从功能清单到可演进系统：AI 求职工作台产品与架构实习报告

Jiarui Guo

Abstract

本报告回顾本人在 AI 求职工作台（AI Job Workspace）中承担产品设计、技术选型、领域建模与架构协调工作的过程。项目面向准备求职的大学生，同时提供 Web 与 Electron 桌面端，将简历、个人知识资料和模拟面试连接为一条可以反复练习、保留证据并持续改进的用户路径。实习期间，我推动团队以领域驱动设计（Domain-Driven Design, DDD）澄清业务边界，以模块化单体（modular monolith）承载尚在生长的模型，以版本化 API 契约协调前后端，并将数据库迁移、可观测性（observability）、容器化和持续集成（continuous integration, CI）纳入同一套交付体系。

在完成这些工作的过程中，我逐渐认识到，架构并不是在开发之前一次性给出的结构答案，而是管理系统未来变化的一种长期工作。AI Agent 提高了代码生成的速度，却没有替团队承担判断和责任；越是可以快速生成的系统，越需要清楚的领域语言、显式的约束和能够反驳错误的运行证据。报告据此总结本次实习的工程成果，并讨论 AI 参与研发后，技术选择、团队协作和架构职责正在发生的变化。

Keywords

产品架构，领域驱动设计，模块化单体，API 契约，Agent 治理，AI-native 团队

Table 1: 面向产品目标的技术选型与取舍

层次	选择	面向产品与治理的理由
跨端交付	TypeScript、React、Vite、Electron、pnpm workspace	同一产品能力服务 Web 与桌面端；宿主和产品应用分离，减少平台特例向业务层渗透。
服务端	Python 3.14、FastAPI、Pydantic、SQLAlchemy Async	以类型、Schema 和显式组合根连接 HTTP、领域用例、异步 I/O 与模型/文件处理。
数据与检索	PostgreSQL 17、pgvector、Alembic	将事务、租户边界、版本化迁移与向量检索置于可运维的持久化底座。
API 标准	版本化 JSONC Schema、样例、契约校验、OAuth/OIDC	让跨端协作以可校验的发布物为准，而非靠口头约定或静默回退。
交付与验证	GitHub Actions、Playwright、Vitest、pytest、Ruff、mypy、Docker Compose	把格式、依赖边界、浏览器旅程、迁移和镜像安全纳入同一质量门禁。

1 实习内容

1.1 技术选型：先判断系统将怎样变化

技术调研没有从框架的功能数量或基准测试开始，而是从项目未来最可能发生的变化开始。产品需要同时进入浏览器和桌面环境，交互会频繁调整；简历、资料和面试都涉及长流程、文件与异步处理，状态模型会持续增长；AI 能力及供应商仍在快速变化，不宜侵入稳定的业务规则；团队规模有限，也没有条件为每一种技术边界长期支付独立运维成本。这些约束比“哪项技术更新”更能决定一项选择是否合适。

前端最终采用 TypeScript、React、Vite 与 Electron，并以 pnpm workspace 组织。选择 Electron 并不是忽略其制品体积和安全面的代价，而是因为项目首先需要复用一套成熟的 Web 交互，同时保留桌面端对本地能力与系统浏览器认证的承载空间。为了不让这个选择反过来绑架产品，Electron 被限制为一个薄宿主：页面不认识 Node.js，领域代码不认识 React，平台桥接也不承载业务判断。这样一来，跨端复用发生在产品能力层，而不是把桌面端特例散落到每个页面。

服务端采用 Python、FastAPI、Pydantic 与异步 SQLAlchemy。Python 在模型调用、文档解析、数据处理和 Web 服务之间拥有较低的整合成本，适合当前项目；但生态便利也伴随着运行期反射、隐式类型转换和依赖装配难以追踪的问题。为此，我没有把框架能力继续扩张到领域内部，而是让 HTTP、应用用例和基础设施通过显式组合根连接。框架负责接住请求，领域模型负责判断业务，二者的边界不因“少写几行代码”而被抹平。

