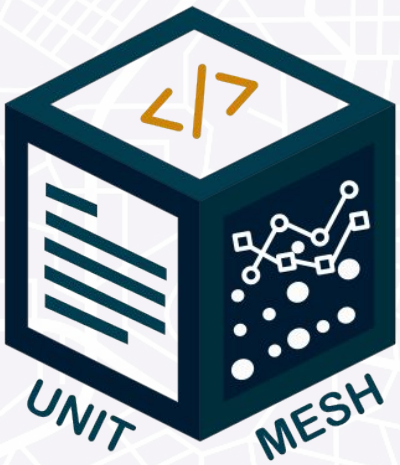
生成式 AI 与开源重塑软件研发

探索新架构开发范式，构建下一代研发组织



https://github.com/unit-mesh

An open-source AI 2.0 + SDLC solution initiated by Thoughtworker

扫描二维维，关注公众号
获取**最新 AI + 软件研发**洞见

目录

软件研发的新潮与旧阻

放大的局部加速或扰乱全局

我们相信成熟的 AI 工具，能更好的提升开发效率。然而，限制交付速度往往不一定是开发速度。

如下图所示，当生成式 AI 放大现状之后，会呈现两个值得思考的新挑战：

- 我们能**更快产生需求**吗？
- 我们能**更快交付功能**吗？

由于生成式 AI 的放大器效应，如果只提升了中间部分的效率，总体的交付节奏不仅不会变化，甚至还可能引起一些混乱。

研发模式成熟度制约增益

如果企业的研发流程、模式不够成熟，不能以 BizDevOps 的思维从全景考虑问题，那么生成式 AI 所带来的提升就将变得极其有限。

对于不同的组织来说，会影响程度是不同的。诸如金融机构过去面临的是：

- 传统稳态和敏态模式共存下的快速业务响应的挑战。
- 传统组织架构协作模式所带来的挑战。
- 效能协作平台灵活性与扩展性演进所面临的挑战。

归纳来说，其集中表现在业务响应、组织架构协作、效能平台演进三个方面。

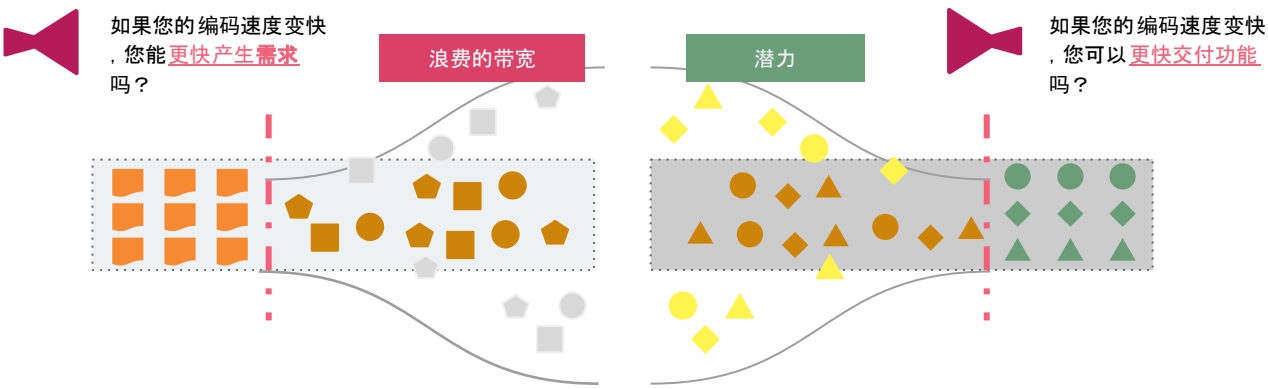
除此，大部分金融企业里还要面临：历史遗留系统演进与维护所面临的挑战。在几个不同因素的作用之下，问题将比原来更显复杂。

生成式 AI 的潜能预期

从国内外企业的探索方向来看，组织预期生成式 AI 能：

1. 增强已有的研发流程。加速编码速度、缩短代码检视时长等。
2. 解决过去的遗留问题。解决需求、编码等规范不落地，技术实践缺失等。
3. 探索新的研发模式。

不同的目标所驱动的路径有所差异，但是有诸多的相似点，如短期目标是：AI 增强人类的能力。



当生成式 AI 放大现状之后

CHAPTER 1:
生成式 AI
如何赋能软件研发

生成式 AI 时代的软件研发

提升个人与团队效能

当前阶段，生成式 AI 明显对个人和团队的提升，加速需求编写、代码实现、软件质量提升。而现有的流程和工具存在瓶颈，或对生成式 AI 并不能很好的支持，以及学习工具产生的额外成本，导致组织整体的效能提升并不明显。

注意：实际花费的成本 - 准备提示词（prompt）的成本 = 降低的成本。也因此，在将 AI 与工具进一步结合后，效能的提升才会更加明显。

改善规范不落地

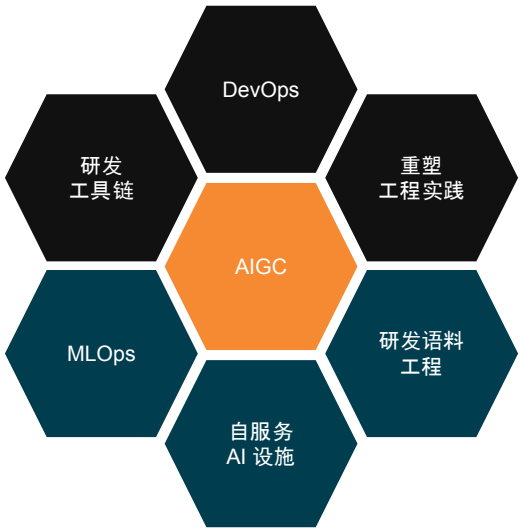
在中大型组织里，由于过去的软件开发流程中种种规范执行不到位，导致上线的软件质量差强人意。而在结合工具后，生成式 AI 可以生成规范化的需求、代码、测试用例等，加速规范的推广。

