Critical Aspects of Dependency Management in Large-Scale Software Engineering

Chai Feng <chaifeng@odd-e.com>
Odd-e
2025-10

A Real Story

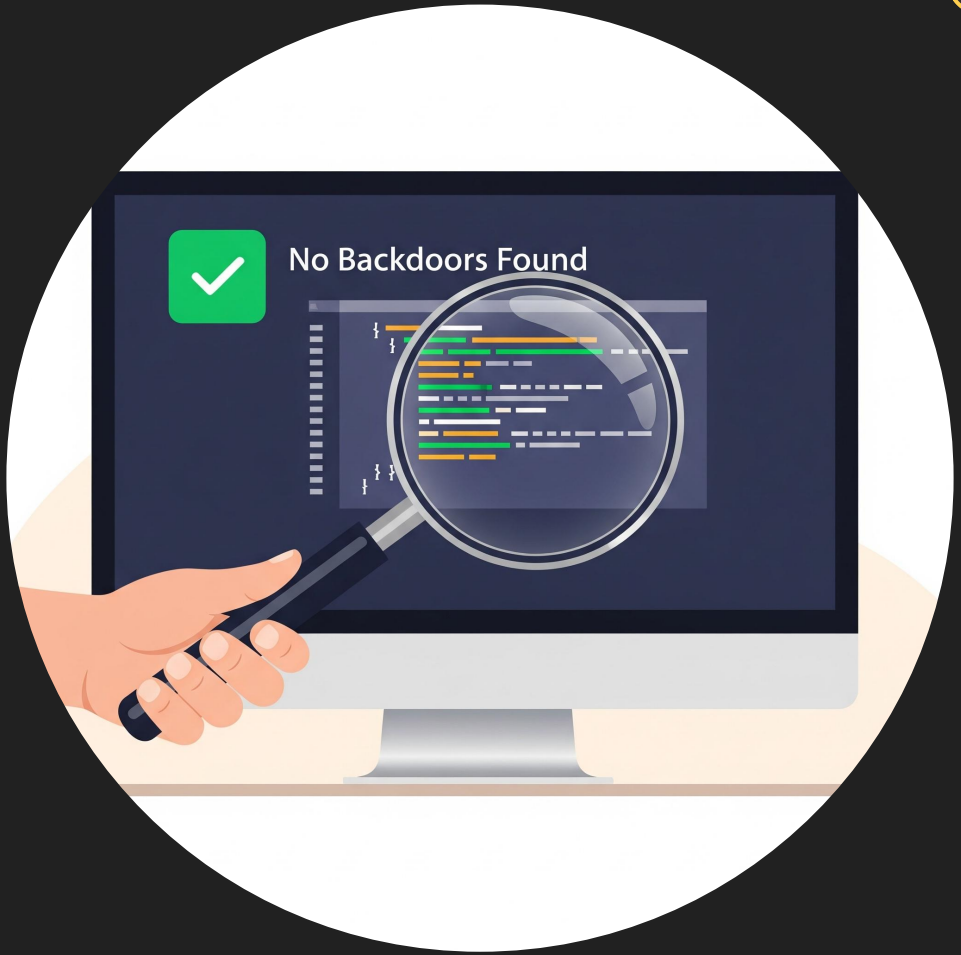
A government questioned if a large telecom company's products had a "backdoor"



Source Code Audit



Source code
is clean,
no backdoors



Source code is safe.

Does it mean the product is safe?





Reproducible Builds

"Is the product really built from the audited source code?"

The Strict Requirements

In a Complete New Environment
Controlled by the Government

&

Bit-for-Bit Identical





Yes, the products were built from this audited source code

Now, it should be TRUSTWORTHY

Source code is safe

Product is really built from this source code



The Ultimate Question



Can you prove that in such a complex build process,

every single step is secure and auditable?

And there was no chance

to secretly introduce any security risks or backdoors?

XZ Backdoor Incident in 2024



The attacker injected the backdoor by changing the build scripts



https://upload.wikimedia.org/wikipedia/en/6/65/OpenSSH_logo.png

The End of the Story, But it’s Not Over



The government still decided not to buy from them in the future

It’s a huge waste for this company, right?

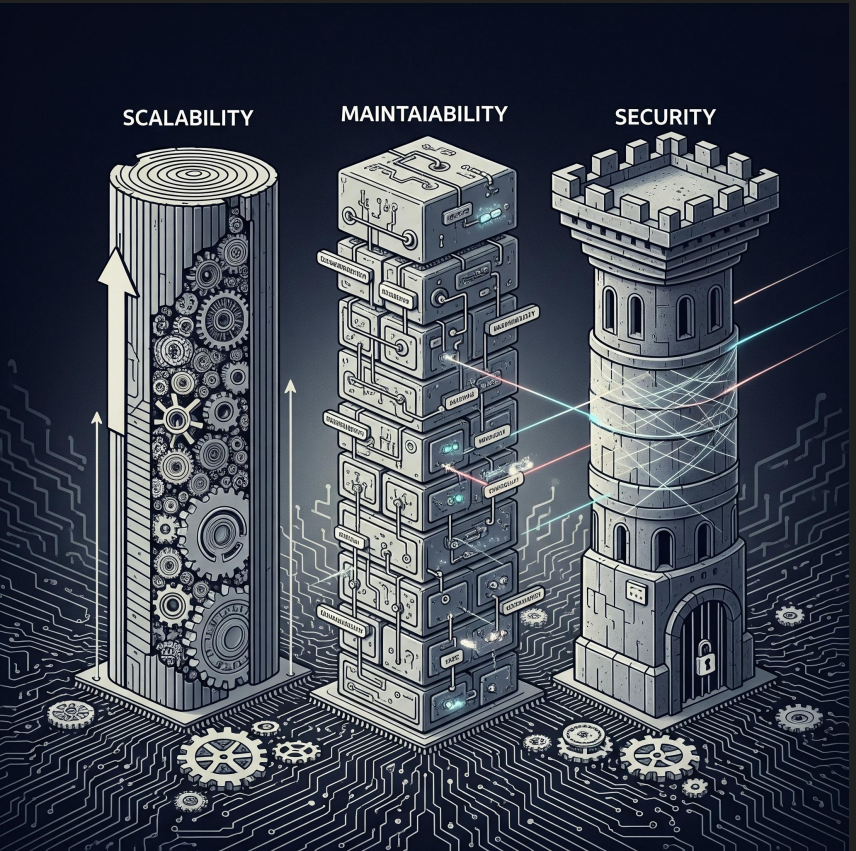
The Lesson from the Story



It was a great opportunity to learn and improve their own engineering capabilities

“Trust” is an engineering capability

Three Core Challenges in Large-Scale Engineering



Is source code trustworthy?

Is product built from this source code?

Is the build process trustworthy?

