A Deep Dive into Capability-Oriented Software Architecture

1. Introduction

The original objective of this project was straightforward: to build a portable game engine. At the time, it seemed reasonable to assume that the primary challenges would be related to rendering, asset management, simulation, or platform support. Instead, a different pattern emerged. As development progressed, an increasing amount of time was spent solving problems surrounding the engine rather than problems within the engine itself.

Building a portable engine required a portable build system. Reusing functionality across projects required a package architecture. Integrating build functionality into editors and other tools required a common execution model. Sharing functionality between tools written in different languages required a language-independent communication mechanism. Eventually, distributing and executing third-party extensions raised questions of trust, security, and responsibility.

None of these systems were originally planned as part of a larger platform. They emerged gradually as solutions to practical problems encountered during the development of increasingly complex software. Over time, however, it became apparent that these solutions shared a common direction. Despite addressing different challenges, they consistently converged towards the same characteristics: open source development, small components with clearly defined responsibilities, and collaboration between independent tools rather than monolithic applications.

The goal was never to build a development platform. The platform emerged as the accumulated solution to a growing collection of practical problems encountered while building larger systems. Several ideas developed during this process. The build system evolved from a standalone command-line utility into a reusable execution environment. The package architecture introduced a structured approach to sharing functionality across projects. Decentralized Method Execution emerged from a simple observation: functionality should not need to be rewritten merely because it exists in a different process, a different programming language, or a different tool. If a capability already exists somewhere within the ecosystem, it should be possible to access and reuse it directly.

As the ecosystem continued to grow, another challenge became increasingly important. Open collaboration enables innovation, but it also raises a fundamental question: how can software safely execute code that was not written by its original author? Traditional approaches often rely on trust, review processes, or social governance. While valuable, these mechanisms alone are insufficient for a highly modular and decentralized environment, and introducing extendability leads to unsafe and probably malicious code executed in a trusted environment sooner or later.

This observation ultimately led to an architectural model inspired by two established systems. From the Linux kernel

comes the concept of a minimal runtime core extended through independently developed modules. From Flatpak comes the idea that extensibility and security should not be treated as opposing goals. Modules should be easy to distribute, discover, and integrate, while the runtime environment remains responsible for defining boundaries, controlling capabilities, and enforcing security policies.

The resulting architecture treats software development as a collaborative ecosystem of small, reusable, and independently evolving components. Tools become modules. Packages become capabilities. Communication becomes data-driven. Responsibility is isolated into atomic units. Complex systems emerge through composition rather than centralization.

This paper describes the next stage of that evolution. It outlines the transition from a collection of specialized tools towards a unified development environment built around modular execution, capability-oriented security, decentralized collaboration, and open-source development. Rather than presenting a theoretical framework, it documents the practical lessons, architectural decisions, and design principles that emerged throughout this journey

2. Purpose and Design Philosophy

The SDK originated from a practical need rather than a predefined architectural vision. While its development began as part of a larger effort to create a portable game engine, many of the challenges encountered during that process were not directly related to engine development itself. Instead, they emerged from the surrounding ecosystem: build systems, dependency management, interoperability between tools, code reuse, and the long-term maintenance of increasingly complex software projects.

Over time, individual solutions were developed to address these challenges. What initially appeared as separate tools and utilities gradually revealed a common direction. Similar design decisions repeatedly emerged across different parts of the ecosystem, eventually forming a consistent architectural philosophy that now serves as the foundation of the SDK.

This philosophy is built upon a small number of fundamental principles which influence every aspect of the system

2.1 Central Installation

The SDK is designed as a centrally installed environment. Rather than embedding complete development infrastructures into individual projects, tools, packages, modules, and supporting resources are maintained within a shared environment that can be accessed from anywhere on a system. Projects consume capabilities provided by the environment rather than carrying independent copies of those capabilities.

This approach reduces duplication, simplifies maintenance, and establishes a common foundation upon which multiple projects can coexist and evolve

2.2 Global Applicability

The SDK is intended to support a wide range of software projects without being tied to a particular domain, framework, or application type. The same architectural principles should apply equally to libraries, tools, services, desktop applications,

games, and future forms of software that may emerge over time.

Likewise, the system is not intended to be restricted to a single implementation language. While C# serves as the foundational language of the SDK itself, the broader architectural model is designed to enable collaboration between components regardless of their implementation language.

The objective is not language independence as an isolated goal, but the ability to reuse functionality wherever it exists

2.3 Open Source

The SDK assumes an open and collaborative development model. Software should be inspectable, understandable, and extensible. Contributors should be able to improve individual parts of the ecosystem without requiring ownership of the system as a whole.

Open source is therefore treated not merely as a distribution model but as a technical requirement for long-term maintainability, transparency, and collaborative development. The architecture encourages independent contributions while maintaining a coherent and predictable system structure

2.4 Convention over Configuration

Wherever possible, information should be derived from existing context rather than manually declared. Structure, naming, relationships, and established conventions often provide sufficient information to infer intent without requiring extensive configuration files or manual setup procedures.

Configuration remains an important part of the system, particularly where explicit decisions must be made. However, configuration is treated as a mechanism for expressing exceptions and special cases rather than the primary means of describing software. The preferred approach is always to derive information from convention before requiring explicit configuration

2.5 Atomic Responsibility

Every component should have a clearly defined and narrowly scoped purpose. Responsibilities should remain isolated, understandable, and independently maintainable. Larger systems should emerge through composition rather than through the continuous expansion of individual components.

This principle applies consistently throughout the SDK, from low-level code structures to higher-level architectural constructs. By limiting the scope of responsibility assigned to individual components, systems become easier to reason about, easier to maintain, and easier to reuse in different contexts

2.6 Composition and Collaboration

Complex systems are built from the interaction of many smaller systems. The SDK therefore emphasizes composition over monolithic design. Components should be capable of evolving independently while remaining able to participate in larger structures through clearly defined interfaces and shared conventions.

Collaboration is not limited to source code contributors. Tools, services, applications, modules, and distributed systems should all

be capable of cooperating through the same underlying principles.

System boundaries, process boundaries, language boundaries, and network boundaries may influence how communication occurs, but they should not fundamentally change the ability of components to collaborate.

The ability to compose independent parts into larger systems is one of the central ideas upon which the SDK is built. The following chapter introduces the fundamental terminology and structural concepts that emerge from these principles and form the basis of the SDK architecture

3. Core Terminology and Structural Model

Unlike many traditional development environments, the SDK does not rely on explicit project descriptions as its primary organizational mechanism. Instead, structure emerges from the interaction between sources, workspaces, and conventions. Higher-level concepts such as code modules, capabilities, modules, and applications are ultimately derived from these relationships

3.1 Sources

A source is any provider of structured data that can participate in the transformation model of the SDK. Sources represent the point at which information enters the system. The physical origin of that information is not relevant to the architecture itself. A source may be backed by a local file system, a package registry, a repository, a virtual file system, a remote service, or any future mechanism capable of providing structured data.

From the perspective of the SDK, these implementations are interchangeable. What matters is the ability to expose information through a common abstraction that can be consumed by subsequent transformation stages.

This distinction is important because the architecture deliberately avoids making assumptions about storage mechanisms or execution environments. Access to files, repositories, packages, or network resources is treated as a capability rather than a requirement. As a result, the same architectural model can operate on local resources, virtualized resources, or resources provided through sandboxed environments.

Sources therefore represent the first architectural element of the SDK. They define where information originates, but not how that information is interpreted

3.2 Workspaces

While sources provide data, they do not exist in isolation. Sources must be discovered, resolved, prioritized, and combined before they can participate in the transformation model. This responsibility belongs to the workspace.

A workspace is a specialized source that aggregates and contextualizes other sources. It defines the environment in which source resolution takes place and provides access to shared resources such as configuration, package definitions, repository definitions, caches, and additional workspace data. Unlike conventional project systems, a workspace is not merely a collection of files. It represents an organizational boundary within which information can be discovered

and resolved according to established conventions.

The SDK distinguishes between two workspace types:

- **Root** workspace represents the globally installed SDK environment. It contains globally available configuration, package sources, repository definitions, and shared resources. The root workspace always exists and serves as the foundation of the system

- **Tree** workspace represents a local extension of the root workspace. Tree workspaces introduce additional configuration, sources, packages, and data that apply within a specific scope. When resolving information, tree workspaces take precedence over the root workspace while still inheriting access to resources it provides

Not every operation requires the existence of a tree workspace. If no local workspace can be resolved, the SDK implicitly operates within the context of the root workspace. This allows arbitrary directories, temporary projects, and standalone experiments to participate in the SDK without requiring explicit workspace creation.

Workspaces therefore form the organizational layer of the architecture. They establish the context in which sources are discovered and resolved while simultaneously participating as sources themselves

3.3 Leafs

Leafs are the smallest structural units recognized by the SDK. A leaf represents a self-contained source of functionality that can participate in the transformation model. While a workspace provides organization and context, a leaf provides content.

Leaves may originate from various sources. A local directory may be interpreted as a leaf. A package obtained from a package registry may be materialized as a leaf. Future source types may introduce additional mechanisms for producing leaves. Regardless of their origin, all leaves participate in the transformation model through a common representation.

This distinction is important because the SDK deliberately separates distribution from structure. The origin of a leaf does not affect how it is processed. A leaf obtained from a local workspace and a leaf obtained from a remote package source are treated identically once they enter the transformation pipeline.

A leaf therefore represents the first structural element that is intrinsic to the SDK itself. Everything preceding it belongs to source resolution and workspace organization. Everything following it belongs to transformation and composition

3.4 Code Modules

A code module is the smallest functional unit within the SDK. While a leaf represents a structural boundary, a code module represents an individual implementation unit that can participate in processing and composition.

A single leaf may contain one or more code modules. In many situations a leaf and a code module appear to be equivalent because a leaf contains only a single implementation. However, the SDK does not require such a relationship. This distinction allows functionality to remain

organized according to responsibility rather than distribution. A package may contain multiple code modules. A workspace may contain multiple leaves. The transformation model ultimately operates on code modules regardless of how those modules were originally obtained.