在经过小范围的探索和试点之后，部分研发组织负责人认为：生成式 AI 可以改善规范和质量，并实现弯道超车。

加速知识传递

生成式 AI 可以大大加快个人学习上手新技术栈，熟悉当前代码上下文，进而加快进入的交付速度。

对于组织来说，结合生成式 AI 、传统检索与 RAG（检索增强生成）技术，可以加快组织内部的知识流动，让信息更易于获取。



不可忽视的挑战

代码质量与安全。在不考虑使用生成式 API 潜在的信息安全暴露风险，组织内也会由于：大量未经仔细甄别、不准确的生成式代码，进而会扩大代码的安全风险。

限制快速迭代的工具与流程。限制迭代速度的，并非只有开发速度，有时可能是内部的工具或者流程。因此只有优先改善 DevOps 能力，才能更好地借力于 AI 。

AI 工具落地难。开发应用时，也会遇到各类挑战：难用的第一个版本限制后续推广、高质量研发语料、如何围绕用户的 AI 体验设计等。

除此，对于金融机构等组织来说，由于网络和安全的限制，挑战扩大到了模型层：他们需要从模型到工具思考问答，选择购买或自建生成式 AI 能力。

生成式 AI 在软件研发中的应用

从研发旅程出发的试点探索

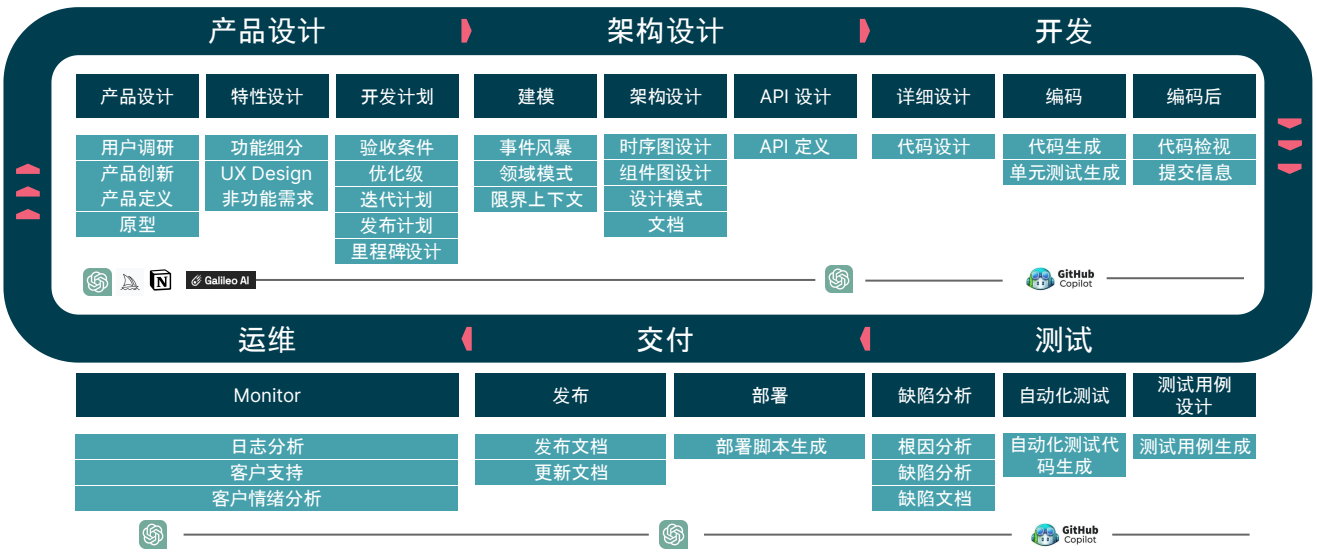
在 2023 年，已经有大量的团队在不同环节采用生成式 AI —— 使用最好的模型，结合自己的研发旅程，探索提升比较明显的节点。并基于此，建立初步的提效评估体系，诸如**接受率、入库率、生成比率**等。

随后，根据自己的企业上下文，或采购国内的 AI 服务、搭建开源 AI 模型，构建自己的 AI 工具，以单点或者多环节上达到更好的提升效果。

单点突破，成效明显

在**编码、测试**两个环节，生成式 AI 效果比较明显，而**需求**环节也是企业中探索非常多的一个环节，此外其也帮助开发人员**快速学习**语言细节、框架细节等知识。

在编码环节，预期其带来的效率提升在 20～50% 左右，效果受不同的语言、场景影响较大。从 Thoughtworks 在国内外采用 GitHub Copilot 的效果来看，静态语言效果优于动态语言，差异在 5% 左右，后端效果优于前端，差距在 10% 左右。



BizDevOps工具链革新

随着工具的成熟和采用率的增加，所有环节的效益都将提高 2 至 3 倍以上。

根据现阶段的优先级和效果，目前关注度最高的是：开发和测试环节，其次是影响比较大的需求，即产品设计环节。

而在运维侧，部分组织已经在过去结合传统 AI 来解放运维，现在也均在采用生成式 AI 来强化优势领域。

此外，我们也可以看到在辅助进行架构设计、架构治理、代码治理、持续集成等环节，也将有所采用和探索。除此，结合内部的 API 市场、知识社区、研发规范等，也有大量的 AI 问答工具产生，以及结合知识库场景的代码生成。

