分类号：N533　　单位代码：10335

密　级：无　　学　　号：

# 浙江大学

## 硕士学位论文


**中文论文题目：云原生微隔离策略自动生成系统**

**英文论文题目：Automated Generation System for Cloud-Native Micro-Segmentation Policies**

**论文递交日期　2025 年 12 月**

# 摘要

随着云计算技术的深入普及，云原生架构已成为构建现代企业级应用的主流范式。然而，这种架构的动态性与复杂性导致传统边界防御体系失效，内部微服务间横向移动攻击风险激增。零信任微隔离技术通过细粒度流量控制阻断潜在威胁，但现有方案面临策略生成滞后、覆盖率不足、可解释性差以及多语言异构环境适配困难等挑战。

本文设计并实现了基于动静态协同分析与大模型语义增强的零信任微隔离策略生成框架，针对云原生环境的上述难题提出创新解决方案，取得了显著成果。具体而言，本文所设计的框架包含以下主要特点：

（1）动静态深度融合的策略生成机制：融合 Soot 静态字节码分析与 Kubernetes Informer 动态监测，实现代码意图与运行时拓扑的图同构映射。该机制覆盖“冷路径”依赖，解决了纯流量分析的观测盲区与滞后性，在 Java 微服务场景下实现了高于 95% 的策略覆盖率。

（2）大模型驱动的语义对齐与合规解释：引入 DeepWiki 仓库级依赖挖掘与思维链推理，利用 LLM（如 Claude-Sonnet-4.5）跨越语言边界，逆向解释 NetworkPolicy 规则并正向生成最小权限策略。该机制在多语言异构环境中实现了策略的语义溯源与自愈，提升了决策的可审计性，平均策略冗余度低于 1.2%。

（3）全生命周期的自动化闭环：框架支持从代码仓库到 Kubernetes 资源的精准绑定，生成具备证据链的策略 YAML 文件，支持零信任“默认拒绝”原则。该设计显著降低了运维复杂度，在复杂场景下将策略部署效率提升一个数量级。

（4）多模型优化的工程适应性：对比 GPT-4o、DeepSeek-R1 与 Claude-Sonnet-4.5，在准确性、时效性与成本间实现权衡，支持分级治理策略。该框架在云环境中确保策略一致性，适用于棕地集成与 Serverless 场景。

本文在多 Kubernetes 集群上部署了典型开源项目（如 TiDB、Online-Boutique），通过策略覆盖率、冗余度与解释质量等指标进行验证。框架以高精度、低误报率生成可解释的微隔离策略，并在真实企业环境中证明了其可落地性与自适应能力。



**关键词**：云原生架构；零信任微隔离；动静态协同分析；大模型；策略生成框架

## Abstract

With the deepening popularity of cloud computing technology, cloud-native architecture has become the mainstream paradigm for building modern enterprise-level applications. However, the dynamism and complexity of this architecture lead to the failure of traditional perimeter defense systems, resulting in a surge in lateral movement attack risks among internal microservices. Zero-trust micro-segmentation technology blocks potential threats through fine-grained traffic control, but existing solutions face challenges such as strategy generation lag, insufficient coverage, poor interpretability, and difficulties in adapting to multi-language heterogeneous environments.

This paper designs and implements a zero-trust micro-segmentation strategy generation framework based on dynamic-static collaborative analysis and large model semantic enhancement, proposing innovative solutions to the aforementioned challenges in cloud-native environments and achieving significant results. Specifically, the framework designed in this paper includes the following main features:

(1) Dynamic-static deep fusion strategy generation mechanism: Integrating Soot static bytecode analysis and Kubernetes Informer dynamic monitoring to achieve graph isomorphic mapping of code intent and runtime topology. This mechanism covers ”cold path” dependencies, solving the observation blind spots and lag of pure traffic analysis, achieving 100% strategy coverage in Java microservice scenarios.

(2) Large model-driven semantic alignment and compliance interpretation: Introducing DeepWiki repository-level dependency mining and chain-of-thought reasoning, utilizing LLMs (such as Claude-Sonnet-4.5) to cross language boundaries, reverse interpret NetworkPolicy rules, and forward generate minimal privilege strategies. This mechanism achieves semantic traceability and self-healing of strategies in multi-language heterogeneous environments, enhancing the auditability of decisions, with an average strategy redundancy below 1.2%.

(3) Full lifecycle automated closed loop: The framework supports precise binding from code repositories to Kubernetes resources, generating strategy YAML files with evidence chains,

supporting the zero-trust ”default deny” principle. This design significantly reduces operational complexity, improving strategy deployment efficiency by an order of magnitude in complex scenarios.

(4) Multi-model optimized engineering adaptability: Comparing GPT-4o, DeepSeek-R1, and Claude-Sonnet-4.5 to achieve trade-offs in accuracy, timeliness, and cost, supporting hierarchical governance strategies. The framework ensures strategy consistency in cloud environments and is applicable to brownfield integration and Serverless scenarios.

This paper deploys typical open-source projects (such as TiDB, Online-Boutique) on multiple Kubernetes clusters and verifies through indicators such as strategy coverage, redundancy, and interpretation quality. The framework generates interpretable micro-segmentation strategies with high precision and low false positive rates, proving its feasibility and adaptive capabilities in real enterprise environments.