Code modules are intentionally small in scope and follow the principle of atomic responsibility introduced in the previous chapter. Each module should contribute a clearly defined piece of functionality which can be analyzed, transformed, composed, and reused independently of other modules. The build system, verification systems, and runtime architecture all operate primarily on code modules rather than leaves or packages

3.5 Capabilities

Capabilities represent the first level of semantic composition within the SDK. A capability is created by combining one or more code modules into a coherent unit of functionality. While code modules define implementation, capabilities define behavior.

Examples of capabilities may include file access, repository access, command processing, network communication, rendering systems, or build target generation. The exact implementation details are not important. What matters is that a capability presents a meaningful functional abstraction that can be consumed by higher-level systems.

Capabilities allow functionality to remain independent while still participating in larger compositions. Multiple capabilities may coexist without requiring direct knowledge of one another. Communication occurs through established interfaces, data flows, and shared conventions rather than through tightly coupled implementation details. This approach encourages reuse and enables functionality to evolve independently while remaining compatible with the broader ecosystem

3.6 Modules

Modules are executable compositions of capabilities. While capabilities define what functionality exists, modules define how that functionality is assembled into a deployable unit.

A module may expose services, process data, participate in application workflows, provide user-facing functionality, or extend existing systems. Internally, a module may consist of many capabilities originating from many independent code modules.

The SDK intentionally separates modules from implementation language and deployment strategy. A module may execute within a local process, a sandboxed environment, a dedicated host, a distributed execution environment, or a future execution model not yet defined.

From an architectural perspective, a module is simply a composed unit of behavior that can participate in a larger system. This distinction becomes particularly important when discussing runtime architecture, decentralized execution, and security models in later chapters

3.7 Applications

Applications represent the highest level of composition within the SDK. An application is formed by combining one or more modules into a complete executable system.

The architecture does not distinguish between traditional applications, services, command-line tools, editors, game engines, automation systems, or distributed environments. All are treated as specialized compositions of modules assembled for a particular purpose.

This uniform view allows tools and applications to share common architectural concepts. Functionality developed for one environment can often be reused in another without modification because both participate in the same transformation and composition model. Applications therefore represent not a special category of software but the final stage of composition within the SDK

4. The Transformation Model

The concepts introduced describe the structural elements of the SDK. Sources, workspaces, leaves, code modules, capabilities, modules, and applications define the architectural vocabulary of the system. What remains unanswered is how these structures emerge and how information moves between them.

Transformation is one of the central architectural principles of the SDK. Rather than operating directly on projects, files, packages, or applications, the SDK operates on data. Every higher-level construct within the system is produced through a sequence of transformations that derive increasingly specialized information from existing data.

This principle extends beyond source code processing. Configuration files, package definitions, repository definitions, command streams, build targets, runtime descriptors, and application structures all participate in the same transformation model. Regardless of their origin, information enters the system as data and is gradually transformed into more specialized representations. The transformation model therefore serves as a common foundation for all major subsystems of the SDK

4.1 Transformation as a Core Principle

A transformation consumes data, applies context and convention, and produces new data. Unlike traditional processing pipelines that are often designed around specific tools or workflows, the SDK treats transformation as a generic architectural mechanism. The same principles are used to discover leaves, resolve package dependencies, process command streams, construct capabilities, generate build targets, and compose applications.

Transformation is recursive by nature. The result of one transformation may become the input of another transformation, allowing complex structures to emerge from a sequence of relatively simple operations.

This recursive approach enables systems to remain modular while still supporting complex compositions. Individual transformations can remain focused on a single responsibility while participating in larger workflows

4.2 Data, Context and Convention

Every transformation within the SDK is based on three inputs:

- **Data** represents the information being processed. This may include source files, package definitions, configuration values, repository descriptors, command tokens,

runtime descriptors, or any other structured information

- **Context** provides information about the environment in which a transformation occurs. Workspaces, source origins, dependency relationships, host environments, and execution scopes are examples of contextual information that may influence a transformation

- **Conventions** define how information should be interpreted without requiring explicit configuration. Naming rules, directory structures, package layouts, command semantics, and capability contracts are all examples of conventions used throughout the SDK

Transformation occurs when data is interpreted within a specific context according to an established convention. The resulting output becomes new data which may participate in subsequent transformations.

This relationship can be summarized as: Data + Context + Convention → Data, while deceptively simple, this principle forms the basis of the entire SDK architecture

4.3 Reactive Transformation

The SDK implements transformation using reactive streams. Reactive programming is a programming model in which data is represented as a sequence of events that are observed and processed over time. Rather than requesting information through direct function calls, processing stages subscribe to data sources and react whenever new information becomes available.

The core abstraction of reactive programming consists of observables and observers:

- An **observable** represents a source of events. It produces values that may be consumed by one or more observers

- An **observer** subscribes to an observable and receives notifications whenever new values are emitted. Observers may transform incoming data, aggregate information, generate new events, or forward information to additional processing stages

- **Subscriptions** establish the relationship between observables and observers. They define how information flows through the system and allow processing stages to be connected dynamically without introducing direct dependencies between producers and consumers.

This model provides several advantages for the SDK. Transformations become incremental. Information can be processed as it becomes available rather than requiring complete datasets to exist beforehand.

Responsibilities remain isolated. Individual processing stages consume only the information relevant to their purpose and remain independent of unrelated transformations. This way, transformation pipelines become extensible. Additional observers can be attached without requiring modifications to existing processing stages.

The same mechanism is used throughout the SDK to process source definitions, package descriptors, repository information, runtime metadata, and other transformation inputs. Reactive streams therefore provide

the execution model of the transformation architecture

4.4 Entity Composition

While reactive streams define how information is transformed, entity composition defines how transformation results are represented. The SDK therefore adopts an Entity Component System (ECS) architecture as its primary composition model.

An entity represents an identity within the system. Entities contain no behavior and do not define inheritance relationships. Their sole purpose is to provide a stable reference to a collection of data. A leaf usually is an entity and may aggregate additional distinct entities into semantic data.

Data is stored in components. Components are independent data structures that describe specific properties of an entity. Examples may include source files, dependencies, references, compiler settings, package metadata, capability descriptors, or runtime permissions.

Components are stored in dedicated component arrays. Each array contains all instances of a specific component type and provides efficient access to homogeneous data. This approach separates data from behavior and avoids the rigid object hierarchies commonly associated with traditional object-oriented designs.

Behavior is implemented through systems. Systems operate on entities that possess specific component combinations and perform transformations based on the available data. To simplify processing, entities may be organized into groups, providing dynamically maintained views over entities that satisfy a particular set of component requirements. Systems can therefore operate on groups rather than performing repeated entity discovery operations.

The SDK additionally extends this model through contracts. A contract formally describes the component requirements needed by a transformation or capability. Rather than depending on concrete object types, systems declare the components they require in order to operate. This distinction is important because it allows processing stages to depend on capabilities of data rather than specific implementations.

An entity containing source files and compiler settings can satisfy a compiler contract regardless of where those components originated. Whether the information was produced by a local leaf, a package source, or a workspace becomes irrelevant once the contract has been satisfied. Entity composition therefore provides the structural foundation of the transformation model. It enables data to evolve independently of processing logic while supporting highly composable systems

4.5 Structural and Semantic Transformation

Reactive transformation and entity composition solve different aspects of the same problem. Reactive streams define how information moves through the system. Entity composition defines how information is represented within the system. Together they form the foundation of the transformation model.

A transformation begins when data enters a reactive stream. Observers subscribe to the stream and consume information according

to their responsibility. Each observer may create, modify, enrich, aggregate, or emit new data. The resulting information is represented through entities and components, making it available for subsequent transformations.

This creates a continuous cycle in which reactive systems transform information and entity compositions preserve the results of those transformations. Additionally, systems add additional on-demand behavior to the transformation model. The distinction between structural and semantic transformation emerges naturally from this process.

Structural transformations create or modify architectural structures. Examples include resolving sources into leaves, discovering code modules, constructing dependency graphs, loading package definitions, or generating workspace representations. Semantic transformations assign meaning to existing structures. Examples include identifying compiler targets, resolving capabilities, validating contracts, constructing permission sets, or interpreting command streams.

Both forms of transformation are implemented using the same underlying mechanisms. The difference lies not in the technology itself but in the purpose of the transformation being performed. The SDK therefore does not distinguish between build-time data, runtime data, configuration data, or package data. All information participates in the same transformation model and is processed through the same architectural principles

4.6 Composition

Transformation alone is not sufficient to build software systems. The purpose of transformation is not merely to convert one form of data into another but to enable composition.
Composition is the process by which independent pieces of information are combined into larger and more meaningful structures. A leaf is a composition of source information, a capability is a composition of code modules, a module is a composition of capabilities, and an application is a composition of modules. The same principle applies at every level of the architecture.

Unlike traditional object-oriented systems, composition within the SDK is not achieved through inheritance hierarchies or tightly coupled object graphs. Instead, composition emerges through the aggregation of components and the interaction of transformation systems. Because entities are defined by their components rather than their inheritance relationships, new functionality can be introduced without modifying existing structures:

- A compiler system does not require a specific project type. It requires an entity that satisfies the compiler aggregation

- A package resolver does not require a specific package implementation. It requires an entity that satisfies the package aggregation

- A runtime host does not require a specific module implementation. It requires an entity that satisfies the module aggregation

This approach allows independent subsystems to evolve without introducing unnecessary coupling. Composition therefore becomes a property of data rather than a property of implementation. As a result, complex systems can emerge from many small and independently developed

parts while remaining predictable and maintainable

4.7 Transformation Consumers

The transformation model is independent of any particular tool or execution environment. Transformation produces information. Consumers decide how that information is used.

The build system is one example of a transformation consumer. It transforms source information into compiler invocations, package dependencies, generated project files, runtime descriptors, and deployable artifacts. However, the build system is not unique in this regard:

- Package managers consume transformation results to resolve dependencies and repositories
- Verification systems consume transformation results to analyze code, validate defined standards, and enforce security policies
- Runtime hosts consume transformation results to construct capabilities, modules, permission sets, and execution environments
- Development tools consume transformation results to generate user interfaces, diagnostics, project structures, and editing experiences

Each of these systems operates on the same underlying information and participates in the same architectural model. This distinction is important because it separates transformation from execution. Transformation describes how information is derived. Consumers describe how that information is used.

