing* – a proxy knows addresses of the instances of the system's components and is able to forward a client's request to a matching shard or a service that can handle it. Clients need to know only the public address of the proxy. A proxy may also respond on its own if the request is invalid or there is a matching response in the proxy's cache.
- *Offloading* – a proxy may implement generic aspects of the system's public interface, such as authentication, authorisation, encryption, request logging, web protocol support, etc. which otherwise would need to be implemented by each of the underlying services. That allows for the services to concentrate on what they are for – the business logic.

Performance

Most kinds of proxies trade latency (the extra network hop) for some other quality:

- *Firewall* slows down the processing of good requests but *protects* the system from attacks.
- *Load Balancer* and *Dispatcher* allow for the use of multiple servers (with identical or specialized components, correspondingly) to improve the system's *throughput* but they still degrade the minimum latency.
- *Adapter* adds *compatibility* but its latency cost is higher than with other *proxies* as it not only forwards the original message but changes the payload, which involves data processing and serialization.

Cache is a bit weird in that aspect. It improves latency and throughput for repeated requests but degrades latency for unique ones. Furthermore, it is often colocated with some other kind of proxy to avoid the extra network hop between the proxies, which makes caching almost free in terms of latency.

Dependencies

Proxies widely vary in their functionality and level of intrusiveness. The most generic proxies, like *firewalls*, may not know anything about the system or its clients. A *response cache* or *adapter* must parse the messages, thus it depends on the communication protocol and message format. A *load balancer* or *dispatcher* is aware of both the protocol and system composition.

In fact, proxies tend to have those dependencies configured on startup or through their APIs, thus there is no need to change the code of the proxy each time something changes in the underlying system.

Applicability

Proxy helps:

- *Multi-component systems.* Having multiple types and/or instances of services means there is a need to know the components' addresses to access them. A proxy encapsulates that knowledge and may also provide other user-facing functionality as an extra benefit.
- *Dynamic scaling or sharding.* The proxy both knows the system structure (which instances of services exist) and delivers user requests, thus it is the place to implement sharding (when a service instance is dedicated to a subset of users) or load balancing (when any service instance can serve any user) and even manage the size of the service instance pool.
- *Multiple client protocols.* The proxy as the endpoint for the system's users may translate multiple external (user-facing) protocols into a unified internal representation.
- *System security.* Though a proxy does not make the system more secure, it takes away the burden of security considerations from the services which implement business logic, improving the separation of concerns and making system components more simple and stupid. An off-the-shelf proxy may be less vulnerable compared to in-house services.

Proxy hurts:

- *Critical real-time paths.* It adds an extra hop in request processing, increasing latency for thoroughly optimized use cases. Such requests may need to bypass the proxy.

Relations

Proxy for a monolith　　**Proxy for shards**　　**Proxy for layers**　　**Proxy for services**
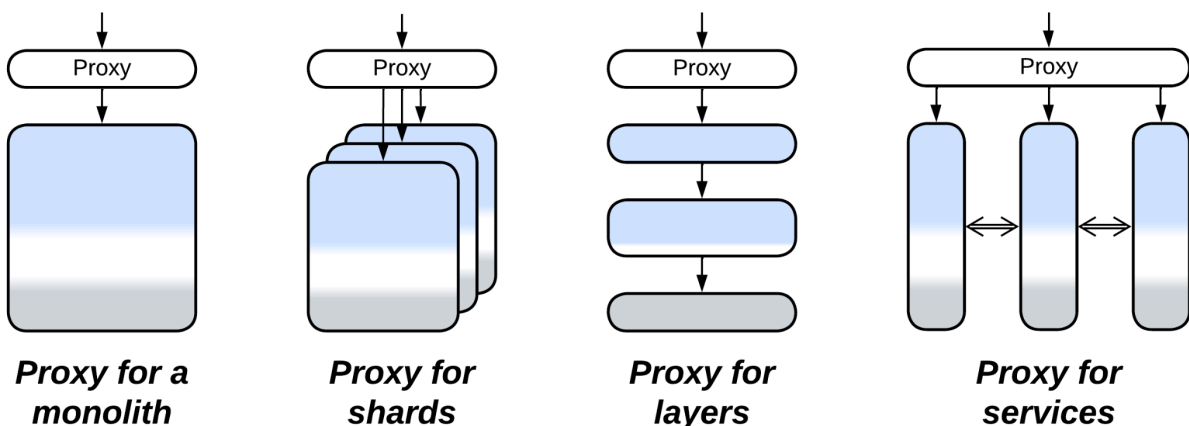
Proxy:

- Extends *Monolith* or *Layers* (forming *Layers*), *Shards* or *Services*.
- Can be extended with another *proxy* or merged with an *orchestrator* into an *API gateway*.
- A *gateway* per *service* is employed by *Service Mesh*, *Enterprise Service Bus* and *Hexagonal Architecture*.
- Is a special case (single user-facing service) of *Backends for Frontends*.

Variants by transparency

A proxy may either fully isolate the system it represents or merely help establish connections between clients and servers. This matches closed and open *layers* as a *proxy* is a *layer* between a system and its clients.

Full Proxy
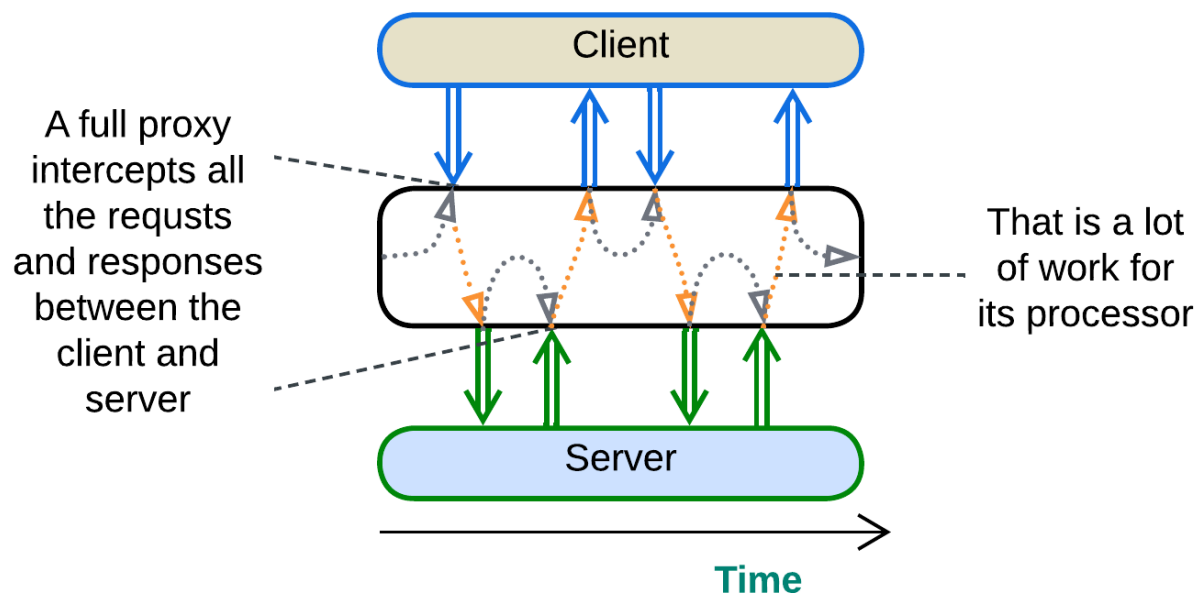
A *full proxy* processes every message between the system and its clients. It completely isolates the system and may meddle with the protocols but is resource-heavy and impairs latency. *Adapters* and *response caches* are always full proxies.

A half proxy intercepts only the session establishment request

Other messages pass through it unhindered

Thus its processor is not loaded

And the client-server communication is faster

Client

Server

Time
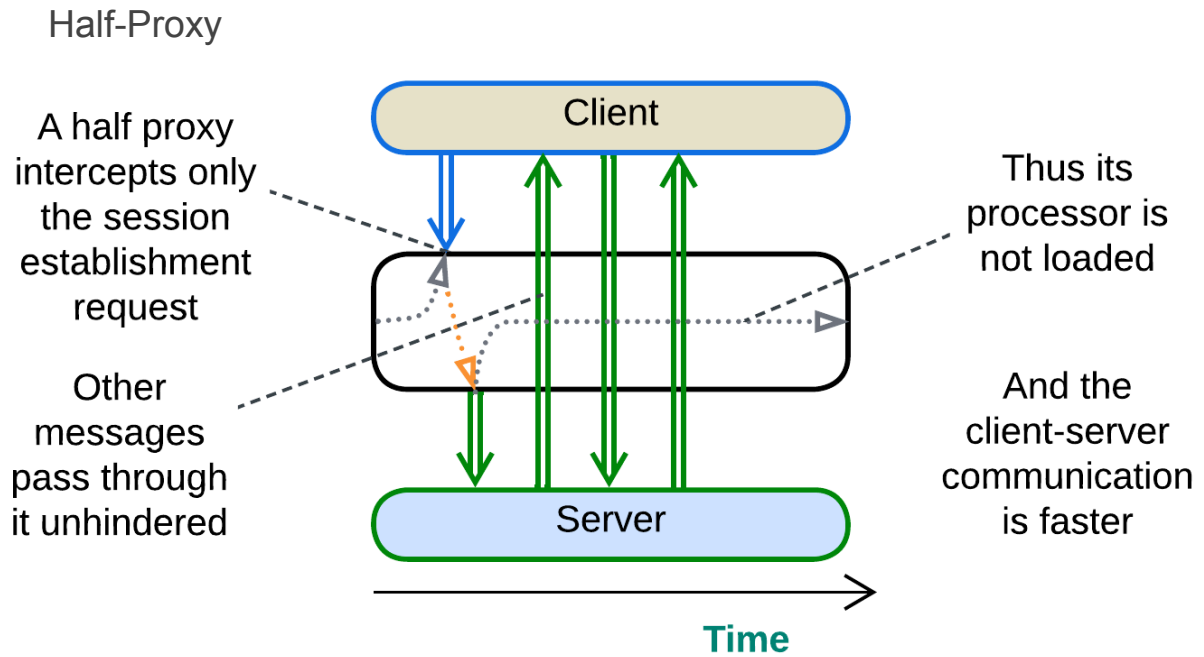
A *half-proxy* intercepts, analyzes and routes the session establishment request form a client but then goes out of the loop. It may still forward the subsequent messages without looking into their content or even help connect the client and server directly, which is known as *Direct Server Return* (DSR). This approach is faster and much less resource-hungry but also less secure and flexible than that of *Full Proxy*. A *firewall*, *load balancer* or *reverse proxy* may act as a *half-proxy*. IP telephony servers often use *DSR*: the server helps call parties find each other and establish direct media communication.

Variants by placement

As a *proxy* stands between a (sub)system and its client(s), we can imagine a few ways to deploy it:

Separate deployment: Standalone

A separately deployed proxy adds an extra network hop to each client request

If the client accessed the service directly, the latency would have been lower

Client

Standalone

Service   **Service**   **Service**
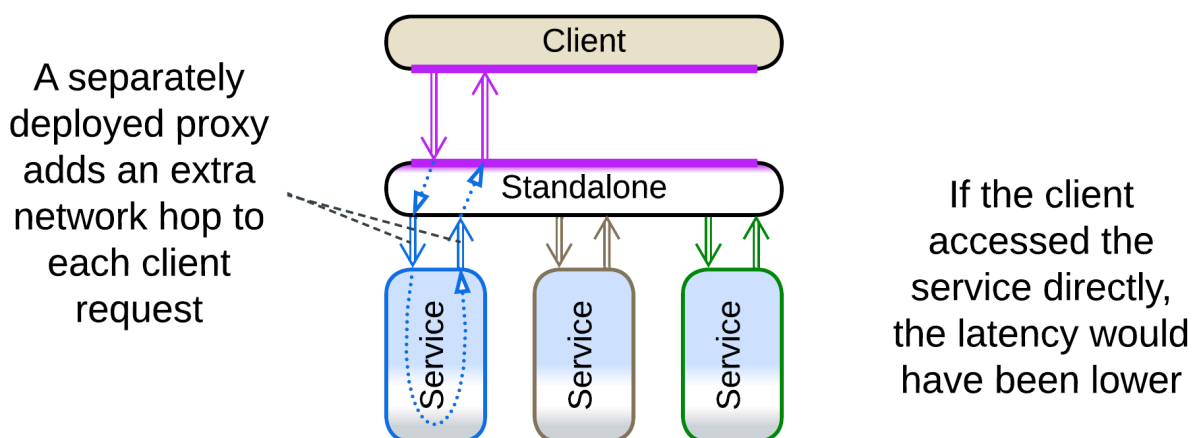
We can deploy a proxy as a separate system component. This has the downside of the extra network hop (higher latency) on the way of each client's request to the system and back but is unavoidable in the following cases:

- The proxy uses a lot of system resources, thus it cannot be colocated with another component. This mostly affects *Firewall* and *Cache*.
- The proxy is stateful and deals with multiple system components, as a *load balancer* for *stateless instances*, *reverse proxy* or *API gateway*.

On the system side: Sidecar

If the proxy is co-located with your service in a sidecar ...

... it has little effect on latency and does not add any traffic
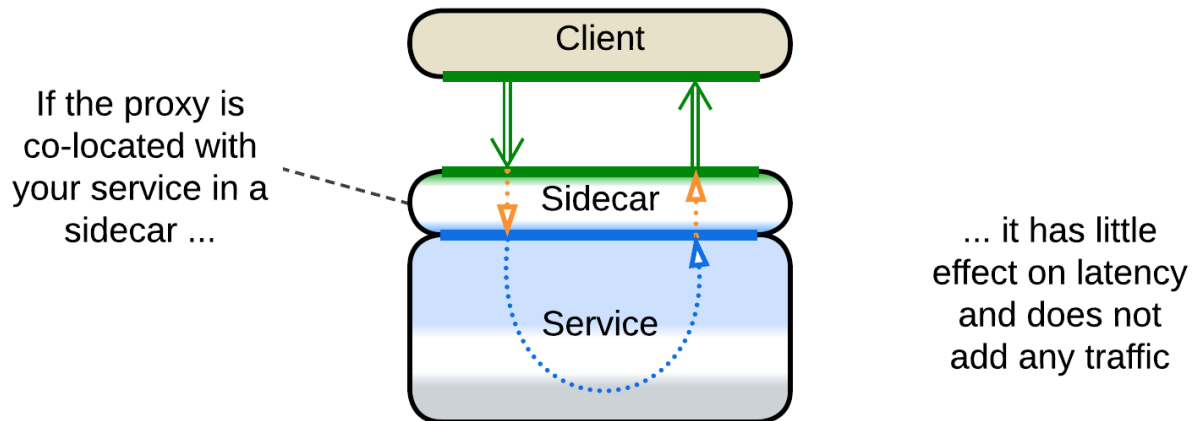
We can often co-locate a proxy with our system when it is not distributed. That avoids the extra network delay, traffic, operational complexity and does not add any new hardware only to fail at the most unlucky moment. Such a placement is called *Sidecar* [DDS] and is applicable to *adapters* and *reverse proxies*.

It should be noted that *Sidecar* – co-locating a generic component and business logic – is more of a DevOps approach than an architectural pattern, thus we can see it used in a variety of ways [DDS]:

- As a *proxy* between a component and its clients.
- As an extra *service* that provides observability or configures the main service.
- As a *layer* with general-purpose utilities.
- As an *adapter* for *middleware*.
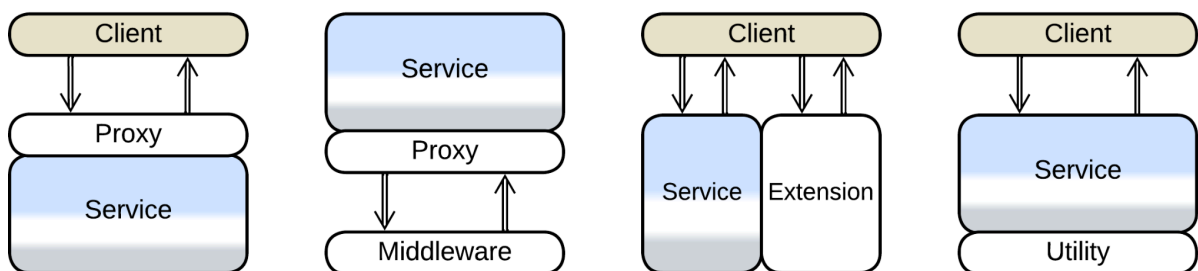
Service Mesh (*Middleware* for *Microservices*) makes heavy use of the last kind of *sidecars*.

On the client side: Ambassador

An ambassador
runs your code on
your client's side

Client

Ambassador

Service

It encapsulates
the knowledge of
your system
from your client's
application
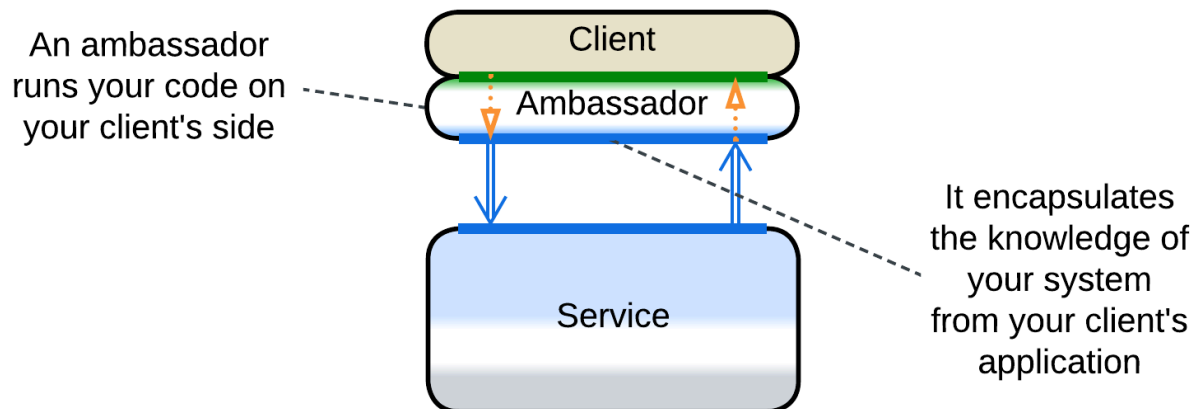
Finally, a *proxy* may be co-located with a component's clients, making an *ambassador* [DDS]. The use cases include:

- Low-latency systems with stateful *shards* – each client should access the *shard* that has their data, which the *proxy* knows how to choose.
- *Adapters* that help client applications use an optimized or secure protocol.

Variants by function

Proxies are ubiquitous in backend systems as using one or several of them frees the underlying code from the need to provide a boilerplate non-business-logic functionality. It is common to have several types of *proxies* deployed sequentially (e.g. *API gateways* behind *load balancers* behind a *firewall*) with many of them pooled to improve performance and stability. It is also possible to employ multiple types of *proxies*, each serving a dedicated kind of client, in parallel, resulting in *Backends for Frontends*.

As *proxies* are used for multiple purposes, there is a variety of their specializations and names. Below is a very rough categorization, complicated by the fact that many real-world *proxies* implement several categories at once.

Firewall

A firewall forwards valid requests to the system it protects ...

... while malicious or repetitive requests are ignored
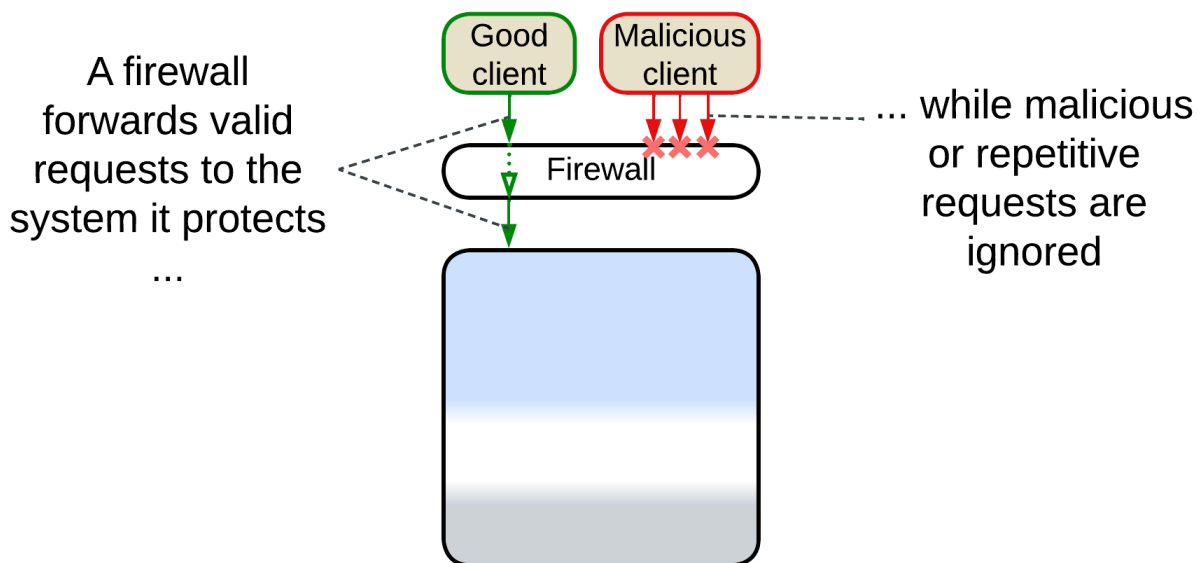
Firewall is a component for white- and black-listing network traffic, mostly to protect against attacks. It is possible to use both generic hardware firewalls on the external perimeter as the means for brute force (D)DoS protection and more complex access rules at a second layer of software firewalls that protects critical internal data and services from unauthorized access.

Response Cache, Read-Through Cache, Write-Through Cache, Write-Behind Cache, Cache

A cache forwards requests ...

... and remembers responses

If a request is repeated, it returns a copy of a stored response ...

... to reduce the load on the underlying system
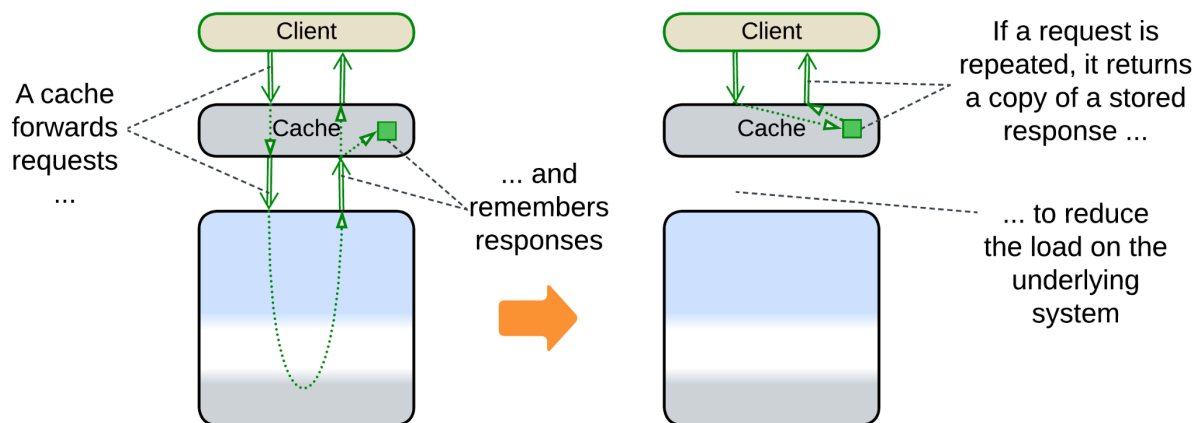
If the system often gets identical requests, it is possible to remember its responses to most frequent of them and return the cached response without fully re-processing the request. The real thing is more complicated because users tend to change the data which the system stores, necessitating a variety of cache refresh policies. A *response cache* may be co-located with a *load balancer* or it may be [DDS] *sharded* (each cache processes a unique subset of requests) and/or *replicated* (all the caches are similar) and thus require a *load balancer* of its own.

It is called *Response Cache* because it stores the system's responses to requests of its users or just *Cache* [DDS] because it is the most common kind of cache in system architecture.

If the cached subsystem is a database, we can discern between read and write requests:

- *Read-Through Cache* is when the cache is updated on a miss of a read request but is transparent to or invalidated by write requests.
- *Write-Through Cache* is when the cache is updated by write requests that pass through it.
- *Write-Behind* is when the cache aggregates multiple write requests to later send them to the database as a batch, saving bandwidth and possibly merging multiple updates of the same key.

Load Balancer, Sharding Proxy, Cell Router, Messaging Grid, Scheduler
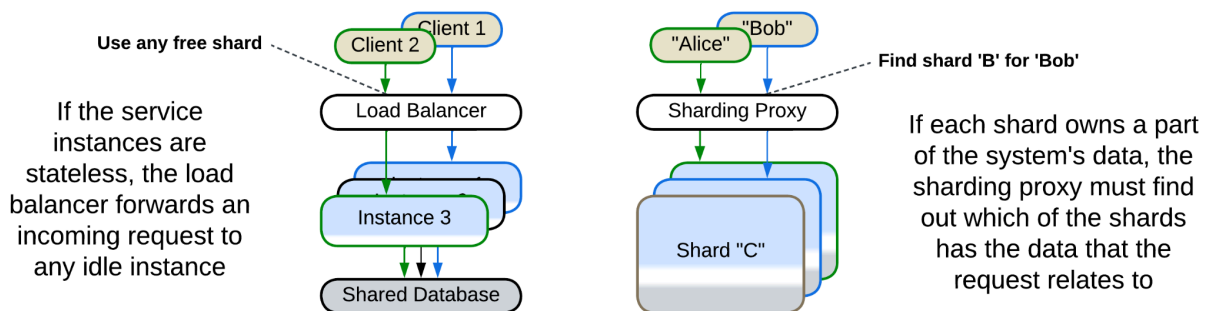
Here we have a hardware or software component which distributes user traffic among multiple *instances* of a service.

- A *sharding proxy* [DDS] selects a *shard* based on some data which is present in the request (OSI level 7 request routing) for a system where each shard owns a part of the system's state, thus only one or a few of the shards have the data required to process the request.
- A *load balancer* [DDS] of a *pool of instances* or *messaging grid* [FSA] of *Space-Based Architecture* evenly distributes the incoming traffic over identical request processors (OSI level 4 load balancing) to make sure that no instance of the underlying system is overloaded. In some cases it needs to be session-aware (process OSI level 7) to make sure that all the requests from a client are forwarded to the same instance of the service [DDS].
- It may forward read requests to read-only replicas of the data while write requests are sent to the master database (*CQRS*-like behavior).
- A *cell router* chooses a data center which is the closest to the user's location.

Load balancers are very common in high-load backends. High-availability systems deploy multiple instances of a *load balancer* in parallel to remain functional if one of the *load balancers* fails. CPU-intensive applications (like 3D games) often post asynchronous tasks for execution by *thread pools* under the supervision of a *scheduler*. A similar pattern is found in OS kernels and fiber or actor frameworks where a limited set of CPU-affined threads is scheduled to run a much larger number of tasks.

Dispatcher, Reverse Proxy, Ingress Controller, Edge Service, Microgateway

A dispatcher exposes an interface that combines the interfaces of the underlying services ...

Client

GetOrange()

Dispatcher

... and forwards each client request to the service that matches the called interface

Apples

Oranges
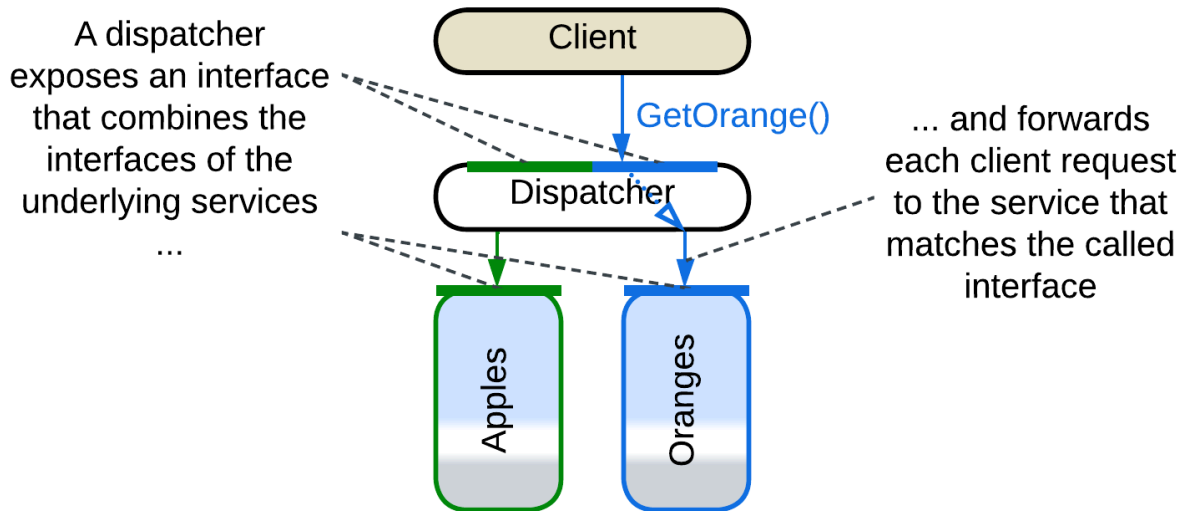
Reverse Proxy, Ingress Controller, *Edge Service* or *Microgateway* is a router that stands between the Internet and the organization's internal network. It allows clients to use a public address of the system without knowing how and where their requests are processed. It parses a user request and forwards it to an internal server based on the request's body. A *reverse proxy* can be extended with a *firewall*, *SSL termination*, *load balancing* and *caching* functionality. Examples include Nginx.

Dispatcher [POSA1] is a similar component for a single-process application. It serves a complex command line interface by receiving and preprocessing user commands only to forward each command to a module which knows how to handle it. The modules may register their commands with the *dispatcher* at startup or there may be a static dispatch table in the code.

You could have noted that *Dispatcher* or *Reverse Proxy* is quite similar to *Load Balancer* or *Sharding Proxy* – they differ mostly in what kind of system lies below them: *Services* or *Shards*.

Adapter, Anticorruption Layer, Open Host Service, Gateway, Message Translator, API Service, Cell Gateway, (inexact) Backend for Frontend, Hardware Abstraction Layer (HAL), Operating System Abstraction Layer (OSAL), Platform Abstraction Layer (PAL), Database abstraction layer (DBAL or DAL), Repository

An adapter translates data formats and logic between APIs or protocols

It may also provide such an extra functionality as SSL termination or logging
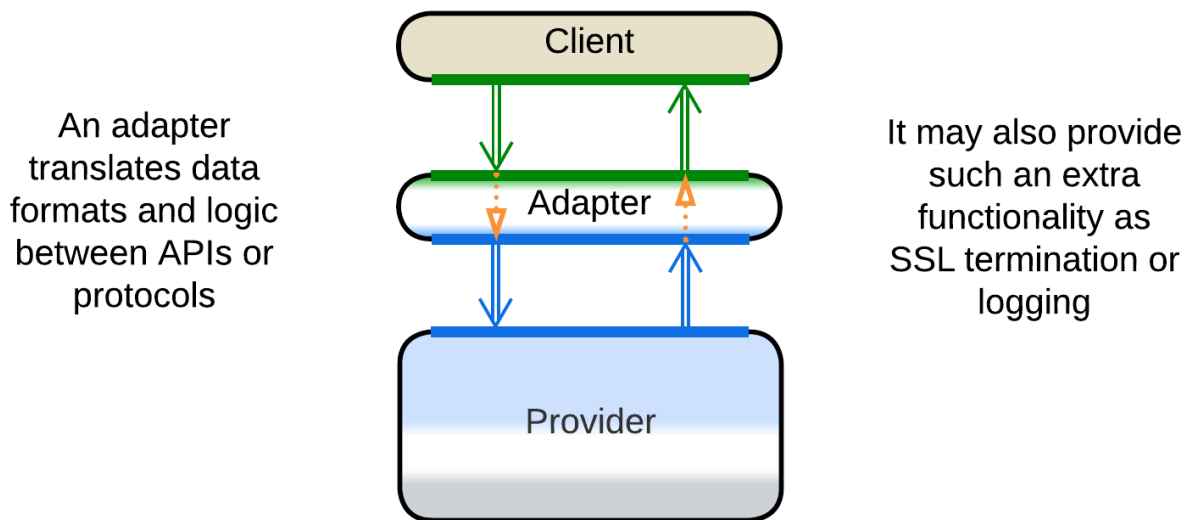
Adapter [GoF, DDS] is a mostly stateless *proxy* that translates between an internal and public protocol and API formats. It may be co-located with a *reverse proxy*. When it adapts messages, it may be called *Message Translator* [EIP, POSA4].

As an *adapter* adapts two ways, it is often found between components (in *Hexagonal Architecture*) or between a component and *middleware* (in *Enterprise Service Bus* and *Service Mesh*).

In [DDD], when one component (*consumer*) depends on another (*supplier*), there may be an adapter in-between to decouple them. It is called *Anticorruption Layer* [DDD] when owned by the *consumer*'s team or *Open Host Service* [DDD] if the *supplier* adds it to grant one or more stable interfaces (*Published Languages* [DDD]).

Gateway [PEAA] or *API Service* often implies an *adapter* with extra functionality, like *Reverse Proxy*, authorization and authentication. *Cell Gateway* is a gateway for a *cell*.

When a *gateway* translates a single public API method into several calls towards internal services, it becomes an *API gateway* [MP] which is an aggregate of *proxy* (for protocol translation) and *orchestrator*.

An *adapter* between an end-user client (web interface, mobile application, etc.) and the system's API is often called *Backend for Frontend*. It decouples the UI from the backend-owned system's API, giving the teams behind them freedom to work with little synchronization.

There is also a whole bunch of adapters that aim to protect the business logic from its environment, the idea which is perfected by *Hexagonal Architecture*:

- *Hardware Abstraction Layer* (*HAL*) hides details of hardware to make the code portable.
- *Operating System Abstraction Layer* (*OSAL*) or *Platform Abstraction Layer* (*PAL*) abstracts the OS to make the application cross-platform.
- *Database abstraction layer* (*DBAL* or *DAL*) attempts to help building database-agnostic applications by making all databases look the same.

- *Repository* [PEAA, DDD] provides methods to retrieve and save an object which is stored in a database.

API Gateway

An API gateway provides protocol translation and generic functionality like a gateway ...

... but also orchestrates the processing of requests like an API composer or process manager
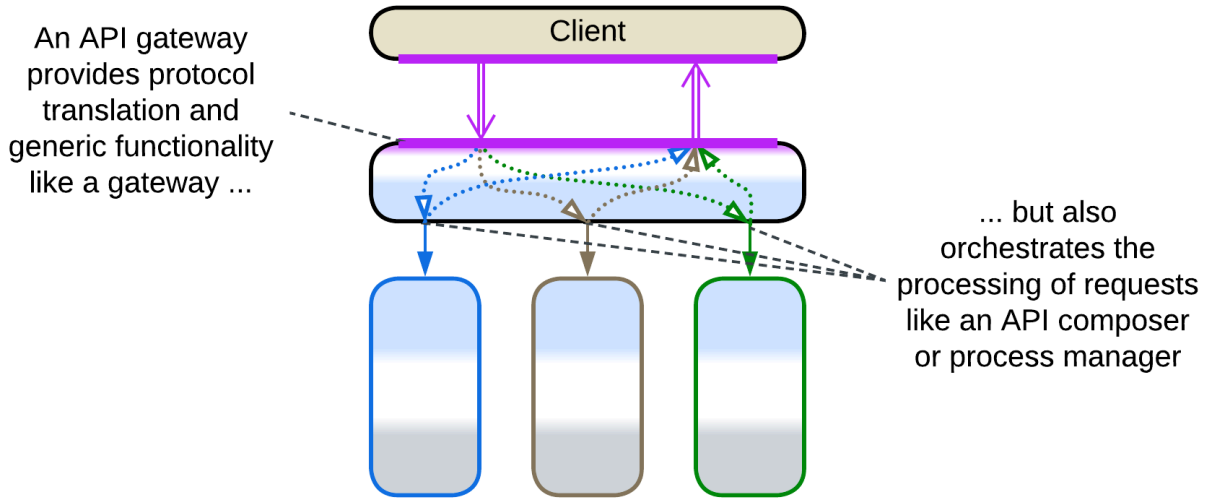
API Gateway [MP] is a fusion of *Gateway* (*Proxy*) and *API Composer* (*Orchestrator*). The *Gateway* aspect encapsulates the external (public) protocol while the *API Composer* translates the system's high-level public API methods into multiple (parallel or sequential) calls to the internal APIs of the component services, collects the results and conjoins them into a response.

API Gateway is discussed in more detail under *Orchestrator*.

Evolutions

It usually makes little sense to get rid of a *proxy* once it is integrated into a system. Its only real drawback is a slight increase in latency for user requests which may be helped through creation of bypass channels between the clients and a service that needs low latency. The other drawback of the pattern, the proxy's being a single point of failure, is countered by deploying multiple instances of the proxy.

As proxies are usually 3rd party products, there is very little we can change about them:
- We can add another kind of a proxy on top of the existing one.

Proxies are independent and can be stacked

- We can use a stack of proxies per client, making *Backends for Frontends*.
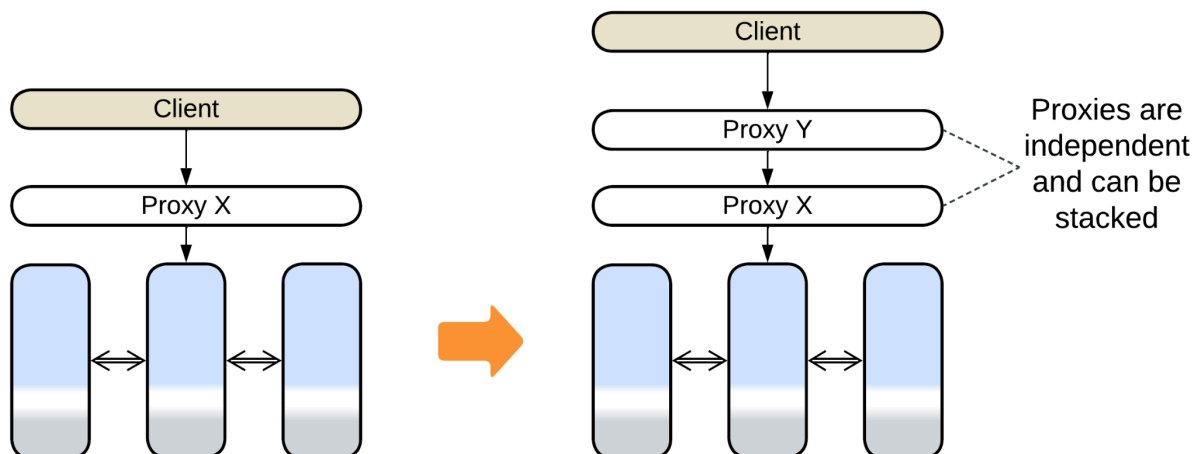
Each kind of a client may enjoy their own stack of proxies
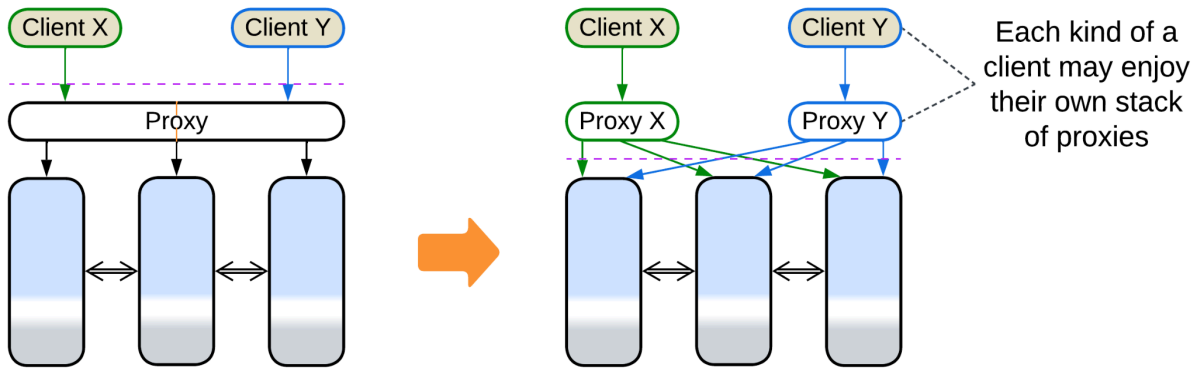
Summary

A *proxy* represents your system to its clients and takes care of some aspects of the communication. It is common to see multiple proxies deployed sequentially.

Orchestrator

Services with an Orchestrator
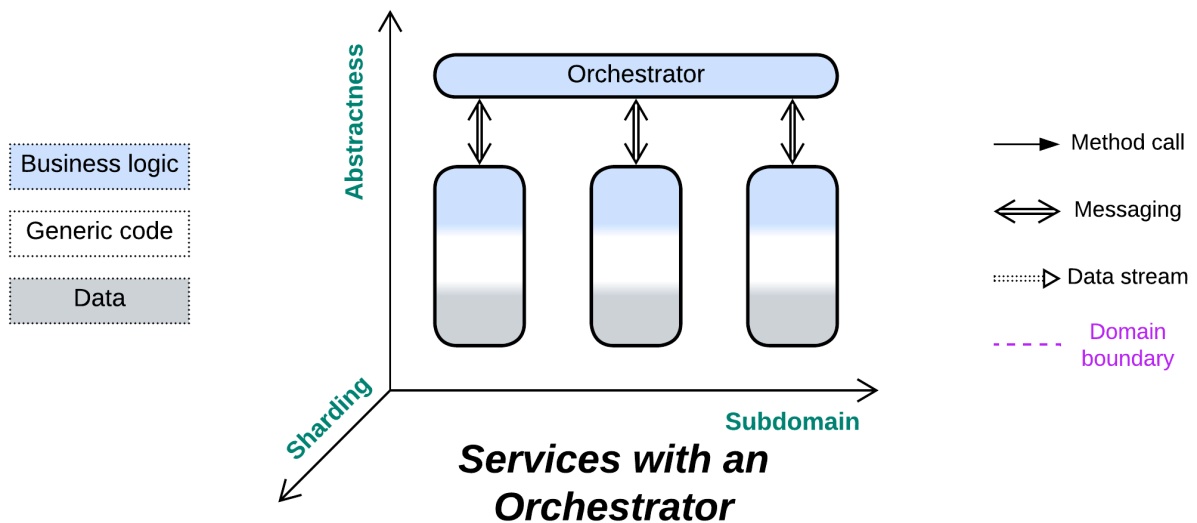
One ring to rule them all. Make a service to integrate other services.

Known as: Orchestrator [MP, FSA], Workflow Owner [FSA] of Microservices, Service Layer [PEAA], Application Layer [DDD], Wrapper Facade [POSA4], Multi-Worker [DDS], Controller / Control, Processing Grid [FSA] of Space-Based Architecture.

Aspects:
- Mediator [GoF, SAHP],
- Facade [GoF].

Variants:

By transparency:
- Closed or strict,
- Open or relaxed.

By structure (not exclusive):
- Monolithic,
- Sharded,
- Layered [FSA],
- A service per client type (*Backends for Frontends*),
- A service per subdomain [FSA] (*Hierarchy*),
- A service per use case [SAHP] (*SOA*-style).

By function:
- API Composer [MP] / Remote Facade [PEAA] / Gateway Aggregation / Composed Message Processor [EIP] / Scatter-Gather [EIP, DDS] / MapReduce [DDS],
- Process Manager [EIP, LDDD] / Orchestrator [FSA],
- (Orchestrated) Saga [LDDD] / Saga Orchestrator [MP] / Saga Execution Component / Transaction Script [PEAA, LDDD] / Coordinator [POSA3],
- Integration (Micro-)Service / Application Service,
- (with *Gateway*) API Gateway [MP] / Microgateway,
- (with *Middleware*) Event Mediator [FSA],
- (with *Middleware* and *Adapters*) Enterprise Service Bus (ESB) [FSA].

Structure: A layer of high-level business logic built on top of lower-level services.

Type: Extension.

Benefits	Drawbacks
Separates integration concerns from the services – decouples the services' APIs	Poor latency for global use cases
Global use cases can be changed and deployed independently from the services	Qualities of the services become coupled to an extent
Decouples the services from the system's clients	API design is an extra step before the implementation can begin

References: [FSA] discusses orchestration in its chapters on *Event-Driven Architecture*, *Service-Oriented Architecture* and *Microservices*. [MP] describes orchestration-based *sagas* and its Order Service acts as an *application service* without explicitly defining the pattern. [POSA4] defines several variants of *Facade*.

An *orchestrator* takes care of global use cases (which involve multiple *services*) thus allowing each service to specialize in its own subdomain and, ideally, forget about the existence of all the other services. This way the entire system's high-level logic (which is usually subject to frequent changes) is kept (and deployed) together, isolated from usually more complex subdomain-specific services. Dedicating a *layer* to global scenarios makes them relatively easy to implement and debug, while the corresponding development team that communicates with clients shelters other narrow-focused teams from the disruptions. The cost of employing an orchestrator is both degraded performance when compared to basic services that rely on *choreography* [FSA, MP] (meaning they call or notify each other directly) and some coupling of the properties of the orchestrated services as the *orchestrator* usually treats them in a uniform way.

An *orchestrator* usually fulfills two closely related roles:

- As a *mediator* [GoF, SAHP] it keeps the states of the underlying components (services) consistent by propagating changes that originate in one component over the entire system. This role is prominent in *control* software, pervading automotive, aerospace and IoT industries. The *mediator* role emerges as *Saga* in *Microservices* [MP].
- As a *facade* [GoF] it builds high-level scenarios out of smaller steps that are provided by the controlled services or modules. This role is obvious for *processing* systems where clients communicate with the *facade*, but it is also featured in *control* systems, for often a simple event triggers a complex multi-component scenario which is managed by the system's *orchestrator*.

In a client-facing system a high-level request is transformed into a sequence of low-level steps

A control system turns a hardware event into a high-level action that spreads through the components

Facade   **Mediator**
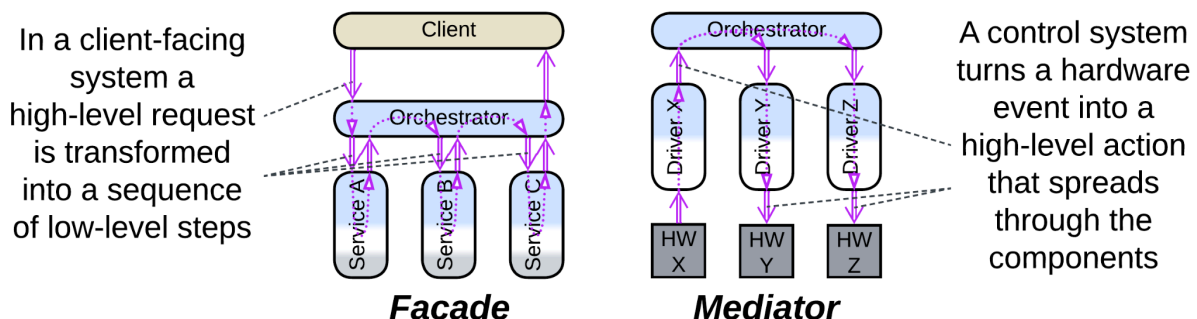
Data processing systems, such as backends, may deploy multiple instances of stateless *orchestrators* to improve stability and performance. Contrariwise, in *control* software the *orchestrator* incorporates the highest-level view of the system's state thus it cannot be easily duplicated (as any duplicated state must be kept synchronized, introducing delay or inconsistency in decision-making).

Performance

When compared to *choreography*, *orchestration* usually degrades latency as it involves extra steps of communication between the orchestrator and the orchestrated components. However, the effect should be estimated on case by case basis, as there are at least the following exceptions:

- An *orchestrator* may cache the state of the orchestrated system, gaining the ability to immediately respond to read requests with no need to query the underlying components. This is quite common with *control* systems.
- An *orchestrator* may persist a write request, respond to the client, then start the actual processing. Persistence grants that the request will eventually be completed.
- An *orchestrator* may run multiple subrequests in parallel, reducing latency when compared to the chain of choreographed events.
- In highly loaded or latency-critical systems the orchestrated services may establish direct data streams that bypass the orchestrator. A classic example is VoIP where the call establishment logic (SIP) goes through an orchestrating server while the voice or video (RTP) streams directly between the clients.

The last known state of the services can be cached and used to quickly respond to some queries

The response to the client can be sent before their request has been fully processed

Requests to the services can be run in parallel, improving latency

Data exchange between the services may bypass the orchestrator
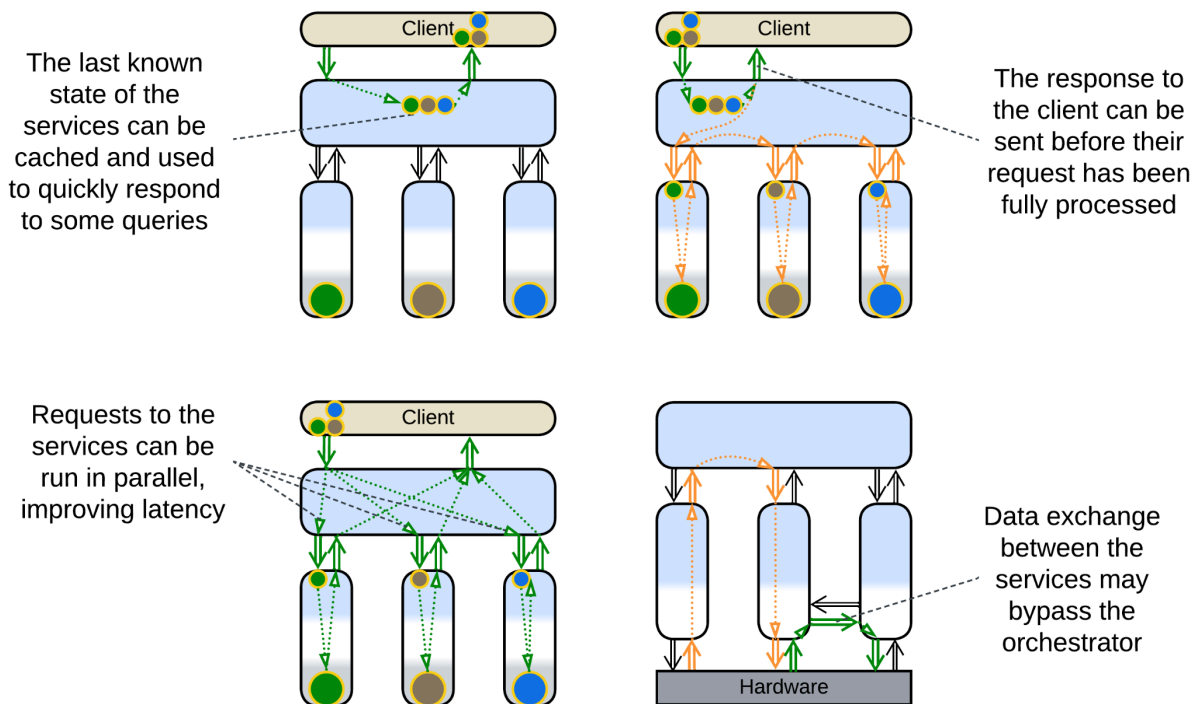
I don't see how orchestration can affect throughput as in most cases an orchestrator can be scaled by deploying several instances of it. However, scaling weakens consistency as now no instance of the orchestrator has exclusive control over the system's state.

Dependencies

An *orchestrator* may depend on *APIs* of the services it orchestrates or define *SPIs* for them to implement, with the first mode being natural for its *Facade* [GoF] aspect and the second one – for *Mediator* [GoF].

In a client-facing system the orchestrator tends to depend on each service

In control systems orchestrators tend to define SPIs which services implement

Facade   **Mediator**
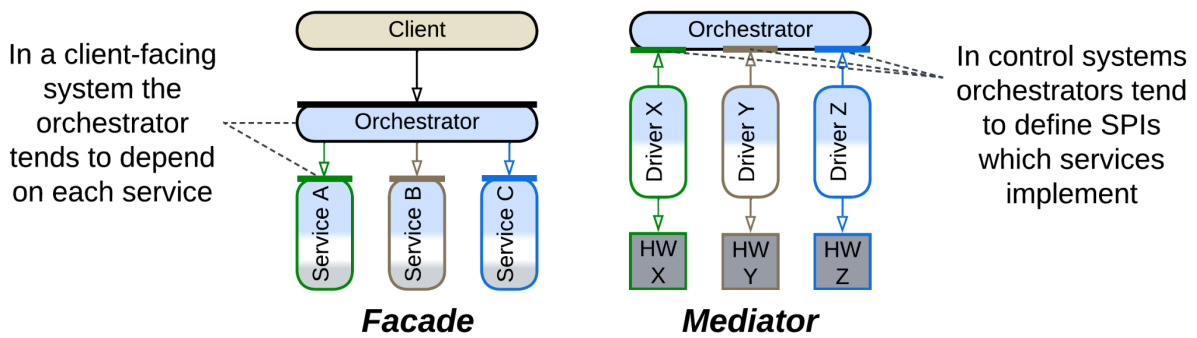
If an orchestrator is added to integrate existing components, it will use their APIs.

In large projects, where each service gets a separate team, the APIs need to be negotiated beforehand, and will likely be owned by the orchestrated services.

Smaller (single-team) systems tend to be developed top-down, with the orchestrator being the first component to implement, thus it defines the interfaces it uses.

Likewise, control systems tend to reverse the dependencies, with their services depending on the orchestrator's SPI – either because their events originate with the services (so the services must have an easy way to contact the orchestrator) or to provide for polymorphism between the low-level components. See the chapter on *orchestration* for more details.

Applicability

Orchestrators <u>shine</u> with:
- *Large projects.* The partition of business logic into a high-level application (*orchestrator*) and multiple subdomain services it relies on provides perfect code decoupling and team specialization.
- *Specialized teams.* As an improvement from *Services*, the teams that develop deep knowledge of subdomains delegate communication with customers to the application team.
- *Complex and unstable requirements*. The *integration* layer (*orchestrator*) should be high-level and simple enough to be easily extended or modified to cover most customer requests or marketing experiments without any help from the domain teams.

Orchestrators <u>fail</u> in:
- *Huge projects.* At least one aspect of complexity is going to hurt. Either the number of subdomain services and size of their APIs will make it impossible for an *orchestrator* programmer to find correct methods to call, or the *orchestrator* itself will become unmanageable due to the number and length of its use cases. This can be addressed by dividing the *orchestrator* into a layer of services (resulting in *Backends for Frontends*) or multiple layers (resulting in *Hierarchy*). The domain services may evolve into *Cells*, which is yet another kind of *Hierarchy*. It is also possible to go for *Service-Oriented Architecture* that has more fine-grained components.
- *Small projects*. The implementation overhead of defining and stabilizing service APIs and the performance penalty they cause are unlikely to be worthy of the extra flexibility that the *orchestrator* architecture provides for the (unsure) future project growth.
- *Low latency*. Any system-wide use case will make multiple calls between the application (*orchestrator*) and services, with each interaction adding to the latency.

Relations

Orchestrator for a monolith · **Orchestrator for layers** · **Orchestrator for shards** · **Orchestrator for services**
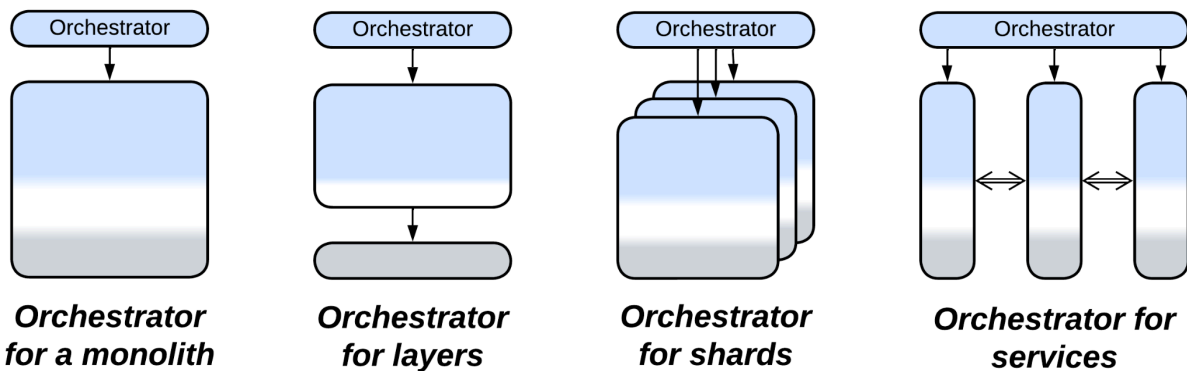
Orchestrator:

- Extends *Services* or rarely *Monolith*, *Shards* or *Layers* (forming *Layers*).
- Can be merged with a *proxy* into an *API gateway*, with a *middleware* into an *event mediator* or with a *middleware* and *adapters* into an *Enterprise Service Bus*.
- Is a special case (single service) of *Backends for Frontends*, *Service-Oriented Architecture* or (2-layer) *Hierarchy*.
- Can be implemented by *Microkernel* (as *Saga* or *Script*).

Variants by transparency

It seems that an *orchestrator*, just like a *layer*, which it is, can be *open* (*relaxed*) or *closed* (*strict*):

Closed or strict

A closed orchestrator intercepts every client request ...

... even if it deals with a single service
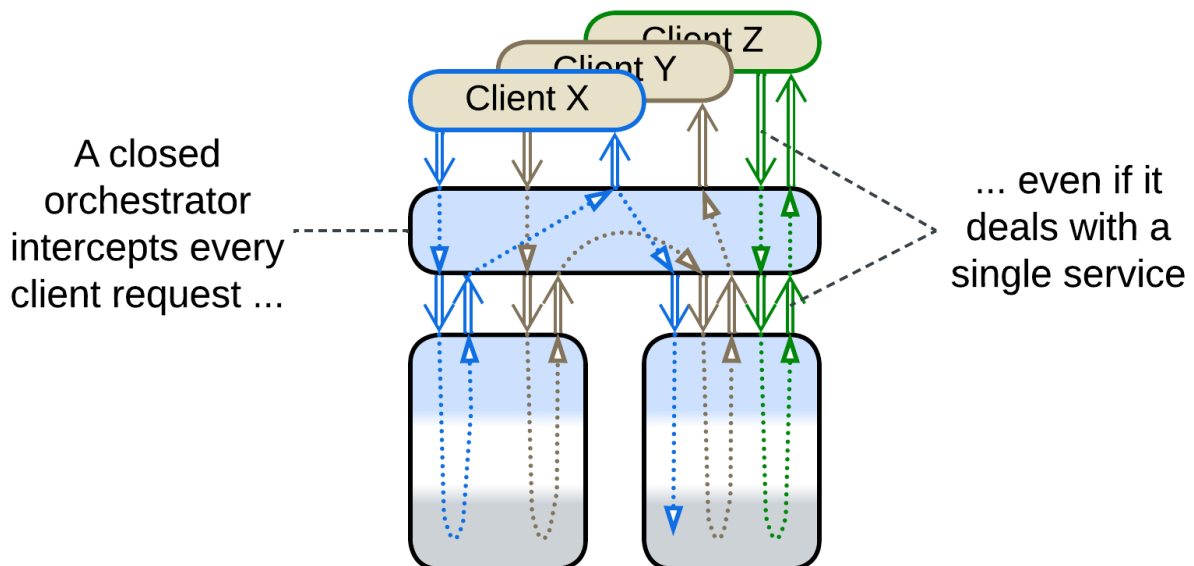
A *strict* or *closed* orchestrator isolates the orchestrated services from their users – all the requests go through the orchestrator, and the services don't need to intercommunicate.

An open
orchestrator
intercepts only
multi-service
requests

Any request
addressed to a
single service
bypasses the
orchestrator

The services
may notify
each other of
changes

An *open* orchestrator implements a subset of system-wide requests that require strict data consistency while less demanding requests go from the clients directly to the underlying services, which rely on *choreography* or *shared data* for communication. Such a system sacrifices the clarity of design to avoid some of the drawbacks of both choreography and orchestration:

- The orchestrator development team, which may be overloaded or slow to respond, is not involved in implementing the majority of use cases.
- Most of the use cases avoid the performance penalty caused by the orchestration.
- Failure of the orchestrator does not paralyze the entire system.
- The relaxed orchestrator still allows for synchronized changes of data in multiple services, which is rather hard to achieve with choreography.

Variants by structure (can be combined)

The orchestration (application [DDD] / integration / composite) layer has several structural (implementation) options and varies in the amount of functionality and flexibility it provides:

Monolithic
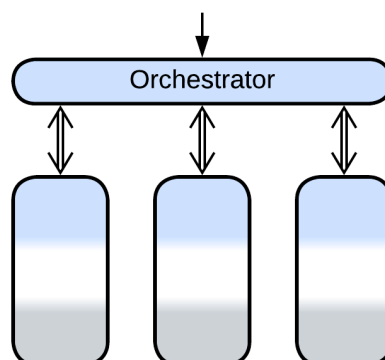
A single *orchestrator* is deployed. This option fits ordinary medium-sized projects.

Scaled

There are multiple
instances of a
stateless orchestrator

The load balancer
forwards an
incoming request to
any idle instance

Each service can
be commanded by
any orchestrator

Load Balancer

Orchestrator
Orchestrator
Orchestrator

Persistent
storage

Service A

Service B

Service C

The orchestrators
persist their actions
to a database so
that if one of them
crashes another
takes up its tasks

High availability requires multiple instances of a stateless *orchestrator* to be deployed. A *mediator* (*saga*, writing *orchestrator*) may store the current transaction's state in a *shared database* to assure that if it crashes there is always another instance ready to take up its job.

High load systems also require multiple instances of *orchestrators* because a single instance is not enough to handle the incoming traffic.

Layered

All the requests come
to the topmost simple
orchestrator

If the request is trivial,
it is run by the simple
orchestrator

That allows for rapid
development of the
majority of scenarios
without losing the
ability to implement
special cases

Simple orchestrator

Complex orchestrator

Requests that
require hand-written
logic are forwarded
to the complex
orchestrator

The lower-level
orchestrator is more
powerful but harder
to code

[FSA] describes an option of a layered *event mediator*. A client's request comes to the topmost layer of the *orchestrator* which uses the simplest (and least flexible) framework. If the request is known to be complex, it is forwarded to the second layer which is based on a more powerful technology. And if it fails or requires a human decision then it is forwarded again to the even more complex custom-tailored orchestration layer.

That allows the developers to gain the benefits of a high-level declaration language in a vast majority of scenarios while falling back to hand-written code for a few complicated cases. The choice is not free as one needs to learn multiple technologies, interlayer

debugging is never easy, and performance will likely be worse than with a monolithic *orchestrator*.

A similar example is using an *API composer* for the top layer, followed by a *process manager* and a *saga engine*.

A service per client type (Backends for Frontends)

Each kind of client gets its own orchestrator

That allows for scenarios to be optimized for each kind of client
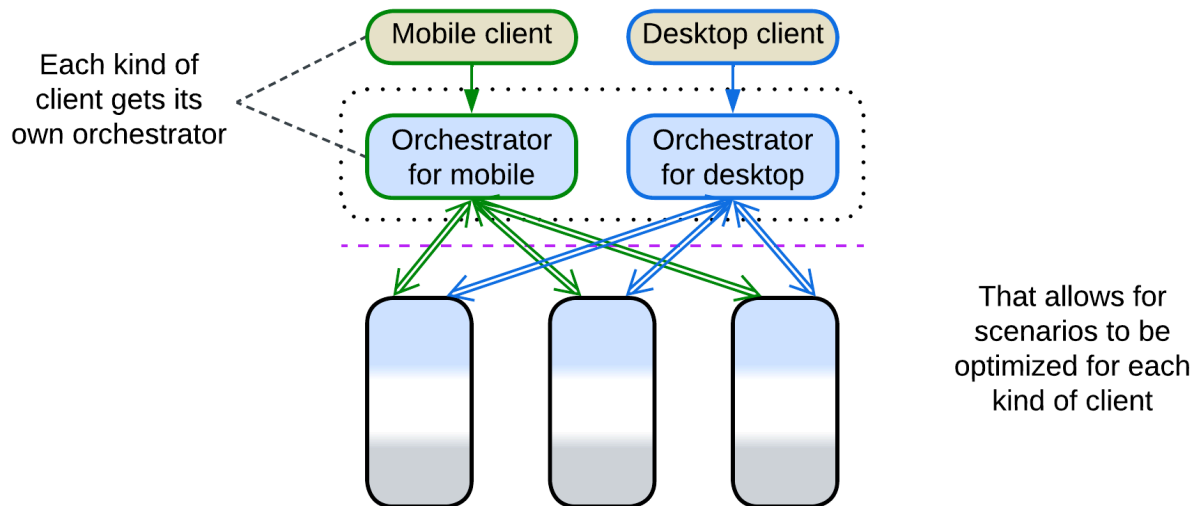
If your clients strongly differ in workflows (e.g. OLAP and OLTP, or user and admin interfaces), dedicated *orchestrators* is an option to consider. That makes each client-specific *orchestrator* to be smaller and more cohesive compared to the unified implementation and gives more independence to the teams responsible for different kinds of clients.

This pattern is known as *Backends for Frontends* and has a chapter of its own.

A service per subdomain (Hierarchy)

The top-level orchestrator builds on high-level APIs of the subdomains ...

... which are translated into multiple service calls by the lower-level orchestrators

That alleviates the system complexity by spreading it over multiple layers of components ...

... but requires the domain to be hierarchical
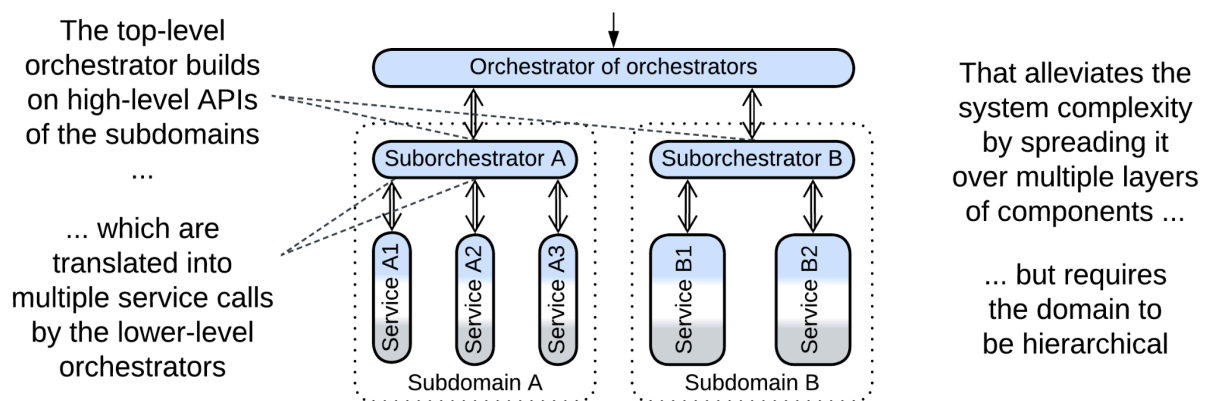
In large systems a single *orchestrator* is very likely to overgrow and become a development bottleneck (see *Enterprise Service Bus*). Building a *hierarchy* of *orchestrators* may help [FSA], but only if the domain itself is hierarchical. The top-level component may even be a *reverse proxy* if no use cases cross subdomain borders or the sub-*orchestrators* employ choreography, resulting in a flat *Cell-Based Architecture*. Otherwise it is a tree-like *Orchestrator of Orchestrators*.

A service per use case (SOA-style)

Each use case
gets a deidcated
orchestrator

The orchestrators
are easy to scale,
but there may be
too many of them
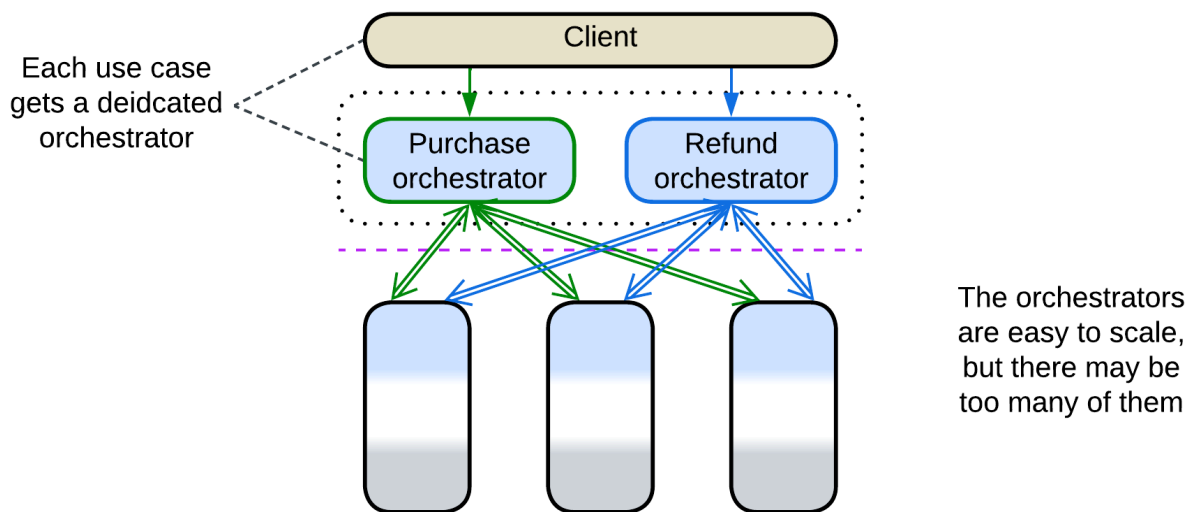
[SAHP] advises for single-purpose orchestrators in *Microservices*: each orchestrator manages one use case. This enables fine-grained scalability but will quickly lead to integration hell as new scenarios are added to the system. Overall, such a use of *orchestrators* resembles the *task layer* of *SOA*.

Variants by function

Orchestrator is a sibling of *Proxy*. While specialized *Proxy* variants take care of generic aspects such as security or protocols, *Orchestrator* defines the high-level business logic in terms of API calls to the underlying services. *API Gateway* stands in between – it meddles with API calls just like *Orchestrator* but also encapsulates cross-cutting concerns like old-fashioned *Proxy*.

API Composer, Remote Facade, Gateway Aggregation, Composed Message Processor, Scatter-Gather, MapReduce

An API composer
splits a client's
request into several
requests to services
and collects the
results

Scatter/Gather
and MapReduce
do the same to
shards or
replicas
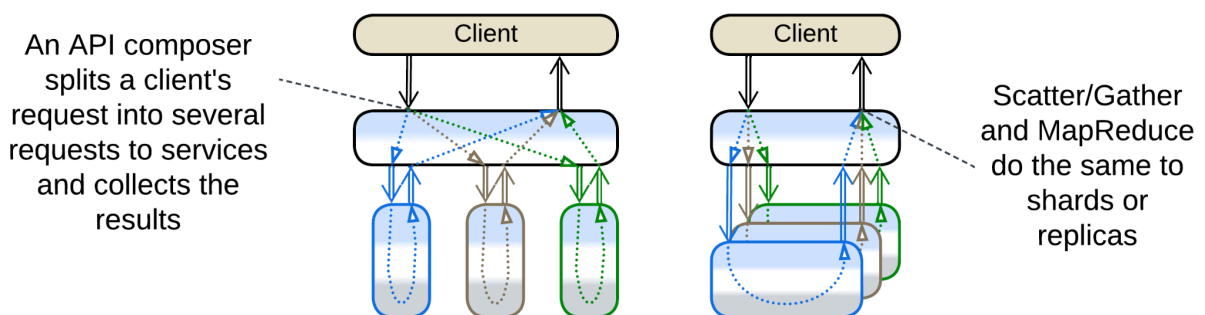
API Composer [MP] is a kind of *Facade* [GoF] that decreases the system's latency by translating a high-level incoming message into a set of lower-level internal messages, sending them to the corresponding *services* in parallel, waiting for and collecting the results into a response to the original message. Such a logic may often be defined declaratively in a 3rd party tool without writing any low-level code. *Remote Facade* [PEAA] is a similar pattern that makes synchronous calls to the underlying components – it exists to implement a

coarse-grained protocol with the system's clients, so that a client may achieve whatever they need through a single request. _Gateway Aggregation_ is the generalization of these patterns.

Composed Message Processor [EIP] is a generalization of the above: it uses _Splitter_ [EIP] to subdivide the request into smaller parts, _Router_ [EIP] to send each part to its recipient and _Aggregator_ [EIP] to collect the responses into a single message. Unlike _API Composer_, it can also address _shards_ or _replicas_. _Scatter-Gather_ [EIP, DDS] broadcasts a copy of the incoming message to each recipient, thus it lacks _Splitter_ (though [DDS] seems to ignore this difference). _MapReduce_ [DDS] is similar but it processes (summarizes) the results to yield a single value instead of combining (merging) them.

If an _API composer_ needs to conduct sequential actions (e.g. first get user id by user name, then get user data by user id), it becomes a _process manager_ which may require some coding.

An _API composer_ is usually deployed as a part of an _API gateway_.

Example: Microsoft has an article on aggregation.