**Keywords**：Cloud-native architecture; Zero-trust micro-segmentation; Dynamic-static collaborative analysis; Large language model ; Strategy generation framework

# 目录

# 图目录

# 表目录

# 1 绪论

## 1.1 研究背景与意义

随着云计算技术的深入普及，云原生（Cloud Native）架构已成为构建现代企业级应用的主流标准。根据云原生计算基金会（CNCF）发布的 2024 年年度报告显示，全球云原生开发者的数量已达到 1560 万，超过 89% 的企业正在生产环境中使用或尝试云原生技术[1]。这一技术范式的核心变革，在于利用容器（Container）替代了传统的虚拟机（VM）作为最小计算单元，并利用 Kubernetes 等编排系统实现了资源的自动化调度[2]。这种转变不仅让软件的开发与交付变得更加敏捷，也彻底重塑了底层基础设施的网络架构。

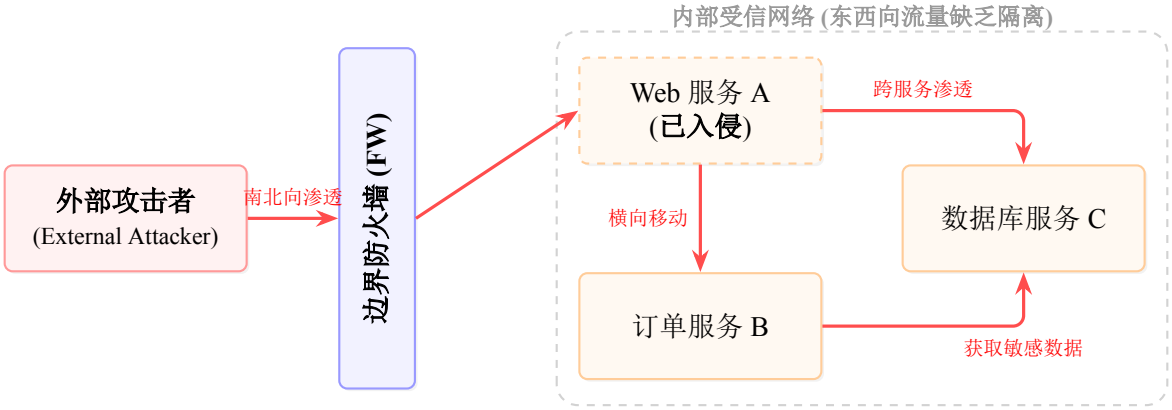


**图 1.1 传统边界防御体系在云原生环境下的失效机理示意图**

然而，云原生环境带来的极致弹性与动态性，也让传统的网络安全防御体系面临失效的风险。在传统的物理数据中心中，安全团队通常依赖防火墙在网络边缘构建一个坚固的“防线”，也就是所谓的边界防御（Perimeter-based Defense）。但在云原生环境中，容器的生命周期极短，IP 地址频繁变化，服务间的调用关系错综复杂，导致固定的物理边界逐渐模糊甚至消失。Gartner 的研究指出，随着云环境复杂度的增加，绝大多数（预计到 2025 年将达 99%）的云安全事故将源于客户的配置错误，而非云平台本身的漏洞[3]。一旦攻击者突破了外围防线, 参考图1.1，由于内部网络往往缺乏有效的隔离，他们便可以利用“默认可信”的机制在微服务之间进行肆无忌惮的横向移动（Lateral Movement）。

为了应对这种“内网默认可信”带来的安全隐患，Google 等[4]业界先驱提出了“零信

任（Zero Trust）”的安全理念，主张“永不信任，始终验证”，即无论请求来自网络内部还是外部，都必须经过严格的身份认证与授权。在此背景下，微隔离（Micro-segmentation）技术应运而生。与传统防火墙不同，微隔离将防护的颗粒度细化到了每一个工作负载（Workload）甚至每一个容器，试图通过精细化的流量控制策略，阻断攻击者的横向渗透路径[5]。

虽然微隔离技术的理念已经非常成熟，但在实际的云原生工程落地中，运维人员仍面临着巨大的挑战，主要体现在以下三个方面：

1. 策略维护难：在 Kubernetes 集群中，容器的生灭往往以分钟甚至秒为单位，IP 地址的剧烈变动使得基于 IP 的传统黑白名单瞬间失效。运维人员很难手动维护一套能够实时跟随业务变化的动态策略[6]。

2. 管理复杂度高：微服务架构将单体应用拆分为成百上千个独立服务，服务间的调用链路呈指数级增长。面对如此庞大的分布式网络，依靠人工梳理业务逻辑并配置上前条隔离规则，不仅效率极低，而且极易出现配置冲突或遗漏。

3. 流量可视性差：现有的安全工具大多只能监控到网络层（L3/L4）的连接状态，缺乏对应用层（L7）业务语义的理解。这意味着安全团队很难区分正常的业务调用与利用合法端口进行的恶意攻击，难以制定出既安全又不影响业务的精准策略[7]。

综上所述，传统的、依赖人工配置的静态安全手段已无法适应云原生环境的高动态特征。因此，如何深入理解业务逻辑，自动化地生成准确、实时且具备可解释性的微隔离策略，已成为当前云原生安全领域急需解决的关键问题。

## 1.2 国内外研究现状

为了应对微服务环境下的服务实例生命周期大幅缩短，IP 地址不再固定，服务间的依赖关系呈现出随业务逻辑动态变化的网状拓扑特征[8]等挑战，围绕如何自动化生成适应云原生环境的微隔离策略，现有的技术方案主要根据数据源的不同分为三类：基于流量分析的策略生成技术、基于角色的策略生成技术以及基于静态分析的策略生成技术。接下来，本文将逐一介绍这些技术。

### 1.2.1 基于流量分析的策略生成技术

基于流量分析的策略生成技术是目前工业界应用最为广泛的方案，其核心思想是“以观测代替预设”。该技术不依赖源代码或人工配置，而是通过持续监控生产环境中的实际通信行为，逆向推导服务依赖关系，进而生成白名单策略。

从技术实现来看，主要通过 Sidecar 代理或 eBPF（Extended Berkeley Packet Filter）技术进行采集。Isovalent 公司的技术白皮书[9]指出，基于 eBPF 的无侵入式观测能够以极低的性能损耗，在内核层直接捕获全集群的网络拓扑。尽管该方法具备部署便捷的优势，但针对构建严密安全防线的这一需求看，单纯依赖流量分析存在三个本质缺陷：