The transformation model therefore represents a shared architectural foundation rather than a feature of any individual tool. By treating transformation as a first-class architectural concept, the SDK establishes a common processing model that can be reused throughout the entire ecosystem, regardless of language, execution environment, distribution mechanism, or application domain

5. Runtime Architecture

The transformation model describes how information moves through the SDK. Sources are transformed into leaves, leaves into code modules, code modules into capabilities, and capabilities into executable systems. The runtime architecture describes the environment in which these transformations become active software.

While the transformation model focuses on data and composition, the runtime focuses on execution, visibility, isolation, and access to infrastructure. It defines how modules are loaded, how capabilities are exposed, how communication occurs between independent parts of a system, and how access to resources is controlled.

A fundamental design goal of the SDK is the separation of logic from infrastructure. Modules should not depend directly on operating systems, processes, network stacks, file systems, or application implementations. Instead, they interact exclusively through contracts and capabilities provided by a host. This allows the same module to be reused in different environments without modification, ranging from standalone command-line tools to editors, distributed services, and future execution environments.

The runtime therefore acts as a mediation layer between modules and the systems in

which they execute. It provides common infrastructure, manages lifecycle and visibility, resolves dependencies, and ensures that communication occurs through controlled and verifiable mechanisms rather than through unrestricted access to implementation details. Just as the transformation model treats code modules as the fundamental building blocks of composition, the runtime treats capabilities as the fundamental building blocks of execution

5.1 Runtime Architecture

At its core, the runtime consists of three conceptual layers:

- **Runtime Capabilities** provide access to infrastructure and shared services. Examples include file systems, network communication, configuration providers, stream registries, ECS registries, process execution, and other platform-dependent functionality. Capabilities are responsible for interacting with the outside world and exposing controlled interfaces to higher layers

- **Modules** contain application logic and transformation logic. Unlike capabilities, modules are not responsible for providing infrastructure. Instead, they consume capabilities and combine them into higher-level behavior. A build system, package installer, repository provider, editor extension, or game system are all examples of modules

- The **Host** owns the runtime instance and determines which capabilities exist, which modules may be loaded, how modules interact with one another, and which security restrictions apply. Every executing system within the SDK ultimately operates under the control of a host

Conceptually, the host is not merely a loader. It acts as the authority responsible for capability resolution, visibility management, security enforcement, lifecycle management, and runtime composition. Two hosts may execute the same module while exposing entirely different capabilities and policies. As a result, modules are written against contracts rather than implementations.

An important consequence of this architecture is that the runtime itself remains composable. Runtime functionality is not treated as a special category of software. Every runtime feature ultimately originates from code modules. A code module may serve as a normal implementation, a runtime capability, a reference assembly, or a source for runtime projections depending entirely on the context in which it is used. For example, a file system implementation may exist as an ordinary code module during development. The same module may later become a runtime capability, generate reference assemblies for compilation, provide metadata for contract extraction, and ultimately participate in runtime projection when modules are loaded. The module itself remains unchanged. Only its role within the system changes.

This principle allows the runtime to grow through composition rather than through special-case infrastructure. New capabilities can be introduced using the same mechanisms already used throughout the SDK, preserving the principles of atomic responsibility, collaboration, and convention over configuration.

The runtime therefore should not be viewed as a separate subsystem sitting beside the SDK. It is itself a product of the same compositional model that governs the rest of the architecture, extending the transformation process into execution, communication, and controlled access to resources

5.2 Hosts

While the runtime provides the common infrastructure required for execution, it is the host that ultimately determines how that infrastructure is exposed and controlled. A host owns a runtime instance and acts as the authority responsible for capability resolution, module loading, visibility management, lifecycle control, and security enforcement. Every module executes within the context of a host and interacts with the outside world exclusively through mechanisms provided by that host.

The architecture intentionally separates runtime functionality from host responsibilities. The runtime provides the mechanisms required for composition and execution, while the host defines the policies under which those mechanisms operate. This distinction allows the same module to execute in different environments while preserving identical behavior and contracts. For example, a module may request access to a filesystem capability. One host may provide unrestricted access to the local filesystem, another may expose only a virtual workspace, while a third may deny the request entirely. The module itself remains unchanged. Only the host's interpretation of the request differs.

This separation is fundamental to the security model of the SDK. Modules never decide what they are allowed to access. They merely declare what they require. The host determines what is available. To support different execution scenarios, the SDK defines three host categories

Thin Host

The thin host represents the simplest execution environment supported by the runtime architecture. In a thin host, modules are typically composed directly into an executable during the build process. The resulting application contains both the host and the modules it requires, allowing execution without dynamic module discovery or runtime composition. Examples include command-line tools such as the build system, repository management tools, package utilities, and other standalone applications.

Because all participating modules are known during compilation, the host can resolve dependencies statically and embed the required runtime capabilities directly into the executable. Startup overhead is minimal and the execution model closely resembles a traditional application while still preserving the architectural principles of capabilities, streams, contracts, and composition. A thin host may still expose runtime services internally, but these services are not intended to be extended dynamically after deployment

Application Host

The application host extends the thin host model by introducing runtime composition. Unlike a thin host, an application host may load additional modules during execution and can adapt its available functionality without requiring recompilation. The host remains responsible for selecting which capabilities are exposed and which modules may participate in the application. Typical examples include editors, asset browsers, package managers, visualization tools, and other interactive applications.

Application hosts commonly provide a richer runtime environment than command-line tools. Multiple modules may coexist within the same runtime instance, share data through streams and ECS contracts, and contribute functionality to a common user experience. Although modules may interact through runtime mechanisms, the host retains authority over visibility and access. A module cannot assume that other modules are present, visible, or accessible unless explicitly permitted by the host

Service Host

The service host represents the most controlled and isolated execution environment within the architecture. A service host is designed to execute multiple modules and applications within a shared runtime while maintaining strict control over communication, visibility, permissions, and lifecycle management. This host category forms the foundation of decentralized method execution (DME), distributed processing, automation services, and future containerized execution environments.

Unlike an application host, where modules often contribute directly to a shared application, modules within a service host are treated as isolated participants managed by the runtime. Communication occurs through controlled runtime mechanisms rather than through direct access to implementation details. Modules are therefore unable to establish arbitrary communication paths or bypass the runtime.

Multiple applications may coexist within the same service host while remaining isolated from one another. The host determines which information may cross these boundaries and under which conditions. This model enables distributed execution without requiring modules to understand whether they execute locally, remotely, inside a container, or as part of a larger system. System boundaries influence only the communication mechanisms selected by the host, not the behavior of the modules themselves

Host Authority

Regardless of host type, every host is responsible for the same fundamental decisions. A host determines: which capabilities exist, which modules may be loaded, which modules may request additional modules, which streams may be registered, which streams may be subscribed to, which ECS groups and contracts may be registered, which data is visible between modules, and which security policies apply.

The runtime provides the mechanisms required to perform these operations. The host decides whether they are allowed. As a result, modules remain portable across different environments while hosts remain free to implement different trust models, permission systems, virtualization layers, and security policies without requiring changes to module implementations

5.3 Runtime Capabilities

Capabilities form the infrastructure layer of the runtime. While modules contain transformation logic and application behavior, capabilities provide access to the resources and services required to execute that behavior. This distinction is intentional and forms one of the central architectural boundaries of the SDK. Modules describe what should happen. Capabilities provide the means through which it can happen.

A capability may expose access to a filesystem, network communication, configuration management, or any other service required by modules. From the perspective of a module, however, the implementation of a capability is irrelevant. Only the contract matters.

This separation ensures that modules remain independent from operating systems, frameworks, applications, and deployment environments. A module may request access to a filesystem capability without knowing whether the underlying implementation uses a local filesystem, a virtual workspace, a remote storage provider, or a restricted sandbox. Capabilities therefore serve as the boundary between application logic and infrastructure

Capabilities as Runtime Services

A capability is fundamentally a runtime service. Unlike modules, capabilities are not loaded to perform a specific transformation. Instead, they exist to provide reusable functionality to other participants within the runtime. Many capabilities expose operations that cannot naturally be represented as event streams. Reading a file, performing an HTTP request, parsing a package definition, or requesting user consent are examples of actions that require explicit interaction.

For this reason, capabilities are not limited to reactive communication. A capability may expose APIs, streams, ECS contracts, resource providers, or combinations of these mechanisms depending on the nature of the service it provides. The runtime imposes no restrictions on how a capability internally performs its work. The only requirement is that interaction occurs through contracts visible to the host

Capabilities as Code Modules

Capabilities do not represent a separate artifact type within the SDK. Like all other executable functionality, capabilities originate from ordinary code modules. Whether a code module becomes a capability depends entirely on context.

A code module may be: a normal implementation module, a runtime capability, a reference assembly, or a source for runtime projections without any changes to its implementation. For example, a filesystem code module may provide the implementation used by a host, generate reference assemblies used during compilation, expose contracts consumed by modules, and participate in runtime projection during module loading.

The SDK therefore avoids introducing special categories of runtime code. The same compositional model used throughout the transformation process continues into runtime execution

Capability Discovery and Resolution

Capabilities are not accessed directly. When a module requires functionality, it requests a capability through the runtime. The host then determines whether that capability is available and how it should be provided. This behavior allows hosts to adapt capabilities to their own security and execution models.

For example, a filesystem request may resolve differently depending on the host:

Module
↓
Filesystem Capability

Thin Host
→ Native Filesystem

Application Host
→ Filesystem + Permission Layer

Service Host
→ Sandboxed Filesystem

The module remains unaware of which implementation is ultimately selected. This mechanism prevents infrastructure decisions from leaking into application logic and allows the same module to execute in vastly different environments without modification

Capability Contracts

Capabilities are defined through contracts rather than concrete implementations. A contract describes the operations, data structures, streams, and ECS definitions exposed by a capability. Modules compile against these contracts and express their requirements through references to them.

The actual implementation is supplied later by the host through runtime projection and capability binding. This design provides several advantages:

- Modules remain portable because they depend only on contracts
- Hosts retain complete authority over implementation details
- Capabilities can evolve independently from the modules that consume them
- The same contract can be bound to multiple implementations without requiring changes to module code

Runtime Infrastructure Capabilities

Certain capabilities form the foundation of the runtime itself. Examples include the stream registry and ECS registry. Unlike application-facing capabilities such as filesystems or network communication, these services provide the mechanisms through which modules communicate and compose behavior.