生成式 AI 对于软件架构的影响

生成式 AI 对软件架构的影响，关注点主要在三方面：

- 如何开发好生成式 AI 应用？
- 整体架构的影响与变化？
- 如何结合生成式 AI 治理架构？

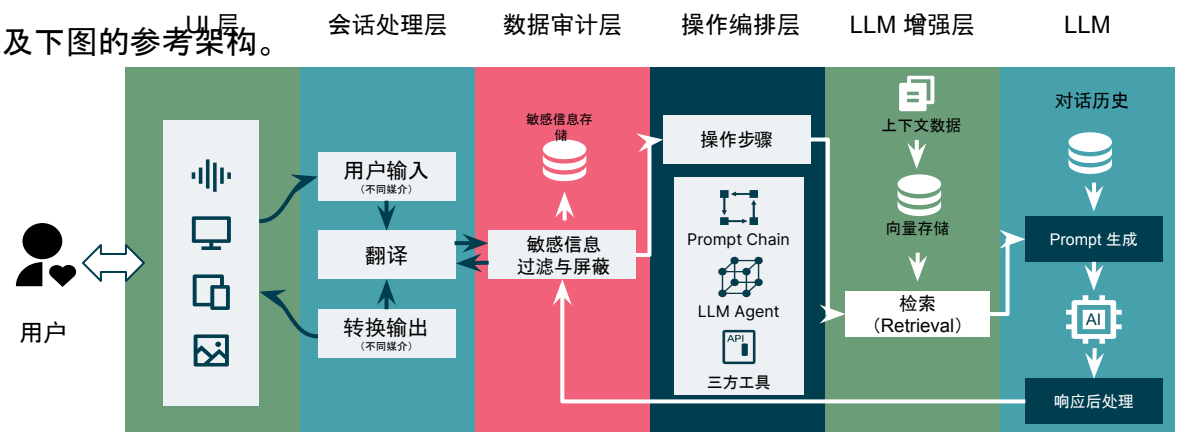
而且相关的影响还在持续变化中。

生成式AI 原生应用的架构范式

结合的项目开发经验，我们总结了生成式 AI 应用的四个架构原则：

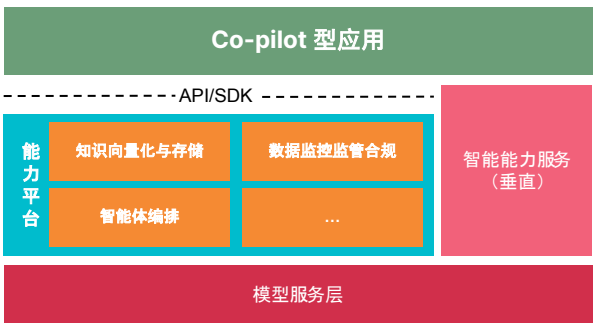
- **用户意图导向设计**。设计全新的人机交互体验，构建领域特定的 AI 角色，以更好地理解用户的意图。
- **上下文工程**。构建适合于获取业务上下文的应用架构，以生成更精准的 prompt，并探索高响应速度的工程化方式。
- **原子能力映射**。分析 LLM 所擅长的原子能力，将其与应用所欠缺的能力进行结合，进行能力映射。
- **语言 API**。以 DSL 作为 API，便于 LLM 对服务能力的理解、调度与编排。

以及下图的参考架构。



技术架构变化显著

在中大型组织内部，可以明显看到生成式 AI 带来的技术架构变化。基础设施团队开始构建 LLM 即服务的模型服务平台，同时还有垂直能力的服务，以及围绕于向量存储、知识管理等的 AI 2.0 平台，还有位于 AI 应用与平台之间的各类 SDK 。



示例：基于生成式 AI 的技术架构示例

加速架构演进，治理愈发关键

随后，编码速度的提升，更加合理和牢靠的架构变得愈发重要。我们需要结合 AI 和治理工具加速软件架构的迭代过程，让架构能更快地演进，以避免成为研发速度的新瓶颈。

CHAPTER 2: 场景与案例

智能需求工程

需求是我们迈向 AI 直接生成可执行代码最难的一步。在有了详尽、准确的需求，结合已有代码与检索技术，能生成测试用例、代码，并自动进行测试、部署等。

需求的形式体现是**内容创作**，好的需求文档严重依赖于创作者，创作者需要知道该领域的表达语言、需求的过去实现、功能的未来方向等等。

结合生成式 AI 意味着，在构建对应的生成工具时，要具备如下的特征：

- **结构化需求格式生成**
- **文档知识工程**
- **沉浸式 AI 创作体验**

由此，将其称为**工程**，即它并非易事。

结构化需求生成

在形成了结构化的需求格式，并结合了格式进行**适当的模型微调**之后。我们还应该关注的点是：**需求产出物要易于与 AI 代码生成工具相结合**。即我们要实现的是：**大模型友好的需求描述格式**。

诸如于，用户故事这种需求格式包含了验收条件，即 Given-When-Then 的三段式表达。对应到单元测试上，便是 should_xx_when_xx 及其变体的命名方式；对应到集成测试上，可直接与 BDD 风格 / Cucumber 表达相结合。对应到内部的需求格式、测试框架，可能产生细微的差异。

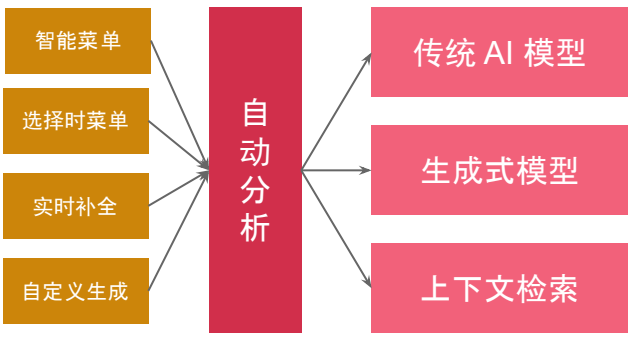
示例：典型的需求生成工具示例

文档知识工程

需求文档涉及到了大量的领域知识，往往需要结合 RAG 与传统搜索才能完成的。而在不同团队会出现巨大差异：有的需要结合 RAG 从代码库理解原来业务逻辑；有的团队包含大量的文档，需要进行过滤、筛选等预处理，再进行相关的 AI 工程化。

沉浸式 AI 创作体验

在绝大部分场景下，AI 无法一次生成满足条件需求，还需要基于生成结果进行修改。那么，便需要结合人的编辑过程来思考，结合编写习惯，将 AI 能力融入到需求编写过程中，诸如于在 UI 上的集成。