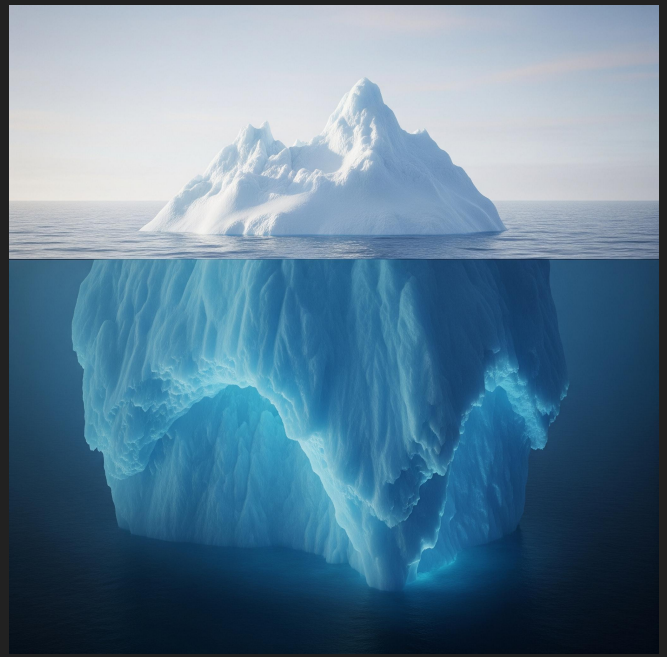
What is a “Dependency”?

When we talk about **dependency management**

- `dependencies` in package.json
- `<dependency>` in pom.xml
- `require` in go.mod
- Git submodules
- ……



Dependency Iceberg



Can cause product failures or introduce critical issues

Also change and get upgraded

General Dependency & Narrow Dependency



Everything that can affect the software production process and the result, **are dependencies**.

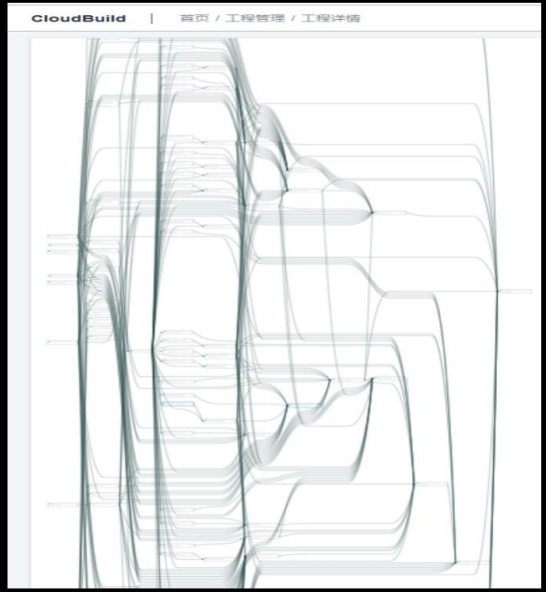
How complex is the General Dependency network in the real world?

Typical telecom hardware product

Components		9000+
Services		300+
Packages/Images		600+
Developers		1500+

Codebase Scale	**C++**	48M
	Go	21M
	Java	12M
	Others	2.6M
	Total	**84M**

Challenges of Just One Platform



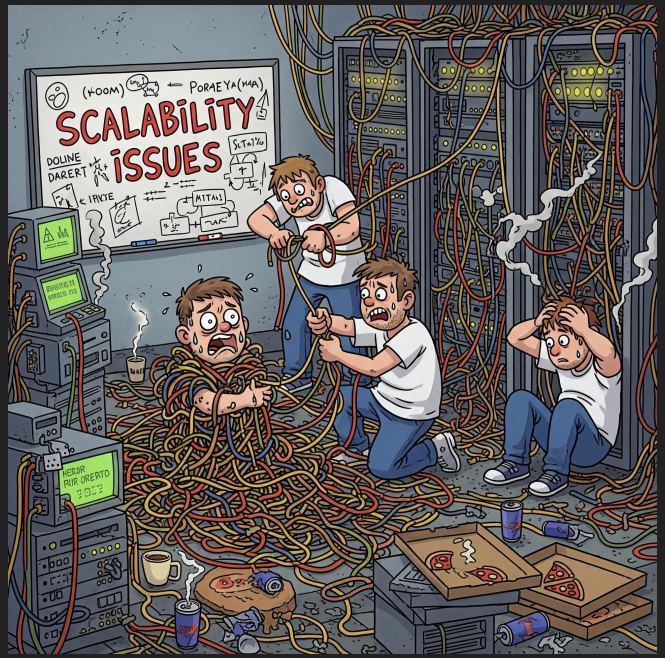
- Only ~15% of the build pipeline
- Involves 260 repos and 300+ pipeline jobs
- Weekly releases
- Each build takes 7-8h
- Release incremental updates

- Frequent updates affect dependencies
- Requiring manual tracing and timely fixes

A large-scale development environment is like a magnifying glass



Core Pain Points of Scale



- Unclear path of change impact
- Integration is late and fragile
- Slow builds

We rely on the “Black-Box Scanning” model of build tools.

But as the scale grows …



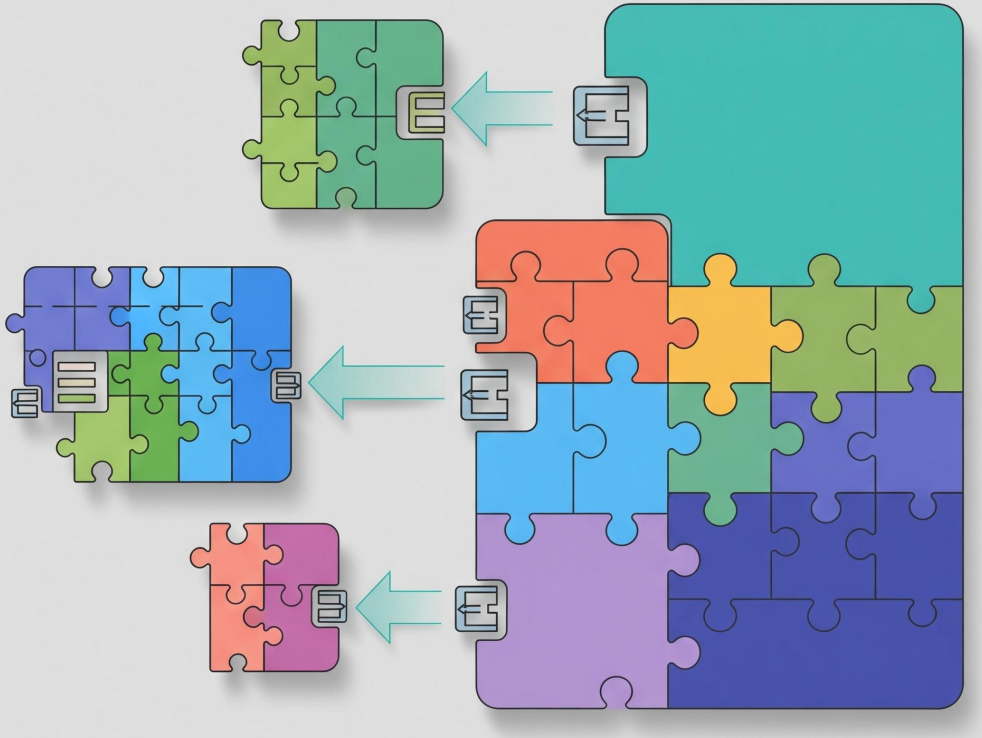
Bottleneck of “Black-Box Scanning”