The stream registry allows modules to publish and subscribe to reactive data streams. The ECS registry provides access to entities, component arrays, groups, and contracts used throughout the system. Together these capabilities establish the shared execution environment from which higher-level functionality emerges.

Although hosts may extend or replace their implementations, the concepts themselves remain common across all runtime environments

Capabilities and Security

Capabilities also serve as the primary enforcement point for security. Modules do not interact directly with operating system resources. Every access request passes through a capability controlled by the host. This allows hosts to introduce additional behavior during capability resolution, including: permission checks, visibility restrictions, virtualization layers, or user approval workflows, without requiring changes to modules or capability contracts.

As a result, security becomes a property of the runtime architecture rather than a responsibility delegated to individual modules

5.4 Modules

Modules are the executable units of behavior within the runtime architecture. While capabilities provide infrastructure, modules provide logic. Every transformation, workflow, tool, service, editor extension, automation task, or game system ultimately exists as one or more modules operating within a host. A module does not own infrastructure. It does not control execution policies, manage visibility, or determine access to resources. Instead, it consumes capabilities exposed by a host and combines them into meaningful behavior.

This distinction forms one of the most important boundaries within the SDK. Capabilities answer the question: What functionality is available? Modules answer the question: How should that functionality be used?

As a result, modules remain focused on transformation and composition while hosts and capabilities remain responsible for execution and access control

Modules as Compositions

A module is fundamentally a composition of contracts, capabilities, streams, ECS definitions, and transformation logic. Modules may consume data produced by other modules, publish new data, expose reusable functionality, or coordinate complex workflows spanning multiple capabilities. Despite the different use cases, all modules follow the same architectural model. They consume capabilities through contracts, process data, and contribute new behavior to the runtime.
The runtime does not distinguish between tool modules, application modules, engine modules, or service modules. These roles emerge from composition and context rather than from special runtime types

Module Dependencies

Modules may depend on other modules. This dependency relationship exists to allow modules to share contracts, ECS definitions, streams, and exported data structures without introducing direct coupling to implementations. A module may therefore reference another module when it requires access to exported types or runtime definitions.

However, modules do not directly load dependencies themselves. The authority to load modules always remains with the host. When a module requires additional functionality, it may request that another module be loaded. The host evaluates the request and determines whether it should be granted. This ensures that dependency management remains a runtime concern rather than a responsibility delegated to module implementations

Module Initialization

When a module is loaded, the runtime invokes its initialization entry point. This initialization phase represents the moment at which the module becomes attached to the runtime environment and may begin interacting with available services. To preserve determinism, portability, and security, initialization is intentionally restricted.

A module may perform only four categories of actions during initialization:

- Request additional modules
- Register streams
- Subscribe to streams
- Register ECS groups and contracts

These actions define the complete integration surface between a module and the runtime. Modules cannot arbitrarily manipulate runtime internals, access host resources directly, or modify execution policies.

Instead, they express their requirements through these controlled mechanisms and rely on the host to determine whether those requirements should be satisfied

Requesting Additional Modules

A module may request that the runtime load additional modules. This mechanism allows functionality to be composed incrementally rather than requiring all participating modules to be known in advance. The request itself does not imply success. The host remains responsible for evaluating whether: the requested module exists, the request is permitted, dependencies can be satisfied, and security policies allow the operation A module therefore expresses intent, while the host retains authority

Stream Registration and Subscription

Streams form one of the primary communication mechanisms within the runtime. During initialization, a module may register new streams or subscribe to existing streams exposed through the runtime.

Registered streams become part of the runtime's communication infrastructure and may be discovered by other authorized modules.
Subscriptions allow a module to react to events and data produced elsewhere in the system.

Because stream visibility is controlled by the host, registration alone does not imply global accessibility. Hosts may restrict visibility to specific modules, applications, security domains, or runtime scopes

ECS Registration

A module may additionally register ECS groups and contracts during initialization. Groups define how components are organized and processed. Contracts define the structures through which data is shared and interpreted. By registering these definitions during startup, modules contribute new data models to the runtime without requiring modifications to existing systems.

As with streams, the host retains authority over visibility and accessibility. Not every registered definition must be visible to every participant

Module Boundaries

Modules are intentionally limited in scope. A module cannot: bypass capability resolution, directly access operating system resources, inject code into other modules, modify runtime policies, load arbitrary assemblies, or circumvent host decisions.

These restrictions are fundamental to the architecture and form the basis for security and verification mechanisms. The goal is not merely to prevent misuse but to ensure that modules remain portable and predictable regardless of where they execute.

A module should behave identically whether it is executed inside a command-line tool, an editor, a service host, or a future execution environment

Modules and Runtime Evolution

The runtime is designed around the assumption that new functionality will emerge through modules rather than through modifications to the runtime itself. As a result, modules represent the primary mechanism by which behavior evolves over time.

New workflows, tools, services, integrations, and execution models can be introduced by composing existing capabilities and exposing new contracts without requiring changes to the architectural foundations of the system. This approach allows the runtime to remain relatively small and stable while enabling continuous growth through composition

5.5 Runtime Composition

The introduced fundamental participants of the runtime architecture: hosts, capabilities, and modules, create the basics for composition, the description of how these elements are assembled into an executable system. Composition is one of the central principles of the SDK. Rather than treating applications as monolithic executables, the architecture treats them as compositions of independent building blocks assembled according to context, available capabilities, and host policies.

The same principle that governs transformation during the build process therefore continues into runtime execution. Applications are not constructed from special runtime artifacts. They emerge from the composition of code modules, capabilities, data, and contracts.

Static Composition

The simplest form of runtime composition occurs during compilation. In this model, all participating modules are known in advance and are assembled directly into a host application. Required capabilities are resolved during the build process and embedded into the resulting executable.

From the perspective of the runtime, the system remains composed of modules and capabilities. The difference is merely that the composition has already been performed before execution begins. This model is particularly suitable for standalone and command-line tools such as:

- Version Control
- Package Manager
- Build

Because no dynamic discovery is required, startup costs remain minimal while preserving the same architectural model used elsewhere in the SDK. The resulting application behaves as a self-contained runtime instance with a predefined set of capabilities and modules

Dynamic Composition

In larger applications, the participating modules may not be known at compile time. Editors, browsers, service containers, and future execution environments often need to load functionality on demand. In these scenarios, composition occurs during execution rather than during compilation. The host remains responsible for discovering modules, resolving dependencies, validating contracts, constructing runtime projections, and binding capabilities.
The composition process therefore becomes an active responsibility of the runtime. Conceptually, the process can be represented as:

Module Request
↓
Module Discovery
↓
Contract Analysis
↓
Runtime Projection
↓
Capability Binding
↓
Module Initialization

Each step remains under host control. Modules participate in the process but never determine its outcome

Runtime Packages

To support dynamic composition, modules may be distributed as runtime packages. A runtime package is a deployable representation of a module together with the metadata required to integrate it into a runtime environment. Unlike a traditional assembly, a runtime package is intended to describe not only executable code but also the requirements and characteristics of that code.

A package contains:

- Executable Code
- Package Metadata
- Dependency Information
- Role and Privilege Definitions
- Cryptographic Signatures

The exact storage format is intentionally independent from the runtime architecture. What matters is that sufficient information exists for the host to determine how the package should be integrated. This design allows modules to be transported, cached, validated, and composed without requiring prior knowledge of their implementation

Module Roles and Privileges

Not every runtime package serves the same purpose. Some modules exist as standalone entry points able to be executed directly by a host. Others exist solely to provide functionality to the host application. For this reason, runtime packages may define different roles and privileges, such as standalone execution.

The assigned role influences how the host interprets the package and how it participates in composition. The role does not change the module itself. It merely describes the intended purpose of the package within the runtime environment

Code Modules as Runtime Artifacts

An important consequence of the SDK architecture is that runtime artifacts are not fundamentally different from ordinary code modules. Every capability, module, reference assembly, projection source, and runtime package ultimately originates from code modules produced by the transformation model.

A runtime package therefore does not introduce a new programming model. It represents a different context in which existing code modules are used. For example, a filesystem implementation may simultaneously exist as: code module, capability, and reference assembly, without requiring multiple implementations. The distinction exists entirely at the architectural level

Composition as an Execution Model

The purpose of runtime composition extends beyond dynamic loading. Because modules interact through contracts, streams, ECS definitions, and capabilities, composition becomes a mechanism for constructing larger systems from smaller and independently developed parts.

A module may consume data produced by another module without requiring knowledge of its implementation. Capabilities may evolve independently from modules. Hosts may introduce new security policies without affecting application logic.

Entire applications may be assembled from reusable building blocks originating from different teams, repositories, programming languages, or deployment environments. Composition therefore serves not only as a deployment mechanism but as the primary method through which functionality grows and evolves within the SDK ecosystem

5.6 Runtime Projection and Contract Resolution

The runtime architecture is designed around the principle that modules depend on contracts rather than implementations. A module is compiled against capability contracts that describe the functionality it requires, while the host remains responsible for supplying concrete implementations at runtime.

This separation allows modules to remain portable across different hosts and execution environments. It also forms the foundation for security enforcement, capability virtualization, and future post-compiler integration. The challenge lies in bridging the gap between compile-time contracts and runtime implementations.

A module may be compiled against a filesystem capability contract, while the host ultimately provides a native filesystem, a virtual workspace, a sandboxed implementation, or no filesystem at all. The runtime must therefore establish a binding layer between the contracts known to the module and the capabilities exposed by the host. This process is referred to as runtime projection

Capability Reference Assemblies

Each capability defines its own reference assembly. Reference assemblies describe contracts only. They contain the types, interfaces, data structures, streams, ECS contracts, and service definitions that a module may consume during compilation. They do not contain implementations other than for debugging purposes at development.

Modules reference only the capability contracts they require. This approach intentionally avoids a monolithic runtime reference assembly.