示例：AI 文本编辑器 Studio B3 的触点

构建端到辅助编程

对于**中小型团队**或者**数据不敏感**的团队，直接采购成熟的辅助编程服务或者私有化部署，便能实现对于编程的辅助。

对于**中大型组织**来说，则需要根据自身的需求来自研辅助编程能力。这时就需要在不同的模型间选择、数据集构建，并考虑后续的持续模型微调及 IDE 插件的构建。

IDE 即辅助能力中心

在 IDE 侧，通常会继续集成组织内部的其他能力，诸如于 API 市场、基础设施文档等。结合生成式 AI 的能力，即能其根据开发人员的上下文、需求，生成符合对应背景的代码；又能提供问答能力，降低平台团队人力成本。

三个模型：大中微

在模型侧，我们需要平衡体验-速度-成本，诸如于：

- **交互场景**：更大的语言模型（32B+ 规模），在回答用户问题、解释代码等场景效果最好。
- **补全场景**：普通量级（6B～ 12B 规模），补全场景对**响应速度要求高**，模型过大会上升模型成本，下降性价比。

在实现更好的补全效果、问答时，还需要使用**本地 embedding 模型**，由于代码是结构化的，因此从业内实践较多的 20～80 M（文件大小）左右的模型对用户安装较为方便。

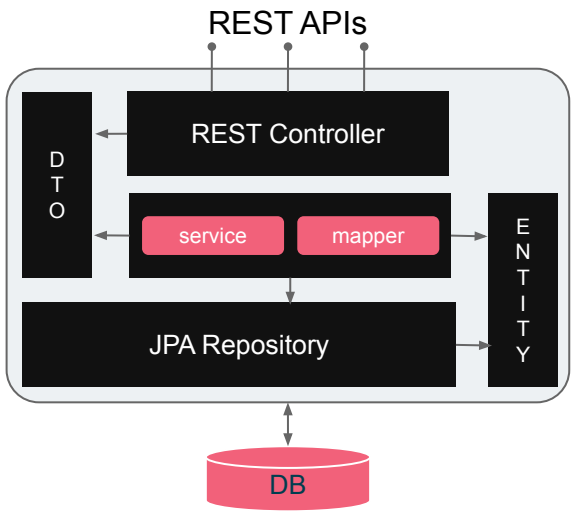
示例：辅助编程技术架构示例

规范化的代码生成

在生成式 AI 进一步普及后，我们将面临的 AI 生成代码不合规范的挑战。

- 在后端，问题是**生成的代码不规范**。需要结合规范生成，或者进行微调。
- 在前端，问题是生成的 UI 与现有的组件库不一致，导致代码接受率下降，需要结合组件库生成或进行代码微调。

不论是哪种方式，我们都需要持续管理、监测生成代码的质量与安全。



示例：典型的 Java MVC 架构示例

软件研发质量提升

测试生成与测试数据生成

在测试改善质量的环节里，我们可以看到四个主要的应用场景：

- 测试设计与用例生成
- 单元测试生成
- 端到端测试生成
- 测试数据生成

代码生成准确性依赖于业务上下文，而测试只依赖于已实现的功能，或者需求文档。在大量软件质量要求高的组织里，实践到的结果是：**AI 生成测试的接受率远高于生成业务代码**，甚至可能**达到业务代码的 2 倍**。

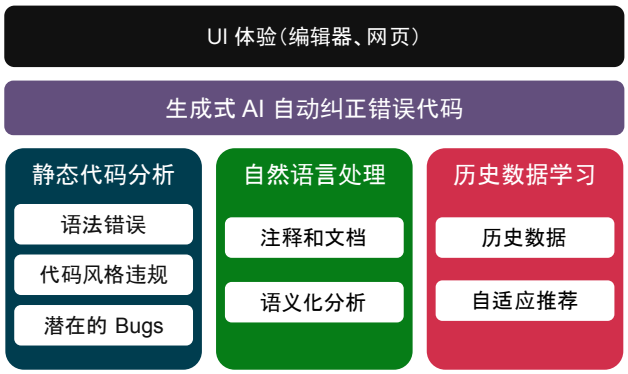
AI 测试辅助平台架构示例

在结合业务知识、测试资产等知识之后，AI 可以更好地辅助我们解决测试问题，对测试过程和结果分析，诸如结合错误日志进行总结等。

尽管，我们也看到了 AI 在其它测试环节有非常不错的潜力，但是它更适合解决某个**较为具体**的任务。

改善既有代码质量与设计

结合 Thoughtworks 的实践经验，以及 Google 和 Turing AI 在 ML/AI 结合代码检视总结，我们认为生成式 AI 在代码质量检查上的侧重点应该是：**帮助开发人员解决已发现的问题**。既通过传统的静态代码分析或者机器学习分析，找到代码中的潜在问题，由生成式 AI 修复问题，并生成 merge request 或者 pull request 交由开发人员来决定是否处理。



AI 代码质量检查（基于 Turing AI）

需要注意的是：代码检视是**在团队内共享知识**的一种方式，现阶段不建议**完全**交由 AI 来生成。除此，我们应该考虑结合 IDE、DevOps 流水线等不同阶段的工具，来协同完成这一目标。

生成式 AI 在帮助我们解决：单个类、数据结构、文件窗口的重构时，效果相对比较好。但是，对于复杂上下文，其能力是比较有限的，重构结果也不理想。

模型微调与高质量语料

改善生成质量的模型微调

受限于中大型组织内部大量的基础设施、领域特定语言，通用的大语言模型生成出来的结果，接受率并不理想。通常做法是，在开源大语言模型的基础上，结合**开源数据集、外部数据集、内部语料**，进行模型微调。随后，围绕于试点团队持续优化，并构建持续反馈环。