Process Manager, Orchestrator

A process manager calls underlying services one by one to process a client's request

It may use the results of a previous call as arguments for the next one
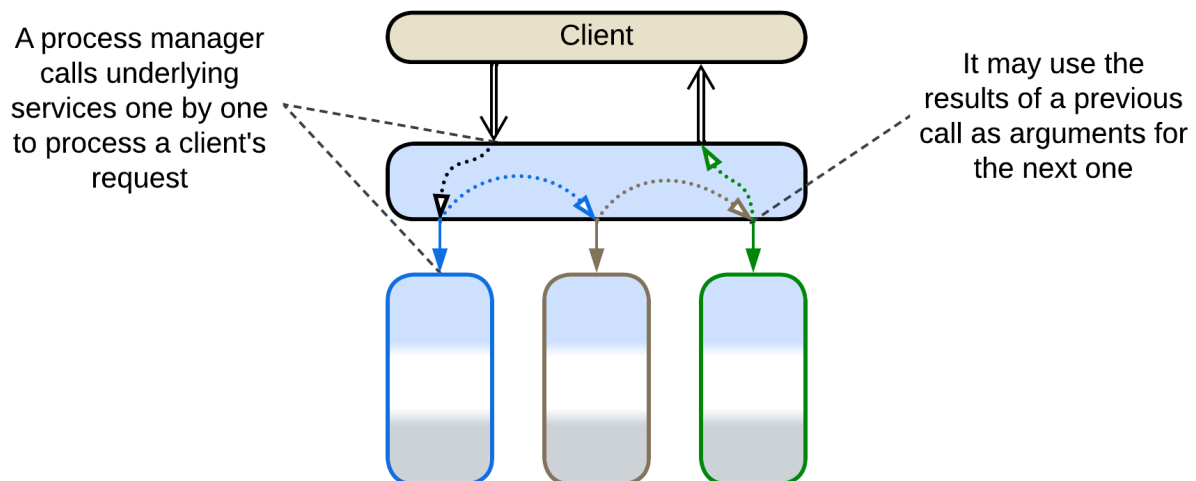
Process Manager [EIP, LDDD] (referred simply as _Orchestrator_ in [FSA]) is a kind of _Facade_ that translates high-level tasks into sequences of lower-level steps. This subtype of _Orchestrator_ receives a client request, stores its state, runs pre-programmed request processing steps and returns a response. Each of the steps of a _process manager_ is similar to a whole task of an _API composer_ in that it generates a set of parallel requests to internal services, waits for the results and stores them for the future use in the following steps or in the final response. The use case may branch on conditions.

A _process manager_ may be implemented in a general-purpose programming language, a declarative description for a 3rd party tool, or a mixture thereof.

A _process manager_ is usually a part of an _API gateway_, _event mediator_ or _enterprise service bus_.

Example: [FSA] provides several examples.

(Orchestrated) Saga, Saga Orchestrator, Saga Execution Component, Transaction Script, Coordinator

Atomic saga calls services just like a process manager

... but if one of its actions fails ...

it knows how to undo all its previous actions

Eventual saga retries a failed call till it succeeds

Any saga persists its state for crash recovery

Atomic Saga   **Eventual Saga**
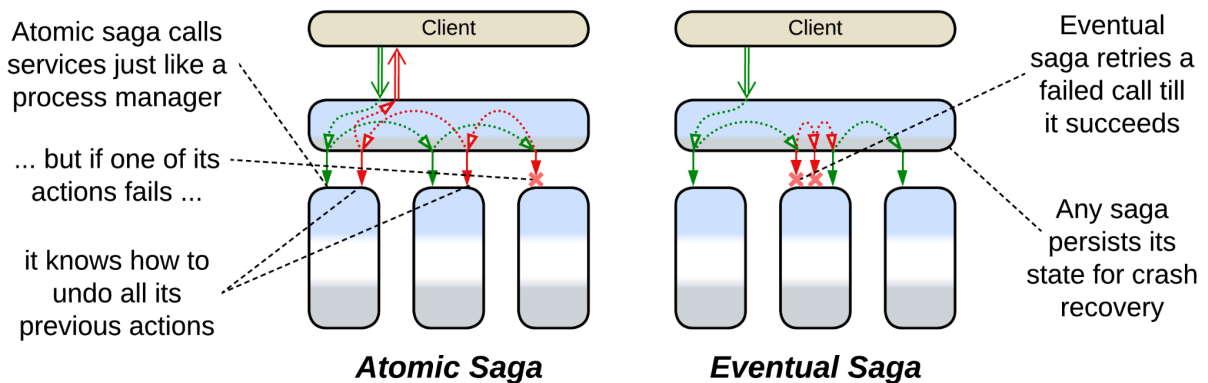
(Orchestrated [SAHP]*) Saga* [LDDD], *Saga Orchestrator* [MP] or *Saga Execution Component* is a subtype of *Process Manager* which is specialized in *distributed transactions*.

An *atomically consistent saga* [SAHP] (which is the default meaning of the term) comprises a pre-programmed sequence of {"do", "undo"} action pairs. When it is run, it iterates through the "do" sequence till it either completes (meaning that the *transaction* succeeded) or fails. A failed *atomic saga* begins iterating through its "undo" sequence to roll back the changes that were already made. Contrariwise, an *eventually consistent saga* [SAHP] always retries its writes till all of them succeed.

A *saga* is often programmed declaratively in a 3rd party *saga framework* which can be integrated into any service that needs to run a *distributed transaction*. However, it is quite likely that such a service is an *integration service* as it seems to orchestrate other services.

Saga plays the roles of both *Facade* by translating a single transaction request into a series of calls to the services' APIs and *Mediator* by keeping the states of the services consistent (the *transaction* succeeds or fails as a whole). Sometimes a *saga* may include requests to external services (which are not parts of the system you are developing).

Transaction Script [PEAA, LDDD] is a procedure that executes a transaction, possibly over multiple databases [LDDD]. Unlike *Saga*, it is synchronous, written in a general programming language and does not require a dedicated framework to run. It operates database(s) directly while *sagas* tend to send commands to services. A *transaction script* may also return data to its caller.

Coordinator [POSA3] is a generalized pattern for a component that manages multiple tasks (e.g. software updates of multiple components) to achieve "all or nothing" results (if any update fails, other components are rolled back).

Example: [SAHP] investigates many kinds of sagas while [MP] has a shorter description.

Integration (Micro-)Service, Application Service

An integration service is a full-featured service ...

Client

Integration Service

... which uses other services to satisfy its clients

Core Service A

Core Service B

Core Service C
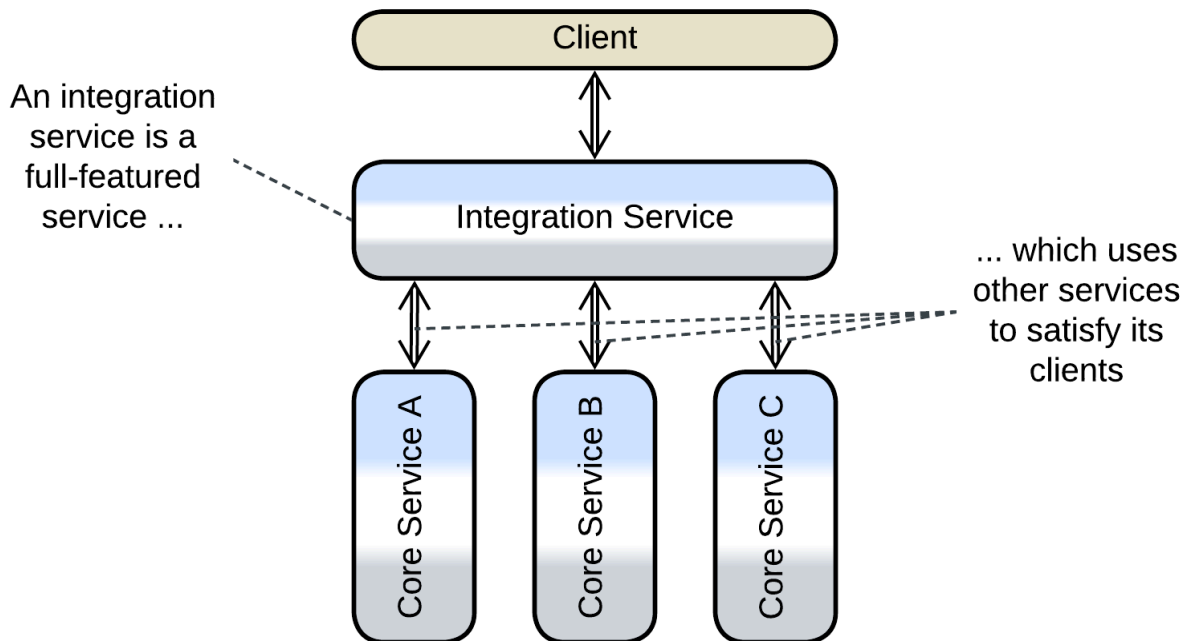
Integration Service is a full-scale service (often with a dedicated database) that runs high-level scenarios while delegating the bulk of the work to multiple other services (remarkably, delegating to a single component forms *Layers*). Though an *integration service* usually has both functions of *Orchestrator*, in control systems its *Mediator* role is more prominent while in *processing* software it is going to behave more like *Facade*. Such a system is closely related to shallow *Hierarchy*.

Example: Order Service in [MP] seems to fit the description.

Variants of composite patterns

Several composite patterns involve *Orchestrator* and are dominated by its behavior:

API Gateway

An API gateway provides protocol translation and generic functionality like a gateway ...

Client

... but also orchestrates the processing of requests like an API composer or process manager
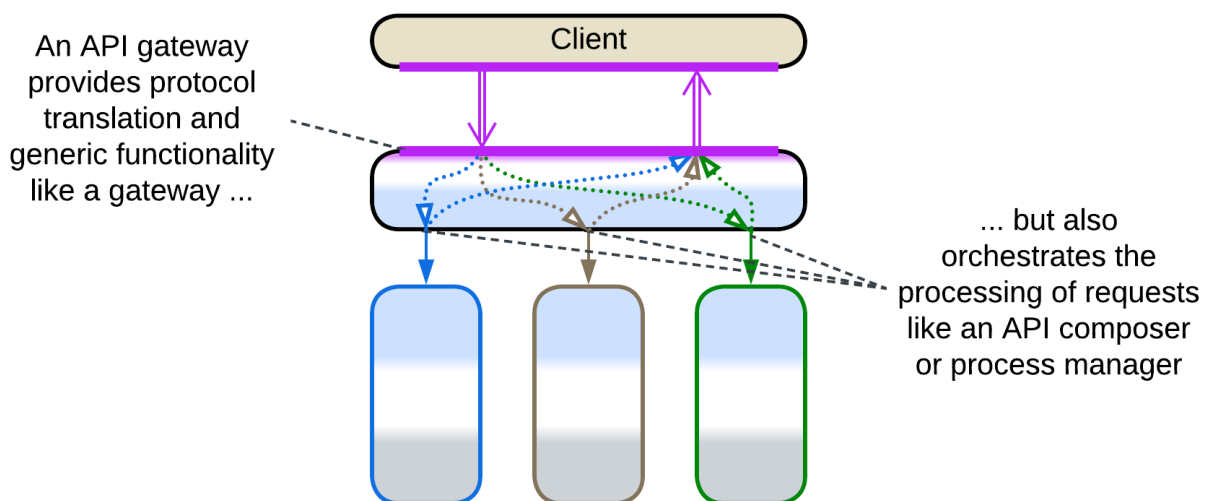
API Gateway [MP] is a component that processes client requests (and encapsulates an implementation of a client protocol(s)) like *Gateway* (a kind of *Proxy*) but also splits each client request into multiple requests to internal services like *API Composer* or *Process Manager* (*Orchestrators*). It is a common pattern for backend solutions as it provides all the means to isolate the stable core of the system's implementation from its fickle clients. Usually a 3rd party framework implements and colocates both its aspects, namely *Proxy* and *Orchestrator*, thus simplifying deployment and improving latency.

Example: a thorough article from Microsoft.
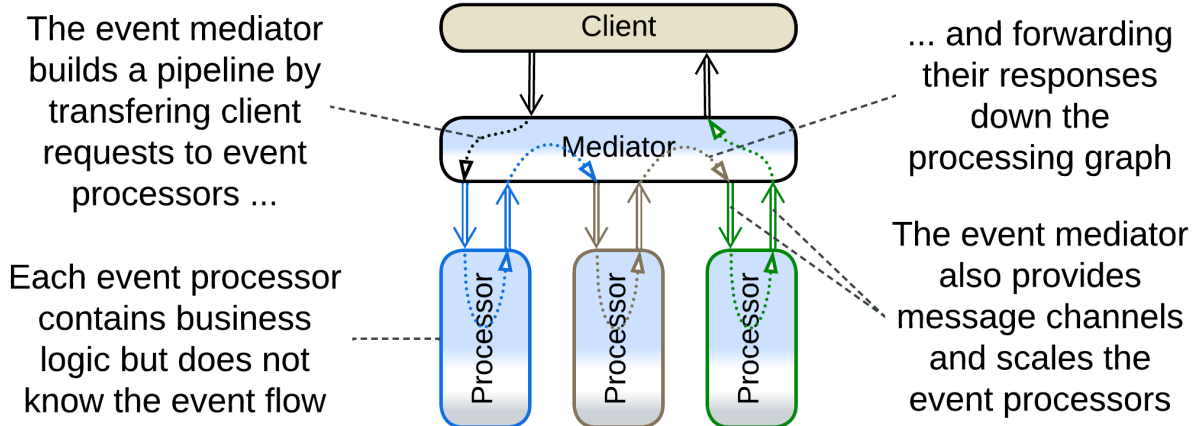
Event Mediator

The event mediator builds a pipeline by transfering client requests to event processors ...

Each event processor contains business logic but does not know the event flow

Client

Mediator

Processor   Processor   Processor

... and forwarding their responses down the processing graph

The event mediator also provides message channels and scales the event processors

Event Mediator [FSA] is an *orchestrating* *middleware*. It not only receives requests from clients and turns each request into a multistep use case (as *Process Manager*) but also manages the deployed instances of the services and acts as a medium that transports requests to the services and receives confirmations from them. Moreover, it seems to be aware of all the kinds of messages in the system and which service each message must be forwarded to, resulting in an overwhelming complexity concentrated in a single component, which does not even follow the separation of concerns principle. [FSA] recommends building a hierarchy of *event mediators* from several vendors, further complicating the architecture.

Example: Mediator Topology in the chapter on Event-Driven Architecture of [FSA].

Enterprise Service Bus (ESB)

An enterprise service bus mediates all calls and messages between services

A service posts its request to the ESB which knows how to handle it and which services are involved

Task   Task

Enterprise Service Bus

Entity   Entity   Entity

Util   Util   Lib   Lib

The ESB also implements the transport and cross-cutting concerns like logging

It employs an adapter per service so that the services may vary in their protocols

All that makes the ESB a complex component which may become a development bottleneck

Enterprise Service Bus (*ESB*) [FSA] is an overgrown *Event Mediator* that incorporates lots of cross-cutting concerns, including protocol translation with an *adapter* deployed per service. The combination of its central role in organizations and its complexity was among the main reasons for the demise of enterprise *Service-Oriented Architecture*.

Example: Orchestration-Driven Service-Oriented Architecture in [FSA].

Evolutions

Employing an *orchestrator* has two pitfalls:

- The system becomes slower because too much communication is involved.
- The single *orchestrator* may be found too large and rigid.

There is a handful of evolutions to counter those weaknesses:

- Subdivide the *orchestrator* by the system's subdomains, forming *Layered Services*.

An overgrown orchestrator may be divided to form an upper layer of services

- Subdivide the *orchestrator* by the type of client, forming *Backends for Frontends*.

Each kind of a client may have their own orchestrator

- Add another *layer* of orchestration.

Simple scenarios may be handled by the high-level orchestrator ...

... while complex requests are forwarded to another one

- Build a *top-down hierarchy*.

Some domains allow for hierarchical solutions
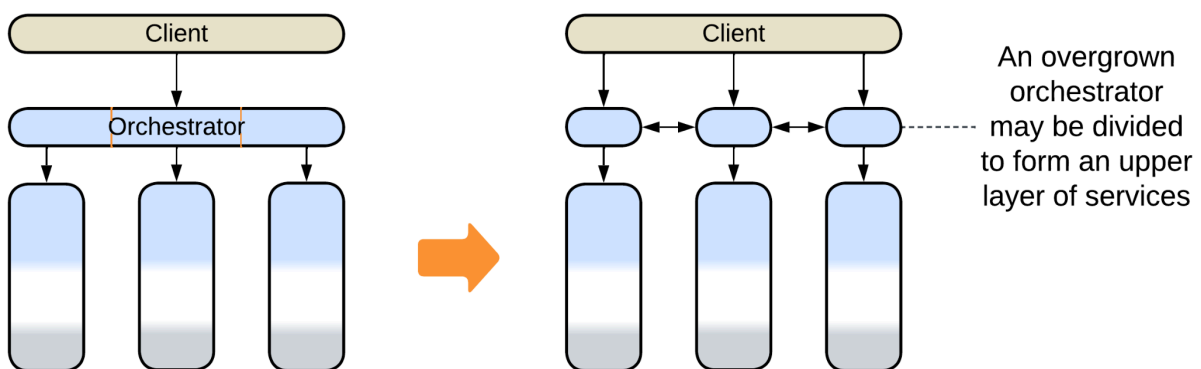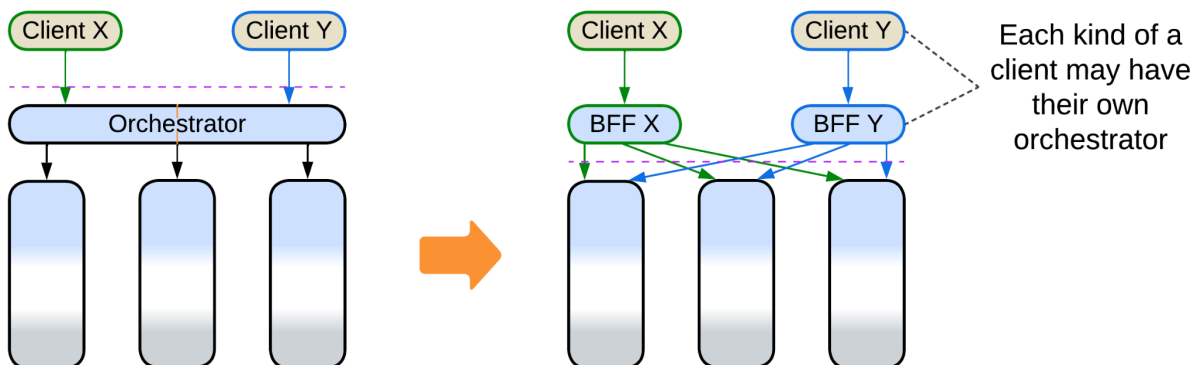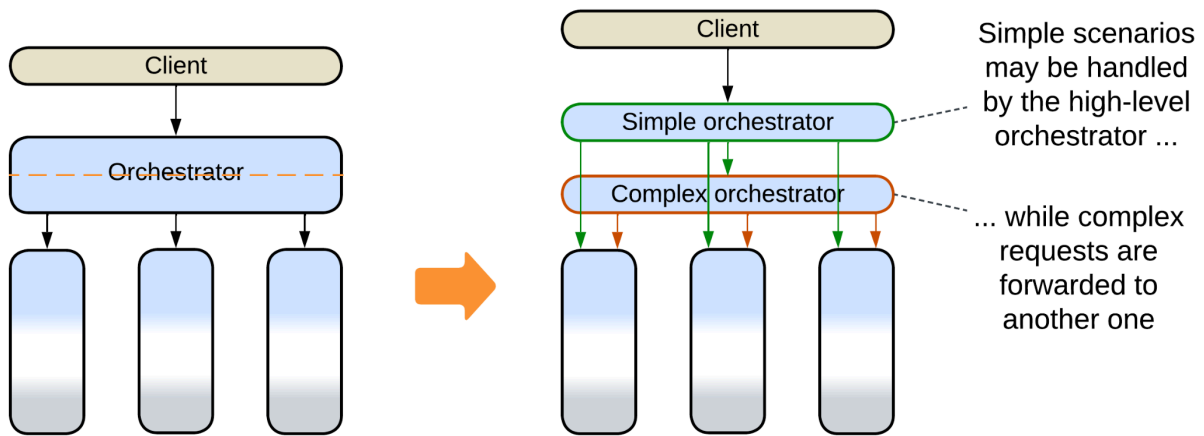
Summary

An *orchestrator* distills the high-level logic of your system by keeping it together in a layer which integrates other components. However, that involves much communication which impairs performance.

Combined Component

Jack of all trades. Use a 3rd party software that covers multiple concerns.
<u>Aspects:</u> those of the individual components it combines.
<u>Variants:</u>
- Message Bus [EIP],
- API Gateway [MP],
- Event Mediator [FSA],
- Front Controller [SAHP but not PEAA],
- Enterprise Service Bus [FSA],
- Service Mesh [FSA],
- Middleware of Space-Based Architecture [SAP, FSA].

<u>Structure:</u> Two or more (usually) extension patterns combined into a single component.
<u>Type:</u> Extension.

Benefits	Drawbacks
Works off the shelf	Is yet another technology to learn
Improved latency	May not be flexible enough for your needs
Less DevOps effort	May become overcomplicated

<u>References:</u> Mostly [FSA], Microsoft on API Gateway.

Two or three metapatterns may be blended together into a *combined component* which is usually a ready-to-use framework that tries to cover (and subtly create) as many project needs as possible to make sure it will never be dropped out of the project. On one hand, such a framework may provide a significant boost to the speed of development. On the other – it is going to force you into its own area of applicability and keep you bound within it.

Performance

A *combined component* tends to improve performance as it removes the network hops and data serialization between the several components it replaces. It is also likely to be highly optimized. However, that matters if you really need all the functionality that you are provided with, otherwise you may end up running a piece of software which is too complex and slow for the tasks at hand.

Dependencies

A *combined component* has all the dependencies of its constituent patterns.

Applicability

Combined patterns <u>work well</u> for:
- *Series of similar projects.* If your team is experienced with the technology and knows its pitfalls, it will be used efficiently and safely.
- *Small- to medium-sized domains.* An off-the-shelf framework relieves you of infrastructure concerns. No thinking, no decisions, no time wasted.

Combined patterns <u>are better avoided</u> in:

- *Research and development.* You may find that the technology chosen is too limiting or a wrong fit for your needs when it is already deeply integrated into your code and infrastructure.
- *Large projects.* Most of the combined patterns include an *orchestrator* which tends to grow unmanageable (requiring some kind of division, see variants of *Orchestrator*) as the project grows.
- *Long-running projects.* You are very likely to run into *vendor lock-in* as the industry evolves, leaving obsolete the technology you have chosen and integrated.

Variants

Combined components vary in structure and properties:

Message Bus

Message Bus employs an adapter per service ...

... to translate between protocols the services use
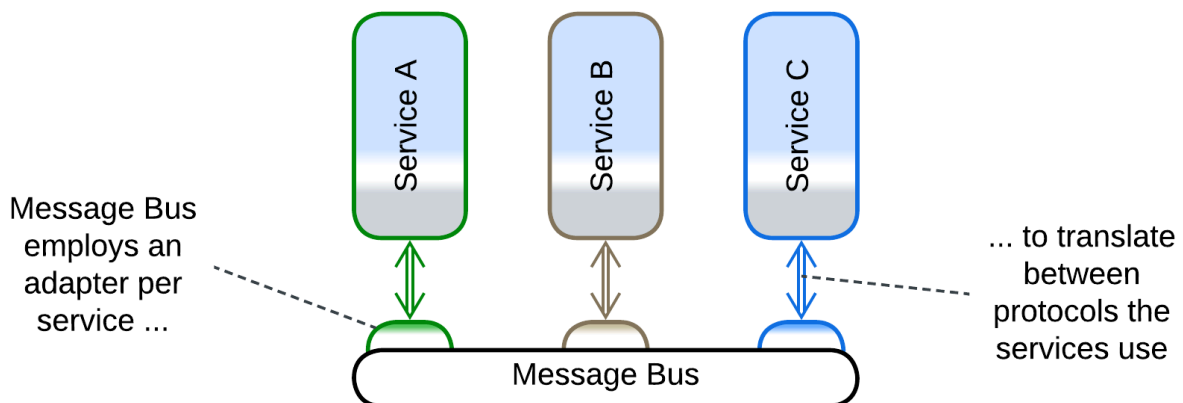
Message Bus [EIP] is a *middleware* that employs an *adapter* per *service* allowing services that differ in protocols to intercommunicate.

API Gateway

An API gateway blends a gateway and an orchestrator (often an API composer)

The gateway deals with the client protocol and secures the internal systems

The orchestrator splits client requests into multiple calls to the services
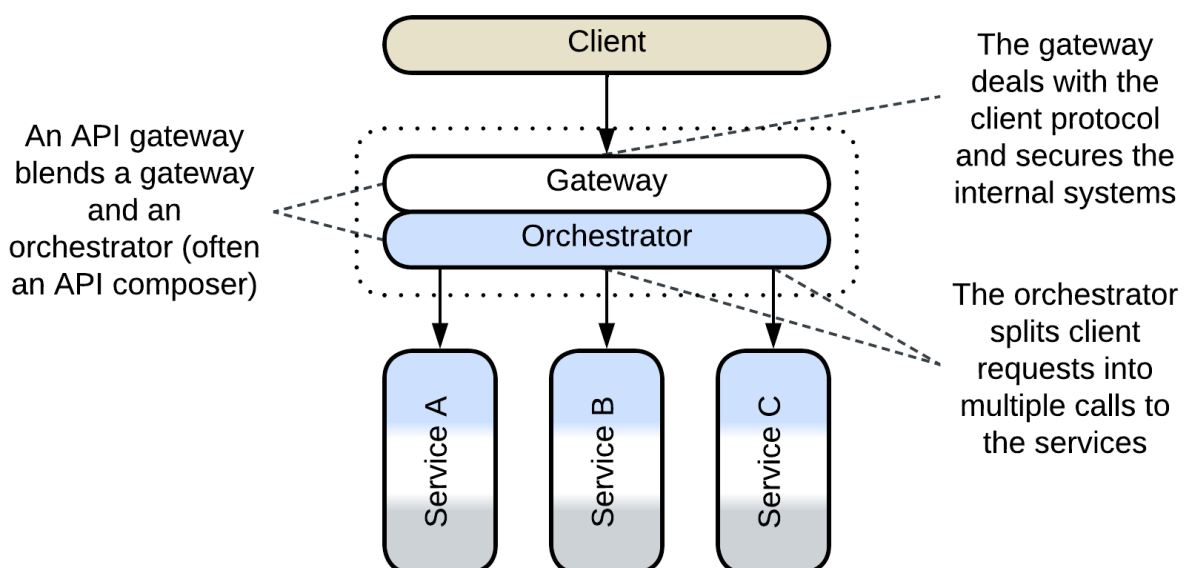
API Gateway [MP] is a component that processes client requests (and encapsulates the client protocol(s)) like a *gateway* (a kind of *Proxy*) but also splits each client request into multiple requests to the underlying services like an *API composer* or *process manager* (*Orchestrators*).

If the orchestration logic of an *API gateway* needs to become more complex, it makes sense to split the component into a separate *gateway* and *orchestrator*, rewriting the latter as a custom *application service*. When there are multiple types of clients that strongly differ in workflows and protocols, an *API gateway* per client type is used, resulting in *Backends for Frontends*. *Cell-Based Architecture* relies on *API gateways* for isolation of its *cells*.

Example: a thorough article from Microsoft.

Event Mediator

An event mediator blends an orchestrator and a middleware

The orchestrator knows which event processors are involved in each client request

The middleware transports messages and scales the event processors

Event Mediator [FSA] is an *orchestrating Middleware*. It not only receives requests from clients and turns each request into a multistep use case (as *Orchestrator* does) but also manages the instances of the service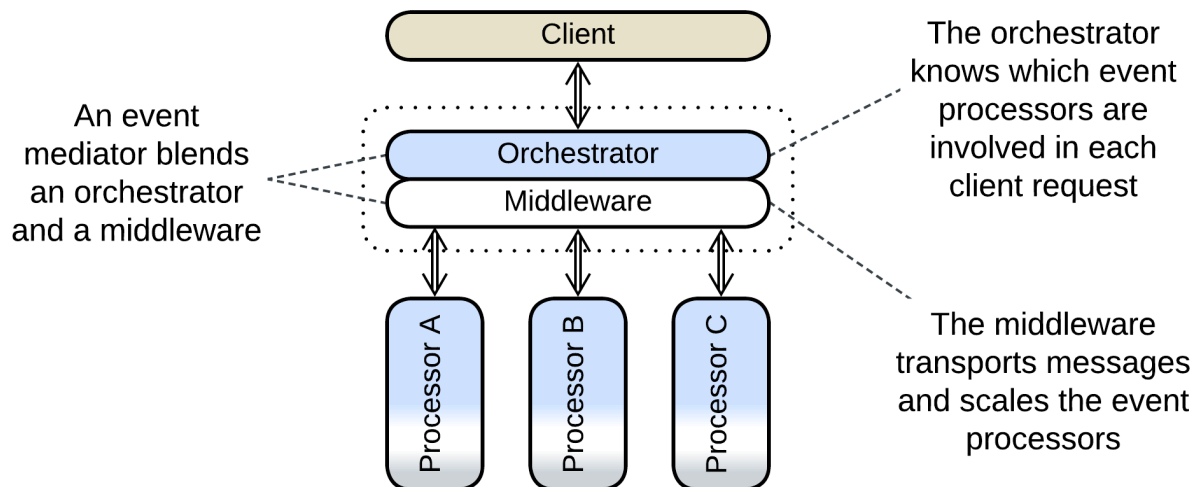s and acts as a medium that transports requests to the services and receives confirmations from them. Moreover, it seems to be aware of all the kinds of messages in the system and which service each message must be forwarded to, resulting in an overwhelming complexity concentrated in a single component, which does not even follow the separation of concerns principle. [FSA] proposes to counter that by using multiple *event mediators* over the next dimensions:

- Client applications or bounded contexts, dividing the *event mediator*'s responsibility by subdomain.
- Complexity of a use case, with simple scenarios handled by a simple first-line *event mediator* and more complicated being forwarded to second- and third-line *event mediators* that employ advanced *orchestration engines*.

strangely resulting in multiple *middlewares* connected to the same set of services.

The pattern seems to be well established, but obviously it may become quite messy for larger projects with nontrivial interactions. Such cases may also be solved by separating the *middleware* from the *orchestrator* and dividing the latter into *Backends for Frontends*.

Example: Mediator Topology in the [FSA] chapter on Event-Driven Architecture.

Front Controller

Front controller faces client's commands and queries

After doing its part it passes a command down the pipeline
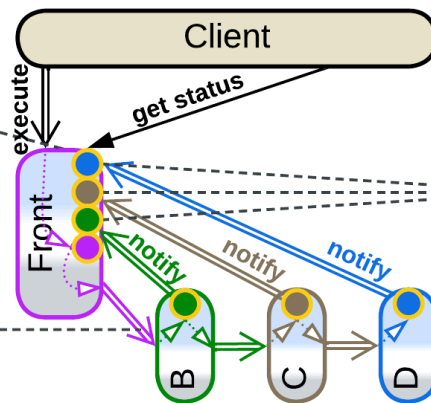
It collects status reports from other services to answer the client's queries

Front Controller [SAHP but not PEAA] is the name for the first (client-facing) service of a *pipeline* in *Choreographed Event-Driven Architecture* when the service collects information about the status of each request that it has processed and forwarded down the *pipeline.* The status is received by listening to notifications by the downstream services and is readily available for the front controller's clients, resembling the function of *Query Service*.

Enterprise Service Bus (ESB)

An enterprise service bus contains an orchestrator, a middleware and an adapter per service
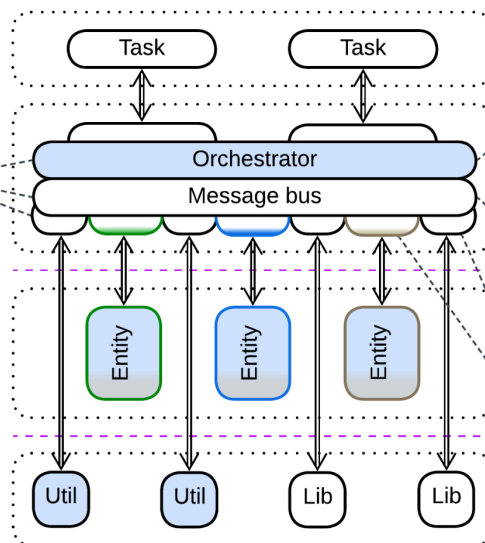
The orchestrator knows which services process each kind of request

The message bus transports messages between the services and provides delivery guarantees

Each adapter translates between its service's protocol and the ESB's protocol

Enterprise Service Bus (*ESB*) [FSA] is a mixture of *Message Bus* (with *adapter* per service) and *Event Mediator* (built-in *orchestrator*) that turns this kind of *Middleware* into a core of the system. The combination of its central role in organizations and its complexity was among the main reasons for the demise of enterprise *Service-Oriented Architecture*.

Example: Orchestration-Driven Service-Oriented Architecture in [FSA], how it is born and how it dies by Neal Ford.

Service Mesh

Service mesh is a distributed middleware for microservices

There is a virtual layer of sidecars - libraries colocated with each service instance

The mesh engine transfers messages between and scales the microservices
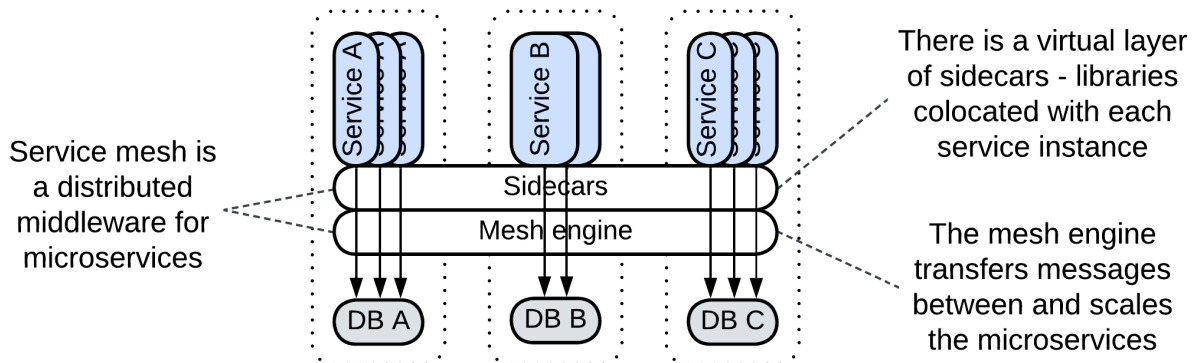
Service Mesh [FSA] is a *middleware* that employs a *sidecar* [DDS] (*adapter*) per service as a place for cross-cutting concerns (logging, observability, encryption, protocol translation). A service accesses its *sidecar* without performance and stability penalties as they are running on the same machine. The totality of deployed *sidecars* makes a system-wide logical layer of shared libraries: though the *sidecars* are physically separate, they are often identical and stateless, so that a service that accesses one *sidecar* may be thought of as accessing all of them.

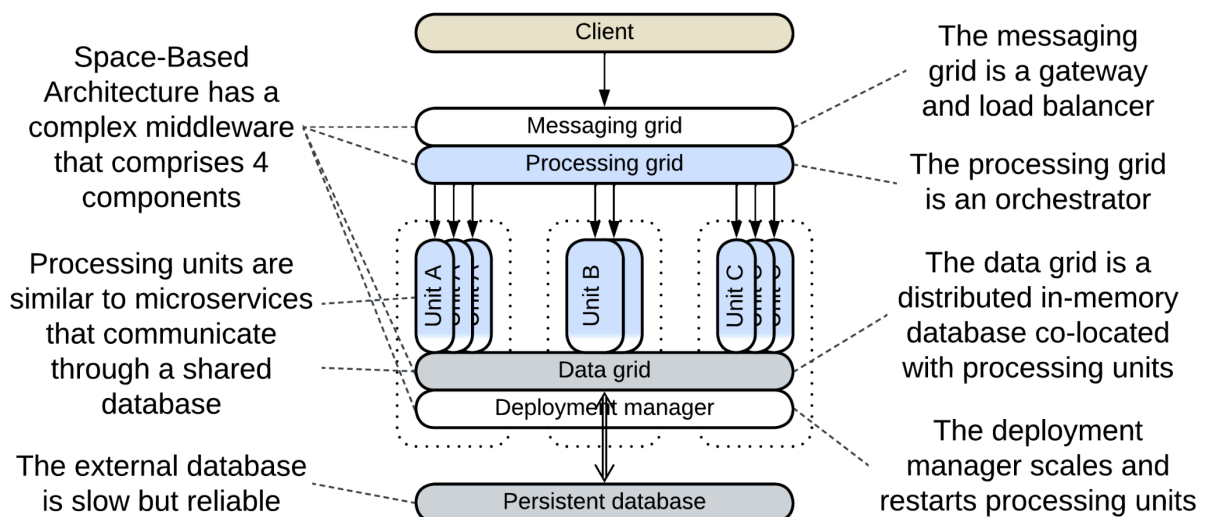The *service mesh* is also responsible for dynamic scaling (it creates new instances if the load increases and destroys them if they become idle) and failure recovery of the services. Last but not least, it provides a messaging infrastructure for the *microservices* to intercommunicate.

Middleware of Space-Based Architecture

Space-Based Architecture has a complex middleware that comprises 4 components

Processing units are similar to microservices that communicate through a shared database

The external database is slow but reliable

The messaging grid is a gateway and load balancer

The processing grid is an orchestrator

The data grid is a distributed in-memory database co-located with processing units

The deployment manager scales and restarts processing units

Space-Based Architecture [SAP, FSA] relies on the most complex *middleware* known – it incorporates all 4 *extension metapatterns*:
- *Messaging Grid* is a *proxy* that receives, preprocesses and persists client requests (as a *gateway*). Simple requests are forwarded to a less loaded *processing unit* while complex ones go to the *processing grid*.
- *Processing Grid* is an *orchestrator* that manages multi-step workflows for complex requests.

- *Data Grid* is a [distributed](#) in-memory [database](#). It is built of nodes which are co-located with instances of *processing units*, making the database access extremely fast. However, the speed and scalability is paid for with stability – any data in RAM is prone to disappearing. Thus the *data grid* backs up all the writes to a slower *external database*.
- *Deployment Manager* is a [middleware](#) that creates and destroys instances of *processing units* (paired to the nodes of the *data grid*) just like [Service Mesh](#) does for [Microservices](#) (paired to *Sidecars* [[DDS](#)]). However, in contrast to *service mesh*, it does not provide a messaging infrastructure because *processing units* communicate by sharing data via the *data grid*, not by sending messages.

The four layers of the *Space-Based Architecture*'s *middleware* are reasonably independent. Together they make a system that is both more scalable and more complex than *Microservices*.

Evolutions

The patterns that involve [orchestration](#) (API Gateway, Event Mediator, Enterprise Service Bus) may allow for most of the evolutions of the [Orchestrator](#) metapattern by deploying multiple versions of the component. There is also a special evolution:
- Replace the combined component with several specialized ones

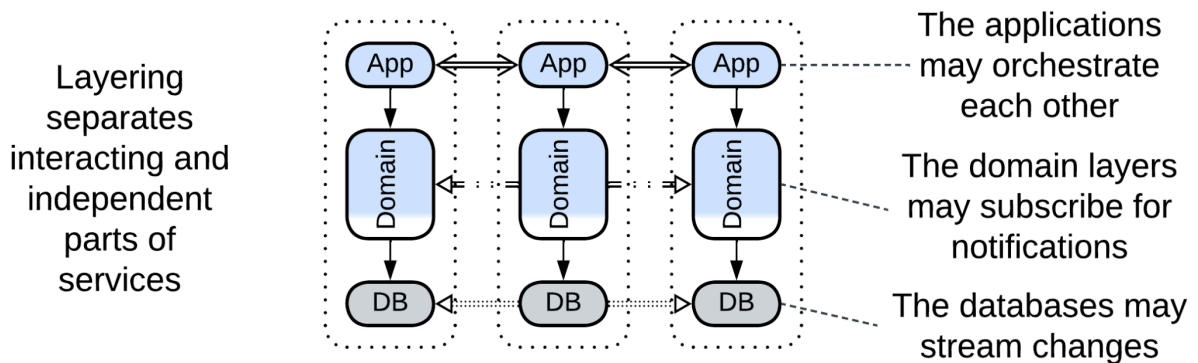Two or three ordinary components can replace a multifunctional one
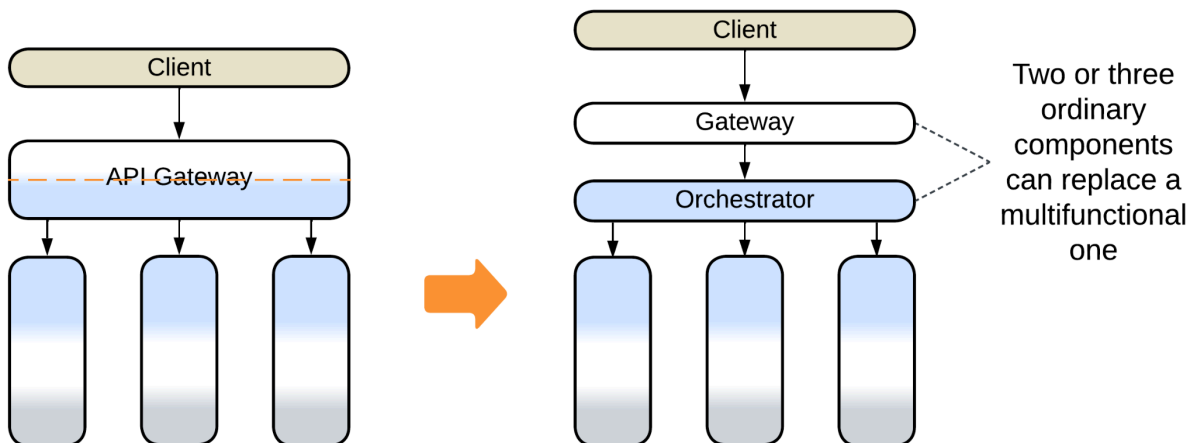
Summary

A *combined component* is a ready-to-use framework that may speed up the development but will likely constrain your project to follow its guidelines with no regard to your real needs.

Part 4. Fragmented Metapatterns

There are patterns with no system-wide layers. Some of them incorporate two or three domains at various abstraction levels, so that a service (limited to a subdomain) in one domain acts as a layer for another domain.

Layered Services

Layering separates interacting and independent parts of services

The applications may orchestrate each other

The domain layers may subscribe for notifications

The databases may stream changes

Layered Services is an umbrella metapattern which highlights implementation details of Services, Pipeline or Monolith.

Includes: orchestrated three-layered services, choreographed two-layered services, Command Query Responsibility Segregation (CQRS).

Polyglot Persistence

The system earns benefits of multiple database technologies
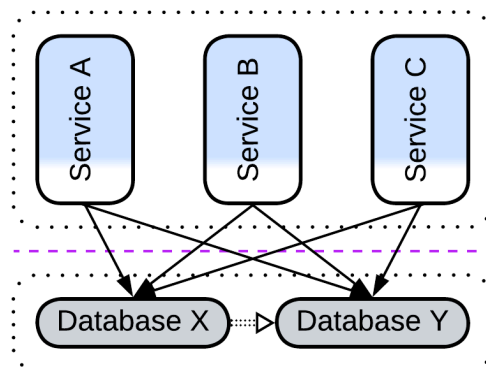
The databases store separate subsets or derived views of the data

Polyglot Persistence is about using multiple data stores which differ in roles or technologies. Each of the upper-level components may have access to any data store. Each data store is a shared repository.

Includes: specialized databases, private and shared databases, data file, Content Delivery Network (CDN); read-only replica, Reporting Database, CQRS View Database, Memory Image, Query Service, search index, historical data, Cache-Aside.

Backends for Frontends

Each BFF is
dedicated to its
kind of client
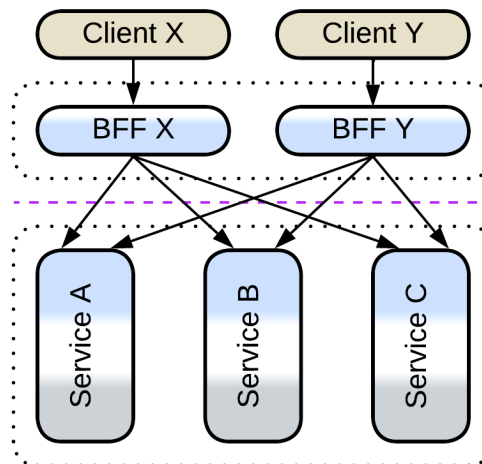
Each BFF
orchestrates
all the services

Backends for Frontends has a service (BFF) for each type of the system's client. The BFF may be a proxy, orchestrator or both. Each BFF communicates with all the components below it. The pattern looks like multiple proxies or orchestrators deployed together.
Includes: Layered Microservice Architecture.

Service-Oriented Architecture

SOA is 3 or 4
layers of
services
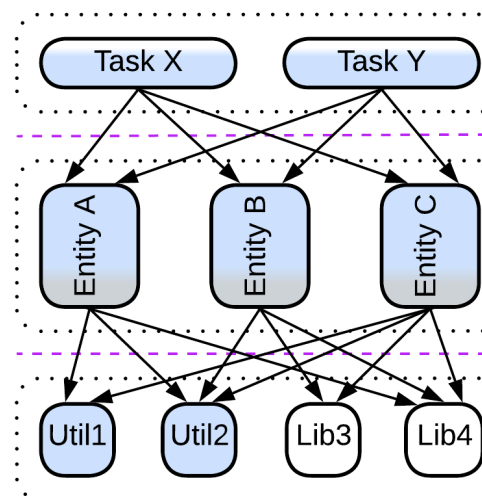
Each task
orchestrates
entities

Entities use
libraries and
utilities

SOA has three or four layers of services, each in its own domain. The upper layer contains orchestrators which are often client-specific, like BFFs. The second layer contains business rules and is divided by the business subdomains. The lower layer(s) are libraries and utilities, grouped by functionality and technologies. Each component may use (orchestrates) all the components below it.
Includes: Segmented Architecture; distributed monolith, enterprise SOA.

Hierarchy

Hierarchy is a tree of components
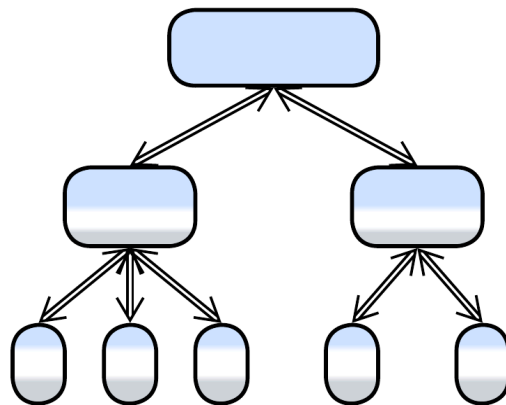
Each root orchestrates its children

Child nodes may be specialised or polymorpfic

Some domains allow for hierarchical composition where the functionality is spread over a tree of components.

Includes: Orchestrator of Orchestrators, Bus of Buses, Cell-Based (Microservice) Architecture (WSO2 version) (Services of Services).

Layered Services

Business logic

Generic code

Data

Layered Services

Cut the cake. Divide each service into layers.
Variants:
- Orchestrated three-layered services,
- (*Pipelined*) Choreographed two-layered services,
- (*Pipelined*) Command Query Responsibility Segregation (CQRS) [MP, LDDD].
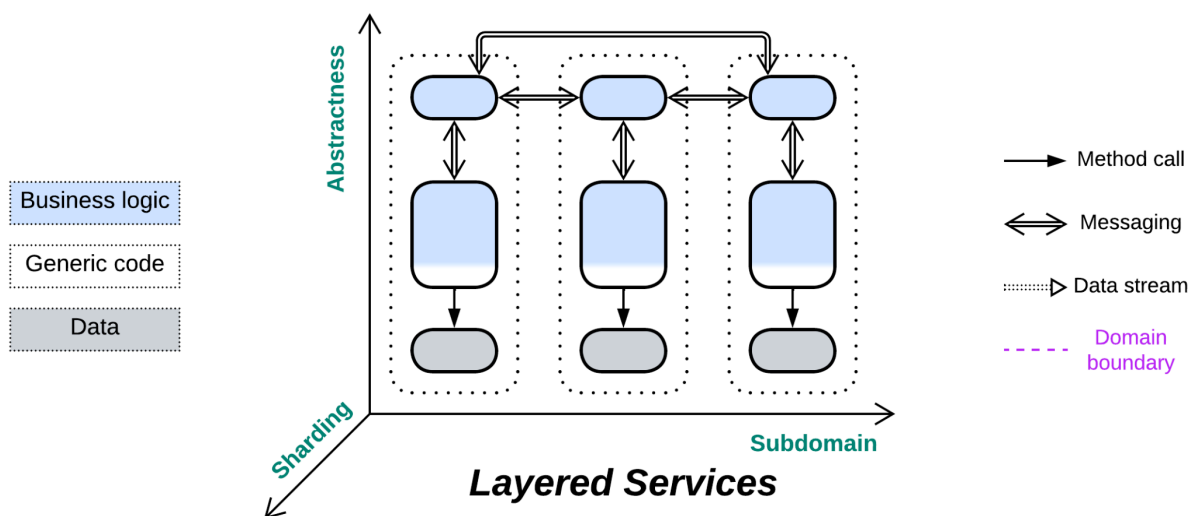
Structure: Domain services, each divided into layers.
Type: Implementation of *Services*, *Pipeline* or *Monolith*.

Layered Services is an umbrella architecture for common implementations of systems of *Services*. It does not introduce any special features as the component layers are completely encapsulated by the services they belong to. Still, as services may communicate through different layers, we can learn a couple of things by looking into the matter.

Performance

Layered Services are similar to *Services* performance-wise: use cases that involve a single service are the fastest, those that need to synchronize states of multiple services are the slowest.

Remarkable features of *Layered Services* include:
- Independent scaling of layers of the services. It is common to deploy multiple instances (the number varies from service to service and may change dynamically under load) of the layers that contain the business logic while the corresponding data layers (databases) are limited to single instances.

Services are deployed behind load balancers

Service instances are stateless and their number depends on load for the service

Interservice traffic goes through load balancers

Databases do not scale
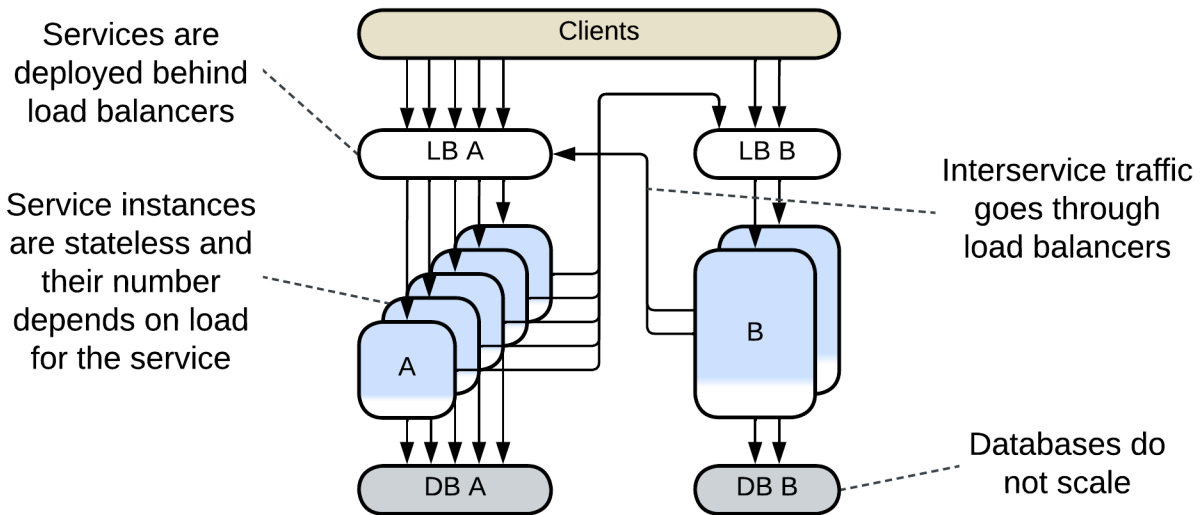
- The option to establish additional communication channels between lower layers to drive *CQRS* (read/write replicas of the same database), materialized views (cached subsets of data from other services) [DDIA] and event sourcing [MP].

Database replication

Event feed to a database

Event sourcing between services
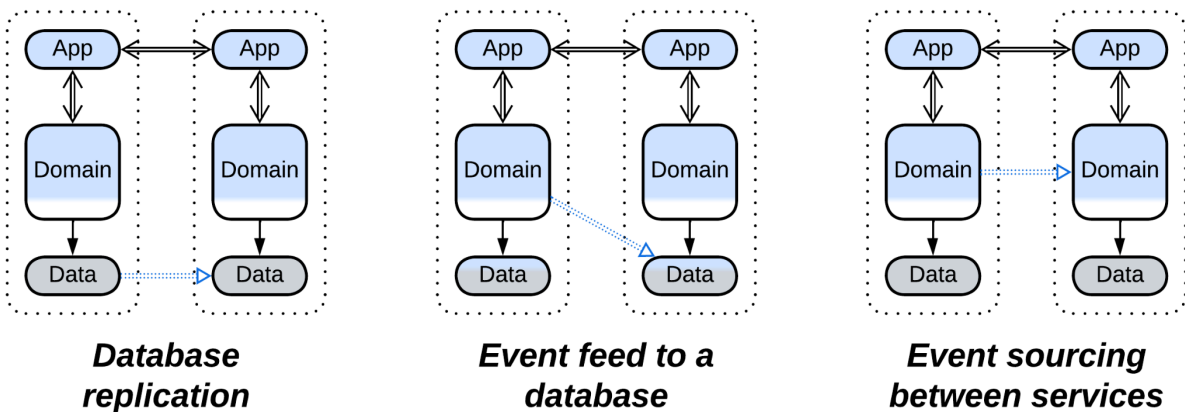
Variants

Layered Services vary in the number of *layers* and the level at which the *services* communicate:

Orchestrated three-layered services

Each of the applications orchestrates its own domain

The "passive" domain services implement business rules but don't run any use cases

But also each application otchestrates every other appplication
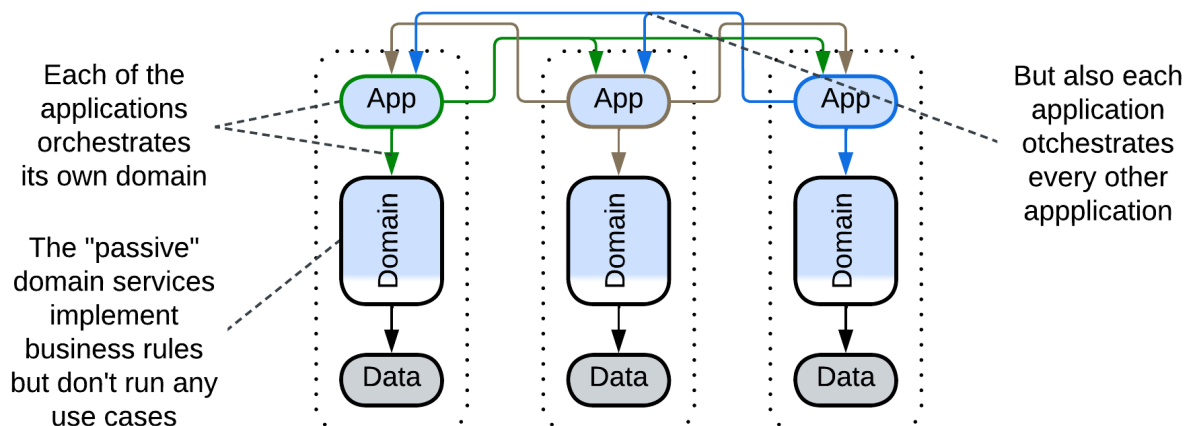
Probably the most common backend architecture has 3 layers: *application*, *domain* and *infrastructure* [DDD]. The *application* layer *orchestrates* the domain layer.

If such an architecture is divided into services, each of them receives its own part of the application layer, which means that now there are as many orchestrators as services in the system. Each orchestrator implements the API of its service by integrating (calling or messaging into) the domain layer of its service and the APIs of other services, making all the orchestrators interdependent:

Dependencies

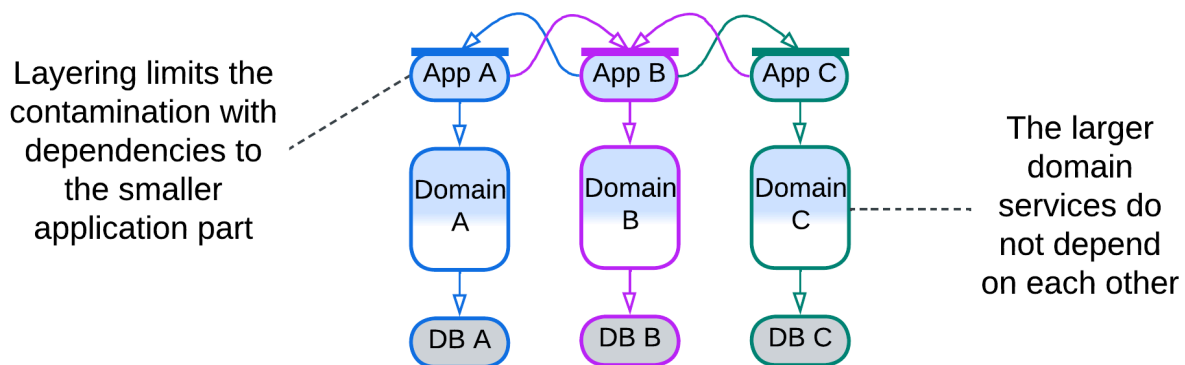The upper (application) layer of each service orchestrates both its middle (domain) layer and the upper layers of other services, resulting in mutual orchestration and interdependencies.

Layering limits the contamination with dependencies to the smaller application part

The larger domain services do not depend on each other

The good thing is that the majority of the code belongs to the *domain* layer which depends only on its *databases*. The bad thing is that changes in the *application* of one service may affect the *application* layers of all the services.

Relations

Three-layered services:
● Implement *Services*.
● Are derived from *Layers* and *Services*.
● Have multiple *integration (sub)services* (*orchestrators*).

Evolutions

Orchestrated Layered Services may become coupled, which is resolved by merging their layers:
● A part of or the whole *application* layer can be merged into an *orchestrator*.
● Some or all the *databases* can be united into a *shared database* or shared as *polyglot persistence*.

Merging orchestrators   **Merging databases**   **Sharing databases**
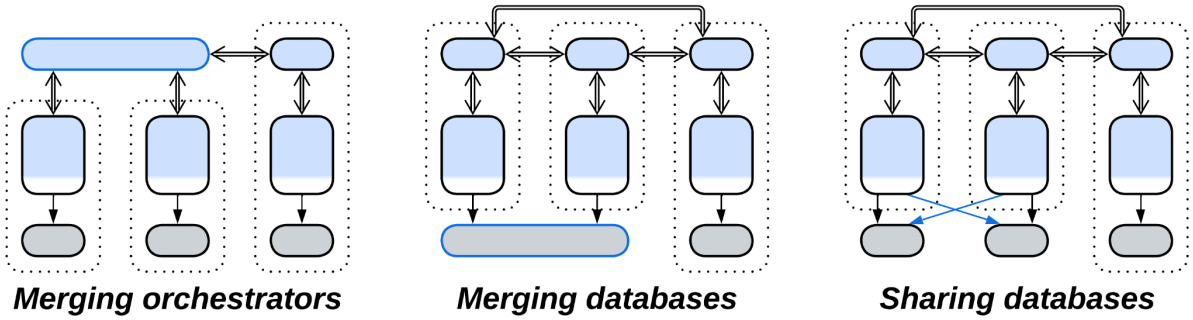
or by building derived datasets:

- A *materialized view* inside a service aggregates event sourcing from other services the owner is interested in.
- A dedicated *query service* captures the whole system's state by subscribing to event sourcing from all the services.
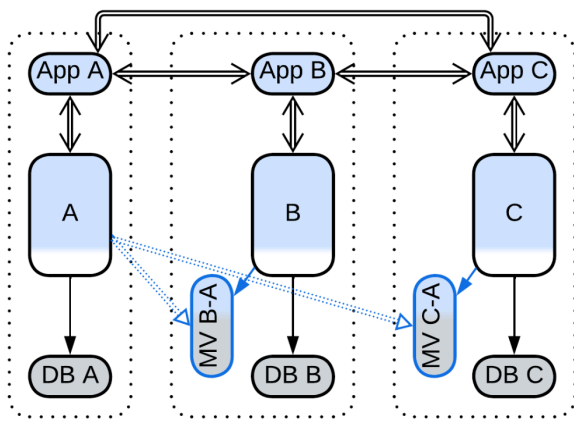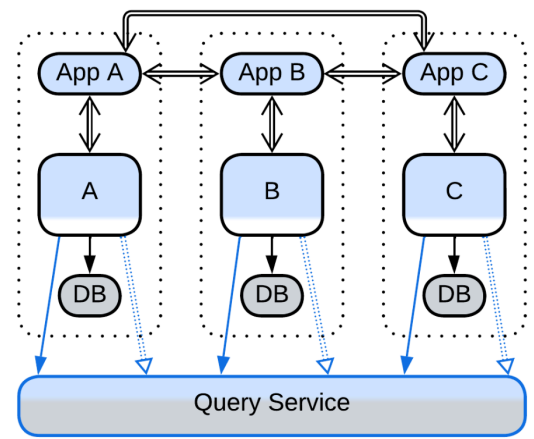
Materialized views   **Query service**

If the services become too large:

- The middle layer can be split into *cells*.

An overgrown service may be turned into a cell

Choreographed two-layered services

Two-layered services make a system of pipelines

The high-level logic resides in the topology of event channels

Use cases may run in different directions
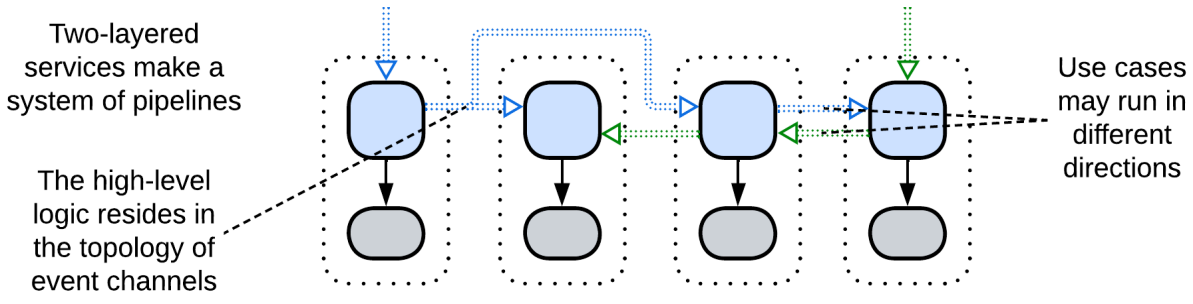
If there is no _orchestration_, there is no role for the *application* layer. _Choreographed_ systems are made up of services that implement individual steps of request processing. The sequence of actions (integration logic) which three-layered systems put in the _orchestrators_ now moves to the graph of *event channels* between the services. This means that with *choreography* the high-level part of the business logic exists outside of the code.

Dependencies

Dependencies are identical to those of a _pipeline_ / _choreographed_ *services* except that each service also depends on its *database*.

Relations

Two-layered services:
- Implement _Pipeline_.
- Are derived from _Layers_ and _Pipeline_.

Evolutions

If *Choreographed Layered Services* become coupled:
- The *business logic* of two or more services can be merged together, resulting in _polyglot persistence_.
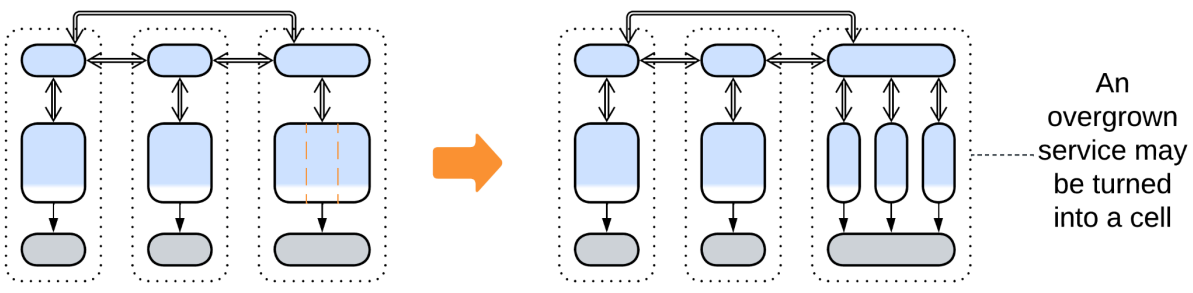- Some databases can be united into a _shared database_ or shared as a _polyglot persistence_.

Merging services   **Merging databases**   **Sharing databases**
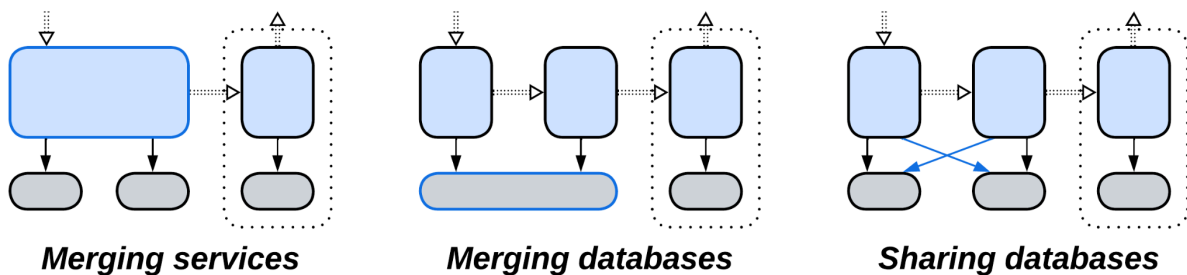
Materialized views or _query services_ are also an option:

Materialized views   **Query service**
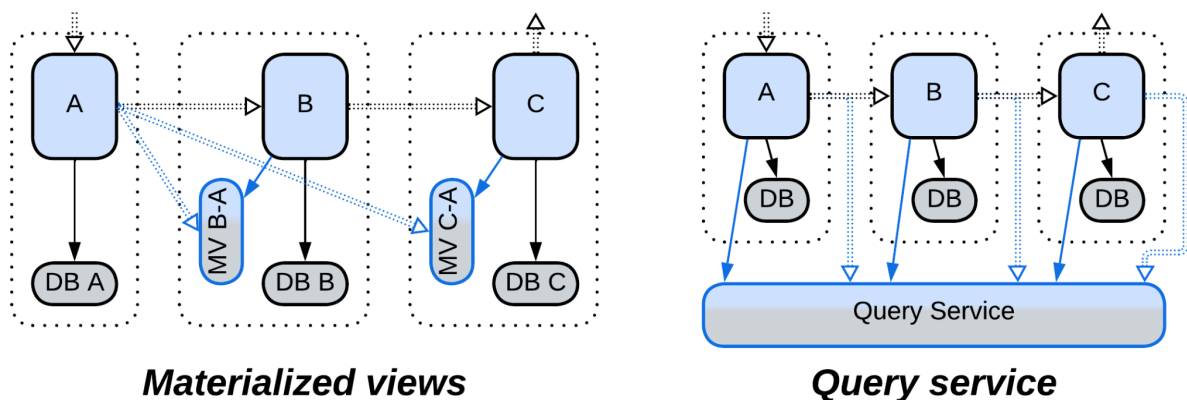
An overgrown service can be:
- Split in two

An overgrown service may be split in two

Command Query Responsibility Segregation (CQRS)

Read and write requests from clients go to different services

The write-enabled database replicates changes to the read-only databases(s)

Each service uses a database optimized for its kind of requests

The read-only database is scalable
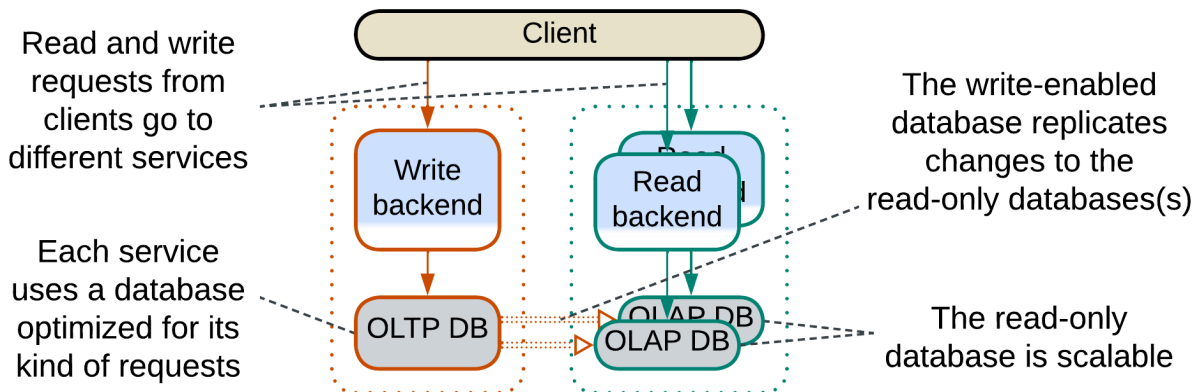
CQRS [MP, LDDD] is, essentially, a division of a *layered* application or a service into two (rarely more) *services*, one of which is responsible for write access (handling *commands*) to the domain data while the other(s) deal with read access (*queries*), creating a data *pipeline* (see the diagram below). Such an architecture makes sense when the write and read operations don't rely on a common vision (model) of the domain, for example, the writes are individual changes (OLTP) that require cross-checks and validation of input, while reads show aggregated data (OLAP) and may take long time to complete (mean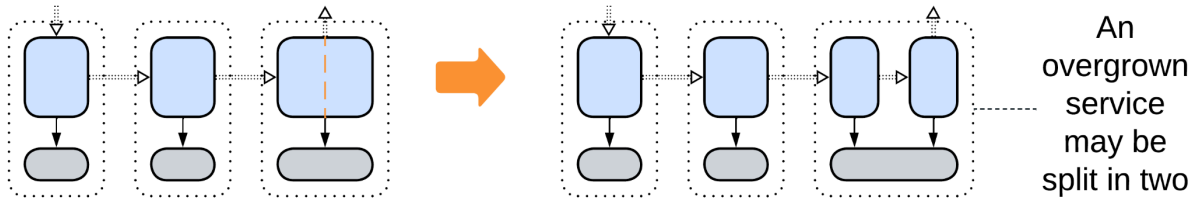ing that *forces* for the read and write paths differ). If there is nothing to share in the code, why not separate the implementations?

CQRS is a pipeline for changes in the data
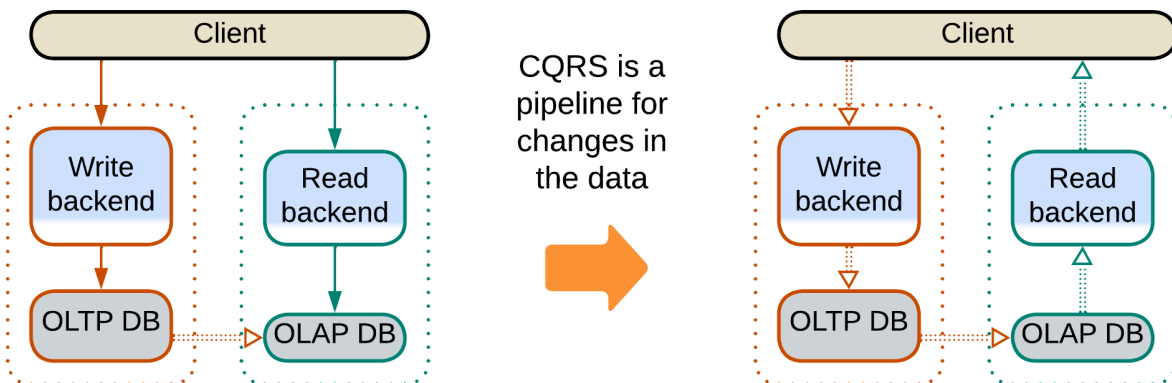
The separation brings in the pros and cons of *Services*: commands and queries may differ in technologies (including optimal database schemas and engines), forces and teams at the expense of consistency (database replication delay) and the system's complexity. In addition, for read-heavy applications, the read database(s) can be easily scaled.

CQRS has several variations:

- The database may be shared, commands and queries may use dedicated databases, or the read service may maintain a *memory image* / *materialized view*

[DDIA] fed by the events from the write service (as in other kinds of *Layered Services*).

- Data replication may be implemented as a *pipeline* between the databases (based on nightly snapshots or log-based replication) or a direct event feed from the OLTP code to the OLAP database.

CQRS through replication   **CQRS through event sourcing**   **CQRS through a shared database**
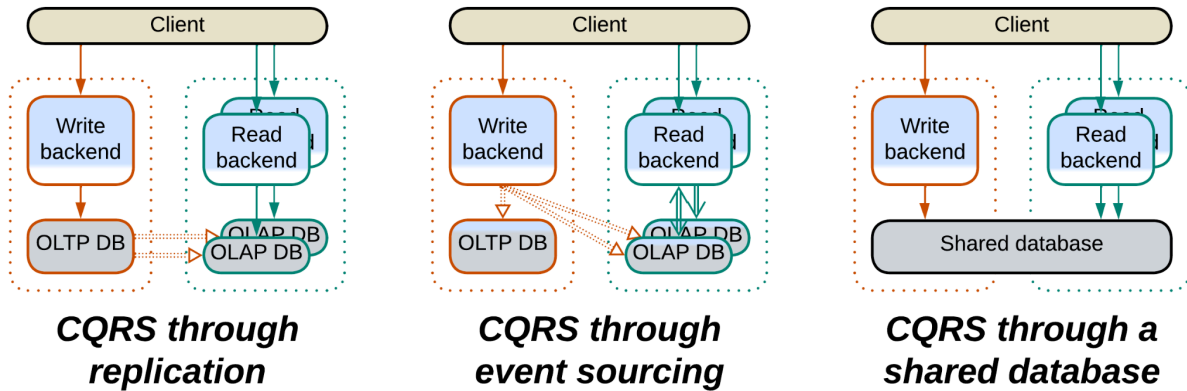
It is noteworthy that while ordinary *layered services* usually intercommunicate through their upper-level components that orchestrate use cases, a *CQRS* system is held together by spreading data changes through its lowest layer.