As the number of capabilities grows, independent capability references remain easier to maintain, easier to version, and more efficient to load. It also allows hosts to construct runtime environments incrementally rather than requiring a complete runtime description before execution begins

Runtime Projection

When a module is loaded, the runtime analyzes the capability contracts referenced by the module. For each required capability,

the host generates or retrieves a runtime projection. A runtime projection is a runtime-specific representation of a capability contract that binds the contract expected by a module to the implementation provided by the host.

The module interacts exclusively with the projection. The projection interacts with the host capability. This indirection allows the host to insert additional behavior without affecting the module itself: permission checks, visibility enforcement, virtualization, sandboxing and fallback behavior if a capability doesn't exist, are possible examples of modifications.

The module remains unaware of these additional layers

Projection Ownership

Runtime projections are generated once per capability within a runtime instance. The first aggregation of modules requiring a capability establishes the projection boundary for that capability. If a lazy loaded or referenced module requires the same capability, the runtime reuses the existing projection and rejects loading that module if projection isn't compatible.

This rule serves several purposes:

- It avoids expensive global analysis before execution begins
- It enables incremental module loading
- It allows projections to be cached and reused
- It guarantees deterministic capability resolution throughout the lifetime of the runtime instance

Modules loaded later must be able to operate within the capability surface already established by the runtime

Host-Controlled Projection

The runtime projection represents the point at which host policies become part of the execution model. Although capability contracts remain identical across hosts, the resulting projection may differ depending on the execution environment. The capability contract remains unchanged. Only the projection differs.

This allows hosts to enforce security requirements without requiring modifications to modules or capability definitions

Service Host Isolation

Service hosts introduce an additional requirement. Unlike thin hosts and application hosts, service hosts must support unloading modules without terminating the runtime itself. To achieve this, each loaded module is placed into an isolated execution container managed by the host.

Each container maintains its own module lifetime while sharing the common runtime infrastructure provided by the host. This allows: module unloading, hot reloading, and fault isolation, without requiring the entire service host to restart.

Communication between containers remains subject to host-controlled visibility and routing rules. Modules therefore continue to interact through runtime mechanisms rather than through direct references

Relationship to the Post-Compiler

The runtime projection system also serves as the architectural foundation for the future post-compiler. In the current architecture, capability requirements are derived from capability reference assemblies and runtime analysis:

Reference Assembly
↓
Contract Analysis
↓
Runtime Projection

The post-compiler will eventually replace the analysis stage with explicit metadata generated during compilation. The projection mechanism itself remains unchanged.

As a result, the runtime architecture can evolve toward the full security model described in the Process Interop paper without requiring fundamental changes to hosts, modules, capabilities, or runtime composition. The post-compiler changes how capability information is obtained. It does not change how capabilities are resolved

5.7 Runtime Registries and Visibility

While modules, capabilities, contracts, streams, and ECS definitions form the visible building blocks of the system, registries provide the infrastructure through which these elements are discovered, resolved, and managed.

A runtime instance is therefore not merely a collection of loaded modules. It is a structured environment consisting of multiple registries coordinated by a host. These registries establish the shared execution context in which modules operate while allowing hosts to enforce visibility, security, and isolation policies without modifying module implementations

Stream Registry

The stream registry manages the reactive communication infrastructure of the runtime. Modules may register streams, discover existing streams, and subscribe to streams exposed by other participants. All stream-related operations pass through the registry. The registry is responsible for: stream registration, stream discovery, and subscription management.

Modules do not communicate with one another directly. Instead, they communicate through streams managed by the runtime. When a module requests access to a stream, the runtime evaluates whether that stream is visible within the current execution context before returning a reference to the stream. This distinction is important.

Visibility does not prevent a module from knowing that another module exists. Visibility determines whether runtime services such as stream discovery and subscription are permitted to expose communication channels between participants

ECS Registry

The ECS registry manages entities, component arrays, groups, and contracts. Like the stream registry, it acts as the

authoritative source for discovery and resolution operations. The registry is responsible for: entity registration, component registration and lookup, groups, and contract registration and lookup.

When a module requests access to ECS data, the runtime determines whether the requested information is visible within the current execution context. A module should not automatically imply that a requested component will be returned.

The runtime may decide that the component is not visible to the requesting module and return no result even though the component exists elsewhere within the system. This allows hosts to control data visibility without requiring modifications to ECS implementations or module code

Projection Registry

The projection registry manages runtime projections generated from capability contracts. Each runtime instance maintains exactly one projection per capability.

Once a projection has been established, subsequent modules requiring the same capability reuse the existing projection rather than generating a new one. This guarantees deterministic capability resolution while avoiding expensive analysis of all potential modules before execution begins.

The projection registry therefore acts as the bridge between compile-time contracts and runtime implementations

Host-Specific Visibility

Visibility requirements vary depending on the host model. Application hosts may apply visibility rules when extensions or dynamically loaded modules participate in the same runtime environment, although these requirements are typically less restrictive.

Service hosts represent the primary motivation for runtime visibility. Because multiple isolated modules may coexist within the same service instance, the host must be able to control which communication channels and data structures are exposed between containers.

Visibility therefore becomes one of the mechanisms through which the service host enforces isolation while still allowing controlled collaboration between modules. This approach enables modules to communicate through runtime-managed contracts without granting unrestricted access to the internal state of other participants

6. Decentralized Method Execution

One of the recurring challenges in software development is the reuse of functionality across application boundaries. Tools are often developed to solve a specific problem, only to become difficult to integrate elsewhere once they mature. Build systems, asset processors, package managers, editors, and automation tools frequently expose their functionality through command-line interfaces, custom network protocols, scripting layers, or application-specific APIs. While effective in isolation, these approaches create additional integration work whenever functionality must be reused by another tool.

The original motivation behind decentralized method execution (DME) emerged from this problem. The goal was not to create a replacement for existing

inter-process communication mechanisms, nor to introduce another service framework. Instead, the objective was to make functionality available independently of the environment in which it was originally developed.

A build system should be usable from an editor. An editor should be able to consume functionality provided by an external tool. A service should be capable of delegating work to another process without requiring both applications to agree on a custom protocol. In all cases, the desired interaction is identical: one participant exposes functionality, another participant invokes it.

Traditional command-line integration solves this problem only partially. Data must be converted into textual arguments, execution must be coordinated externally, and results must be interpreted from textual output. While universally available, command-line interaction is inherently limited to string-based communication and places the burden of integration on the caller.

DME approaches the problem differently. Instead of exchanging commands and textual output, applications exchange executable requests and structured data. The objective is not to hide process boundaries, but to make them less relevant to the design of software systems. Within the SDK architecture, DME therefore extends the concept of composition beyond the boundaries of a single process. The same principles used to compose capabilities, modules, and transformations inside a runtime can be applied to functionality hosted elsewhere

6.1 DME as an Execution Capability

Decentralized method execution is a runtime capability that allows functionality to be invoked across process boundaries. An application publishes a set of functions that are available for execution within the standardized binary execution engine. Other applications may discover these functions and submit execution requests. The receiving application performs the requested operation and returns a result to the caller. Conceptually, the interaction is straightforward.

The important aspect of DME is that neither participant needs to establish a custom communication protocol for every interaction. Instead, both applications rely on the standardized binary execution engine. DME intentionally separates the concept of execution from the mechanism used to transport requests. Shared memory, network transports, or future communication mechanisms may all be used to exchange messages. The transport itself is considered an implementation detail of the DME provider rather than part of the execution contract.

From the perspective of participating applications, the interaction remains identical regardless of where the execution occurs. Whether a function is executed in another process, another service, or on another machine does not fundamentally change the invocation model. DME should therefore be viewed as an execution capability rather than a communication protocol. It provides a standardized method of invoking functionality across boundaries while remaining independent of the underlying transport layer

6.2 DME as a Transformation Layer

Within the broader SDK architecture, DME can be understood as an extension of the transformation model. The transformation model describes how data is transformed through a sequence of processing stages. Traditionally, these transformations occur within a single runtime instance.

DME extends this model by allowing individual transformation stages to exist outside the local process. The transformation itself remains unchanged. Only its location changes.

This distinction is important because it preserves the architectural principles established throughout the SDK. Modules continue to consume data, perform transformations, and produce results. DME merely provides a mechanism for routing transformations to external participants when appropriate. As a consequence, process boundaries become part of the infrastructure rather than part of the application logic. A module does not need to know whether a transformation is performed locally or remotely. It interacts with a capability that produces results.

This approach aligns naturally with the runtime architecture. Modules may consume DME through the same mechanisms used to access other capabilities. Hosts remain responsible for determining whether DME is available, how requests are routed, and which participants are permitted to communicate. In practice, this allows transformation pipelines to span multiple applications while preserving the compositional model established by the runtime

6.3 Applications, Services, and Distributed Execution

DME enables applications to cooperate without requiring direct integration between their internal implementations. An editor may delegate build operations to a build service. A package management tool may expose package discovery and installation functions to external consumers. A deployment system may invoke build operations hosted by another process. In each case, the participating applications remain independent while sharing functionality through a common execution model.

This principle becomes increasingly important as systems grow more distributed. A service host may expose functionality provided by isolated modules. Other applications may interact with those modules through DME without requiring direct access to the runtime environment in which the modules execute. The service remains responsible for routing requests, enforcing visibility rules, and applying any security policies required by the execution environment. The same model also applies to distributed systems. A local process, a remote service, and a networked application can all participate in the same execution model as long as a suitable transport mechanism exists.

One particularly interesting application of this concept is the delivery of functionality on demand. Rather than embedding every feature directly into an application, systems may delegate specialized tasks to external services or dynamically loaded modules. This allows capabilities to evolve independently while remaining accessible through a stable execution interface.

For game engines, this model provides opportunities beyond traditional tooling. Content processing and server-side services may be composed from independent modules without requiring

those modules to become part of the engine itself. The engine remains responsible for execution and presentation, while functionality can be developed, deployed, and maintained independently.

DME therefore represents more than a communication mechanism. It extends the compositional principles of the SDK beyond the boundaries of a single runtime instance, allowing applications, services, and distributed systems to participate in a shared transformation model while remaining independent in their implementation and deployment.

7. Security and Verification

The original motivation for the security model emerged from a practical problem rather than a theoretical one. Modern software systems increasingly rely on plugins, extensions, mods, third-party libraries, and community contributions. While these approaches encourage collaboration and accelerate development, they also introduce uncertainty. Applications frequently execute code that was not written by the original author of the system itself.