从结果来看，结合企业代码模型微调、训练，可以产出**更规范化的代码**。并在在**特定领域**，如在数据协议、通信协议，接受率的提升会达到 10%+。

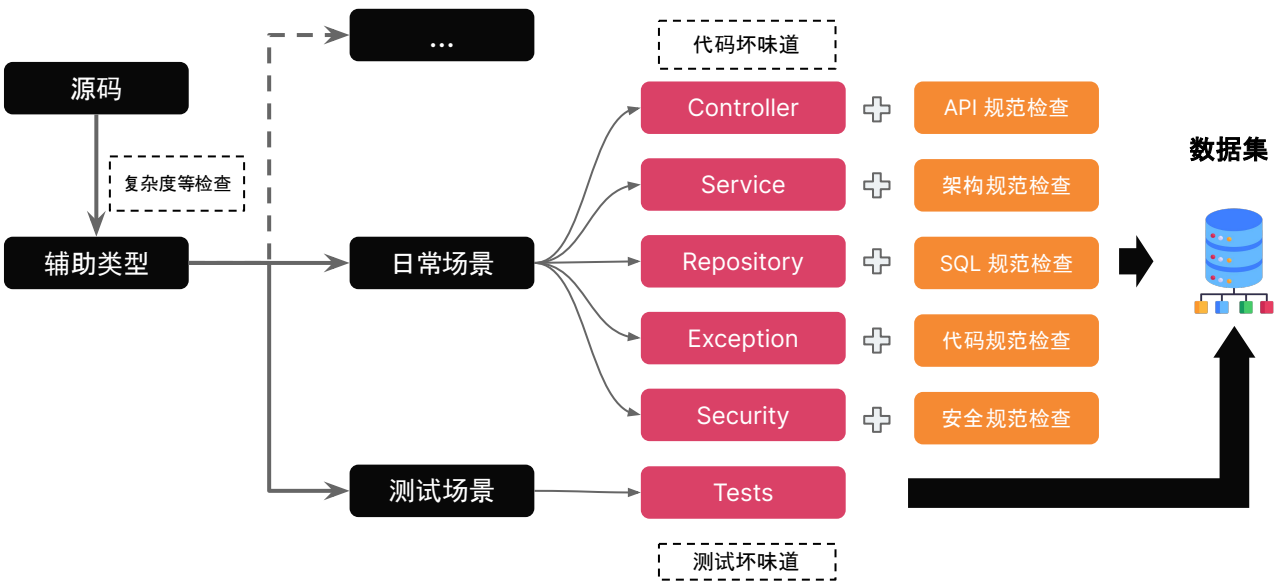
在选开进上，组织应该优先关注于生态丰富的开源模型，诸如 **LLaMA 架构**（非指模型），又或者是更好的**中文支持**。其次是，如何与工具链构成闭环的设计，最后则是语料相关的数据工程。

高质量语料的数据流程线

除了微调技巧，大型组织还面临数据挑战：从研发资产中筛选出高质量语料。它们可能是：

- 功能详尽的需求文档
- 边界完备的测试用例
- 优质的业务代码、测试代码

在不同的语料与 AI 工具场景之下，所采用的**数据要求和筛选机制**是不同的。诸如于，如果我们的目标是：与 IDE 相结合的补全微调。那么，我们需要根据组织内的规范，构建对应的代码检查机制 —— 根据组织规范、代码坏味道、分层类型。随后，再结合 IDE 所需要的相关上下文，来构建最后的微调指令，下图 Unit Eval 项目中设计的代码质量筛选机制。



示例：Unit Eval 代码筛选流水线

CHAPTER 3: 解决方案设计

场景驱动的辅助编程插件设计

辅助编程工具的架构会受限于在设计整体架构，通常采用两不同种模式：

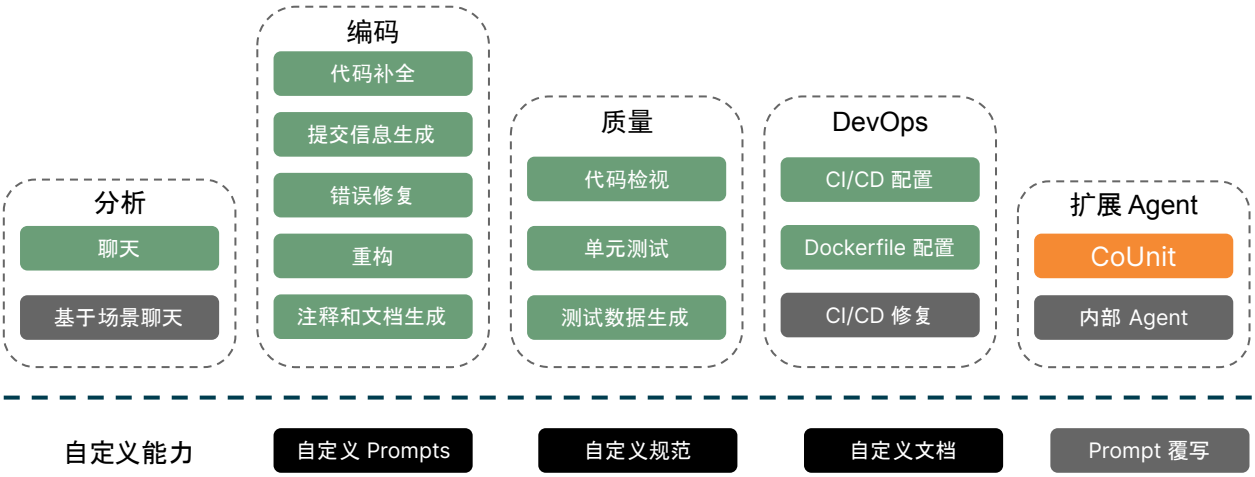
- **跨平台架构**。即提取基础的能力，如近期阅读、修改代码的相似性计算等，以实现更好的跨编辑器、IDE 支持。
- **平台绑定架构**。与相关 IDE 的能力紧密结合，如借助 JetBrains IDE 上的静态代码分析，构建更精准的上下文。

跨平台开发成本较低，准确性略微低。

平台绑定架构 AutoDev

AutoDev 是基于 JetBrains IDE 平台的插件，深度与其平台接口绑定。而借助平台的静态代码分析能力，能在单元测试、代码生成等场景，构造更准确的上下文。