数据层选择 PostgreSQL 与 pgvector，也遵循同样的克制。当前阶段的核心问题是保证用户、Workspace、简历版本、知识来源和异步任务之间的一致关系，而不是分别运营事务数据库、向量数据库和消息系统。让事务数据与向量索引先共处于一个可迁移、可备份、可授权的底座，牺牲了一部分针对单项负载的极致优化，却显著减少了数据同步和故障恢复中的分叉。只有当真实规模证明这种集中已经成为瓶颈时，拆分才有事实依据。

1.2 架构决策：让领域模型成为共同语言

1.2.1 **模型不是名词表，而是对业务事实的取舍.** 技术栈确定以后，最费时间的工作并不是画出分层结构，而是决定系统究竟承认哪些业务事实。以简历为例，如果把它理解为一份可以覆盖保存的 JSON，那么模型生成一次修改似乎只需要一次写操作；可一旦把用户审阅、历史比较、并发编辑和 PDF 渲染放在一起，这种模型便立即失效。我们最终将 Resume 视为带有不可变 revision 的聚合：修改必须说明自己基于哪个版本，Agent 只能产生 Proposal，是否进入正式简历仍由用户确认。这里的版本不是数据库附属字段，而是产品对“谁有权改变用户叙事”的回答。

同样的推演也改变了其他概念。Workspace 不是出现在 URL 里的分组名称，而是资源归属与重新授权的边界；Knowledge 不是一堆可供模型任意读取的文本，而是有来源、有版本、有访问策略的证据；Interview 也不是消息数组，而是一段具有开始、进行、结束和报告状态的练习过程。Agent、Job、Artifact 与 Audit 则分别承接模型执行、长任务、生成物和责任记录，避免把所有不确定性塞进一个无所不包的“AI 服务”。

这些讨论逐渐形成了项目的界限上下文（bounded context）。领域驱动设计在这里的作用，不是要求代码拥有一套固定的实体和值对象目录，而是允许不同业务语境各自保持精确，并明确它们在何处交换信息 [4]。当同一个名词进入产品说明、接

Table 2: 核心界限上下文及其架构责任

上下文	主要模型与职责	关键不变量或协作边界
Identity / Workspace	用户、登录、OAuth、成员资格、角色与资源归属	每次请求依据真实 principal 与 Workspace 重新授权；路径身份不由客户端 header 决定。
Resume	简历聚合、不可变 revision、操作批次、导入/渲染 Job	修改由稳定 operation ID 和版本控制保护；模型修改只产生待审阅 Proposal。
Knowledge / Connection	文件、来源版本、上传、索引、检索与外部连接	不可信文档先隔离解析；检索结果是证据而不是可覆盖系统指令的命令。
Agent / Interview	对话、运行、引用、面试 Session、报告与 rubric	模型输出受结构化 Schema 约束；证据、授权和最终状态由服务端物化。
Platform	Job、Artifact、Event、Audit、SSE 与幂等 receipt	异步副作用采用事务外盒（transactional outbox）与可恢复租约；最终资源读取是权威状态。
Observability / Operations	信号信封、Dashboard、迁移、维护命令与部署边界	业务应用、迁移器和只读运维面使用独立权限与独立数据访问路径。

口、数据库与测试时，团队应当能够追问它是否仍然指向同一项业务事实。

在部署形态上，我选择先建设模块化单体，而不是按这些上下文立即拆分微服务。边界刚刚被认识时，最需要的是让它们能够被修正，而不是让每次修正都变成跨仓库、跨网络和跨数据所有权的迁移。Parnas 很早便指出，模块应隐藏可能变化的设计决定，而不是简单复刻处理步骤 [9]。据此，项目中的模块边界优先围绕身份规则、简历版本、知识来源和任务生命周期等变化原因建立；进程暂时共享，并不意味着模型混为一体。

图 1 是这一思路的整体投影。它没有按“前端、后端、运维”重复组织结构，而是从用户工作流向下追踪到契约、领域、数据和运行视图。复杂性留在拥有业务语义、能够独立验证的端点中，基础设施只提供尽可能稳定的通道。这也保留了日后服务化的可能：当某个上下文真正拥有独立的扩缩容、可用性目标、数据生命周期和发布节奏时，它已经具备可以拆出的模型边界；在此之前，网络不会替我们创造边界，只会把尚未解决的边界问题变成分布式故障。