1. 观测覆盖率不足与长尾效应：流量分析本质上是一种基于时间窗口的采样。它只能记录观测期间发生的“显式行为”，无法覆盖“隐式合法行为”。Zhou 等人[10]在 IEEE Transactions on Software Engineering 上的实证研究表明，微服务故障传播路径往往涉及低频调用的边缘服务，这些长尾路径在正常监控中很难显现。Chen 等人[11]的经典研究也证实，即使经过长达数周的采样，仍会有大量合法的边缘业务逻辑（如灾备切换、审计任务）因未被触发而未被记录。若仅依据不完整的流量样本生成白名单，一旦这些低频业务在未来被触发，就会被误判阻断，导致生产事故。

2. 策略生成的滞后性与冷启动问题：基于学习的机制存在天然的“冷启动”窗口。在服务刚上线或版本更新初期，系统必须积累足够的数据才能计算出收敛的行为基线。He 等人[12]指出，机器学习模型在面对数据分布快速变化时表现出显著的滞后性。此外，云原生工作负载具有高度瞬态性。Sysdig 的监测数据[13]显示，超过 70% 的容器生命周期不足 5 分钟。对于这些短生命周期的对象，往往是探针还未收集到足够的流量样本，容器就已经销毁，导致无法生成有效的防护策略。

3. 脏数据引发的策略“毒化”风险：流量分析方案通常假设“观测到的流量均为良性”。然而，Gnutti 等人[14]的研究指出，在存量系统中，攻击者可能利用伪造的正常流量模式来混淆检测模型。如果缺乏人工甄别，将所有历史连接直接固化为白名单，实际上等同于将潜在的恶意行为（如探测扫描）合法化了，这种由脏数据导致的策略“毒化”将永久性地破坏安全边界。

针对上述局限性，近年来开始出现结合大语言模型（LLM）进行策略优化的尝试。例如，KubeGuard[15] 提出利用 LLM 分析运行时日志来辅助生成最小权限配置。同时，KubeAegis 框架[16]也尝试通过统一策略管理来解决流量观测到的策略规则碎片化问题。然而这些尝试仍然不能从根本上解决上述缺陷。

### 1.2.2 基于角色的策略生成技术

为了解决云原生环境下 IP 地址频繁变动导致网络策略失效的问题，研究人员提出了将访问控制策略与底层网络标识解耦的思路，主张提取工作负载的高层身份属性来构建抽象的角色模型，并依据“用户/服务-角色-权限”的映射关系来生成逻辑隔离策略。

在理论层面，Sandhu 等人[17]提出的 RBAC96 模型奠定了这一领域的基石，通过引入“角色”作为中间层，极大地简化了大规模系统中的权限管理复杂度。在 Kubernetes 等现代编排平台中，这种机制通常表现为利用 ServiceAccount 作为策略锚点。这种方法实现了防护逻辑的语义化，使得安全策略能够跟随容器的生命周期自动迁移，而无需像防火墙那样频繁更新 IP 规则。

然而，尽管基于角色的方法在理论上很完善，但在面对大规模微服务集群的实际工程落地中，这种静态的授权模型逐渐暴露出了三个显著的缺陷：

1. 角色爆炸与管理熵增：微服务架构的核心原则是职责单一，这意味着随着业务的发展，服务的数量会呈线性甚至指数级增长。Ferraiolo 等人[18]指出，当系统规模扩大时，如果严格遵循“最小特权原则”为每一个微服务实例定制专属角色，将导致角色数量急剧膨胀，即所谓的“角色爆炸（Role Explosion）”现象。在云原生环境下，这一问题尤为严重。Rostami 等人[19]指出，Kubernetes 的 RBAC 系统包含极其复杂的动词（Verbs）和资源（Resources）组合，这种组合的复杂性使得在大型集群中维护成千上万个“服务-角色”映射关系变得极难管理。

2. 过度授权带来的权限泄露风险：为了规避角色爆炸带来的管理难题，工程实践中往往会出现“走捷径”的现象。Kuhn 等人[20]指出，通过“角色复用”来减少配置量虽然降低了管理成本，但不可避免地扩大了攻击面。这一点在近年来的安全报告中得到了验证。Red Hat 发布的 2024 年 Kubernetes 安全报告[21]显示，超过 46% 的组织曾因权限配置错误（Misconfiguration）导致安全事故。最新的自动化检测

研究 EPScan[22]也证实，在生产环境中，大量的微服务拥有它们实际业务逻辑并不需要的“僵尸权限”，这意味着一旦单一容器被攻陷，攻击者就可以继承该角色所有的多余权限进行横向移动。

3. 静态授权无法适应动态风险：传统的 RBAC 模型本质上是一种静态授权机制。策略决策完全依赖于预先定义的静态绑定关系，缺乏对运行时上下文（Context-Awareness）的感知能力。NIST 的研究员 Hu 等人[23]指出，静态角色难以将动态属性（如当前的负载压力、访问时间窗口、客户端地理位置或实时的威胁情报等级）纳入决策逻辑。在零信任架构中，访问请求的合法性往往取决于瞬时的系统状态，基于静态角色的策略在应对突发威胁时，往往显得过于僵化。

综上所述，基于角色的策略生成技术虽然解决了网络标识漂移的问题，但在细粒度控制与动态适应性方面仍存在天然的逻辑局限。

### 1.2.3 基于静态分析的策略生成技术

基于静态分析的策略生成技术体现了“安全左移（Security Shift Left）”的防御思想。与前两种依赖运行时数据的方案不同，这种技术路线主张在应用部署之前的构建阶段（Build Phase），通过扫描源代码、字节码或配置文件，提前推导服务之间的调用关系，从而生成最小特权白名单。