- More than 2000 developers are collaborating
- Millions of lines of code are being written
- Full scanning takes 6-8 minutes
- The cache for a build: 40+ GB
- Thousands of build nodes work simultaneously
- Extensive downloads cause network storm



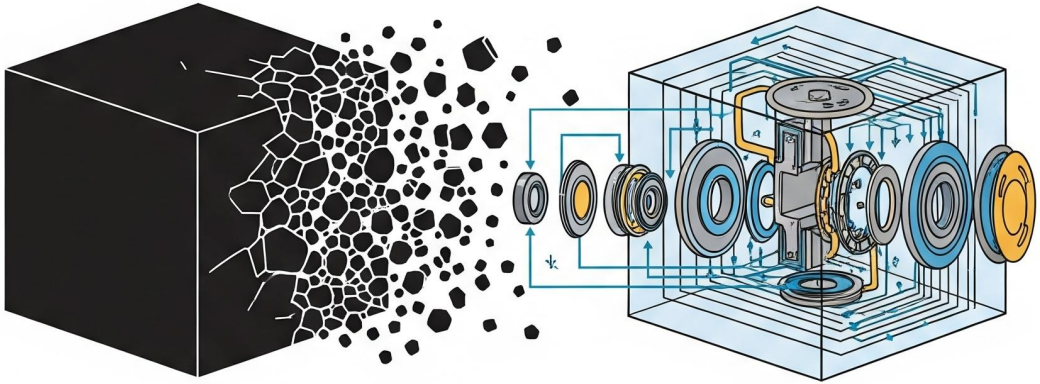
The Revolution

Explicit Engineered Governance



Divide and Conquer

“Partial Integration” V.S. “Divide and Conquer”



From

“Build less”

To

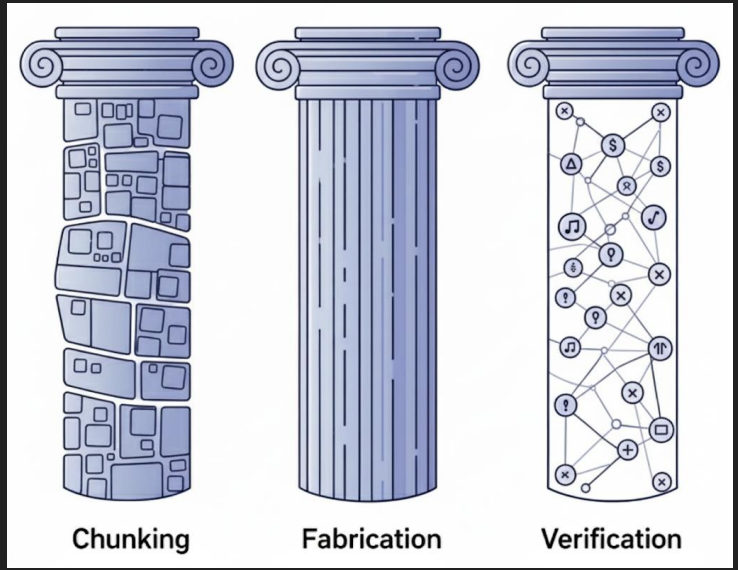
“Build only what is necessary”