Examples: Martin Fowler has a short article and Microsoft a longer one.

Dependencies

Each backend depends on its database (technology and schema). The OLTP to OLAP data replication requires an additional dependency, which corresponds to way the replication is implemented:

CQRS through replication   **CQRS through event sourcing**   **CQRS through a shared database**
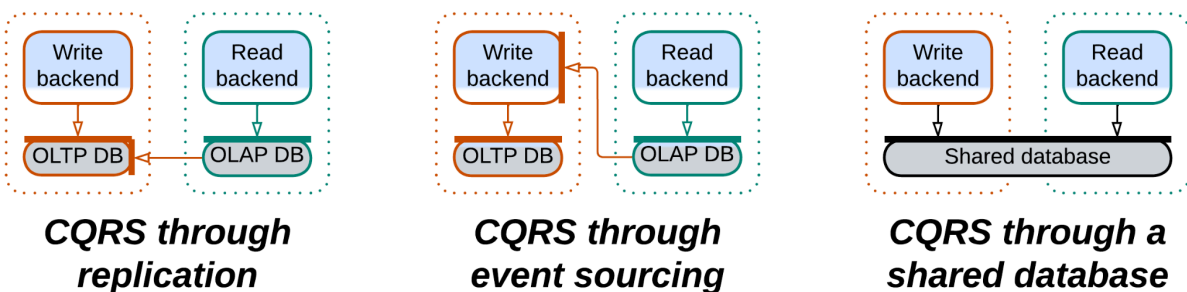
Relations

CQRS:
- Implements *Monolith (*a whole *system* or a *service)*.
- Is derived from *Layers* and *Pipeline*.
- Is a development of *Polyglot Persistence*.

Evolutions

- You will usually need a *reverse proxy* or an *API gateway* to dispatch commands and queries to different services.
- If the commands and queries become intermixed, the business logic can be merged together but the databases are left separate, resulting in *Polyglot Persistence*.
- Both read and write backends can be split into *layers* or *services*.

- *Space-Based Architecture* may further improve performance.
- You can use multiple schemas or even kinds of OLAP databases simultaneously (*Polyglot Persistence*).
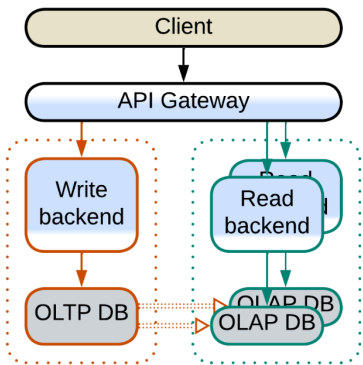
CQRS with an API Gateway
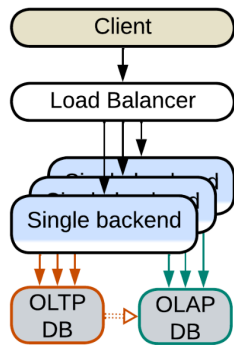
CQRS as Polyglot Persistence
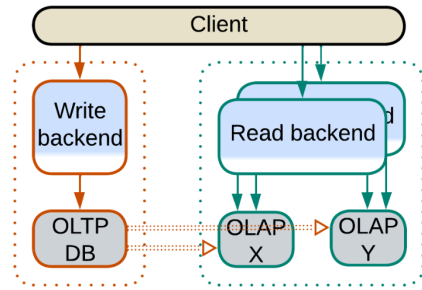
CQRS with OLAP Polyglot Persistence
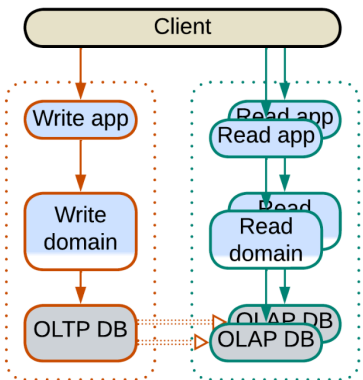
Layered CQRS
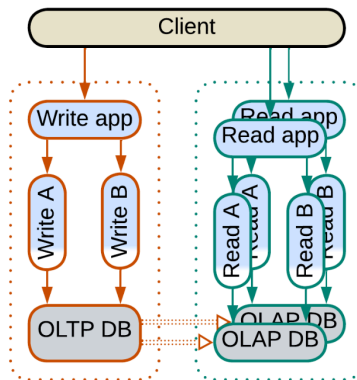
CQRS Cells
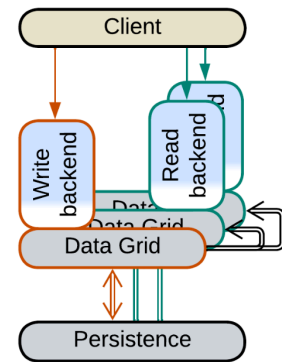
CQRS via Space-Based Architecture

Summary

Layered Services is an umbrella pattern that conjoins:
- Three-layered services where each service *orchestrates* other services.
- Two-layered services that form *pipelines*.
- *CQRS* that separates read and write request processing paths.

Polyglot Persistence

Polyglot Persistence
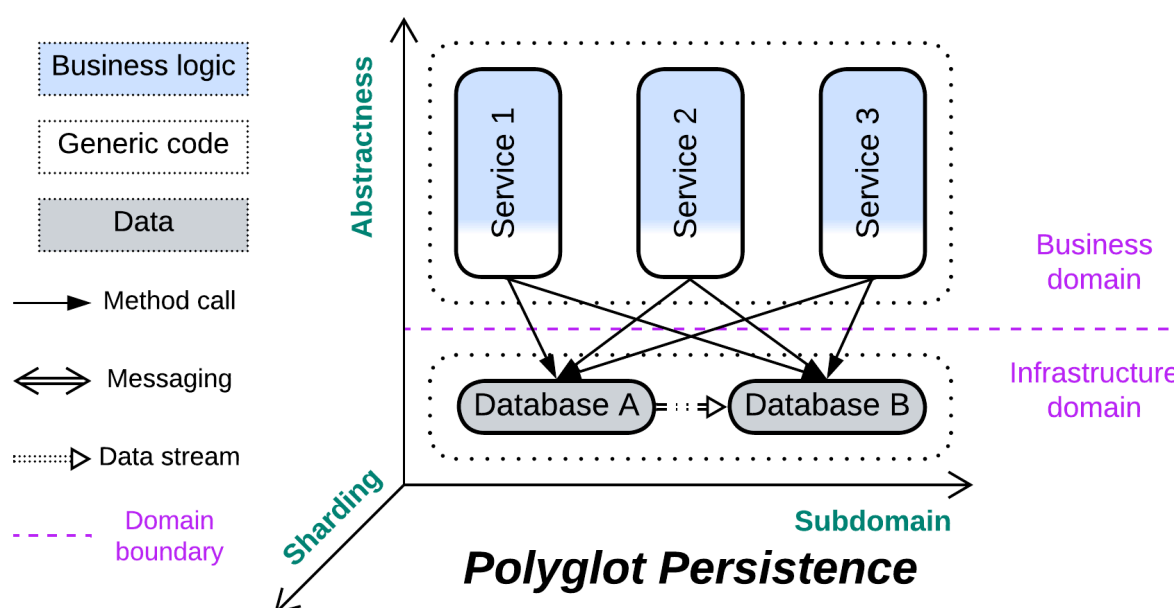
Unbind your data. Use multiple specialized databases.
Known as: Polyglot Persistence.
Aspects: those of the databases involved.
Variants:
Independent storage:
- Specialized databases,
- Private and shared databases,
- Data file / Content Delivery Network (CDN).

Derived storage:
- Read-only replica,
- Reporting Database / CQRS View Database [MP] / Source-aligned (native) Data Product Quantum (DPQ) of Data Mesh [SAHP],
- Memory Image / Materialized view [DDIA],
- Query Service [MP] / Front Controller [SAHP but not PEAA] / Data Warehouse [SAHP] / Data Lake [SAHP] / Aggregate Data Product Quantum (DPQ) of Data Mesh [SAHP],
- Search index,
- Historical data / Data archiving.
- Database cache / Cache-Aside.

Structure: A layer of data services used by higher-level components.
Type: Extension, derived from *Shared Repository*.

Benefits	Drawbacks
Performance is fine-tuned for varied data use cases	Each database needs to be learned
Less load on each database	Much more work for the DevOps team
The databases may satisfy conflicting forces	More points of failure in the system
	Consistency is hard or slow to achieve

References: The original article, the closely related CQRS article, chapter 7 of [MP], chapter 11 of [DDIA] and much information dispersed over the Web.

You can choose a dedicated repository for each kind of data or pattern of data access in your system. That improves performance (as each database engine is optimized for a few use cases), distributes load between the databases and may solve conflicts of forces (like when you need both low latency and large storage). However, you'll likely have to hire several experts to get the best use of and to support the multiple databases. Moreover, having your data spread over multiple databases makes it the application's responsibility to keep the data in sync (by implementing some kind of transactions or making sure that the clients don't get stale data).

Performance

Polyglot Persistence is applied to improve performance. That is achieved through the following means:
- Optimize for specific data use cases. It is impossible for a single database to be good at everything.
- Redirect read traffic to read-only database replicas. The write-enabled master database processes only write requests.
- Cache frequently used data to a fast in-memory database to let the majority of client requests be served without hitting the persistent storage.
- Build a view of the state of other services in the system to avoid querying them.
- Maintain an external index or memory image for use with tasks that don't need the historical data.
- Purge old data to a slower storage.
- Store read-only sequential data as files, often close to end users that download them.

Still, the read-write separation introduces the replication lag, which is a pain when data consistency is important for the system's clients.

Dependencies

In general, each service depends on all the databases it uses. There may also be an additional dependency between the databases if they share the dataset (one or more databases are derived).

Independent storage   **Derived storage**
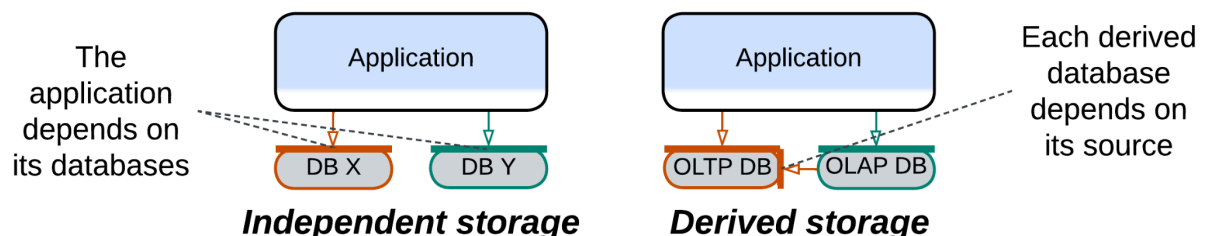
Applicability

Polyglot Persistence helps:
- *High-load and low latency projects.* Specialized databases shine when given fitting tasks. Caching and read-only replicas unload the main database. External indices save the day.

- *Event sourcing.* Materialized views maintain current states of the system's components.
- *Conflicting forces*. An instance of a stateless module inherits many qualities of the database it uses for each request it processes. Thus, if there are several databases, qualities of a service instance may vary from request to request.

Polyglot Persistence may <u>hurt</u>:

- *Small projects.* Properly setting up and maintaining multiple databases is not that easy.
- *High availability*. Each database your system uses tends to fail in its own crazy way.
- *User experience*. For systems with read-write database separation the replication lag between the databases will make you [choose](#) between writing synchronization code to wait for the read database to be updated and risking returning outdated results to the users.

Relations
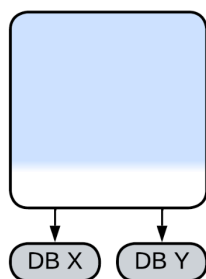
Polyglot Persistence for a monolith   **Polyglot Persistence for layers**   **Polyglot Persistence for shards**   **Polyglot Persistence for services**
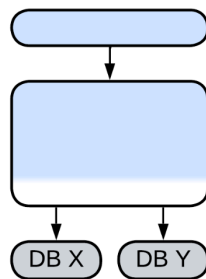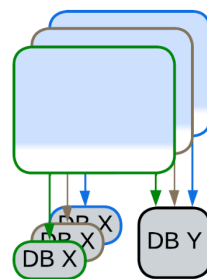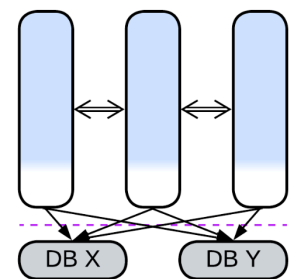
Polyglot Persistence:

- Extends [*Monolith*](#), [*Shards*](#), [*Layers*](#) or [*Services*](#).
- Is derived from [*Layers*](#) (persistence layer) or [*Shared Repository*](#).
- The variant with derived databases inherits from [*Pipeline*](#) and is closely related to [*CQRS*](#).

Variants with independent storage

Many cases of *Polyglot Persistence* use multiple repositories just because there is no single technology that matches all the application's needs. The databases used are filled with different subsets of the system's data:

Specialized databases

Client

Polyglot Persistence is about using several specialized databases

The application may use multiple databases to process a single request

SQL DB   **NoSQL**

If the databases vary in technologies, the application earns the benefits of each of them
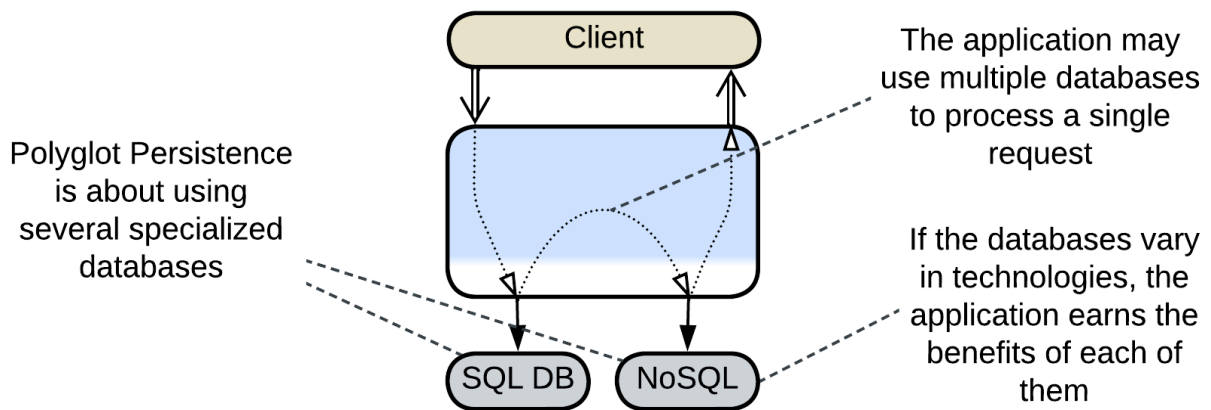
Databases vary in their optimal use cases. You can employ several different databases to have the best performance for each kind of data that you persist.

Private and shared databases

If shards or services become coupled through a part of the system's data ...

Shard 1

Service X

Service Y

... that can be resolved by sharing the data via a dedicated database

DB 3
DB 2
DB 1   Shared DB   DB X   Shared DB   DB Y
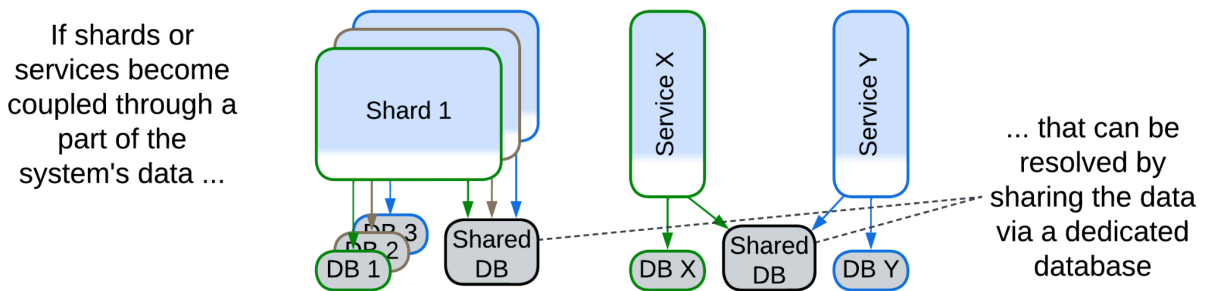
If several *services* or *shards* become coupled through a subset of the system's data, that subset can be put into a separate database which is accessible to all the participants. All the other data remains private to the *shards* or *services*.

Data file, Content Delivery Network (CDN)

Frontend

Backend

Web page templates are often stored as ordinary files

Images and videos are also distributed as files via a content delivery network

Page Template   Database   CDN
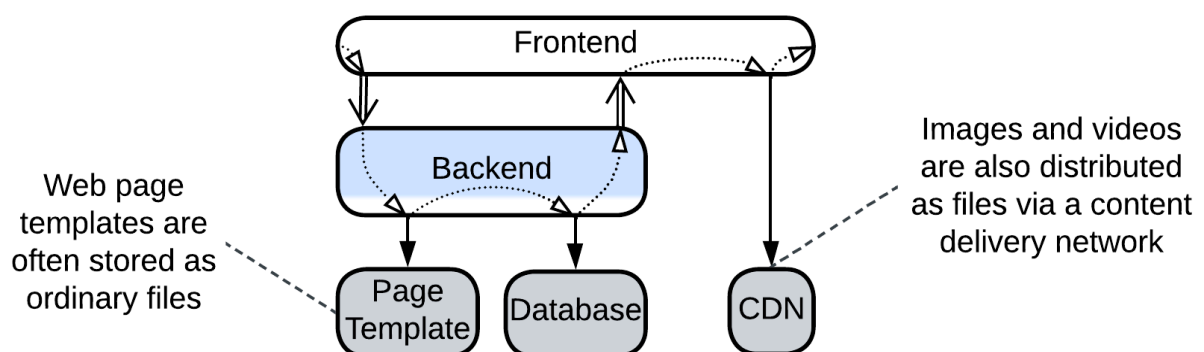
Some data is happy to stay in files. Web frameworks load web page templates from OS files and store images and videos in a *content delivery network (CDN)* that replicates the data all over the world so that each user downloads the content from the closest server (which is faster and cheaper).

Variants with derived storage

In other cases there is a single writable database (*system of record* [DDIA]) which is the main *source of truth* while other databases are derived from it. The primary reason is to relieve the main database of read requests and often support additional qualities: special kinds of queries, aggregation for materialized views, (full text) search for indices, huge dataset size for historical data, low latency for an in-memory cache.

The updates to the derived databases may come from:
- the main database as *change data capture* [DDIA] (which is a log of changes),
- the application after it changes the main database (see caching strategies below),
- another service as *event sourcing* [DDIA, MP] (a stream of application events),
- a dedicated indexer that periodically crawls the main database or web site.

The derived database may be kept up to date by processing the feed of changes to the main database

A dedicated indexer may update the derived database by scanning the main database or the web

The application may update the derived database after a transaction to the main database succeds

The derived database may process an event feed from another system component

Read-only replica

Several identical databases are deployed

A backend can acess any database

Database replicas are read-only

One of them serves write requests - it is called *leader*

The master propagates changes to the replicas

Multiple instances of the database are deployed. One of them is the *leader* [DDIA] instance that accepts all the writes to the data. The changes are replicated to the other

passive instances which are used for read requests. Distributing the workload over multiple instances increases the maximum throughput of the system. Having multiple running *replicas* greatly improves reliability and allows for nearly instant recovery of database failures as any *replica* may quickly be promoted to the *leader* role to serve the write traffic.

Reporting Database, CQRS View Database, Source-aligned (native) Data Product Quantum (DPQ) of Data Mesh

The CQRS View subscribes to events from another service ... to let its host know the other's latest state

A reporting datasase stores its service's data in a format convenient to analysts

It is common wisdom that a database is good for either OLTP (transactions) or OLAP (queries). Here we have two databases: one optimized for commands (write traffic protected with transactions) and an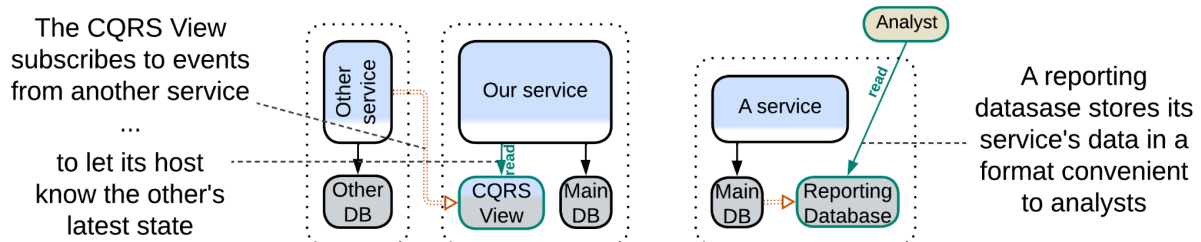other one for complex analytical queries. The databases differ at least in schema (OLAP schema is optimized for queries) and often vary in type (e.g. SQL vs NoSQL).

A *reporting database* (or *Source-aligned (native) Data Product Quantum* of *Data Mesh* [SAHP]) derives its data from a write-enabled database in the same subsystem (service) while a *CQRS view* [MP] is fed a stream of events from another service from which it filters the data of interest to its owner service (see the last case on the diagram above). This way the *CQRS view* lets its owner service query (its replica of) the data that originally belonged to other services.

Memory Image, Materialized view

On starting up the service replays events from its event store to recreate its state in the memory image

Then the memory image becomes the service's primary data, but all the changes are persited to the event store
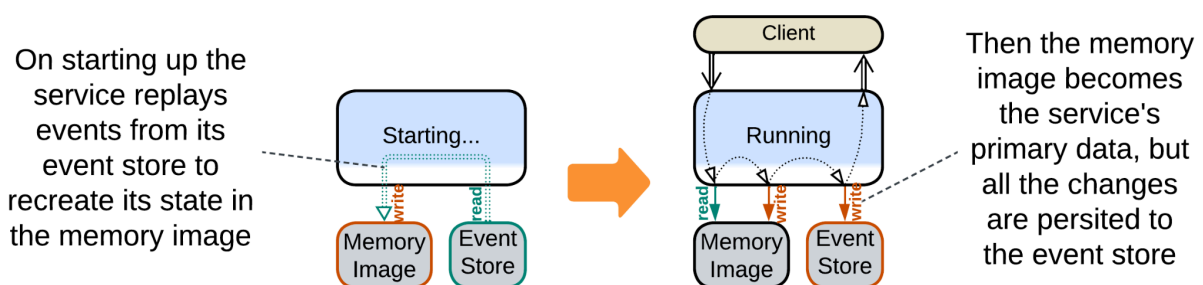
Event sourcing (of *Event-Driven Architecture* or *Microservices*) is all about changes. Services publish and persist whatever events happen in the system, forgoing the current state in favor of the history of changes. As a result, a service needs to aggregate the history into a *memory image* (*Materialized View* [DDIA]) by loading a snapshot and replaying any further events to rebuild the current state (which other architectural styles store in databases) to become up-to-date and start operating.

Query Service, Front Controller, Data Warehouse, Data Lake, Aggregate Data Product Quantum (DPQ) of Data Mesh

Each service can read the system's state from the query service

The query service subscribes to and aggregates events from all the services
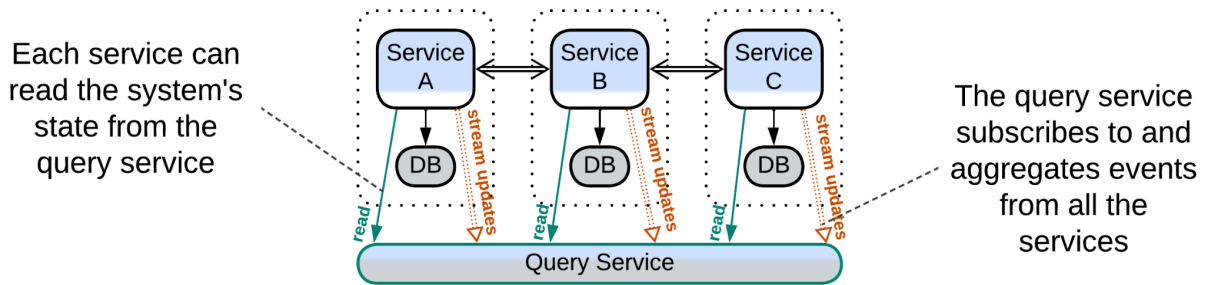
A *query service* [MP] (or *Aggregate Data Product Quantum* of *Data Mesh* [SAHP]) subscribes to events from several full-featured services and aggregates them to its database, making it a *CQRS view* of several services or even the whole system. If any other service or a data analyst needs to process data that belongs to multiple services, it retrieves it from the *query service* which has already joined the streams and represents the join in a convenient way through a single API method.
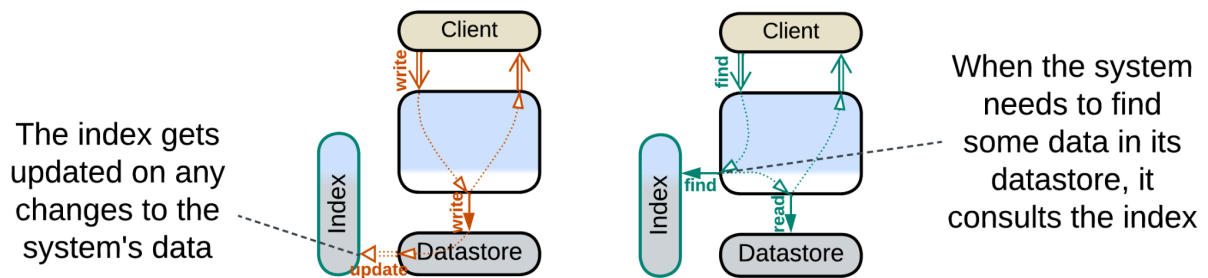
Front Controller [SAHP but not PEAA] is a *query service* embedded in the first (user-facing) service of a *pipeline*. It collects status updates from the downstream components of the pipeline to know the state of every request being processed by the pipeline.

Data Warehouse [SAHP] and *Data Lake* [SAHP] are analytical databases that connect directly to and import all the data from the (operational) databases of all the system's services. *Data Warehouse* translates the imported data to its own unified schema while *Data Lake* stores the imported data in its original formats.

Search index

The index gets updated on any changes to the system's data

When the system needs to find some data in its datastore, it consults the index

Some domains require a kind of search which is not naturally supported by ordinary database engines. Full text search, especially NLP-enabled, is one such case. Geospatial data may be another. If you are comfortable with your main database(s), you can set up an external search index by deploying a product dedicated to the special kind of search that you need and feeding it updates from your main database.

Historical data, Data archiving

The system's main database may become overburdened with old data

To avoid that an archiver script moves the old data to a cheaper archive storage
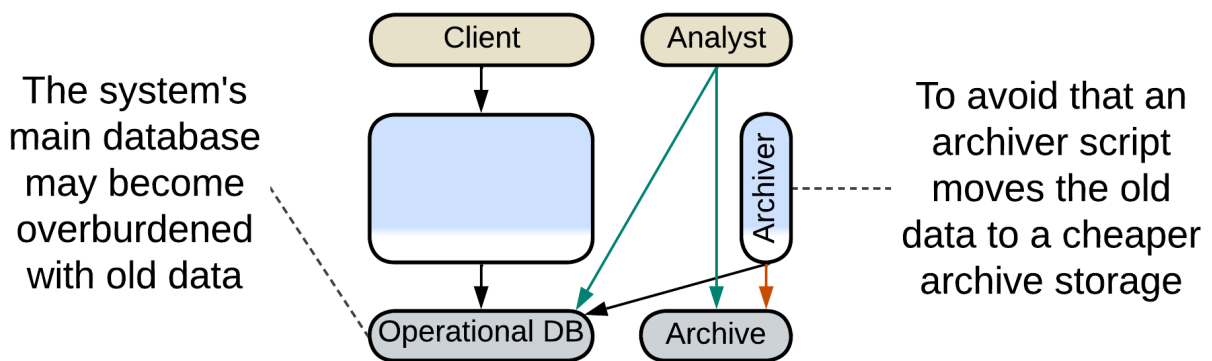
It is common to store the history of sales in a database. However, once a month or two has passed, it is very unlikely that the historical records will ever be edited. And though they are queried on very rare occasions, like audits, they still slow down your database. Some businesses offload any data older than a couple of months to a cheaper archive storage which does not allow for changes to the data and has limited query capabilities – in order to keep their main datasets small and fast.

Database cache, Cache-Aside

Cache hit   **Cache miss**   **Write**

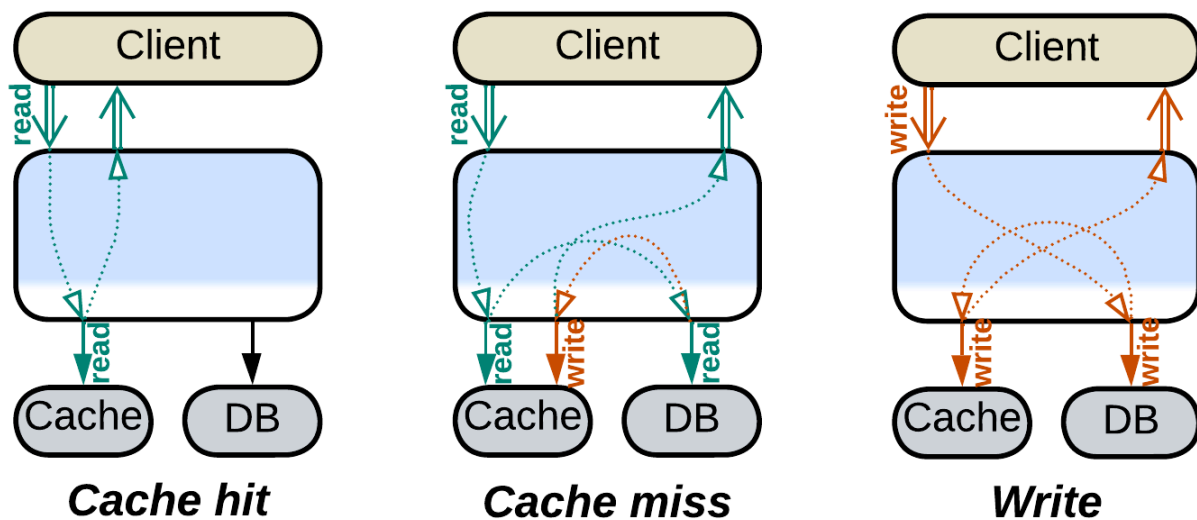Database queries are resource-heavy while databases scale only to a limited extent. That means that a highly loaded system benefits from bypassing its main database in as many queries as possible, which is usually achieved by storing recent queries and their results in an in-memory database (*cache-aside*). Each incoming query is first looked in the fast cache, and if it is found then you are lucky to return the result immediately without turning to the main database.

Keeping the *cache* consistent with the main database is the hard part. There are quite a few strategies (some of them treat the cache as a *proxy* for the database): write-through, write-behind, write-around and refresh-ahead.

Evolutions

Polyglot Persistence with derived storage can often be made subject to *CQRS*:

- The service that uses the read and write databases is [split into separate read and write services](#).

CQRS processes read and write requests in separate services
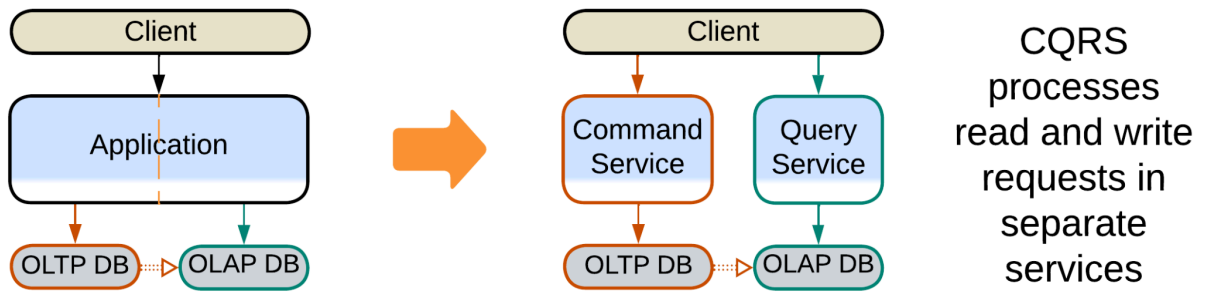
Summary

Polyglot Persistence employs multiple specialized databases to improve performance, often at the cost of implementing transactions in the application or delayed data replication.

Backends for Frontends (BFF)

Backends for Frontends
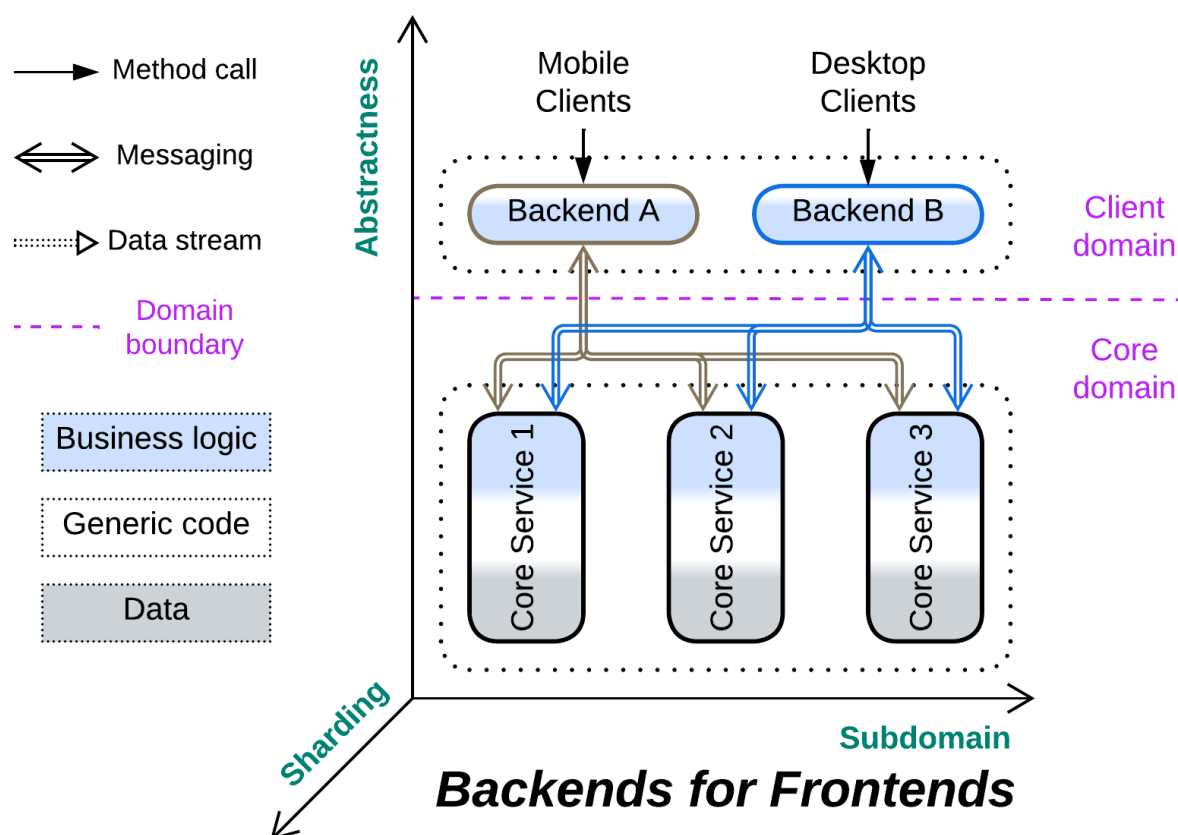
Hire a local guide. Dedicate an orchestrator to every kind of client.

Known as: Backends for Frontends (BFF), Layered Microservice Architecture.

Aspects:

● Proxy,
● Orchestrator.

Variants: Applicable to:

● Proxies,
● Orchestrators,
● Proxy + Orchestrator pairs,
● API Gateways,
● Event Mediators.

Structure: A layer of integration services over a shared layer of core services.

Type: Extension, derived from Orchestrator and/or Proxy.

Benefits	Drawbacks
Clients become independent in protocols, workflows and, to an extent, forces	No single place for cross-cutting concerns
A specialized team and technology per client may be employed	More work for the DevOps team
The multiple *orchestrators* are smaller and more cohesive than the original universal one	

References: The original article and a smaller one from Microsoft. Here are reference diagrams from WSO2 (notice multiple Microgateway + Integration Microservice pairs).

If some aspect(s) of serving our system's clients strongly vary by client type (e.g. OLAP vs OLTP, user vs admin, buyer vs seller vs customer support), it makes sense to use a dedicated component (the titular *backend for frontend* or *BFF*) per client type to encapsulate the variation. Protocol variations call for multiple *proxies*, workflow variations – for several *orchestrators*, both coming together – for *API gateways* or *proxy + orchestrator* pairs. It is even possible to vary the BFF's programming language on a per client basis. The drawback is that once the clients get their dedicated BFFs it becomes hard to share a common functionality between them, unless you are willing to add yet another new utility service that can be used by all of them (and strongly smells of *SOA*).

Performance

As the multiple *orchestrators* of *BFF* don't communicate with each other, the pattern's performance is identical to that of *Orchestrator*: it also slows down request processing in the general case, but allows for several specific optimizations, including the use of direct communication channels between the orchestrated *services*.

Dependencies

Each *BFF* depends on all the services it uses (usually every service in the system). The services themselves are likely independent, as is common in *orchestrated* systems.

Each BFF depends on all the services

BFF X   BFF Y

Service A   Service B   Service C

The services are independent
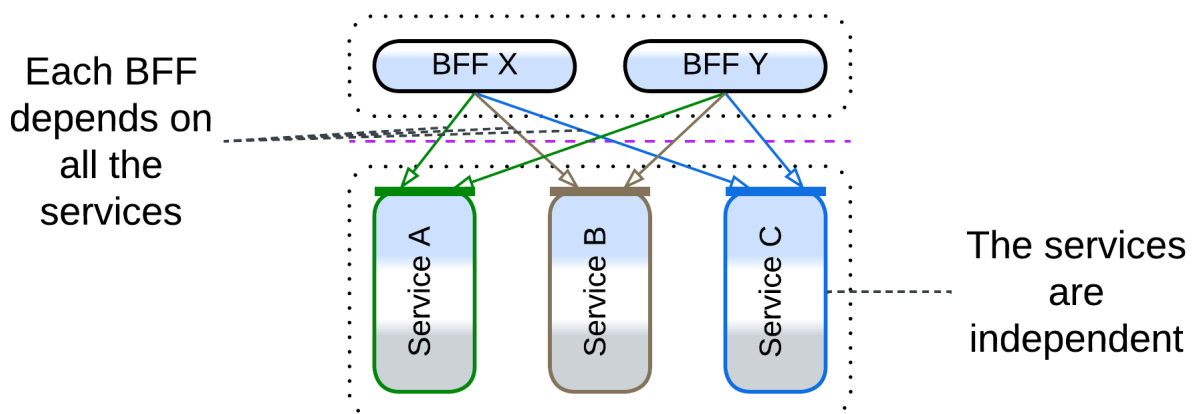
Applicability

Backends for Frontends are good for:
- *Multiple client protocols.* Deploying a *gateway* per protocol hides the variation from the underlying system.
- *Multiple UIs.* When you have a team per UI, each of them may want to use an API which they feel comfortable with.
- *Drastically different workflows.* Let each client-facing development team own a component and choose the best fitting technologies and practices.

Backends for Frontends should be avoided when:
- *The clients are mostly similar.* It is hard to share code and functionality between *BFF*s. If the clients have much in common, the shared aspects either find their place in a shared monolithic layer (e.g. multiple client protocols call for multiple *gateways* but a shared *orchestrator*) or are duplicated. Maybe *BFF* is a bad choice – use OOD

(conditions, factories, strategies, inheritance) to account for the clients' differences within a single codebase.

Relations

BFF for a monolith **BFF for layers** **BFF for shards** **BFF for services**
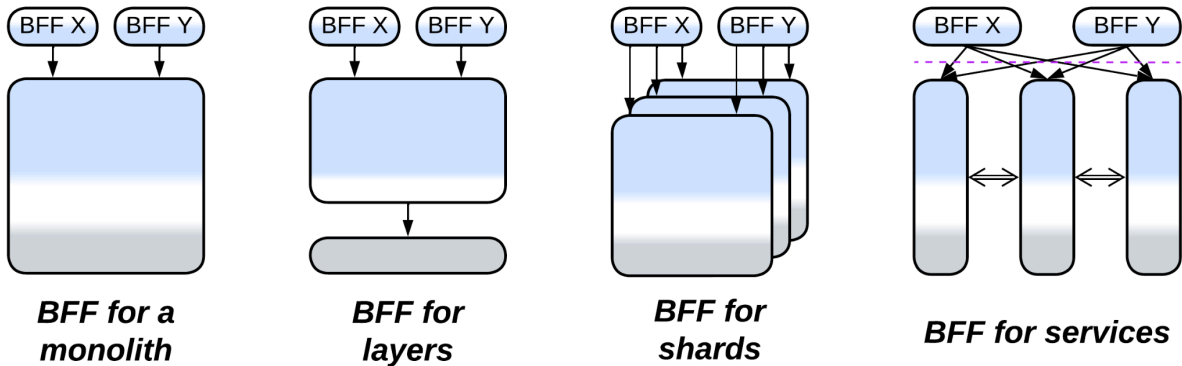
Backends for Frontends:
- Extends *Services* or rarely *Monolith*, *Layers* or *Shards*.
- Is derived from a client-facing extension metapattern: *Gateway*, *Orchestrator*, *API Gateway* or *Event Mediator*.

Variants

Backends for Frontends vary in the kind of component that gets dedicated to each client:

Proxies

If clients vary in protocol or security ...

... you may dedicate a gateway per client type

The services rely on choreography
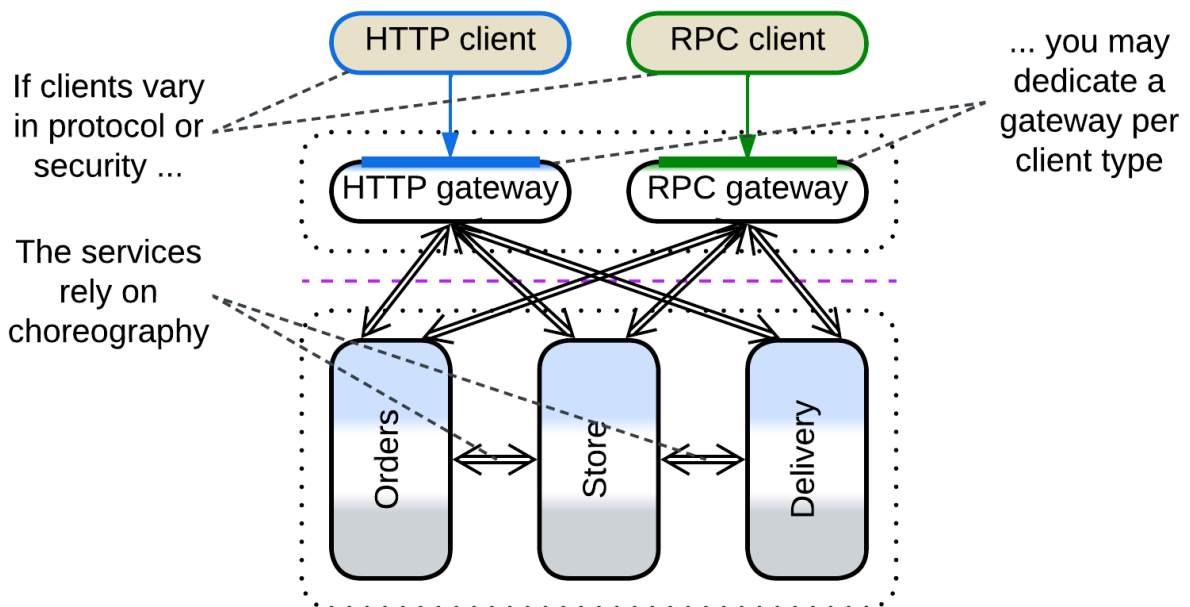
Dedicating a *gateway* per client is useful when the clients differ in the mode of access to the system (protocols / encryption / authorization) but not in workflows.

Multiple *adapters* match the literal meaning of "*Backends for Frontends*" – each UI team (backend, mobile, desktop; or end-device-specific teams) gets some code on the backend side to adapt the system's API to its needs by building a new, probably more high-level specialized API on top of it.

Orchestrators

Buyer client   Seller client

If clients differ
in functionality
...

... an
orchestrator
per client is
written

Buyer logic   Seller logic

The services are
orchestrated and
don't
intercommunicate

Orders   Store   Delivery
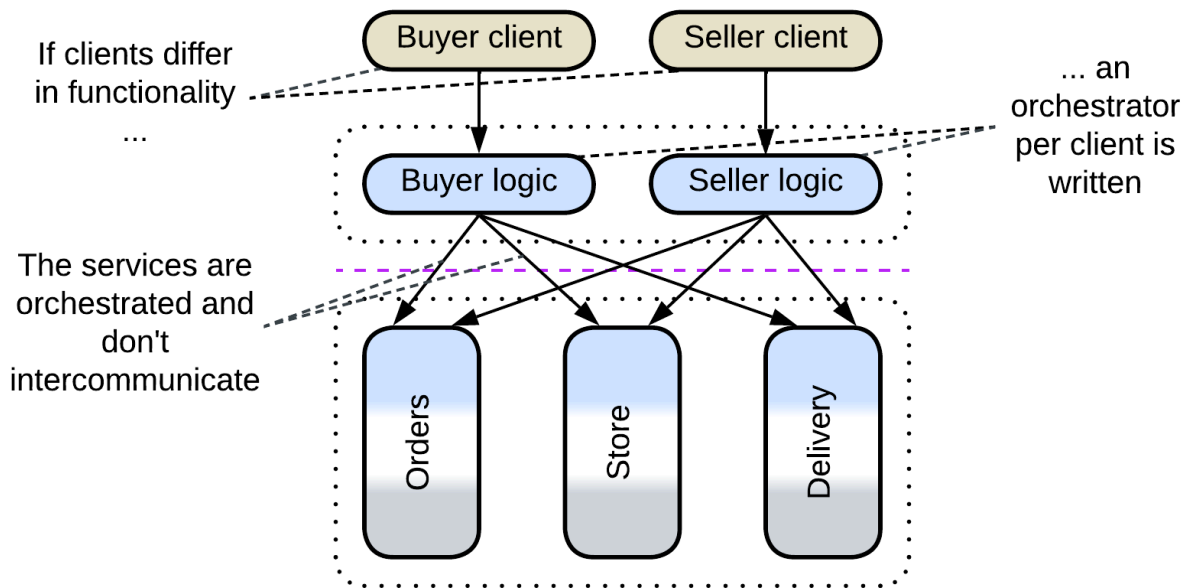
An *orchestrator* per client makes sense if the clients work in the system in completely unrelated ways, e.g. a shop's customers have little to share with its administrators.

Proxy + Orchestrator pairs

Buyers from
the Internet   Sellers via
VPN

If clients
differ in both
protocol and
type ...

... you may
dedicate a
gateway per
client type to
deal with the
protocol ...

Firewall   VPN endpoint

Buyer logic   Seller logic

... and an
orchestrator
per client for
business
logic

Orders   Store   Delivery
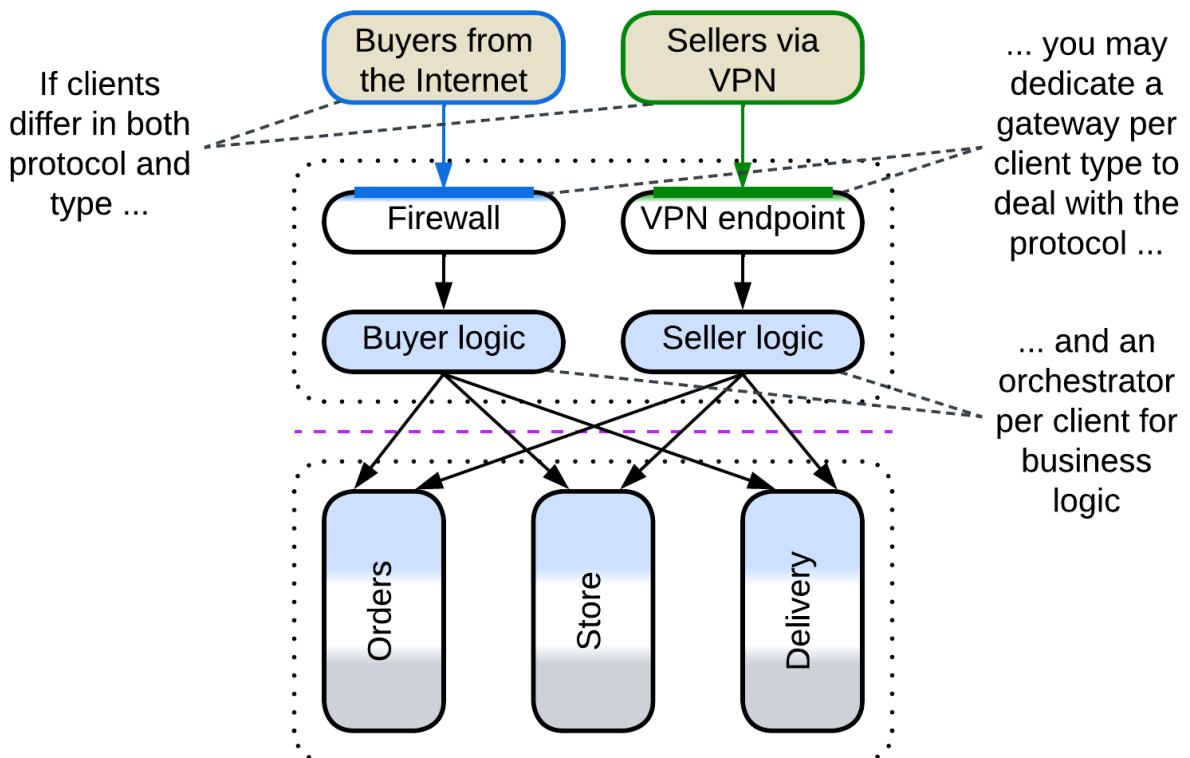
Clients vary in both access mode (protocol) and workflow. *Orchestrators* or *proxies* may be reused if there are many kinds of clients.

API Gateways

If clients differ in both protocol and type ...

Buyers from the Internet   Sellers via VPN

API gateway for external buyers   API gateway for internal sellers

Orders   Store   Delivery

... an API gateway per client type is an option
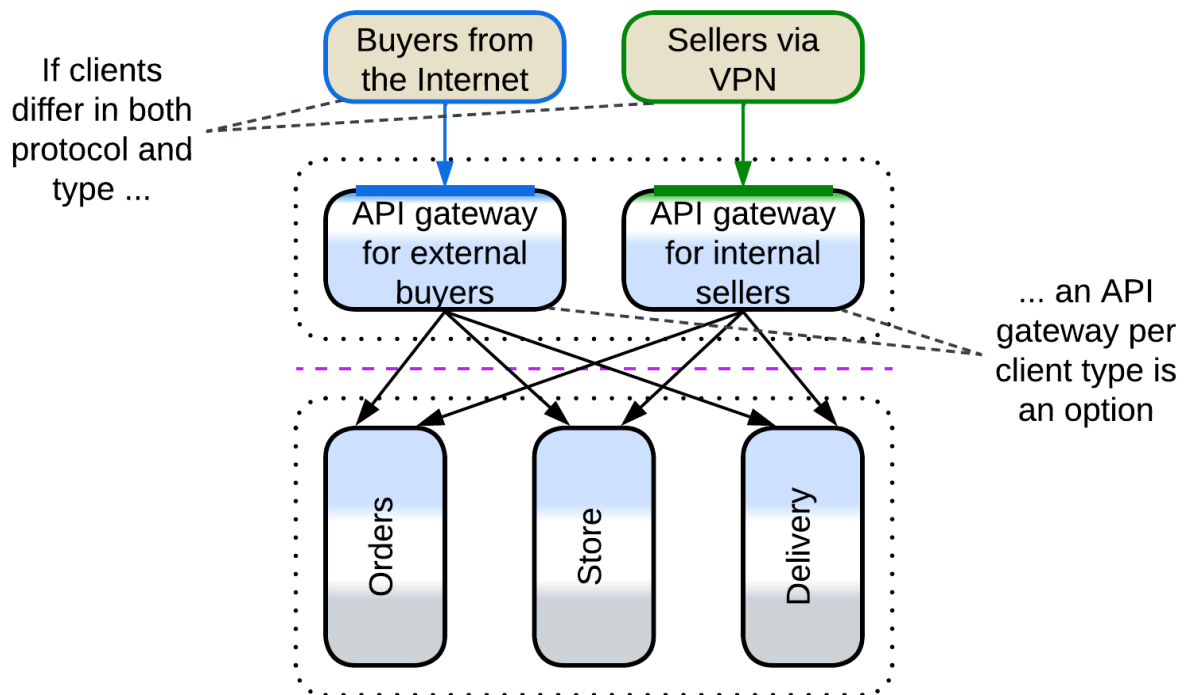
Clients vary in access mode (protocol) and workflow and there is a 3rd party *API Gateway* framework which seems to fit your requirements off the shelf.

Event Mediators

If clients of an event-based system differ in functionality ...

Buyer client   Seller client

Buyer mediator   Seller mediator

Orders   Store   Delivery

... having an event mediator per client type may improve the code and stability
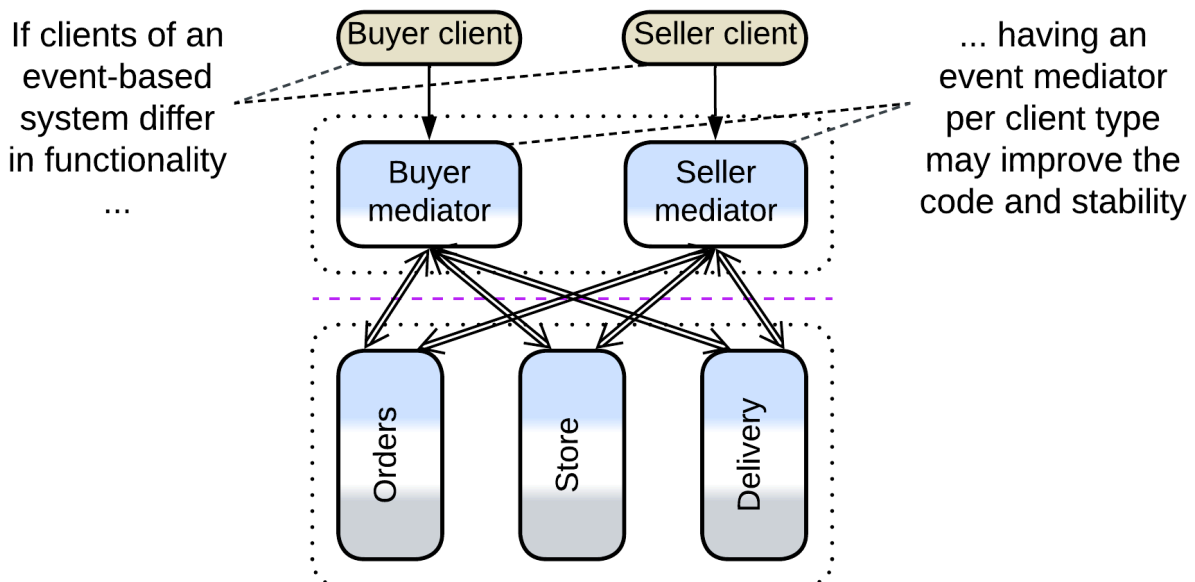
[FSA] mentions that multiple *event mediators* may be deployed in *Event-Driven Architecture* to split the codebase and improve stability.

Evolutions

BFF-specific evolutions aim at sharing logic between the *BFF*s:

- The *BFF*s can be merged into a single [orchestrator](#) if their functionality becomes mostly identical.
- A shared *orchestration [layer](#)* with common functionality may be added for use by the *BFF*s.
- A layer of *integration services* under the *BFF*s simplifies them by providing shared high-level APIs for the resulting [cells](#).
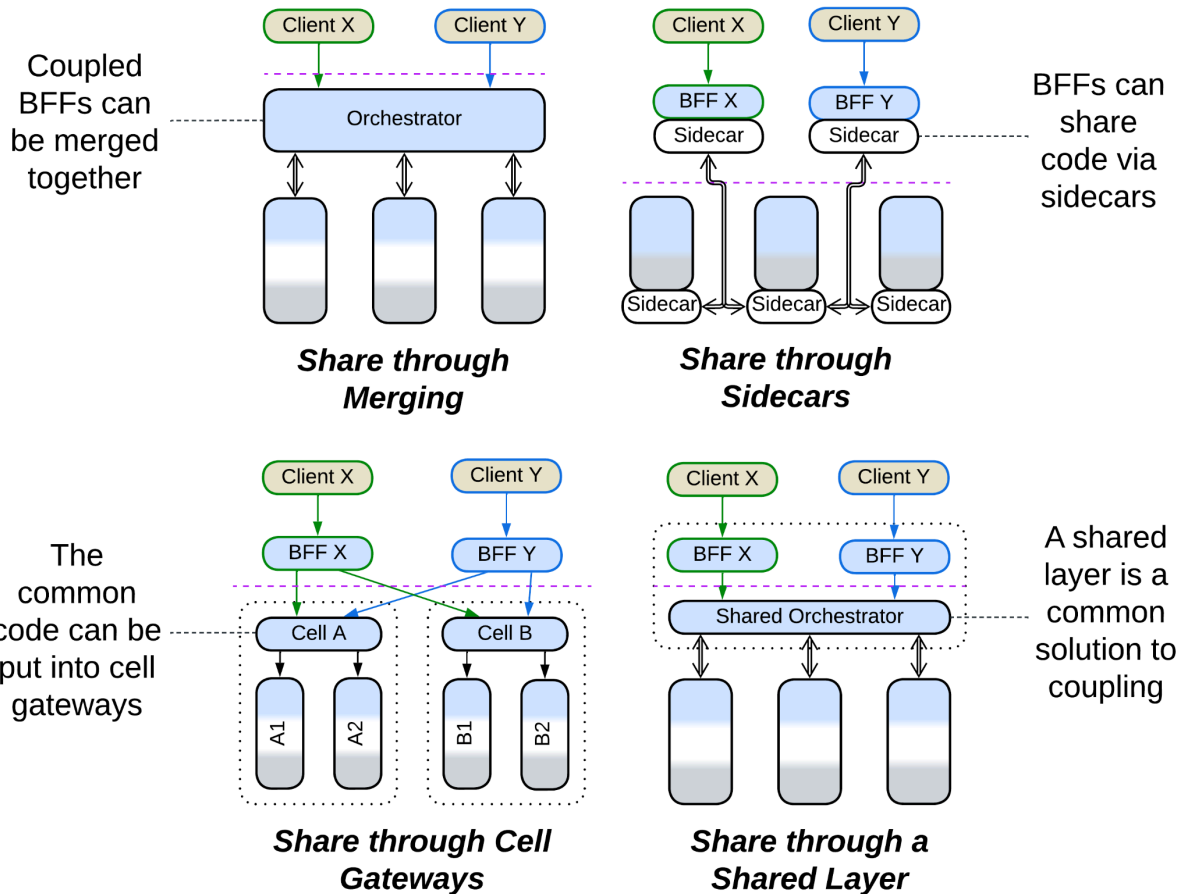- *Sidecars* [[DDS](#)] of [*service mesh*](#) can share libraries among the *BFF*s.

Coupled BFFs can be merged together

Share through Merging

BFFs can share code via sidecars

Share through Sidecars

The common code can be put into cell gateways

Share through Cell Gateways

A shared layer is a common solution to coupling

Share through a Shared Layer

Summary

Backends for Frontends assigns a [proxy](#) and/or [orchestrator](#) per each kind of a system's client to encapsulate client-specific use cases and protocols. The drawback is that there is no good way for sharing functionality between the *BFF*s.

Service-Oriented Architecture (SOA)
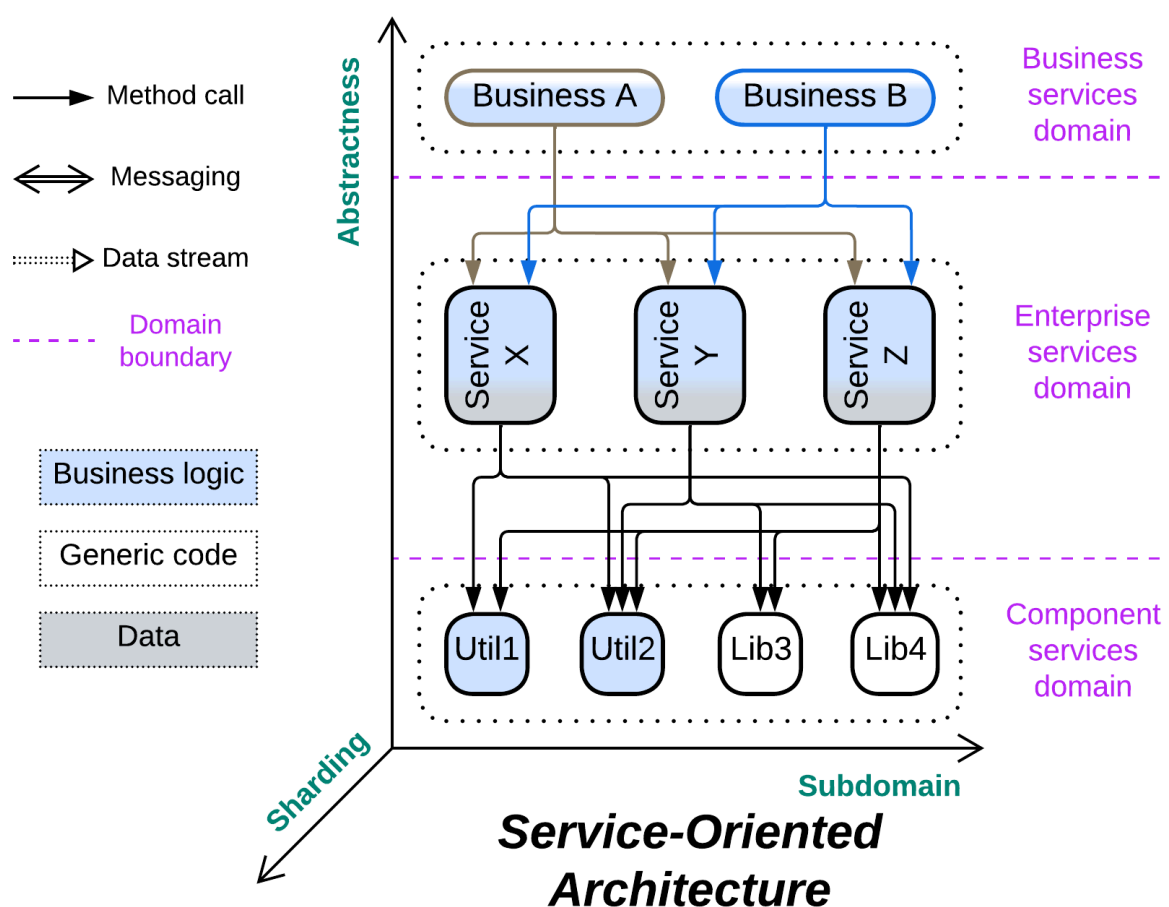
Service-Oriented Architecture

The whole is equal to the sum of the parts. Distributed Object-Oriented Design.

Known as: Service-Oriented Architecture (SOA), Segmented Architecture.

Variants:

- Distributed Monolith,
- Enterprise SOA,
- (misapplied) Automotive SOA,
- Nanoservices.

Structure: Usually 3 layers of services where each service can access any other service in its own or lower layers.

Type: Main, derived from *Layers*.

Benefits	Drawbacks
Supports huge codebases	Very hard to debug
Multiple development teams and technologies	Hard to test as there are many dependencies
Forces may vary between modules	Very poor latency
Deployment to dedicated hardware	Very high DevOps complexity
Fine-grained scaling	The teams are highly interdependent

References: [FSA] has a chapter on Orchestration-Driven (Enterprise) Service-Oriented Architecture. [MP] mentions Distributed Monolith. There is also much (though somewhat conflicting) content over the Web.

Service-Oriented Architecture looks like the application of modular or object-oriented design followed by distribution of the resulting components over a network. The system usually contains 3 (rarely 4) *layers* of *services* where every service has access to all the services below it (and rarely some in its own layer). The services stay small, but as their number grows it becomes hard to cognize all the API methods and contracts available for the component's use. Another issue comes from the idea of reusable components – multiple applications, written for different clients with varied workflows, require the same service to behave in (subtly) different ways, either causing its API to bloat or impairing its usability (which means that a new customized duplicate service will be added to the system). Use cases are slow because there is much interservice communication over the network. Teams are interdependent as any use case involves many services, each owned by a different team. Testability is poor because there are too many moving (and being independently updated!) parts. The foundational idea of service reuse failed in practice, but its child architecture, *SOA*, still survived in historical environments.

Even though *SOA* fell from grace and is rarely seen in modern projects, it may soon be resurrected by low-code and no-code frameworks for serverless systems (e.g. *Nanoservices*) – it has everything ready: code reuse, granular deployment and elastic scaling.
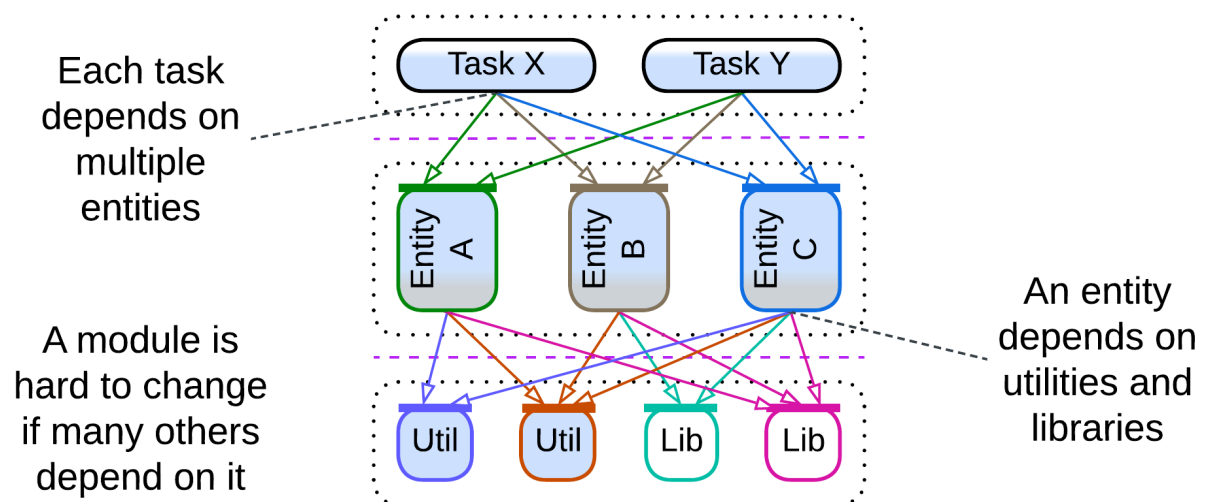
Performance

SOA is remarkable for its poor latency that results from extensive communication between its distributed components. There is hardly any way to help that as processing a request usually involves multiple services from all the layers.

Nevertheless, the pattern allows for good throughput as its stateless components can be scaled individually, leaving the system's scalability to be limited only by its databases, *middleware* … and funding.

Dependencies

Each service of each layer depends on everything it uses. As a result, development of a low-level (utility) component may be paralyzed because too many services already use it, thus no changes are welcome. Hence, the team writes a new version of their utility as a new service, which defeats the idea of component reuse which *SOA* was based on.

Each task depends on multiple entities

A module is hard to change if many others depend on it

An entity depends on utilities and libraries

Applicability

Service-Oriented Architecture <u>is useful</u> in:

- *Huge projects.* Multiple teams can be employed, each handling a moderate amount of code. However, dependencies between the teams and the combined length of the APIs in the system may [stall the development](#) anyway.
- *A system of specialized hardware devices.* If there is a lot of different hardware interacting in complex ways, the system may naturally fit the description of *SOA*. Don't fight this kind of [Conway's law](#).

Service-Oriented Architecture <u>hurts</u>:

- *Fast-paced projects.* Any feature requires coordination of multiple teams, which is hard to achieve in practice.
- *Latency-sensitive domains.* Over-distribution means too much messaging causing too high latency.
- *High availability systems.* Components may fail. A failure of a lower-level component is going to stall a large part of the system because every low-level component is used by many high-level services.
- *Life-critical systems with frequent updates.* SOA is hard to test comprehensively. Either all the components must be certified with a strict standard and a comprehensive test suite or any single component update requires re-testing of the entire system.
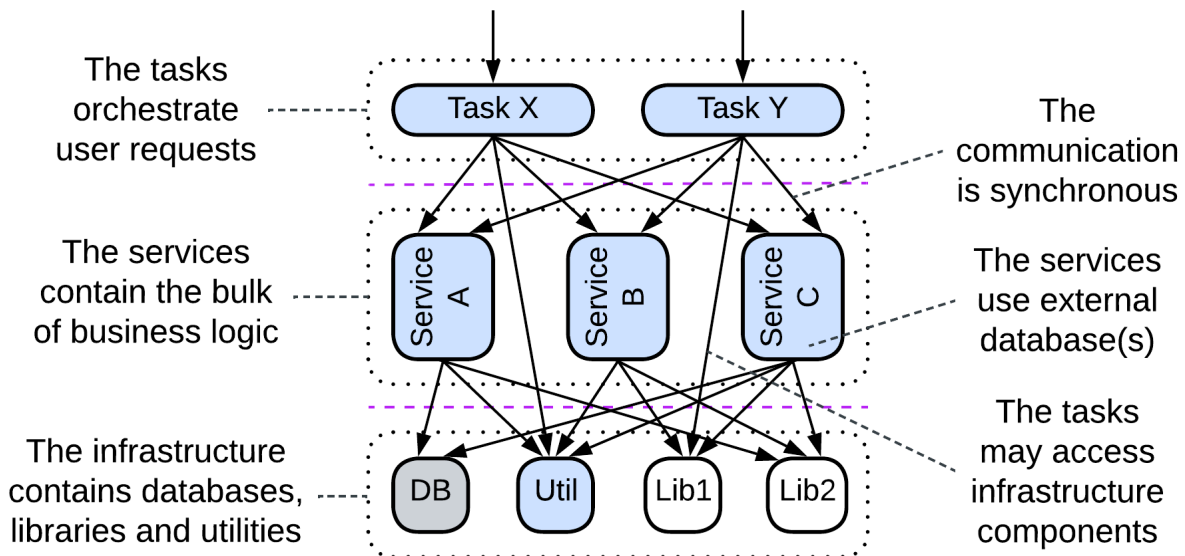
Relations

Service-Oriented Architecture:

- Is a stack of *[layers](#)* each of which is divided into *[services](#)*.
- Is often extended with an *[enterprise service bus](#)* (a kind of *[orchestrating](#) [middleware](#)*) and one or more *[shared databases](#)*.
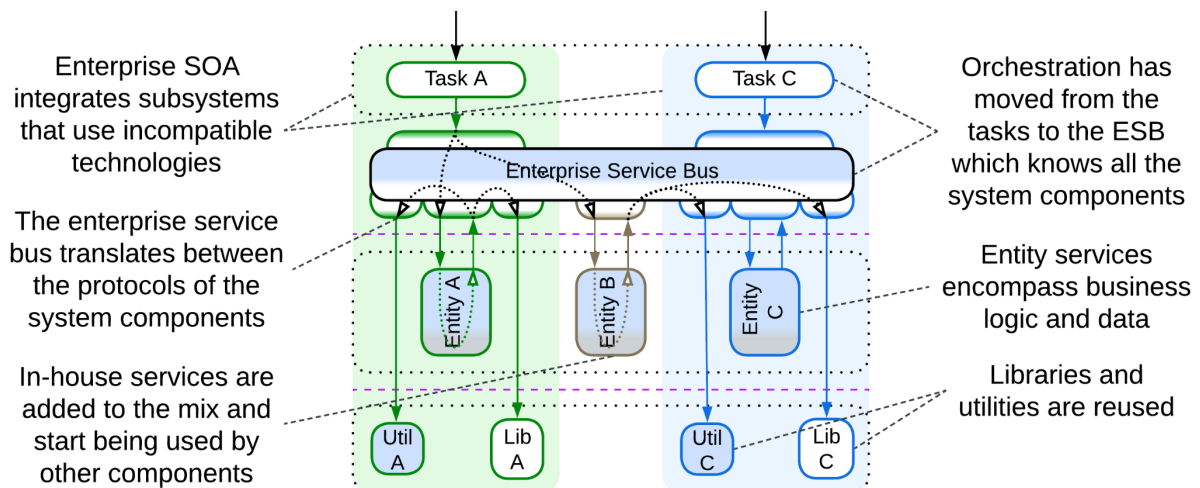
Variants

This architecture was applied mostly during the time when enterprises were expanding by merging smaller companies and conjoining their IT systems. The systems that resulted were still heterogeneous and the development experience unpleasant, inclining the popular opinion towards the then novel notion of *[Microservices](#)*. As everybody has turned from merging existing systems to failing to apply *Microservices* in practice, the chance to find a pure greenfield *SOA* project in the wild is quite low. Many systems which are marketed as *SOA* are strongly modified:

Distributed Monolith

The tasks orchestrate user requests

The services contain the bulk of business logic

The infrastructure contains databases, libraries and utilities

The communication is synchronous

The services use external database(s)

The tasks may access infrastructure components

If a *monolith* gets too complex and resource-hungry, the most simple&stupid way out of the trouble is to deploy each of its component modules to a separate hardware. The resulting modules still communicate synchronously and are subject to domino effect on failure. Such an architecture may be seen as a (hopefully) intermediate structure in transition to more independent and stable event-driven *Services* (or *Cells*).
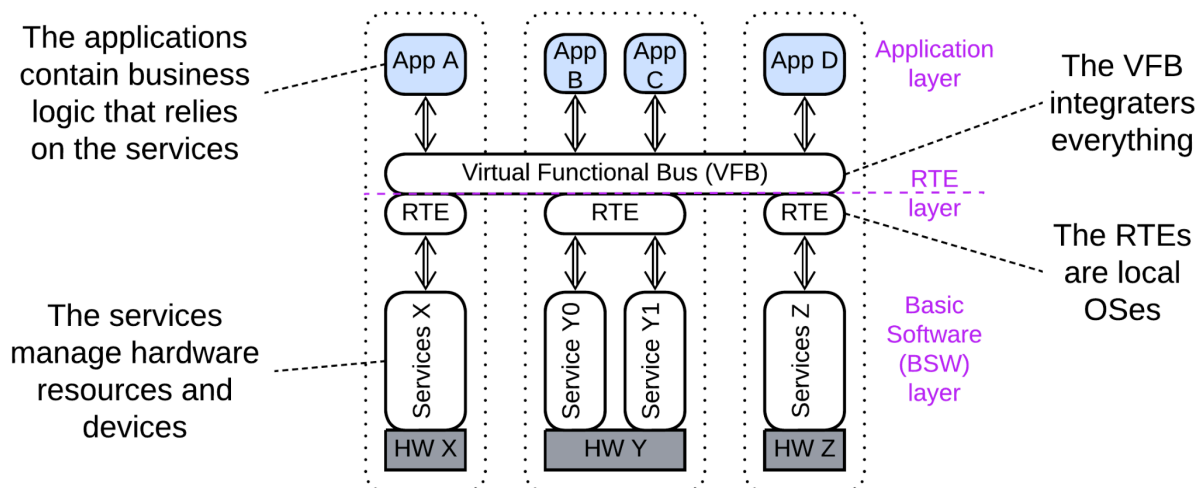
Enterprise SOA

Enterprise SOA integrates subsystems that use incompatible technologies

The enterprise service bus translates between the protocols of the system components

In-house services are added to the mix and start being used by other components

Orchestration has moved from the tasks to the ESB which knows all the system components

Entity services encompass business logic and data

Libraries and utilities are reused

Multiple systems of *Services* each featuring an *API gateway* and a *shared database* are integrated, resulting in new cross-connections. Much of the orchestration logic is removed from the *API gateways* and reimplemented in an *orchestrating middleware* called *Enterprise Service Bus (ESB)*. This option allows for fast and only moderately intrusive integration (as no changes to the *services*, which implement the mass of the business logic, are required), but the single orchestrating component (*ESB*) often becomes the bottleneck for the future development of the system due to its size and complexity. It is likely that if the *orchestration* were encapsulated in the individual *API gateways*, the system would be easier to deal with (making what is now marketed by WSO2 as *Cell-Based Architecture*).