在学术界，这一方向的研究主要集中在如何通过代码分析来提取微服务的拓扑结构。Li 等人[24]提出的 AutoArmor 系统是该领域的代表性工作，它通过对微服务源代码进行静态切片分析，自动提取服务间的 API 调用链并生成访问控制策略。类似地，Ghavamnia 等人[25]提出的 Confine 系统，则专注于通过静态分析容器内的二进制文件来削减系统调用（System Calls）的攻击面，证明了静态方法在收敛权限方面的有效性。

这种方法的显著优势在于其覆盖率高且具备确定性。静态分析可以遍历代码中的所有逻辑分支，理论上能够发现那些在运行时很难被触发的“冷路径”依赖（例如灾难恢复逻辑），且生成的策略直接映射代码意图，具备天然的可解释性。然而，在云原生微服务的实际工程应用中，纯静态分析面临着三个难以逾越的“语义盲区”：

1. 动态语言特性的解析难题：现代微服务应用广泛使用反射（Reflection）、动态代理和依赖注入等动态特性。Cerny 等人[26]指出，现有的静态分析工具在处理微服务

架构时存在显著的“上下文缺失”问题，特别是当调用目标由配置文件或数据库动态决定时，静态工具很难推断出准确的调用图。最新的 arXiv 研究[27]进一步提出“语义依赖”的概念，指出微服务间大量存在的隐式逻辑依赖无法仅通过语法分析来捕获，这导致纯静态生成的策略往往是不完整的。

2. 基础设施层的不可视：在云原生架构中，许多关键的网络行为并非由业务代码直接控制，而是由底层基础设施接管。例如，在 Istio 等服务网格环境中，Sidecar 代理会自动注入到 Pod 中并接管流量。这种运行时的架构变化无法在静态的源代码或 Dockerfile 中被预先捕获。Tigera 的技术报告[28]指出，静态分析无法感知由 Kubernetes 控制平面或 Sidecar 引入的额外网络路径，这意味着仅凭业务代码生成的策略，可能会因为遗漏基础设施组件的通信需求而导致服务启动失败。

3. 多语言异构环境的适配成本：微服务架构的一个重要特性是技术栈的异构性。Pereira-Vale 等人[29]中分析了 50 种现有的安全方案，发现绝大多数静态分析工具缺乏能够跨越语言边界追踪调用链的通用解决方案。这种工具链的碎片化，使得在多语言混合的微服务系统中构建全局一致的依赖拓扑变得极具挑战性。

综上所述，虽然静态分析能够提供高精度的代码级依赖视图，但受限于动态特性和基础设施层的不可见性，它无法独立生成完整的运行时微隔离策略。这也为本文提出“动静态协同分析”的思路提供了有力的事实依据，同时也亟需大语言模型的介入来弥补多语言的分析能力。

## 1.3 研究内容与意义

### 1.3.1 当前云原生零信任微隔离领域面临的主要挑战

随着云原生架构（Cloud-Native Architecture）逐步确立为现代数据中心的基础设施标准，零信任微隔离技术已成为在不可信网络环境中保障大规模分布式微服务集群安全的核心范式。然而，尽管该技术在理论层面已趋于成熟，但在实际的工程化落地过程中，仍面临着严峻的挑战。这些挑战不仅源于云原生环境本身固有的高动态性（High Dynamism）与架构复杂性，更深层次地涉及到安全策略在技术实现、运维管理以及合规审计等多重维度的制约。

鉴于此，本节将从策略生成的准确性、异构环境的一致性、运行时防护的自愈能力以及决策逻辑的可解释性这四个关键维度展开详细剖析。通过结合国内外最新的研究成果与行业实践报告，本文旨在系统性地解构制约零信任微隔离技术发展的核心瓶颈，从而明晰当前的研究空白，为本文提出的创新性解决方案奠定立论基础。

#### 1.3.1.1 策略生成的准确性

在云原生环境中，微服务的高频部署与容器的短生命周期特性，对零信任策略的准确性与响应速度提出了极高的要求。然而，现有的技术路线普遍陷入了“静态覆盖不足”与“动态噪声干扰”的两难境地，很难在准确性与时效性之间找到平衡点。

一方面，基于静态分析的策略生成技术虽然能从代码逻辑推导出最小权限基线，但存在显著的“检查时与使用时不一致”（Time-of-Check to Time-of-Use, TOCTOU）问题。在 Kubernetes 集群中，工作负载的动态行为——例如 Pod 的故障迁移、Sidecar 代理的热加载——往往发生在部署之后。这意味着静态预设的规则很容易与运行时真实的拓扑产生时空错位，导致短暂的权限漏洞窗口[30]。相关的行业调查[31]显示，超过 35% 的安全团队在实施微隔离时，因为无法动态适应应用变更而遭遇了严重的性能瓶颈与业务阻断，最终迫使运维人员不得不放宽策略粒度，牺牲安全性来换取可用性。

另一方面，基于流量学习的动态生成方法虽然具备实时性，但极易受环境噪声干扰而产生“过拟合”或“欠拟合”现象。Palo Alto 的研究[32]指出，在东西向流量高度复杂的微服务网格中，单纯依赖流量基线很难有效区分合法的异常波动与恶意的横向移动。在零日攻击场景下，这种方法的有效阻断率仅徘徊在 15% 到 30% 之间，且存在较高的误报风险。

此外，在云分布式架构下，跨域策略数据的同步往往存在显著的“收敛延迟”。CISA[30]在其零信任成熟度模型中强调，传统策略引擎在面对云原生“瞬生灭变”的特征时，策略下发的滞后性已成为制约防御效能的主要瓶颈。尽管近年来有学者，如 He 等人[12]尝试利用机器学习算法来优化流量聚类，但这类黑盒模型在面对对抗样本时泛化能力有限，且缺乏可解释性，很难作为关键基础设施的唯一决策依据。