在算力不足时，通过准确性的提升来弥补一部分差异，再提供更好的自定义能力，并结合类似于 CoUnit 的内部文档和 API 实现集成与对接。



结合模型能力的场景设计

在补全、注释、代码检测、单元测试等场景，借助小模型可以取得不错的效果，并且响应速度快。

而对于大部分组织来说，内部的系列基础设施如 CI/CD、文档等也期望接入到 IDE 中，以快速实现代码。因此，我们需要在 IDE 上构建结合 function calling 的能力，让模型能根据用户的上下文，来接合不同的 AI Agent 来帮助用户解决对应的问题。

上图是 AutoDev 结合内部、外部、社区的需求所构建的已有功能、少部分未来能力（灰色部分）的功能全景。

- 在模型上，支持用户接入自己的模型，用户可以使用公开或者私有化的模型。
- 在功能上，AutoDev 充分借助社区的反馈，构建丰富、可扩展的功能体系。

用户可以根据自己的场景，自定义对应的 IDE 助手功能、结合组织的规范用于自动生成代码等等。

研发角色的 Co-pilot 工具设计

生成式 AI 工具辅助形式

根据不同角色的岗位定位，工具产生的形式也略有不同：

- 编辑器为中心的全面辅助。即类似 IDE 在各个环节提供 AI 辅助创作能力。
- 过程式辅助。即在某一环节，结合生成式 AI 来提供辅助，如判断结果等。
- 对话式辅助。即通过问答的方式，来向用户提供更多的知识、指导等。

每种模式都有各种适用的不同场景，相同场景根据不同背景，会选择不同的形式。

1. 编辑器为中心的全面辅助

从市面流行的工具，可以看到明显的趋势：以**用户创作旅程为中心**构建全面的 AI 辅助能力。

- 在代码侧，编程插件提供了补全、聊天、测试等一系列的功能。
- 在 UI 侧，类似于 v0.dev 提供可交互的 UI 设计编辑器，直接在 UI（即组件）上对话需求，以生成前端代码。
- 在文本侧，如 Notion 等编辑器提供了全面的 AI 辅助创作。

在需求、测试用例等场景上，同样需要历史上下文、资料库等进行相关的工作。为了更好的与现有的各类系统相结合，我们创建了 Studio B3 项目，以探索一站式需求编写工具，同时封装底层 AI 编辑器抽象，作为能力的一部分，支持其它场景。

2. 过程式辅助

根据不同的角色职责，在工具、流程中引入生成式 AI 进行：结果判断、丰富细节等。

如下是 ArchGurad Co-mate 用于结合生成式 AI 治理架构的 DSL，其中的 security, misc 会结合模型进行检查，剩余的内容则是结合传统的代码分析工具检查。

```
rest_api {
  uri_construction {
pattern("/api\\/[a-zA-Z0-9]+\\/v[0-9]+\\/[a-zA-Z0-9\\/\\
-]+")
  }
  http_action("GET", "POST", "PUT", "DELETE")
  status_code(200, 201, 202, 204, 400, 401, 403, 404, 500,
502, 503, 504)
  security("""Token Based Authentication
(Recommended) Ideally, ...""")
  misc("""...""")
}
```

示例：ArchGuard Co-mate 治理 DSL 示例

除此，我们构建了试验式的项目 DevOps Genius，以探索流程工具中的可能性。

3. 对话式辅助

在对话式上，在设计系统时考虑更多的是，如何引导用户输出系统所需的上下文，以清晰用户的真正意图。

其通用的构建方式是：围绕该领域的上下文，以 JSON 等数据格式为基础，将每个值作为一个槽位，收集用户的相关信息，以完成对应的诊断工作。在我们实施过的案例里：自然语言转 SQL、与系统交互是比较多的应用场景。

智能应用基础设施

智能应用框架

对于组织而言，构建应用框架/SDK 可以沉淀相关的开发模式、示例实践指南等。

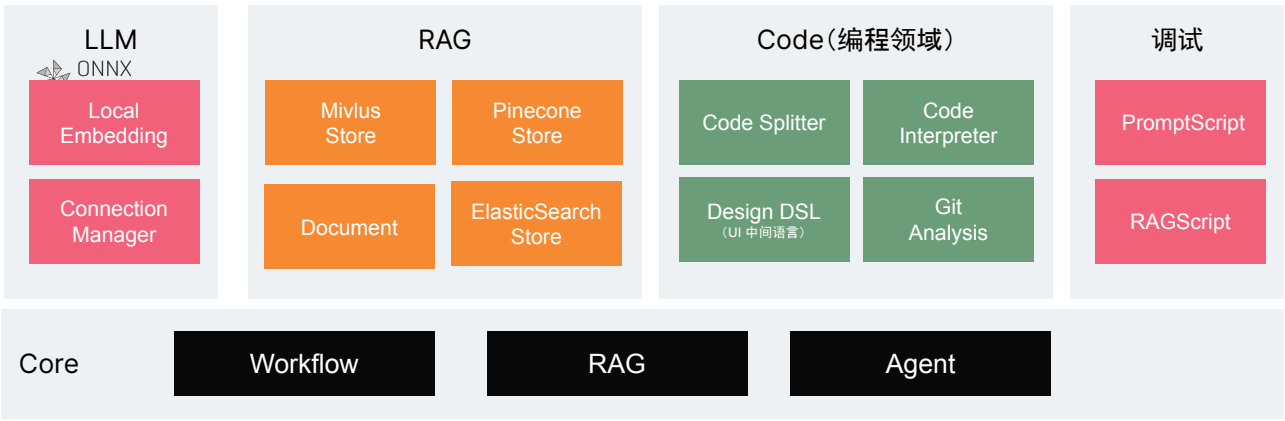
在我们发现 Langchain 不能与**现有**基础设施结合时，便构建了 JVM + Chocolate Factory 作为我们的 AI 辅助研发开发体系。在能力上，该 SDK 提供了模型接入、RAG 抽象接口与实现等基础能力。