The layers of *SOA* are:

- *Business Process* (*Task*) – the definitions of use cases for a single business department, similar to the *API gateways* layer of [BFF](#).
- *Services* (*Enterprise*, *Entity*) – the implementation of the business logic of a subdomain, to be used by the *tasks*.
- *Components* (*Application* & *Infrastructure*, *Utility*) – external libraries and in-house utilities that are designed for shared use by the *services*.

(misapplied) Automotive SOA

The applications contain business logic that relies on the services

The services manage hardware resources and devices

The VFB integraters everything

The RTEs are local OSes

Automotive architectures are marketed as *SOA*, but the old *AUTOSAR Classic* looks more like [Microkernel](#) (which indeed is similar to a 2-layered *SOA* with an [ESB](#)) while the newer system diagrams resemble [Hierarchy](#). Therefore, they are treated in the corresponding chapters.

Nanoservices

It seems that some proponents of [Nanoservices](#) [take them](#) for a novel version of *SOA* – with the old good promise of reusable components. However, as that promise was failing miserably ever since the ancient days of OOP, it is no surprise that [in practice](#) *nanoservices* are used instead to build [pipelines](#) with little to no reuse.

Evolutions

SOA suffers from excessive reuse and fragmentation. To fix that, first and foremost, each service of the *componentes* (*utility*) layer should be duplicated:

- Into every *service* that uses it, giving the developers who write the business logic full control of all the code that they use. Now they have several projects to support on their own (instead of asking other teams to make changes to their components).
- Or into *sidecars* [[DDS](#)] if you employ a [service mesh](#), resulting in much fewer network hops (thus lower latency) in request processing, but retaining the inter-team dependencies.

That removes a large and the most obvious part of the fragmentation, making the [ESB](#) (if you use one) orchestrate only the *entity* layer.

Each service may get replicas of components it uses

All the components may be replicated to sidecars
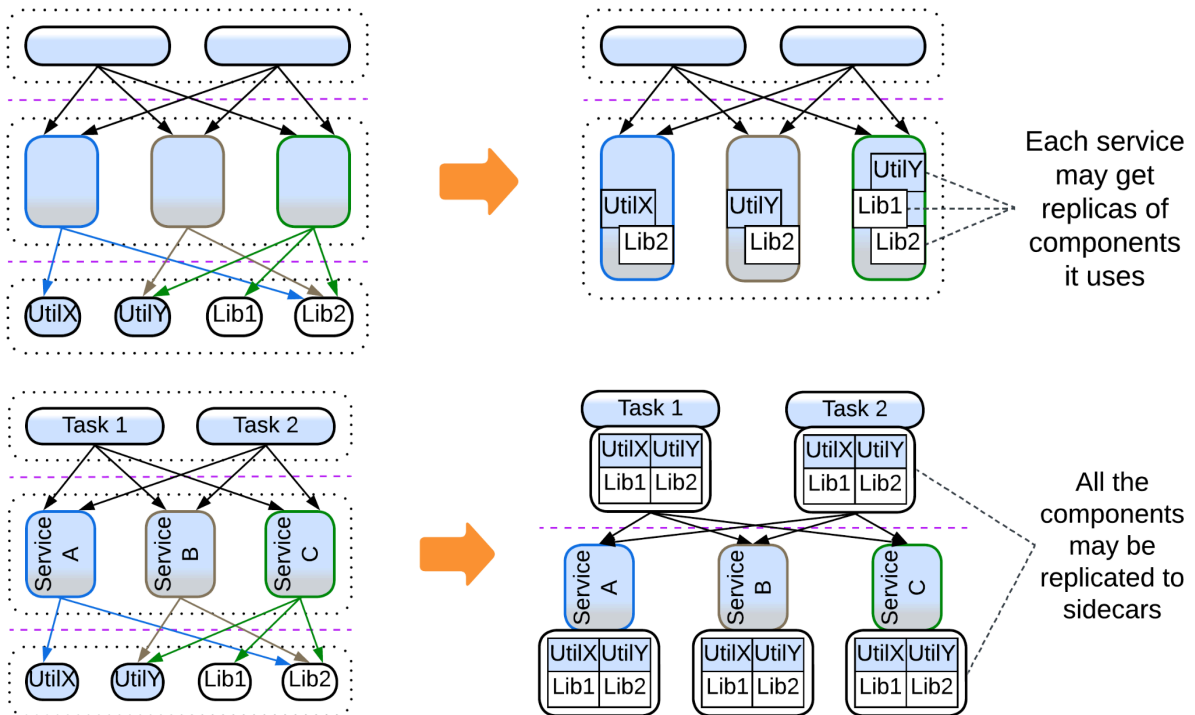
After that you may deal with the remaining orchestration. The idea is to move the orchestration logic from the *ESB* to an explicit *layer* of *orchestrators*:

● Either a monolithic *orchestrator* over all the services.
● Or *Backends for Frontends* with an orchestrator per client type (department of an enterprise) if each client uses most of the services.
● Or go for *cells* with an orchestrator per subdomain if your clients are subdomain-bound.
● Or a combination of the above.

Services with an Orchestrator

Backends for Frontends

Cell-Based Architecture
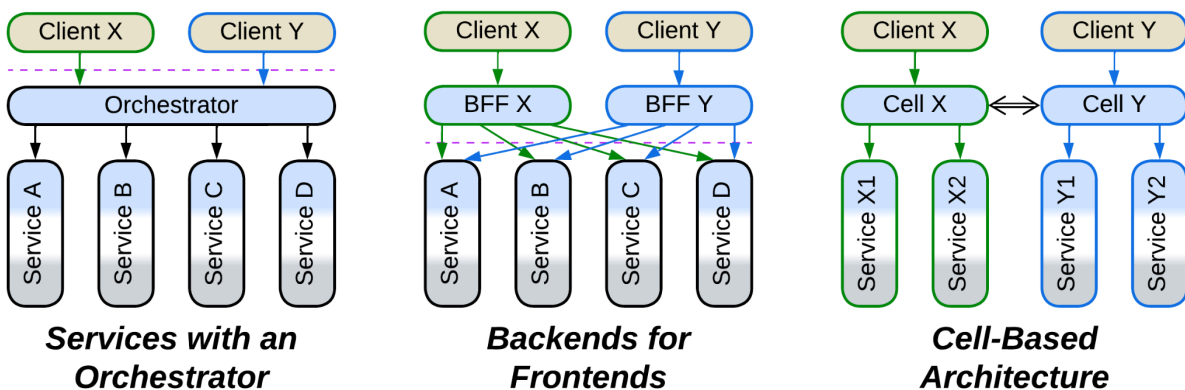
Still another step is unbundling *middleware*, which supports multiple protocols via adapters:

● If you use a *service mesh*, the adapter may be put to *sidecars* [DDS].
● Otherwise there is an option of a *hierarchical middleware* (*bus of buses*) if closely related components share protocols.
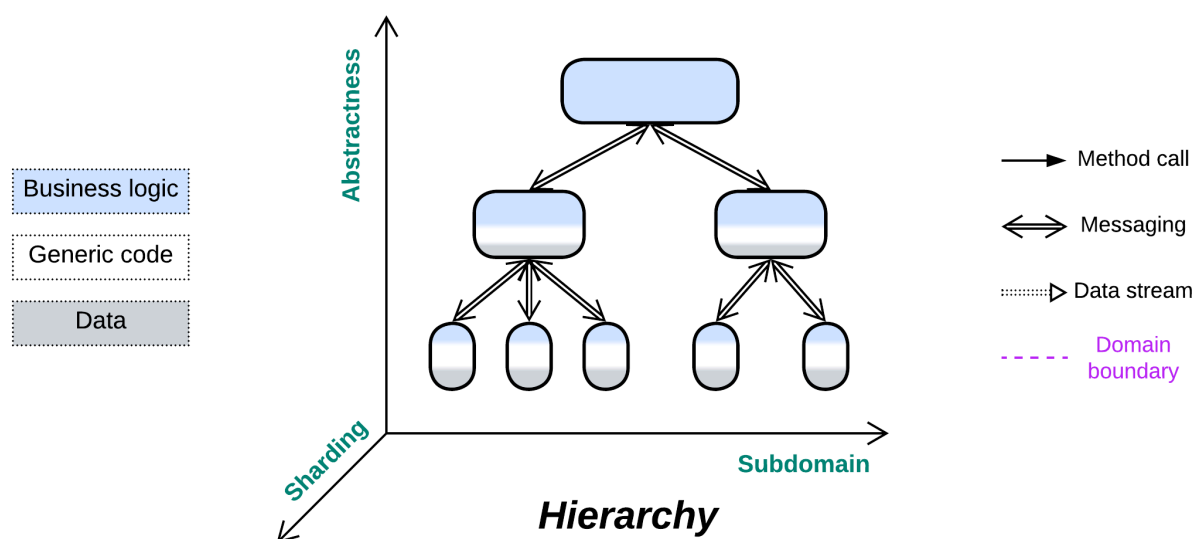
Still, the evolution of *middleware* may not bring any real benefit except for removing the *ESB*, which may not be that bad after all, when it is not misused.

In any case the evolutions will likely be very expensive, thus it makes sense to conduct some of them gradually via a kind of [strangler fig approach](). Or let the architecture live and die as it is.

Summary

Service-Oriented Architecture divides each of: *integration*, *domain* and *utility* layers into shared services. The extensive fragmentation and reuse degrade performance and speed of development. Nevertheless, huge projects are known to survive with this architecture.

Hierarchy

Business logic

Generic code

Data

Method call

Messaging

Data stream

Domain boundary

Abstractness

Sharding

Subdomain

Hierarchy

Command and conquer. Build a tree of responsibilities.

Variants:

By structure:
- Polymorphic children,
- Functionally distinct children.

By direction:
- Top-down Hierarchy / Orchestrator of Orchestrators,
- Bottom-up Hierarchy / Bus of Buses / Network of Networks,
- In-depth Hierarchy / Cell-Based (Microservice) Architecture (WSO2 version) / Segmented Microservice Architecture / Services of Services.

Structure: A tree of modules.

Type: Main or extension.

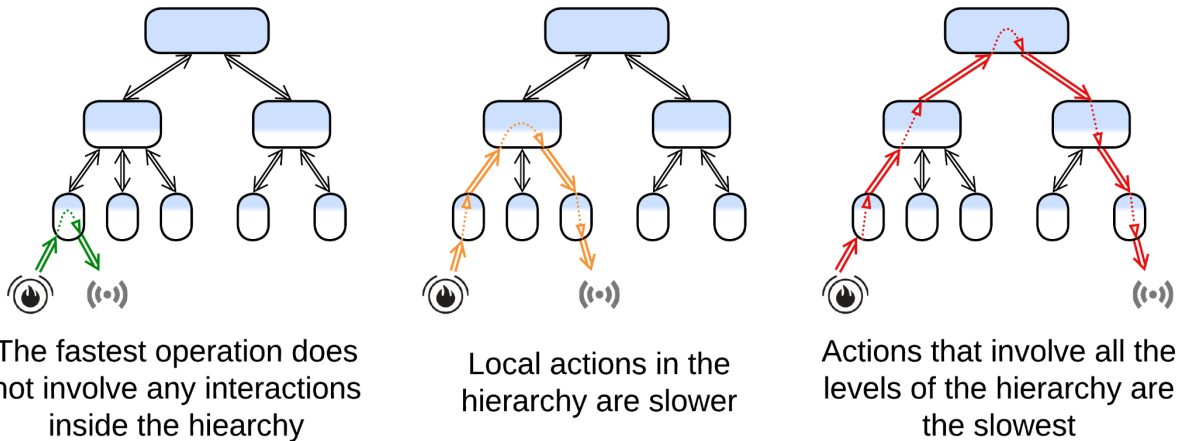| Benefits	Drawbacks
Very good in decoupling logic	Global use cases may be hard to debug
Supports multiple development teams and technologies	Poor latency for global use cases
Components may vary in qualities	Operational complexity
Low-level components are easy to replace	Slow start of the project

References: None good I know of. *Presentation-Abstraction-Control* [POSA1, POSA4] is long dead and buried.

Though not applicable to every domain, hierarchical decomposition is arguably the best way to distribute responsibilities between components. It limits the connections (thus the number of interfaces and contracts to keep in mind) of each component to its parent and few children, allowing for building complex (and even complicated) systems in a simple way. The hierarchical structure is very flexible as it features multiple layers of indirection (and often polymorphism), thus making the addition, replacement or stubbing of leaf components trivial. It is also very fault-tolerant as individual subtrees operate independently.

This architecture is not ubiquitous because few domains are truly hierarchical. Its high fragmentation results in increased latency and poor debugging experience. Moreover, component interfaces should be designed beforehand and are hard to change.

Performance

No kind of distributed *hierarchy* is latency-friendly as many use cases involve several network hops. The fewer layers of the hierarchy are involved in a task, the better its performance.


| The fastest operation does not involve any interactions inside the hiearchy | Local actions in the hierarchy are slower | Actions that involve all the levels of the hierarchy are the slowest |

Maintaining high throughput usually requires deploying multiple instances of the root component, which is not possible if it is stateful (in *control systems*) and the state cannot be split to *shards*. The following tricks may help unloading the root:

- *Aggregation* (first met in *Layers*): a node of a hierarchy collects reports from its children, aggregates them into a single package, and sends the aggregated data to its parent. That greatly reduces traffic to the root in large IIoT networks.
- *Delegation* (resembles strategy injection and batching for *Layers*): a node should try to handle all the low-level details of communication with its children without consulting its parent node. For a control system that means that its mid-level nodes should implement control loops for the majority of incoming events. For a processing system that means that the mid-level nodes should expose coarse-grained interfaces to their parent(s) while translating each API method call into multiple calls to their child nodes.
- *Direct communication channels* (previously described for *Orchestrator*): if low-level nodes need to exchange data, their communication should not always go through the higher-level nodes. Instead, they may negotiate a direct link (open a socket) that bypasses the root of the hierarchy.
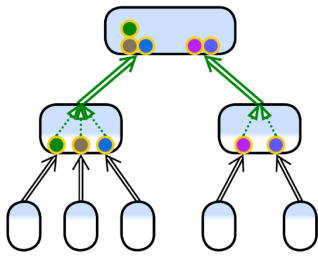
Mid-level nodes should aggregate the events they receive from child nodes
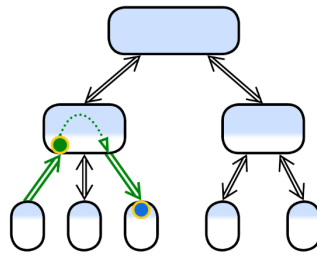
Mid-level nodes should act autonomously whenever possible
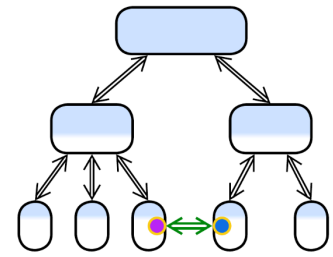
Low-level nodes can intercommunicate directly

Dependencies

Usually each parent node provides one (for polymorphic children) or more (otherwise) SPIs for child nodes to communicate with them through. The interfaces reside on the parent side because low-level nodes tend to be less stable (new types of them are often added and old ones replaced).

In hierarchy parent nodes tend to provide interfaces for their children
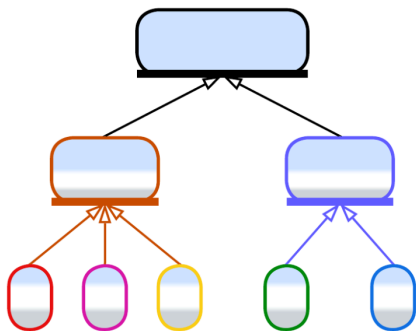
Hierarchy with polymorphism
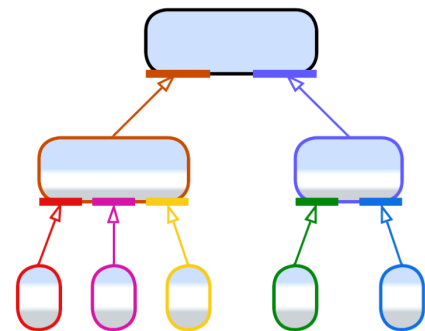
Hierarchy without polymorphism

Applicability

Hierarchy fits:

- *Large and huge projects.* The natural division by both level of abstraction and subdomain allows for using smaller modules, ideally behind intuitive interfaces. The APIs to learn for each team are limited to the few which their component interacts with directly.
- *Systems of hardware devices.* Real-world IIoT systems may use a hierarchy of controllers to benefit from autonomous decision-making and data aggregation.
- *Customization.* The tree structure provides opportunities for easy customization. A medium-sized hierarchical system may integrate hundreds of leaf types.
- *Survivability.* A distributed hierarchy retains a limited functionality even after failure of multiple nodes.

Hierarchy fails with:

- *Cohesive domains.* Horizontal interactions (those between nodes that belong to the same layer) bloat interfaces as they have to go through the parent nodes.
- *Quick start.* Finding (and proving) a good hierarchical domain model may be hard if possible at all. Debugging the initial implementation will not be easy.

- *Low latency*. System-wide scenarios involve many cross-component interactions which are slow in distributed systems.

Relations

Orchestrator of orchestrators

Middleware of middlewares

Services of services
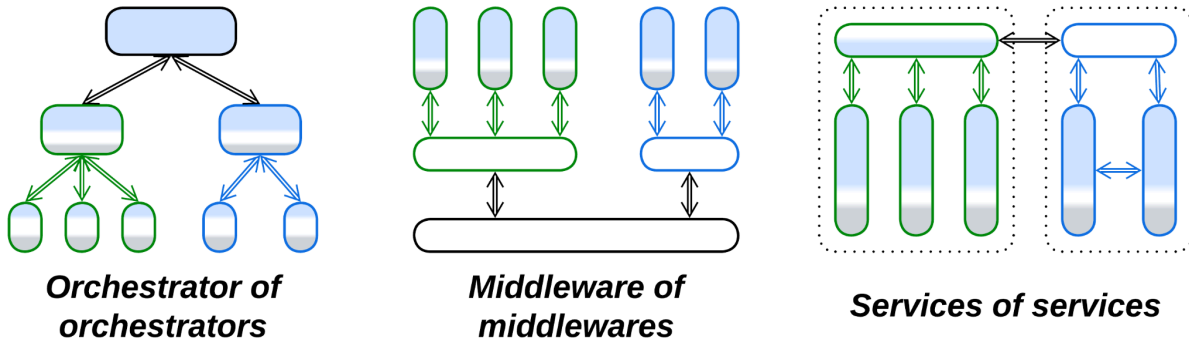
Hierarchy:
- Can be applied to *Orchestrator*, *Middleware* or *Services*.

Variants by structure (may vary per node)

Hierarchy comes in various shapes as it is more of a design approach than a ready-to-use pattern:

Polymorphic children

All the managed child nodes expose the same interface and contract. This tends to simplify the implementation of the parent node and resembles inheritance of OOD.

Example: a fire alarm system may treat all the fire sensors as identical devices, though the real hardware comes from many manufacturers.
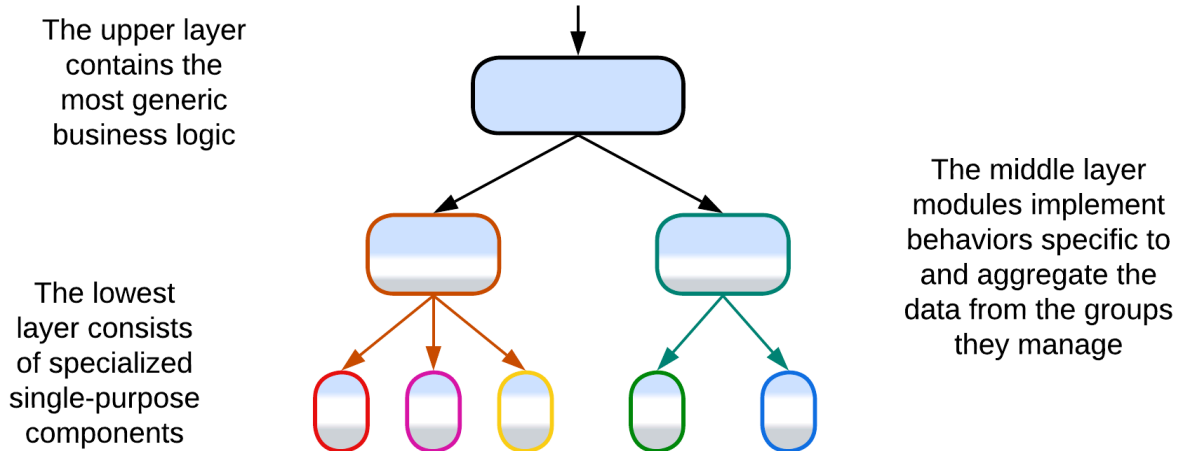
Functionally distinct children

The managing node is aware of several kinds of children that vary in their APIs and contracts, just like with composition in OOD.

Example: an intrusion alarm logic may need to discern between cat-affected IR sensors and cat-proof glass break detectors.
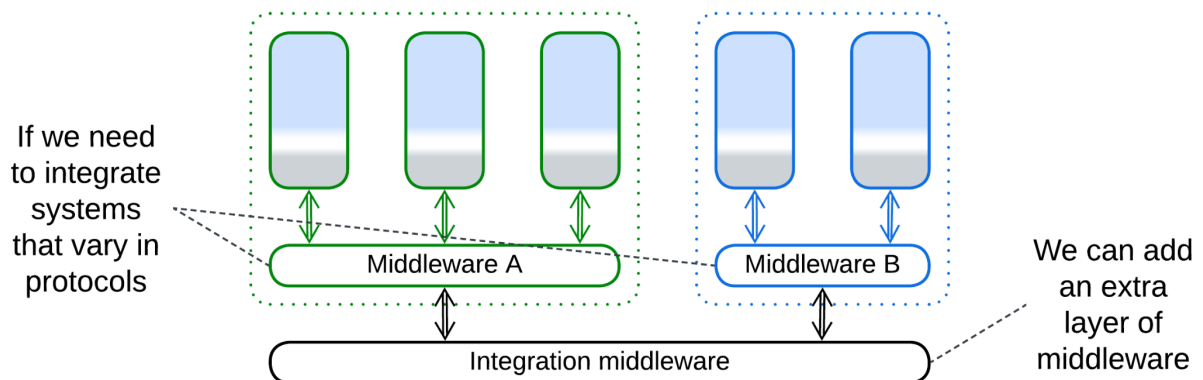
Variants by direction

Top-down Hierarchy, Orchestrator of Orchestrators

The upper layer contains the most generic business logic

The middle layer modules implement behaviors specific to and aggregate the data from the groups they manage

The lowest layer consists of specialized single-purpose components

In the most common case *Hierarchy* is applied to business logic to build a layered system that grows from a single generic high-level root to a swarm of specialized low-level pieces. The most obvious applications are protocol parsers, decision trees, IIoT (e.g. the fire alarm system of a building) and modern automotive networks. A marketplace that allows for customized search and marketing algorithms within each category of goods available may also be powered by a hierarchy of category-specific services.
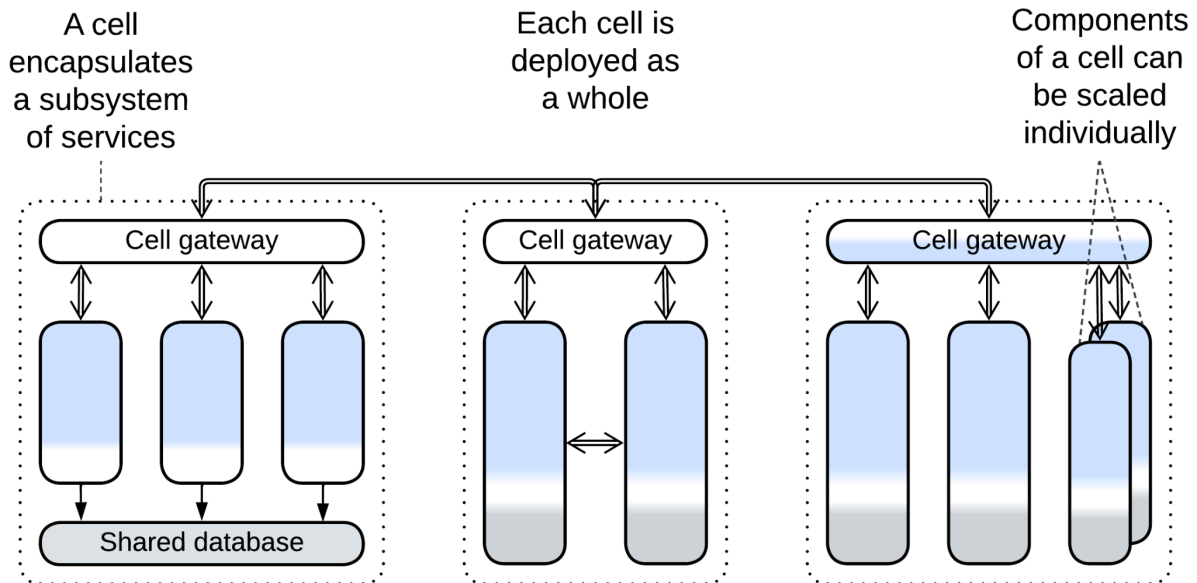
Bottom-up Hierarchy, Bus of Buses, Network of Networks

If we need to integrate systems that vary in protocols

Middleware A

Middleware B

We can add an extra layer of middleware

Integration middleware

Other cases require building a common base for intercommunication of several networks that vary in their protocols (and maybe their hardware). The root of such a *hierarchy* is a *middleware*, generic and powerful enough to cover the needs of all the specialized networks which it interconnects.

Example: Automotive networks, integration of corporate networks, the Internet.

In-depth Hierarchy, Cell-Based (Microservice) Architecture (WSO2 version), Segmented Microservice Architecture, Services of Services

A cell encapsulates a subsystem of services

Each cell is deployed as a whole

Components of a cell can be scaled individually

When several *services* in a system grow large, in some cases it is possible to divide each of them into subservices. Each group of the resulting subservices (known as *cell*) is hidden behind its own *cell gateway* and may even use its own *middleware*. The subservices in a *cell* may *share a database* and may be deployed as a single unit. This keeps the system's integration complexity (the length of APIs and the number of deployable units) reasonable while still scaling development among many teams, each owning a service. If each instance of the *cell* owns a *shard* of its database, the system becomes more stable as there is no single point of failure (except for the *load balancer* called *cell router*). Another benefit is that *cells* can be deployed to regional data centers to improve locality for users of the system. However, that will likely cause data synchronization traffic between the data centers.

Cell-Based Architecture (*Segmented Microservice Architecture*) may be seen as a combination of *Orchestrator of Orchestrators* and *Bus of Buses* where the subservices are leaves of both *hierarchies* while the *API gateways* of the *cells* are their internal nodes.

Example: Systems that contain many *microservices*.

Evolutions

- The upper component of a *top-down hierarchy* can be split into *Backends for Frontends*.

The upper layer of a hierarchy may be divided per client type
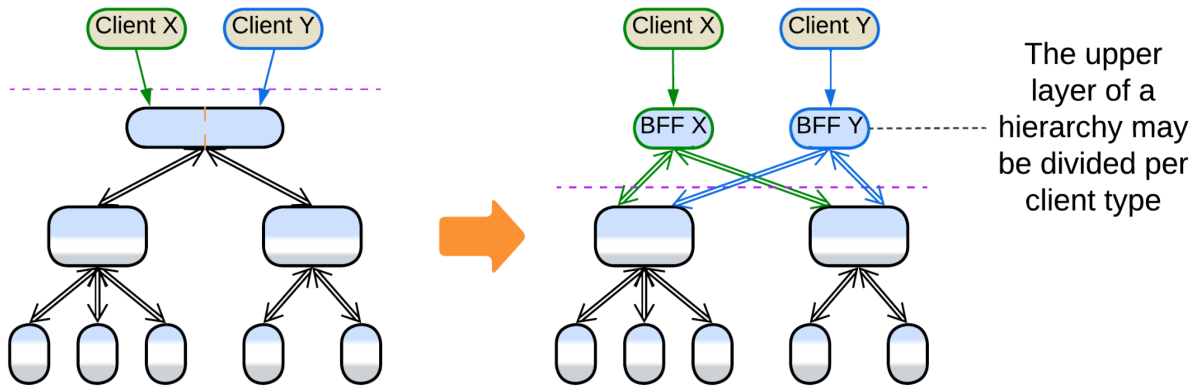
Summary

Hierarchy fits a project of any size as it evenly distributes complexity among the many system's components. However, it is not without drawbacks in performance, debuggability and operational complexity. Moreover, very few domains allow for application of this architecture.

Part 5. Implementation Metapatterns

Several patterns emerge in the internal structure of components.

Plugins

Plugins allow for customization of the system's functionality

Client

Addons modify input and output

Plugins implement aspects of the system's behavior
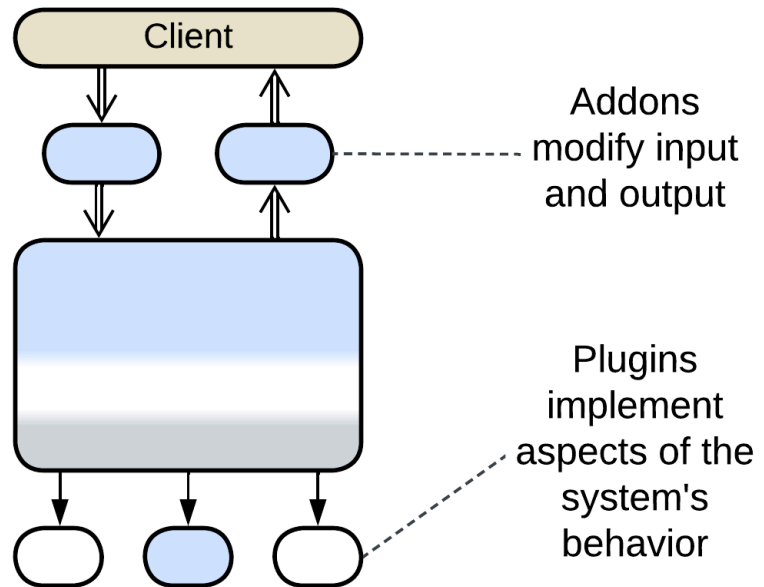
The Plugins pattern is about separating the main logic from customizable details of the system's behavior. That allows for the same code to be used for multiple flavors or customers.
Includes: Plug-In Architecture, Add-ons, Strategy, Hooks.

Hexagonal Architecture

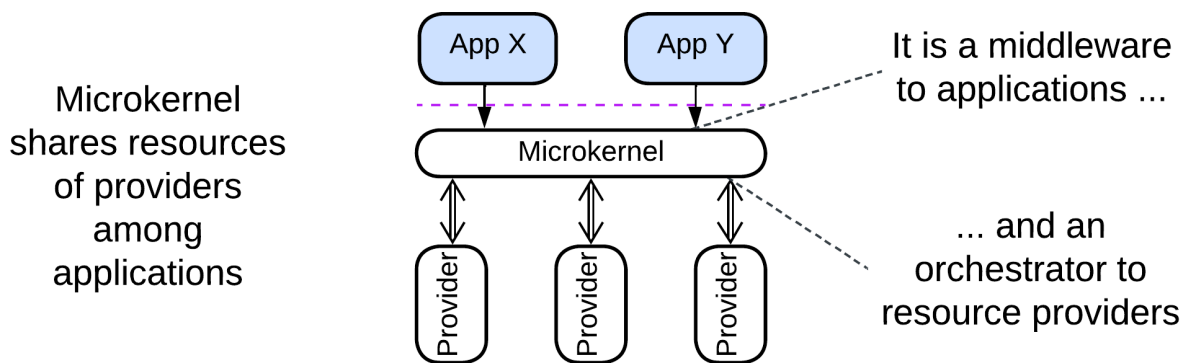Hexagonal Architecture assigns an adapter to each external component

Input   Output

The goal is to make the business logic independent

DB   LibX   LibY
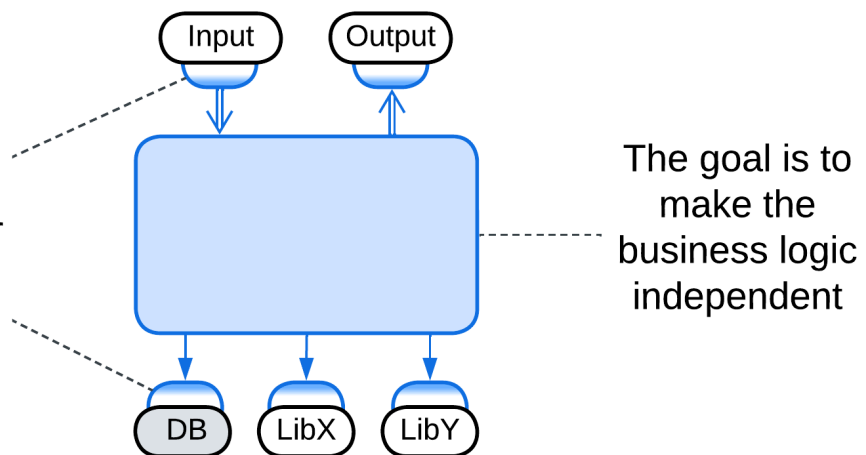
Hexagonal Architecture is a specialization of Plugins where every external dependency is isolated behind an adapter and thus is easy to update or replace.
Includes: Ports and Adapters, Onion Architecture, Clean Architecture; Model-View-Controller (MVC).
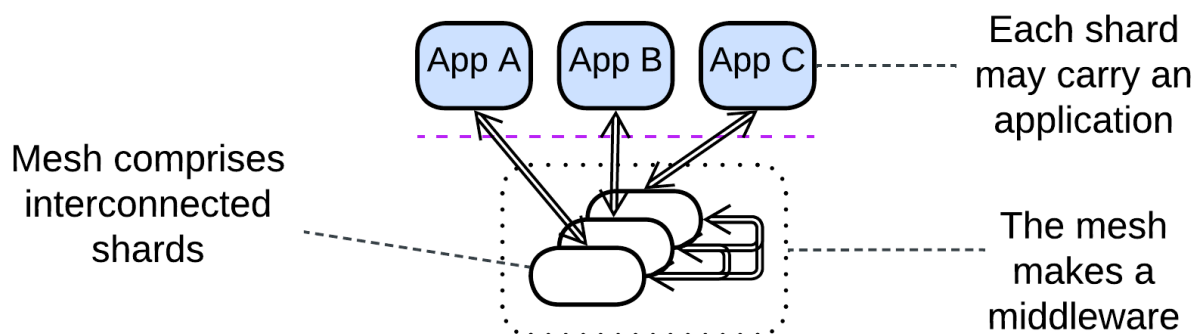
Microkernel

Microkernel shares resources of providers among applications

App X   App Y

Microkernel

Provider   Provider   Provider

It is a middleware to applications ...

... and an orchestrator to resource providers

This is another derivation of Plugins, with a rudimentary core component which mediates between resource consumers (applications) and resource providers. The microkernel is a middleware to the applications and an orchestrator to the providers.

Includes: operating system, software framework, virtualizer, distributed runtime, interpreter, configuration file, saga engine, AUTOSAR Classic Platform.

Mesh

Mesh comprises interconnected shards

App A   App B   App C

Each shard may carry an application

The mesh makes a middleware

A mesh consists of intercommunicating shards, each of which may host an application. The shards make a fault-tolerant distributed middleware.

Includes: grid; peer-to-peer networks, Leaf-Spine Architecture, actors, Service Mesh, Space-Based Architecture.

Plugins
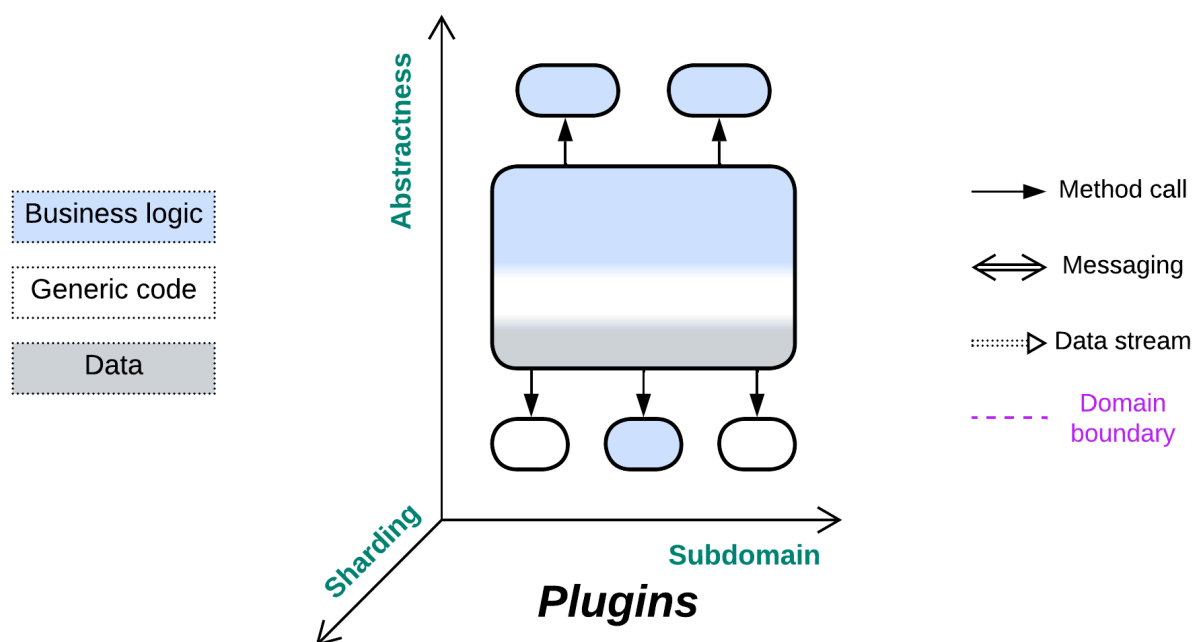
Plugins

Overspecialize, and you breed in weakness. Customize the system through attachable modules.

Known as: Plug-In Architecture [FSA], (misapplied) Microkernel (Architecture) [POSA1, POSA4, SAP, FSA], Plugin [PEAA], Add-ons, Strategy [GoF, POSA4], Reflection [POSA1, POSA4], Aspects, Hooks.

Variants: *Hexagonal Architecture* and *Microkernel* got dedicated chapters. Plugins differ in many ways.

Structure: A monolith is extended with one or more modules which customize its behavior.

Type: Implementation, extension.

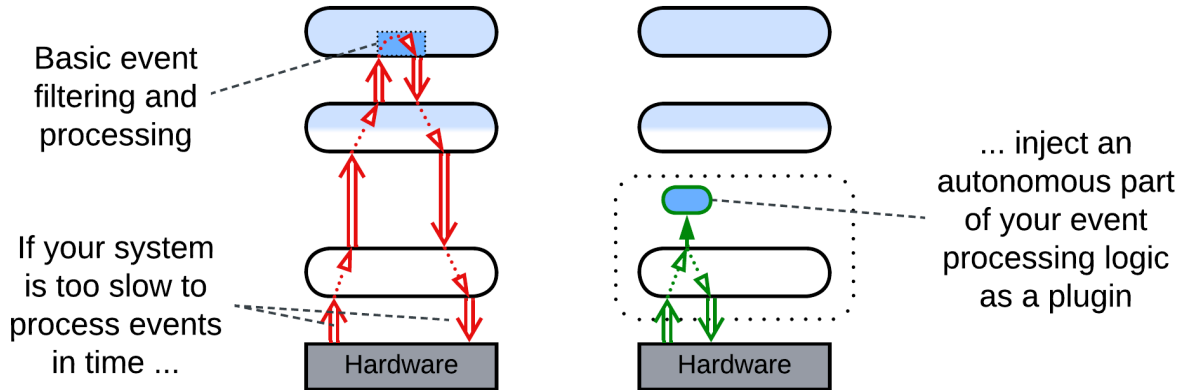| Benefits	Drawbacks
Some aspects are easy to customize	Testability is poor (too many combinations)
A customized system is relatively light-weight	Performance is not optimal
Platform-specific optimizations are possible	API design is hard
The custom pieces may be written in a different programming language or DSL	

References: [SAP] and [FSA] mistakenly call this pattern *Microkernel* and dedicate chapters to it.

Most systems require some extent of customizability: from the basic codec selection by a video player to screens full of tools and wizards unlocked once you upgrade your subscription plan. This is achieved by keeping the *core* functionality separate from its extensions, which are developed by either your team or external enthusiasts to modify the behavior of the system. The cost of flexibility is paid in the complexity of design – the need to predict which aspects must be customizable and what APIs are good for known and unknown uses by the extensions. Heavy communication between the *core* and *plugins* negatively impacts performance.
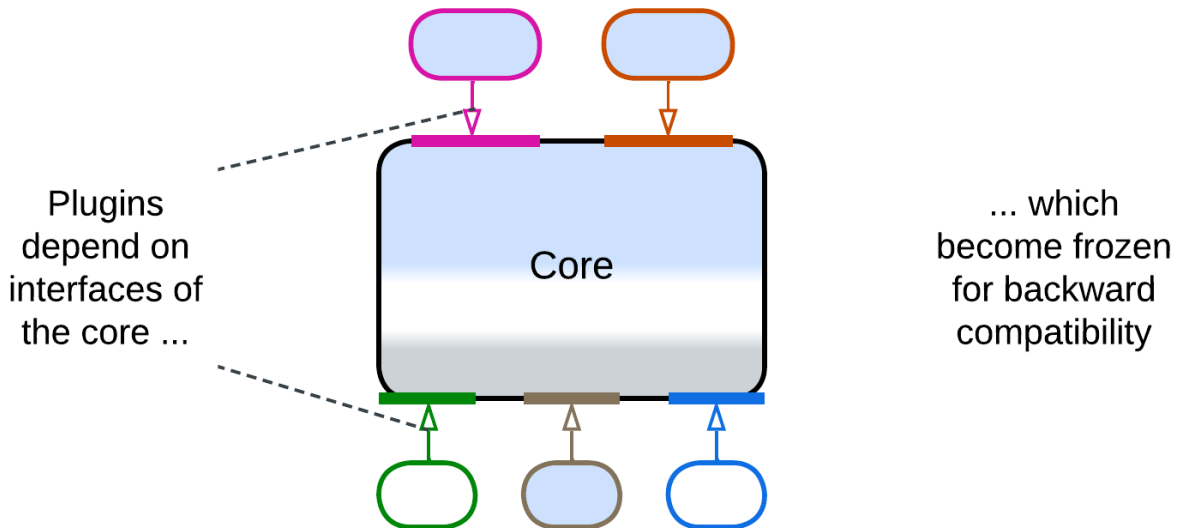
Performance

Using *plugins* usually degrades performance. The effect may be negligible for in-process plugins (such as strategies or codecs) but it becomes noticeable if inter-process communication or networking is involved.

The only case for a plugin to improve performance of a system that I can think up is when a part of the client's business logic moves to a plugin in a lower layer of a system. That is similar to the strategy injection optimization for *Layers*. A real-world example is the use of stored procedures in databases and it is also likely that such an approach is good for HFT.

Basic event filtering and processing

If your system is too slow to process events in time ...

... inject an autonomous part of your event processing logic as a plugin

Hardware

Dependencies

Each *plugin* depends on the *core*'s *API* (for *add-ons*) or *SPI* (for *plugins*) for the functionality it extends. That makes the APIs and SPIs nearly impossible to change, only to extend, as there may be many plugins in the field, some of them out of active development, that rely on any given method of the already published interfaces.

Plugins depend on interfaces of the core ...

Core

... which become frozen for backward compatibility

Applicability

Plugins are good for:
- *Product lines.* The shared *core* functionality and some *plugins* are reused by many flavors of the product. Per-client customizations are largely built of existing *plugins*.

- *Frameworks.* A framework is the functional *core* to be customized by its users. When shipped with *plugins* it becomes ready-to-use.
- *Platform-specific customizations. Plugins* allow both for native look and feel (e.g. desktop vs mobile vs console) and for the use of platform-specific hardware.
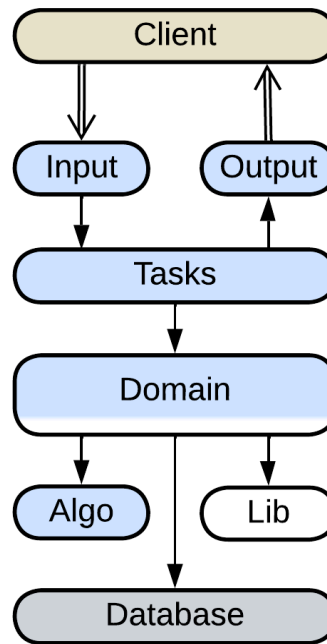
Plugins do not perform well in:
- *Highly optimized applications.* Any generic code tends to be inefficient. A generic API that serves a family of *plugins* is unlikely to be optimized for the use by any of them.

Relations
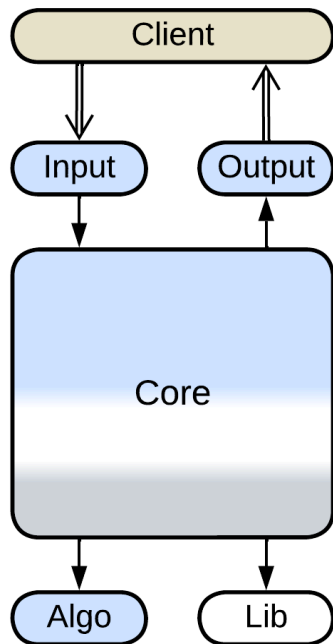
Monolith with plugins

Layers with plugins

Plugins:
- Implement *Monolith* or sometimes *Layers*.
- Extend *Monolith* or *Layers* with one or two layers of services.

Variants

Plugins are highly variable and omnipresent if we take *Strategy* [GoF] for a kind of plugin:

By the direction of control

The terminology is not settled, but according to what I found over the Web:
- True *Plugins* are registered with and called by the system's *core*, they may call back into the *core* or return results – they are parts of the system.
- *Addons* are built on top of the system's API and call into the system from outside – they are more like external *adapters* for the system.

By abstractness

A system may use *plugins* that are more abstract than the *core*, less abstract or both:
- High-level *plugins* tend to be related to user experience, statistics or metadata. They use the core in their own business logic. *Addons* belong here.
- Low-level *plugins* encapsulate algorithms that are used by the *core*'s business logic.
- Some customizations require multiple *plugins*: a high-level user-facing *plugin* relies on algorithms or pieces of business logic which are implemented by several complementary low-level *plugins*.

By the direction of communication

A *plugin* may:
- Provide input to the *core* (as UI screens and CLI connections).
- Receive output from the *core* (e.g. collection of statistics).
- Participate in both input and output (like health check instrumentation).
- Take the role of *controller* – the *plugin* processes events from the *core* and decides on the *core*'s further behavior.
- Be a data processor – the *plugin* implements a part of a data processing algorithm which is run by the *core*.

By linkage

A *plugins* may be *built-in* or dynamically selected:
- *Flavors* in product lines tend to have a single set of *plugins* (configuration) built into the application.
- Other systems choose and initiate their *plugins* at the start according to their configuration file.
- Still others may attach and detach a *plugin* dynamically at runtime.

By granularity

Plugins come in different sizes as:
- Small functions or classes built into the core. These are not true architectural *Plugins* but rather *Strategies* [GoF] / *Plugins* [PEAA].
- Aspects that pervade the system and are accessed from much of the code, like logging or memory management. Also not true *Plugins*, but rather *Aspects* / *Reflection* [POSA1, POSA4].
- Modules that are plugged in as separate system components. Such a kind of *plugins* matches the topic of this book. *Hexagonal Architecture* and *Microkernel* deal with system-scale components.

By the number of instances

A *plugin* may be:
- Mandatory (1 instance), like a piece of algorithm used by the core for a calculation.
- Optional (0 or 1 instance), like a smart coloring scheme for a text editor.
- Subscriptional (0 or more instances), like the log output which may go to a console, the system log, a log file and network connections all at once.

By execution mode

Plugins may be:
- Linked as a binary code called from within the *core*.
- Written in a domain-specific language which is interpreted by the *core*.
- Communicated with over a network.

Summary

Plugins allow for customization of a component's behavior at the cost of increased complexity, poor testability and somewhat reduced performance.

Hexagonal Architecture
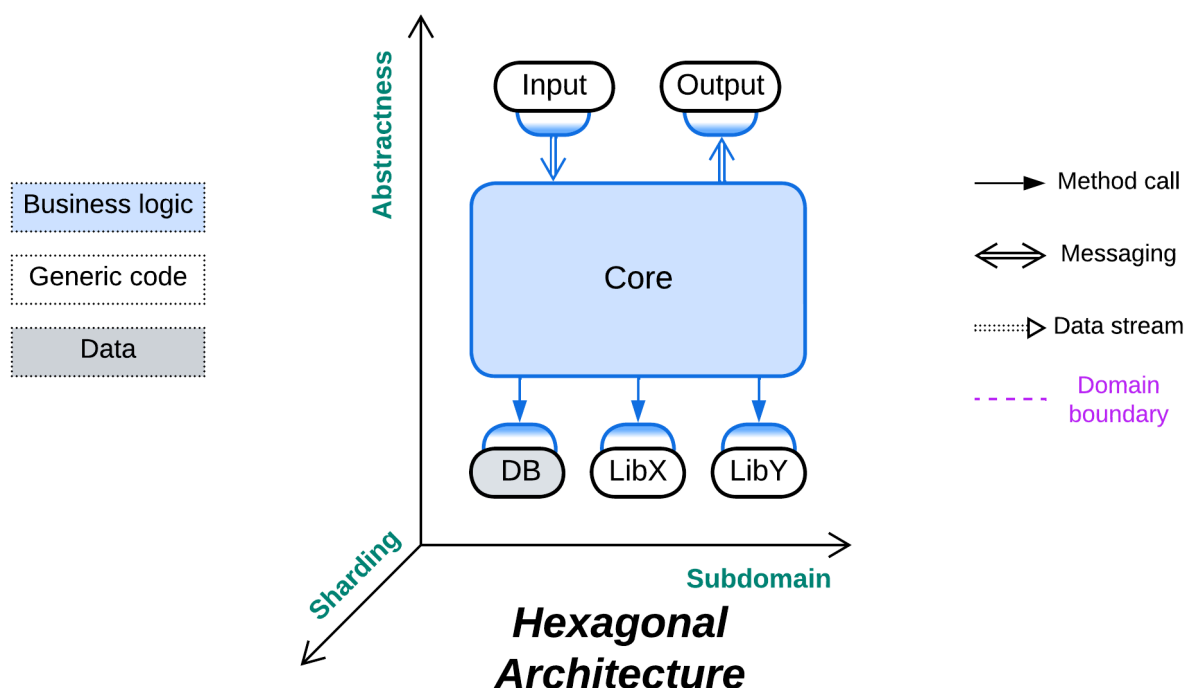
Trust no one. Protect your code from external dependencies.

<u>Known as:</u> Hexagonal Architecture, Ports and Adapters.

<u>Variants:</u>

By placement of adapters:

● Adapters on the external component side.
● Adapters on the core side.

Examples:

● (*Pipelined*) Model-View-Controller (MVC) [POSA1, POSA4],
● Hexagonal Architecture / Ports and Adapters,
● DDD-style Hexagonal Architecture [LDDD] / Onion Architecture / Clean Architecture.

<u>Structure:</u> A monolithic business logic is extended with a set of (adapter, service) pairs that encapsulate external dependencies.

<u>Type:</u> Implementation.

| Benefits	Drawbacks
Decouples external dependencies from the business logic	Suboptimal performance
Facilitates the use of stubs for testing and development	The vendor-independent interfaces must be designed before the start of development
Allows for qualities to vary between the external components and the business logic	
The programmers of business logic don't need to learn any external technologies	

<u>References:</u> The original article; Herberto Graça's summary, a brief summary of the layered variant in [LDDD].

Hexagonal Architecture is a variation of *Plugins* that aims for the total self-sufficiency of the business logic written in-house. Any 3rd party tools, whether libraries, services or databases, are hidden behind *adapters* [GoF] that translate the external module's interface into a *Service Provider Interface* (*SPI*) defined by the *core* module and called "*port*". The business logic depends only on the *ports* that its developers defined – a perfect use of dependency inversion – and manipulates the interfaces that were designed in the most convenient way. Free benefits of this architecture include the *core*'s cross-platform nature, easy development and testing with stubs, support for event replay and protection from vendor lock-in. The architecture allows for the external libraries to be switched to another implementation at a late stage of the project. The flexibility is paid for with a somewhat longer system design stage and lost optimization opportunities. There is also a high risk to design a leaky abstraction – an *SPI* which looks generic but its contract matches that of the vendor used at the start of the project, making it much harder than expected to change the vendor.
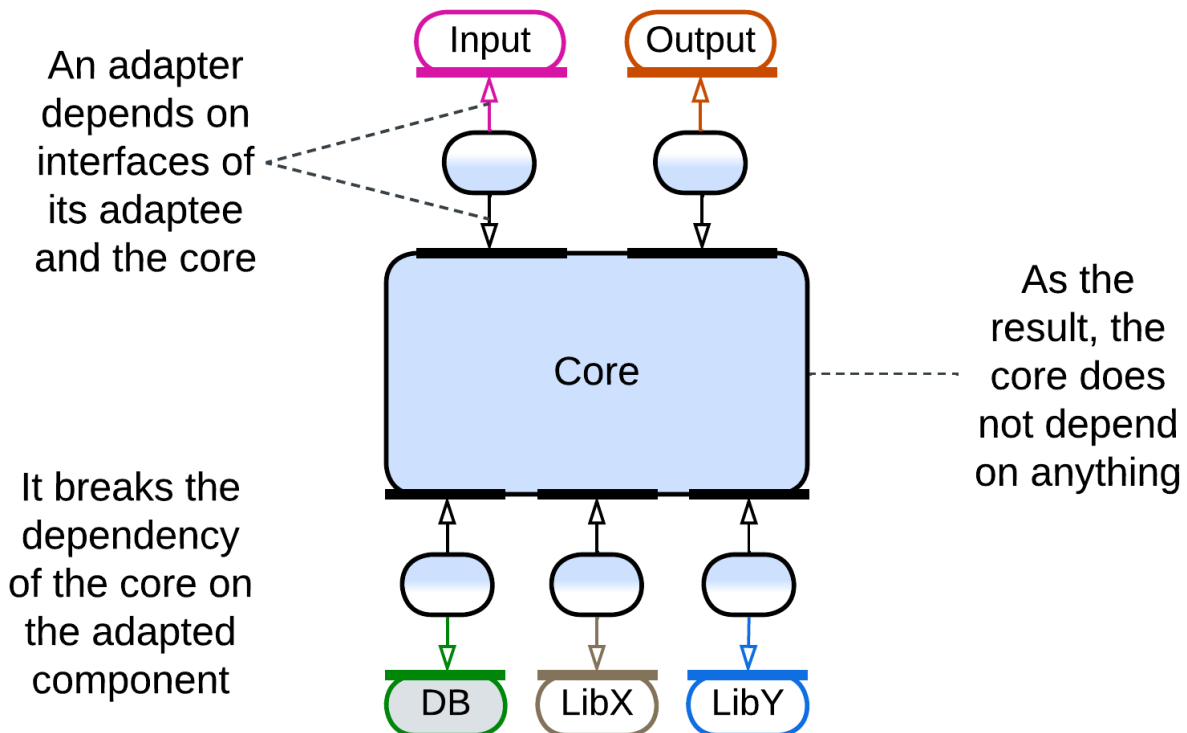
Performance

Hexagonal Architecture is a strange beast performance-wise. The generic interfaces (*ports*) between the core and adapters stand in the way of whole-system optimization and may add a context switch. But, at the same time, each adapter concentrates all the vendor-specific code for each external dependency, which makes it a perfect single place for aggressive optimization by an expert or consultant who has mastered the 3rd party software but does not have time to learn the details of your business logic. Thus, some opportunities for optimization are lost while others emerge.

In rare cases the system may benefit from direct communication between the adapters. However, that requires several adapters to have compatible types, in which case your *Hexagonal Architecture* may in fact be a kind of shallow *Hierarchy*. Examples include a service that uses several databases that must be kept in sync or a telephony gateway that interconnects various kinds of voice devices.
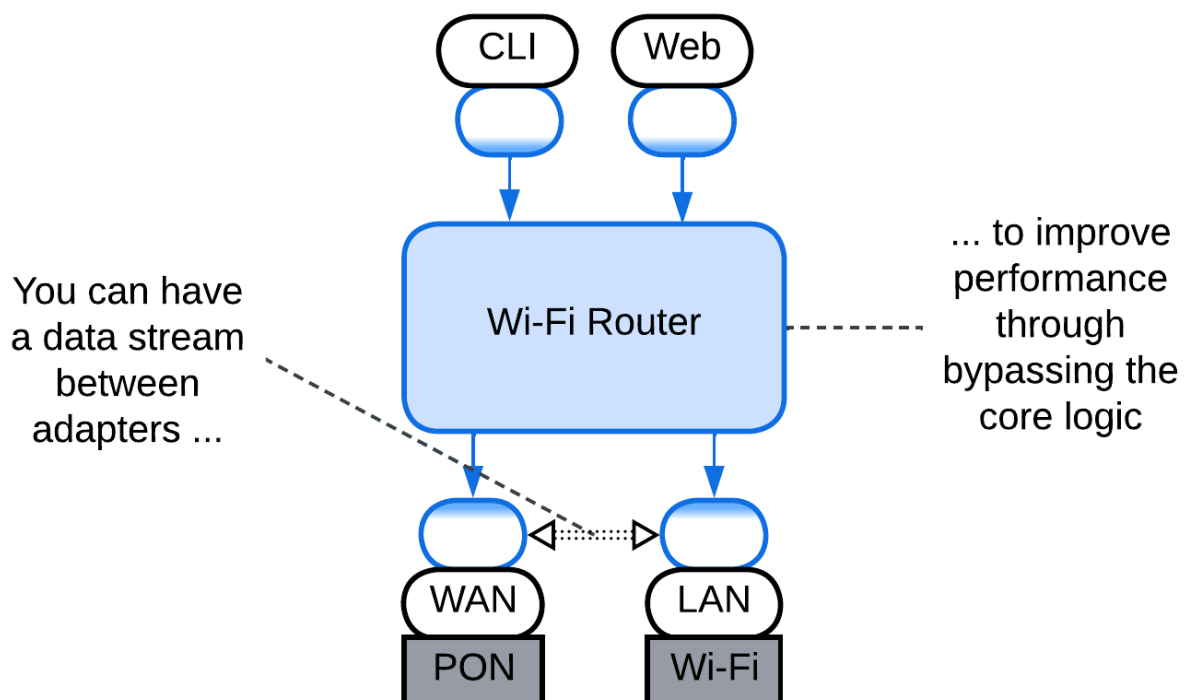
Dependencies

Each *adapter* breaks the dependency between the *core* that contains the business logic and the adapted component. That makes all the system's components mutually independent – and easily interchangeable and evolvable – except for the adapters, which are small enough to be rewritten as need arises.

An adapter depends on interfaces of its adaptee and the core

It breaks the dependency of the core on the adapted component

As the result, the core does not depend on anything

Input   Output

Core

DB   LibX   LibY

Applicability

Hexagonal Architecture <u>benefits</u>:
- *Medium-sized or larger modules.* The programmers don't need to learn details of external technologies and may concentrate on the business logic instead. The size of the *core* code becomes smaller as all the details about managing external components are moved into the *adapters*.
- *Cross-platform development.* The *core* is naturally cross-platform as it does not depend on any (platform-specific) libraries.
- *Long-lived projects.* Technologies come and go, your project remains. Be always ready to change the technologies.
- *Unfamiliar domain.* You don't know how much load you'll need your database to support. You don't know if the library you selected is stable enough for your needs. Be prepared to change the vendors even after the initial release of your product.
- *Automated testing.* Stubs work great in reducing load on the test server. And stubs for the *SPI*s which you wrote yourself are the easiest for you to write.
- *Zero bug tolerance.* The SPIs allow for event replay. If the business logic is deterministic, you can reproduce your user's bugs at your development station.

Hexagonal Architecture <u>is not good</u> for:
- *Small modules.* If there is little business logic, there is not much to protect, while the overhead of defining *SPI*s and writing *adapters* is high compared to the total development time.

- *Write-and-forget projects.* You don't want to waste your time on the long-term survivability of your code.
- *Quick start*. You need to show the results right now. No time for good architecture.
- *Low latency*. The *adapters* slow down the communication. This is somewhat alleviated by creating direct communication channels between the *adapters* to bypass the *core*.
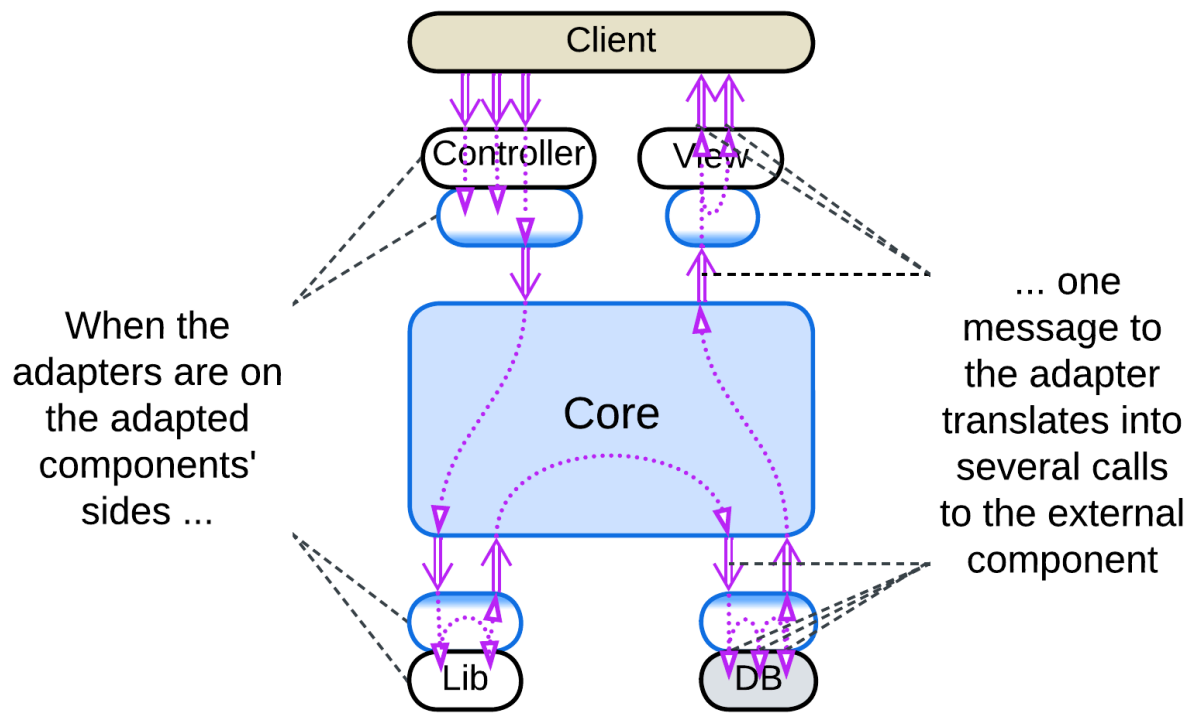
Relations

Monolithic hexagonal architecture   **Layered hexagonal architecture**   **Modular hexagonal architecture**
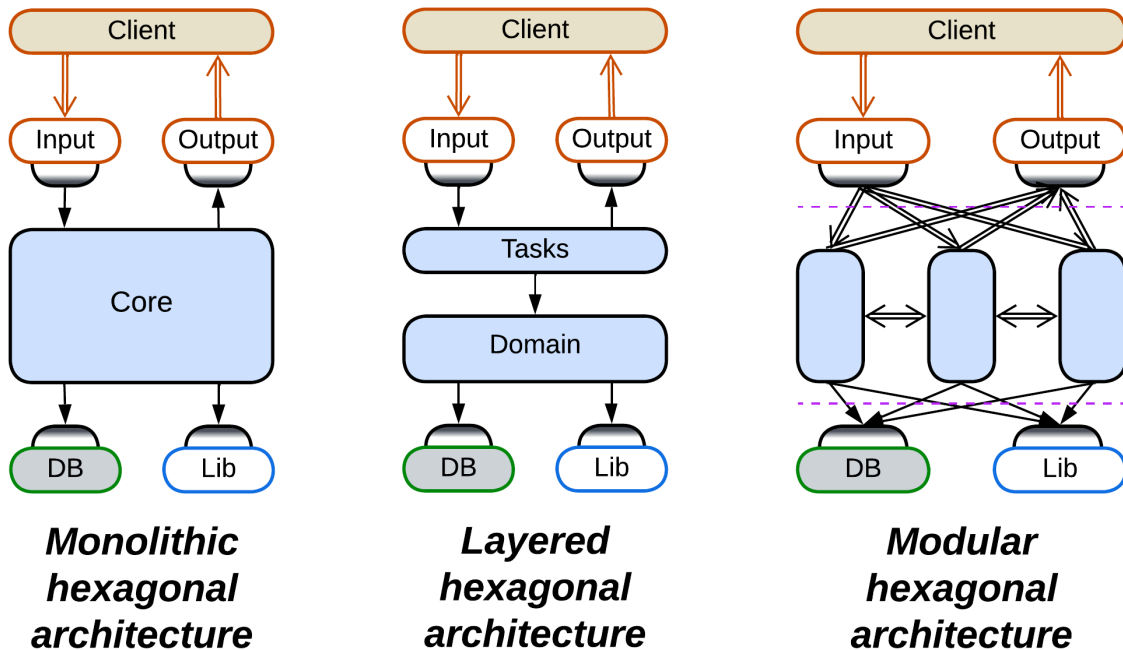
Hexagonal Architecture:
- Is a kind of *Plugins*.
- May be a shallow *Hierarchy*.
- Implements *Monolith* or *Layers*.
- Extends *Monolith*, *Layers* or, rarely, *Services* with one or two layers of *services*.
- *MVC* is also derived from *Pipeline*.

Variants by placement of adapters

One possible variation in a distributed or asynchronous *Hexagonal Architecture* is the deployment of *adapters*, which may reside with the *core* or with the components they adapt:

Adapters on the external component side

When the adapters are on the adapted components' sides ...

... one message to the adapter translates into several calls to the external component

If the team owns the component adapted, the *adapter* may be placed next to it. That usually makes sense because a single domain message (in the terms of the business logic) tends to unroll into a series of calls to the external component. The less messages you send, the faster your system is.

This resembles *Ambassador* [DDS] and *Open Host Service* [DDD].

Adapters on the core side

If you don't own the external components ...

... you need to adapt them on your side
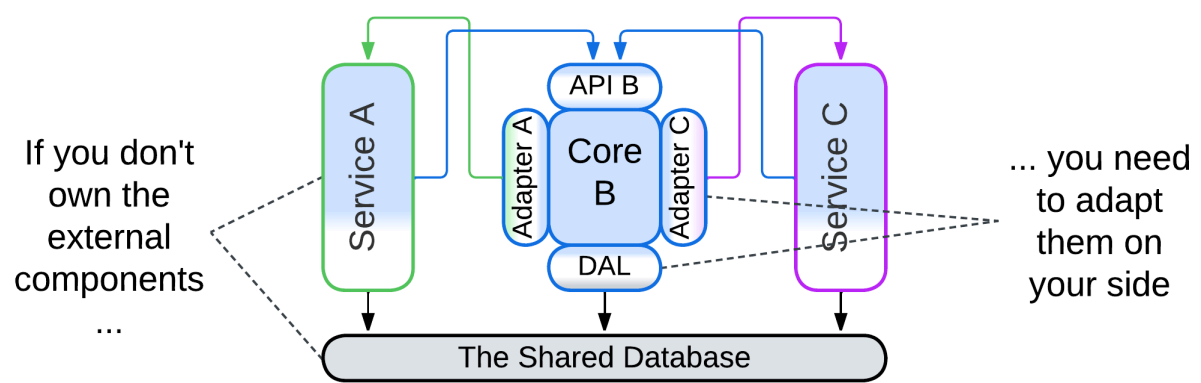
Sometimes you need to adapt external services which you don't control. In that case the only real option is to place their *adapters* together with your core logic. In theory, the *adapters* can be deployed as separate components, maybe in a *sidecar* [DDS], but that may slow down the communication.