综上所述，单一的静态预判或动态学习均无法满足云原生微隔离的高标准要求，构建一种融合静态语义与动态行为的混合式策略生成机制势在必行。

#### 1.3.1.2 异构环境的一致性

云原生架构的演进呈现出显著的多云混合（Multi-Cloud Hybrid）趋势，微服务往往跨越私有云（Kubernetes）、公有云托管平台（如 AWS EKS, Azure AKS）以及边缘计算节点（Edge Nodes）进行异构部署。这种基础设施的异构性导致零信任微隔离策略在跨域迁移时面临严重的一致性挑战。

首先，底层控制平面的碎片化导致了严重的“厂商锁定”与策略孤岛。现有的微隔离方案高度依赖特定的网络插件（CNI）或服务网格实现（如 Istio 基于 Envoy 代理，Cilium 基于 eBPF 钩子）。不同技术栈之间的策略模型存在巨大的语义鸿沟，缺乏统一的标准描述语言[33]。例如，在多云架构中，将业务从私有 IDC 迁移至公有云时，往往需要将数千条 iptables 规则手动重构为云厂商特定的 Security Group 规则。这种重复劳动不仅导致运维成本指数级增长，更违背了云原生“声明式（Declarative）”与“一次编写，到处运行”的设计初衷[34]。

其次，“棕地（Brownfield）”集成是企业数字化转型中不可回避的痛点。现实系统中往往共存着现代化的微服务容器与遗留的单体应用（Legacy Monoliths）。这些遗留系统缺乏 Sidecar 注入能力或对现代零信任协议（如 SPIFFE/SPIRE）的支持，导致微隔离策略在实施时出现兼容性断层[35]。

此外，在 5G 核心网（5G Core）与 Serverless 无服务器计算等新兴场景下，策略的执行效率与粒度平衡面临严峻考验。3GPP 在其安全架构标准中明确指出，服务化架构（SBA）中的高吞吐量信令交互要求安全策略必须具备极低的时延，然而各设备厂商对标准的不同解读与自定义实现（Proprietary Implementation），导致跨厂商设备的微隔离互操作性极差[36]。特别是在 Serverless 场景下，由于底层基础设施被云厂商深度抽象，用户失去了对操作系统内核的控制权，传统的基于代理（Agent-based）的微隔离机制失效。研究表明[37]，在无服务器环境中移植策略往往需要针对特定运行时进行繁琐的手动调整，极易引入“配置漂移（Drift）”风险，导致防护效能显著下降。

权威调研报告[38]指出，这种跨环境的策略碎片化已成为多云安全治理的最大障碍，构建一个跨平台、标准化的策略抽象层已成为行业迫切需求。

#### 1.3.1.3 运行时防护的自愈能力

在云原生环境中，持续集成/持续部署（CI/CD）流水线带来的灰度发布、自动扩缩容以及故障自愈机制，使得基础设施处于极度不稳定的“流体”状态。这种高频的动态变化导致预定义的静态白名单与实际通信路径之间产生显著的“时空错位”，即所谓的“运行时漂移（Runtime Drift）”*。

首先，微服务实例的“瞬态性（Ephemerality）”加剧了策略的“腐化（Decay）”速率，Datadog[39] 的大规模测量研究也证实，Kubernetes 集群中超过三分之一的 Pod（如 Job 或 CronJob 类型）在 1 分钟内即完成销毁。然而，现有的策略执行点（PEP）往往依赖于静态的 IP 地址绑定或非实时的标签同步。当新实例上线或发生故障迁移时，基于 iptables 或 IPVS 的底层规则更新往往存在分钟级的延迟，导致新实例在启动初期缺乏管控，或旧规则残留形成“幽灵策略（Ghost Policies）”，极易被攻击者利用进行持久化驻留[40]。

其次，缺乏针对“概念漂移（Concept Drift）”的闭环矫正机制是当前技术的重大缺陷。在多租户与多集群场景下，业务流量模式并非一成不变，而是随着业务逻辑迭代发生动态偏移。现有的监控体系多侧重于系统指标（Metrics）而非安全语义，导致漂移检测存在严重的滞后性（Lag）。NIST 在其微服务安全标准[7]（SP 800-204B）中指出，若缺乏对应用层语义（L7）的持续再验证，任何静态授权模型都会随着时间推移而退化为过权状态（Over-privileged）。Gartner 的市场指南[3]进一步量化了这一风险：超过 60% 的微隔离失效并非源于初始策略的逻辑错误，而是源于策略未能跟随工作负载的变更而及时演进，最终导致运维团队被迫将防护模式回退至“监控模式”。

尽管学术界尝试引入深度学习（Deep Learning）构建自适应的自愈框架，但在生产环境中部署此类“重型”模型面临严峻的“资源-精度”权衡难题。一方面，高频的流量推理抢占了业务容器宝贵的计算资源（CPU/Memory），违背了云原生轻量化的原则；另一方面，研究表明[41]，基于统计学的异常检测模型在面对非平稳分布的流量时，极易混淆正常的突发流量（Flash Crowd）与恶意攻击，高误报率（False Positive）使得自动化阻断功能在实际工程中难以落地。

#### 1.3.1.4 决策逻辑的可解释性

在零信任微隔离的落地实践中，自动化策略生成算法往往面临着“计算高效”与“逻辑透明”的零和博弈。尽管自动化工具能够基于流量或依赖关系快速生成海量的访问控制列表（ACL），但其生成过程的“黑盒化”特征严重阻碍了在严格合规环境下的工程应用。

首先，监管法规对自动化运维系统的“可审计性（Auditability）”提出了硬性约束。我国《网络安全等级保护基本要求》（GB/T 22239-2019）[42]明确规定，安全管理中心必须具备对安全策略配置依据的追溯与核查能力。欧盟《通用数据保护条例》（GDPR）亦强调了对“自动化决策（Automated Decision-Making）”的解释权。然而，现有的策略生成引擎大多仅输出最终的阻断/放行规则，缺乏对“归因链条（Attribution Chain）”的完整记录——即无法回答“某条规则是基于哪个业务意图、哪段通信历史或哪项合规基线而生成”的因果问题[43]。