This challenge becomes particularly visible in areas such as game development, development tooling, automation systems, and open-source ecosystems. A system may wish to encourage contributions from external developers while simultaneously protecting itself from accidental misuse, incompatible changes, or intentionally malicious behavior. As software becomes more composable, the origin of individual components becomes increasingly diverse. Applications may depend on modules written by different contributors, consume capabilities provided by external services, or integrate functionality developed without direct control of the application author.

Traditional approaches often address this problem through strict platform control, limited extension mechanisms, or by exposing carefully selected functionality through scripting environments. While effective in certain scenarios, these approaches frequently reduce flexibility and make composition more difficult.

The SDK follows a different philosophy. Instead of restricting extensibility, it attempts to establish clear boundaries between functionality, permissions, responsibilities, and execution environments. Security is therefore not treated as an isolated subsystem but as a property emerging from the architecture itself

7.1 Trust Boundaries

A central principle of the SDK security model is that trust does not originate from modules. Trust originates from hosts: while modules describe functionality and capabilities describe requirements. This distinction is fundamental because it separates software behavior from execution authority.

A module may request access to a filesystem capability. It may attempt to communicate through DME. It may register streams, consume ECS contracts, or interact with external services. None of these actions imply that the requested functionality will be made available. The final decision always belongs to the host.

This principle allows the same module to operate in multiple environments with different security requirements. This inversion of responsibility is one of the defining characteristics of the architecture. Rather than requiring modules to implement their own security policies, the runtime

establishes a consistent model in which authority is concentrated within the host. Only the trust relationship between the host and the module differs.

As a result, security decisions remain visible, auditable, and enforceable at the boundaries of the system rather than being distributed across individual implementations. The practical consequence is that modules do not possess permissions. Modules possess requirements, while permissions emerge only when a host chooses to satisfy those requirements

7.2 Capability-Based Security

The security model of the SDK is built around capabilities rather than permissions embedded directly into modules. A capability represents access to a specific area of functionality provided by the runtime. Modules do not gain access to these capabilities automatically. Instead, capabilities must be explicitly requested through the runtime and resolved by the host. This design establishes a clear separation between functionality and authority.

A module may declare that it requires access to a certain capability. The declaration itself does not grant access to the filesystem. It merely communicates a requirement. The host remains responsible for deciding whether the capability should be provided, restricted, virtualized, or denied entirely. As a result, capability contracts become the primary boundary between application logic and system resources.

This approach provides several advantages:

- Capabilities remain portable across execution environments. The same module may operate against a native filesystem in one host, a virtualized workspace in another host, or a restricted implementation in a service environment

- Security decisions become centralized. Rather than distributing authorization logic across individual modules, capability resolution occurs at a well-defined boundary controlled by the runtime

- Capabilities remain composable. New functionality can be introduced without modifying existing modules, provided appropriate contracts exist.

The runtime projection system extends this model further. Modules interact only with capability contracts, while projections bind those contracts to concrete implementations supplied by the host. Additional layers such as visibility policies, auditing, virtualization, or permission prompts may be inserted transparently without affecting module implementations. Capabilities therefore serve two purposes simultaneously: they not only define how functionality is accessed, but also where trust boundaries are enforced

7.3 Verification

Capability-based security controls what software is allowed to access at runtime. Verification addresses a different problem. A verification authority determines whether software complies with the architectural assumptions of the platform before execution begins.

The SDK assumes that modules interact with their environment through capabilities, runtime contracts, streams, ECS structures, and approved interfaces. Software that attempts to bypass these mechanisms undermines the security guarantees provided by the runtime and must therefore be detected as early as possible.

Verification is performed in multiple stages. The first stage exists today in the form of static analysis performed by the build system and associated tooling. At this stage, source code and compiled assemblies may be inspected for patterns that violate architectural constraints. Examples include direct filesystem access outside approved capabilities, unrestricted reflection, dynamic assembly loading, unmanaged interoperability, runtime code generation, or other techniques that bypass the runtime model. The objective of this analysis is not merely code quality. It is the preservation of architectural integrity.

Modules are expected to interact with the system through contracts rather than implementation details. Verification ensures that these expectations remain enforceable. Future versions of the SDK introduce an additional verification stage through the post-compiler architecture described in the Process Interop paper. Unlike traditional compilation, the post-compiler operates after the language compiler has produced an assembly. The generated assembly is analyzed, validated, and transformed into a deployment format suitable for controlled execution environments. The post-compiler therefore performs several tasks:

- It validates references against approved contracts
- It extracts dependency information and capability requirements
- It removes information that is unnecessary for execution
- It generates metadata used by the runtime
- It may additionally sign, encrypt, or otherwise protect the resulting module package

Most importantly, the post-compiler establishes a verifiable description of what a module requires in order to execute. This information can then be consumed by hosts during module loading, capability resolution, projection generation, and runtime policy enforcement.

The current verification model should therefore be viewed as an evolutionary step rather than a separate design. Static analysis and linting provide immediate architectural safeguards while the broader runtime architecture is being established. The post-compiler extends these safeguards into a more comprehensive verification pipeline without fundamentally changing how modules, capabilities, or hosts interact.

Verification is therefore not a standalone security feature. It is the mechanism through which the runtime confirms that software continues to participate in the architectural model defined by the platform

7.4 Isolation and Containment

Isolation introduces an additional layer of control by determining how functionality is exposed and how participants are separated from one another. The SDK approaches isolation as a layered concept rather than a single mechanism.

Different forms of isolation address different concerns, ranging from runtime policy

enforcement to complete process containment. These layers operate independently but may be combined when stronger guarantees are required

Runtime Isolation

The primary isolation boundary within the SDK is established through runtime projections. A capability contract defines what functionality is available to a module. The projection defines how that functionality behaves. Between the module and the host implementation, the runtime may introduce additional layers that alter, restrict, monitor, or virtualize behavior without modifying the capability contract itself. This allows the same capability to be exposed differently depending on the execution environment.

A filesystem capability may provide unrestricted access within a trusted application while exposing only a virtual workspace within a service environment. A DME provider may expose all available endpoints to trusted modules while restricting communication to a predefined subset for untrusted participants. Network capabilities may apply address translation, filtering, monitoring, or auditing before requests reach the underlying implementation.

As a result, security becomes only one possible application of the projection system. The same mechanisms may be used to implement permission management, visibility filtering, virtualization, monitoring, policy enforcement, compatibility layers, or environment-specific behavior.

Capabilities therefore define access, while projections define behavior. This distinction allows hosts to introduce additional controls without requiring modifications to modules or capability contracts

Module Isolation

Modules must remain independently loadable, unloadable, and replaceable without compromising the stability of the host environment. This requirement becomes particularly important in long-running applications and service environments where modules may be loaded dynamically throughout the lifetime of the process. The exact implementation depends on the underlying platform.

In managed environments such as .NET, modules may execute within isolated loading contexts that allow individual modules to be released without terminating the host. Native environments may employ different mechanisms while preserving the same architectural boundaries. Regardless of implementation details, modules communicate through runtime-defined interfaces rather than direct memory access.

The runtime remains responsible for determining which communication channels are visible and which interactions are permitted. This model provides both technical and architectural isolation. Modules remain independent execution units while still participating in a shared runtime environment

Process Isolation

Not all software participates directly in the capability model. Development tools frequently interact with existing command-line applications, external compilers, platform-specific utilities, legacy systems, and third-party software that has no knowledge of the SDK runtime.

Such applications cannot rely on capability contracts, projections, visibility rules, or

runtime verification. They execute independently and therefore require a different containment strategy. In these situations, hosts may employ operating-system-level isolation mechanisms. Examples include container technologies, virtualized filesystems, restricted execution environments, network isolation, application sandboxes, or other platform-specific techniques that limit access to host resources.

Unlike runtime isolation, these mechanisms operate outside the SDK architecture itself. Their purpose is not to control how a capability behaves but to constrain the behavior of software that exists outside the capability model entirely. This distinction is important.

Operating-system-level containment does not replace capability-based security. Instead, it complements the runtime architecture by providing an additional boundary around software that cannot participate directly in the SDK security model

Layered Containment

These forms of isolation are intentionally independent;

- A trusted module may execute with minimal restrictions while still participating in runtime projections

- A service-hosted module may execute within an isolated module container while communicating only through approved runtime channels

- An external application may execute within an operating-system sandbox while interacting with the SDK through DME, data adapters, or command-line integration

Each layer addresses a different category of risk and may be applied according to the requirements of the host.

The result is a security model that remains flexible without sacrificing control. Runtime projections regulate access to functionality. Module boundaries regulate interaction within the runtime. Process containment regulates software operating outside the capability model

7.5 Security as an Architectural Property

The security model of the SDK does not rely on a single mechanism. Verification, capabilities, projections, visibility rules, isolation boundaries, and host policies each address different aspects of the problem. No individual layer is expected to provide complete protection on its own. Verification cannot prevent misuse of granted capabilities. Capability contracts cannot control software operating outside the runtime model. Isolation cannot determine whether a module should be trusted. Hosts cannot enforce policies that have not been defined.

Instead, security emerges from the interaction between these layers. This principle influences the architecture as a whole. Security is not implemented inside modules, attached to individual capabilities, or delegated to a specific runtime component. It is distributed across the boundaries established throughout the system.

As a result, the same security model remains applicable regardless of whether functionality is executed locally, loaded dynamically as a module, exposed through DME, hosted within a service environment, or composed into larger applications. The

architecture therefore treats security not as a feature, but as a property of the system itself

8. Build System

The build system occupies a unique position within the SDK architecture. While the SDK consists of multiple tools, services, runtimes, and supporting infrastructure, the build system is the first tool created during environment setup and the final tool executed before software is distributed or deployed. It serves as the primary consumer of the transformation model and provides the foundation upon which the remainder of the ecosystem is built.

This role extends beyond traditional compilation. Software projects often depend on platform-specific build systems, project formats, and toolchains that are difficult to reuse outside their intended environment. Integrating the same functionality into different applications frequently requires additional wrappers, custom scripts, duplicated configuration, or direct coupling to specific development tools. Over time, this challenge expanded beyond compilation itself.
Project generation, package management, dependency discovery, repository integration, verification, and runtime packaging all exhibited similar characteristics. Each consumed data, applied conventions, and produced new data. The differences lay primarily in the processors being executed rather than in the underlying workflow. The build system models them as transformations operating on structured data. Each processor consumes data, applies conventions and contextual knowledge, and produces new data that may be consumed by subsequent processors.