1.2.2 API V2: **在团队之间建立稳定的接缝**. 领域模型一旦进入前后端协作，就需要一条不会随某一侧实现细节漂移的接缝。项目早期最危险的情形并不是接口缺失，而是两端都能运行，却分别相信不同的状态语义：前端把请求成功当作任务完成，后端却只表达了任务已经受理；客户端提交了基于旧版本的修改，服务端若沉默覆盖，用户便会在“成功”中丢失内容。为解决这类问题，我组织发布了 API V2，将可阅读的契约说明、

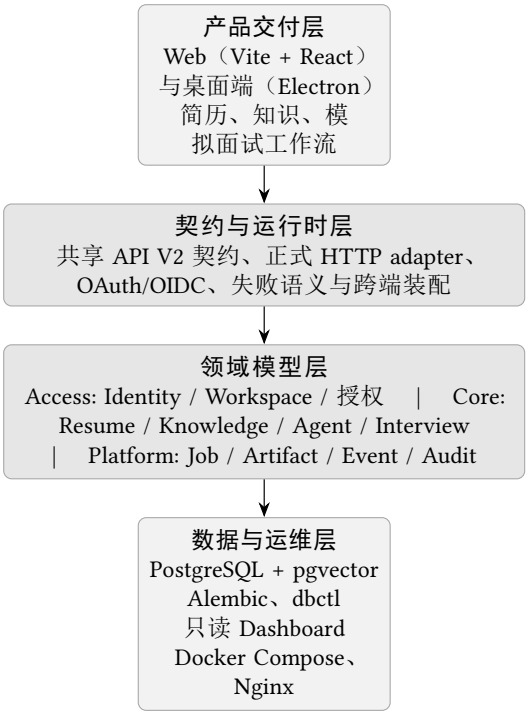

Figure 1: 从产品工作流到运维真相的交付架构。业务复杂性停留在具有领域语义的端点，基础设施提供可靠但不过度智能的通道。

机器可校验的 JSONC Schema、端到端样例和版本差异放入共享发布物，前后端以固定 revision 消费。契约缺失时构建直接失败，不再用旧接口或内存 Mock 填补一段看似顺畅、实际失真的流程。

V2 的重点不在于增加接口数量，而在于统一那些决定系统能否长期演进的语义。版本修改以 `ETag`/`If-Match` 显式暴露并发冲突，具有副作用的请求以 `Idempotency-Key` 承受重试，异步任务以 Job 和最终资源区分“已经接收”与“已经完成”，问题响应和请求标识使失败能够跨端追踪。OAuth/OIDC 与 Authorization Code + PKCE 则把 Web、桌面公开客户端和资源服务端的信任关系说清楚 [6]。这些机制并不显眼，却共同守住一条底线：系统不能用表面成功掩盖未知状态。

Agent 的接入进一步验证了契约的价值。模型返回的内容先经过闭合 Schema 解析；当它建议修改简历时，服务端生成的是附带证据和基准版本的 Proposal，而不是直接改写 Resume。模型可以扩大候选方案的空间，却不能越过授权、版本和用户确认。到这里，产品原则、领域模型和 API 不再是三份平行文档，而成为同一项责任边界的不同表达。

1.3 从“能够运行”走向“能够长期运行”

当主流程可以联调后，项目进入了另一个容易被忽略的阶段：开发环境中的成功，是否能够穿过迁移、部署、故障和恢复。基础设施工作正是在这里展开。它并非给既有产品补上一层运维包装，而是在回答系统生命周期中的另一组产品问题：服务重启后任务是否还在，迁移失败后数据处于什么状态，操作者看到的绿色究竟代表健康还是没有采集到任何信息。

数据库治理首先被从应用启动中分离出来。`dbctl` 将初始化、迁移、遥测清理、交互式 shell 和容器启动表达为明确的运维用例；后端不会在启动时暗中改库，应用凭证也不能获得迁移权限。owner、migrator、application 与 dashboard 四类角色分别拥有不同的数据能力，Workspace 隔离则继续由行级安全（Row-Level Security, RLS）约束 [10]。这样设计的意义不只在最小权限，而在于让一次高风险状态变化拥有清楚的开始、责任人和失败位置。