This resembles *Sidecar* [DDS] and *Anticorruption Layer* [DDD].
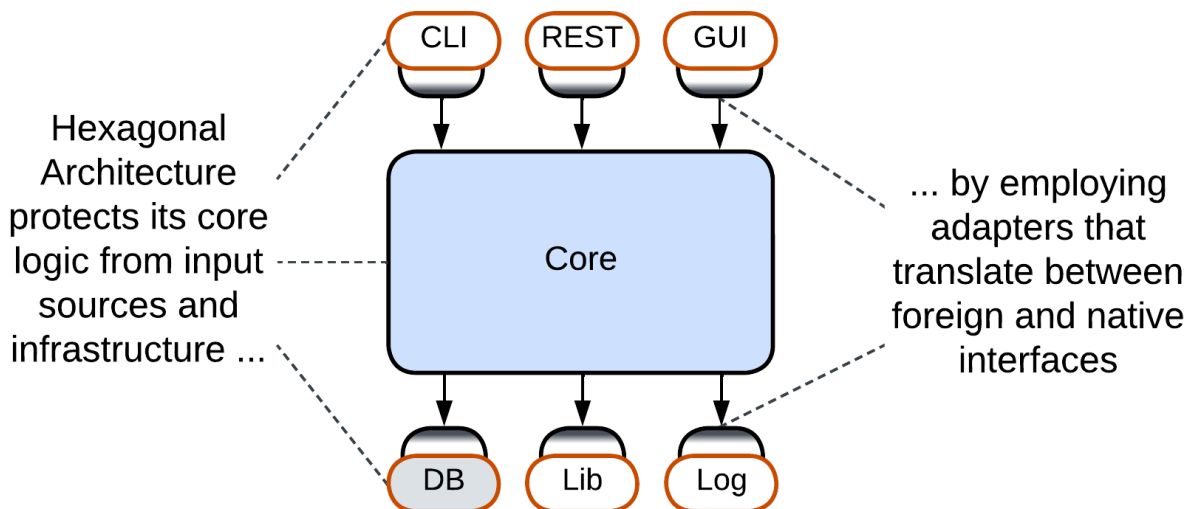
Examples

Hexagonal Architecture is simple and unambiguous. It does not come in many shapes:

Model-View-Controller (MVC)

The controller receives keyboard and mouse inputs and interprets them as user requests

Controller   View

The view shows updates and notifications to the users

Model

The view and controller make a platform-specific user interface

The model is the platform-agnostic business logic.
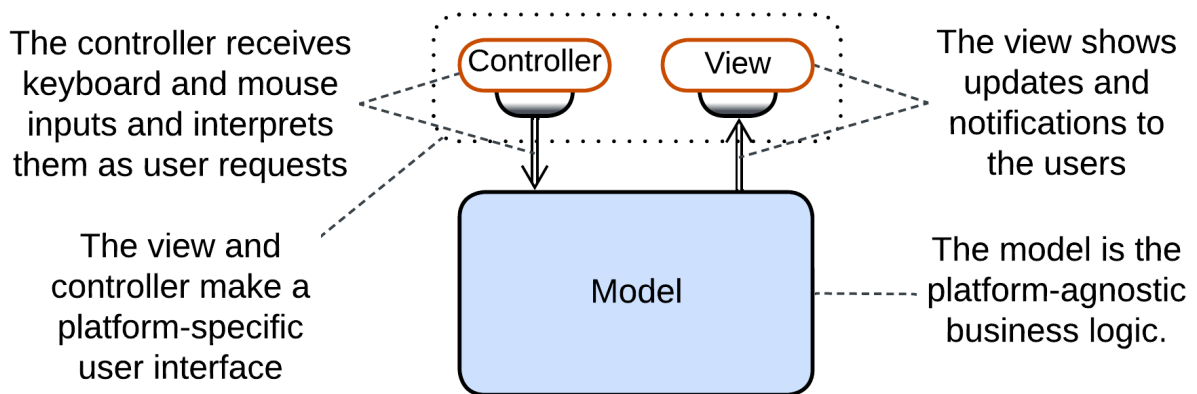
Model-View-Controller (*MVC*) [POSA1, POSA4] abstracts the *core* (called "*model*") from the user interface, which is divided into input ("*controller*") and output ("*view*"). The pattern allowed for cross-platform development at the time when there were no cross-platform UI frameworks and its convenience made it an industry standard – just as it happened to the more generic *Hexagonal Architecture* which was defined two decades later. An interesting detail about *MVC* is that it features a unidirectional event flow, making it a kind of *Pipeline*. This results from the pattern's omitting low-level dependencies and concentrating on the means of input (event producer) and output (event consumer).

Hexagonal Architecture, Ports and Adapters

CLI   REST   GUI

Hexagonal Architecture protects its core logic from input sources and infrastructure ...

Core

... by employing adapters that translate between foreign and native interfaces

DB   Lib   Log

Just like *MVC* it is based on, the original *Hexagonal Architecture* (*Ports and Adapters*) does not care about the contents or structure of its *core* – it is all about isolating the *core* from the environment. There may be layers or modules or even plugins inside, but the pattern does not have anything to say about them.

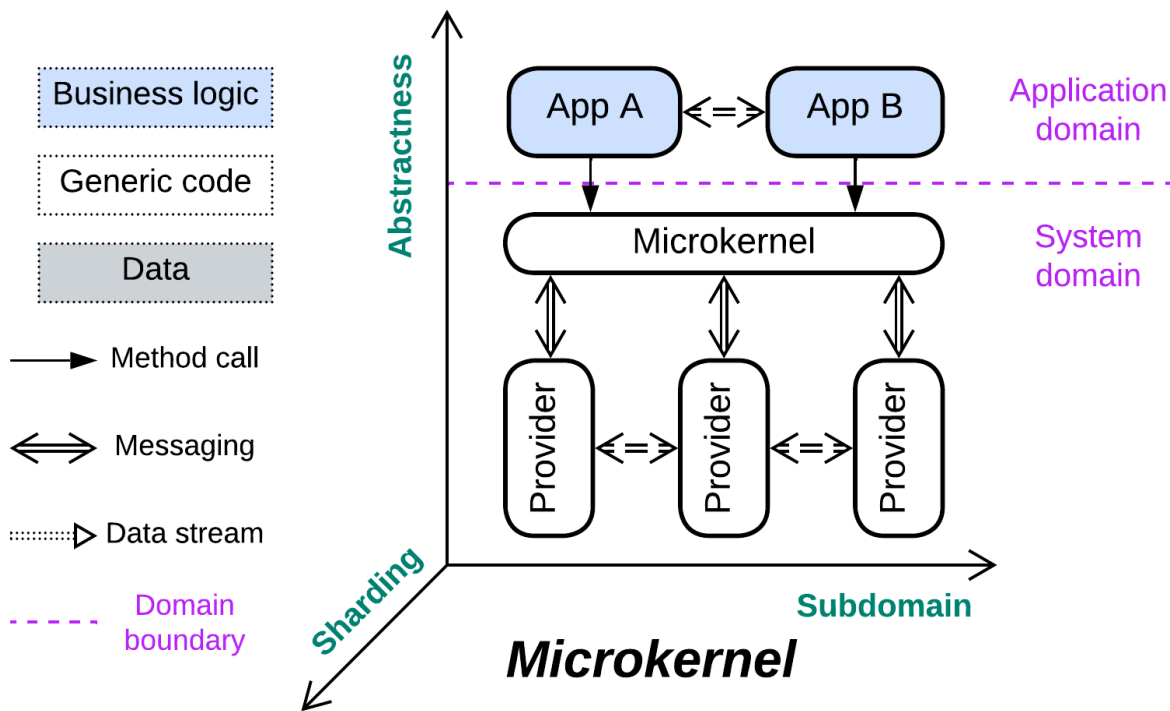DDD-style Hexagonal Architecture, Onion Architecture, Clean Architecture

Puristic DDD approach   **Pragmatic DDD approach**   **Onion Architecture**
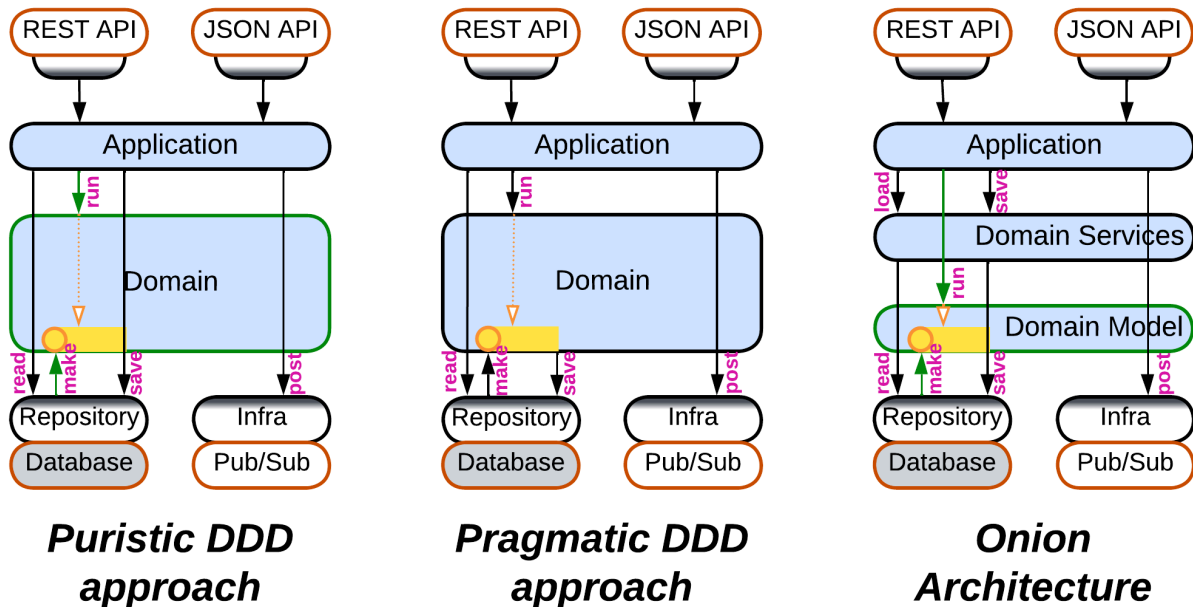
As *Hexagonal Architecture* built upon the DDD idea of isolating business logic with adapters, it was quickly integrated back into DDD [LDDD]. However, as it appeared later than the original DDD book, there is no universal agreement on how the thing should work:

- The cleanest way is for the *Domain* layer to have nothing to do with the database – with this approach the *application* asks the *repository* (the database adapter) to create *aggregates* (domain objects), then executes the business actions on the aggregates, and asks the *repository* to save the changed *aggregates* back to the database.
- Others say that in practice the logic inside an *aggregate* may have to read additional information from the database or even depend on the result of storing parts of the aggregate. Thus it is *aggregate*, not *application*, that saves the changes, and the logic of accessing the database leaks into the *Domain* layer.
- *Onion Architecture*, one of the early developments of *Hexagonal Architecture* and DDD, splits the *Domain* layer into *Domain Model* and *Domain Services*. The *Domain Model* layer contains classes with business data and business logic, which are loaded and saved by the *Domain Services* layer just above it. And the upper *Application Services* layer drives use cases by calling into both *Domain Services* and *Domain Model*.
- There is also *Clean Architecture* which seems to generalize the approaches above without delving into the practical details – thus the way it saves its *aggregates* remains a mystery.

Summary

Hexagonal Architecture isolates a component's business logic from any external dependencies by the use of 2-way adapters between them. It protects from *vendor lock-in* and allows for late changes of 3rd party components but relies on preliminary API design and often hinders performance optimizations.

Microkernel

Business logic

Generic code

Data

→ Method call

⟺ Messaging

⊳ Data stream

- - - - Domain boundary

App A ⟨=⟩ App B — Application domain

Microkernel — System domain

Provider ⟨=⟩ Provider ⟨=⟩ Provider — Subdomain

Abstractness

Sharding

Microkernel

Communism. Share resources among consumers.
Known as: Microkernel [POSA1, POSA4].
Aspects:
- Middleware,
- Orchestrator.

Variants:
- Operating system,
- Software framework,
- Virtualizer / Hypervisor / Container Orchestrator [DDS] / Distributed runtime,
- Interpreter [GoF] / Script / Domain-Specific Language (DSL),
- Configurator / Configuration file,
- Saga Engine,
- AUTOSAR Classic Platform.

Structure: A layer of *orchestrators* over a *middleware* over a layer of *services*.
Type: Implementation.

Benefits	*Drawbacks*
The system's complexity is evenly distributed among the components	The API and SPIs are very hard to change
Polymorphism of the resource providers	Performance is suboptimal
Most of the components are independent in qualities	Latency is often unpredictable
A resource provider can be implemented and tested in isolation	
Each application is sandboxed by the microkernel	

The system is platform-independent

References: Microkernel pattern in [POSA1].

While vanilla *Plugins* and *Hexagonal Architecture* keep their business logic in the monolithic core component, *Microkernel* treats the core as a thin *middleware* ("*microkernel*") that connects user-facing applications ("*external services*") to resource providers ("*internal services*"). The resource in question can be anything from CPU time or RAM to business functions. The *external services* communicate with the *microkernel* through its *API* while the *internal services* implement the *microkernel's Service Programming Interfaces* (*SPIs*) (usually there is a kind of *internal service* and an *SPI* per resource type).

On one hand, the pattern is very specific and feels esoteric. On the other – it is indispensable in many domains, with many more real-life occurrences than would be expected. *Microkernel* is used when there are a variety of applications that need to use multiple shared resources, with each resource being independent of others and requiring complex management.

Performance

The *microkernel*, being an extra layer of indirection, degrades performance. The actual extent varies from a few percent for OSes and virtualizers to an order of magnitude for scripts. A more grievous aspect of performance is that latency becomes unpredictable as soon as the system runs short of one of the shared resources: memory, disk space, CPU time or even hard drive space for deleted objects. That is why real-time systems rely on "real-time OS"es with rudimentary features or even run on bare metal.

It is common to see system components communicate directly via shared memory or sockets bypassing the *microkernel* to alleviate the performance penalty it introduces.

Dependencies

The *applications* depend on the *API* of the *microkernel* while the *providers* depend on its *SPIs*. On one hand, that isolates the *applications* and *providers* from each other, letting them develop independently. On the other hand, the *microkernel's API* and *SPIs* should be very stable to support older versions of the components which the *microkernel* integrates.

Applications rely on the microkernel's API

Service providers implement the microkernel's SPIs
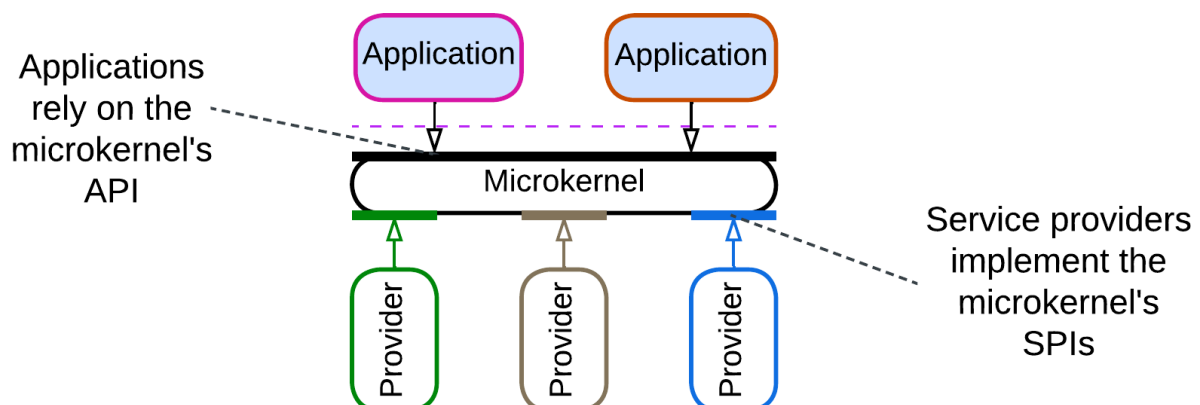
Applicability

Microkernel is applicable to:

- *System programming.* You manage system resources and services which will be used by untrusted client applications. Hide the real resources behind a trusted proxy layer. Be ready to change the hardware platform without affecting the existing client code.
- *Frameworks that integrate multiple subdomains.* The *microkernel* component coordinates multiple specialized libraries. Its *API* is a *facade* [GoF] for the managed functionality.
- *Scripting or DSLs.* The *microkernel* is an *interpreter* [GoF] which lets your clients' code manage the underlying system.

Microkernel does not fit:

- *Coupled domains.* Any degree of coupling between the *internal services* complicates the *microkernel* and *SPIs* and is likely to degrade performance. The performance may often be salvaged by introducing direct communication channels between the *services*.

Relations

Microkernel as middleware   **Microkernel as orchestrator**   **Microkernel as BFF**
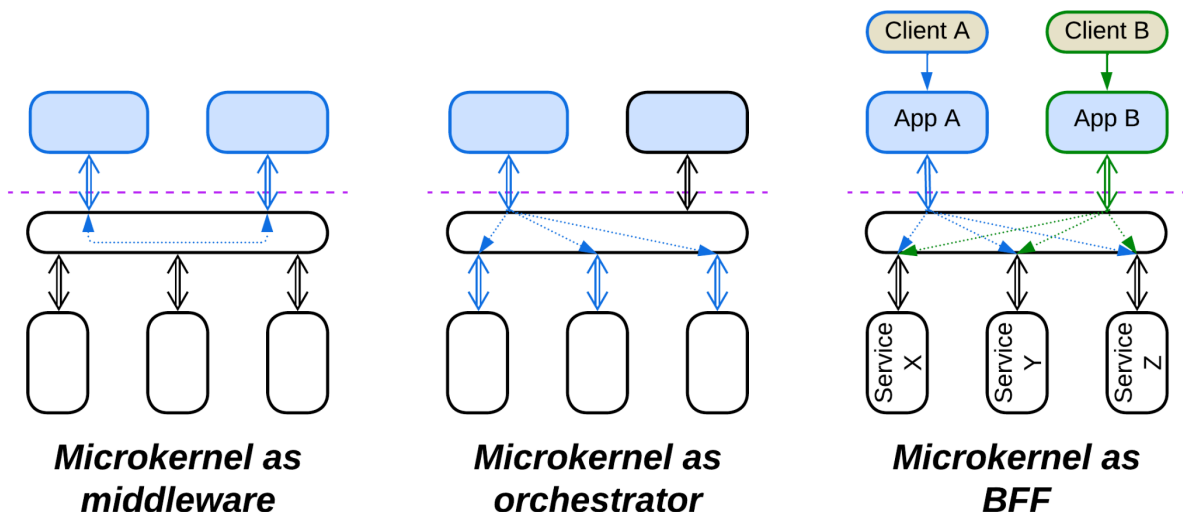
Microkernel:

- Is a specialization of *Plugins*.
- Is related to *Backends for Frontends*, which is a layer of *orchestrators* over a layer of *services*: *Microkernel* adds a *middleware* in between.
- Is a kind of 2-layered *SOA* with an *ESB*.
- The *microkernel* layer serves as (implements) a *middleware* for the upper (*external*) *services* and often as an *orchestrator* for the lower (*internal*) *services*.
- May be implemented by *Mesh*.

Variants

Microkernel appears as many a species:

Operating system

All the user applications share system resources

Each application gets an instance of runtime that interfaces the kernel and housekeeps allocated resources

Each hardware module has a dedicated driver

User space

Kernel space

The kernel manages and glues all the components
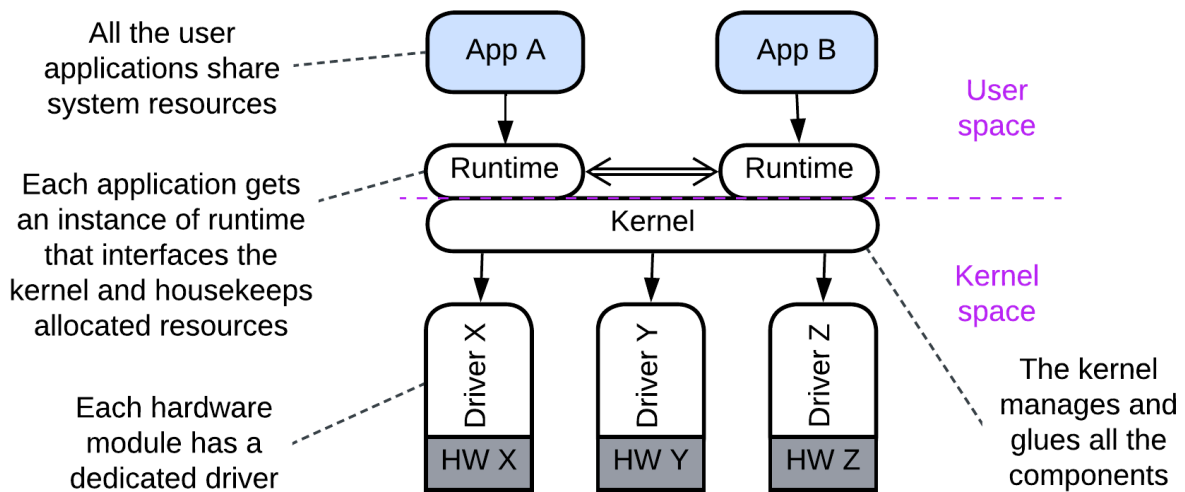
The original inspiration for *Microkernel*, operating systems provide a nearly perfect example of the pattern, even though their kernels are not that "micro-". Device *drivers* (*internal services*) encapsulate available hardware resources and make them accessible to user-space *applications* (*external services*) via an OS *kernel*. The *drivers* for the same kind of subsystem (e.g. network adapter or disk drive) are polymorphic and need to match the installed hardware.

Software framework

A framework provides a simple unitified interface to an underlying system

User applications build on the framwork's API without having to face its internal complexity
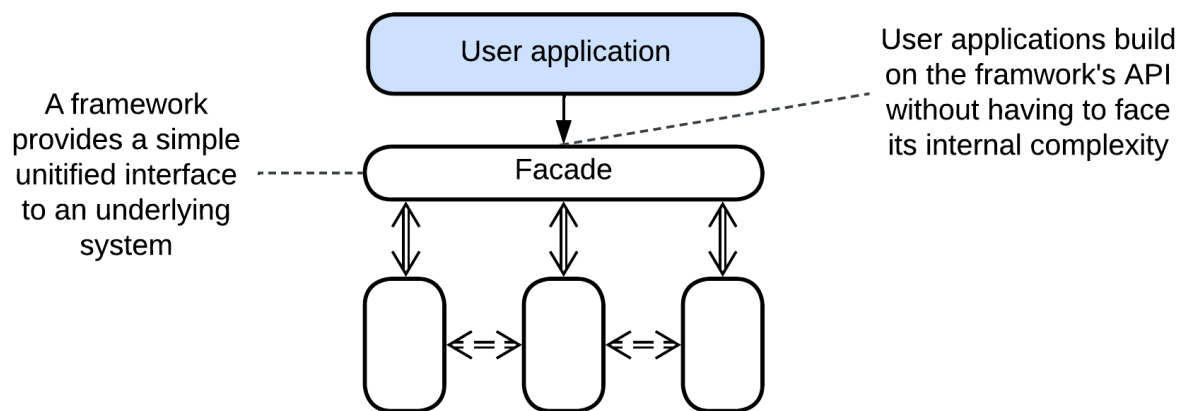
The *microkernel* is a *facade* [GoF] that integrates a set of libraries and exposes a user-friendly high-level interface. PAM looks like a reasonably good example.

Virtualizer, Hypervisor, Container Orchestrator, Distributed runtime

The virtualizer runs user applications on computers that have enough free resources

App A

App B

It also creates means of communication for the user applications

Virtualizer

It manages the running processes and can even move one between servers

OS

Hardware
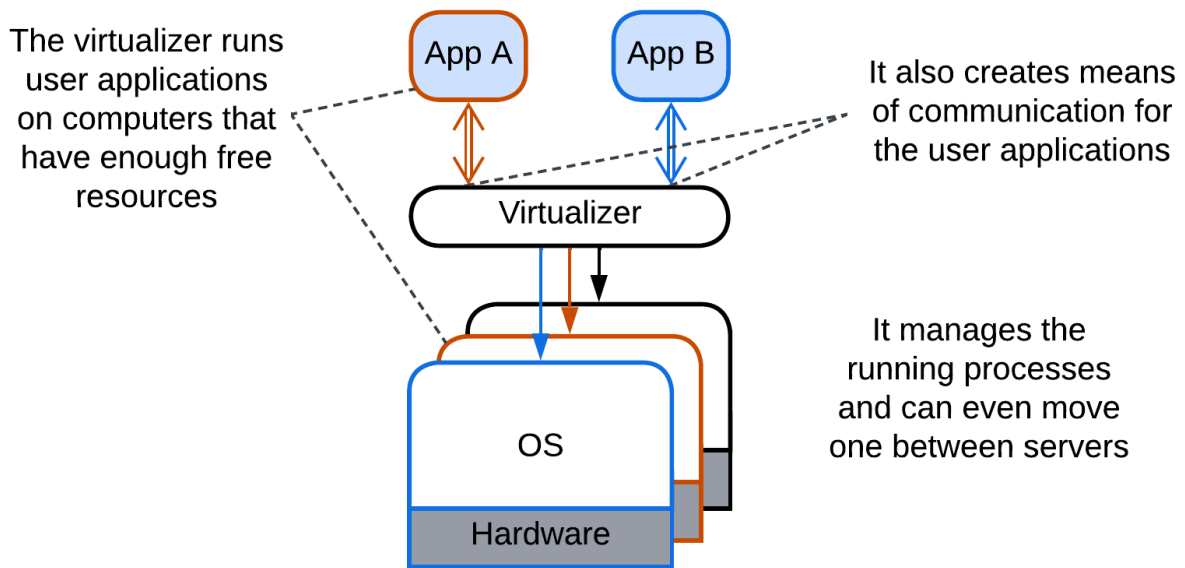
Hypervisors (Xen), PaaS and FaaS, container orchestrators (Kubernetes) [DDS], distributed *actor frameworks* (Akka, Erlang/Elixir/OTP) use resources of the underlying computer(s) to run guest applications. A *hypervisor* virtualizes resources of a single computer while a *distributed runtime* manages those of multiple servers – in that case there are several instances of the same kind of an *internal server* which abstracts a host system.

Interpreter, Script, Domain-Specific Language (DSL)

The scripts contain custom business logic which builds on (orchestrates) the libraries

Script A

Script B

Each user script gets its own instance of interpreter

Interpreter

Interpreter

The libraries implement heavy calculations and wrap system interfaces

Lib X

Lib Y

Lib Z

The interpreter (obviously) interprets the script and provides it with access to the underlying libraries
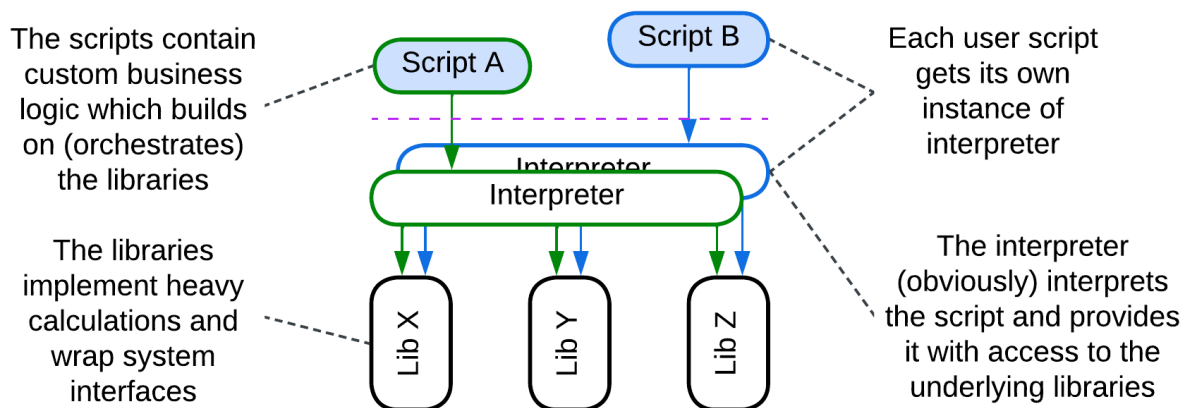
User-provided scripts are run by an *interpreter* [GoF] which also allows them to access a set of installed libraries. The *interpreter* is a *microkernel*, the syntax of the script or DSL it interprets is the *microkernel's* API.

Configurator, Configuration file

The configuration file contains setting for the system

Configuration file

The configurator runs at an early stage of the system's startup and reads the configuration ...

Configurator

... to know how to create and initialize the components of the system

The system is what we want to stay there and do the job
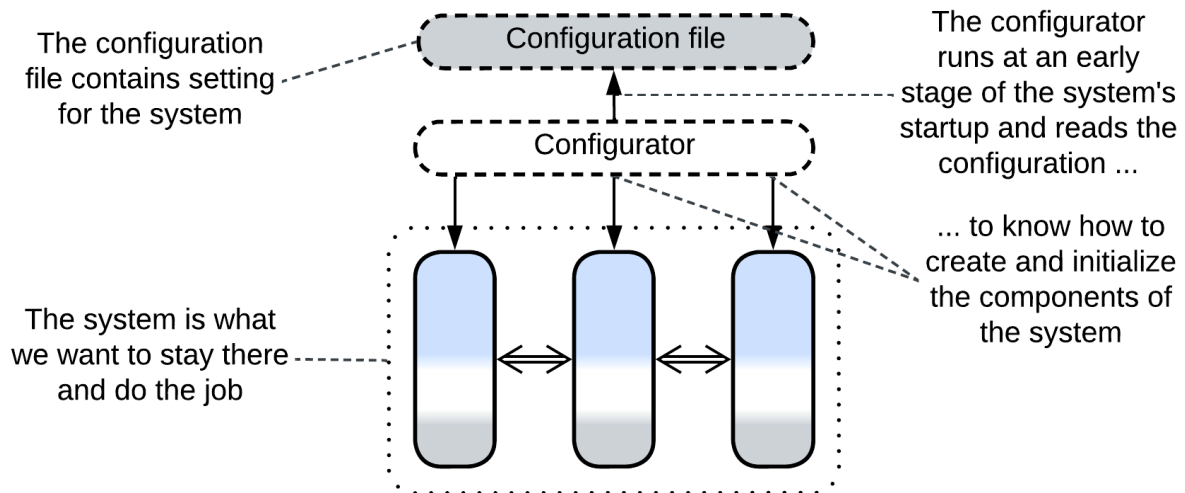
Configuration files may be regarded as short-lived scripts that configure the underlying modules at the start of the system. The parser of the configuration file is a transient *microkernel*.

Saga Engine

Sagas oversee distributed transactions - "all or nothing" writes to several services

Saga A
Saga A
Saga A

Saga B
Saga B

The saga engine manages saga instances, sends their requests to the services, and brings back responses

Saga engine

Service X

Service Y

Service Z

Each service owns its data
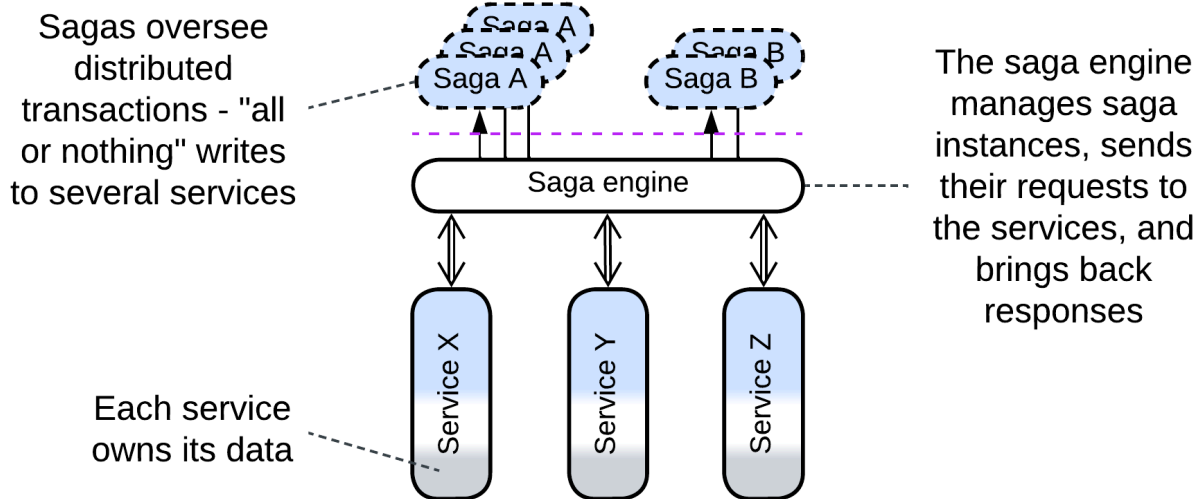
A *saga* [MP] orchestrates distributed transactions. It may be written in a DSL which requires a compiler or interpreter, which is a *microkernel*, to execute.

AUTOSAR Classic Platform

VFB matches applications to hardware and interconnects the applications

RTE virtualizes hardware

The applications contain business logic. They message each other, RTE services and drivers.

App X App A	Application layer
App B App C	
App Z	

Virtual Functional Bus (VFB)

— — — — RTE layer

| Runtime Env (RTE) | Runtime Env | Runtime Env (RTE) |

Driver X	Services X		Generic services	Driver Z	Generic services	Basic Software (BSW) layer
	ECU abstraction		ECU abstraction		ECU abstraction	
	µC X abstraction		µC Y abstraction		µC Z abstraction	

| Hardware X | Hardware Y | Hardware Z |

The notorious automotive standard, though promoted as *SOA*, is structured as a distributed / virtualized *microkernel*. The application layer comprises a network of *software components* spread over hundreds of chips for some reason called *electronic control units* (*ECU*s). The communication paths between the *software components* and much of the code are static and auto-generated. A *software component* may access hardware of its *ECU* via standard interfaces.

The *microkernel* shows up as *Virtual Functional Bus* (*VFB*) which, as a *distributed middleware*, provides communication between the *applications* by virtualizing multiple *Runtime Environments* (*RTEs*) – the local system interfaces.

Summary

Microkernel is a ubiquitous approach to sharing resources among consumers, where both resource providers and consumers may be written by external companies.

Mesh

Business logic

Generic code

Data

→ Method call

⟺ Messaging

⫶⫶⫶▷ Data stream

- - - - Domain boundary

App A   App B   App C

User domain

Mesh domain

Node
Node
Node

Abstractness

Sharding

Subdomain
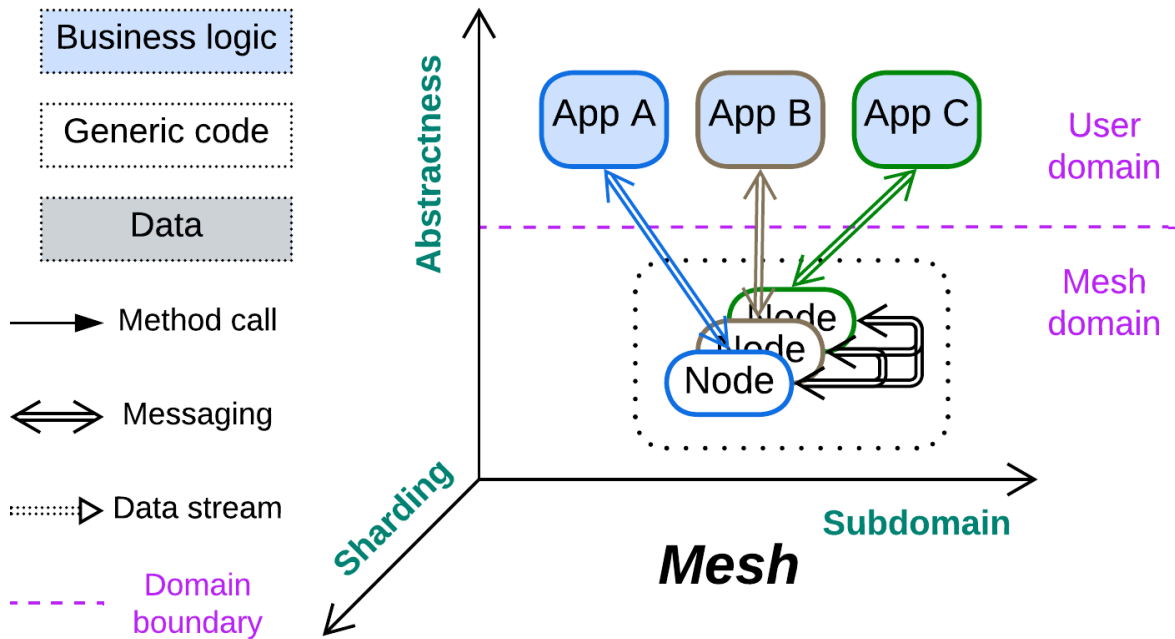
Mesh

Hive mind. Go decentralized.
Known as: Mesh, Grid.
Aspects: similar to Middleware.
Variants: mashes very greatly. Examples include:
- Peer-to-Peer Networks,
- Leaf-Spine Architecture / Spine-Leaf Architecture,
- Actors,
- Service Mesh [FSA, MP],
- Space-Based Architecture [SAP, FSA].

Structure: A system that uses interconnected *layered shards* as a *middleware*.
Type: Implementation.

Benefits	Drawbacks
No single point of failure	Overhead in administration and security
The system is able to self-heal	Performance is likely to suffer
Great scalability	The *mesh* itself is very hard to debug
Available off the shelf	Unreliable communication must be accounted for in the code

References: Wiki and [DDIA] on topology and protocols. [FSA] on *Service Mesh* and *Space-Based Architecture*. A long and short article on *Service Mesh*.

If a system is required to survive faults, all its components must be sharded and interconnected, which makes a *mesh* – a network of interacting instances. In most cases the lower layer of a shard implements connectivity while the business logic resides in the upper layer(s). Whereas the connectivity component tends to be identical in all the nodes in a system, the upper components may be identical – forming *Shards*, or different – forming *Services*.

Most *meshes* allow for dynamically adding and removing parts of their networks, as required for scaling up, scaling down and fault recovery. That is achieved through a flexible network topology, which has the downside of *communication artifacts* [MP], i.e. missing or duplicated requests, a single request being processed by two instances of a service or by the same instance twice, etc. Moreover, a *mesh*-mediated communication is likely to be slower than the direct one.

Performance

In most (all?) implementations the user *application* is colocated with a *node* of the *mesh*, thus communicating through the *mesh* does not add an extra network hop (which strongly degrades performance). However, that holds only when the node knows the destination of the message it should send – when it has already established a communication channel. Finding the destination may not always be easy – that often requires consulting registries, and sometimes – waiting for the network topology to stabilize, which may involve timeouts – as often seen with torrents. On the other hand, no other architecture is known to seamlessly support huge networks.

Dependencies

Mesh, as a *sharded middleware*, inherits dependencies from both of its parent patterns:
- As with *Middleware*, each service that runs over the *mesh* depends on the *mesh's* API. The *services* may also depend on each other or on a *shared database*, if they use one to communicate through.
- As with *Shards*, the *nodes* of the *mesh* should communicate through a backward- and forward-compatible protocol as there will likely be periods of time when multiple versions of the *mesh nodes* coexist.

Applicability

Mesh is perfect for:
- *Dynamic scaling.* Instances of services may be quickly added or removed.
- *High availability.* A *mesh* is very hard to disable or kill because it creates new instances of failed services and finds ways around failed connections.

Mesh fails in:
- *Low latency domains.* Spreading information through a *mesh* is slow and sometimes unreliable.
- *Security-critical systems.* A public *mesh* exposes a high attack surface while the scalability of private deployments is limited by the installed hardware.
- *Quick and dirty programming*. The possible message duplication may cause evil bugs if the APIs ignore the risk.

Relations

Mesh:
- Misuses *Shards*.
- Uses *Layers*.
- Is a base for running multiple instances of *Monolith*, *Layers* or *Services*.
- Implements distributed *Middleware*, *Shared Repository* or *Microkernel*.

Variants

Meshes are known to vary:

By structure

The connections in a *mesh* may be:
- Structured / pre-defined – the *mesh* is pre-designed and hard-wired. This kind of *meshes* provides redundancy but not scalability.
- Unstructured / ad-hoc – *nodes* may be added and removed at runtime, causing restructuring of the *mesh*. The flexibility of the structure is the source of *artifacts* [MP].

By connectivity

Each node is:
- Connected to all other nodes – a *fully connected mesh*. Such *meshes* are limited in size because the number of interconnections grows as a square of the number of the *nodes*. Notwithstanding, they offer the best communication speed and delivery guarantees.
- Connected to some other *nodes*. There are many possible topologies with the correct choice for a given task better left to experts.

By the number of mesh layers

The connected nodes of a *mesh* may be:
- Identical (one-layer *Mesh*). A *node* behaves according to its site in the network.
- Specialized (multi-layer *Mesh*). Some *nodes* implement *trunk* (route messages and control the topology) while others are *leaves* (run user applications).

Examples

Peer-to-Peer Networks

Instances of the application implement the business logic, they use the mesh to access each other

Each instance of the application is colocated with a mesh node

The registry knows of all the nodes

The nodes intercommunicate directly

A node queries the registry for addresses of other nodes it is interested in
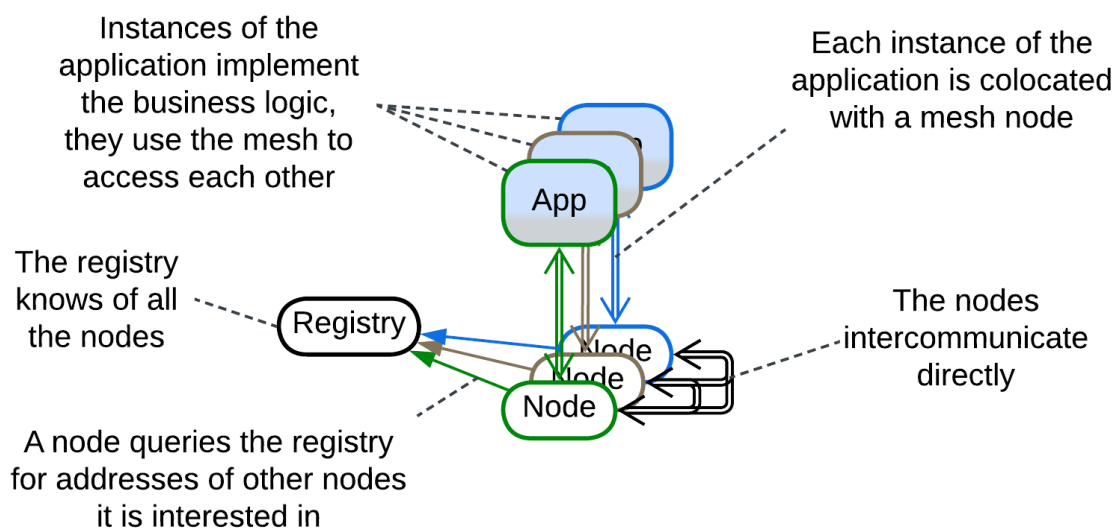
Peer to peer (*P2P*) networks are intended for massive resource sharing over unstable connections. The *resource* in question may be data (torrents, blockchain, P2PTV), CPU time (volunteer computing, distributed compilation) or Internet access (Tor). In most applications it is shared over an *unstructured* (as participants join and leave) *2-layer* (there are dedicated servers that register and coordinate users) network which is *overlaid* on top of the Internet. All the leaf nodes run identical narrowly specialized (i.e. either file share or blockchain but not both) software which provides the clients with access to resources of other nodes, working as a kind of distributed *middleware*.

Examples: torrent, onion routing (tor), blockchain.

Leaf-Spine Architecture, Spine-Leaf Architecture

Clients are servers in a datacenter

In such a configuration a failure of a component affects at most one client

Each client is connected to a leaf node

Each leaf is connected to all spine nodes

Parallel communication paths result in high bandwidth with commodity hardware
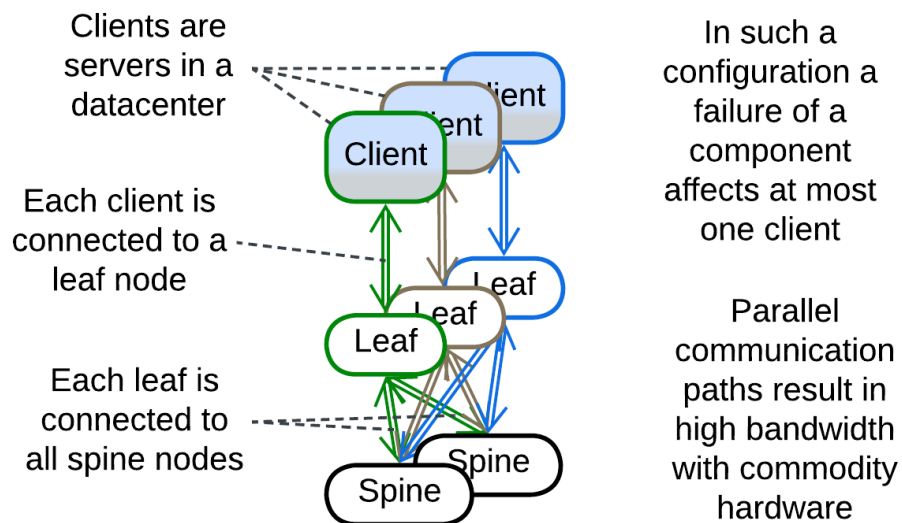
This datacenter network architecture is a rare example of a *structured fully connected mesh*. It consists of client-facing (*leaf*) and internal (*spine*) switches. Each *leaf* is connected to each *spine*, allowing for very high bandwidth (by distributing the traffic over multiple routes) that is almost insensitive to failures of individual hardware as there are always multiple parallel paths.

Actors

The single-threaded actors contain the business logic and often the data

Each actor can post messages to message queues of all the actors it knows about

Each actor is driven by its message queue which serializes incoming requests

There is no other means of communication
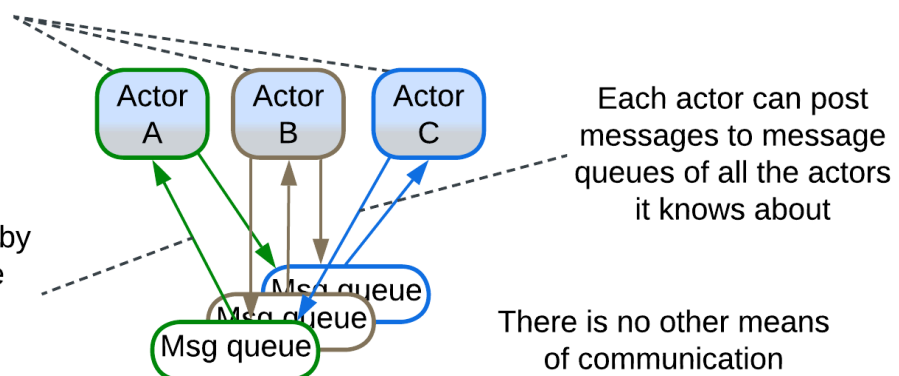
A system of *actors* may be classified as a *fully connected mesh* with the *actors'* message queues being the nodes of the *mesh*. Any *actor* can post messages to the queue of any

other *actor* it knows about, as all the *actors* share a virtual namespace or physical address space.

Service Mesh
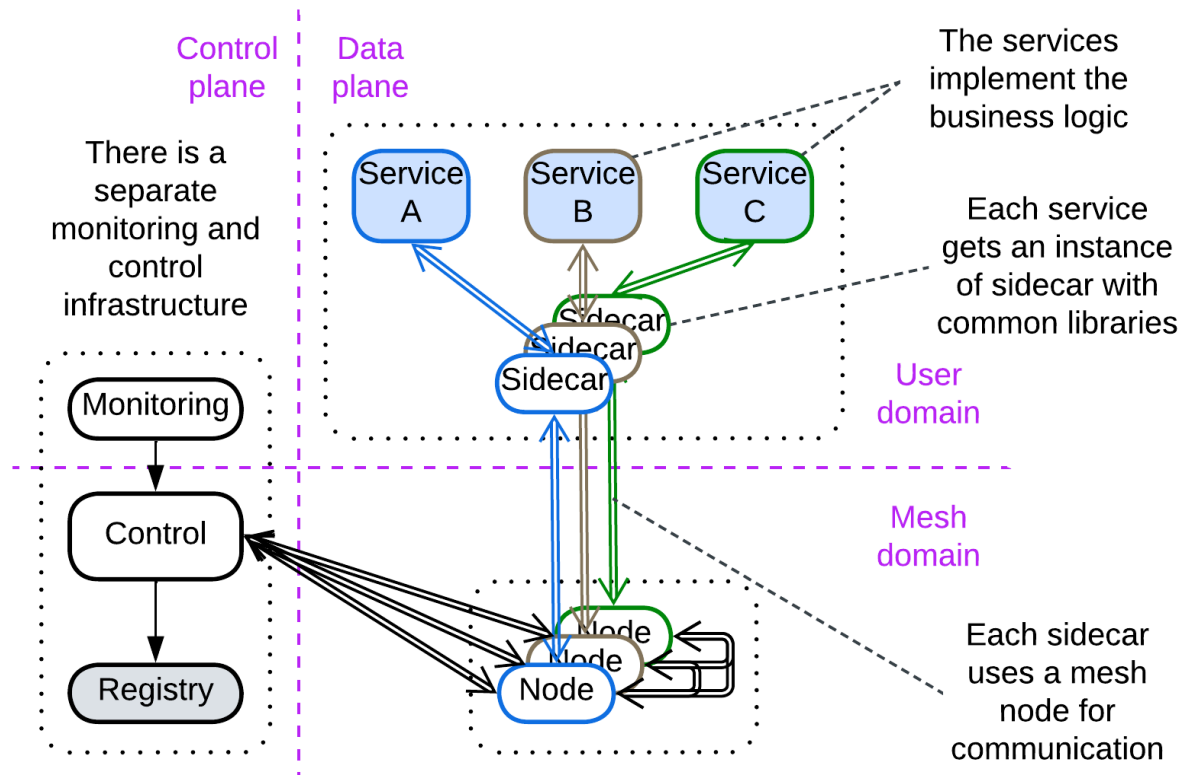
Service Mesh [FSA, MP] is a distributed *middleware* for running *Microservices*. It is a 2-layer *mesh* which contains one or few management nodes (*control plane*) and many user nodes (*data plane*). Each *data plane* node colocates:

- A *mesh engine node* that deals with connectivity,
- One or more *sidecars* [DDS] (*proxies* where the support of cross-cutting concerns – the identical code for use by all the *services*, like logging or encryption – resides),
- A user *application* (the *microservice*) which differs from node to node.

The *control plane* (re-)starts, updates, scales and collects statistics from the nodes of the *data plane*.

Service Mesh addresses several weaknesses of naive *Services*: it provides tools for centralized management and allows for virtual sharing (through creating physical copies) of libraries to be accessed by all the *service* instances. It also takes care of scaling and load balancing. Ready-to-use *Service Mesh* frameworks are popular with *Microservices* architecture.

Space-Based Architecture

Space-Based Architecture

Processing units are services with the business logic

Unit A Unit B Unit C

User domain

Framework domain

Each service gets a node of an in-memory mesh database called data grid

Node Node Node

The nodes of the data grid synchronize their data ...

The data grid content is initialized from an external database

Reader Writer

... and persist any changes to the external database

Persistent database

Space-Based Architecture [SAP, FSA] is a kind of *service mesh* with an integrated *shared database* (a tuple space – shared dictionary – called *data grid*) and *orchestrator* (called *processing grid*). The user services are called *processing units*. They may be identical (yielding *Shards*) or different (resulting in *Services*). The architecture is used for:

- Highly scalable systems with smaller datasets, in which case the entire database contents are replicated to the memory of each node. This works around the throughput and latency limits of a normal database.
- Huge datasets, with each node owning a part of the total data. This hacks around the storage capacity and latency limits of a database.

There are multiple instances of the same data in *processing units*. Any change to the data in one unit must propagate to other units. That can be done:

- Asynchronously, causing conflicts if the same data is changed elsewhere simultaneously.
- Synchronously, waiting for the propagation results and conflict resolution – a kind of distributed transaction with poor latency.
- Take write ownership of the data before the write. That is not good for latency as well, but it may be a good choice for an evenly distributed load if the *mesh* provides temporary locality of requests, i.e. it forwards requests that touch the same data to the same node.

The choice of the strategy depends on the domain.

The in-memory data in the nodes is loaded from an external database at the start of the system and any change in it is replicated asynchronously to the external database. The persistent database serves as a means of fault recovery and a single source of truth.

Summary

Mesh is a layer of intercommunicating instances of an infrastructure component that makes a foundation for running custom services in a distributed environment. The architecture is famous for its scalability and fault tolerance but is too complex to implement in-house and may incur performance, administration and development overhead.

Part 6. Analytics

This part is dedicated to analysis of the architectural metapatterns, for if the classification is a step forward from the current state of the art, it should bear fruits to collect.

I had no time to research all the ideas that were gathered while the book was being written and its individual chapters published for feedback. Some of those pending topics, which may make additional chapters in the future, are listed below:

- Some architectural patterns (CQRS, *Cache*, *Microservices*, etc.) appear under multiple metapatterns. Each individual case makes a story of its own, teaching both of the needs of software systems and of uses of metapatterns.
- There are several grades of independence of components between cohesive *Monolith* and distributed systems. We may track how a dozen of architectural qualities of a system change as system components become more and more isolated.
- A component may be split into subcomponents, so that *Services* become *Layered Services* and *Layers* become *Layers of Services*. The effects of such a change on the system's qualities should be investigated.
- An architectural quality may depend on the structure of the system. For example, a client request may be split into three subrequests to be processed sequentially or in parallel, depending on the topology (e.g. *pipeline* or *orchestrated services*), thus it is topology which defines latency. We may even draw formulas like:
  - L = L1 + L2 + L3 for pipeline,
  - L = MAX(L1, L2, L3) for orchestrated services which run in parallel,
  - L = MIN(L1, L2, L3) for broadcasting the request to replicas and using the first response received.

Other smaller topics that I was able to look into made the following chapters:

Comparison of architectural patterns

This chapter is a compilation of small articles, each analyzing one aspect of the architectural patterns included in the book. It shows the value of having the list of metapatterns to iterate over and analyze.

Sharing functionality or data among services

Architectural patterns manifest several ways of sharing functionality or data among their components. Let's consider a basic example: calls to two pieces of business logic need to be logged, while the logger is doing something more complex than console prints. They also need to access a system-wide counter.

Direct call

The simplest use of a shared functionality is by directly calling the module that implements it. This is possible if the users and the provider of the aspect reside in the same process, as in _Monolith_ or module-based (single application) _Layers_.

Sharing data inside a process is similar, but usually requires some kind of protection, like RW lock, around it to serialize access from multiple threads.

If everything runs in the same process ...

Monolith   **Layers**

... components and data can be accessed directly
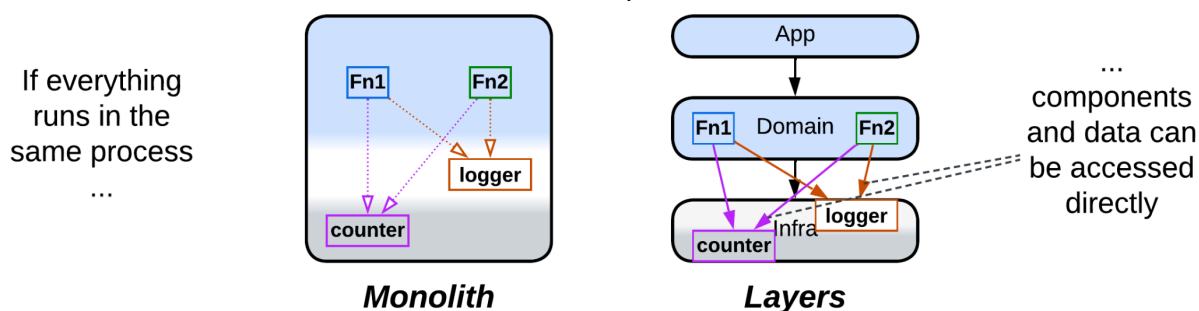
Make a dedicated service

In a distributed system you can place the functionality or data to share into a separate service to be accessed over the network, yielding _Service-Oriented Architecture_ for shared utilities or _Shared Repository_ / _Polyglot Persistence_ for shared data.

Both shared functionality ...

Service-Oriented Architecture   **Shared Database / Polyglot Persistence**

... and shared data can be deployed as separate services
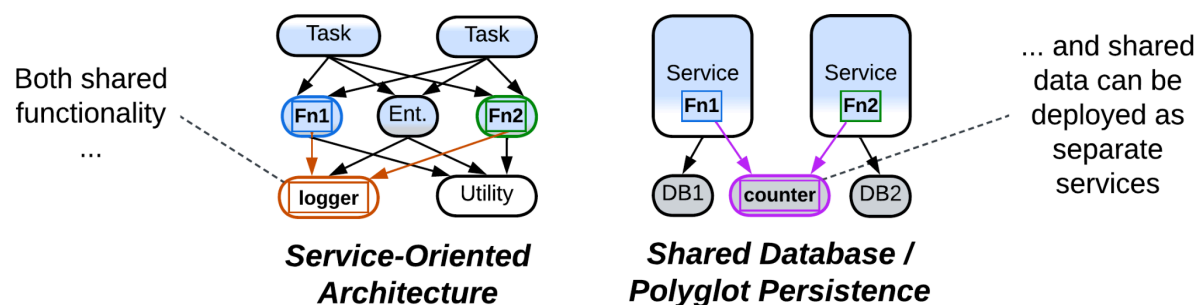
Delegate the aspect

A less obvious solution is delegating our needs to another layer of the system. To continue our example of logging, a *proxy* may log user requests and a *middleware* – interservice communication. In many cases either of these kinds of logging reveals the calls to the methods we need to log – with no changes in the code of the methods!

In a similar way a service may behave as a function: receive all the data it needs in the input message and send back all its work as the output – and let the database access be the responsibility of its caller.

We can use the
request log from
the proxy and the
log of internal
messages from
the middleware

**Services with a Proxy
and Middleware**

And we can let
our gateway
application read
and update the
shared data

**Pipeline with a
Gateway**
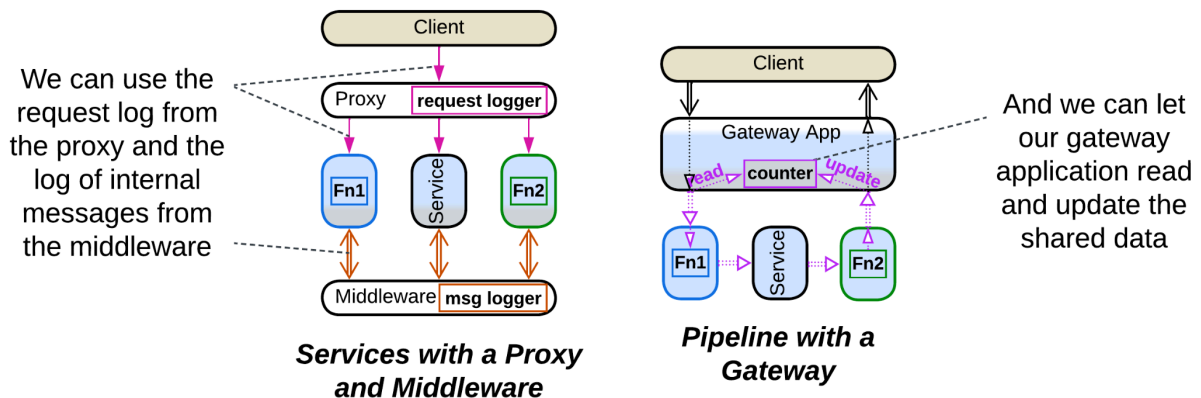
Replicate it

Finally, each user of a component can get its own replica. This is done implicitly in *Shards* and explicitly in *Service Mesh* of *Microservices* for libraries or *Data Grid* of *Space-Based Architecture* for data.

Another case of replication is including the same code in multiple services, which happens in single-layer *Nanoservices*.

Any module is
replicated
among shards

Shards

A nanoservice
may inlude the
code for logging

Nanoservices

Service Mesh
has sidecars
for shared
libraries

Service Mesh

Data Grid
replicates the
data to each
service

Data Grid
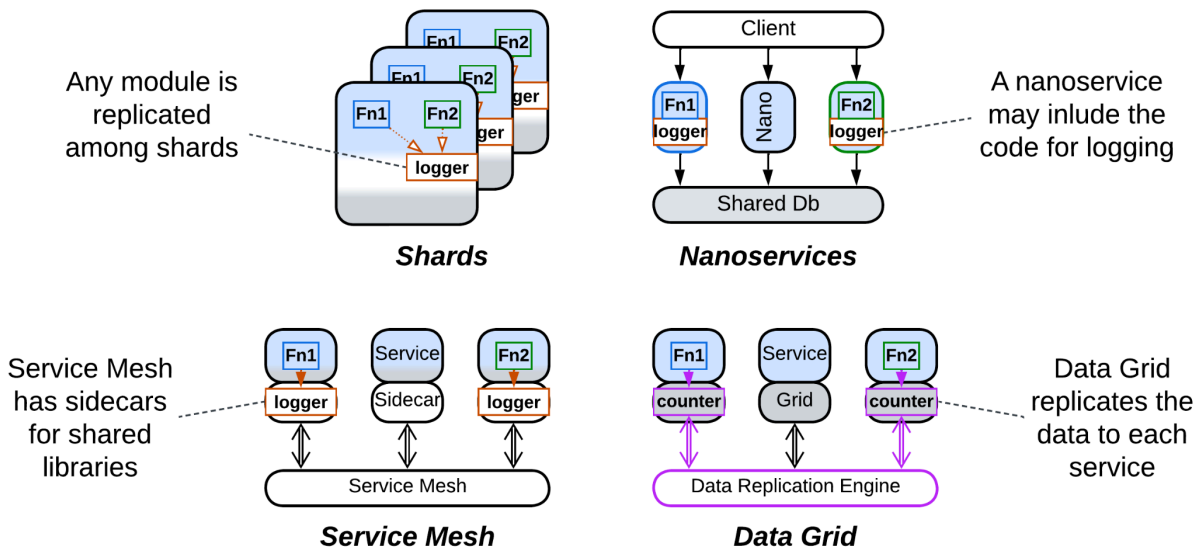
Summary

There are four basic ways of sharing functionality or data in a system:
- Deploy everything together – messy but fast and simple.

- Place the component in question into a shared service to be accessed over the network – slow and less reliable.
- Let another layer of the system both implement and use the needed function on your behalf – easy but generic, thus may not fit your needs.
- Make a copy of the component for each of its users – fast, reliable, but the copies are hard to keep in sync.

Pipelines in architectural patterns

Several architectural patterns involve a unidirectional data flow – a *pipeline*. Strictly speaking, every data packet in a pipeline should:
- Move through the system over the same *route* with no loops.
- Be of the same *type*, making a *data stream*.
- Retain its *identity* on the way.
- Retain *temporal order* – the sequence of packets remains the same over the entire pipeline.

Staying true to all of the above points yields *Pipes and Filters* – one of the oldest known architectures. Yet there are other architectures that discard one or more of the conditions:

Pipes and Filters

Each packet of data passes through the entire pipeline

A filter is a single step of processing the data stream

Pipes transfer data between the filters

Filter 1 → Filter 2 → Filter 3

The source inputs the data

Source

The data is never reordered

Sink

The sink outputs the results
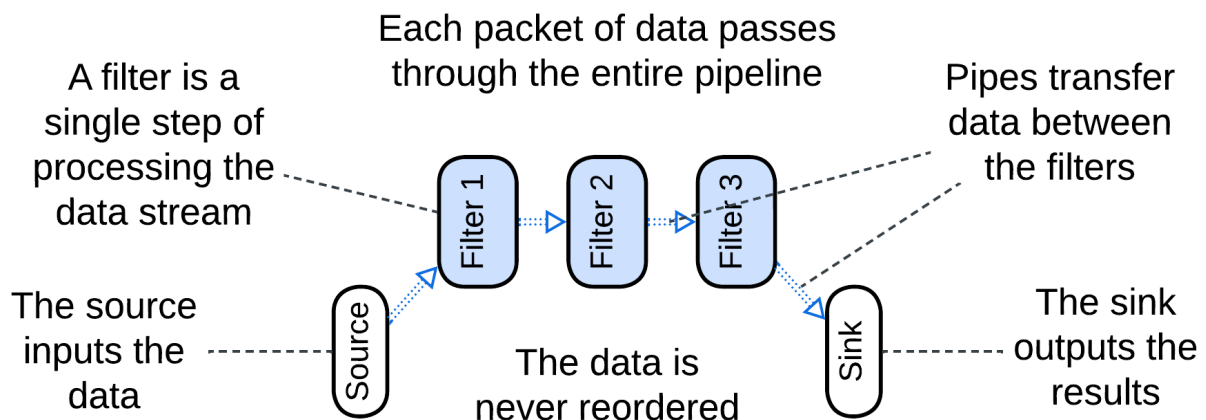
Pipes and Filters [POSA1] is about stepwise processing of a data stream. Each piece of data (a video frame, a line of text or a database record) passes through the entire system.

This architecture is easy to build and has a wide range of applications, from hardware to data analytics. Though a pipeline is specialized in a single use case, a new one can often be built of the same set of generic components – the skill mastered by Linux admins through their use of shell scripts.

Choreographed Event-Driven Architecture

Individual
parcels

Trucks of
parcels

The services
subscribe for
each other's
notifications

A service may
process
multiple inputs

Orders Warehouse Logistics Tracking

The pipeline
is branched

There are
multiple
pipelines
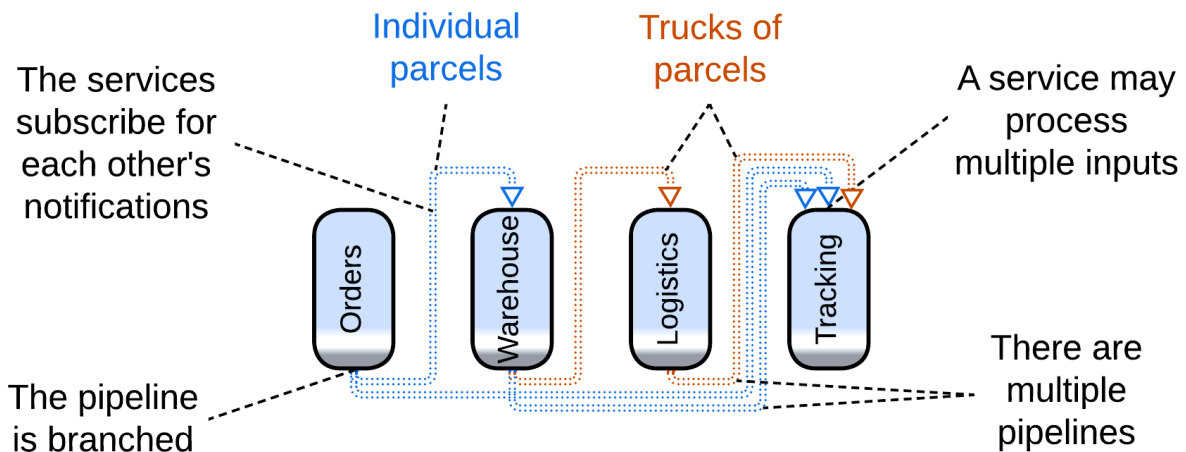
Relaxing the *type* and loosening the *identity* clauses opens the way to *Choreographed Event-Driven Architecture* [SAP, FSA], in which a service publishes notifications about anything it does that may be of interest to other services. In such a system:

- There are multiple types of events going in different directions, like if several branched pipelines were built over the same set of services.
- A service may aggregate multiple incoming events to publish a single, seemingly unrelated, event later when some condition is met. For example, a warehouse delivery collects individual orders till it gets a truckful of them or till the evening comes and no new orders are accepted.

This architecture covers way more complex use cases than *Pipes and Filters* does as multiple pipelines are present in the system and as processing an event is allowed to have loosely related consequences (as with the parcel and truck).

Command Query Responsibility Segregation (CQRS)

Client

Write traffic
(new, edited and
deleted records)

Write
backend

Read
backend

Read traffic
(analytics
and views)

OLTP DB OLAP DB

Each record is
present in both
databases

Some data, once
written, may
never be queried
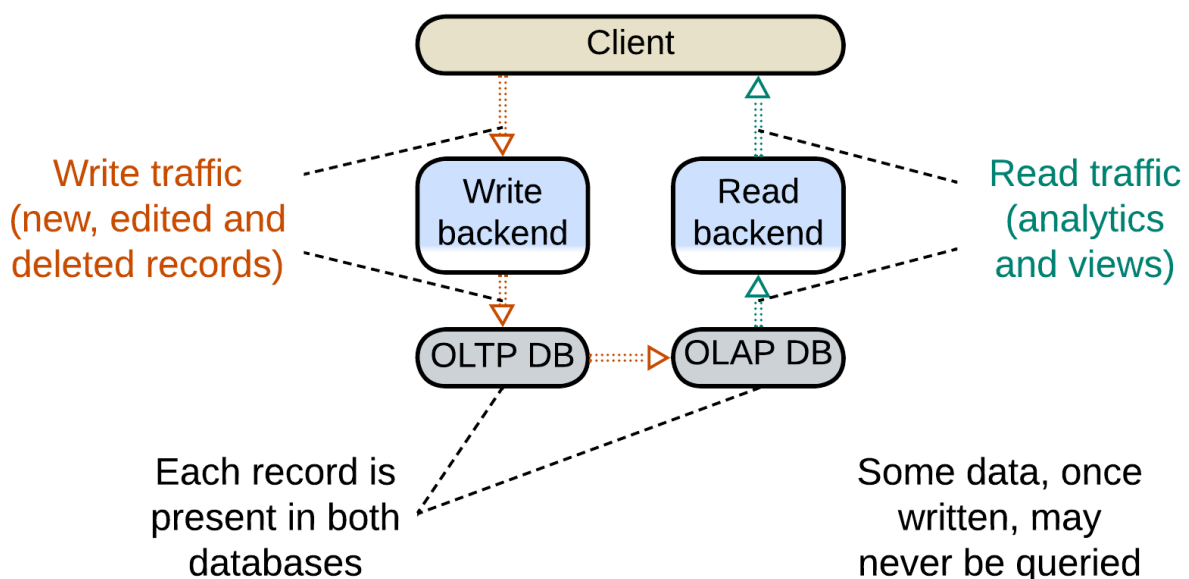
When the data from the events is stored for a future use (as with the aggregation above), the *type* and *temporal order* are ignored but the data *identity* may be retained. A CQRS-based system [MP] separates the paths for write (*command*) and read (*query*) requests, making a kind of data processing pipeline, with the database, which stores the

events for an indeterminate amount of time, in the middle. It is the database that reshuffles the order of the events, as a record it stores may be queried at any time, maybe in a year from its addition – or never at all.

Model-View-Controller (MVC)

The controller receives keyboard and mouse inputs and interprets them as user requests

The view and controller make a platform-specific user interface

Controller   View

Model

The view shows updates and notifications to the users

The model is the platform-agnostic business logic.
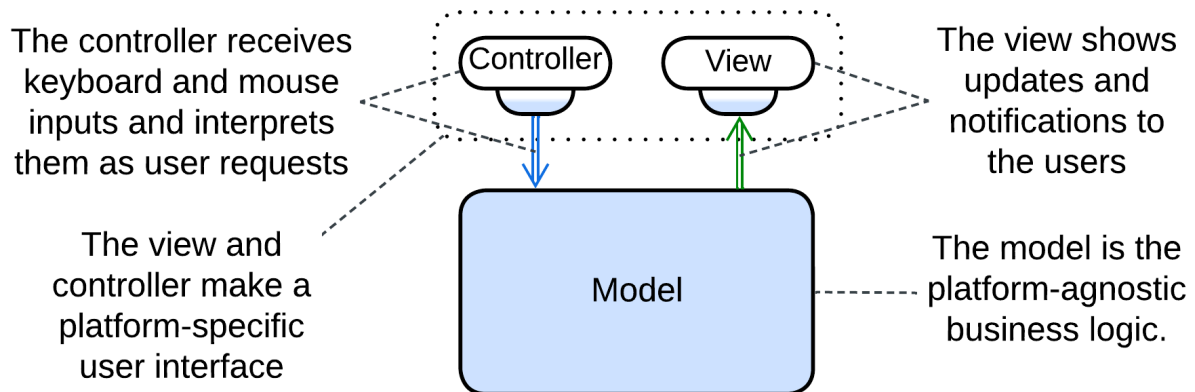
Model-View-Controller [POSA1] completely neglects the *type* and *identity* limitations. It is a coarse-grained pattern where the input source produces many kinds of events that go to the main block which does something and outputs another stream of events of no obvious relation to the input. A mouse click does not necessarily result in a screen redraw, while a redraw may happen on timer with no user actions. In fact, the pattern conjoins two different short pipelines.

Summary

There are four architectures with unidirectional data flow, which is characteristic of pipelines:
- Pipes and Filters,
- Choreographed Event-Driven Architecture (EDA),
- Command (and) Query Responsibility Segregation (CQRS),
- Model-View-Controller (MVC).

The first two, being true pipelines, are built around data processing and transformation, while for the others it is an aspect of implementation – their separation of input from output yields the pairs of streams.

Dependency inversion in architectural patterns

I am no fan of SOLID – to the extent of being unable to remember what those five letters mean – thus I was really surprised to notice that one of its principles – dependency inversion – is quite common with architectural patterns, which means that it is way more generic than OOP it is promoted for.

Let's see how dependency inversion is used on the system level.

Patterns that build around it

A plugin implements an SPI of the core

Several plugins match the same SPI

An adapter translates between a core's SPI and an external API

It isolates the core from its dependency
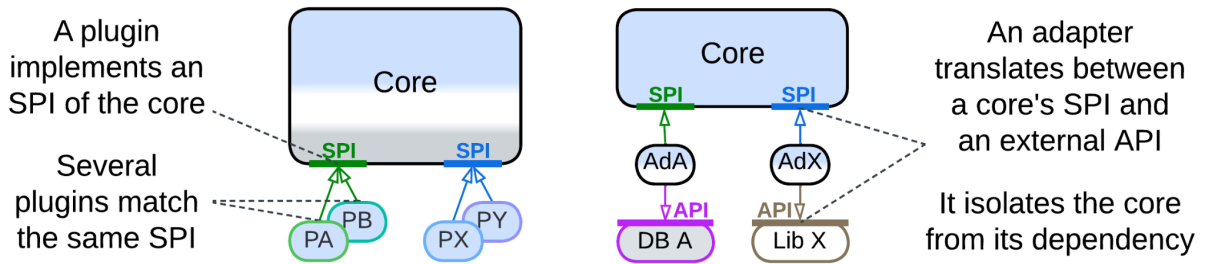
Both *Plugins* and the derived *Hexagonal Architecture* rely on dependency inversion for the same reason – to protect the *core*, which contains the bulk of the code, from variability in the external components that it uses. The *core* operates interfaces (SPIs) which it defines so that it may not care what exactly is behind an interface.