Compilation therefore becomes only one possible transformation among many. The build system should be understood not as a compiler wrapper, but as the first practical implementation of the transformation model described. Whether the output is a binary, a project file, a package reference, a repository definition, a verification report, or a runtime module becomes a consequence of the selected processors rather than a fundamental property of the system itself. This approach allows the build system to remain closely aligned with the principles established throughout the SDK.

As a result, the build system acts as the bridge between architectural concepts and executable software, acting as the primary transformation engine of the SDK ecosystem. It transforms the structural model into concrete artifacts that can be consumed by runtimes, hosts, services, and development tools, while convention over configuration reduces unnecessary setup

8.1 The Build System as a Transformation Host

The build system is the first complete implementation of the transformation model. Rather than operating directly on projects, source files, or compiler invocations, the build system processes structured information through a sequence of transformations. Each stage consumes data, enriches it with context, and produces new data that may be consumed by subsequent processors.

The exact nature of the data is not important. Source code, package definitions, repository descriptions, command streams, configuration files, and

generated artifacts all participate in the same model. Processing begins with one or more sources. A source represents an origin of information that can be resolved within the context of a workspace. Sources may be backed by the local filesystem, package registries, repository definitions, generated content, or other providers introduced by the runtime.

Resolved sources are represented as leaves. A leaf forms the smallest structural unit that participates in transformation. It provides a normalized representation of data regardless of where that data originated. Leaves are subsequently aggregated into code modules. A code module represents a logical unit of processing and serves as the primary input consumed by processors. The distinction is important. Leaves describe structure and origin, while code modules describe intent and participation in a transformation pipeline.

Processors consume one or more code modules and generate new information. The produced output may itself become input for additional processors, allowing complex workflows to be composed from smaller and independently reusable transformation steps. The final result of a transformation is an artifact.

Artifacts may take many forms depending on the selected processors and execution context. Examples include compiled assemblies, native binaries, project files, package references, repository metadata, verification reports, runtime projections, capability references, or deployable module packages. From the perspective of the transformation model, however, these outputs are treated identically. They are simply the result of applying conventions and processing rules to a collection of structured inputs.

This perspective is one of the defining characteristics of the build system. Compilation, package resolution, project generation, verification, and deployment preparation are not implemented as separate workflows. They are different applications of the same transformation architecture operating on different forms of data. As a result, the build system functions as a transformation host rather than a traditional build pipeline. Its responsibility is not to produce a specific type of output, but to provide an environment in which data can be discovered, composed, transformed, and ultimately materialized as artifacts

8.2 Discovery and Composition

The transformation model depends on the ability to discover information from multiple sources and combine it into a coherent view of the system. The build system achieves this through a discovery process that operates within the context of a workspace. Rather than relying on explicit project definitions, the build system traverses available sources, resolves structural relationships, and constructs the data required by subsequent processors.

This approach reflects one of the central principles of the SDK: structure should emerge from convention wherever possible. Sources may originate from the local filesystem, package registries, repository definitions, generated content, or other providers. Regardless of origin, all sources participate in the same discovery process and are ultimately represented through a common structural model. The workspace provides the context in which this discovery occurs.

A workspace defines the environment responsible for resolving sources, applying conventions, and exposing configuration.

While a workspace may contain packages, configuration files, metadata, and other information, its primary role within the build system is to establish the context required to interpret available sources.

The root extends this process by providing a global lookup layer. Sources that cannot be resolved within the current workspace may continue resolution through parent relationships until an appropriate context is located. This allows local workspaces to remain self-contained while still participating in larger structures when required. An important consequence of this design is that the build system does not distinguish between local and distributed content at the architectural level.

A leaf originating from the filesystem and a leaf originating from a package registry are treated identically once discovery has completed. Both participate in subsequent transformations through the same interfaces and structural representations. This abstraction allows processors to operate on structure rather than origin. As a result, composition becomes a natural consequence of discovery.

Multiple sources may contribute leaves to the same transformation. Packages may extend local projects. Repository definitions may introduce additional dependencies. Generated content may augment existing structures. Each source contributes information that becomes part of a larger composition without requiring processors to understand how that information was obtained.

The build system therefore constructs a unified view of available data before transformation begins. Processors consume this composed representation and remain largely independent of the mechanisms used to discover or resolve it. Discovery answers the question of where information originates. Composition answers the question of how that information becomes a coherent whole. Together they establish the foundation upon which all subsequent transformations are built

8.3 Processors and Pipelines

Once discovery and composition have completed, processing is performed through a sequence of processors organized into transformation pipelines. A processor consumes structured data, applies a specific transformation, and produces new data as output. The responsibility of a processor is intentionally narrow. Each processor performs a single task and exposes its results through well-defined structures that may be consumed by subsequent stages.

This design follows the principle of atomic responsibility introduced earlier. Rather than constructing large monolithic workflows, complex behavior emerges through the composition of many small processors. Each processor contributes a specific transformation while remaining independent from the larger pipeline in which it participates.

The build system itself does not impose a fixed interpretation on processors. A processor may analyze source code, resolve package dependencies, generate project files, validate architectural constraints, compile assemblies, produce deployment artifacts, or generate metadata required by the runtime. From the perspective of the transformation model, these activities are equivalent. Each consumes data and produces new data.

Pipelines emerge by chaining processors together. The output of one processor becomes the input of another, allowing

information to be progressively refined as it moves through the system. Intermediate results remain part of the transformation process and may be inspected, extended, filtered, or consumed by additional processors when necessary.

This model provides several advantages:

- Processors remain reusable across different workflows
- New functionality can be introduced without modifying existing pipelines
- Multiple processors may operate on the same data without creating direct dependencies between implementations
- The build system remains extensible without requiring special treatment for new artifact types or processing stages

As the SDK evolves, new processors can participate in existing pipelines while continuing to follow the same transformation model. The architecture therefore scales through composition rather than specialization.

The practical result is a system in which compilation, verification, package management, project generation, runtime packaging, and future processing stages all share a common execution model. The differences lie in the transformations being applied, not in the architecture responsible for executing them

8.4 Commands, Configuration and Data

A fundamental characteristic of the build system is that all inputs are treated as data. Traditional build environments often distinguish between command-line arguments, configuration files, project definitions, package manifests, repository descriptions, and other forms of input. Each is processed through specialized mechanisms and frequently requires dedicated tooling or integration layers. The transformation model does not make this distinction.

From the perspective of a processor, the origin of information is largely irrelevant. What matters is the structure of the data being consumed and the transformation being applied. A command stream, for example, represents structured input describing a requested transformation. Configuration files describe structured input intended to influence processing behavior. Package definitions describe structured input used for dependency discovery and composition. Repository definitions describe structured input used for source resolution.
While these artifacts may appear different from a user perspective, they share a common architectural role. Each contributes data that becomes part of the transformation process. The distinction therefore exists primarily at the level of representation rather than meaning.

A value provided through a command-line argument may express the same information as a value stored in a configuration file. Likewise, information contained within a package definition may ultimately influence the same processors that consume command streams or workspace configuration. The build system normalizes these inputs into common structures before processing begins. Once normalized, processors operate on the resulting data rather than on the mechanism through which that data was supplied. This approach simplifies both composition and extensibility.

New input formats may be introduced without requiring changes to existing processors. Additional processors may consume existing data without requiring modifications to command-line interfaces or configuration systems. Different representations may coexist while continuing to participate in the same transformation pipeline. As a result, commands are not treated as special instructions executed directly by the build system. They are treated as data that describe a desired transformation.

The same principle applies to configuration, package metadata, repository definitions, and other structured inputs throughout the SDK ecosystem. Each contributes information to the transformation model and participates through the same underlying processing architecture. This perspective reinforces one of the central ideas of the SDK. Transformation operates on data. The representation of that data is a secondary concern

8.5 Verification and Runtime Code Modules

Verification serves a broader purpose than enforcing architectural constraints. While the verification process is responsible for identifying unsupported behavior and validating compliance with SDK conventions, it also establishes the relationship between code modules and the runtime architecture. This relationship is essential because the runtime depends on clearly defined contracts that can be consumed by hosts, capabilities, and other modules without exposing implementation details unnecessarily.

A code module does not become part of the runtime merely because it provides functionality. Runtime participation is determined through context. When a code module is designated as a runtime component, additional processing stages are applied during verification. These stages identify exported contracts, consumed contracts, runtime dependencies, capability boundaries, and other information required by the runtime architecture. Verification therefore performs both validation and classification.

It ensures that runtime code modules conform to the expectations of the platform while simultaneously collecting the information required for later composition and execution. One important result of this process is the generation of reference assemblies.

Reference assemblies are derived directly from runtime code modules and preserve the public contracts exposed by those modules. Unlike purely descriptive interface definitions, reference assemblies contain a functional implementation generated from the originating code module. The reference assembly serves two different purposes:

- During development it provides a concrete implementation that can be consumed by development tools, IDEs, debuggers, tests, and other processors without requiring the complete runtime environment to be present

- During runtime it acts as the foundation from which runtime projections can be generated and bound to host-provided implementations

Because reference assemblies originate from the runtime code modules themselves, the contracts used during development remain consistent with those available

during execution. This reduces duplication, avoids manually maintained interface definitions, and ensures that runtime behavior remains aligned with the structures used throughout the development process.

The build system therefore treats runtime code modules as first-class participants in the transformation model. They are discovered, composed, verified, and processed through the same mechanisms as any other code module. The difference lies only in the additional information produced during verification and the generation of runtime-specific artifacts required by hosts and module consumers. In this way, verification forms the bridge between the transformation architecture of the build system and the execution architecture of the runtime

8.6 Build Artifacts

The final result of a transformation is an artifact. Artifacts represent the materialized output of one or more processors and form the primary mechanism through which information leaves the build system and becomes available to other tools, services, runtimes, or deployment environments. The architecture intentionally avoids assigning special status to any particular artifact type.