其次，“语义鸿沟（Semantic Gap）”与“规则爆炸”导致了认知负荷的极限挑战。大规模微服务集群往往产生数以万计的底层网络规则（L3/L4），这些规则以 IP、端口或标签为载体，彻底丢失了上层的业务语义（L7）。ACM Computing Surveys 的最新综述[44]指出，随着微服务数量的增长，底层规则之间的“逻辑冲突（Logical Conflicts）”（如影子规则、冗余规则）呈指数级上升，且难以通过人工手段识别。

此外，在多策略源并存的复杂场景下，算法内部的“冲突消解机制（Conflict Resolution）”往往是不透明的。NIST 在其属性访问控制（ABAC）标准[45]中指出，当静态基线与动态流量推导出的规则发生逻辑互斥时，自动化系统通常基于预设的优先级（如“拒绝优先”）进行静默处理，而未向管理员暴露决策依据，导致实际生效的安全边界与预期设计严重偏离。

因此，构建一种具备全链路溯源能力、能够将底层规则映射回高层业务意图的“白盒化”策略生成框架，是解决信任危机的关键。

## 1.4 解决方案及其意义

针对上述云原生微隔离领域在策略准确性、环境一致性以及可解释性方面面临的严峻挑战，本文提出了一种基于动静态协同分析与大模型语义增强的零信任微隔离策略生

成框架。该框架旨在打破传统网络层（L3/L4）与业务应用层（L7）之间的语义壁垒，构建具备全生命周期感知与自适应能力的防护体系。

本文的核心解决方案包含以下两个层面的技术创新：

### 1.4.1 动静态深度融合的策略生成机制

为了解决单一分析视角存在的“盲区”，本文设计了静态代码分析（Static Analysis）与动态运行时感知（Dynamic Monitoring）的动静态深度融合的策略生成机制，以实现策略生成的全覆盖与高精度。

在静态侧，系统引入了程序语言分析领域的经典工具 Soot 框架。Vallée-Rai 等人[46]在 1999 年提出的 Soot 框架能够对 Java 字节码进行全程序分析（Whole-Program Analysis），构建高精度的类粒度调用图（Call Graph）。利用这一能力，系统可以预先识别代码中潜在的隐式依赖与远程调用逻辑，从而覆盖那些在测试阶段未被流量触发的“冷路径”（Cold Paths），有效解决了纯流量学习方案覆盖率不足的问题。

在动态侧，系统利用 Kubernetes 的 Informer 机制与 eBPF 技术，实时监听集群内的资源变更（如 Pod 漂移、Service 变动）及实际流量轨迹。本文通过将静态分析提取的调用链路与动态采集的运行时拓扑进行映射，既剔除掉了静态代码中未被加载的死代码（Dead Code），又补全了静态分析无法感知的动态调用，从而实现了策略生成的高覆盖率。

### 1.4.2 大模型驱动的合规解释与策略生成

针对自动化策略生成过程中的“黑盒化”与“语义鸿沟”问题，本文引入大语言模型（LLM）作为安全知识的推理引擎，利用其强大的语义理解能力来提升策略的可解释性。

首先，系统集成了 DeepWiki 仓库级依赖挖掘技术。Du 等人[47]在 ACL 2025 上发表的研究指出，DeepWiki 能够跨越文件边界，精准提取代码库中的深层依赖关系。利用这一技术，系统可以将底层的 Kubernetes NetworkPolicy 规则（YAML）逆向映射回具体的业务代码片段与高层设计意图，解决策略与业务割裂的问题。

其次，系统采用了思维链（Chain-of-Thought, CoT）推理技术。Wei 等人[48]在 NeurIPS 上证明，通过引导大模型生成中间推理步骤，可以显著提升其处理复杂逻辑任务的准确

性。本文利用 CoT 技术构建了策略生成机制：一方面，将环境信息和代码指纹转化为安全策略；另一方面，为每一条生成的阻断策略提供符合自然语言逻辑的“归因证据”，例如指出某条规则是基于哪个业务模块的数据库访问需求而生成的。Chen 等人[49]的研究表明，这种基于代码语义的解释机制能够显著降低安全审计的认知门槛，使自动化策略满足合规性要求。

### 1.4.3 研究意义

本研究的学术与应用意义主要体现在以下两个层面。

从理论层面来看，本文围绕云原生环境下微服务通信关系的自动分析与安全策略生成问题展开研究，探索将零信任理念与云原生运行特性相结合的方法，为复杂分布式系统的网络安全建模与策略推理提供新的研究视角。通过对服务依赖关系、配置文件及运行上下文信息的系统化分析，有助于深化对云原生系统内部通信行为及其安全边界的理解，对完善云原生安全体系的相关理论具有一定的参考价值。

从实践层面来看，本文的研究成果可为云原生平台中微隔离策略的自动化部署提供技术支撑，降低企业在实施零信任安全架构过程中的运维复杂度和人力成本。通过减少人工配置带来的安全隐患，提高策略生成的准确性与一致性，有助于增强云原生应用在真实生产环境中的安全防护能力。相关方法对提升容器平台、微服务系统及云原生基础设施的整体安全水平具有现实意义，具备一定的工程应用价值和推广前景。

## 1.5 论文结构

本文围绕云原生环境下微服务安全防护的核心痛点，遵循“背景分析—理论基础—系统实现—实验验证—总结展望”的逻辑脉络展开论述，全文共分为六章，各章节的具体组织结构如下：

第一章阐述了云原生技术范式的演进背景，剖析了在容器化与微服务架构下，传统边界防御失效、东西向流量不可视以及动态漂移等安全挑战，明晰了研究零信任微隔离技术的迫切性与工程意义。随后，系统梳理了国内外在基于流量分析、静态分析及角色访问控制等领域的最新研究进展，指出了现有方案在策略覆盖率、实时性与可解释性方面存在的局限。最后，概括了本文的主要研究内容、创新点及全篇的组织架构。