It is the nature of the polymorphic components that distinguishes the patterns:

- *Plugins* allow for small pieces of code, typically contributed by outside developers, to provide customizable parts of the system's algorithms and decision making. Oftentimes the *core* team has no idea of how many diverse plugins will be written for their product.
- *Hexagonal Architecture* is about breaking dependency of the *core* on external libraries or services by employing *adapters*. Each adapter depends both on the core's SPI and on the API of the component which it adapts. As interfaces and contracts vary among vendors and even versions of software, while we want it to be interchangeable, we need adapters to wrap the components to make them look identical to our core. Besides, stub adapters help develop and test the core in isolation.

Patterns that often rely on it

Drivers for similar hardware are polymorphic

Child nodes in a hierarchy may be polymorphic

In Cell-Based Architecture incoming traffic comes through a gateway

Cells call each other through adapters
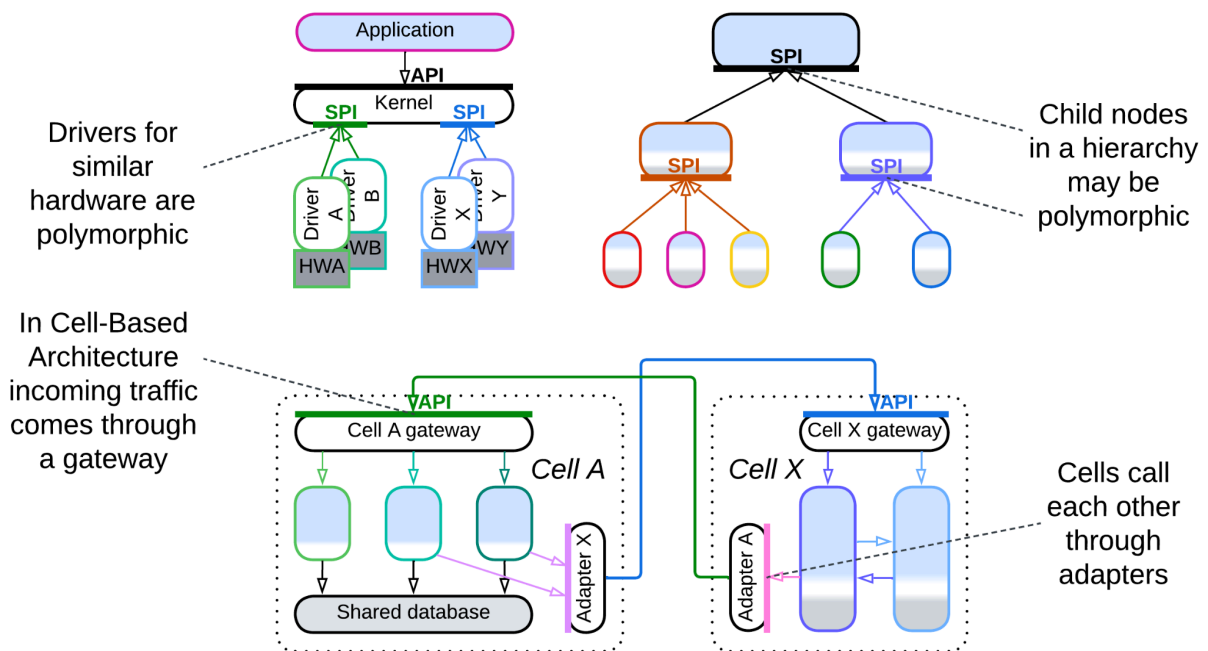
A few more metapatterns tend to apply the approach to earn its benefits, even though dependency inversion is not among their integral features:

- *Microkernel*, yet another metapattern derived from *Plugins*, distributes resources of providers among consumers. Polymorphism is crucial for some of its variants, including *operating system*, but may rarely benefit others, such as *software framework*.
- *Top-Down Hierarchy* spreads responsibility over a tree of components. If the nodes of the tree are polymorphic, they are easier to operate, and we have dependency inversion. However, in practice, a parent node may often be strongly coupled to the types of its children and access them directly.
- In another kind of *Hierarchy*, namely *Cell-Based Architecture* (aka *Services of Services*), each *cell* may employ a *cell gateway* and outbound *adapters* to isolate its business logic from the environment – just like *Hexagonal Architecture* does for its monolithic *core*.

Patterns that may use it
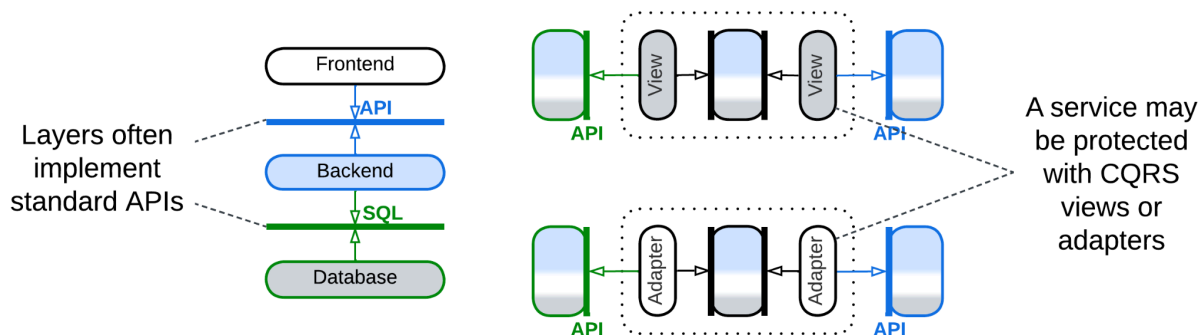
Finally, the basic architectures, *Layers* and *Services*, may resort to something similar to dependency inversion to decouple their constituents:
- We often see a higher layer to depend on and a lower layer to implement a standardized interface, like POSIX or SQL, to achieve interoperability with other implementations (which is yet another wording for polymorphism).
- A service may follow the concept of *Hexagonal Architecture* by using an *anti-corruption layer* [DDD] or *CQRS views* [MP] as *adapters* that protect it from changes in other system components.

Summary

Many architectural patterns employ dependency inversion by adding:
- an *interface* to enable polymorphism of their lower-level components or
- *adapters* to protect a component from changes in its dependencies.

The two approaches apply in different circumstances:
- If you can enforce your rules of the game on the suppliers of the external components, you merely *define an SPI*, expecting the suppliers to implement and obey it.
- If the suppliers are independent and it is your side that adapts to their rules, you should *add adapters* to translate between your lovely SPI and their whimsical APIs.

Indirection in commands and queries

We can solve any problem by introducing an extra level of indirection – states the [fundamental theorem of software engineering](). We will not investigate how this rule drives [deep learning](), at least for now. Instead, let's concentrate our effort on indirection in communication between services.

Each component operates its own *domain model* [DDD] which translates into objects and/or procedures convenient for use in the component's subdomain. However, should a system cover multiple subdomains, the best models for its parts to operate start to mismatch. Furthermore, they are likely to progressively diverge over time as requirements heap up and [the project matures]().

If we want for each module or service to continue with a model that fits its needs, we have to protect it from the influence of models of other components by employing indirection – a translator – between them.

Command (OLTP) systems

Anticorruption Layer   **Open Host Service**   **Orchestrator**
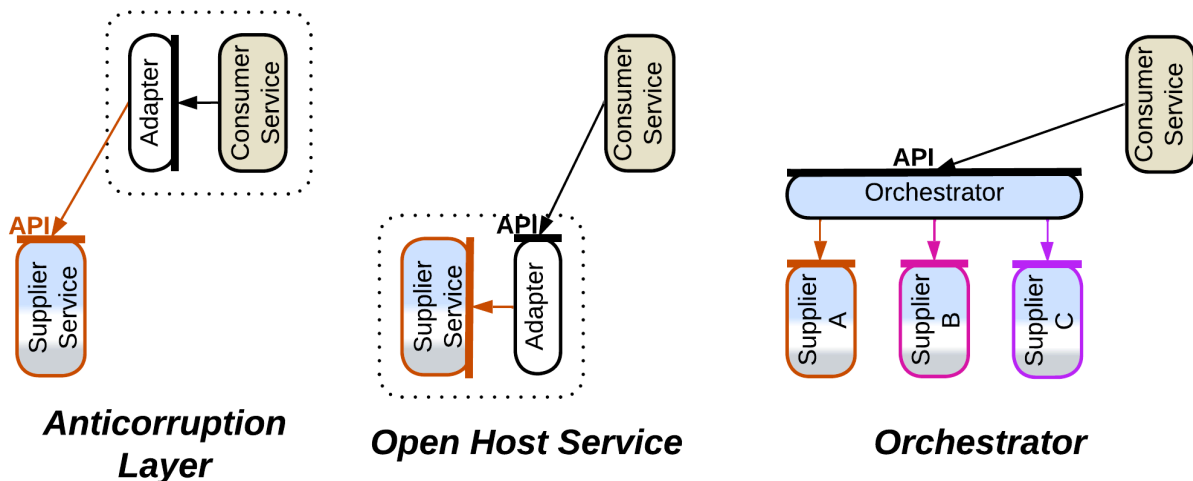
More often than not our system consists of services that command each other: via [RPC]()s, requests or even notifications – no matter how, one component makes a call to action which other(s) should obey.

In such a case we employ an *[adapter]()* between two services or an *[orchestrator]()* when cooperation of several services is needed to execute our command:

- *[Anticorruption Layer]()* [DDD] is an *adapter* on the dependent service's side: as soon as we call another service, we start depending on its interface, while it is in our interest to isolate ourselves from its peculiarities and possible future changes. Thus we should better write and maintain a component to translate the foreign interface, defined in terms of the foreign *domain model*, into terms convenient for use with our code. Even if we subscribe to notifications, we may also want an *adapter* to transform their payload.
- *[Open Host Service]()* [DDD] resides on the other side of the connection – it is an *adapter* that a service provider installs to hide the implementation details of its service from its users. It will typically translate from the provider's *domain model* into a subdomain-agnostic interface suitable for use by services that implement other subdomains.

- *Orchestrator* (*API Composer* [MP], *Process Manager* [EIP, LDDD] or *Saga Orchestrator* [MP]) spreads a command to multiple services, waits for each of them to execute it and cleans up after possible failures. It tends to be more complex than other translators because of the coordination logic involved.

Query (OLAP) systems

CQRS View   **Reporting Database**   **Query Service**
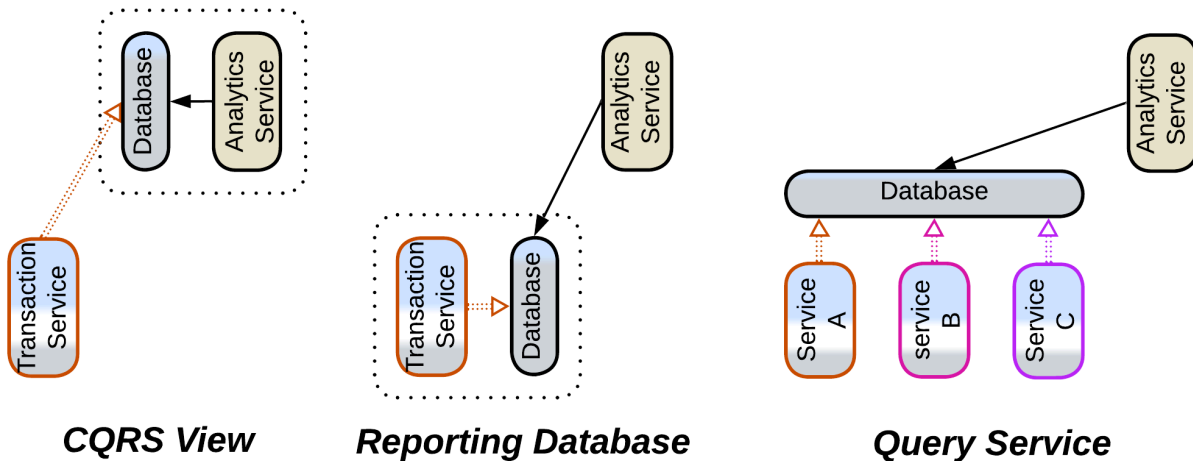
There is often another aspect of communication in a system, namely, information collection and analysis. And it meets a different set of issues which cannot be helped by mere interface translation.

Each service operates and stores data in its own format and schema which matches its *domain model*, as discussed above. When another service needs to analyze the foreign data according to its own *domain model*, it encounters the fact that the foreign format(s) and schema(s) don't allow for efficient processing – in the worst case it would have to read and re-process the entire foreign service's dataset to execute a query.

The solution employs an intermediate database as a translator from the provider's to consumer's preferred data access mode, format and schema:

- *CQRS View* [MP] resides in the data consumer and aggregates the stream of changes published by the data provider. This way the consumer can know whatever it needs about the state of the provider without even making an interservice call.
- *Data Mesh* [LDDD] is about each service exposing a general-use public interface for streaming and/or querying its data. Maintaining one often requires the service to set up an internal *reporting database*.
- *Query Service* [MP] aggregates streams from multiple services to collect their data together, making it available for efficient queries.

Summary

We see that though command-dominated (*operational* or *transactional*) and query-dominated (*analytical*) systems differ in their problems, the architectural solutions which they employ to decouple their component services match perfectly:

- *Anticorruption Layer* or *CQRS View* is used on the consumer's side,
- *Open Host Service* or *Data Mesh*'s *reporting database* is on the provider's side,
- *Orchestrator* or *Query Service* deals with multiple providers,

showing that the principles of software architecture are deeper than the CQRS dichotomy.

Ambiguous pattern names

We've seen a single pattern come under many names, as it happens with *Orchestrator*, and also one name used for multiple topologies, as with *services*, which may orchestrate each other, make a *pipeline* or be components of a *SOA*. On top of that, there are several pattern names that are often believed to be unambiguous while each of them sees conflicting definitions in the books or over the web. Let's explore the last kind, which is the most dangerous both for your understanding of other people and for your time wasted on arguments.

Monolith
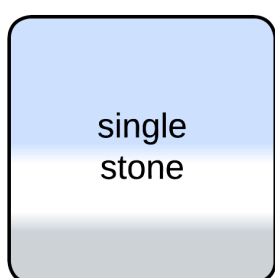
Ye Olde Monolith

New Gen Monolith
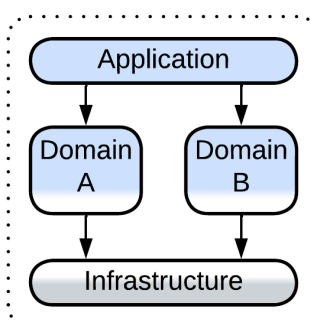
Distributed Monolith
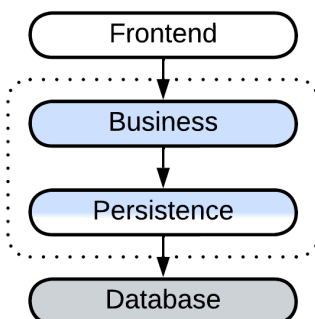
Layered Monolith
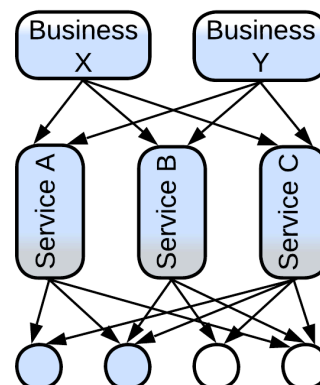
Modular Monolith

The old books, namely [GoF] and [POSA1], described a tightly coupled unstructured system, where anything depends on everything, as *monolithic*, which matched the meaning of the word in Latin – "single stone".

Then something evil happened – I believe that the proponents of *SOA*, backed by the hype and money they had earned from corporations, started labeling any *non-distributed* system as *monolithic*, obviously to contrast the negative connotation of the word to their most progressive design.

It took only a decade for the karma to strike back – when the new generation behind *Microservices* redefined *monolithic* as a single unit of deployment – to call the now obsolescent *SOA* systems *distributed monoliths* [MP] because their services often grew so coupled that they had to be deployed together.

The novel misnomers, *Layered Monolith* [FSA] and *Modular Monolith* [FSA], which denote an application partitioned by abstractness or subdomain, correspondingly, add to the confusion.

Reactor
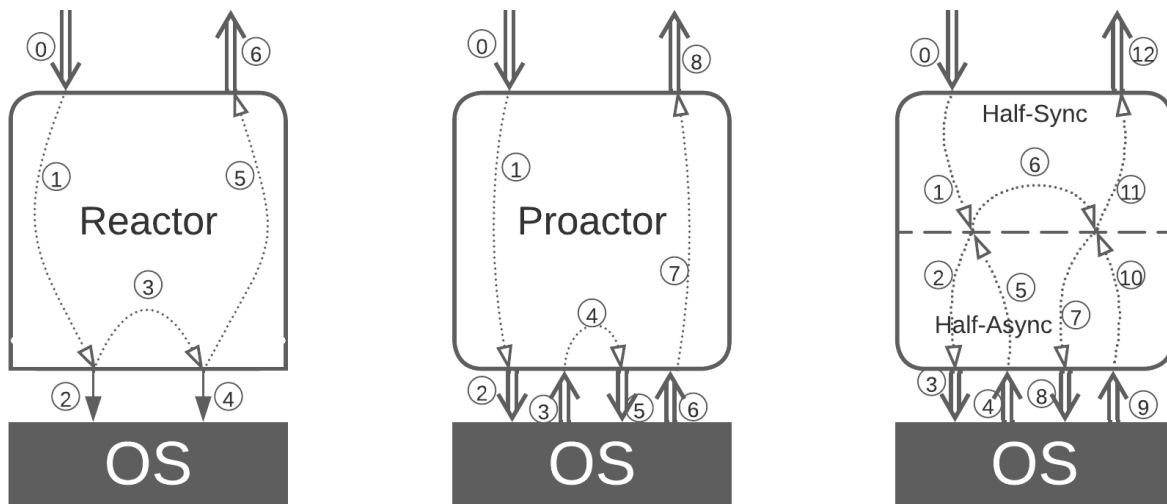
People tend to call any event-driven service *Reactor*. In fact, there are three patterns that describe threading models for an event-handling system:

- In *Reactor* [POSA2] each task runs in its own thread and blocks on any calls outside of the component. That allows for normal *procedural programming* but is resource-consuming and not very responsive or flexible.
- *Proactor* [POSA2] relies on a single thread to serve all the tasks in an interleaved manner, just like an OS uses a CPU core to run several processes. The resulting non-blocking code is fragmented (often called *callback hell*) but can immediately address any incoming event. This is how real-time control systems are implemented.
- *Half-Sync/Half-Async* [POSA2] is what we better know as *coroutines* or *fibers* – there are multiple *Reactor*-like lightweight threads that block on a *Proactor*-like engine which turns synchronous calls from the user code into asynchronous internal events.

In most cases we'll hear of *Proactor* being called *Reactor* – probably because *Reactor* was historically the first and the simplest of the three patterns, and it is similar in name to *reactive programming* which is what we use in *Proactor*.

Microkernel
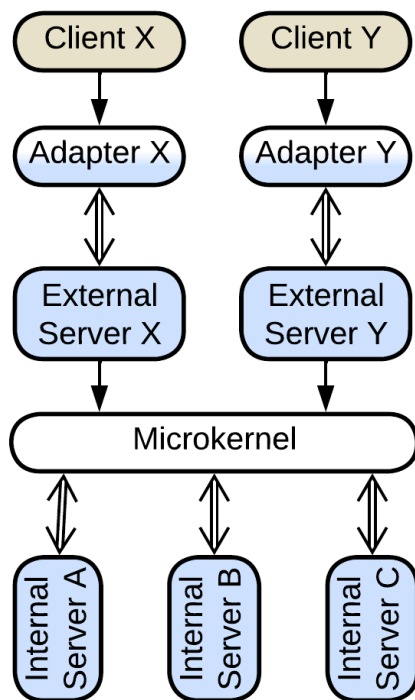
**The Original
Microkernel**
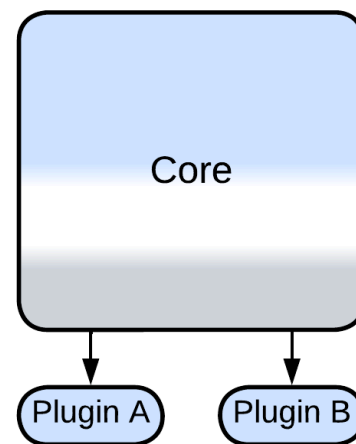
**Microkernel
aka Plugins**

Microkernel is another notable case. The mess goes all the way back to [POSA1] which used operating systems for examples of *Plugin Architecture*. I believe that it was a mismatch:

- An operating system is mainly about sharing resources of producers among consumers, where both producers and consumers may be written by external teams. The *kernel* itself does not feature much logic – its role is to connect the other components together.
- *Plugins*, on the other hand, extend or modify the business logic of the *core* – which alone is the reason for the system to exist and is in no way "*micro-*" as it got the bulk of the system's code. In many such systems *plugins* are utterly optional – which cannot be said of *OS drivers*.

Thus, here we have two architectural patterns of arguably (*Microkernel/Plugins* of [SAP, FSA] omit 3 of 5 components of the original *Microkernel* of [POSA1, POSA4]) similar structure but very different intent and action known under the same name.

Domain Services
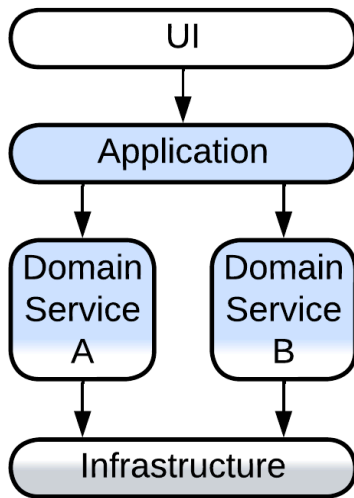
**Domain Services
of DDD**
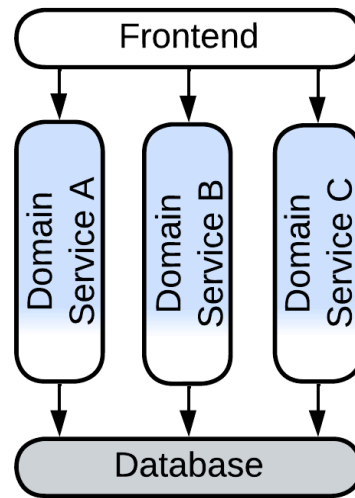
**Domain Services
of FSA**

I was told that *Domain Services* of [FSA] are incorrect – because a *domain service* is always limited to the *domain* layer of [DDD] while those of [FSA] also cover the *application* and, maybe, *infrastructure*.

I believe that both definitions are technically correct, if the difference of the meaning of *domain* is accounted for. In [FSA] *domain* is almost synonymous to a *bounded context* of [DDD], while [DDD] more often uses that word for the name of its middle layer that contains business rules.

Front Controller
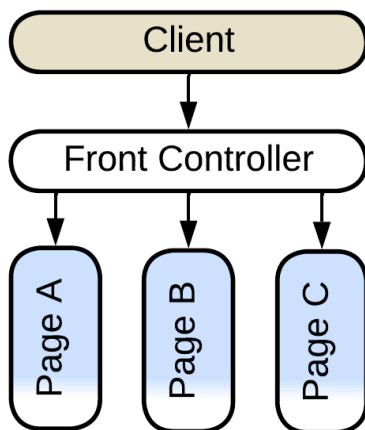
**Front Controller
of PEAA**
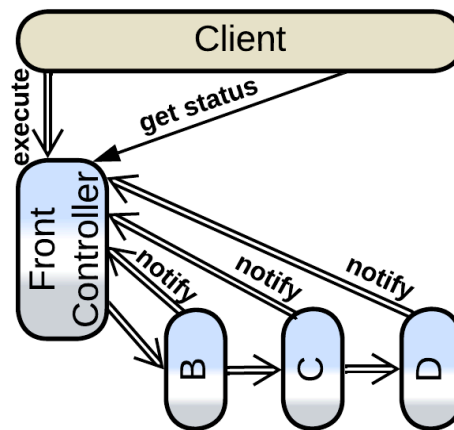
**Front Controller
of SAHP**

[PEAA] defines *Front Controller* as an *MVC* derivative for backend. In this pattern multiple web pages share a request processing component which turns the incoming requests into commands and forwards them to appropriate page classes.

The one of [SAHP] is much more interesting – it describes an *choreographed event-driven architecture* with a *query service* embedded in the first (client-facing) service.

The *front controller* subscribes to notifications from downstream services to know the status of every request it has passed to the *pipeline*.

Cells

WSO2 Cells

Amazon Cells
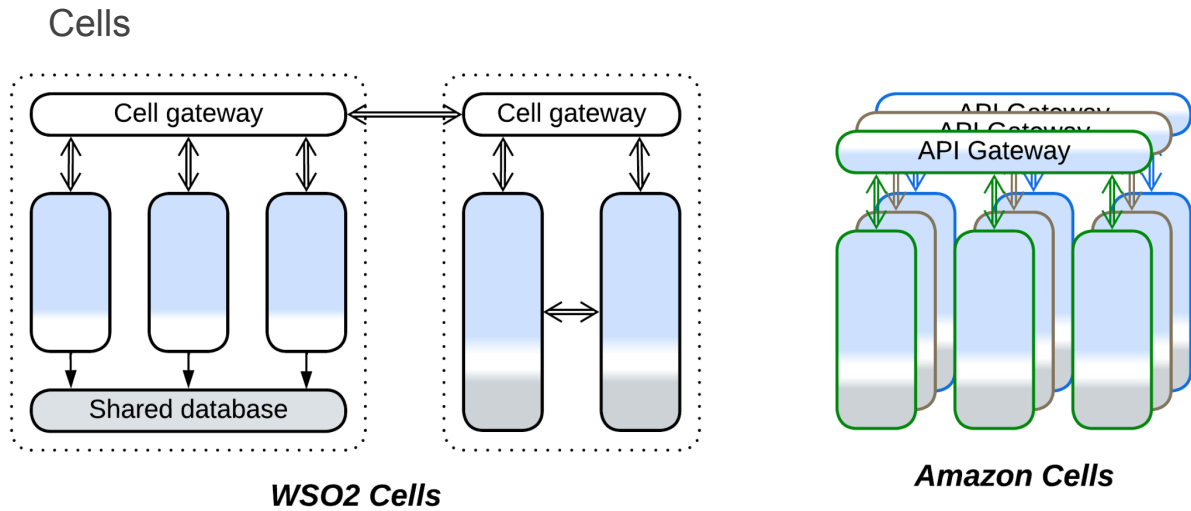
The fresh *Cell-Based Architecture* also got multiple definitions.

- WSO2 wrote about a *cell* as a group of services which is encapsulated from the remaining system by a *gateway and adapters* and often uses a dedicated *middleware* – letting each *cell*, though internally distributed, be treated by other components as a single service. That makes designing and managing a large system a bit simpler by introducing a *hierarchy*.
- Amazon promotes its *cells* as *shards* of the whole system which run in multiple regions. That grants fault tolerance and improves performance as each client has an instance of the system deployed to a nearby datacenter, but does not have much impact on organization and complexity of the code.

The case looks like Amazon's hijacking and redefining a popular emerging technology, though I may be wrong about that as I did not investigate the history of the term.

Nanoservices

Nanoservices as API Layer

Nanoservices as Pipeline

Nanoservices as Actors

Nanoservices as SOA

Nanoservices as Space-Based Architecture
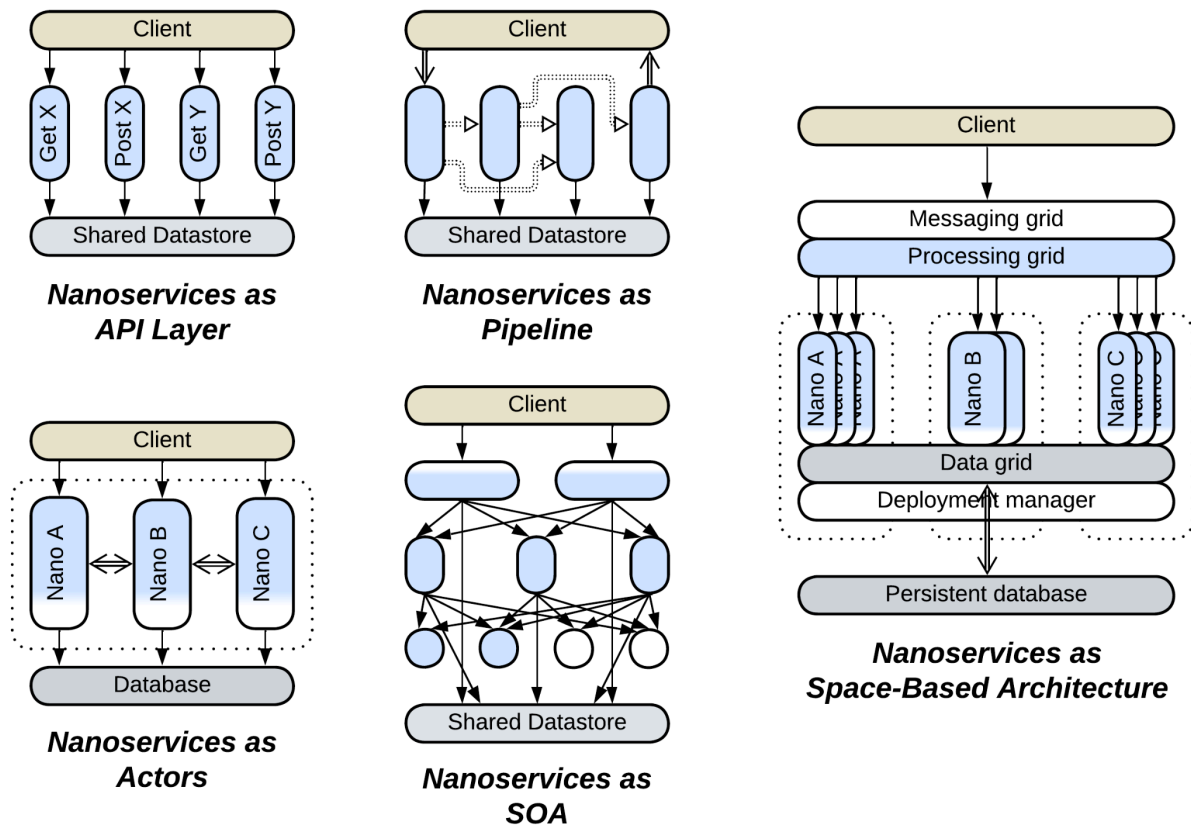
Nanoservices is another emerging technology, and it seems to have never been strictly defined. Most sources agree that a *nanoservice* is a cloud-based function (*FaaS*), similar to a *service* with a single API method but, just as with the old good *services*, they differ in the ways they use the technology:

- Diego Zanon in *Building Serverless Web Applications* proposes a single layer of nanoservices, each implementing a method of the system's public API, to be used as a thin backend.
- Here we have *nanoservices* built into a *pipeline*, similar to *Choreographed Event-Driven Architecture* [FSA].
- Another article proposes to (re)use them in *SOA* style.

Nonetheless, there are a couple of sources which call a *nanoservice* something totally different:

- There is a concept of *nanoservice* as a module that can run both as a separate service and as a part of a binary – allowing for the team to choose if they want their system to execute as a single process or become distributed. *Nano-* is because an in-process module is more lightweight than a *microservice*. This idea resembles *Modular Monolith* [FSA] and actor frameworks.
- And here we got something akin to *Space-Based Architecture* but it is also called *Nanoservices* – as the proposed framework makes them so easy to create that programmers tend to write many smaller *nanoservices* instead of a single *microservice*.

In my opinion, the disarray happened because the notion of "making *smaller* microservices" got hyped but was never adopted widely enough to become an industry standard, therefore everybody follows their own vision about what *smaller* means.

Summary

A few names of architectural patterns cause confusion as the meaning of each of them changes from source to source. The current book aims at identifying such issues and building a cohesive understanding of software and system architecture, similar to the *ubiquitous language* of [DDD].

Architecture and product life cycle

In my practice, architecture changes during a product's lifetime. For R&D, when there is none with a relevant experience on the team, it starts small, gradually gains flexibility through fragmentation, grows and restructures itself according to the ever-changing domain knowledge and business requirements, then solidifies as the project matures, and dies to performance optimizations and loss of experience as main developers leave. More mundane projects may omit the first stages (as no research needs to be done), and oftentimes a project is canceled way before its architecture dies under its own weight. Anyway, let's observe the full life cycle.

Infancy (proof of concept) – monolith

Monolith
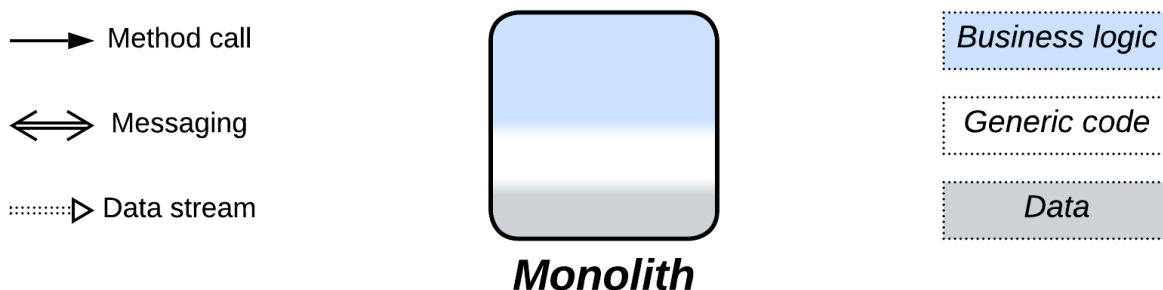
A project in an unknown domain starts timid and small, likely as a proof of concept. You need to write quickly to check your ideas about how the domain works without investing much time – as you may be wrong here or there making you rethink and rewrite.

Childhood (prototype) – layers

Layers   **Hexagonal Architecture**
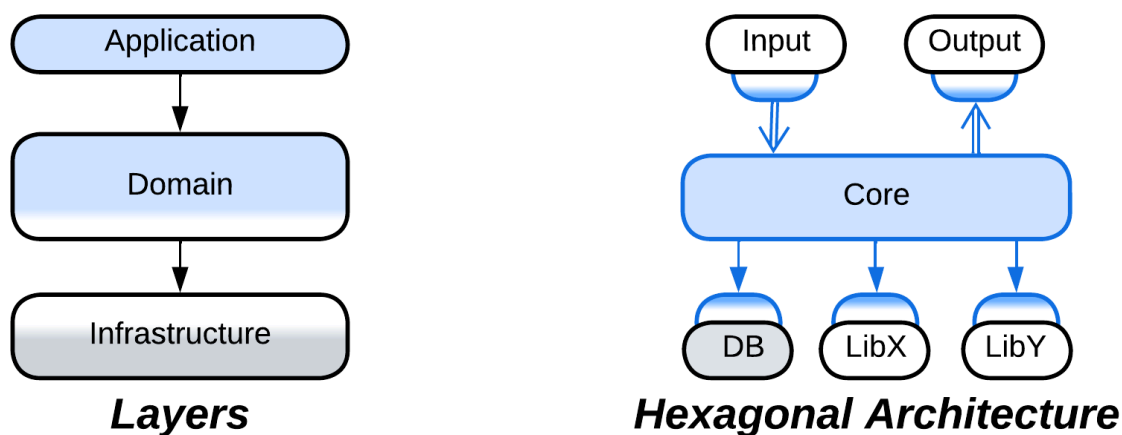
When you have the thing working, you may start reflecting on the rules and the code you wrote. What belongs where, what is changeable, which tests will you need? At this point you clearly see the levels of abstractness: the high-level *application* (integration, orchestration) logic, the lower-level *domain* (business) rules and the generic *infrastructure* [DDD]. Now that you better know the whats and the hows, you divide the (either old or rewritten from scratch) code into *layers* or *hexagonal architecture* to achieve both structure and flexibility, still without much development overhead caused by interfaces.

Youth (development of features) – fragmented architectures

Layered Services   **Services with an Orchestrator**   **Hierarchy**
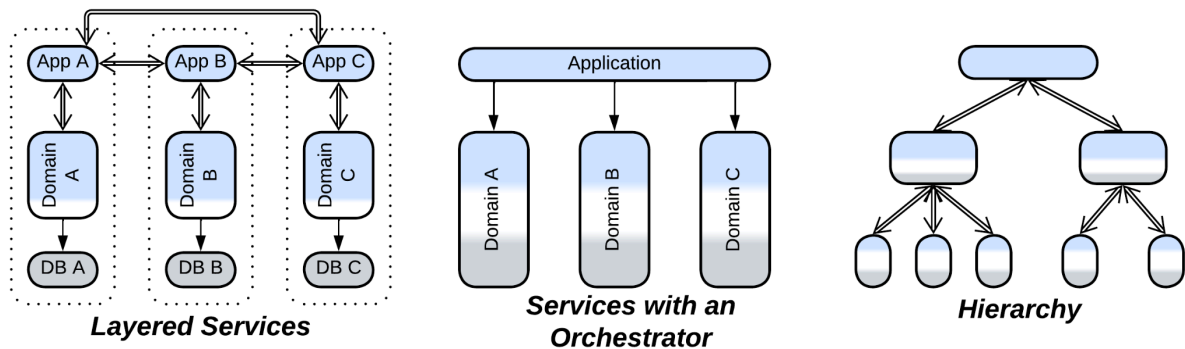
As you earn domain experience, you start discerning subdomains (sort of *bounded contexts* [DDD]) and isolating them to reduce the complexity of your code. The layered structure turns to a system of subdomain-dedicated components: modules, services, device drivers – whatever you used to name them in your career. The actual architecture follows the structure of the domain, with *layered services*, *services with orchestrator* and *hierarchy* among the common examples. The fragmentation of the system brings in development by multiple teams with diverse technologies and styles, reduces ripple effect of changes and helps testability. However, use cases for the system as a whole become harder to understand and fix – if only because they traverse the parts of the code owned by multiple teams – which is not extremely bad given you have enough humanpower.

Adulthood (production) – ad-hoc composition

Layered Services   **Pragmatic Architecture**
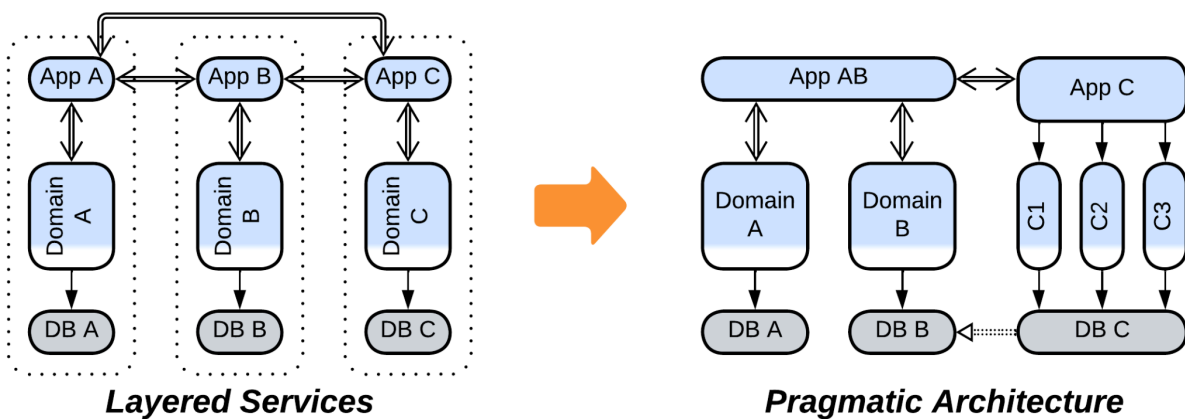
As the product enters the market, the development will likely slow down with more attention given to details and user experience. Some (likely, the most active) people are going to get bored and leave the project, while your understanding of the domain changes again – based on the user experience and real-life business needs [DDD]. You may find that some of the components which you designed as independent become strongly coupled – and you are lucky if they are small enough to be merged together – this is where the fragmentation pays off. Other parts of the system may overgrow the comfort zone of programmers and need to be subdivided. The architecture becomes asymmetrical and pragmatic.

Old age (support) – back to layers
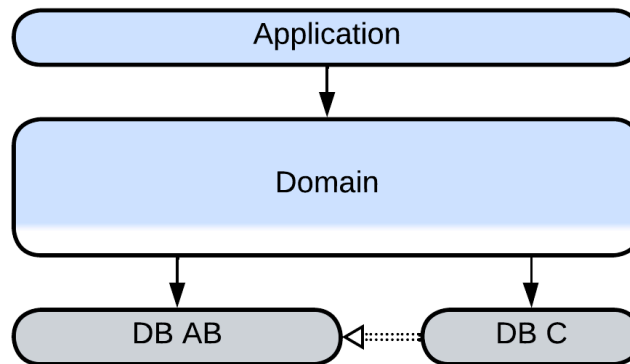
Back to Layers

When the active development ceases, you lose even more people and funding as you drift into the support phase. You are unlikely to retain your best programmers – actually, you'll get novices or an outsourced team instead. They will struggle to retain the structure of the system – with its collection of hacks from the previous years – against progressively more weird requests from the management and customers – as all their normal desires have already been satisfied. That will cause many more hacks to be added – and components merged to allow for the hacks to land – bringing the architecture back to *layers*, though heavily oversized this time.

Death (the ultimate release) – monolith
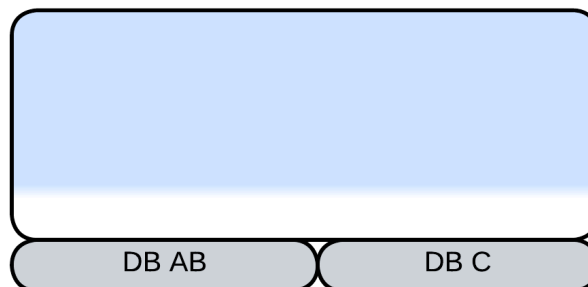
Down to Monolith

If the project is allowed to die, it may still have a chance of a final release which aims at improving performance and leaving a golden standard for the generations of users to come. Heavy optimizations will likely require merging the layers to avoid all kinds of communication overhead, bringing you back to *monolith*.

So it goes

Even though I observed the cycle of architecture expanding and collapsing in embedded software, I believe the forces apply to most kinds of systems. First you need to go fast, and overly generic interfaces are a burden. Then you need that extra flexibility they provide to reserve space for future design changes. And as the flow of changes ceases, you may optimize the flexibility away to make programming easier and the code smaller and faster. However, the rule is not always applicable: a distributed system will oppose compacting it if it was written in multiple programming languages or needs several specialized components for proper operation.

Going back in time

It happens that you need to step back through the life cycle – for example, when the domain itself changes drastically: a new standard emerges, or the management decides that your application for washing machines fits coffee machines pretty well, as they are doing the same three basic things: heating water, adding powder and stirring – but you have never wrote for coffee machines before, thus you are back to the R&D phase.

In such cases it may be easier to rewrite affected modules from scratch than try to rejuvenate and refit the old code as you keep your years of experience and what was originally implemented as an improvised hack will be accounted for in the redesigned architecture. This means that each time a module is rewritten adds to its longevity as its architecture more closely fits the domain and requires fewer hacks (which are inflexible and confusing by definition) to get to production.

Real-world inspirations for architectural patterns

As architectural patterns are usually technology-independent, they must be shaped mostly by foundational principles of software engineering. And as the same principles are likely at work at every level of a software system, we may expect similar structures to appear on many levels of software, given similar circumstances – which is not always attainable, for the system-wide scope (which means that there are multiple clients and libraries) and distributed nature (which deals with faults of individual components) of many patterns of system architecture don't have direct counterparts in single-process software. Thus we can expect to observe the fractal nature of more generic patterns, while the narrowly specialized ones are present at only one or two scopes of software design.

Another thought to consider is that it's not in human nature to invent something new – we are much more adept in imitating and combining whatever we see around. That is why it's so hard to find a genuine xenopsychology in literature or movies – to the extent that the eponymous Alien is just an overgrown [parasitoid wasp](). Hence there is another pathway to course – looking for the patterns which we know from software engineering in the world around us, as the authors of [[POSA2]()] did decades ago.

Let's go!

Basic metapatterns

The basic patterns lay the foundation for any system by paving ways to *divide* it into components to *conquer* its [complexity](). We are going to observe them all around:

Monolith
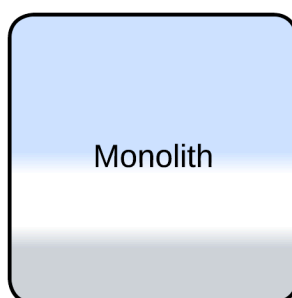
Monolith is a (sub)system the internals of which we prefer not to see	Monolith	It may lack an internal structure or its components may be coupled

[Monolith]() means encapsulation – we use one without looking inside:
- You interact with your dog (or your smartphone) through their interface without thinking of their internals.
- A function exposes its name, arguments and, probably, some comments. The implementation is hidden from its users.
- An object has a list of public methods.
- A module or a library exports several functions for use by its clients.
- A program is configured through its command line parameters and managed through its CLI. We don't care how the Linux utilities (like *top* or *cat*) work – we just run them.
- A distributed system hides behind a web page in your browser – and you never imagine its complexity, unless you have worked on something of a kind.

Shards

Shards are multiple instances of a subsystem
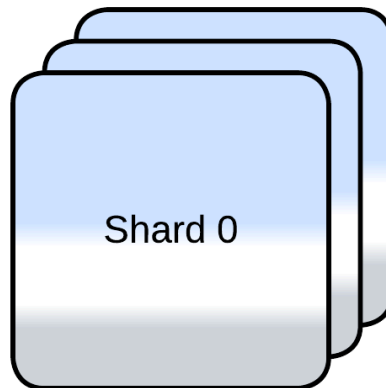
Shard 0

A shard may be stateless or stateful

Shards is having multiple instances of something, which often differ in their data:
- A company employs many programmers to accelerate development of its projects.
- Wearing two cell phones from different operators fits the pattern as well.
- This is how they make modern processors more powerful: by adding more cores, not by running them faster.
- Objects in OOP are the perfect example of having multiple instances that vary in their data.
- Running several shells in Linux is a kind of sharding.
- A client application of a multi-user online game is a shard.

Layers

Layers divide the system by the level of abstractness

Application ----- High-level scenarios

Domain layer ----- Business rules

Database ----- Infrastructure
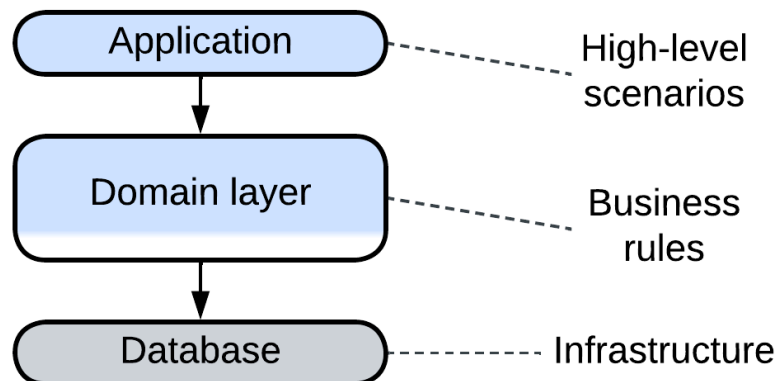
Layers is the separation of responsibilities between external and internal components:
- In winter we wear soft clothes on our body, a warm sweater over them and a wind-proof jacket as the external layer.
- An object comprises high-level public methods, low-level privates and data.
- An OS has a UI which runs over user-space software over an OS kernel over device drivers over the hardware.
- Your web browser executes a frontend which communicates to a backend which uses a database.

Services

Each service implements a subdomain
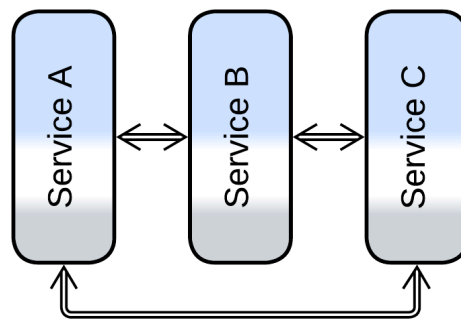
The subdomains should be loosely coupled

Services boil down to composition and separation of concerns:
- We have legs, arms and other specialized members.
- A gadget contains specialized chips for the activities it supports.
- [GoF] advocates for an object to incorporate smaller objects (composition over inheritance).
- Applications often delegate parts of their logic to specialized modules or libraries.
- An OS dedicates a driver to each piece of hardware installed. Moreover, it provides many tools to its users – instead of tackling all the user needs in the kernel.
- [DDD] describes the way to subdivide a large system into loosely coupled components.

Pipeline

The system processes data in a sequence of steps
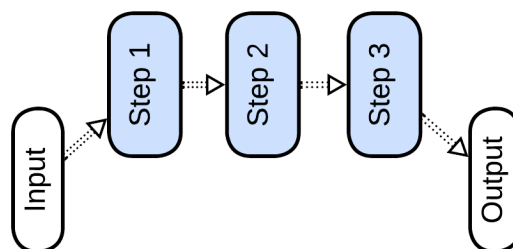
Each step knows nothing of its neighbors

Pipeline is about stepwise transformation of data:
- The pattern got its name from real-world plumbworks.
- You'll see similar arrangements in cellular metabolism.
- It is the basis for functional programming.
- Linux command line tools are often skillfully composed into pipelines.
- Hardware is full of pipelines: from CPU and GPU to audio and video processing.
- Finally, a UI wizard passes its users through a series of screens.

Extension metapatterns

An extension pattern encapsulates one or two aspects of the system's implementation. It may appear only on design levels that have those aspects:

Middleware

The middleware provides communication for the services
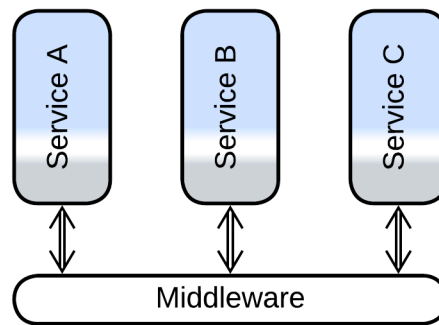
It may also manage their instances

Middleware abstracts scaling and/or intercommunication:
- The network of post offices is a middleware – you push the letter into a mailbox – and it automagically appears at its destination's door.
- A bus depot may mean a bus garage which deploys as many buses as needed to service the traffic or a bus station where people come to have a ride, disregarding the exact bus model they'll take.
- Hardware is full of other kind of buses that unify means of communication.
- TCP and UDP sockets hide the details of the underlying network.
- An actor framework allows an actor to address another one without knowing where it is deployed.

Shared Repository

The shared repository owns the system's data
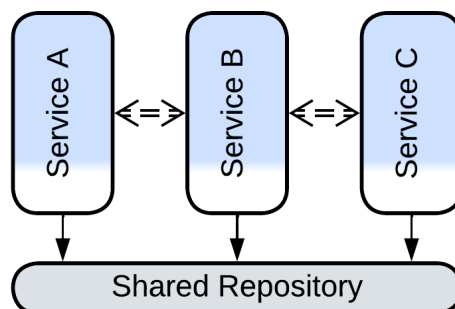
The services communicate directly or through the repository

Shared Repository provides data storage and/or data change notifications:
- Everybody in the room may use the blackboard to express and exchange their ideas.
- An internet forum works in a similar way – people post their arguments there for others to see them and get notified of answers.
- RAM and CPU caches are kinds of shared repositories. CPU caches are kept synchronized through notifications.
- [GoF] *Observer* is about getting notified when a shared object changes.
- Several services may share a database.

Proxy

A proxy stands between a (sub)system and its clients
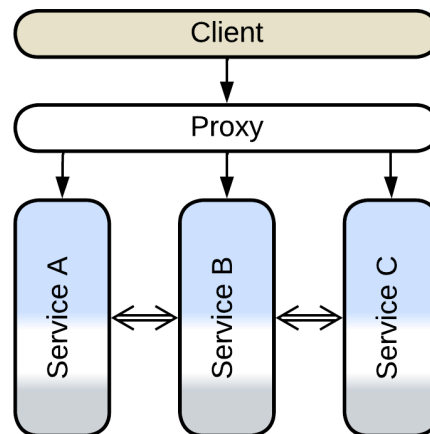
It implements a generic aspect of the system's behavior

Proxy isolates a system from its environment by translating between the internal and external protocols and/or implementing generic aspects of communication:
- You may need a translator to understand foreign people or a secretary to deal with routine tasks. A local guide combines both roles.
- Hardware (plugs) and software (frameworks) is full of adapters.
- Your Wi-Fi router is a proxy between your laptop and the Internet.
- A compiler is a kind of proxy between source code and bytecode.

Orchestrator

The orchestrator runs high-level scenarios by using the services
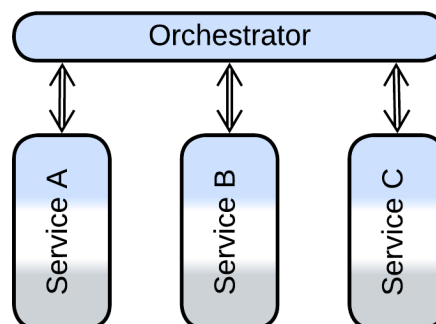
It also keeps the data of the services consistent

Orchestrator integrates several components by implementing high-level use cases and/or keeping them in sync:
- A taxi driver orchestrates the car's internal components.
- [GoF] *Facade* provides a high-level interface for a system while [GoF] *Mediator* manages a system by spreading changes initiated by the system's components.
- A linker composes a working program out of disjunct modules.

Fragmented metapatterns

A fragmented pattern uses small specialized components to approach a case which is hard to resolve with more generic means. The high degree of specialization means even fewer examples:

Polyglot Persistence
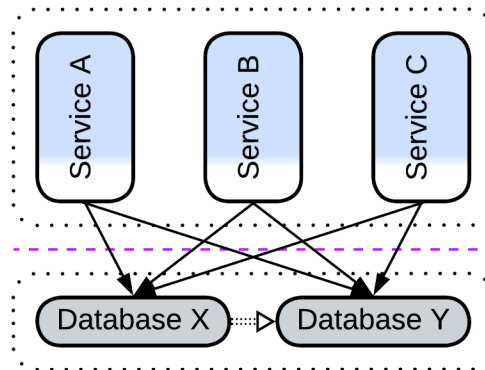
The system earns benefits of multiple database technologies

The databases store separate subsets or derived views of the data

Polyglot Persistence is about having multiple containers for data:
- A warehouse or a cargo ship has dedicated storage areas with extra facilities for combustible, toxic and frozen goods.
- A computer got CPU caches, RAM, flash and hard drives for temporary or permanent data storage.
- There is the map, list and array – each with its pros and cons. A large class would often use two or three kinds of containers – not for no reason.

Backends for Frontends

Each BFF is dedicated to its kind of client
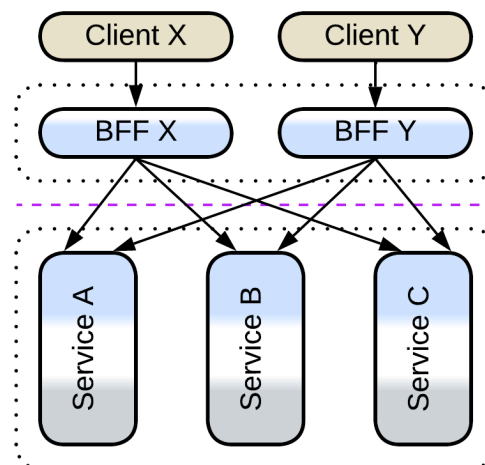
Each BFF orchestrates all the services

Backends for Frontends is treating clients individually:
- A bank is likely to reserve a couple of employees to serve rich clients.
- A Wi-Fi router has many management interfaces: web, mobile application, CLI and probably TR-069.
- A multiplayer game may provide desktop and mobile client applications.

Service-Oriented Architecture

Each task
orchestrates
entities

SOA is 3 or 4
layers of
services

Entities use
libraries and
utilities
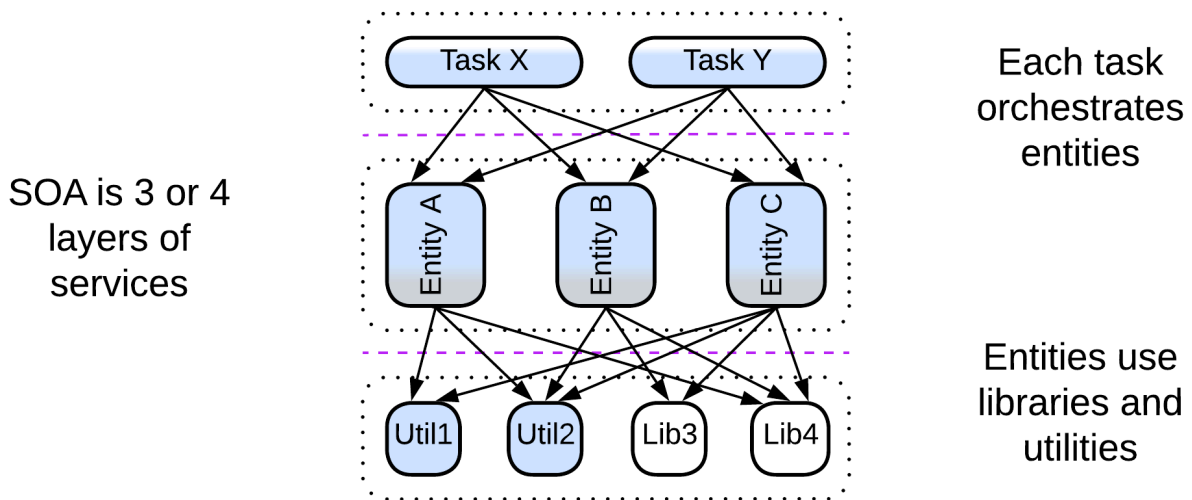
SOA applies OOP techniques, including component reuse, to deal with complex systems:

- That's what you have inside your car. Many of its internals rely on the car's battery for power supply – instead of having a small battery installed inside every component.
- Cities are built in the same way – schools, markets and railways serve multiple houses.
- It's the same with user space of operating systems: there is a shared UI framework which interfaces as-many-as-needed applications, each of which calls into shared libraries (DLLs).

Hierarchy

Each root
orchestrates
its children

Hierarchy
is a tree of
components

Child nodes
may be
specialised or
polymorpfic
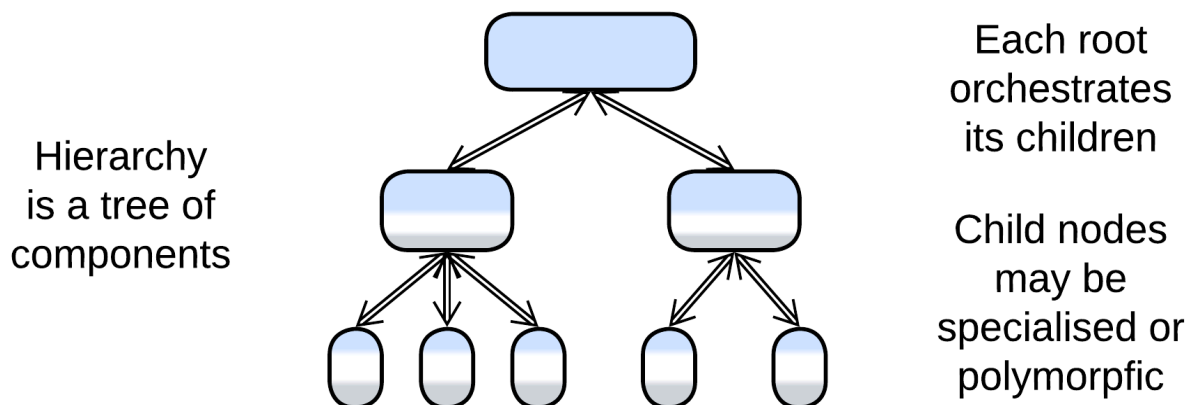
Hierarchy distributes system's complexity over multiple levels:

- This is how large companies and the army are managed.
- Large projects contain services which contain modules which contain classes which contain methods.

Implementation metapatterns

An implementation pattern exposes peculiar internal arrangements of a component. Such patterns are deeply specialized:

Plugins

Plugins allow for customization of the system's functionality

Addons modify input and output

Plugins implement aspects of the system's behavior
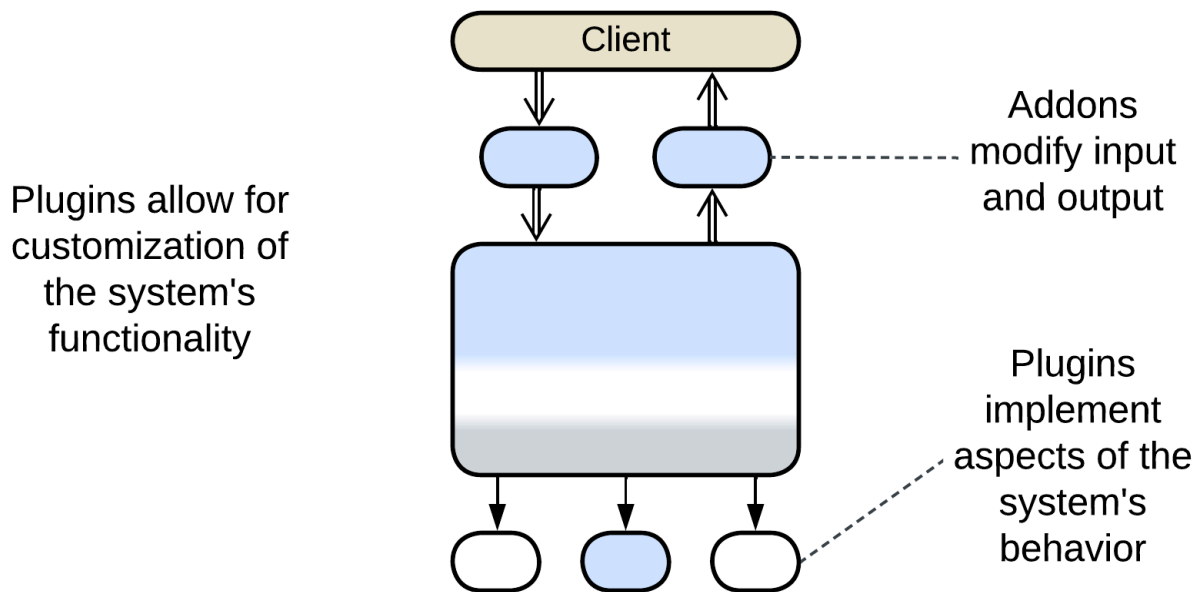
Plugins make the component's behavior flexible through delegating its parts to small external additions:
- This is how we use tools for our work – a man becomes a digger when given a shovel.
- [GoF] *Strategy* is the thing.

Hexagonal Architecture

Hexagonal Architecture assigns an adapter to each external component

The goal is to make the business logic independent
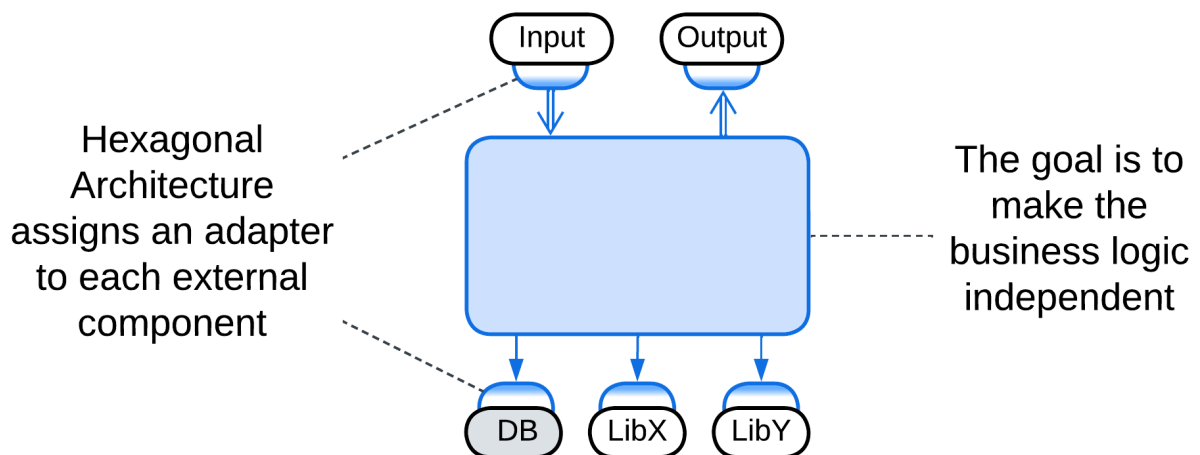
Hexagonal Architecture protects the internals of a system from its environment:
- A drill or a screwdriver have exchangeable bits.
- *OS Abstraction Layer* and *Hardware Abstraction Layer* in embedded systems or *Anti-Corruption Layer* in [DDD] are all about that.

Microkernel

Microkernel
shares resources
of providers
among
applications

App X   App Y

Microkernel

Provider   Provider   Provider

It is a middleware
to applications ...

... and an
orchestrator to
resource providers
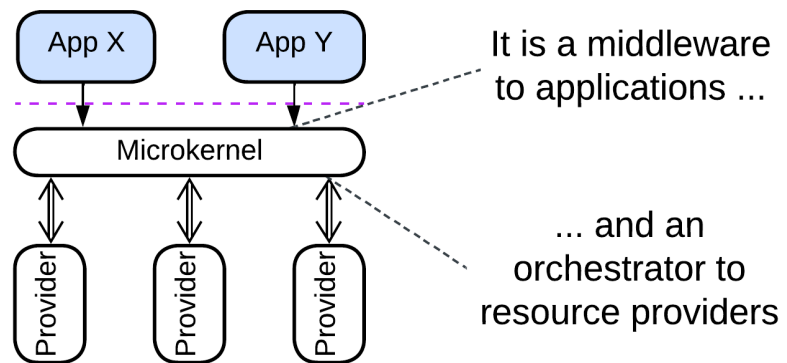
Microkernel shares the goods of resource providers among resource users:
- It's like a bank that takes money from the rich to distribute them among the poor.
- This is what an OS is for. Its scheduler shares CPU, the memory subsystem shares RAM while the device drivers provide access to the periphery.
- Cloud services are based on sharing computation resources among clients.

Mesh

App A   App B   App C

Mesh comprises
interconnected
shards

Each shard
may carry an
application

The mesh
makes a
middleware
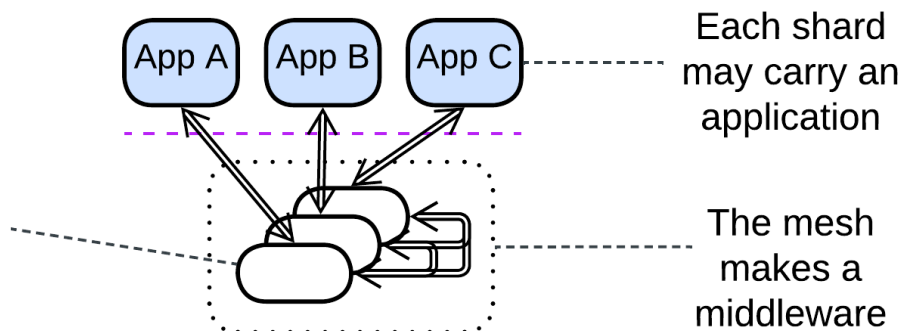
Mesh is like grassroots movements – self-organization and survival through redundancy:
- Ants and bees are small, autonomous and efficient. Their strength comes from their numbers.
- Road networks and power grids don't collapse if several of their components are damaged as they are highly redundant.
- Torrents, mobile communications and the Internet infrastructure are known for their high survivability.

Summary

Architectural patterns have parallels in the natural world, our society and/or different levels of computer hardware and software. Learning of them helps us to feel the driving forces behind the patterns and become more flexible and creative in using the patterns we know and devising new ones.

Part 7. Appendices

Appendix A. Acknowledgements.

I remember Avraham Fraenkel of DSPG who showed me what a true manager is like.

Thanks to Alexey Nikitin and Maxim Medvedev of Keenetic who let me design a subsystem from scratch and see how it fared through years of heavy changes.

It was from discussion with Sergey Ignatchenko aka IT Hare that I got the difference between control and data processing systems. Mark Richards read my previous series of articles and encouraged me to press on with the classification of patterns. Kiarash Irandoust noticed my articles on Medium and invited me to publish them in ITNEXT where many people could see them.

Thanks to Max Grom and other participants of the Ukrainian software architecture chat for hours of heated discussions about the meaning of patterns, which resulted in several analytical chapters of the book and for guiding me through the intricacies of DDD.

I must thank my mother and our neighbor Halyna for helping me throughout the war.

I want to thank everybody who prayed for me.

This book was made possible by many people who sacrificed their happy years, their limbs and their lives to protect those who stayed behind.

Appendix B. Books referenced.

DDD – Domain-Driven Design: Tackling Complexity in the Heart of Software. *Eric Evans. Addison-Wesley (2003).* (Most of these patterns are also well-described in [LDDD])

DDIA – Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. *Martin Kleppmann. O'Reilly Media, Inc. (2017).*

DDS – Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. *Brendan Burns. O'Reilly Media, Inc. (2018).*

EIP – Enterprise Integration Patterns. *Gregor Hohpe and Bobby Woolf. Addison-Wesley (2003).*

FSA – Fundamentals of Software Architecture: An Engineering Approach. *Mark Richards and Neal Ford. O'Reilly Media, Inc. (2020).*

GoF – Design Patterns: Elements of Reusable Object-Oriented Software. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley (1994).*

LDDD – Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. *Vlad Khononov. O'Reilly Media, Inc. (2021).* (Duplicates [DDD] thus I marked as [LDDD] only patterns not covered by [DDD])

MP – Microservices Patterns: With Examples in Java. *Chris Richardson. Manning Publications (2018).*

PEAA – Patterns of Enterprise Application Architecture. *Martin Fowler. Addison-Wesley Professional (2002).*

POSA1 – Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. John Wiley & Sons, Inc. (1996).*

POSA2 – Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. *Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, Inc. (2000).*

POSA3 – Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. *Michael Kircher, Prashant Jain. John Wiley & Sons, Inc. (2004).*

POSA4 – Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. John Wiley & Sons, Ltd. (2007).*

POSA5 – Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. *Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. John Wiley & Sons, Ltd. (2007).*

SAHP – Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. *Neal Ford, Mark Richards, Pramod Sadalage, and Zhamak Dehghani. O'Reilly Media, Inc. (2021).*

SAP – Software Architecture Patterns. *Mark Richards. O'Reilly Media, Inc. (2015).* (All of the architectures referenced here are in [FSA] as well, but [SAP] is free)

Appendix C. Copyright.

material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

10. <u>Sui Generis Database Rights</u> means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

11. <u>You</u> means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

1. **License grant** .
    1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:
        1. reproduce and Share the Licensed Material, in whole or in part; and
        2. produce, reproduce, and Share Adapted Material.
    2. **Exceptions and Limitations** . For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.
    3. **Term** . The term of this Public License is specified in Section 6(a) .
    4. **Media and formats; technical modifications allowed** . The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.
    5. <u>Downstream recipients</u> .
        1. <u>Offer from the Licensor – Licensed Material</u> . Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.
        2. <u>No downstream restrictions</u> . You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.
    6. <u>No endorsement</u> . Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i) .
2. **Other rights** .

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
2. Patent and trademark rights are not licensed under this Public License.
3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

1. **Attribution** .
    1. If You Share the Licensed Material (including in modified form), You must:
        1. retain the following if it is supplied by the Licensor with the Licensed Material:
            1. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);
            2. a copyright notice;
            3. a notice that refers to this Public License;
            4. a notice that refers to the disclaimer of warranties;
            5. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;
        2. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and
        3. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.
    2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.
    3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.
    4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

1. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;
2. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and
3. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

1. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.
2. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.
3. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

1. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.
2. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or
2. upon express reinstatement by the Licensor.
3. For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.
4. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.
5. Sections 1 , 5 , 6 , 7 , and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

1. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.
2. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

1. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.
2. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.
3. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.
4. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Appendix D. Disclaimer.

Appendix E. Evolutions.

This appendix details a dozen or two evolutions of metapatterns to show how they connect together. They probably have some practical value through listing prerequisites, benefits and drawbacks, but I am not sure of how many readers get through them without being bored to death. The metapatterns in the main parts of the book include abridged versions of the sections below.

Duplicate and similar evolutions are omitted, and I did not write the ones for fragmented metapatterns as you should be able to infer them on your own after reading the book. Furthermore, I don't know of any evolutions of implementation metapatterns for some reason.

Monolith: to Shards

One of the main drawbacks of monolithic architecture is its lack of scalability – a single running instance of your system may not be enough to serve all its clients no matter how much resources you add in. If that is the case, you should consider *Shards* – *multiple instances* of a monolith. There are following options:

- Self-managed *shards* – each instance owns a part of the system's data and may communicate with all the other instances (forming a *mesh*).
- *Shards* with a *load balancer* – each instance owns a part of the system's data, with an external component to select a shard for a client.
- A *pool of stateless instances* with a *load balancer* and a *shared database* – any instance can process any request, but the database limits the throughput.
- A *stateful instance per client* with an external persistent storage – each instance owns the data related to its client and runs in a virtual environment (i.e. web browser or an actor framework).