随后建立的遥测与 Dashboard，使系统第一次能够从运行结果反过来审视设计。HTTP、WebSocket、业务 Job 和浏览器诊断被投影到统一的 metric、log 与 span 信号中；观测队列有界，写入失败不会拖垮业务请求；只读 Dashboard 则明确区分“指标正常”和“根本没有数据”。这一点看似朴素，却改变了团队讨论问题的方式：一个架构判断不再因为文档写得完整便被认为成立，它必须在延迟、错误、任务终态和用户路径中留下可以被检查的结果。

最后完成的容器化部署延续了相同的思路。镜像采用多阶段构建、非 root 用户和只读根文件系统，Nginx 只公开身份与 API V2 所需的入口；不可信文档解析和 XeLaTeX 渲染只有在进程隔离能力成立时才可用于生产。这里没有把“安全”写成部署说明中的愿望，而是让环境在缺少必要能力时拒绝启动。系统能够失败并不稀奇，困难的是让它在错误的前提下不要假装继续工作。

1.4 让架构进入团队的日常工作

这些设计若只停留在我个人的理解中，便不会成为项目架构。实习期间，我与前端、后端和运维同学围绕 Workspace、Resume、Knowledge、Agent 与 Interview 反复校对状态、权限和失败语义。工作也不再简单按页面、接口和服务器横向切开：前端同学可以沿用户旅程追踪到正式运行时，后端同学可以从领域命令追踪到持久化和任务终态，运维同学拥有迁移、部署与只读运行视图。专业分工仍然存在，但每项工作都尽量保留一条通向用户结果的纵向路径。

GitHub 的拉取请求（pull request, PR）由此成为设计继续发生的地方。前端的契约校验、类型检查、真实 Chromium 用户旅程、构建与 Electron smoke，后端的静态检查、领域测试、PostgreSQL 迁移验证和非 root 镜像验证，共同把讨论从“这段代码像不像某种架构”引向“它是否仍遵守我们共同承认的业务事实”。架构决策记录（Architecture Decision Record, ADR）和交接文档也与代码同仓演进；它们记录的不是最终答案，而是当时为什么作出选择、什么变化会使选择失效。对团队而言，知识共享的真正尺度不是文档页数，而是下一位成员能否独立找到事实、复现判断并安全地修改它。

2 实习总结

2.1 Agent 改变的是工程成本，而不是责任归属

在本次实习中，Agent 确实承担了大量搜索、生成、比较和修改工作。它最有价值的时刻，往往不是一次写对全部代码，而是在边界已经明确、验证方法已经准备好之后，迅速探索多种实现并消化机械性工作。反过来，当问题描述含混、仓库里存在两份事实源，或测试只覆盖顺利路径时，它也会以同样的速度放大误解。生成速度和工程进度并不是同一个量。

这段经历让我把 AI 辅助研发理解为监督式工程（supervised engineering）。人需要决定问题值得怎样建模，哪些状态不可接受，什么证据足以合并；Agent 负责扩大搜索宽度，测试、评审和遥测则不断淘汰错误候选。NIST 对生成式 AI 风险治理所强调的责任划分与人类监督（human oversight）[7]，在软件工程中并不是抽象合规要求，而是一条非常具体的事实：Agent 可以生成提交，却不能成为事故、数据丢失或错误授权的责任主体。

这也重新排列了工具的成本。过去，“少写样板代码”足以成为重框架、反射式依赖注入（dependency injection, DI）和运行期拼装的重要理由；当样板代码可以廉价生成后，这部分收益正在缩小，而隐式行为造成的定位和验证成本依然存在。相反，Schema、迁移文件、类型化联合、显式组合根和架构测试看似增加了文字，却给人、Agent 与 CI 提供了同一份可读取的约束。真正有价值的抽象不以隐藏了多少代码衡量，而以它是否留下稳定事实源、可定位失败点和可重复验证路径衡量。