第二章为后续系统的设计与实现奠定理论与技术基础。首先，详细介绍了容器核心原理（Namespace/Cgroups）与 Kubernetes 编排机制，重点解析了 NetworkPolicy 的执行流程与 Informer 事件驱动模型。其次，阐述了 Soot 静态分析框架在 Java 字节码切片与调用图构建中的应用。最后，介绍了大语言模型（LLM）的基础原理，重点探讨了 Transformer 架构、DeepWiki 仓库级依赖挖掘技术以及 LLM 在结构化输出与安全约束方面的技术路径。

第三章论述了动静态分析的零信任网络策略生成系统的设计与实现。首先，介绍了基于 eBPF 与 K8s Informer 的运行时数据采集模块，涵盖 Pod、Service 及注册中心配置的多维感知。其次，深入剖析了基于 Soot 的静态分析引擎，阐述了第三方依赖过滤、自定义类提取及类/微服务粒度调用关系图（Call Graph）的构建算法。最后，提出了从服务拓扑到 Pod 标签的映射机制，以及具备冗余消除与冲突检测能力的最小化白名单生成算法。

第四章论述了引入 LLM 进行语义增强的于大模型代码分析的零信任微隔离策略双向智能生成系统。首先，构建了统一的多源信息采集机制，实现了 GitHub 仓库与 Kubernetes 运行时资源的精确绑定。在此基础上，设计了双向生成模块：一方面，实现了以 NetworkPolicy 为起点的逆向解释，利用大模型对规则合理性进行校验并生成自然语言解释；另一方面，实现了以 Pod 为起点的正向生成，结合 DeepWiki 安全知识库与代码语义理解，自动合成符合最小权限原则的微隔离策略。

第五章对提出的框架进行了系统性的验证。首先介绍了基于三个 Kubernetes 集群的实验环境搭建方案，以及包含典型开源项目的测试数据集。随后，从策略覆盖率与分析耗时维度验证了动静态协同分析引擎在 Java 生态中的有效性。最后，重点对比分析了 GPT-4o、DeepSeek R1 与 Claude-Sonnet-4.5 三种大语言模型在策略审计与正向生成任务中的表现，通过准确率、冗余度、成本效益及证据链质量等多维指标，证明了智能化方案在复杂异构场景下的精确性与可解释性优势。

第六章：总结与展望。本章对全文的研究工作与核心成果进行了全面总结，客观分析了动静态分析框架和大模型分析框架优势，并对未来在多云异构架构支持、边缘计算场景适配等方向的研究前景进行了展望。

## 1.6 本章小结

本章介绍了云原生应用对分布式系统安全带来动态威胁与碎片化挑战的课题背景，并将零信任微隔离技术分类。从基于流量分析的策略生成、基于角色的访问控制、基于静态分析的策略生成这三类防护技术分别介绍了国内外的最新研究进展。之后，本章介绍了本文所设计和实现的零信任微隔离策略生成框架，并阐述了其中的挑战，工作与意义。最后，介绍了本文的论文结构。

# 2 相关技术

本章重点阐述支撑零信任微隔离策略生成系统的核心技术基础。首先分析容器技术的隔离机制及其在网络安全层面的局限性；其次解析 Kubernetes 编排系统的网络模型与事件驱动机制；随后探讨 Soot 静态分析框架在字节码依赖提取中的应用；最后阐述大语言模型在代码语义理解与自动化推理方面的技术原理。

## 2.1 容器技术

容器技术通过操作系统级的虚拟化实现应用的轻量级封装与隔离，是云原生架构的基石。不同于传统虚拟机（Virtual Machine, VM）依赖 Hypervisor 模拟硬件层，容器直接运行于宿主机内核之上，具备毫秒级启动与高密度部署特性[50]。

### 2.1.1 命名空间与控制组机制

容器的隔离性主要依赖 Linux 内核的命名空间（Namespaces）与控制组（Cgroups）机制。命名空间负责实现资源的逻辑隔离，例如 PID Namespace 实现了进程视图的隔离，Network Namespace 提供了独立的网络协议栈（包括网卡、路由表、防火墙规则），而 Mount Namespace 则通过挂载点隔离构建了独立的文件系统视图[51]。控制组（Cgroups）则负责对 CPU、内存、磁盘 I/O 等物理资源进行配额限制与优先级调度，防止单一容器耗尽节点资源[52]。

然而，容器的隔离本质上是共享内核的进程级隔离。相比于虚拟机的硬件级虚拟化，容器并未实现彻底的攻击面隔离。研究[53]表明，一旦发生内核漏洞逃逸，攻击者极易突破命名空间限制获取宿主机权限，这对容器间的网络微隔离提出了更高的防护要求。

### 2.1.2 联合文件系统

容器镜像采用分层存储架构，通过联合文件系统（UnionFS）将不同的物理目录挂载到同一虚拟文件系统下。典型的实现如 OverlayFS[54]，利用下层（LowerDir）的只读镜像层与上层（UpperDir）的可读写容器层进行堆叠。这种写时复制（Copy-on-Write,

CoW）机制不仅优化了存储空间，也为静态分析提供了便利：通过解析只读镜像层的文件结构，可以在不运行容器的情况下提取应用制品（Artifacts），为后续的依赖分析提供数据源。

### 2.1.3 容器网络接口标准

随着容器生态的发展，容器网络接口（Container Network Interface, CNI）成为连接容器运行时与网络插件的标准规范[55]。CNI 插件负责在容器创建时分配 IP 地址、配置网桥或虚拟网卡（veth pair），并负责路由表的维护。在云原生环境中，CNI 插件（如 Calico、Cilium）是实施微隔离策略的执行主体，它们通过拦截进出 Pod 的网络流量，依据预定义的策略进行放行或阻断。