Three Core Practices

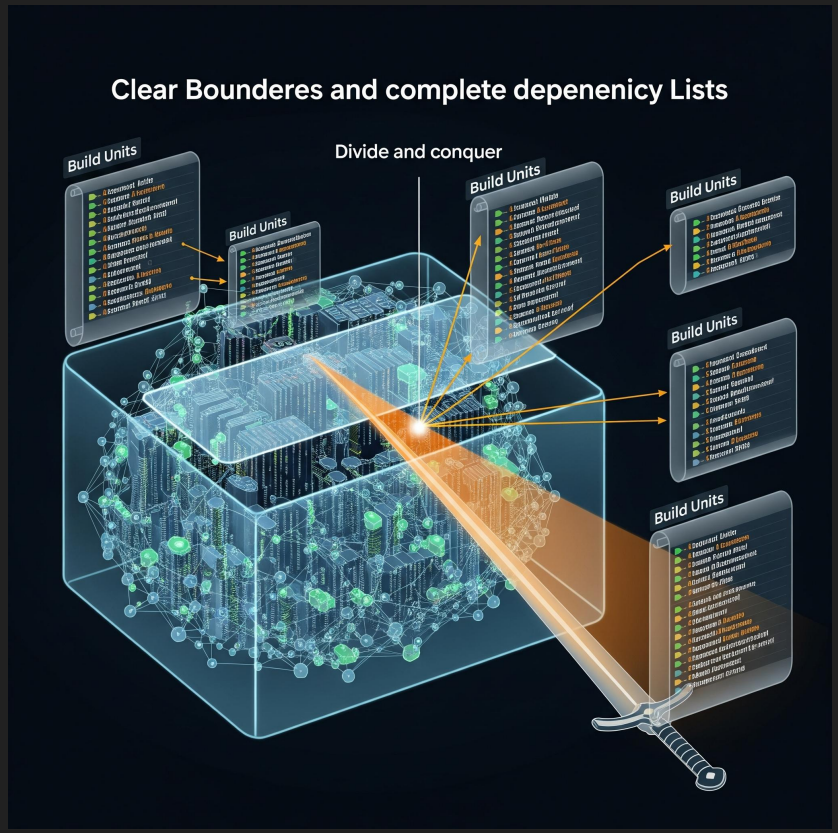
Chunking： break down the system

Fabrication： build the product

Verification： prove the result

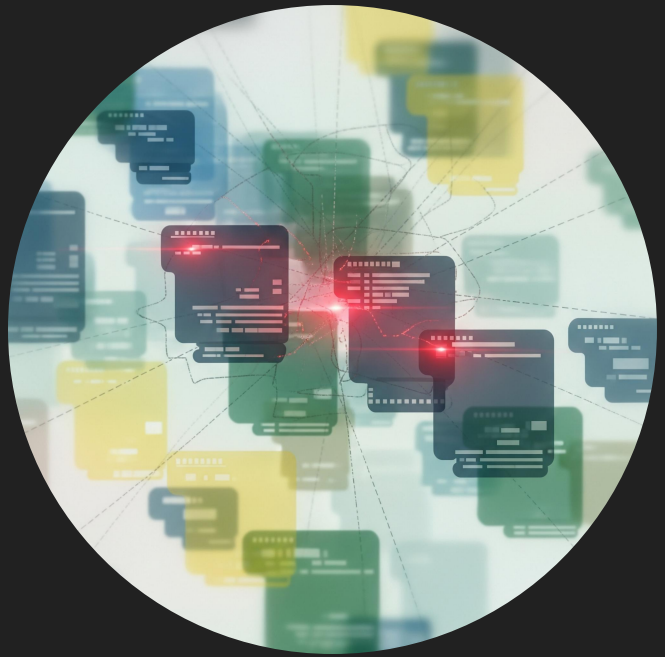


Chunking



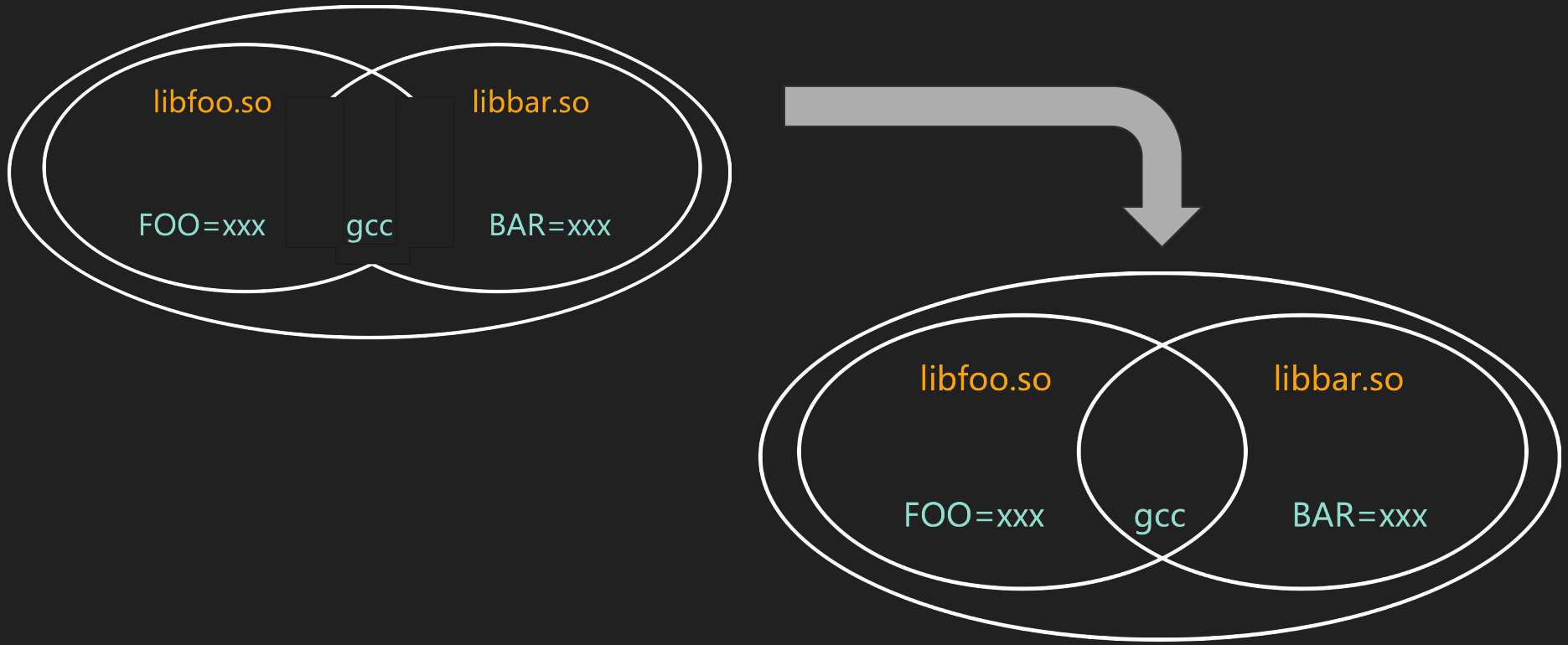
The goal of Chunking is “Divide & Conquer”

Challenge for Chunking



Unclear Boundaries

Challenge for Chunking



Challenge for Chunking

Hidden Dependencies



Challenge for Chunking

```
project:
  name: hidden_deps_app
  sources:
    - src/main.cpp
    - src/utils.cpp
build_environment:
  compiler: gcc-9.3.0
  system_packages:
    - name: libssl-dev-1.0.2
    - name: libcurl4-openssl-dev-7.8.10
```

Challenge for Chunking



Broken Connections

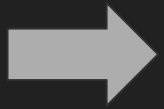
Challenge for Chunking

```
cd encryption-engine-cpp || exit 1

mkdir -p build
cd build
cmake ..
make
cd ../..

cp encryption-engine-cpp/build/libencrypt.so \
    payment-service-java/src/main/resources/

cd payment-service-java || exit 1
mvn clean package
```



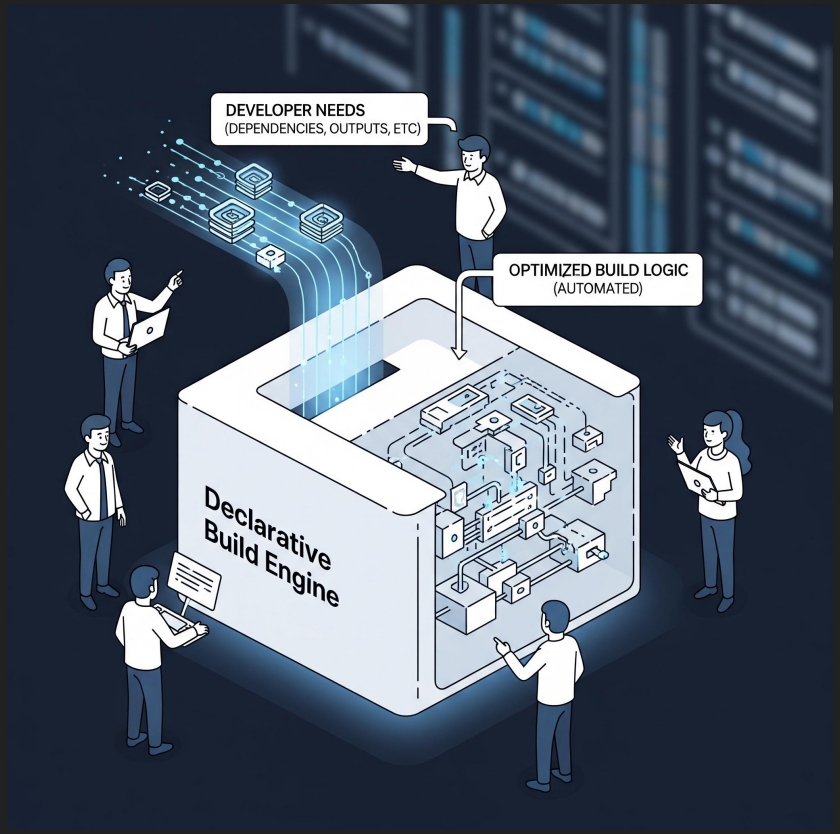
```
project_name: secure-payment-gateway
services:
  encryption_engine:
    type: cpp_library
    sources: ...
    build_environment: ...
  payment_service:
    type: java_app
    sources: ...
    build_environment: ...
    dependencies:
      - type: service
        name: encryption_engine
```