Implement a mesh of self-managed shards

Each client should know their shard

Splitting to stateful (true) shards scales the system but takes much work and requires the data of users to be mostly independent
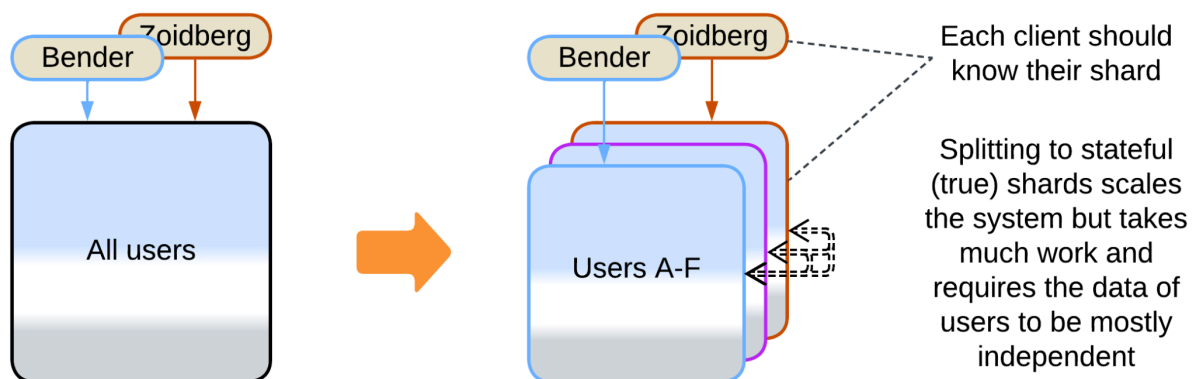
Patterns: Shards, Mesh.
Goal: scale a low-latency application with weakly coupled data.
Prerequisite: the application's data can be split into semi-independent parts.

It is possible to run several instances of an application (*shards*), with each instance owning a part of the data. For example, a chat may deploy 16 servers, each responsible for a subset of users with specific last 4 bits of the user name's hash. However, some scenarios

(renaming a user, adding a contact) may require the shards to intercommunicate. And the more coupled the shards become, the more complex *mesh engine* is required to support their interactions, up to the point of implementing distributed transactions, which means that you will have written a distributed database.

Pros:
- The system scales to a predefined number of instances.
- Perfect fault tolerance if replication and error recovery is implemented.
- Latency is kept low.

Cons:
- Direct communication between *shards* (the *mesh engine*) is likely to be very complex.
- Intershard transactions are slow and/or complicated and may corrupt data if undertested.
- A client must know which shards own their data to benefit from the low latency. A *load balancer ambassador* [DDS] may be used on the client's side.

Split data to isolated shards and add a load balancer

The load balancer knows which shard has the client's data

Slicing to independent shards scales the system perfectly but needs a load balancer which increases latency
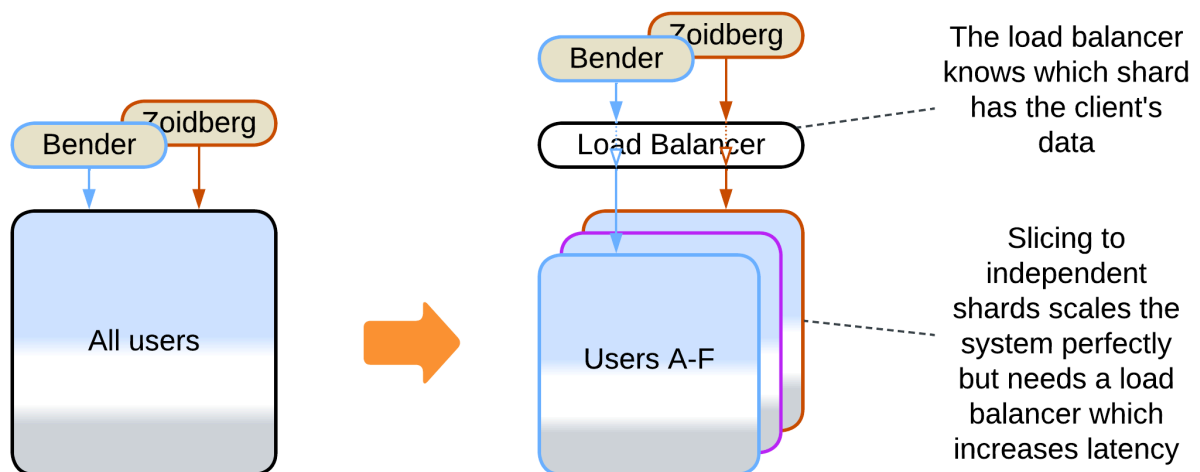
Patterns: Shards, Load Balancer (Proxy), Layers.
Goal: scale an application with sliceable data.
Prerequisite: the application's data can be sliced to independent self-sufficient parts.

If all the data a user operates on is never accessed by other users, multiple independent instances (*shards*) of the application can be deployed, each owning an instance of a database. A special kind of *proxy*, called *load balancer*, redirects a user request to a shard that has the user's data.

Pros:
- Perfect static (predefined number of instances) scalability.
- Failure of a shard does not affect users of other shards.
- Canary release is supported.

Cons:
- The *load balancer* increases latency and is a single point of failure unless duplicated, unless it is replicated to each client as an *ambassador* [DDS].

Separate the data layer and add a load balancer
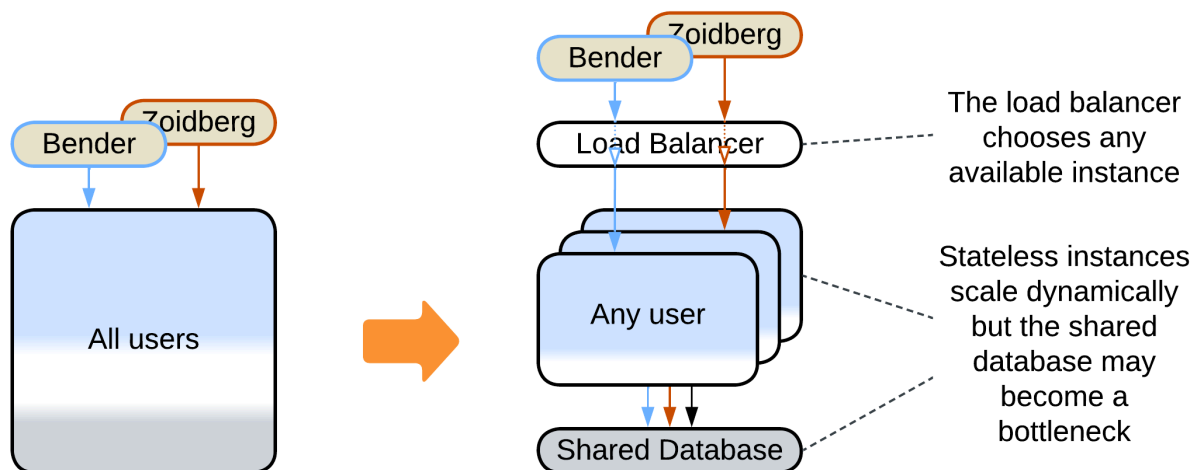
The load balancer chooses any available instance

Stateless instances scale dynamically but the shared database may become a bottleneck

Patterns: Pool (Shards), Shared Database (Shared Repository), Load Balancer (Proxy), Layers.

Goal: achieve scalability with little effort.

Prerequisite: there is persistent data of manageable size.

As data moves to a dedicated layer, the application becomes stateless and instances of it can be created and destroyed dynamically depending on the load. However, the *shared database* becomes the system's bottleneck unless *Space-Based Architecture* is used.

Pros:
- Easy to implement.
- Dynamic scalability.
- Failure of a single instance affects few users.
- Canary release is supported.

Cons:
- The database limits the system's scalability and performance.
- The *load balancer* and *shared database* increase latency and are single points of failure.

Dedicate an instance to each client

Every user gets a dedicated instance for the duration of their session

Each instance loads its user's data on creation and persists any changes to the database
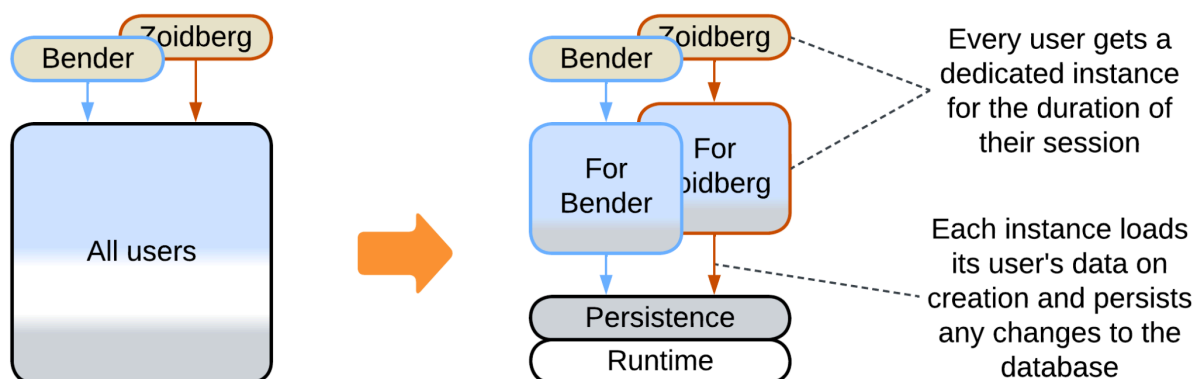
Patterns: Create on Demand (Shards), Shared Repository, Virtualizer (Microkernel), Layers.

Goal: very low latency, dynamic scalability and failure isolation.

Prerequisite: each client's data is small and independent of other clients.

Each client gets an instance of the application which preloads their data into memory. This way all the data is instantly accessible and a processing fault for one client never affects other clients. As systems tend to have thousands to millions of clients, it is inefficient to spawn a process per client. Instead, more lightweight entities are used: a page in a web browser or an actor in a distributed framework.

Pros:
- Nearly perfect dynamic scalability (limited by the persistence layer).
- Good latency as everything happens in RAM.
- Fault isolation is one of the features of distributed frameworks.
- Frameworks are available out of the box.

Cons:
- Virtualization frameworks tend to have a performance penalty.
- You may need to learn an uncommon technology.
- Scalability and performance are still limited by the shared persistence layer.

Further steps

In most cases sharding does not change much inside the application, thus the common evolutions for *Monolith* (to *Layers*, *Services* and *Pipeline*) remain applicable after sharding. We'll focus on their scalability:

- *Layers* can be scaled and deployed individually, often to a drastic extent, as exemplified by *Three-Tier Architecture*.
- *Services* allow for subdomains to scale independently, with the help of *load balancers* or a *middleware*. They also kind of scale the data as each service uses its own database, which is often chosen to best fit its distinct needs.
- Granular scaling can apply to *pipelines*, but in many cases that does not make much sense as pipeline components tend to be lightweight and stateless, making it easy to scale the pipeline as a whole.

Layers   **Services with a Middleware**   **Pipeline**
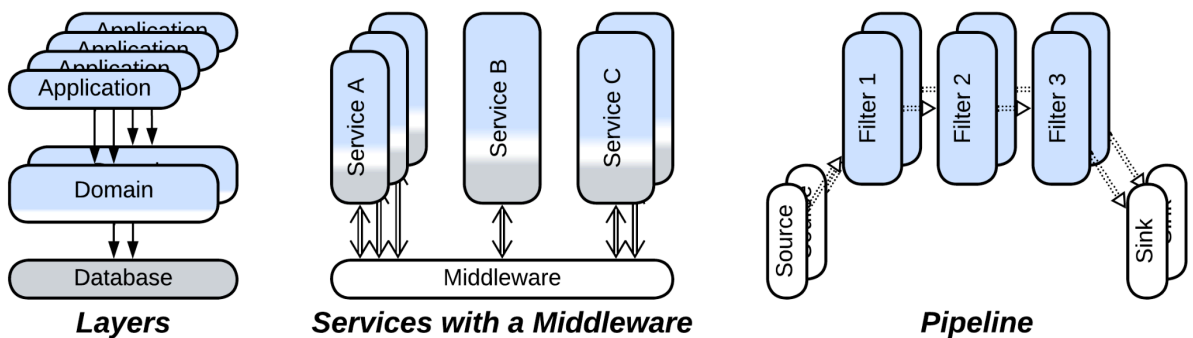
There are specific evolutions of *Shards* that deal with their drawbacks:

- *Space-Based Architecture* reimplements *Shared Repository* with *Mesh*. Its main goal is to make the data layer dynamically scalable, but the exact results are limited by the CAP theorem thus, depending on the mode of action, it can provide very high performance for a small dataset with no consistency guarantees or a reasonable performance for a huge dataset. It blends the best features of stateful *Shards* and *Shared Database* (thus is an option for either to evolve to) but may be quite expensive to run and lacks algorithmic support for analytical queries.
- *Orchestrator* is a mirror image of *Shared Database*, another option to implement use cases that deal with data in multiple shards without the need for the shards to

communicate to each other. As orchestrators are stateless they scale well, but if two orchestrators write to overlapping sets of data, they may corrupt it.

Data grid colocates cached data with every application instance

Space-Based Architecture

Each orchestrator accesses all the shards and integrates their data

Shards with Orchestrators
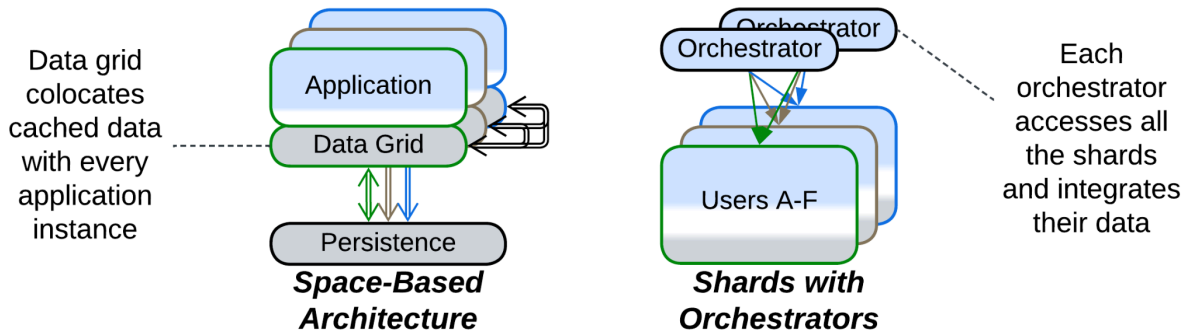
Monolith: to Layers

Another drawback of *Monolith* is its… monolithism. The entire application exposes a single set of qualities and all its parts (if they ever emerge) are deployed together. However, life awards flexibility: parts of a system may benefit from being written in varying languages and styles, deployed with different frequency and amount of testing, sometimes to specific hardware or end users' devices. They may need to vary in security and scalability as well. Enter *Layers* – a subdivision by the *level of abstractness*:

- Most *monoliths* can be divided into 3 or 4 *layers* of different abstractness.
- It is common to see the database separated from the main application.
- *Proxies* (e.g. *Firewall*, *Cache*, *Reverse Proxy*) are usual additions to the system.
- An *orchestrator* adds a layer of indirection to simplify the system's API for its clients.

Divide into layers

Layers are intuitive and they may differ in qualities and technologies and scale independently
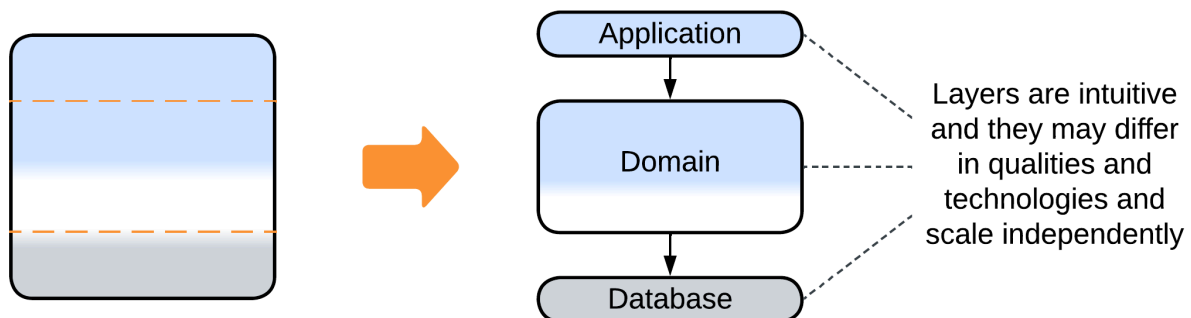
Patterns: Layers.

Goal: let parts of the system vary in qualities, improve the structure of the code.

Prerequisite: there is a natural way to split the high-level logic from the low level implementation details and dependencies.

Most systems apply *layering* by default as it grants much flexibility at very little cost. The usual set of layers contains: UI, tasks (orchestration), domain (detailed business rules) and infrastructure (database and libraries).

Pros:

- It is a natural way to specialize and decouple two or three development teams.
- The layers may vary in virtually any quality:
  - They are deployed and scaled independently.

- ○ They may run on different hardware, including client devices.
- ○ They may vary in programming language, paradigm and release cycle.
- Most changes are isolated to one layer.
- Layering opens a way to many evolutions of the system.
- The code becomes easier to read.

Cons:
- Dividing an existing application into layers may take some effort.
- There is a small performance penalty.

Use a database

Stateful Application

Stateless Application

Your application becomes stateless, thus multiple instances of it can be deployed

Database

The database persists and queries the data
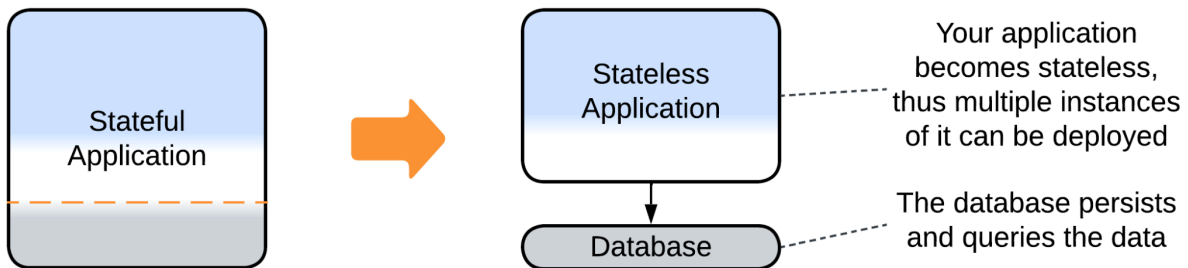
Patterns: Layers, Shared Database (Shared Repository).

Goal: avoid implementing a database.

Prerequisite: the system needs to query (maybe also persist) a large amount of data.

A database is untrivial to implement. While ordinary files are good for small volumes of data, as your needs grow so needs to grow your technology. Install a database.

Pros:
- A well-known database is sure to be more reliable than any in-house implementation.
- Many databases provide heavily optimized algorithms for querying data.
- You can choose hardware to deploy the database to.
- Your (now stateless) application will be easy to scale.

Cons:
- Databases are complex and require fine-tuning.
- You cannot adapt the engine of the database to your evolving needs.
- Most databases do not scale.
- You are stepping right into vendor lock-in.

Further steps:
- Continue the transition to *Layers* by dividing the high-level and low-level business logic.
- *Polyglot Persistence* improves performance of the data layer.
- *CQRS* passes read and write requests through dedicated services.
- *Space-Based Architecture* is low latency and allows for dynamic scalability of the whole system, including the data layer.
- *Hexagonal Architecture* will allow you to switch to another database in the future.

Add a proxy

A proxy provides a generic functionality on behalf of your system

That saves you time for implementing business logic
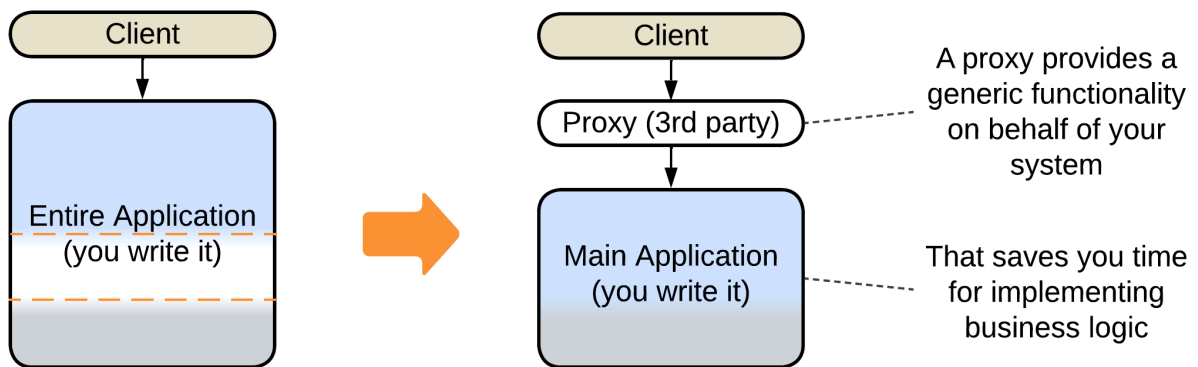
Patterns: Layers, Proxy.
Goal: avoid implementing a generic functionality.
Prerequisite: Your system serves clients (as opposed to controlling hardware).

A *proxy* is placed between your system and its clients to provide a generic functionality that otherwise would have to be implemented by the system. The kinds of *Proxy* to use with *Monolith* are: *Firewall*, *Cache*, *Reverse Proxy*, *Adapter*. Multiple *proxies* can be deployed.

Pros:
● You save some time (and money) on development.
● A well-known proxy is likely to be more reliable than an in-house implementation.
● You can select hardware to deploy the proxy to.

Cons:
● Latency degrades, except for *Response Cache* where it depends on the history of requests.
● The proxy may fail, which increases the chance of failure of your system.
● Beware of vendor lock-in.

Further steps:
● Another kind of *Proxy* may be added.
● Some environments employ a *proxy* per client, leading to *Backends for Frontends*.

Add an orchestrator

The orchestrator lets clients do many things with one request

The internal connection is much faster than the connection to clients
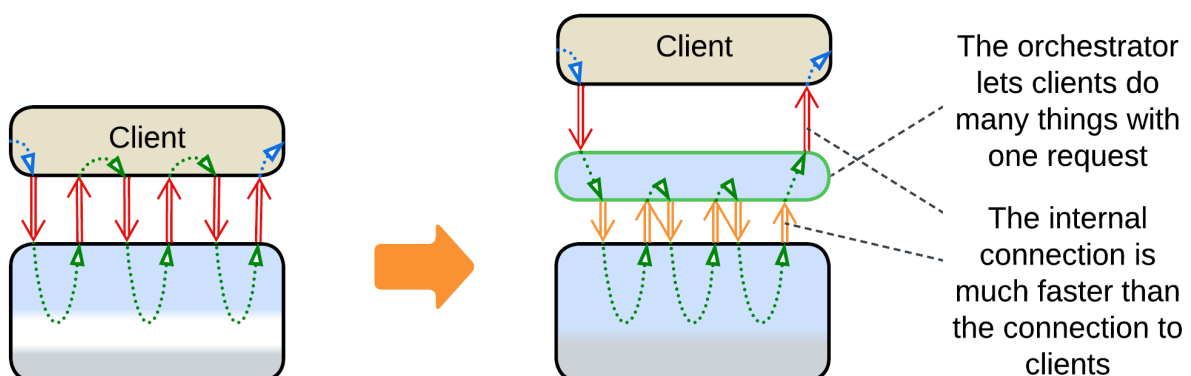
Patterns: Layers, Orchestrator.
Goal: provide a high-level API for clients to improve developer experience and performance.
Prerequisite: the API of your system is fine-grained and there are common use cases that repeat certain sequences of calls to your API.

A well-designed orchestrator should provide a high-level API which is intuitive, easy to use and coarse-grained to minimize the number of interactions between the system and its clients. An old way to access the original system's API may be maintained for uncommon use cases or legacy client applications. As a matter of fact, you perform some programming on behalf of your clients.

Pros:
- Client applications become easier to write.
- The latency improves.

Cons:
- You get yet another moving part to design, test, deploy and observe. With lots of extra meetings between development teams.
- The new coarse-grained interface will likely be less powerful than the original one.

Further steps:
- *Backends for Frontends* use an orchestrator per client type.

Further steps

Applying one of the evolutions discussed above does not prevent you from following another of them, or even the same one for the second time:
- A layer can be split in two layers.
- A database can be added.
- Multiple kinds of proxies are OK.
- If you don't have an orchestration layer, you may add one.

Those were evolutions from *Layers* to *Layers*.

Another set of evolutions stems from splitting one or more *layers* into *services*:
- Splitting a *proxy* and/or *orchestrator* yields *Backends for Frontends* where requests from each kind of client are processed by a dedicated module.
- Splitting the *layer* with the main business logic results in *Services*, possibly augmented with layers of *middleware*, *shared database*, *proxies* and/or *orchestrator*.
- Splitting the *database layer* leads to *Polyglot Persistence* with specialized storages.
- If all the *layers* share the domain dimension and are split by it, *Layered Services* (or its subtype *CQRS*) emerge.
- If each *layer* is split along its own domain, the system follows *Service-Oriented Architecture* that is built around component reuse.
- Finally, some domains support *Hierarchy* – a tree-like architecture where each layer takes a share of the system's functionality.
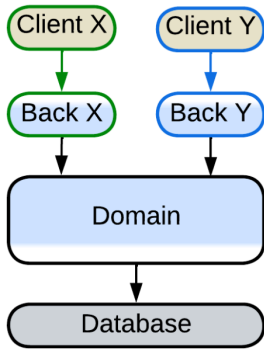
Backends for Frontends
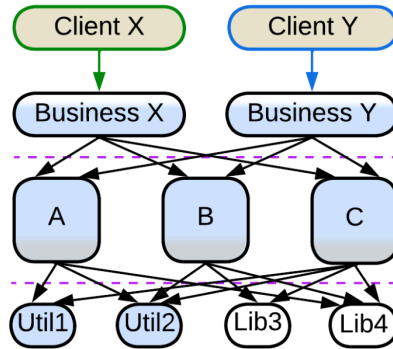
Service-Oriented Architecture
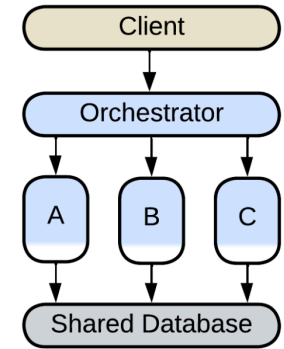
Services with an Orchestrator and a Shared Database
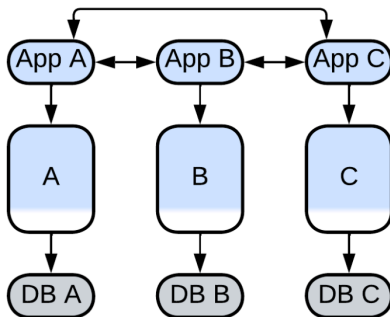
Layered Services
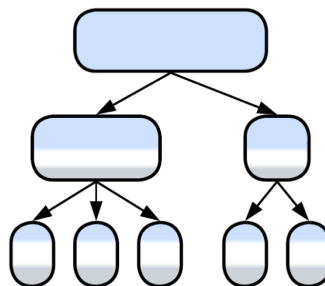
Hierarchy
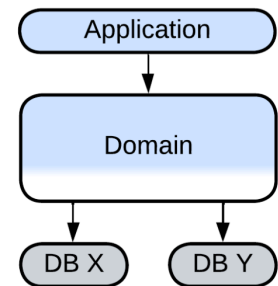
Polyglot Persistence

In addition,

- Many domains allow for scaling one or more layers.
- A layer may employ *plugins* for better customizability.
- The UI and infrastructure layers may be split and abstracted according to the rules of *Hexagonal Architecture* (or its subtype *Model-View-Controller*).
- The system can often be extended with *scripts*, resulting in a kind of *Microkernel*.
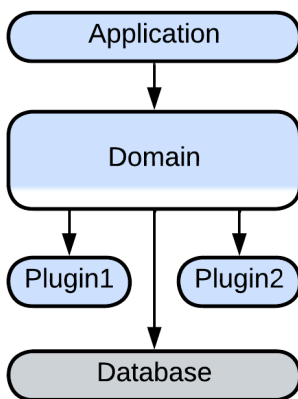
Layers with Plugins
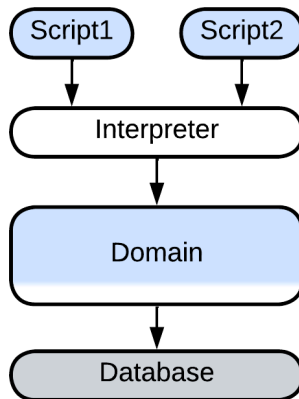
Layers with Scripts
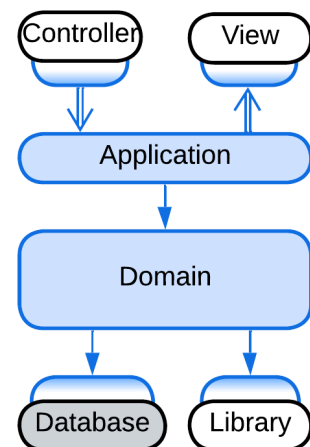
Layered Hexagonal Architecture

Monolith: to Services

The final major drawback of _Monolith_ is the cohesiveness of its code. The rapid start of development begets a major obstacle as the project grows: every developer needs to know the entire codebase to be productive, the changes made by individual developers overlap and may break each other. Such a distress is usually solved by dividing the project into components along _subdomain boundaries_ (which tend to match bounded contexts). However, that requires much work, and good boundaries and APIs are hard to design. Thus many organizations prefer a slower iterative transition.

- A _monolith_ can be split to _services_ right away.
- Or only new features may be added as _services_.
- Or weakly coupled parts of existing functionality may be separated, one at a time.
- Some domains allow for sequential data processing best described by _pipelines_.

With services there is a single component that owns a user's request

It may ask other services for help

Services   **Pipeline**

With pipeline a request passes through all the components, each doing its part

Divide into services

Services divide the codebase into parts of comparable size that can be developed almost independently
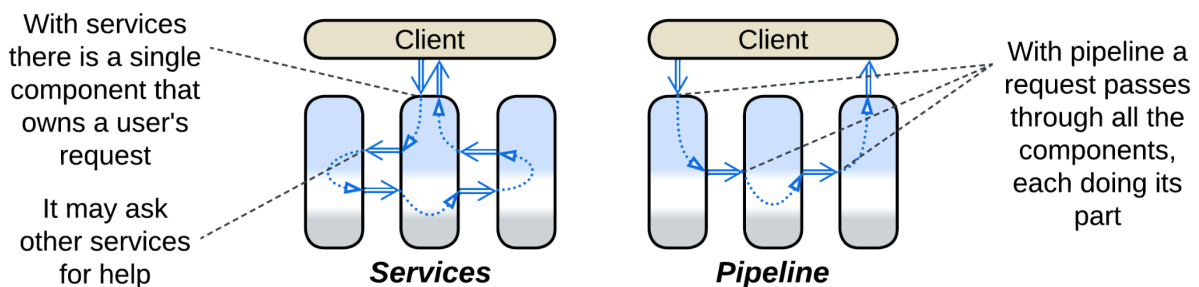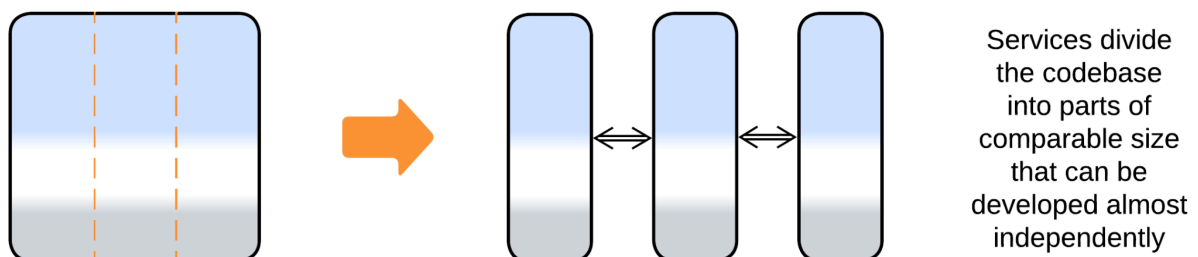
Patterns: Services.

Goal: facilitate development by multiple teams, improve the code, decouple qualities of subdomains.

Prerequisite: there is a natural way to split the business logic into loosely coupled subdomains, and the subdomain boundaries are sure to never change in the future.

Splitting a monolith into services by subdomain is risky at early stages of projects while the domain understanding is evolving. However, this is the way to go as soon as the codebase becomes unwieldy due to its size.

Pros:
- Supports multiple relatively independent and specialized development teams.
- Lowers the penalty imposed by the project's size and complexity on the velocity of development and product quality.
- Each team may choose the best fitting technologies for its service.
- The services can differ in non-functional requirements.
- Flexible deployment and scaling.
- A certain degree of error tolerance for asynchronous systems.

Cons:

- It takes lots of work to split a monolith.
- Future changes in the overall structure of the domain will be hard to implement.
- Sharing data between services is complicated and error-prone.
- System-wide use cases are hard to understand and debug.
- There is a moderate performance penalty for system-wide use cases.

Add or split a service

Splitting a service allows you to modify its functionality with little danger of breaking unrelated things
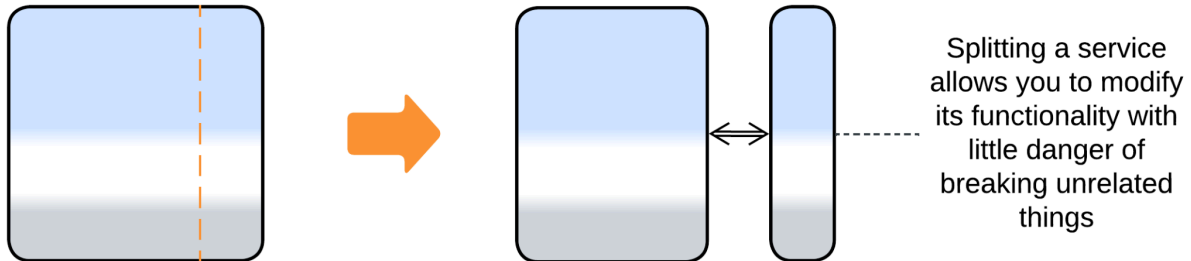
Patterns: Services.

Goal: stop digging, get some work for novices who don't know the entire project.

Prerequisite: the new functionality you are adding or the part you are splitting is weakly coupled to the bulk of the existing monolith.

If your *monolith* is already hard to manage, but a new functionality is needed, you can try dedicating a separate service to the new feature(s). This way the *monolith* does not become larger – it is even possible that you will move some of its code to the newly established *service*.

If you are not adding a new feature but need to change an old one – use the chance to make the existing *monolith* smaller by first separating the functionality that you are going to change. At the very minimum the two-step process lowers the probability of breaking something unrelated to the changes of behavior required.

Pros:
- The legacy code does not increase in size and complexity.
- The new service is transferred to a dedicated team that does not need to know the legacy system.
- The new service may be experimented with and even rewritten from scratch.
- The likely faults of the new service don't crash the main application.
- The new service can be tested and deployed in isolation.
- The new service can be scaled independently.

Cons:
- The new service will have a hard time sharing data or code with the main application.
- Use cases that involve both the new service and the old application are hard to debug.
- There is a moderate performance penalty for using the service.

Further steps:
- Continue disassembling the *monolith*.

Divide into a pipeline

Pipelines use small replaceable components to process streams of data
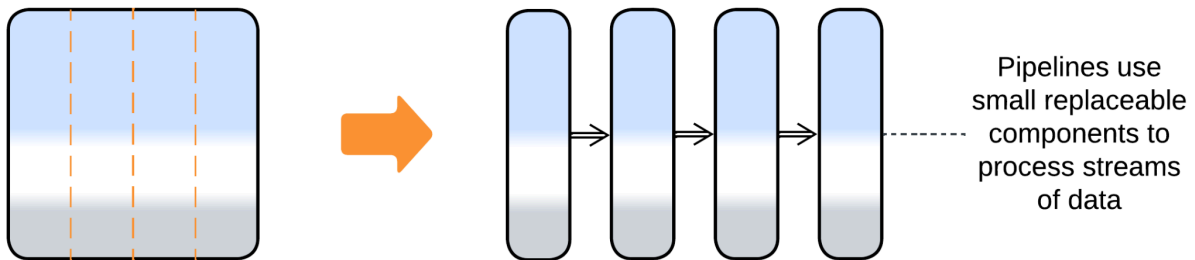
Patterns: Pipeline (Services).

Goal: decrease complexity of the code, make it easy to experiment with the steps of data processing, simplify the use of multiple CPU cores, processors or computers.

Prerequisite: the domain can be represented as a sequence of coarse-grained data processing steps.

If you can treat your application as a chain of independent steps that transform the input data, you can rely on the OS to schedule them and you can also dedicate a development team to each of the steps. This is the default solution for a system that processes a stream of a single type of data (video, audio, measurements). It has excellent flexibility.

Pros:
- Nearly abolishes the influence of the total project size on the development velocity.
- The project's teams become almost independent.
- Flexible deployment and scaling.
- Naturally supports event replay as the means of reproducing bugs or testing / benchmarking individual components.
- It is possible to provide multiple implementations for each of the steps of the data processing.
- Does not need any manual scheduling or thread synchronization.

Cons:
- Latency is likely to skyrocket.
- As the number of supported scenarios grows, so does the number of building blocks for your pipelines. Soon there'll be none who understands the system in whole.

Further steps

As your knowledge of the domain and your business requirements change, you may be in need to move some functionality between the services to keep them loosely coupled. Sometimes you have to merge two or three services together. So it goes.

Systems of services or pipelines are quite often extended with special kinds of layers:
- *Middleware* helps with deployment, intercommunication and scaling of *Services*.
- *Shared Repository* lets *Services* operate on and communicate through shared data.
- *Proxies* are ready-to-use components that add generic functionality to the system.
- *Orchestrator* encapsulates use cases that involve multiple services, so that the services don't need to know about each other.
- Finally, there are *combined components* that implement two or more of the above patterns in a single framework.

A proxy stands between the system and its clients and acts on the system's behalf

Services with a Proxy

An orchestrator uses the underlying services to execute client requests

Services with an Orchestrator

A middleware provides communication for and manages the service instances

Services with a Middleware

A shared database is the simplest way for the services to persist data

Services with a Shared Database

Each service, being a smaller *monolith*, may evolve on its own. Most of the evolutions of *Monolith* are applicable. The most common examples include:

- *Scaled (Sharded) Service* with a *load balancer* and *shared database* to support high load.
- *Layered Service* to improve the code structure and decouple deployment of parts of a service.
- *Cell (Service of Services)* to involve multiple teams and technologies with a single service.
- *Hexagonal Service* to escape vendor lock-in.

Scaled Service   **Layered Service**   **Cell**   **Hexagonal Service**
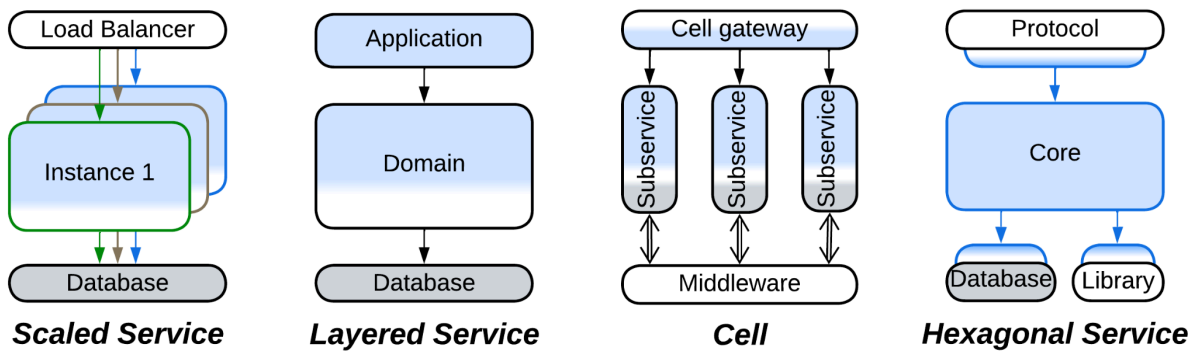
Monolith: to Plugins

The last group of evolutions does not really change the monolithic nature of the application. Instead, its goal is to improve the customizability of the *monolith*:

- Vanilla *Plugins* is the most direct approach which relies on replaceable bits of logic.
- *Hexagonal Architecture* is a subtype of *Plugins* that is all about isolating the main code from any 3rd party components it uses.
- *Scripts* is a kind of *Microkernel* – yet another subtype of *Plugins* – which gives users of the system full control over its behavior.

Support plugins

Each plugin customizes a single aspect of the application's behavior
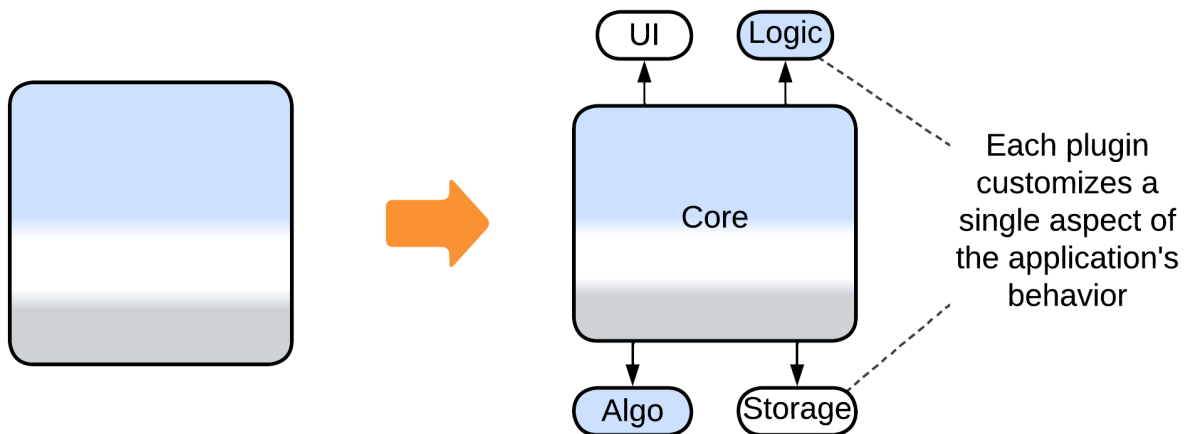
Patterns: Plugins.
Goal: simplify the customization of the application's behavior.
Prerequisite: several aspects need to vary from customer to customer.

Plugins create points of access to the system that allow engineers to collect data and govern select aspects of the system's behavior without having to learn the system's implementation.

Pros:
- The system can be modified by internal and external programmers who don't know its internal details.
- Multiple customized versions become much easier to release and support.

Cons:
- Extensive changes may be required to expose the tunable aspects of the system.
- Testability becomes poor because of the number of possible configurations.
- Performance degrades.

Isolate dependencies with Hexagonal Architecture

All the external dependencies are isolated with adapters
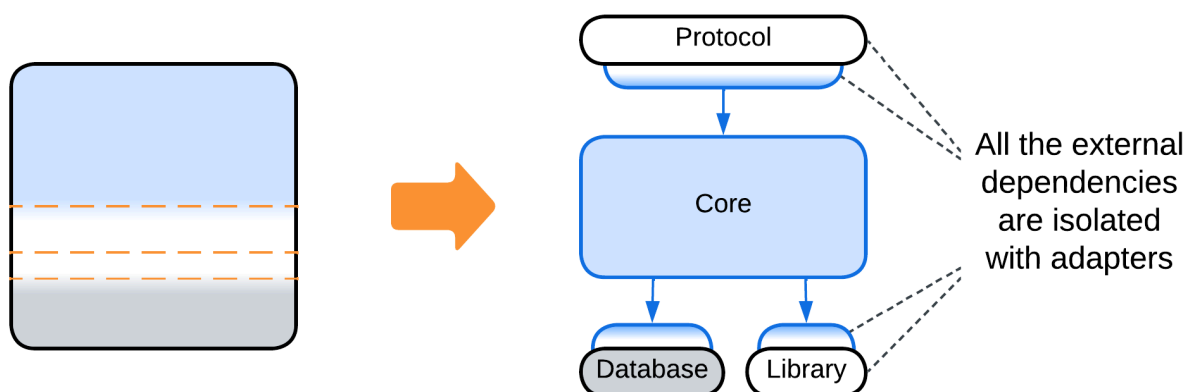
Patterns: Hexagonal Architecture (Plugins).
Goal: isolate the business logic from external dependencies.
Prerequisite: there are 3rd party or unstable components in the system.

The main business logic communicates with all the external components through APIs or SPIs defined in the terms of the business logic. This way it does not depend on anything at all, and any component may be replaced by another implementation or a stub.

Pros:

- Vendor lock-in is ruled out.
- A component may be replaced till the very end of the system's life cycle.
- Stubs are supported for testing and local or early development.
- It is possible to provide multiple implementations of the components.

Cons:
- Some extra effort is required to define and use the interfaces.
- There is a performance degradation, mostly due to lost optimization opportunities.

Add an interpreter (support scripts)
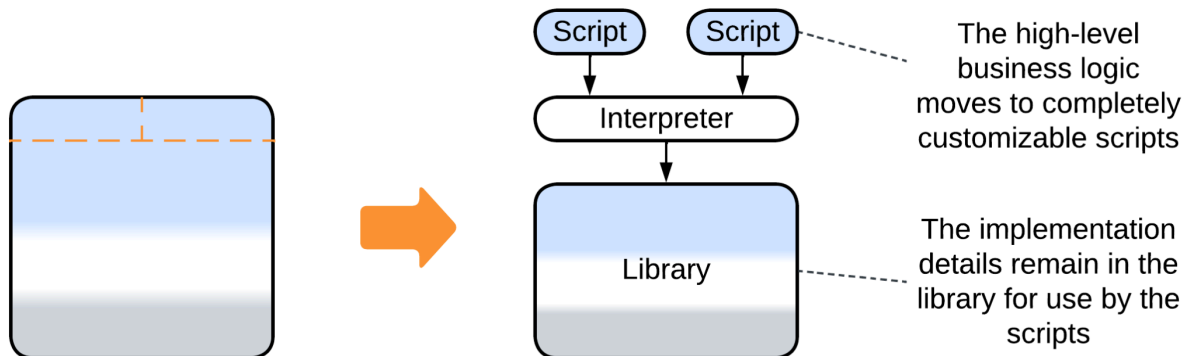
Patterns: Scripts aka Interpreter (Microkernel (Plugins)).

Goal: allow the system's users to implement their own business logic.

Prerequisite: the domain is representable in high-level terms.

Interpreter lets the users develop high-level business logic from scratch by programming interactions of pre-defined building blocks which are implemented by the lower layer of the system. That provides unparalleled flexibility at the cost of performance and design complexity.

Pros:
- Perfect flexibility and customizability for every user.
- The high-level business logic can be written in high-level terms, making it fast to develop and easy to grasp.

Cons:
- Requires much effort to design correctly.
- There may be a heavy performance penalty if the API is too fine-grained.
- Testability may be an issue.

Shards: share data

The issue peculiar to *Shards* is that of coordinating the deployed instances, especially if their data becomes coupled. The most direct solution is to let the instances operate a shared data:
- If the whole dataset needs to be shared, it can be split to a *shared repository* layer.
- If data collisions are tolerated, *Space-Based Architecture* promises low latency and dynamic scalability.
- If a part of the system's data becomes coupled, only that part can be moved to a *shared repository*, causing each instance to manage two stores of data: private and shared.

- Another option is to split a *service* that owns the coupled data and is always deployed as a single instance. The remaining parts of the system become coupled to that service, not each other.

Move all the data to a shared repository
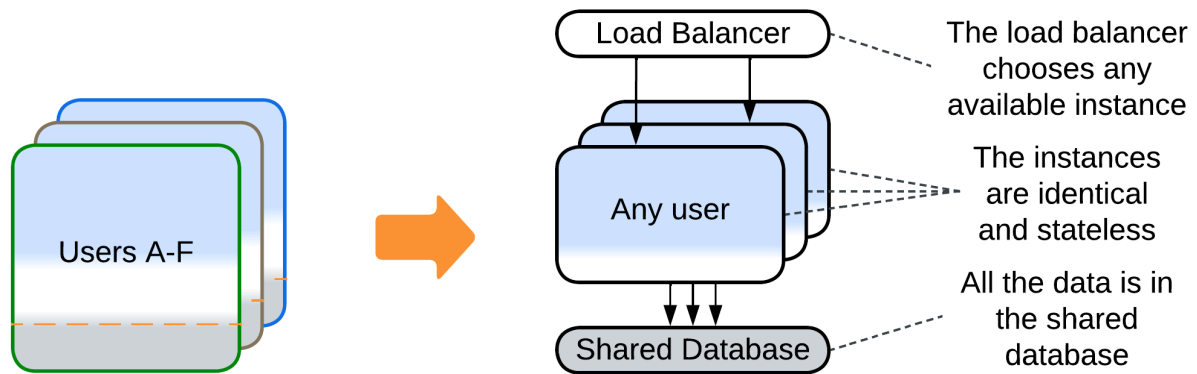
Patterns: Pool (Shards), Shared Database (Shared Repository), Load Balancer (Proxy), Layers.

Goal: don't struggle against the coupling of the shards, keep it simple and stupid.

Prerequisite: the system is not under pressure for data size or latency (addressed by the further evolutions).

In case a shard needs to access the data owned by any other shard, the prerequisite of the independence of shards starts to fall apart. Grab all the data of all the shards and push it into a *shared database*, if you can (there may be too much data or the database access may be too slow). As all the shards become identical, you'll likely need a *load balancer*.

Pros:
- You can choose one of the many specialized databases available.
- The stateless instances of the main application become dynamically scalable.
- Failure of a single instance affects few users.
- Canary release is supported.

Cons:
- The database limits the system's scalability and performance.
- The *load balancer* and *shared database* increase latency and are single points of failure.

Further steps:
- *Hexagonal Architecture* will let you switch to another database in the future.
- *Space-Based Architecture* decreases latency by co-locating subsets of the data and the application runtime.
- *Polyglot Persistence* uses multiple specialized databases, often by separating commands and queries. That may greatly relieve the primary (write) database.
- *CQRS* goes even further by processing read and write requests with dedicated services.

Use Space-Based Architecture

The data grid provides any instance with fast access to all the data

The data is backed up to the persistent database
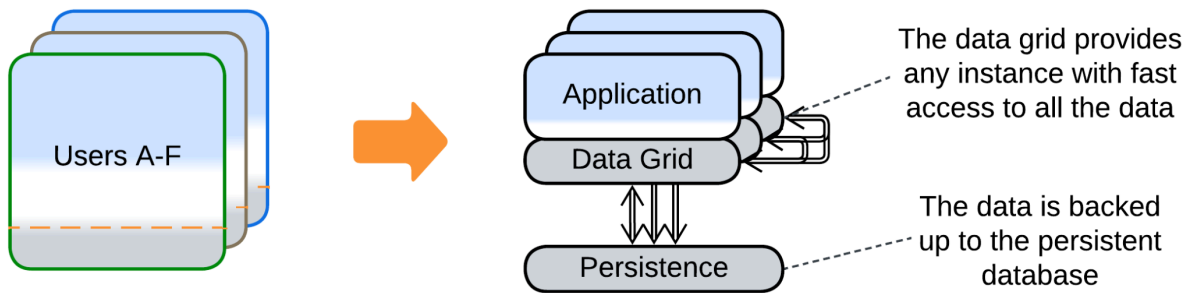
Patterns: Space-Based Architecture (Mesh), Data Grid (Shared Repository), Shards, Layers.

Goal: don't struggle against the coupling between the shards, maintain high performance.

Prerequisite: data collisions are acceptable.

Space-Based Architecture is a *mesh* of nodes that comprise the application and a cached subset of the system's data. A node broadcasts any changes to its data to the other nodes and it may request any data that it needs from the other nodes. Collectively, the nodes of the mesh keep the whole data cached in RAM.

Though *Space-Based Architecture* may provide multiple modes of action, including single write / multiple read replicas, it is most efficient when there is no write synchronization between its nodes, meaning that data consistency is sacrificed for performance and scalability.

Pros:
● Unlimited dynamic scalability.
● Off-the-shelf solutions are available.
● Failure of a single instance affects few users.

Cons:
● Choose one: data collisions or mediocre performance.
● Low latency is supported only for datasets that fit in RAM of a node.
● High operational cost because the nodes exchange huge amounts of data.
● No support for analytical queries.

Add a shared repository for the coupled subset of the data

Each shard retains its identity and data

The coupled subset of data moves to a shared DB
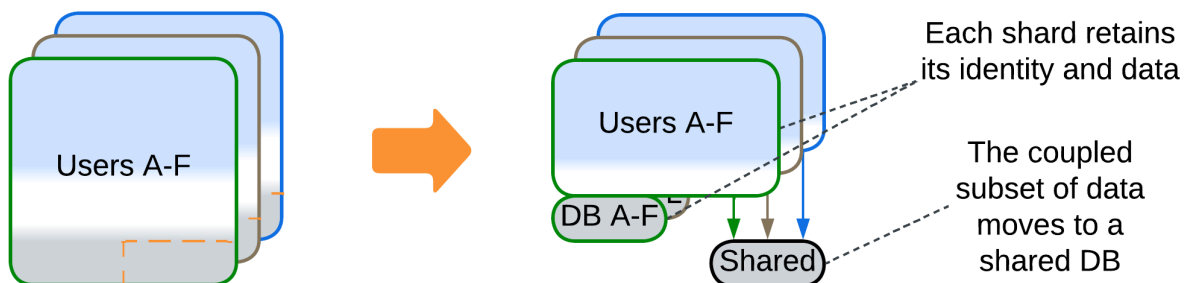
Patterns: Shards, Polyglot Persistence, Shared Database (Shared Repository), Layers.

Goal: solve the coupling between the shards without losing performance.

Prerequisite: the shards are coupled through a small subset of data.

If a subset of the data is accessed by all the shards, that subset can be moved to a dedicated database, which is likely to be fast if only because it is small. Using a distributed database that keeps its data synchronized on all the shards may be even faster.

The approach resembles *Shared Kernel* of [DDD].

Pros:

● You can choose one of the many specialized databases available.

Cons:

● The *shared database* increases latency and is the single point of failure.

Split a service with the coupled data

Each shard retains its identity and data

The coupled data and logic moves to a shared service
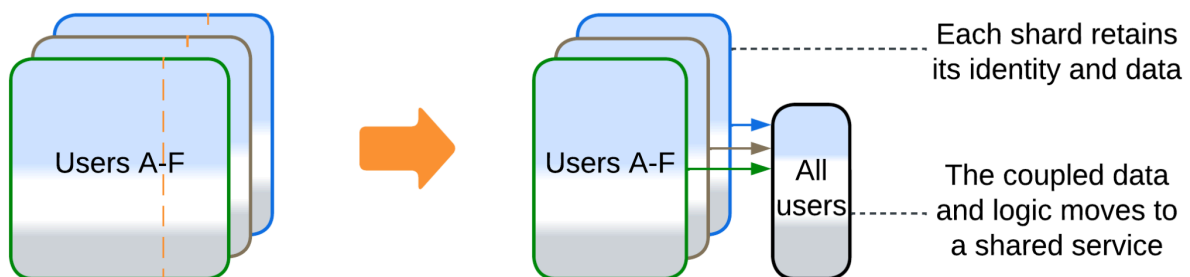
Patterns: Services, Shards.

Goal: solve the coupling between the shards in an honorable way.

Prerequisite: the part of the domain that causes the coupling between the shards is weakly coupled to the remaining domain.

If a part of the domain is too cohesive to be sharded, we can often move it from the main application into a dedicated service. That way the main application remains sharded, while the new service exists as a single instance. In rare cases there is a chance to re-shard the new service with the key that is different from the one used for sharding the main application.

The approach resembles *Shared Kernel* of [DDD].

Pros:

● The main code should become a little bit simpler.
● The new service can be given to a new team.
● The new service may choose a database that best fits its needs.

Cons:

● Now it's hard to share data between the new service and the main application.
● Scenarios that use the new service are harder to debug.
● There is a moderate performance penalty for using the service.

Shards: share logic

Other cases are better solved by extracting the logic that manipulates multiple *shards*:

● Splitting a *service* (as discussed above) yields a component that represents both shared data and shared logic.
● Adding a *middleware* lets the shards communicate to each other without keeping direct connections. It also may do housekeeping: error recovery, replication and scaling.
● A *load balancer* decouples clients from the knowledge about the existence of the shards.

- An *orchestrator* calls multiple shards to serve a user request. That relieves the shards of the need to coordinate their states and actions by themselves.

Add a middleware

The middleware manages the shards and provides them with a means of communication
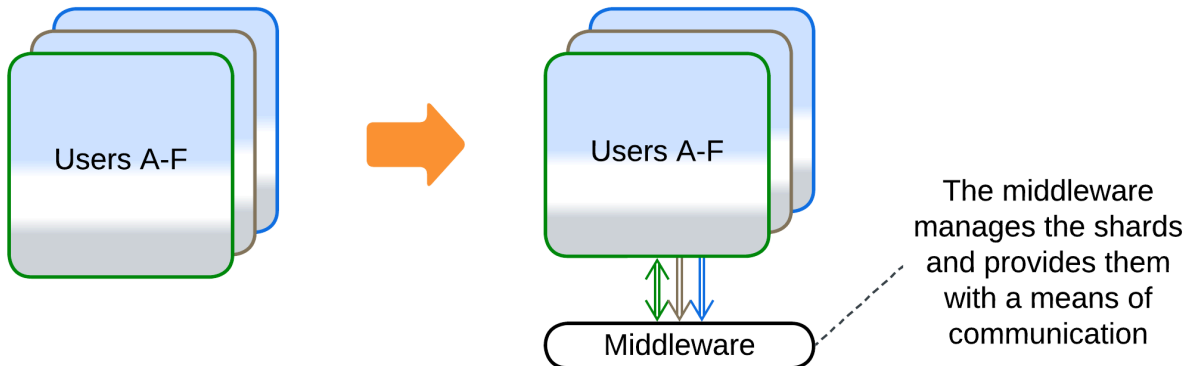
Patterns: Shards, Middleware, Layers.
Goal: simplify the communication between shards, their deployment and recovery.
Prerequisite: many shards need to exchange information, some may fail.

A *middleware* transports messages between *shards*, checks their health and recovers any crashed ones. It may manage data replication and deployment of software updates as well.

Pros:
- The shards become simpler because they don't need to track each other.
- There are many good 3rd party implementations.

Cons:
- Performance may degrade.
- Components of the *middleware* are new points of failure.

Add a load balancer

The load balancer knows which shard has the client's data
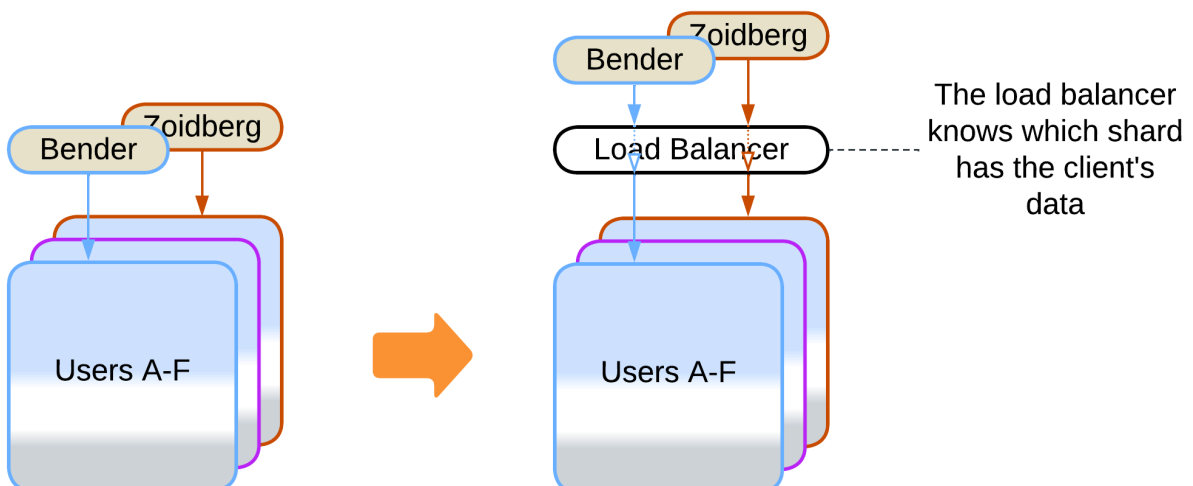
Patterns: Shards, Load Balancer (Proxy), Layers.
Goal: simplify the code on the client side.
Prerequisite: each client connects directly to the shard that owns their data.

The client application may know the address of the shard that serves it and connect to it without intermediaries. That is the fastest means of communication, but it prevents you from

changing the number of shards or other details of your implementation without updating all the clients, which may be unachievable. Use an intermediary.

   Pros:
   ● Your system becomes isolated from its clients.
   ● You can put generic aspects into the *proxy* instead of implementing them in the shards.
   ● Proxies are readily available.

   Cons:
   ● The extra step increases latency unless you deploy the *load balancer* as *ambassadors* [DDS] co-located with every client.
   ● The *load balancer* is a single point of failure unless duplicated.

Move the integration logic to an orchestrator

The orchestrator integrates information owned by multipls shards
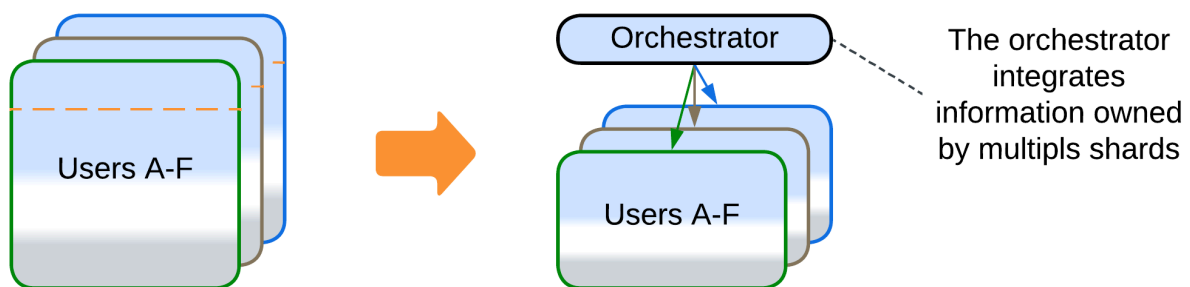
   Patterns: Shards, Orchestrator, Layers.
   Goal: isolate the shards from the knowledge of each other.
   Prerequisite: the shards are coupled in their high-level logic.

   When a high-level scenario uses multiple shards, the way to follow is to extract all such scenarios into a dedicated stateless module. That makes the shards independent of each other.

   Pros:
   ● The shards don't need to care about each other.
   ● The high-level logic can be written in a high-level language by a dedicated team.
   ● The high-level logic can be deployed independently.
   ● The main code should become much simpler.

   Cons:
   ● Latency will increase.
   ● The *orchestrator* becomes a single point of failure with a good chance to corrupt your data.

   Further steps:
   ● *Shard* the *orchestrator* (run multiple instances) to support higher load and to remain online if it fails.
   ● *Persist* the *orchestrator* (give it a dedicated database) to make sure that it does not leave half-committed transactions on failure.
   ● *Divide* the *orchestrator* into *Backends for Frontends* if you have multiple kinds of clients or workflows.

Layers: make more layers

Not all the layered architectures are equally layered. A *monolith* with a *proxy* or database has already stepped into the realm of *Layers* but is far away from reaping all its benefits. Such a kind of system may continue its journey in a few ways that were earlier discussed for *Monolith*:

- Employing a *database* (if you don't use one) lets you rely on a thoroughly optimized state-of-the-art subsystem for data processing and storage.
- *Proxies* are similarly reusable generic modules to be added at will.
- Implementing an *orchestrator* on top of your system may improve programming experience and runtime performance for your clients.

A proxy represents the system to its clients and takes care of one or more aspects of the communication

An orchestrator turns high-level requests into series of low-level steps

The domain layer implements business rules

The database provides persistence and analytics
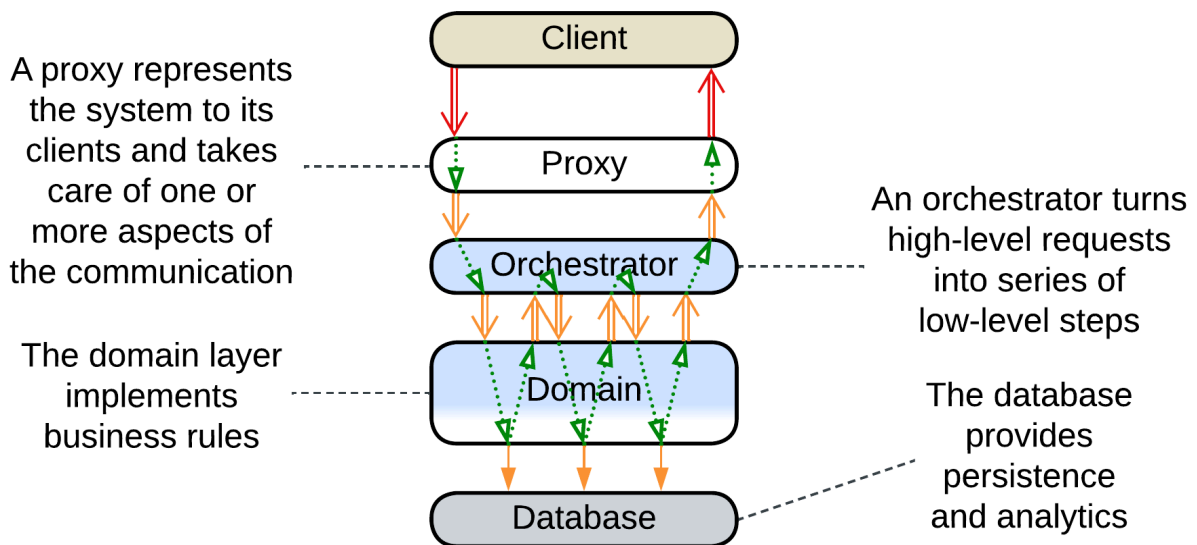
It is also common to:

- Have the business logic divided in two *layers*.

Split the business logic in two layers
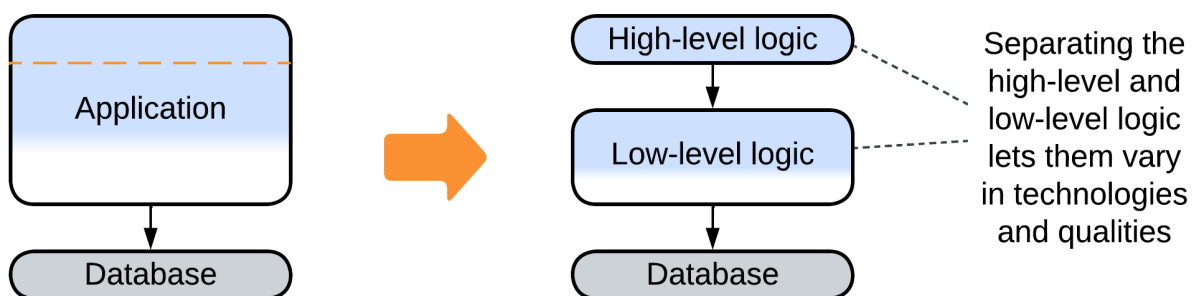
Separating the high-level and low-level logic lets them vary in technologies and qualities

Patterns: Layers.
Goal: let parts of the business logic vary in qualities, improve the structure of the code.
Prerequisite: the high-level and low-level logic are loosely coupled.

It is often possible to split a backend into integration (orchestration) and domain layers. That allows for one team to specialize in customer use cases while the other one delves deep into the domain knowledge and infrastructure.

Pros:
- You get an extra development team.

- The high-level use cases may be deployed separately from the main domain logic.
- The layers may diverge in technologies and styles.
- The code may become less complex.

Cons:
- There is a small performance penalty.

Layers: help large projects

The main drawback (and benefit as well) of *Layers* is that much or all of the business logic is kept together in one or two components. That allows for easy debugging and fast development in the initial stages of the project but slows down and complicates work as the project grows in size. The only way for a growing project to survive and continue evolving at a reasonable speed is to divide its business logic into several smaller, thus less complex, components that match subdomains (*bounded contexts* [DDD]). There are several options for such a change, with their applicability depending on the domain:
- The middle layer with the main business logic can be divided into *services* leaving the upper *orchestrator* and lower *database* layers intact for future evolutions.
- Sometimes the business logic can be represented as a set of directed graphs which is known as *Event-Driven Architecture*.
- If you are lucky, your domain makes a *Top-Down Hierarchy*.

Divide the domain layer into services

The domain layer, the largest component, is usually easy to split by subdomains
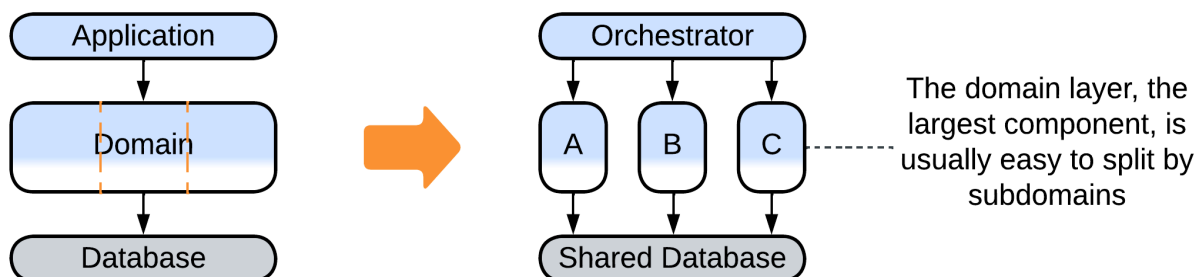
Patterns: Services, Shared Database (Shared Repository), Orchestrator.
Goal: make the code simpler and let several teams work on the project efficiently.
Prerequisite: the low-level business logic comprises weakly coupled subdomains.

It is very common for a system's domain to consist of weakly interacting bounded contexts [DDD]. They are integrated through high-level use cases and/or relations in data. For such a system it is relatively easy to divide the domain logic into services while leaving the integration and data layers shared.

Pros:
- You get multiple specialized development teams.
- The largest and most complex piece of code turns to several smaller modules.
- There is more flexibility with deployment and scaling.

Cons:
- Future changes in the overall structure of the domain will be harder to implement.
- System-wide use cases become somewhat harder to debug as they use multiple repositories.
- Performance may degrade, especially if the services and the orchestrator are not co-located.

Further steps:
- Continue by splitting the *orchestrator* and *database* to reach the *Layered Services* architecture.
- Divide the orchestrator (by the type of client) into *Backends for Frontends*.
- Use multiple databases in *Polyglot Persistence*.
- Scale well with *Space-Based Architecture*.

Build event-driven architecture over a shared database

Each kind of request passes through a dedicated pipeline
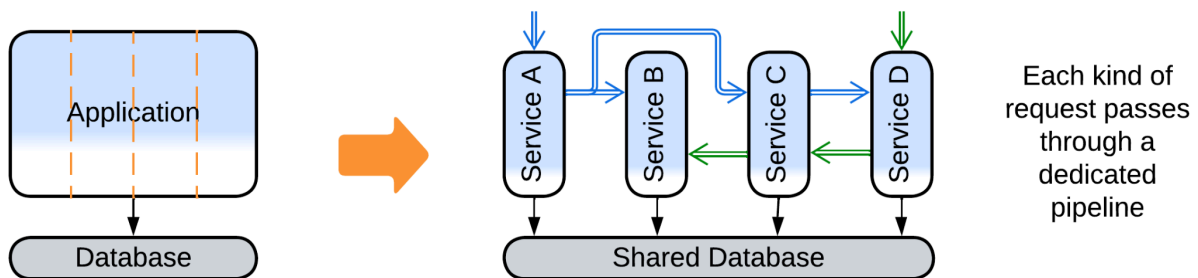
Patterns: Event-Driven Architecture (Pipeline (Services)), Shared Database (Shared Repository).

Goal: untangle the code, support multiple teams, improve scalability.

Prerequisite: use cases are sequences of loosely coupled coarse-grained steps.

If your system has a well-defined pipeline for processing every kind of input request, it can be divided into several services, each hosting a few related steps of multiple use cases. Each service subscribes to input and publishes its output events.

Pros:
- The code is divided into much smaller (and simpler) segments.
- It is very easy to add new steps or use cases as the structure is extremely flexible.
- You open a way to have several almost independent teams, one per service.
- You get flexible deployment and scaling as the services are stateless, but you need a *middleware* for that.
- The architecture naturally supports event replay as the means of reproducing bugs or testing / benchmarking individual components.
- There is no need for explicit scheduling or thread synchronization.

Cons:
- The system is hard to debug.
- You are going to live with high latency.
- You may end up with too many components which are interconnected in too many ways. Any change in a component or subscription may break a seemingly unrelated use case.

Further steps:
- Add a *middleware* that supports scaling and failure recovery.
- Split the *shared database* by subdomain to move closer to *Microservices*.
- Scale with *Space-Based Architecture*.
- Move the logic of use cases to an *orchestrator*.

Build a top-down hierarchy

Hierarchy distributes decision-making over multiple layers and subdomains
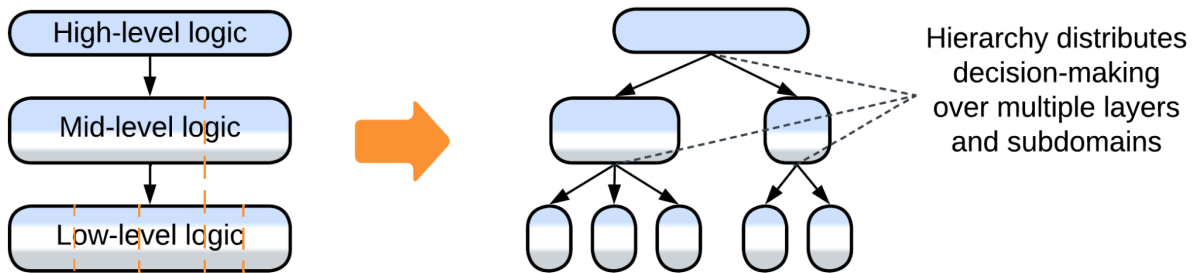
Patterns: Top-Down Hierarchy (Hierarchy).
Goal: untangle the code, support multiple teams, earn fine-grained scalability.
Prerequisite: the domain is hierarchical.