2.2 Agent 工程：客户驱动，契约优先

在 Agent 参与开发以后，我采用的设计思想可以概括为客户驱动（customer-driven）与契约优先（contract-first）。客户驱动决定系统为什么变化：从真实用户任务、失败体验和业务结果出发，而不是让已有代码结构或模型能力反过来定义产品。契约优先决定变化如何进入系统：先把领域语言、状态转换、权限、失败语义和验收证据写成机器可以检查的边界，再让 Agent 在边界内部搜索实现。前者防止团队高效地解决错误问题，后者防止 Agent 用表面正确的代码掩盖语义偏离。

这种方法同时打破了传统代码评审（code review）的成本模型。当代码主要由人串行编写时，再投入一名工程师逐行检查，尚可被视为与开发成本同量级的质量交换；当多个 Agent 可以并行生成和修改代码时，人的注意力却不能并行扩张。若每次生成仍必须排队等待另一名工程师阅读 diff，团队吞吐量最终只取决于评审者最稀缺的几个小时。Agent 增加得越多，评审队列反而越长，代码生成的速度优势被一道人工串行门禁全部抵消。

更关键的是，这道门禁原本就没有兑现团队对它的主要期待。微软对大规模工程实践的总结指出，代码评审往往找不到足以阻止提交的功能缺陷；其数据中只有约 15% 的评审评论涉及潜在缺陷，而人工评审通常又是代码集成中耗时最长的环节 [3]. 另一项覆盖微软多个团队的实证研究也发现，缺陷发现虽然是评审最主要的动机，实际产出却更多是知识转移、团队感知和替代方案，深层、细微或宏观问题尤其少被发现 [1]. 把 code review 当作质量保证的核心，只是用昂贵的人类注意力购买一种并不可靠的安全感。

过去仍能支撑 code review 的理由，是它同时承担团队知识共享与一致性维护：工程师通过阅读他人的变更知道系统正在发生什么，通过评论传播约定和设计理由。但这项职能也正在被 Agent 取代。Agent 可以持续读取代码、契约、ADR、测试和变更历史，在成员真正需要时解释一个边界的由来、检索相似决策并指出受影响路径。知识不再需要借一次偶然的 PR 广播给所有人，而可以在具体任务中按需重建。团队一致性也不应寄托于评论区中的说服，而应沉淀为共享领域模型、版本化契约和自动执行的工程门禁；Agent 负责让这些知识随时可达，人负责决定哪些知识应当成为约束。

因此，我认为强制、全量的人工 code review 应当逐步从团队工作流中移除。它应从“每次提交必须经过的默认仪式”退回为少量、按风险触发的专项审查，只保留在公开契约、数据迁移、安全边界和难以回滚的架构决策上。评审的对象也由每一行实现代码上移为领域模型、契约和运行证据。团队不再靠“另一个人看过”证明质量，而是靠系统能够持续给出可重复的反证。

这不是撤掉质量机制，而是把质量工作的重心从静态审查迁向运行时剖析（runtime profiling）与打桩（instrumentation）。一份 diff 最多说明代码可能怎样运行，真实负载才会暴露哪条路径真正高频、延迟消耗在哪里、队列何时饱和、内存为何增长，以及错误究竟集中在哪类用户和状态上。优化也因此不应从经验猜测或代码洁癖出发，而应由剖析结果引导：先在运行中找到成本，再沿 span、metric 和 log 追到领域动作与资源边界，最后修改实现并重新测量。静态检查适合守住类型、依赖方向和已知不变量，运行时证据则负责揭露那些只有系统真正活起来才会出现的问题。

这也解释了我为何在项目中部署遥测。它不是产品做完后附加的一套 Dashboard，而是 Agent 工程的反馈器官：HTTP、WebSocket、后台 Job 与浏览器行为都经过打桩，被还原成可以查询和关联的运行事实。测试是在发布前预测系统会怎样，遥测是在发布后观察系统实际上怎样；二者之间的差异，才是下一轮修复、优化乃至重画领域边界的起点。用户在工单、测试群或 X 上报告的“感觉变慢了”“任务卡住了”，也只有落到具体 trace、任务终态和资源曲线上，才能从情绪信号变成工程判断。