Challenge for Fabrication



For the same set of inputs, always produce bit-for-bit identical output

Challenge for Fabrication



Unreliable Processes

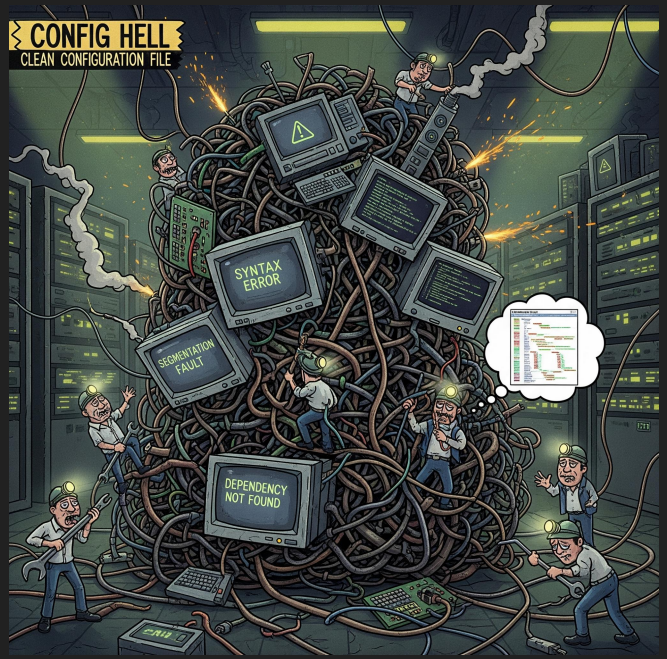
Challenge for Fabrication

```
cf_add_library(foobar::secure::ssl
  ...
)

cf_add_library(foobar::secure::crypto_core
    COMPILE_OPTIONS -DENABLE_FIPS_MODE=1 -O3
    SRCS src/aes.c    src/sha256.c
    PUBLIC_HEADERS include/crypto_core
    DEPENDS foobar::secure::ssl
            openssl::crypto
            sys::threading
            utils::logger
)

cf_add_library(foobar::service::key_manager
    SRCS src/key_manager.c    src/random.c
    PUBLIC_HEADERS include/key_manager.h
    DEPENDS foobar::secure::crypto_core
            sys::threading
            utils::logger
)
```

Challenge for Fabrication



Config Hell

Challenge for Fabrication

```yaml
product_id: "atlas-lx-2025"
model_name: "Atlas Luxury"
features:
  powertrain:
    engine_type: "V6_TURBO"
    transmission: "8_SPEED_AUTO"
    drive_mode: "AWD"
  infotainment:
    screen_size: "12.3_inch"
    navigation:
      enabled: true
      maps_provider: "here"
    premium_sound:
      enabled: true
      brand: "bose"
    connectivity:
      - "apple_carplay"
      - "android_auto"
      - "4g_hotspot"
  driver_assist:
    adaptive_cruise_control: true
    lane_keep_assist: true
    blind_spot_monitor: true
    parking_assist: "360_CAMERA"
  climate_control:
    zones: "dual"
    heated_seats: "front_and_rear"
    air_purification: true
```

```yaml
product_id: "atlas-st-2025"
model_name: "Atlas Standard"
features:
  powertrain:
    engine_type: "I4_NA"
    transmission: "6_SPEED_AUTO"
    drive_mode: "FWD"
  infotainment:
    screen_size: "8_inch"
    navigation:
      enabled: false
      maps_provider: null
    premium_sound:
      enabled: false
      brand: null
    connectivity:
      - "apple_carplay"
      - "android_auto"
  driver_assist:
    adaptive_cruise_control: false
    lane_keep_assist: false
    blind_spot_monitor: true
    parking_assist: "REAR_CAMERA"
  climate_control:
    zones: "single"
    heated_seats: "front_only"
    air_purification: false
```