Splitting the lower layers into independent components with identical interfaces simplifies the managing code and allows the managed components to be deployed, developed and run independently of each other. Ideally, the mid-layer components should participate in decision-making so that the uppermost component is kept relatively simple.

Pros:
● Hierarchy is easy to develop and support with multiple teams.
● It is easy to modify or exchange individual components.
● The components scale, deploy and run independently.
● The system is quite fault tolerant.

Cons:
● It takes time and skill to deduce good interfaces.
● There are many components to administer.

Layers: improve performance

There are several ways to improve performance of a *layered system*. One we have already discussed for *Shards*:

● *Space-Based Architecture* co-locates the database and business logic and scales both dynamically.

The data grid is a highly-scalable in-memory database

The data is backed up to the persistent database
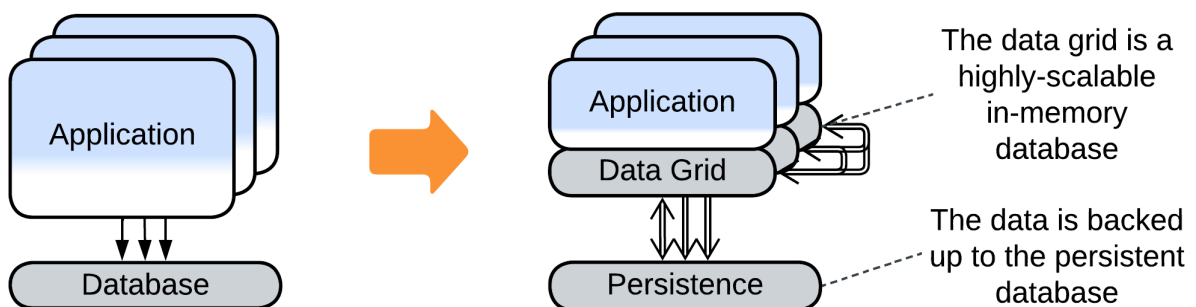
Others are new and thus deserve our attention:
● Merging several layers improves latency by eliminating the communication overhead.
● Scaling some of the layers may improve throughput but degrade latency.
● *Polyglot Persistence* is the name for using multiple specialized databases.

Merge several layers
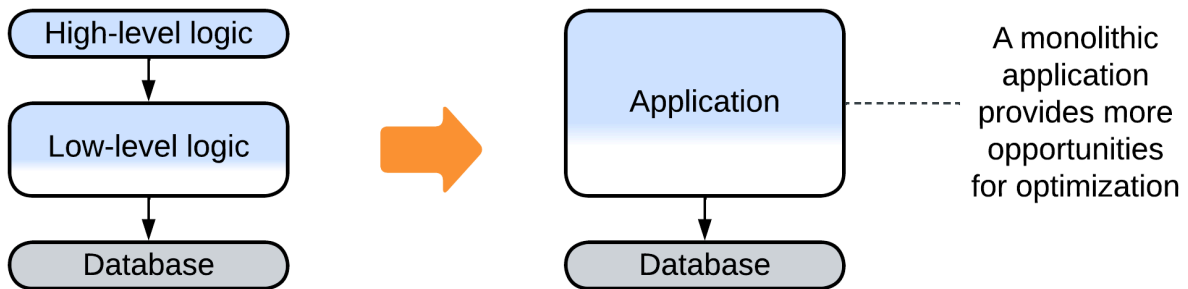
Patterns: Layers or Monolith
Goal: improve performance.
Prerequisite: the layers share programming language, hardware and qualities.

If your system's development is finished (no changes are expected) and you really need that extra 5% performance improvement, you can try merging everything back into a *Monolith* or *3-tier* (front, back, data).

Pros:
- Enables aggressive performance optimizations.
- The system may become easier to debug.

Cons:
- The code is frozen – it will be much harder to evolve.
- Your teams lose the ability to work independently.

Further steps:
- *Shard* the entire system.

Shard individual layers
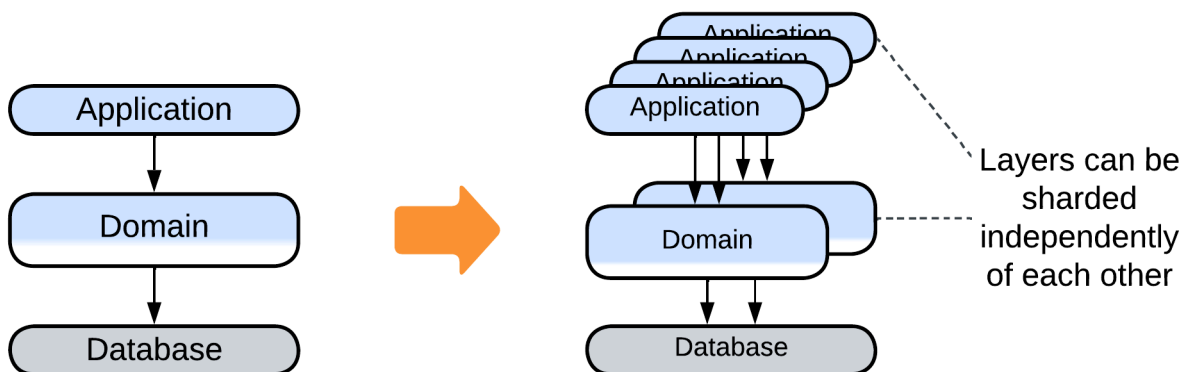
Patterns: Layers, Shards, Load Balancer (Proxy).
Goal: scale the system.
Prerequisite: some layers are stateless or limited to the data of a single client.

Multiple instances or layers can be created, with their number and deployment varying from layer to layer. That may work seamlessly if each instance of the layer that receives an event which starts a use case knows the instance of the next layer to communicate to. Otherwise you will need a *load balancer*.

Pros:
- Flexible scalability.
- Better fault tolerance.
- Co-deployment with clients.

Cons:
- More complex operations (more parts to keep an eye on).

Further steps:
- *Space-Based Architecture* scales the data layer.
- *Polyglot Persistence* improves performance of the data layer.

Use multiple databases

Each kind of use case or data type may call for a dedicated database
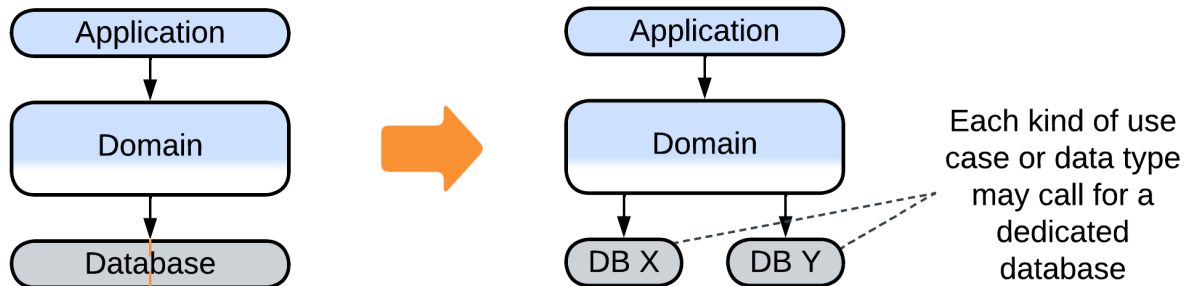
Patterns: Layers, Polyglot Persistence.
Goal: let parts of the business logic vary in qualities, improve the structure of the code.
Prerequisite: there are isolated use cases for or subsets of the data.

If you have separated commands (write requests) from queries (read requests), you can serve the queries with read-only replicas of the database while the main database is reserved for the commands.

If your types of data or data processing algorithms vary, you may deploy several specialized databases, each matching one of your needs. That lets you achieve the best performance for widely diverging cases.

Pros:
- Best performance for all the use cases.
- Specialized data processing algorithms out of the box.
- Replication may help with error recovery.

Cons:
- Someone will need to learn and administer all those databases.
- Keeping the databases consistent takes some effort and the replication delay may negatively affect UX.

Further steps:
- Serve read and write requests with different backends according to *Command-Query Responsibility Segregation (CQRS)*.
- Separate the backend into *services* which match the already separated databases.

Layers: gain flexibility

The last group of evolutions to consider is about making the system more adaptable. We have already discussed the following evolutions for *Monolith*:
- The behavior of the system may be modified with *plugins*.
- *Hexagonal Architecture* allows for abstracting the business logic from the technologies used on the project.
- *Scripts* allow for customization of the system's logic on a per client basis.

Layers with Plugins

Layers with Scripts

Layered Hexagonal Architecture

There is also a new evolution that modifies the upper (orchestration) layer:
- The orchestration layer may be split into *backends for frontends* to match the needs of several kinds of clients.

Divide the orchestration layer into backends for frontends

Each kind of a client may have a dedicated application
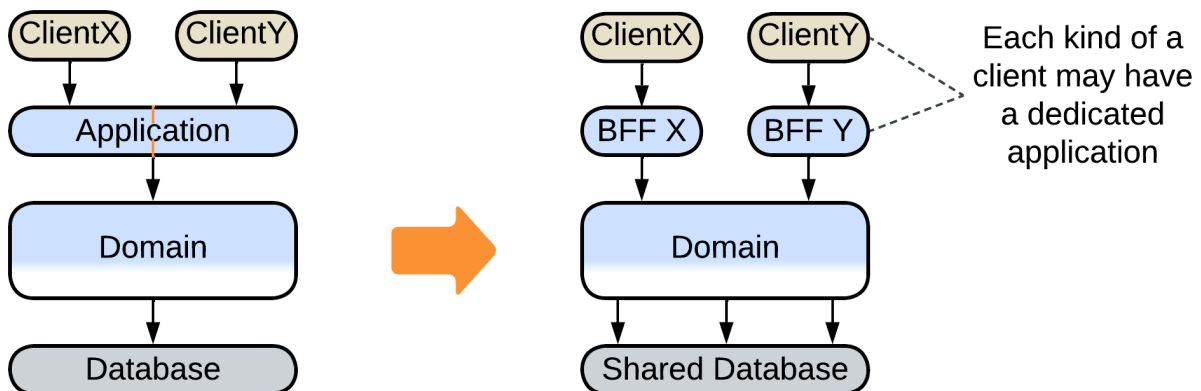
Patterns: Layers, Backends for Frontends aka BFFs.
Goal: let each kind of client get a dedicated development team.
Prerequisite: no high-level logic is shared between client types.

It is possible that your system has different kinds of users: buyers, sellers and admins; or web and mobile applications, etc. It may be easier to support a separate integration module per kind of client than to keep all the unrelated code together in the integration layer.

Pros:
- Each kind of client gets a dedicated team which may choose best fitting technologies.
- You get rid of the single large codebase of the integration layer.

Cons:
- There is no good place to share any code between the *BFFs*.
- There are more components to administer.

Services: add or remove services

Services work well when each service matches a subdomain and is developed by a single team. If those premises change, you'll need to restructure the services:

- A new feature request may emerge outside of any of the existing subdomains, creating a new *service*.
- A *service* may grow too large to be developed by a single team, calling for division.
- Two services may become so strongly coupled that fare better merged together.
- The entire system may need to be glued back into a *monolith* if the domain knowledge changes or interservice communication strongly degrades performance.

Add or split a service

Overgrown services can be split by subdomain
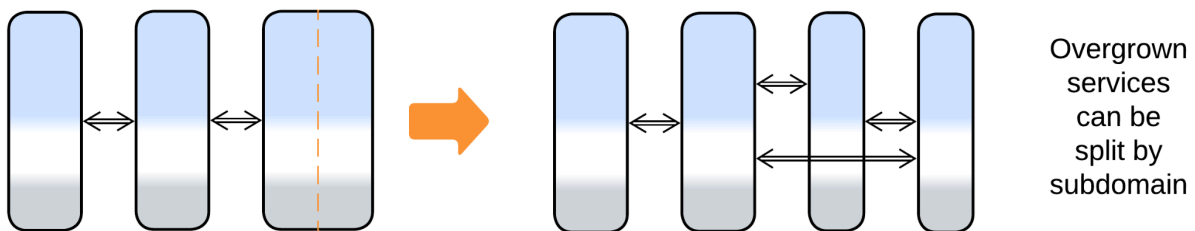
Patterns: Services.

Goal: get one more team to work on the project, decrease the size of an existing service.

Prerequisite: there is a loosely coupled (new or existing) subdomain that does not have a dedicated service (yet).

If you need to add a new functionality that does not naturally fit into one of the existing services, you may create a new service and probably get a new team for it.

If one of your services has grown too large, you should look for a way to subdivide it (probably through a *cell* stage with shared *orchestrator* and *database*) to decrease the size and, correspondingly, complexity of its code and get multiple teams to work on the resulting (sub)services. However, that makes sense only if the old service is not highly cohesive – otherwise the resulting subsystem may be more complex than the original service.

Pros:

- You get an extra development team.
- The complexity of the code decreases (if splitting) or does not increase (if adding).
- The new service is independently scalable.

Cons:

- You add to the operations complexity by creating a new system component and several inter-component dependencies.
- There is a new point of failure, which means that bugs and outages become more likely.
- Performance (or at least the efficiency) of the system will deteriorate because interservice communication is slow.
- You may have a hard time debugging use cases that spread over the old and new service.

Merge services

Coupled services can be merged
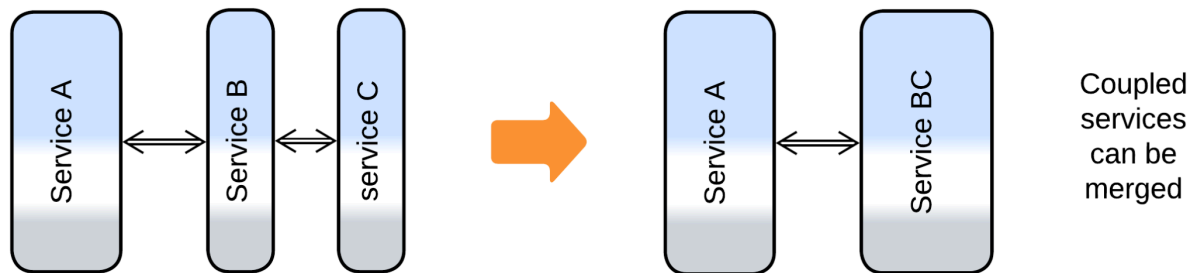
Patterns: Services, Monolith or Layers.
Goal: accept the coupling of subdomains and improve performance.
Prerequisite: the services use compatible technologies.

If you see that several services communicate to each other almost as intensely as they call their internal methods, they probably belong together.

If your use cases have too high latency or you pay too much for the CPU and traffic, the issue may originate with the interservice communication. No services, no pain.

Alternatively, if the domain knowledge changes, you may have to merge much of the code together only to subdivide it later along updated subdomain boundaries. Which means lots of work for no reason.

Pros:
- Improved performance.
- It becomes easy for parts of the merged code to access each other and share data.
- The new merged service or monolith is easier to debug.

Cons:
- The development teams become even more interdependent.
- There is no good way to vary qualities by subdomain.
- You lose granular scaling by subdomain.
- The merged codebase may be too large for comfortable development.
- If something fails, everything fails.

Services: add layers

The most common modifications of a system of services involve supplementary system-wide layers which compensate for the inability of the services to share anything among themselves:
- A *middleware* knows of all the deployed service instances. It mediates the communication between them and may manage their scaling and failure recovery.
- *Sidecars* [DDS] of a *service mesh* make a virtual layer of shared libraries for the *microservices* it hosts.
- A *shared database* simplifies the initial phases of development and interservice communication.
- *Proxies* stand between the system and its clients and take care of shared aspects that otherwise would need to be implemented by every service.
- An *orchestrator* is the single place where the high-level logic of all use cases resides.

Those layers may also be combined into *combined components*:
- *Message Bus* is a *middleware* that supports multiple protocols.

- *API Gateway* combines *Gateway* (a kind of *Proxy*) and *Orchestrator*.
- *Event Mediator* is an *orchestrating middleware*.
- *Enterprise Service Bus (ESB)* is an *orchestrating message bus*.
- *Space-Based Architecture* employs all the four layers: *Gateway*, *Orchestrator*, *Shared Repository* and *Middleware*.

Add a middleware

A middleware manages service instances and provides them with a means of communication
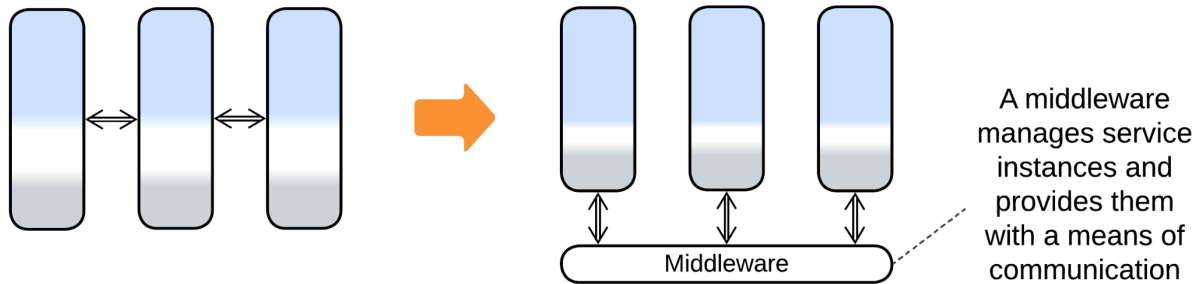
Patterns: Middleware, Services.

Goal: take care of scaling, recovery and interservice communication without programming it.

Prerequisite: the communication between the services is uniform.

Distributed systems may fail in a zillion ways. You want to ruminate neither on that nor on heisenbugs. And you probably want to have a framework for scaling the services and restarting them after failure. Get a 3rd party *middleware*! Let your programmers write the business logic, not infrastructure.

Pros:
- You don't invest your time in infrastructure.
- Scaling and error recovery is made easy.

Cons:
- There may be a performance penalty which becomes worse for uncommon patterns of communication.
- The middleware may be a single point of failure.

Further steps:
- Use a *service mesh* for dynamic scaling and as a way to implement shared aspects.

Use a service mesh

Service mesh is a distributed middleware for microservices

There is a virtual layer of sidecars - libraries colocated with each service instance

The mesh engine transfers messages between and scales the microservices
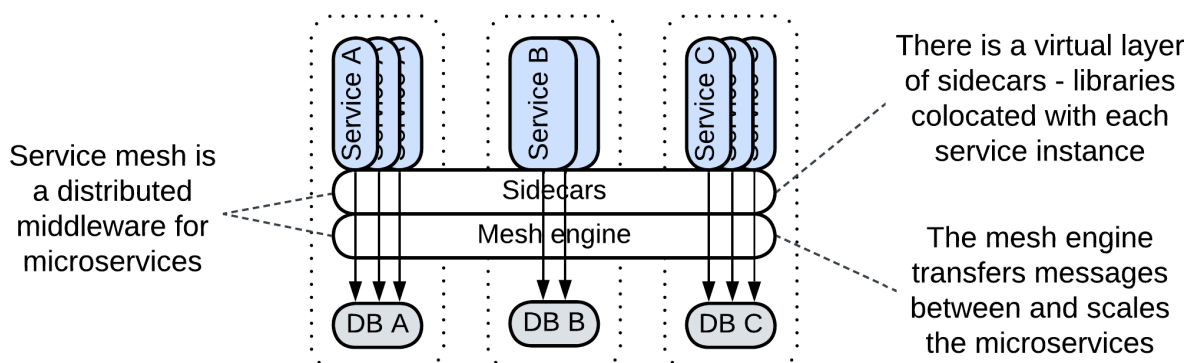
Patterns: Service Mesh (Mesh, Middleware), Proxy, Services.

Goal: support dynamic scaling and interservice communication out of the box; share libraries among the services.

Prerequisite: service instances are mostly stateless.

Microservices architecture boasts dynamic scaling under load thanks to its *mesh*-based *middleware*. It also allows for the services to share libraries in sidecars [DDS] – additional containers co-located with each service instance – to avoid duplication of generic code among the services.

Pros:
- Dynamic scaling and error recovery.
- Available out of the box.
- Provides a way to implement shared aspects (cross-cutting concerns) once and use the resulting libraries in every service.

Cons:
- Performance degrades because of the complex distributed infrastructure.
- You may suffer from vendor lock-in.

Use a shared repository

It is often easier to start a project with a single shared database
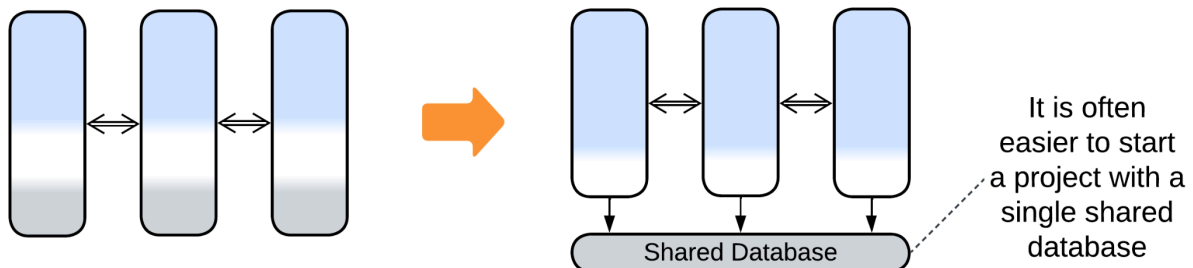
Patterns: Shared Repository, Services.

Goal: let the services share data, don't invest in operating multiple databases.

Prerequisite: the services use a uniform approach to persisting their data.

You don't really need every service to have a private database. A shared one is enough in many cases.

Pros:
- It is easy for the services to share and synchronize data.
- Lower operational complexity.

Cons:
- All the services depend on the database schema which becomes hard to alter.
- The single database will limit the performance of the system.
- It may also become a single point of failure.

Further steps:
- *Space-Based Architecture* scales the data layer but it is a simple key-value store.
- *Polyglot Persistence* uses several specialized databases.

Add a proxy

A proxy stands between the system and its clients and does something useful
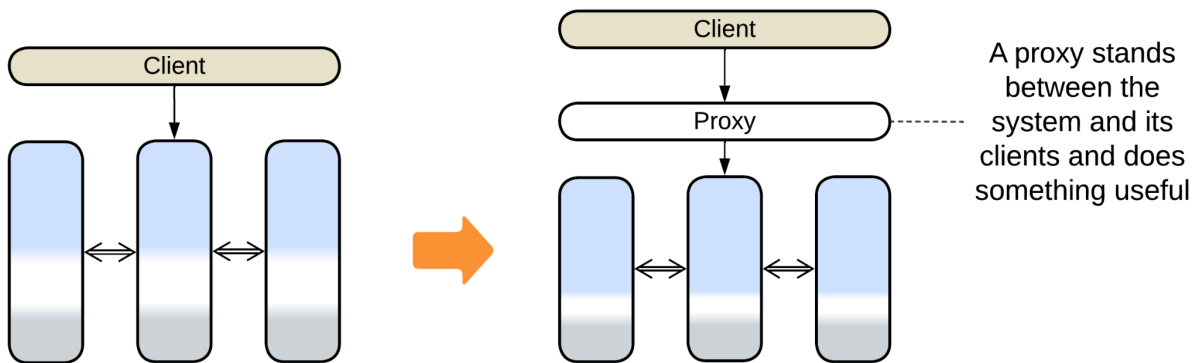
Patterns: Proxy, Services.

Goal: use a common infrastructure component on behalf of your entire system.

Prerequisite: the system serves its clients in a uniform way.

Putting a generic component between the system and its clients helps the programmers concentrate on the business logic rather than protocols, infrastructure or even security.

Pros:
- You get a select generic functionality without investing development time.
- It is an additional layer that isolates your system from both its clients and attackers.

Cons:
- There is a latency penalty caused by the extra network hop.
- Each proxy may be a single point of failure or at least needs some admin oversight.

Further steps:
- You can always add another kind of *Proxy*.
- If there are multiple clients that differ in their protocols, you can employ a stack of proxies per client, resulting in *Backends for Frontends*.

Use an orchestrator

The orchestrator uses the services to execute client requests

The services don't need to intercommunicate
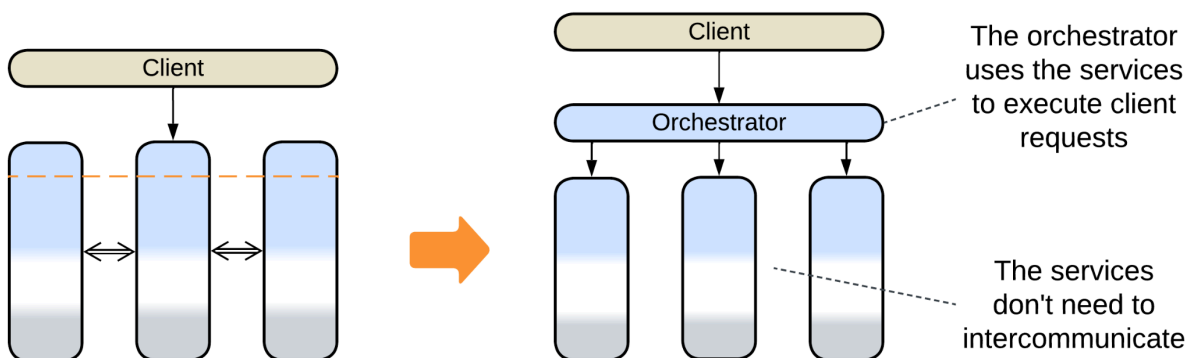
Patterns: Orchestrator, Services.

Goal: have the high-level logic of use cases distilled as intelligible code.

Prerequisite: the use cases comprise sequences of high-level steps (which is very likely to be true for a system of subdomain services).

When a use case jumps over several services in a dance of choreography, there is no easy way to understand it as there is no single place to see it in the code. It may be even worse with pipelined systems where use cases are embodied in the structure of event channels between the services.

Extract the high-level business logic from the choreographed services or their interconnections and put it into a dedicated service.

Pros:
- You are not limited in the number and complexity of the use cases anymore.
- Global use cases become much easier to debug.
- You have a new team dedicated to the interaction with the customers, so that the other teams are free to study their parts of the domain or work on improvements.
- Many changes in the high-level logic can be implemented and deployed without touching the main services.
- The extra layer decouples the main services from the system's clients and from each other.

Cons:
- There is a performance penalty because the number of messages per use case doubles.
- The orchestrator may become a single point of failure.
- Some flexibility is lost as the orchestrator couples qualities of the services.

Further steps:
- If there are several clients that strongly vary in workflows, you can apply *Backends for Frontends* with an *orchestrator* per client.
- If the orchestrator grows too large, it can be divided into layers, services or both, the latter option resulting in a *top-down hierarchy*.
- The orchestrator can be deployed as multiple instances and can have its own database.

Pipeline:

Pipeline inherits its set of evolutions from *Services*. Filters can be added, split in two, merged or replaced. Many systems employ a *middleware* (a pub/sub or pipeline framework), a *shared repository* (which may be a database or a file system) or *proxies*.

There are a couple of pipeline-specific evolutions:
- The first service of the *pipeline* can be promoted to *Front Controller* [SAHP] which tracks the status updates for every request it handles.
- Adding an *orchestrator* turns *Pipeline* into *Services*. As the high-level business logic moves to the orchestration layer, the filters don't need to interact directly, the interfilter communication channels disappear and the system becomes identical to *Orchestrated Services*.

Promote a service to Front Controller

The front controller subscribes to notifications to know the state of each running request
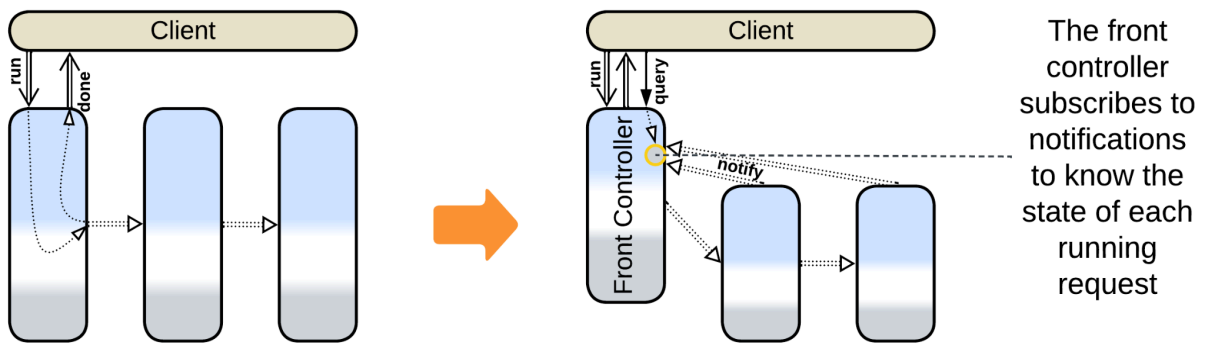
Patterns: Front Controller (Polyglot Persistence), Pipeline.

Goal: allow for clients to query the state of their requests.

Prerequisite: request processing steps are slow (usually depend on human action).

If request processing steps require heavy calculations or manual action, clients may want to query the status of their requests and analysts may like to see bottlenecks in the pipeline. Let the first service in the *pipeline* know the state of all the running requests by subscribing to status notifications from other services.

Pros:
- The state of each running request is readily available.

Cons:
- The first service in the pipeline depends on every other service.

Further steps:
- The front controller may be further promoted to *Orchestrator* if there is a need to support many complex scenarios.

Add an orchestrator

An orchestrator encapsulates the logic of use cases

The services don't need to communicate to each other directly
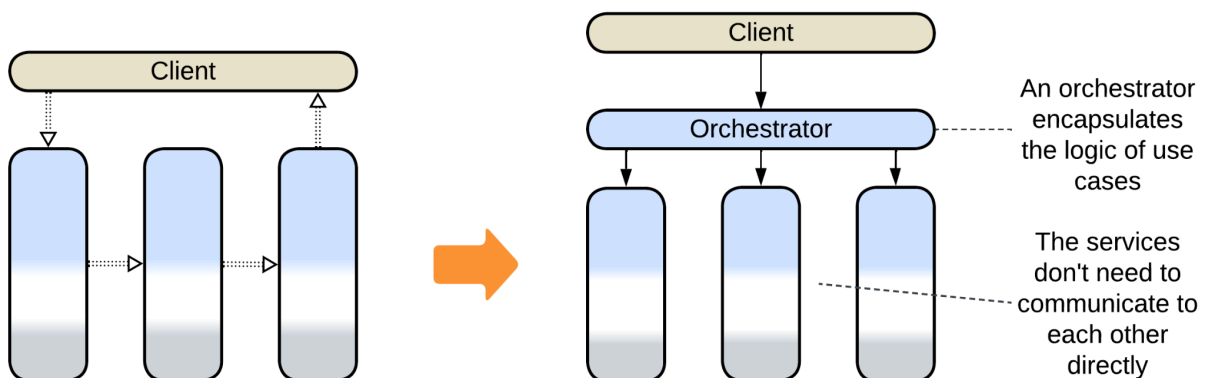
Patterns: Orchestrator, Services.

Goal: support many use cases.

Prerequisite: performance degradation is acceptable.

When an orchestrated system is extended with more and more use cases, it is very likely to fall into an integration hell when nobody understands how its components communicate. Extract the communication into a dedicated service.

Pros:
- New use cases are easy to add.
- Complex scenarios are supported.
- The domain services don't depend on each other.

- You get a single client-facing team, others are not under pressure from the business.
- It is easier to run actions in parallel.
- Global scenarios become debuggable.
- The services don't need to be redeployed when the high-level logic changes.

Cons:
- The number of messages in the system doubles, thus its performance may degrade.
- The orchestrator may become a bottleneck or a single point of failure.

Further steps:
- If there are several clients that strongly vary in workflows, you can apply *Backends for Frontends* with an *orchestrator* per client.
- If the orchestrator grows too large, it can be divided into layers, services or both, the latter option resulting in a *top-down hierarchy*.
- The orchestrator can be deployed as multiple instances and can have its own database.

Middleware:

A *middleware* is unlikely to be removed (though it may be replaced) once it is built into a system. There are few evolutions as a middleware is a 3rd party product and is unlikely to be messed with:
- If the middleware in use does not fit the preferred mode of communication between some of your services, there is an option to deploy a second specialized *middleware*.
- If several existing systems need to be merged, that is accomplished by adding yet another layer of *middleware*, resulting in a *bottom-up hierarchy (bus of buses)*.

Add a secondary middleware
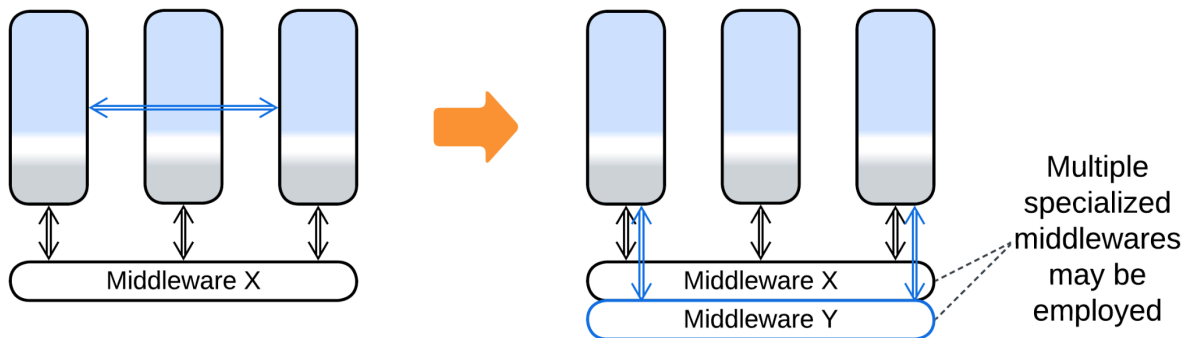
Patterns: Middleware.
Goal: support specialized communication between scaled services.
Prerequisite: the system relies on a middleware for scaling.

If the current middleware is too generic for the system's needs, you can add another one for specialized communication. The new middleware does not manage the instances of the services.

Pros:
- Supports specialized communication with no need to write code for tracking the instances of services.

Cons:

- You still need to notify the new middleware when an instance of a service is created or dies.
- There is an extra component to administer.

Merge two systems by building a bottom-up hierarchy
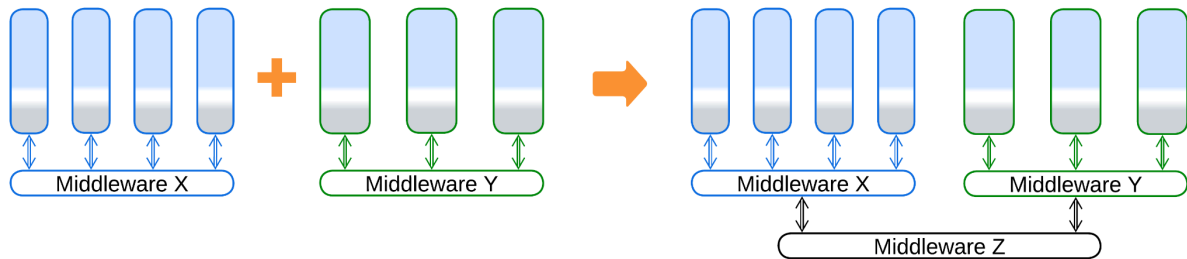
Patterns: Bottom-up Hierarchy (Hierarchy, Middleware).
Goal: integrate two systems without a heavy refactoring.
Prerequisite: both systems use middleware.
If we cannot change the way each subsystem's services use its *middleware*, we should add a new *middleware* to connect the existing *middleware*s.
Pros:
- No need to touch anything in the existing services.
Cons:
- Performance suffers from the double conversion between protocols.
- There is a new component to fail (miserably).

Shared Repository:

Once a database appears, it is unlikely to go away. I see the following evolutions to improve performance of the data layer:
- *Shard* the database.
- Use *Space-Based Architecture* for dynamic scalability.
- Divide the data into a private database per service.
- Deploy specialized databases (*Polyglot Persistence*).

Shard the database

Sometimes data can be sharded to multiple instances of the database
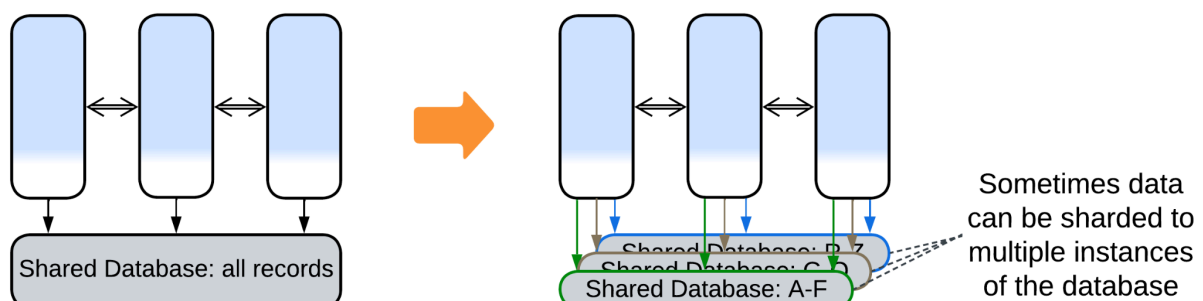
Patterns: Shards, Shared Repository.
Goal: improve performance of the database.
Prerequisite: the data is shardable (consists of independent records).

If your database is under heavy load, but the data which it contains covers independent entities (users, companies, sales) you can deploy multiple instances of the database, with a subset of the data in each instance. Your services will need to know which instance to access – probably by hashing the primary key [DDIA]. There is a good chance that you'll still need several smaller tables to be replicated to all the instances.

Modern distributed databases support sharding out of the box, but an overgrown table may still impact the performance of the database.

Pros:
● Unlimited scalability.
● You don't need to change your database vendor.
● Failure of a single database instance affects few users.

Cons:
● You need to take care of many instances of the database.
● The application or a custom script may have to synchronize shared tables among the instances.
● There is no way to do joins or run aggregate functions (such as *sum* or *count*) over multiple shards – all that logic moves to the services that use the database.

Further steps:
● *Polyglot Persistence* or *CQRS* may be used to pre-calculate aggregates to another database deployed for analytical purposes (*reporting database*).
● *Space-Based Architecture* may be cheaper as it scales dynamically. However, in its default and highly performant configuration it is prone to write collisions.

Use Space-Based Architecture

Each service is co-located with an instance of a shared in-memory database

The instances intercommunicate, forming a mesh
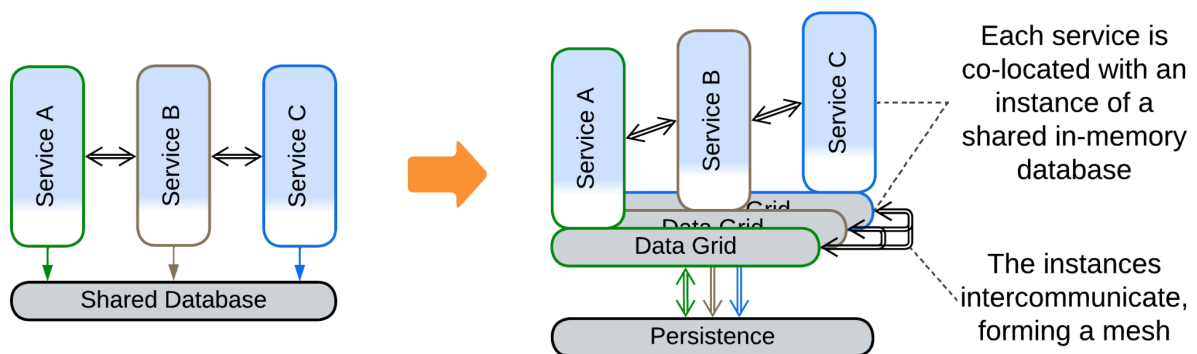
Patterns: Space-Based Architecture (Mesh, Shared Repository).
Goal: scale throughput of the database dynamically.
Prerequisite: data collisions are acceptable.

Space-Based Architecture duplicates the contents of a persistent database to a distributed in-memory cache which is co-located with the services managed by its middleware. That makes most database access operations very fast unless one needs to avoid write collisions. The mesh middleware autoscales under load both the services and the associated data cache, granting nearly perfect scalability. However, the architecture is costly because of the amount of traffic and CPU time spent on replicating the data over the mesh.

Pros:
● Nearly unlimited dynamic scalability.
● Off-the-shelf solutions are available.
● Very high fault tolerance.

Cons:
● Choose one: data collisions or poor performance.
● Low latency is guaranteed only when the entire dataset fits in the memory of a node.
● High operational cost because the nodes will send each other lots of data.
● No support for analytical queries.

Move the data to private databases of services

Services are more independent when each of them owns its data
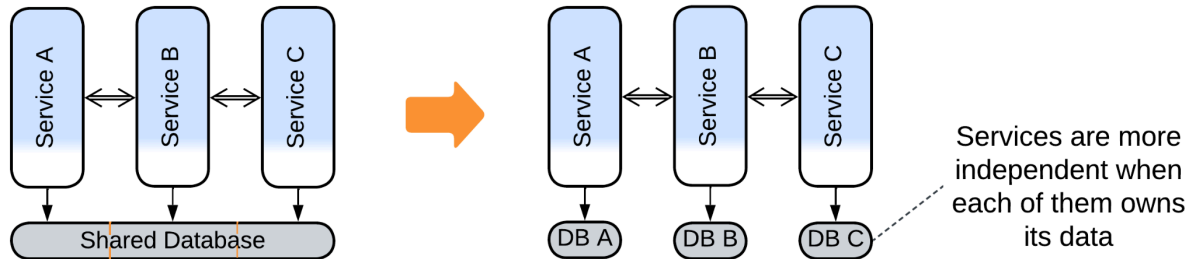
Patterns: Services or Shards, Layers.

Goal: decouple the services or shards, remove the performance bottleneck (shared database).

Prerequisite: the domain data is weakly coupled.

If the data clearly follows subdomains, it may be possible to subdivide it accordingly. The services will become choreographed (or *orchestrated* if they get an *integration layer*) instead of communicating through the shared data.

Pros:
● The services become independent in their persistence and data processing technologies.
● Performance of the data layer, which tends to limit the scalability of the system, will likely improve thanks to the use of smaller specialized databases.

Cons:
● The communication between the services and the synchronization of their data becomes a major issue.
● Joins of the data from different subdomains will not be available.
● Costs are likely to increase because of data transfer and duplication between the services.
● You will have to administrate multiple databases.

Further steps:
● *Materialized views* [DDIA] or a *query service* [MP] help a service access and join data which is owned by other services.

Deploy specialized databases

Specialized databases have good performance if used appropriately
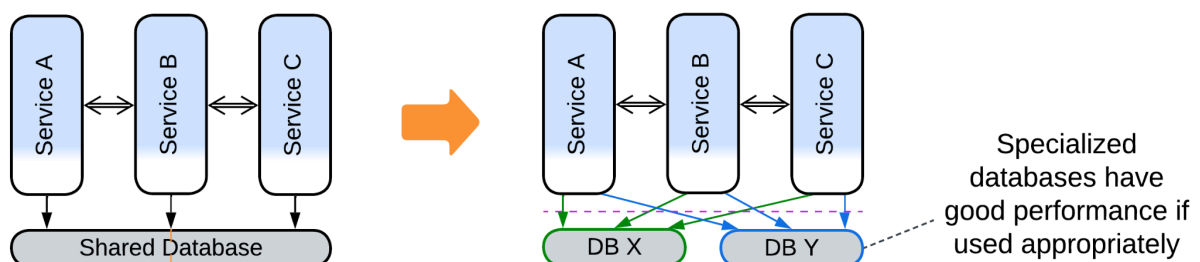
Patterns: Polyglot Persistence.

Goal: improve performance of the data layer.

Prerequisite: there are diverse data types or patterns of data access.

It is very likely that you can either use specialized databases for various data types or deploy read-only replicas of your data for analytics.

Pros:

- You can choose one of the many specialized databases available on the market.
- There is a good chance to significantly improve performance.

Cons:

- It may take quite an effort to learn the new technologies to use them efficiently in your system.
- Someone needs to see to the new database(s).
- You'll likely need to work around the *replication lag* [MP].

Proxy:

It usually makes little sense to get rid of a *proxy* once it is integrated into a system. Its only real drawback is a slight increase in latency for user requests which may be helped through creation of bypass channels between the clients and a service that needs low latency. The other drawback of the pattern, the proxy's being a single point of failure, is countered by deploying multiple instances of the proxy.

As proxies are usually 3rd party products, there is very little we can change about them:

- We can add another kind of a *proxy* on top of the existing one.
- We can use a stack of proxies per client, making *Backends for Frontends*.

Add another proxy

Proxies are independent and can be stacked
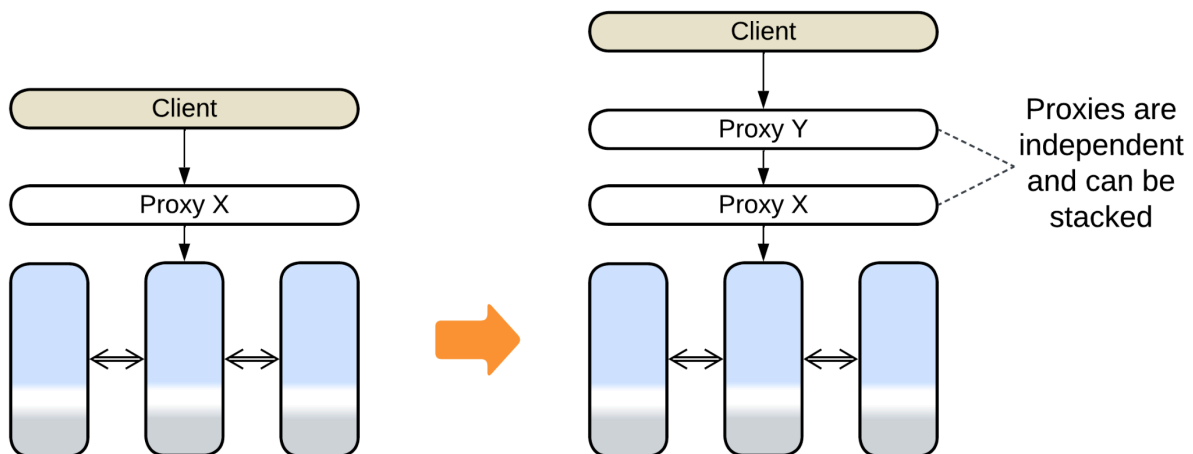
Patterns: Proxy, Layers.

Goal: avoid implementing generic functionality.

Prerequisite: you don't have this kind of a proxy yet.

A system is not limited to a single kind of proxies. As a proxy represents your system without changing its function, proxies are transparent, thus they are stackable.

It often makes sense to colocate software proxies or use a multifunctional proxy to reduce the number of network hops between the clients and the system. However, in a highly loaded system proxies may be resource-hungry, thus in some cases colocation strikes back.

Pros:

- You get another aspect of your system implemented for you.

Cons:

- Latency degrades.
- More work for admins.
- Another point of possible failure.

Deploy a proxy per client type
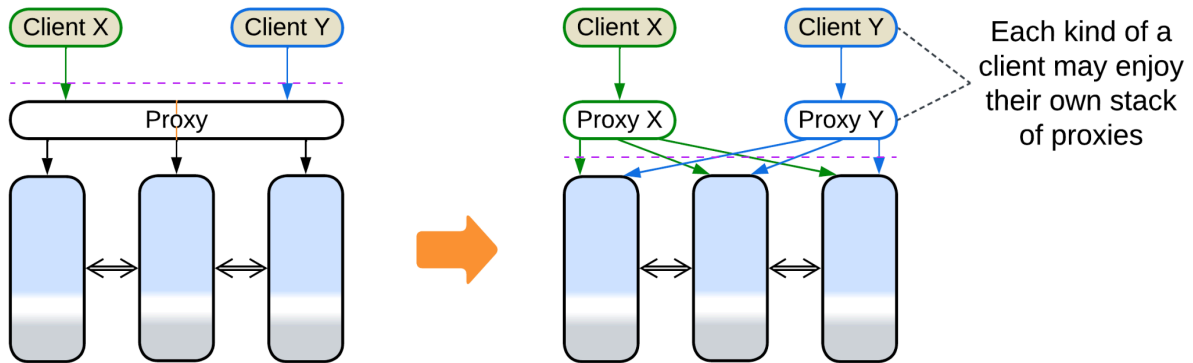
Patterns: Proxy, Backends for Frontends.

Goal: let the aspects of communication vary for different kinds of clients.

Prerequisite: your system serves several kinds of clients.

If you have internal and external clients, or admins and users, you may want to differ the setup of proxies for each kind of client, sometimes to the extent of physically separating the network communication paths, so that each kind of client is treated according to its bandwidth, priority and permissions.

Pros:

- It is easy to set up various aspects of communication for a group of clients.

Cons:

- More work for admins as the proxies are duplicated.

Orchestrator:

Employing an *orchestrator* has two pitfalls:

- The system becomes slower because too much communication is involved.
- The single *orchestrator* may be found too large and rigid.

There is a handful of evolutions to counter those weaknesses:

- Subdivide the *orchestrator* by the system's subdomains, forming *Layered Services*.
- Subdivide the *orchestrator* by the type of client, forming *Backends for Frontends*.
- Add another *layer* of orchestration.
- Build a *top-down hierarchy*.

Subdivide to form layered services

An overgrown orchestrator may be divided to form an upper layer of services
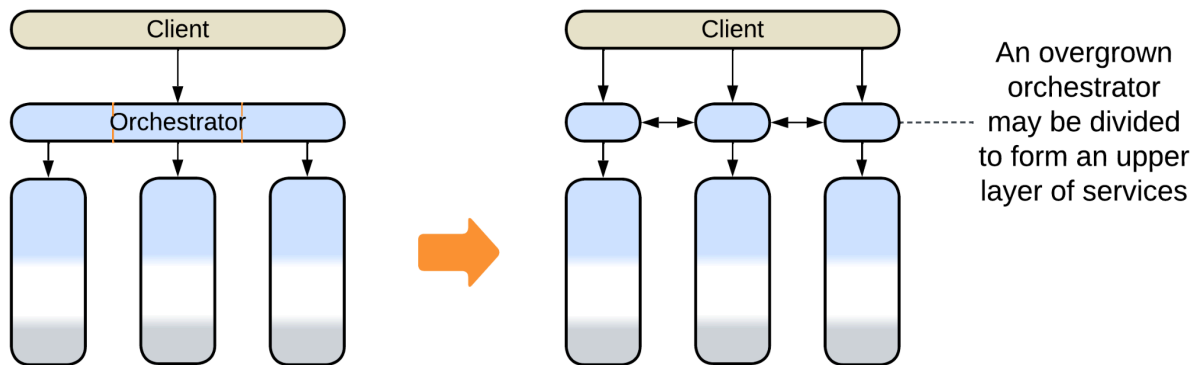
Patterns: Three-Layered Services (Layered Services, Services, Layers).

Goal: simplify the orchestrator, let the service teams own the orchestration, decouple forces for the services, improve performance.

Prerequisite: the high-level (orchestration) logic is weakly coupled between the subdomains.

If the orchestration logic mostly follows subdomains, it may be possible to subdivide it accordingly. Each service gets a part of the orchestrator that mostly deals with its subdomain but may call other services when needed. As a result, each service orchestrates every other service. Still, a large part of the orchestration becomes internal to the service, meaning that fewer calls over the network are involved.

Pros:
- You subdivide the large orchestrator codebase.
- Performance is improved.
- The services become more independent in their quality attributes.

Cons:
- You lose the client-facing orchestration team – now each service's team will need to face its clients.
- The service teams become interdependent (while having equal rights), often causing slow development and suboptimal decisions.
- There is no way to share code between different use cases or even take a look at all of them at once.

Further steps:
- *Materialized views* [DDIA] or a *query service* [MP] help a service access and join the data owned by other services, further reducing the need for interservice communication.

Subdivide to form backends for frontends

Each kind of a client may have their own orchestrator
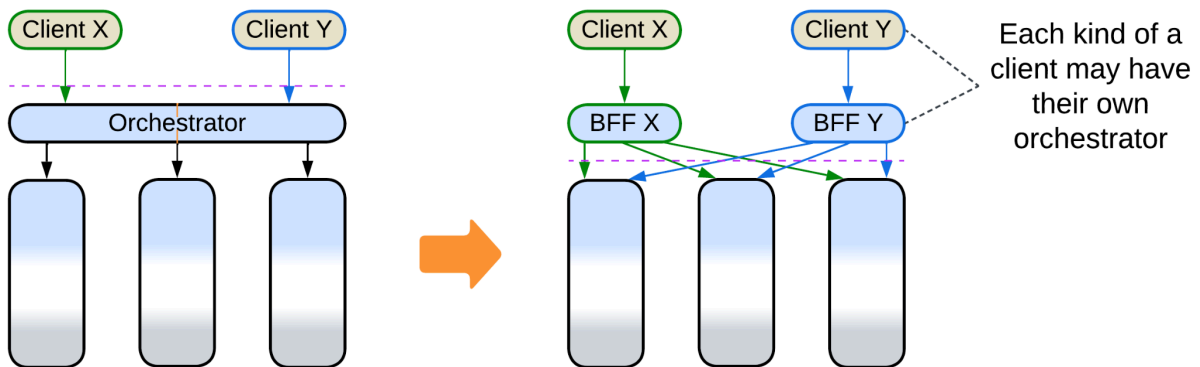
Patterns: Backends for Frontends, Orchestrator.

Goal: simplify the orchestrator, employ a team per client type, decouple qualities for clients.

Prerequisite: clients vary in workflows and forces.

When use cases for clients vary, it makes sense for each kind of a client to have a dedicated orchestrator.

Pros:
- The smaller orchestrators are independent in qualities, technologies and teams.
- The smaller orchestrators are … well, smaller.

Cons:
- There is no good way to share code between the orchestrators.

Further steps:
- You may want to add client-specific *proxies* and, maybe, co-locate them with the orchestrators to avoid the extra network hop.
- Adding another shared *orchestrator* below the ones dedicated to clients creates a place for sharing functionality among the orchestrators.
- If you are running *microservices* over a *service mesh*, *sidecars* [DDS] may help to share generic code among the orchestrators.

Add a layer of orchestration

Simple scenarios may be handled by the high-level orchestrator ...

... while complex requests are forwarded to another one
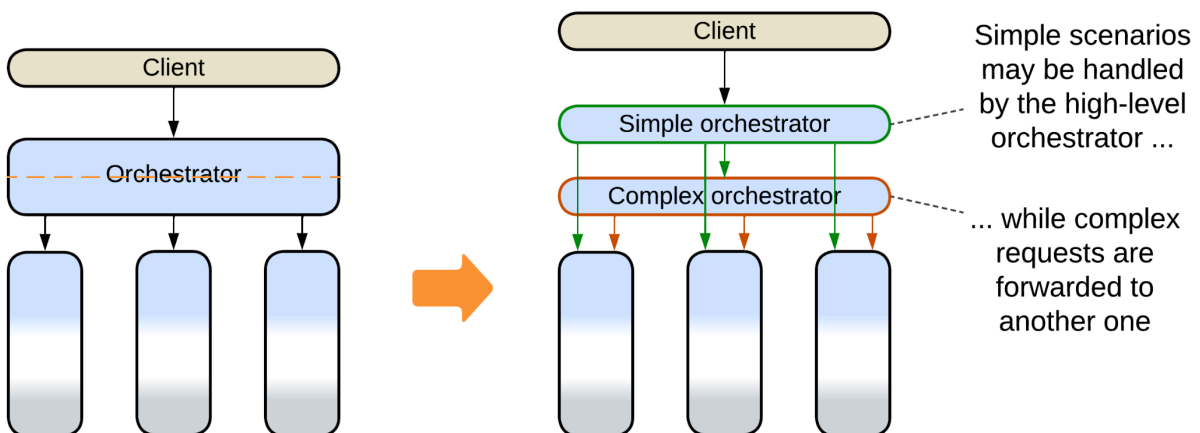
Patterns: Orchestrator, Layers.

Goal: implement simple use cases quickly, while still supporting complex ones.

Prerequisite: use cases vary in complexity.

You may use two or three orchestration frameworks that differ in complexity. A simple declarative tool may be enough for the majority of user requests, reverting to custom-tailored code for rare complex cases.

Pros:
- Simple scenarios are easy to write.
- You retain good flexibility with hand-written code when it is needed.

Cons:
- Requires learning multiple technologies.
- More components mean more failures and more administration.
- Performance of complex requests may suffer from more indirection.

Further steps:
- Divide one or more of the orchestration layers to form *Layered Services*, *Backends for Frontends* or *Hierarchy*.

Form a hierarchy

Some domains allow for hierarchical solutions
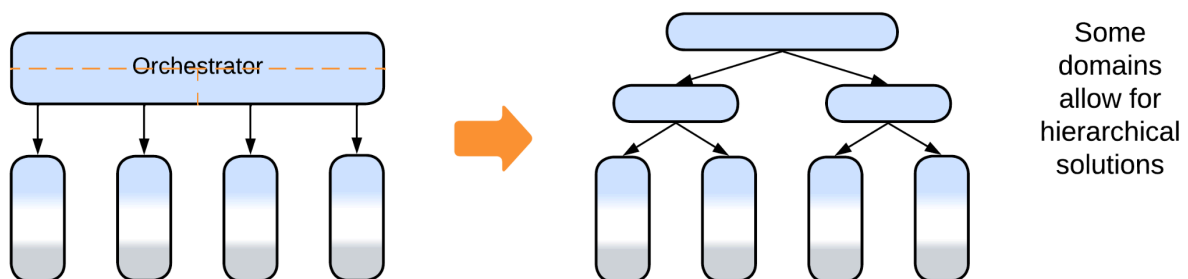
Patterns: Top-Down Hierarchy (Hierarchy).
Goal: simplify the orchestrator and, if possible, the services.
Prerequisite: the domain is hierarchical.

If an *orchestrator* becomes too complex, some domains (e.g. IIoT or telecom) encourage using a tree of *orchestrators*, with each layer taking care of one aspect of the domain, serving the most generic functionality at the root.

Pros:
- Multiple specialized teams and technologies.
- Small code base per team.
- Reasonable testability.
- Some decoupling of quality attributes.

Cons:
- Hard to debug.
- Poor latency unless several layers of the *hierarchy* are colocated.

Combined Component:

The patterns that involve *orchestration* (*API Gateway*, *Event Mediator*, *Enterprise Service Bus*) may allow for most of the evolutions of the *Orchestrator* metapattern by deploying multiple versions of the component. There is also a special evolution:
- Replace the combined component with several specialized ones

Divide into specialized layers
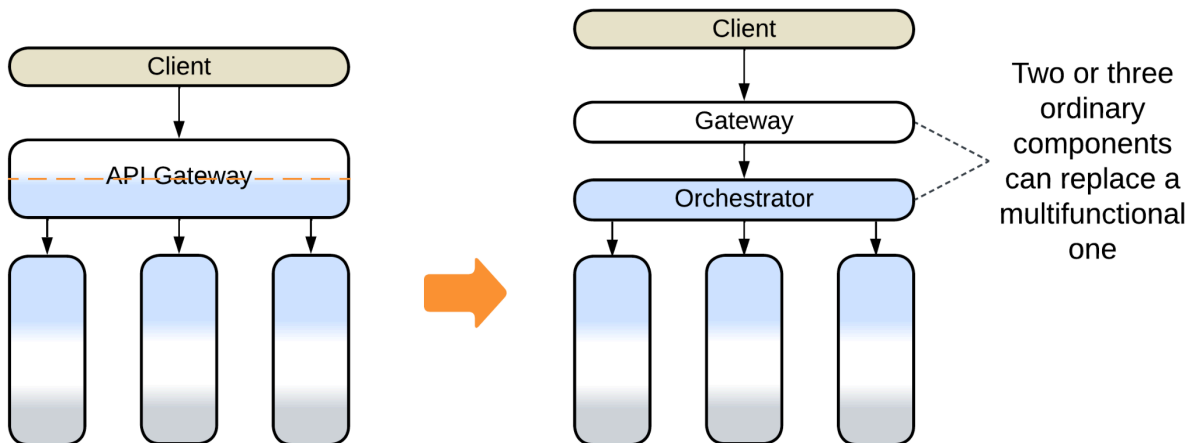
Patterns: Layers.
Goal: break out of *vendor lock-in* [DDD], gain flexibility.
Prerequisite: you have lots of free time.

If you feel that the *combined component* your system relies on does not cover all your needs or is too expensive or unstable then you may want to get rid of it by replacing it with generic single-purpose tools or a homebrewed implementation that will always adapt to your needs.

Pros:
● It's free.
● You'll own your code.
● Anything you write will fit your needs for as long as you spend time supporting it.

Cons:
● Takes lots of work.
● Performance may become worse because there will be more components on the requests' path and also because the industry-grade framework that you used could have been highly optimized.

Appendix F. Format of a metapattern.

The descriptions of most metapatterns follow the same format:

Diagram

The structural diagram (in *abstractness-subdomain-sharding* coordinates) of a typical application of the metapattern. Please note that in practice the number and types of components and their interactions may vary:

- Even though most diagrams show 3 *layers* or *services*, there are many 2-layered or 4-layered systems, while the number of services may often be greater than 10.
- Extension metapatterns add a *layer* (or a *layer of services*) to an existing system, which is shown as *services*, but may instead comprise *shards*, *layers* or even a *monolith*.
- There are several kinds of *Hierarchy* or *Mesh* that differ in their topologies. Only one is shown.
- The components of metapatterns may communicate in various ways, including in-process calls, RPC or asynchronous messaging. Only one of them is shown. Optional communication pathways may appear as dashed arrows.

Abstract

Motto and the design goal.

Known as: the list of aliases for the basic metapattern.

Aspects: an optional list of roles the component may have.

Variants: one or more lists of special or notable architectures or patterns that derive from or implement the current metapattern.

Structure: a one-line description of the structure of the metapattern.

Type: Root, main, extension or implementation.

- The root of all the metapatterns is *Monolith*, as any system both looks monolithic to its clients and comes about through division of the continuous (monolithic) design space.
- Main metapatterns (*Layers*, *Services* and few others derived from them) stay at the core of any architecture.
- An extension adds components to an architecture, built around a main metapattern, to modify its properties.
- An implementation metapattern shows the internal structure of a component that is usually treated as monolithic. Many of them implement *Middleware*.

A short *table of benefits and drawbacks*.

References: some articles and books that relate to the topic.

Then follow two or three paragraphs of main facts and ideas about the metapattern.

Performance

This section discusses performance of the metapattern in scenarios of various extent: usually simple requests or events, that relate to a single subsystem, are processed way faster than those that touch multiple components.

There are two kinds of performance: latency and throughput. Low latency is possible only if few components are involved as inter-component communication, especially in distributed systems, increases latency. Contrariwise, throughput depends on the number of components that work in parallel, thus it scales together with the system.

The section may also discuss optimization techniques that apply to the metapattern.

Dependencies

Some components of the metapattern depend on others of its components. If a component changes, everything that depends on it may need to be re-tested with the updated version. If a component's interface changes, all the components that depend on it must be updated. Thus, components that quickly evolve should depend on others, not the way around.

Some patterns, like *Hexagonal Architecture*, use adapters to break dependencies. An adapter depends on components on both its sides, making the components themselves independent of each other. The adapters are small enough to update quickly and may easily be replaced with stubs for testing or running a component in isolation.

Applicability

Here follows a list of kinds of projects that may benefit from applying the architecture under review, and another list of those which it is more likely to hurt.

Relations

An optional sequence of diagrams, showing the (extension or implementation) metapattern applied to various kinds of architectures, followed by a list of relations between the current and other metapatterns.

Variants and examples

A metapattern usually unites many variations of several patterns. Here we have a section per dimension of variability and a section for well-known variants of the pattern.

On some occasions I had to include several variants that do not properly belong to the metapattern under review, just to avoid confusion with terminology and point the reader to the correct chapter. For example, *Modular Monolith* has a module per subdomain, thus it belongs to *Services* rather than *Monolith*. Still, when the chapter on *Monolith* did not mention it, I was blamed for misunderstanding the monolithic architecture. Such patterns are marked as (misapplied) or (inexact).

I tried to show the difference between synonymous names for each variant or example whenever I could identify one.

Evolutions

A brief summary of possible changes to the architecture under review. Each change leads to a new architecture which usually represents another metapattern.

Appendix E discusses many evolutions in greater detail:

- A diagram that shows the original and resulting structure.
- The list of patterns, present in the resulting architecture. More general forms of each pattern are given in parentheses, i.e. Pattern (Metapattern (Parent Metapattern)).
- The goal(s) of the transition.
- The prerequisites that enable the change.
- A short description of the change and the resulting system.
- Lists of pros and cons of the evolution.
- An optional list of metapatterns that the resulting system may evolve into and their benefits in the context of the current evolution.

Appendix G. Glossary.

Abstractness – the scope of information that a *concept* operates. Highly abstract *components* describe the *system's* behavior in less words.

Action – an act of a *system* that changes its environment.

API (*application programming interface*) – a set of *methods* or *messages* that a *component* provides to its clients.

Application – the most *abstract* layer that usually *integrates components* of a less *abstract* layer.

Architectural pattern – a way to structure a *system* or a part of a *system* to achieve desirable properties (address a set of *forces*).

Architectural style – see *architecture*.

Architecture – the structure of a system. It comprises *components* and their *interactions*.

ASS diagram – a *structural diagram* with *abstraction*, *subdomain*, *sharding* for coordinates.

Asynchronous communication – the mode of *communication* when the sender of the *request message* does not stop the execution of the scenario to wait for the confirmation message.

Attack surface – the amount of *components* and functionality that faces an external network (potentially exposed to hackers).

Availability – the percentage of time that the *system* is operational (satisfies its *users*).

Bounded context – a subset of *requirements* and code that shares a set of *concepts*. Usually consists of internals of a *component* and *APIs* of all the other *components* it uses.

Business logic – the thing that *users* pay for. It is the heart of the business and is usually the largest part of a *project*. You cannot buy the *business logic*, only *implement* it.

Choreography – a kind of *workflow* in which *components* at the same *abstraction* level cooperate to implement a *use case*.

Client – an external *component* or *system* that makes use of a *component* or *system* in question.

Cohesion – the density of logical connections among entities inside a *component*.

Colocated – running in the same address space (process) on the same hardware.

Communication – transfer of data or signals in a *system*.

Complexity – the cognitive load caused by the quantity of entities (*concepts* or *modules*) and their relations that a programmer needs to operate.

Component – an encapsulated part of a *system*. It exposes an *API* to the system's *clients* and/or other *components* of the *system*.

Concept – a notion of an element of a *system's* behavior, usually present in *requirements*.

Contract – the informal rules of the behavior of a *component* expected by its *clients*.

Control – a kind of system that supervises physical entities or external programs.

Coupling – the density of logical connections between *components*.

Cross-cutting concern – a functionality that should be present in multiple *components*.

Debugging – trying to force the code to behave correctly from a user's point of view.

Design – the planning for the best way to write code.

Design – see *architecture*.

Design space – the multitude of possible ways to *design* a given project.

Development – running a *project* for its *users*. Usually involves intermixed *design*, *implementation*, *debugging* and *testing* phases.

Development complexity – the *complexity* of internals of a *component* and *APIs* they reference – the contents of a *bounded context*.

Distributed – spread over multiple computers that *communicate* via a network.

Domain – the whole of knowledge (including *requirements*) that is needed to build a *system*.

Domain – the middle layer of a *system* that contains the bulk of its *business logic*.

Event – a signal that has some meaning for a *system* or a *component*. *Events* may carry data.

Fault tolerance – the ability of a *system* to remain (at least partially) operational if one or more of its components fail (become inaccessible due to a hang, crash or a hardware failure).

Forces – expected properties of a system (such as its stability or response time) which are crucial for the system to be built, deployed and used successfully.

Functional requirements – the *requirements* that describe *inputs* and *outputs* of a *system*, but not its speed or stability.

Global use case – a use case that involves most of the *components* of a *system*. Such *scenarios* are deeply affected by the *system's* structure.

Implementation – the process of writing code.

Infrastructure – the lowest layer of a *system* that provides general-purpose functionality (tools) to its upper layers.

Input – *events* or data that a *system* reacts to.

Integration – see *orchestration*.

Integration complexity – the *complexity* of understanding how individual *components* *interact* to make a system.

Interactions – the kinds and routes of *communication* among *components* of a system.

Interface – see *API*.

Latency – the delay between a *system's* receiving input and producing a corresponding output.

Messaging – communication via sending short pieces of data.

Method call – invocation of an interface method (or procedure) of a component by another component.

Metapattern – a cluster of *patterns* that have similar *structural diagrams* and address related issues.

Module – a *colocated component*.

Non-functional requirements (NFRs) – see *forces*.

Notification – an *event* that one *component* sends to another *component*(s) to inform them of a change.

Operational complexity – see *integration complexity*.

Orchestration – a kind of *workflow* where a single dedicated *component* (*orchestrator*) makes use of (usually multiple) less *abstract components*. *Facade* [GoF] is a good example.

Output – *actions* or data that a *system* produces.

Pattern – a documented approach (blueprint) for solving a recurrent programming issue.

Pattern Language – a set of interrelated *patterns* intended to cover most aspects of *designing systems* in a target domain.

Performance – the ratio or the *system's* speed to the *resources* it consumes.

Processing – transformation of input data into output data.

Project – the process of making a *system*.

Pub/sub (publish/subscribe) – a mode of *communication* when one *component* (*subscriber*) receives a subset of *notifications* from another *component* (*publisher*). It is the *subscriber* that chooses which *notifications* it is interested in.

Qualities – the properties a *component* or (sub)*system* manifests to satisfy the *forces*.

Real-time – a *force* that requires the *system* to respond to incoming *events* immediately.

Request/confirm – a pair of *messages* between two *components* (Requestor and Executor). The request describes the action that the requester *component* wants the executor *component* to run (R => E). The confirm describes the results of the execution (R <= E).

Requirements – a set of rules that describes the correct (expected) behavior of the *system*.

Resources – CPU, memory, network bandwidth and other stuff that costs money.

Scaling – ability to increase *performance* of a system by providing it with more *resources*.

Scenario – see *use case*.

Service – a *distributed* component.

Sharding – deploying multiple instances of a *component*.

Single point of failure – a software or hardware *component* which if fails makes the whole *system* non-operational. High-*availability systems* should avoid *single points of failure*.

SPI (*service provider interface*) – a set of *methods* or *messages* that a *component* expects to be supported by the *components* it uses.

State – data that a component keeps between processing its *inputs*.

Structural complexity – see *development complexity*.

Structural diagram – a graphical representation of the structure of a (sub-)system that shows components and their interactions.

Stub – a very simple *implementation* of a *module* that allows other *components* that use it to run without starting the original *module*. *Stubs* are used to *implement modules* concurrently or test them in isolation.

Subdomain – a distinct *cohesive* part of the *domain* knowledge.

Synchronous communication – the mode of communication when the requesting *component* waits for the results of its *request* to another *component* before continuing to run its *task*.

System – a self-sufficient set of *communicating components* that were brought together or *implemented* to satisfy its *users* (by running *use cases*).

Task – a high-level sequence of execution steps. Similar to *use case* or *scenario*.

Team – few programmers and testers that work on a *component*. Teams of more than 5 members lose productivity to communication overhead.

Testing – checking how satisfactory the system behaves.

Throughput – the amount of data a system can *process* per unit of time.

Use case – a behavior expected by *system's* users. A *system* is *implemented* to run *use cases*.

User – a human that uses a *system* and usually pays well if satisfied with its behavior.

Vendor lock-in – a pitfall when a *system* relies on an external provider so much that it is impossible to change the provider. It is similar to falling prey to a monopoly.

Workflow – a sequence of actions (*messages* or *method calls*) required to *implement* a *use case*.

Appendix H. History of changes.

0.1 (2020) – Description of my semisynchronous proactor architecture for a VoIP gateway, published by dou.ua. Got very positive feedback and lots of comments from the community.

0.2 (2020) – The same in a more official style for the PLoP'20 conference.

0.3 (2021) – Comparison of choreography and orchestration for dou.ua. No impact.

0.4 (2022) – A series of 5 articles that looked into local and distributed architectures by applying the actor model. Positive feedback at dou.ua, but the series was interrupted by the war.

0.5 (2023) – The same series in English, published by ITNEXT and upvoted by r/softwarearchitecture.

0.6 (2023) – I attempted to rebuild the series for InfoQ but the first article was rejected as impractical (technology-agnostic).

0.7 (09-2024) – Chapters from this book, published by ITNEXT. Some of them got boosted by Medium.

0.8 (11-2024) – The complete book as a pdf. Clients were changed to mid-brown. Detailed evolutions were moved to the appendix. Rejected by Manning (the free license and color diagrams make the book unprofitable) and O'Reilly (it would get in the way of their bestsellers).

0.9 (12-2024) – Integrated patterns from [DDS, LDDD, SAHP] and Internet sources, mostly affecting Shards, Pipeline, Proxy, Orchestrator and Hexagonal Architecture. Added diagrams for Polyglot Persistence with derived storage and detailed evolutions for Pipeline. Downgraded analytical chapters to sections and added a couple of new ones. Extended the ambiguous patterns chapter. Improved the structure of the variants sections of metapatterns: now each synonym has a short description. Fixed alignment of text and figures.

Appendix I. Index of patterns and architectures.