OpenAI 最近公开的 agent-first 实验提供了一个鲜明近例：一个由 Codex 写下全部代码、测试、CI 与可观测性配置的产品，在五个月内合并约 1,500 个 PR，平均达到每位工程师每天 3.5 个 PR；团队采用极少的阻塞式合并门禁，因为在 Agent 吞吐量远高于人类注意力的条件下，修正已经变得便宜，等待反而昂贵。更重要的是，他们把日志、指标和链路直接暴露给 Agent，使其能够复现问题、修改代码、重启服务、重放用户路径，再依据运行信号继续修正 [8]. 这不是对“vibe coding”的浪漫化：文章坦率承认产品会发布、会损坏、也会被修好，甚至需要每日清理 Agent 复制出来的坏模式。真正值得借鉴的不是容忍粗糙，而是让发现、定位和修正错误的速度与生成代码的速度一起增长。

这种节奏也真实地延伸到了公开用户反馈中。2026 年 7 月，Codex 负责人 Thibault Sottiaux 在 X 上复盘一次产品发布：团队在过去 24 小时持续阅读反馈、观察使用模式并与用户交流，随后承认高算力选项、桌面端改版、插件和多 Agent 工作流都存在首发问题；第一批修复当天落地，用户在同一天便会看到数次更新，更大的调整则安排在下一周 [12]. 这的确带有 vibe coding 的外观—先让 Agent 把产品高速推入现实，再让现实迅速反驳它—但支撑这种速度的不是“凭感觉上线”，而是两条并行的反馈线：X 上的抱怨告诉团队哪里值得看，使用数据与遥测解释问题实际发生在哪里。前者提供用户经历，后者提供运行机制，修复则把两者重新缝合。

由此，日常交付链路应当改写为：从客户场景形成契约，Agent 生成实现与测试，CI 机械验证边界与不变量，部署后的遥测和剖析暴露真实偏差，修复与重构再反过来更新契约。快速失败（fail fast）不是在生产中莽撞试错，而是让错误尽早显形、影响范围可控、原因可追、版本可退；快速迭代也不是一天发布多少次，而是一次用户反馈穿过观测、判断、修复和验证所需的时间有多短。

2.3 AI-native 团队：以领域责任重画组织边界

这次实习使我形成了一个明确判断：Agent 改变的不只是个人生产率，还会改变组织成立的前提。过去按产品、前端、后端、测试和运维划分团队，是因为每一种工具链都需要长期训练，一个人难以跨越全部技术环节。Agent 成为能够承担多种执行职能的多面手以后，这种稀缺性正在消退。若组织仍按旧工种层层交接，结果不会是专业分工更加稳固，而是人负责转述信息，Agent 在每道职能墙内重复生成彼此脱节的局部实现。

康威定律（Conway’s Law）意味着，组织的沟通边界最终会变成系统的技术边界 [2]。因此，AI-native 团队不能只在原有部门中增加 Agent 席位，而应当重画责任版图。我的部署设想是两次合并、一次分离：产品与前端合并，围绕用户旅程共同承担需求、交互、页面状态与客户端诊断；后端与运维合并，围绕服务生命周期共同承担领域用例、数据迁移、部署、遥测与事故恢复。前者对用户是否真正完成任务负责，后者对系统是否真实、稳定地完成任务负责。原本需要跨部门传递的许多信息，由此成为同一责任单元内部的连续判断。

与此同时，跨领域业务团队必须与基础设施平台团队分离。业务团队沿界限上下文组建，拥有领域模型、用户结果和从接口到运行信号的完整纵向切片；平台团队则提供契约校验、CI、身份基线、数据库治理、部署与可观测性等共享能力。平台负责修路，却不替业务团队决定去哪里。把业务规则吸入总线、中台或万能工作流引擎，看似减少重复，实则会重新制造一个掌握全部语义、任何变化都必须经过的中央组织。基础设施应当统一机械约束，而不统一业务判断。

这种组织的最小责任单元，不再是只会操作某一层技术的开发人员，而是能够掌握一个完整系统的高级工程师（senior engineer）。他不必亲手写完每一行代码，但必须理解产品目标、领域模型、运行边界和失败后果，能够为 Agent 划定搜索空间、审阅关键取舍，并为最终进入生产的结果负责。