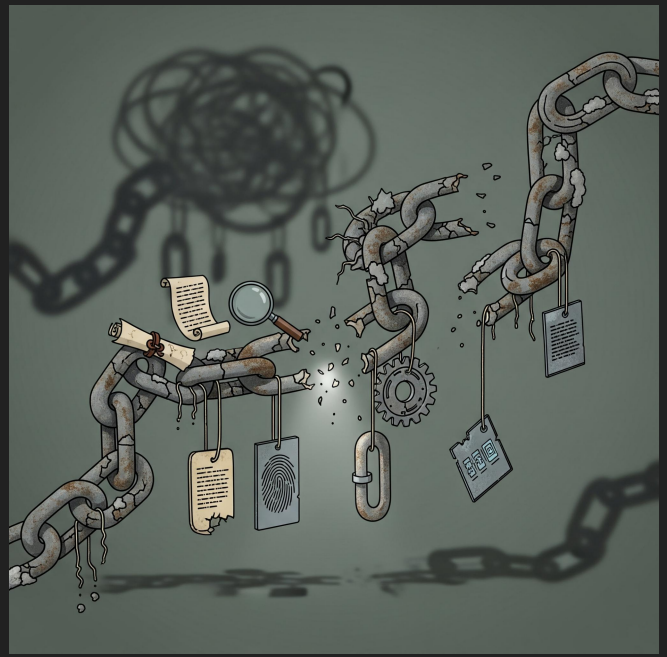
Verification



Create a "birth certificate" for every artifact we build

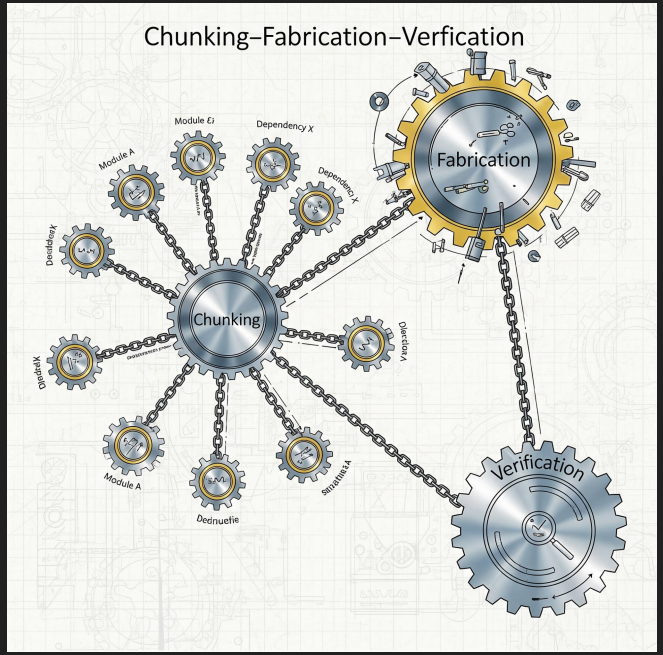
Challenge for Verification

Complete & Accurate



Core Conclusion

The lack of General Dependency management



Benefits in Practice

- CI Pipeline Orchestration, from 300+ to ~20 tasks
- Autonomous driving department, add a new car model from 2 months to 3 days
- Pipelines are independent of CI server, our system's build logic must be identical
- Security Risk Tracing
- Architecture Governance
- ……

What Can You Do Tomorrow

Refactor Your Build Scripts

Automate Your CI/CD Infrastructure

Thank you!

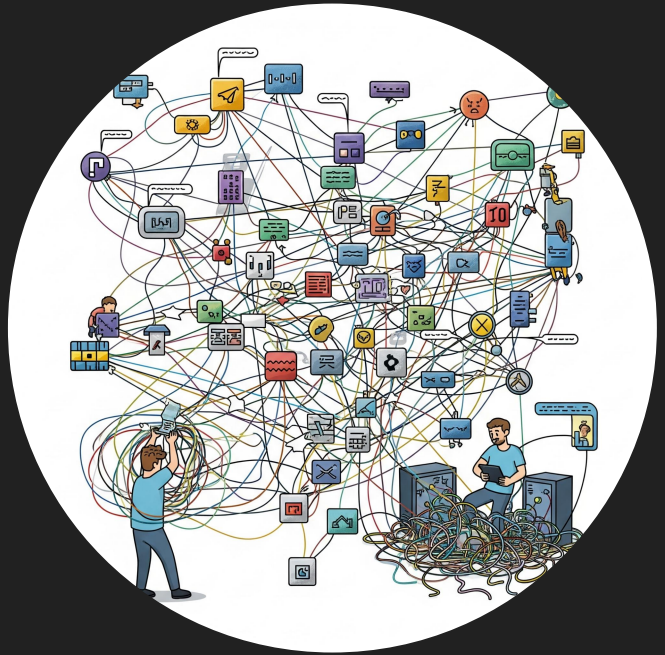
Chai Feng <chaifeng@odd-e.com>

2025-10

规模下的核心痛点

当“广义依赖”网络变得如此庞大且不可见时：

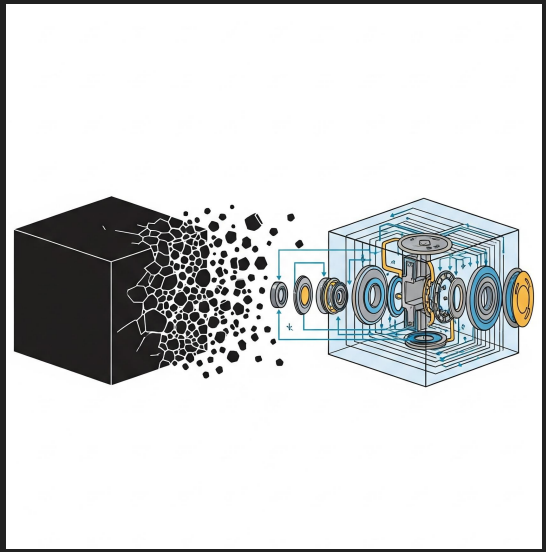
- 变更扩散路径不清晰
  - 底层库的一个小改动，影响范围是哪里？没人能说清。
  - 无法实现精确、自动化的系统更新。
- 集成滞后且脆弱
  - 底层依赖变更后，无法及时、完整地触发全系统构建。
  - 集成过程充满人工干预和“祈祷”。
- 构建效率低下
  - 一次完整的发布构建需要 7-8 小时。
  - “全局扫描”和“全量构建”成为常态，严重拖慢迭代速度。



必然的演进：从“黑盒遍历”到“显性工程化治理”

黑盒遍历

- 工作方式：构建工具（如 Make/Gradle/Bazel）扫描所有源码和依赖，自动计算出需要做什么。
- 优点：简单、直接。
- 缺点：随着规模增长，即使无代码变更，全局扫描本身也成为巨大浪费，是研发流程的最大瓶颈。



显性工程化治理

- **显性化“分而治之”**：我们必须主动地、显式地将庞大的工程体系，拆分成多个独立的、有清晰边界的单元。
- **提升依赖管理维度**：我们必须将基于“广义依赖”的管理，从一项边缘的配置工作，提升为驾驭规模化复杂性的核心工程能力。

显性治理的三大核心实践

拆解 (Chunking：make it small and bounded)

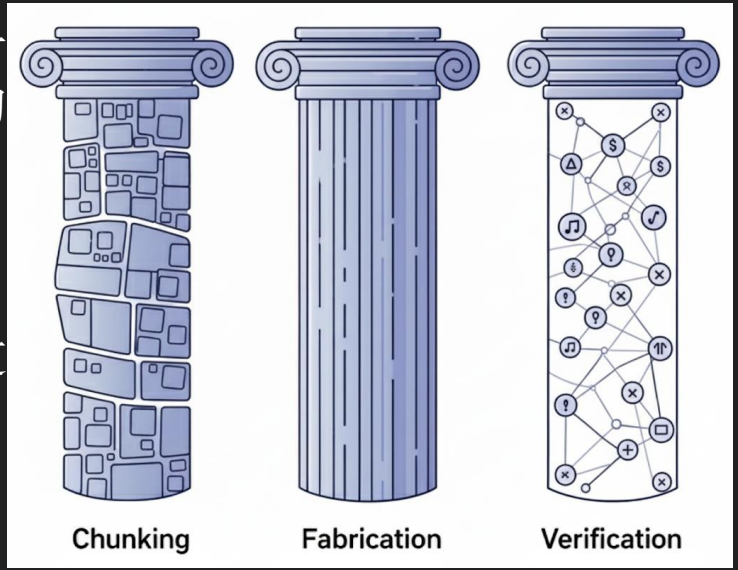
- 基于其广义依赖关系，把庞大的整体，逐层拆解成一系列具有清晰边界、可独立管理、存在层级关系的构建单元。