在软件研发场景开发应用时，它还包含了大量研发领域特有的模块。以及研发结合生成式 AI 的相关模块，更好的将**代码切分**成 chunk、代码向量化（embedding）、本地向量化能力等。

除了支撑 Unit Mesh 的相关开源应用，我们还构建了大量的辅助研发示例，诸如于**基于组件库的辅助前端代码生成**、**语义化代码搜索与解释**、知识问答等。

同时，我们使用其中的 **RAGScript** 来展示如何快速开发一个 RAG 应用。

PromptScript 也作为了开发 Unit Mesh 相关应用的 prompt 测试工具，同时也是评估工具 Unit Eval 的评估支持引擎。



端侧开发框架

EdgeInfer 用于在 Android、iOS 或其它资源受限设备上运行小型模型（包括 embedding 和 Onnx 模型），以实现高效的边缘智能、进行实时决策。

模型微调与数据工程

模型-工具-评估一体化

AI 工具的高质量门槛：达成模型、工具、评估一体化。在没有工具的情况下，对模型评估缺乏场景，进而导致测试用例集的不完整。除此，工具在构建辅助功能时，会依赖于特定的上下文内容，它也是作为评估体系的一部分。

右侧是 AutoDev 与 Unit Eval 生成单元测试的提示词，除了待测试的类，它还包含了测试规范、依赖上下文、相关代码等信息。此时，生成的测试代码准确性远高于只提供代码，接受率更高。

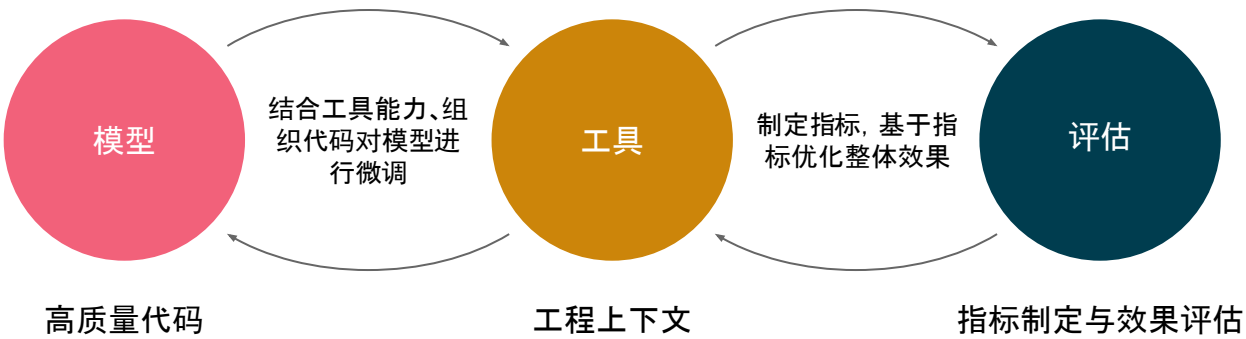
```
Write unit test for following code.

When testing controller, you MUST use
MockMvc and test API only.

You are working on a project that uses
Spring MVC,Spring WebFlux,JDBC to build
RESTful APIs.

// class name: BookMeetingRoomRequest
// class fields: meetingRoomId
// class methods:
// super classes: [Object]

```java
@PostMapping("/{meetingRoomId}/book")
public ResponseEntity<BookMeetingRoomResponse>
bookMeetingRoom(@PathVariable String meetingRoomId,
@RequestBody BookMeetingRoomRequest request) {
   // 业务逻辑
   BookMeetingRoomResponse response = new
BookMeetingRoomResponse();
   // 设置 response 的属性
   return new ResponseEntity<>(response,
HttpStatus.CREATED);
}
```
```



对于参数较大的语言模型，可以理解提示词，而较小的模型则需要进行模型微调才能达到这样的目标。这时，就需要构建配套微调数据与评估工具流水线，以持续演进整体方案。

Unit Eval 是我们针对于一体化设计的开源解决方案，根据不同场景生成不同的微调指令方案。可以**从已有代码库生成新的数据集**，以用于评估微调的结果。

要达成**模型-工具-评估**一体化，比较简单的方式是：**统一的模板化提示词**，并构建质量流水线进行质量控制。

```
Write unit test for following code.

${context.framework}

${context.related_code}

```${context.language}
${context.selection}
```
```

CHAPTER 4: 2024 展望

总结与 2024 展望

结合我们在探索全流程辅助研发，以及落地生成式 AI 的研发工具、平台经验，我们将 AI 辅助研发划分三个阶段：

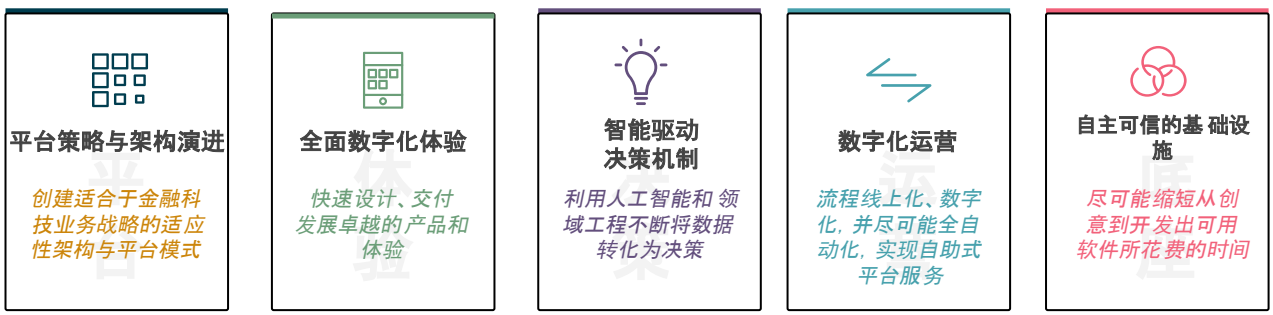
- LLM as Co-pilot。辅助单角色完成任务，影响个体工作。
- LLM as Co-Integrator。解决不同的角色沟通提效，影响角色互动。
- LLM as Co-Facilitator。辅助计划、预测和协调工作，影响组织决策。