我的进一步判断是，研发与开发应当合并。过去由研发人员提出方案、开发人员照单实现的分工，建立在实现能力稀缺且传递成本尚可接受的前提上；Agent 已经使这项交易倒挂。研发人员应当直接完成从问题发现、领域建模、契约设计到可运行软件的闭环，由 Agent 承担大部分具体开发，研发人员监督其搜索与实现，并根据生产证据继续修正模型。只负责把需求翻译成代码的“开发”不再需要成为独立组织层，做出判断的人也不能把软件结果交给下游岗位负责。

这不是用“全栈”名义要求每个人无限扩张技能，而是把专业能力从固定岗位中释放出来。专业者仍然存在，但以评审、平台能力和高风险决策的形式进入团队，而不是成为每项变更都必须跨越的组织关卡。最终的组织应当比今天更扁平：少一些仅负责转交信息的层级，多一些能够拥有领域结果的工程师；少一些按技术层划出的领地，多一些从用户需求一直追到生产运行的责任链。这与 Team Topologies 所强调的价值流与认知负载相呼应 [11]，但在 Agent 参与后，端到端所有权不再只是理想的团队形态，而正在成为最经济的组织方式。

2.4 做架构，是推演系统在时间中的枯荣

过去谈架构，习惯谈系统由哪些部分组成：分几层，如何通信，数据放在哪里。这些问题当然重要，但它们只描述了系统某一刻的剖面。架构真正面对的是时间。用户需求不会沿着路线图整齐增长，它会突然加入一种能力，也会抛弃曾经重要的流程；有些模块因使用量增长而需要分化，有些功能则在产品转向后成为负担。加功能、砍功能和重构并非开发过程中的偶发插曲，而是软件与现实长期相处的基本方式。

我把这种过程称为系统的“枯荣”。“荣”不只是功能越来越多，而是系统能够吸收新的需求，让新的模型在旧结构中找到位置；“枯”也不等于失败，而是能够承认一项能力已经失去价值，切断依赖、迁移数据并把它完整移除。现实中的软件属于 Lehman 所说的 E-type 系统：只要它仍在解决真实世界的问题，就必须持续随环境改变；如果团队只允许系统增长而不会

修剪，复杂度便会随着每次变化不断沉积 [5]。所谓可维护性、可扩展性，归根结底都是系统承受这种需求变迁的能力。

由此回看本项目，模块化单体的意义并不是宣布永不采用微服务，而是在领域仍然生长时允许边界移动；API V2 的意义也不是冻结接口，而是让新旧客户端可以经历一段有秩序的交替；不可变 revision、显式迁移和 ADR 则为重构与删除保留来路。一次好的架构决策，不只要想象某项功能怎样被加入，还要推演它有一天怎样被替换、拆出或彻底砍掉。只能添加、不能删除的系统并不强大，它只是把所有历史都背在身上。

因此，架构师观察的尺度应当长于一次迭代。既要进入代码，与开发者一起校准命令、状态机、数据迁移和失败路径，也要站到系统演化的上游，判断今天的依赖是否正在剥夺明天的选择。领域模型记录团队此刻如何理解业务，契约和测试使这种理解可以被执行，而真实用户的新需求又不断迫使模型修正。架构不是替系统规定一个永恒形态，而是让它在生长时不失控，在收缩时不崩塌，在重构后仍能继续演化。

2.5 AI-native 的能力：在开放性中建立判断

传统考试偏爱封闭问题：条件已经给定，目标已经给定，评价标准也已经给定，答题者只需在规定时间内找到那条预设路径。算法题尤其如此。它可以训练解题，却常被拿来遮蔽对真实工程能力的无知。现实项目面对的则是开放性（open-endedness）：问题的边界尚未划定，目标本身可能互相冲突，评价标准会随用户反馈而改变，甚至一次行动取得的新证据也会改写原来的问题。它不是一道更难的题，而是根本不存在预设题面的世界。