Compiled assemblies, native binaries, generated project files, package metadata, repository information, verification reports, runtime code modules, reference assemblies, and deployment assets are all treated as products of the same transformation model. This approach provides a consistent processing pipeline regardless of the final output being generated.

New artifact types can be introduced without requiring fundamental changes to the architecture. Existing processors may contribute additional information to previously generated artifacts, and future tooling can consume build outputs without requiring knowledge of how those artifacts were originally produced. The build system therefore focuses on the transformation process rather than the artifact itself.

Artifacts are considered the result of composition, convention, and transformation applied to structured data. Their format and purpose may vary, but their origin remains the same. This principle concludes the role of the build system within the SDK architecture.

The build system is not defined by compilation, packaging, or deployment. It is defined by its ability to discover information, compose structure, perform transformations, verify architectural intent, and materialize the resulting artifacts. As the first complete implementation of the transformation model, it demonstrates how the concepts described throughout this paper become executable software, reusable modules, and deployable systems

9. Workspaces and Package Distribution

The SDK is designed around the principle that development environments should adapt to the requirements of projects rather than requiring projects to adapt to predefined environments. This principle extends beyond tooling and runtime architecture. It also influences how software is distributed, shared, and composed across teams, repositories, and applications.

Traditional package management systems typically focus on dependency distribution. Packages are installed into a project and consumed as external artifacts. While effective for binary distribution, this approach often separates reusable functionality from the structural context in which it was created. The SDK adopts a different perspective.

Packages are not primarily mechanisms for distributing dependencies. They are mechanisms for extending and composing workspaces. A package may contribute source code, runtime components, tools, repository definitions, assets, configuration, conventions, or other forms of structure. These contributions become part of the workspace in which the package is resolved and participate in discovery, composition, and transformation through the same mechanisms as local content.

This approach allows development environments to evolve alongside the projects they support. Instead of adapting projects to a fixed collection of tools and conventions, workspaces can be extended through composition. New capabilities can be introduced through packages, additional repositories can become available through package metadata, and specialized workflows can be distributed alongside the projects that require them.

As a result, package management becomes an extension of the composition model described earlier: packages compose workspaces, workspaces compose sources, sources produce leaves and leaves participate in transformations. The result is a distribution model in which collaboration is achieved through the composition of structure rather than the installation of isolated dependencies

9.1 Workspaces as Composition Contexts

A workspace represents the primary context in which package composition occurs. While workspaces have already been introduced as structural containers for sources, configuration, metadata, and packages, their role within distribution is equally important. A workspace defines how external contributions become part of a project and how those contributions interact with existing structure.

From this perspective, packages should not be viewed as additions to a project. They are additions to the workspace itself. When a package is resolved, its contents become part of the workspace composition process. Sources contributed by the package participate in discovery. Configuration contributes to contextual information. Repository definitions may introduce additional locations from which further content can be resolved. Runtime code modules, tools, assets, and other structures become available through the same mechanisms as locally defined content. This model allows workspaces to evolve through aggregation.

Composition follows the same resolution model established elsewhere in the SDK. Local context takes precedence over inherited context. Packages resolved within a local workspace therefore override equivalent structures originating from broader scopes such as the root workspace. This behavior allows projects to specialize or replace functionality without requiring changes to shared environments.

The root workspace continues to provide common conventions and globally available structure, while local workspaces remain free to extend, refine, or override those

conventions according to project requirements. This balance between inheritance and specialization is essential to the composition model. Packages act as the mechanism through which both can evolve without sacrificing either objective.

A package therefore contributes more than content, it contributes context. Through composition, packages influence how workspaces discover information, interpret structure, apply conventions, and perform transformations. Distribution is consequently not limited to code reuse but extends to the reuse of workflows, tooling, capabilities, and development environments themselves

9.2 Package Identity and Context

Packages participate in workspace composition through identity and context rather than through rigid classification. Traditional package management systems frequently distinguish between package categories such as libraries, applications, plugins, or tools. While these distinctions may simplify distribution, they often introduce assumptions about how packages are intended to be consumed.

The SDK intentionally avoids this approach. A package is defined by its contribution to a workspace rather than by a predefined package type. The same package may participate in different environments and fulfill different roles depending on the context in which it is resolved. Package identity provides the foundation for this process.

Each package exposes identifying information through the distribution mechanism from which it originates. The exact structure of this identity is intentionally independent from any specific registry implementation. Different registries may employ different identity models while continuing to participate in the same workspace composition process. Identity alone, however, does not determine meaning.

The role of a package emerges from the combination of identity, configuration, conventions, and the structures contributed to the workspace. A package may introduce runtime code modules, tools, source code, repository definitions, assets, processors, or other forms of functionality. These contributions are interpreted according to their context rather than through a fixed package classification system. This approach allows the same composition model to remain applicable across a wide range of use cases while avoiding the need for specialized package formats. Context therefore becomes more important than type.

Packages are not categorized according to what they are intended to be. They are interpreted according to how they participate in workspace composition. The result is a system in which packages contribute structure, capabilities, and behavior without requiring the architecture to impose rigid distinctions between different forms of reusable content

9.3 Peer Dependencies and Aggregation

Composition within the SDK extends beyond simple dependency resolution. While packages may reference other packages through conventional dependency relationships, peer dependencies serve a different purpose. Rather than expressing only compatibility requirements, they define aggregation relationships between packages.

A peer dependency indicates that the referenced package participates directly in the composition of the consuming package. This relationship allows functionality to be merged into a larger structural unit rather than existing as an isolated dependency resolved independently at runtime or execution time.

The build system provides a practical example of this behavior: support for additional platforms, languages, project generators, or processing stages can be implemented as separate packages while remaining part of the same transformation environment. When these packages are declared as peer dependencies, they are automatically aggregated into the consuming package during composition.

As a result, functionality such as C++ support or Visual Studio project generation can be developed independently while still becoming part of the build system itself when the workspace is composed.
This mechanism encourages modular development without sacrificing integration. Capabilities remain isolated during development but become part of a larger composition when required. The resulting structure behaves as a coherent system while preserving the organizational benefits of smaller and independently maintained packages.

Peer dependencies therefore represent more than dependency metadata. They are an explicit composition mechanism that allows workspaces, tools, and runtime environments to evolve through aggregation rather than through monolithic implementation

9.4 Repositories and Decentralized Distribution

Repositories provide the distribution layer through which packages become available to workspaces. Their primary responsibility is not the storage of artifacts but the publication and discovery of reusable structure. Repositories expose packages, metadata, and related information in a form that can participate in workspace composition while remaining independent from any particular implementation or hosting model.

This design intentionally avoids a dependency on centralized infrastructure. A repository may be public or private, local or remote, organizational or personal. Multiple repositories may coexist within the same workspace and contribute packages to the same composition process. The architecture does not require a single authoritative source of distribution and does not assume that all participants operate under the same administrative control. Repositories therefore function as providers of context rather than merely providers of content.

A repository introduces a namespace in which packages can be discovered, identified, versioned, and composed. The specific mechanisms used to implement these capabilities remain independent from the architectural model, allowing different repository implementations to coexist without affecting the composition process itself. This flexibility is particularly important for long-term sustainability.

Projects, organizations, and communities evolve over time. Distribution mechanisms change, hosting providers disappear, and infrastructure requirements shift. By separating package composition from repository implementation, the SDK allows workspaces to adapt without requiring changes to the underlying transformation model.

Repositories consequently become part of the composition ecosystem rather than external infrastructure attached to it. They provide the means through which reusable structure can be shared, discovered, and incorporated into workspaces while preserving the decentralized nature of the architecture

9.5 Collaboration Through Composition

Sources, workspaces, leafs, code modules, capabilities, transformations, packages, repositories, and runtimes are not isolated concepts. Each represents a different perspective on the same architectural principle. Complex systems should emerge from the composition of smaller and independently maintainable structures. This principle extends beyond software architecture and influences how projects are shared and developed.

Contributors do not merely exchange binaries or source files. They exchange structure. Packages may introduce new capabilities, development tools, runtime components, processing stages, workflows, conventions, repositories, or complete workspace extensions. These contributions become part of a larger composition while remaining independently reusable. The resulting ecosystem encourages incremental growth.

Small contributions can participate in larger systems without requiring centralized ownership of the entire architecture. Individual components can evolve independently while continuing to cooperate through shared conventions and contracts. New functionality can be introduced through composition rather than modification. This model also reflects the broader design philosophy of the SDK:

- Open source is not viewed solely as a distribution model. It is a collaboration model
- Atomic responsibility is not merely a software engineering principle. It is a mechanism for enabling independent contribution
- Composition is not only a technical implementation strategy. It is the process through which individual contributions become larger systems

The SDK therefore treats software development as an exercise in assembling structure from reusable building blocks. Workspaces provide context. Packages provide composition. Transformations provide evolution. Runtimes provide execution. Together these elements form an architecture in which development environments adapt to the needs of projects, functionality remains reusable across domains, and collaboration emerges naturally through composition

10. Bootstrap and Distribution

The SDK is distributed according to the same architectural principles that govern its design. Rather than shipping a complete development environment, the SDK is distributed as a minimal bootstrap package containing only the components required to establish an initial workspace and construct the remaining environment through composition.

The bootstrap distribution currently consists of a root workspace, the runtime packages and code modules required by the build

system, supporting metadata, and a small platform-specific payload. The payload is responsible for locating a compatible .NET SDK, initializing the build environment, and generating the initial build system instance. Once this process has completed, responsibility is transferred to the SDK itself.

From that point onward, package management, workspace composition, tool generation, runtime components, and subsequent updates are handled through the mechanisms previously described. The build system becomes both the first consumer of the architecture and the foundation upon which the remainder of the environment is constructed. This approach intentionally minimizes the size and complexity of the initial distribution.

Instead of requiring users to download and maintain a complete toolchain regardless of project requirements, the SDK begins with a minimal installation and expands through workspace composition. Additional functionality is introduced only when required and remains subject to the same discovery, transformation, and package management processes used elsewhere in the ecosystem.

The result is a development environment that is assembled rather than preinstalled. Projects acquire the tools, capabilities, and runtime components they require through composition, while the initial distribution remains small, portable, and independent of any specific workflow. In this sense, the SDK not just defines strong principles but is also distributed according to them as well