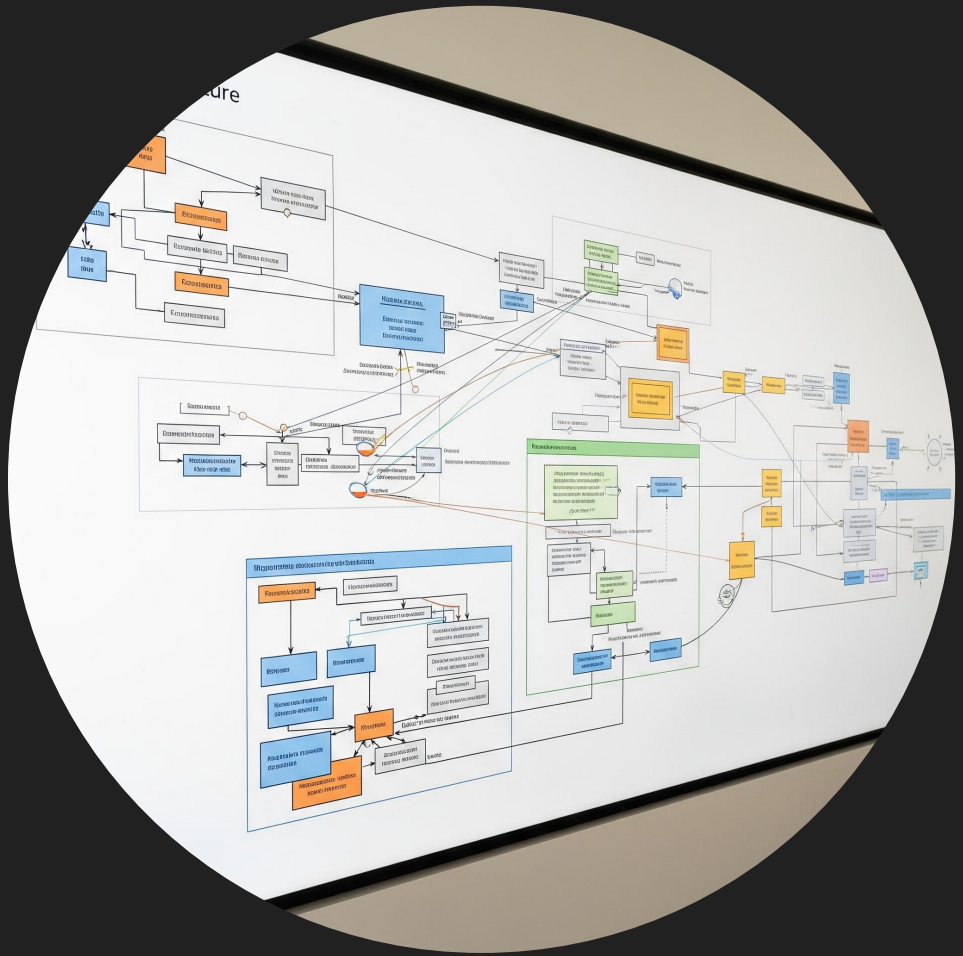
封装 (Fabrication：make it whole and shippable)

- 依据单元声明的广义依赖，通过确定性的生产制造过程，将其可靠地构建并集成组装为可交付的产品。

验证 (Verification：prove it and trace it)

- 建立可追溯的证据，证明最终产品的构成与其声明的广义依赖关系完全一致。





拆解（Chunking）

目标：

为了实现有效的“分而治之”，我们必须让拆分出的每一个单元，都做到边界清晰，并且其广义依赖的声明必须精准且完整。

这为什么难？

拆解的挑战：广义依赖全景图的“不可见”与“不完整”

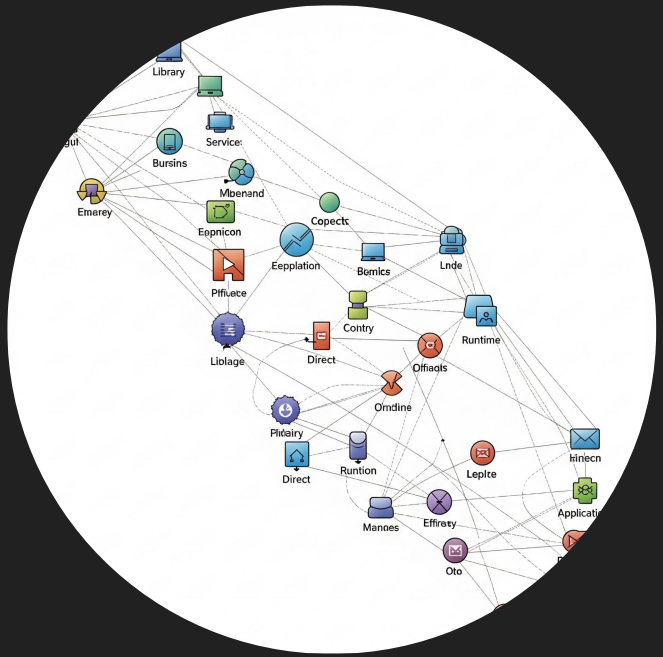


我们失败的根源在于，我们操作的“地图”（广义依赖全景图）本身就是残缺的。

- 看不见的“暗物质” (不完整)
  - CI服务器上的环境变量、基础镜像里的curl版本、开发者本地的PATH路径...
  - 这些隐式依赖真实地影响着构建结果，但它们从未被正式声明和管理。
- 被切断的连接 (不可见)
  - 一个C++库的变更如何影响Java服务？依赖关系通常被隐藏在CI脚本的实现细节中，任何单一的依赖视图都看不到这条连接。
  - 通过相对路径 (../../common/include) 建立的“幽灵依赖”，完全绕开了构建系统的感知。

拆解 (Chunking)，就是要把构成广义依赖的所有“暗物质”和“幽灵依赖”全部显性化。

“拆解”的重点



- **精准拆解构建上下文**
  - 拆的不仅是源码，更是其背后的完整构建上下文。这个上下文必须完整地捕获该单元所有的广义依赖：工具链、库、配置、环境变量等。
  - 每一个构建单元都必须拥有一个独立的、精准且完备的构建上下文定义。
- **发现并固化隐式依赖**
  - 将CI环境、基础镜像、外部工具等所有生产要素（即广义依赖）版本化、声明化。
  - 让所有的“不可见”都变得“可见”。
- **治理连接处的依赖断裂**
  - 单元间的依赖关系，是广义依赖全景图的一部分，必须被显式声明
  - 用声明式关系取代人工脚本和相对路径，将隐式的单元间依赖显性化