即从个体到团队，再到组织，探索更多的可能性。

赋能各角色，提升质量与效率

在 2023 年，构建 Copilot 型的工具，是整个行业达成可落地的共识。而由于模型上的限制以及自身 AI 工程化能力的缺乏，不能更系统地考虑问题，使得 AI 应用落地的效果并不是很理想。

而随着行业内越来越多的落地经验分享，围绕于开发者体验、生成式 AI 工程化将会使得工具与模型的体验更好，进而提升软件质量与开发效率。然而，如何度量效率的提升依旧是一个待解决难题。



构建跨角色 AI，释放生产力

在组织建立了规范化的角色环节产出，如需求文档、测试用例等，借助 AI 来辅助跨角色、团队之间的沟通就变得愈发重要。

因此 AI + BizDevOps 打通整体的工具链会变成下一阶段的重点探索性目标，只是这并非易事，涉及到了人员能力、角色职责重塑、流程改造与优化等一系列的环节。

智能驱动的数字化架构

生成式 AI 的变化，最终要体现对业务的价值上，诸如于更好的演进系统架构、提供更好的数字化体验等等。而位于其底层的是则是实现智能驱动，我们则需要关注于：

- **快速数据与 AI 试验平台**。
- **持续智能赋能。**
- **自服务的 AI 分析能力**。

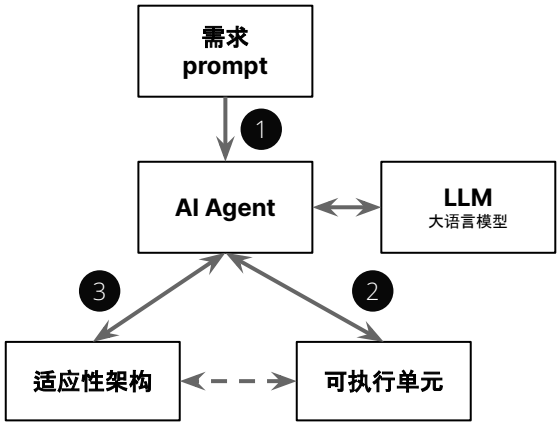
以更好地辅助人类进行决策。

Unit Mesh 目标和愿景

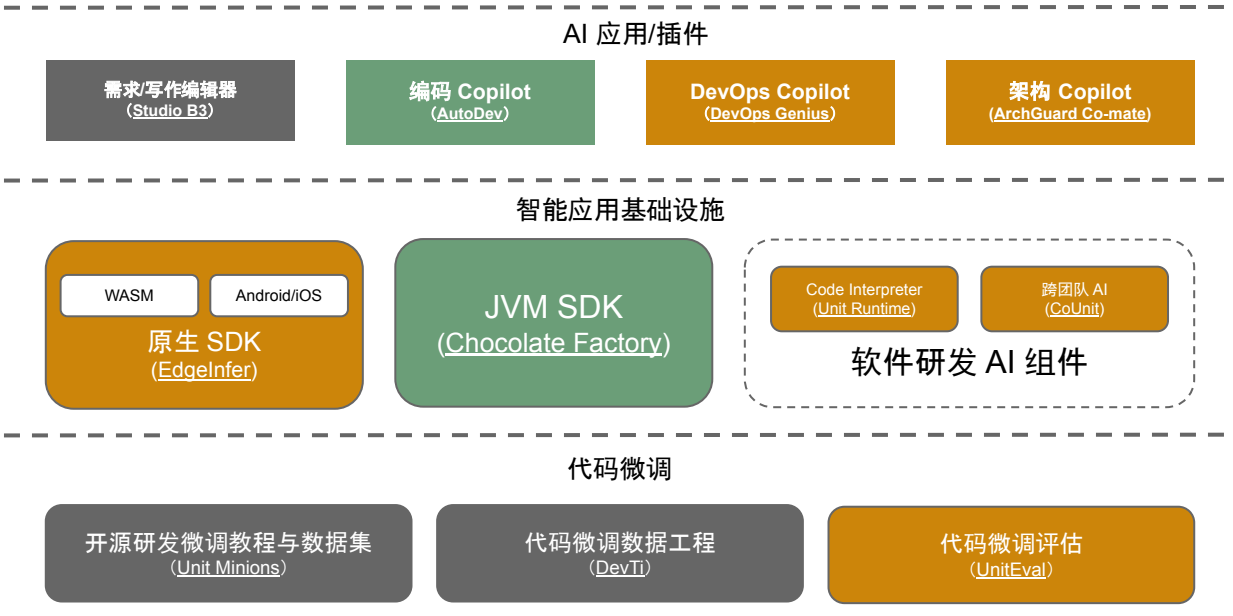
Unit Mesh 架构愿景

基于生成式 AI 的能力，构建 Unit Mesh 构架，并此探索未来的软件架构。在 Unit Mesh 架构下，提示词(prompt)即可执行单元：

1. 用户只需要说一句需求，就会由 AI Agent 将代码上下文与大模型通信，将需求转换为代码单元(Unit)
2. 将代码单元交由代码编译器编译，转变为可执行的单元，如 Web API、前端组件等。
3. AI Agent 来决定此个单元应该如何部署，需要与哪些组件相结合，向最终用户提供服务。



而受限于现有模型的能力限制，并不能很好地实现如此的架构方式，Unit Mesh 优先探索更好适配现有的软件架构。对于组织而言，市面上虽然有一系列的大语言模型、代码模型，但是缺少相关的工具链和基础设施。Unit Mesh 便构建了如下的 AI 应用/插件、基础设施、代码微调三层相关的开源应用。



(注：绿色表示可采用，黄色表示已完成小范 围试验，灰色表示已完成概念 验证(PoC)。

UNIT Mesh

An open-source AI 2.0 + SDLC solution initiated by Thoughtworker

https://github.com/unit-mesh

扫描二维码，关注公众号
获取**最新 AI + 软件研发**洞见