Agent 擅长的，恰恰是边界已经给定之后的搜索与执行。记忆接口、熟悉框架、迅速写出样板代码，过去需要长期训练，如今可以被模型低成本复制；甚至在一个陌生技术栈中，Agent 也能比人更快完成局部迁移。继续把某项工具的熟练度当作核心竞争力，相当于把个人价值押在一种加速贬值的资产上。程序员真正不可外包的工作，只剩下两件：塑造 Agent 搜索问题的空间，以及用自己的先验判断提高它搜索的 token 效率。前者决定什么被视为问题、哪些答案值得到达，后者决定搜索要在多少错误道路上浪费时间。

这两件事背后是同一种能力：在开放性中不断建立、检验并修正判断。成熟的技术人员不必预先知道答案，甚至不必假装问题已经被完整定义；他能识别当前缺少什么，把含混争论暂时收敛为可验证的假设，再让行动产生的新事实反过来重画问题边界。他会把庞大系统切到第一个可以观测的断面，把一次失败变成下一轮搜索的高信息量先验，也会在目标变化后放弃一条曾经正确的路径。这里需要的不是更多孤立知识，而是经过长期实践酝酿出来、能够跨领域迁移的思想：怎样建模，怎样怀疑，怎样安排证据，怎样在不可逆的决定前保留退路。

因此，AI-native 时代的学习不应继续围绕“熟练掌握某项技术栈”组织。技术仍然要学，但学习的终点不是形成肌肉记忆，而是从技术中提炼可以迁移的判断。真正拉开差距的，也不再是谁能在一条清晰路径上走得更快，而是谁能在问题、目标与路径都尚未封闭时，仍然推进认识，并让系统随新的现实不断改变。

3 结语

本次实习让我看到，产品判断只有进入领域模型，领域模型只有进入契约、代码和运行边界，才会真正成为系统的一部分。我所完成的技术选型、领域建模、API 标准、基础设施与团队协调，并不是几组彼此独立的成果，而是在为同一件事建立条件：当需求继续变化时，团队仍然能够理解系统、改变系统，也能够删除已经不再属于它的部分。

代码会增长，也会被砍掉；边界会形成，也会重新分裂；今天最合适的技术，终有一天会成为需要替换的历史。架构并不能阻止这些变化，也不应试图阻止。它要做的，是让变化发生时不必一次次推倒重来，让系统经历自己的枯荣之后，仍有能力进入下一段生命。

References

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, San Francisco, CA, USA, 712–721. https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

[2] Melvin E. Conway. 1968. How Do Committees Invent? *Datamation* 14, 5 (1968), 28–31. https://www.melconway.com/Home/pdf/committees.pdf

[3] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. IEEE. https://www.microsoft.com/en-us/research/publication/code-reviews-do-not-find-bugs-how-the-current-code-review-best-practice-slows-us-down/

[4] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, Boston, MA.

[5] Meir M. Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. *Proc. IEEE* 68, 9 (Sept. 1980), 1060–1076. doi:10.1109/PROC.1980.11805

[6] Torsten Lodderstedt, John Bradley, Aaron Labunets, and Daniel Fett. 2025. Best Current Practice for OAuth 2.0 Security. RFC 9700. doi:10.17487/RFC9700

[7] National Institute of Standards and Technology. 2024. *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile.* Technical Report NIST AI 600-1. National Institute of Standards and Technology. doi:10.6028/NIST.AI.600-1

[8] OpenAI. 2026. Harness Engineering: Leveraging Codex in an Agent-First World. OpenAI Engineering. https://openai.com/index/harness-engineering/ Accessed: 2026-07-23.

[9] David L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. doi:10.1145/361598.361623

[10] PostgreSQL Global Development Group. 2026. PostgreSQL 17 Documentation: Row Security Policies. PostgreSQL Documentation. https://www.postgresql.org/docs/17/ddl-rowsecurity.html Accessed: 2026-07-23.

[11] Matthew Skelton and Manuel Pais. 2019. *Team Topologies: Organizing Business and Technology Teams for Fast Flow.* IT Revolution Press, Portland, OR.

[12] Thibault Sottiaux. 2026. A Quick Update on ChatGPT Work, Codex, and the Updates We Shared. Post on X. https://x.com/thsottiaux/status/2075641131002700120 Posted: 2026-07-10; accessed: 2026-07-23.