封装（Fabrication）

目标：

实现可复现式构建（Reproducible Build），确保对于同一组广义依赖输入，其产出是可被验证为一致的。

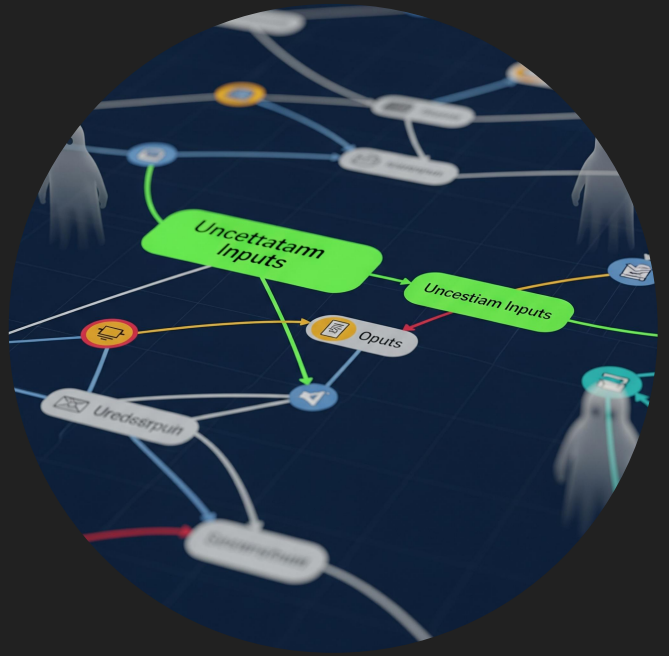
这为什么难？



“封装”的挑战：输入的“非确定性”与过程的“不可靠”

即使我们看清了所有广义依赖（输入），如果处理这些输入的过程本身充满“惊喜”，结果依然是灾难。

- 输入的非确定性
  - 版本与配置的组合爆炸：同一套代码，需要支持多平台、多硬件、多客户定制，导致有效的广义依赖输入组合呈指数级增长。
  - “幽灵依赖”的破坏：当一个未被声明的广义依赖发生变更时，构建系统会错误地复用旧的缓存，导致构建出一个“错误”的正确结果。
- 过程的不可靠
  - 非封闭行为 (Non-hermetic Builds)：构建过程中依赖了未精准声明的输入（如绝对路径、依赖时间戳、依赖随机数等），污染了广义依赖集。
  - 手工“热修复”：为了应急而绕过标准流程的手工操作，引入未被追踪的输入变更，破坏了过程的稳定性和可重复性。



封装 (Fabrication)，就是要消除所有“非确定性”，建立一个纯粹、封闭、幂等的生产过程。

“封装”的重点

- **建立封闭的构建行为 (Hermetic Builds)**
  - 构建沙箱化：禁止构建过程中引入未声明的输入，确保广义依赖的纯净和确定。
  - 消除所有不确定性来源（时间戳、随机主机名、绝对路径等）。
- **保障过程的幂等性**
  - 任何构建任务，无论执行一次还是一百次，只要其声明的广义依赖不变，结果必须完全相同。
  - 杜绝因上游不稳定或手工干预破坏流程稳定性。
- **管理配置组合爆炸**
  - 通过特征开关（Feature Flags）和声明式配置，将产品配置显式地作为广义依赖的一部分来管理。



验证：为何如此难？



目标：

为每一个交付的制品提供无可辩驳的“身份证明”，使其全部的广义依赖和生产过程都透明、可追溯、可审计。

这为什么难？

验证的挑战：缺乏统一证据链



当信任必须被工程化时，口头承诺和个人经验毫无意义，证据是唯一的通行证。而这份证据，就是对一个制品全部广义依赖的完整记录。

供应链安全的三重质询，完美揭示了证据链断裂的风险：

- “源码可信吗？”(仅验证了广义依赖中的一小部分)
- “产品真是用这份源码构建的吗？”(挑战“源码”到“产品”的转换过程)
- “构建过程本身安全吗？”(挑战构建过程所依赖的其它所有广义依赖，如基础镜像)

验证 (Verification)，就是要为每一个制品，建立一条从源头（所有广义依赖）到终点，完整、统一、不可篡改的证据链。

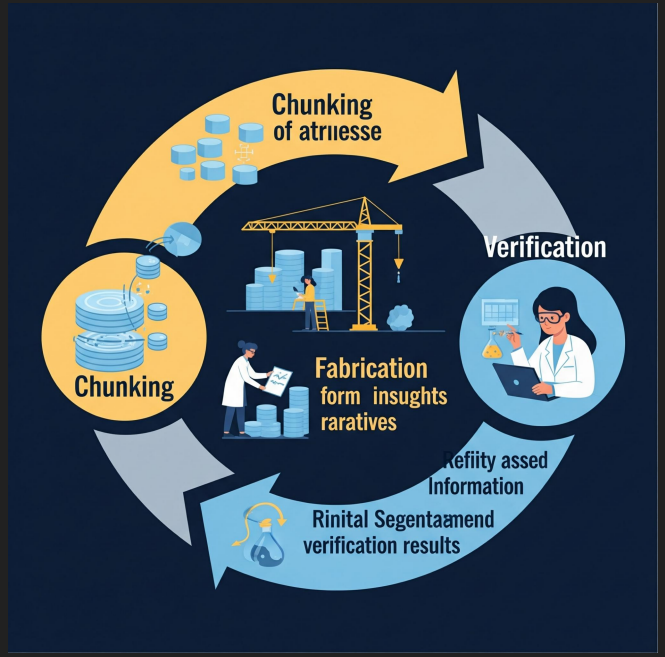
“验证”的重点



- 建立完整的证据链
  - 超越源码版本：记录所有影响构建的广义依赖——工具链版本、环境变量、基础镜像哈希等。
  - 生成详细的构建产物清单和构建过程的来源证明。
- 确保证据的准确性
  - 制品的任何一个字节级差异，都必须能在其广义依赖证据链中找到对应的输入变更。
- 实现可复现构建 (Reproducible Build)
  - 验证的黄金标准：这是对整条广义依赖证据链最强有力的物理保证。
  - 它将信任从 “相信我 (Trust me)” 转变为 “你可以自己验证 (You can verify it)”。

构建基于广义依赖的自验证闭环

- 精确的 “拆解” (Chunking)，**定义**了清晰、有边界的单元及其广义依赖。
- 清晰的单元及其广义依赖，**驱动**了可靠的 “封装” (Fabrication) 生产过程。
- 可靠的过程，**产出**了带完整广义依赖证据链的制品，可用于 “验证” (Verification)。
- “验证” 的结果（如一次不可复现的构建），反过来**暴露**了“拆解”的不足（发现了新的隐式依赖），驱动我们持续改进。



核心结论

我们开头提到的构建效率、可靠性、漏洞追溯、CI编排、架构治理等看似孤立的问题...

其共同根源都是:

广义依赖管理的缺失

通过 “拆解 (Chunking) - 封装 (Fabrication) - 验证 (Verification)” 的显性治理框架，我们可以将这些问题在一个统一的视角下，系统性地解决。





最终目标

建立一个数据驱动的

可追溯、可自验证的

软件生产体系



谢谢