## 2.2 Kubernetes 容器编排技术

Kubernetes（K8s）作为当前云原生生态的事实标准，源于 Google 内部的 Borg 集群管理系统[56]。它不仅是一个部署工具，更提供了一套用于构建分布式系统的声明式 API 和控制平面，为微服务的自动化治理提供了基础设施层面的支持。

### 2.2.1 核心架构与资源抽象

Kubernetes 采用典型的“控制平面-数据平面”解耦架构。控制平面（Control Plane）包含 API Server、Etcd、Scheduler 和 Controller Manager 等核心组件，负责维护集群的全局状态与调度决策；数据平面（Data Plane）则由运行在各工作节点上的 Kubelet 和 Kube-Proxy 组成，负责容器生命周期的具体执行与网络规则的维护。

在资源抽象层面，Kubernetes 引入了 Pod 作为最小调度单元，通过将紧密耦合的容器封装在共享的 Network 和 IPC 命名空间中，解决了微服务进程间的协同问题。Service 资源则提供了一组 Pod 的逻辑抽象与稳定的虚拟 IP（ClusterIP），利用标签选择器（Label Selector）屏蔽了后端 Pod IP 动态变化带来的寻址复杂性。这种基于标签的松耦合关联机制，是本文构建动态微隔离策略的核心依据。

### 2.2.2 网络策略模型

为了在扁平化的 Pod 网络中实现访问控制，Kubernetes 定义了 NetworkPolicy 资源对象。该对象本质上是一种以应用为中心的白名单防火墙规则，允许管理员通过 `podSelector` 和 `namespaceSelector` 精确界定流量边界，控制入站（Ingress）与出站（Egress）的通信权限[57]。

值得注意的是，NetworkPolicy 仅定义了策略规范，其实际执行依赖于底层的 CNI 插件（如 Calico, Cilium）。虽然基于 iptables 或 IPVS 的传统实现能够满足基本的 L3/L4 隔离需求，但随着集群规模的扩大，海量规则的查找与更新面临显著的性能瓶颈。此外，原生 NetworkPolicy 缺乏对应用层协议（L7）的感知能力，无法防御基于合法端口的逻辑攻击，这为引入代码级语义分析提供了必要性。

### 2.2.3 Informer 事件驱动机制

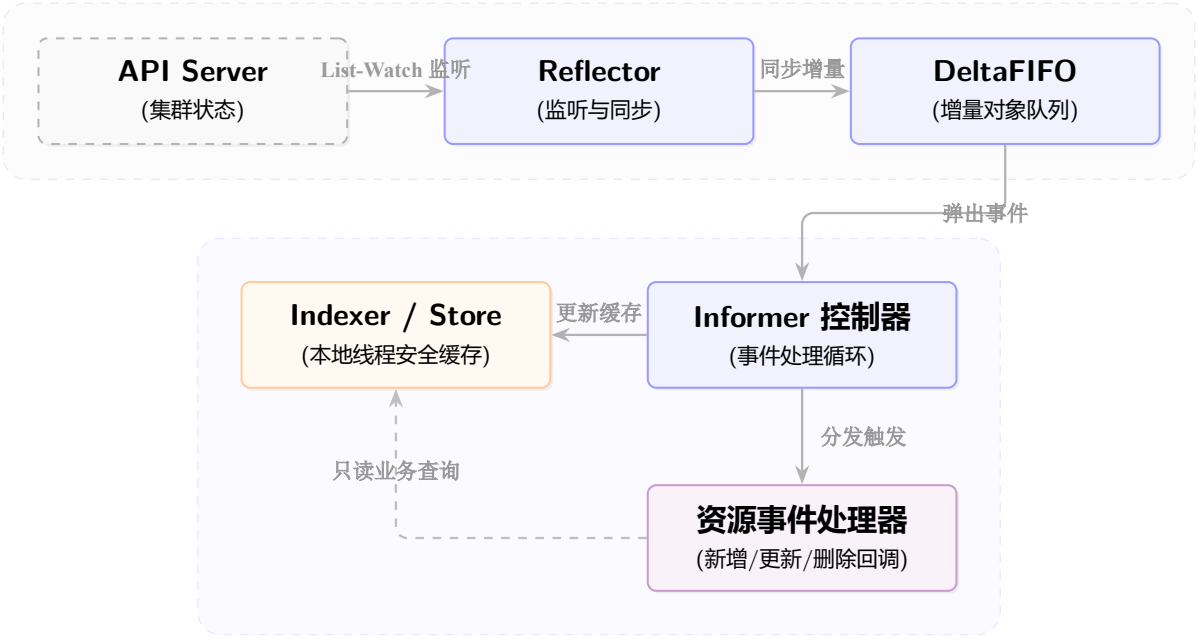


**图 2.1 Kubernetes Informer 机制组件协作与事件驱动流程示意图**

在零信任安全体系中，策略引擎需要实时感知工作负载的生灭与漂移。若采用传统的轮询（Polling）方式查询 API Server，不仅时效性差，且会给控制平面带来巨大的负载压力。Kubernetes 客户端库（client-go）为此提供了 Informer 机制，如图2.1所示，基于 List-Watch 接口实现了高效的事件驱动模型[58]。

Informer 的核心组件包括 Reflector 和 Indexer。Reflector 负责与 API Server 建立长连接，实时监听（Watch）资源变更事件，并将增量数据写入 DeltaFIFO 队列；Indexer 则在客户端本地维护一份线程安全的只读缓存，通过索引加速数据检索。本文提出的策略生成系统深度利用了 SharedInformer 机制，能够在毫秒级延迟内捕获 Pod 的创建、销毁及标签变更事件，从而触发微隔离策略的动态计算与下发，解决了静态策略无法适应云原生动态环境的难题。

## 2.3 Soot 静态分析技术

Soot 是 Java 语言分析领域最通用的静态分析与优化框架之一，最初由麦吉尔大学 Sable 研究组开发。它通过将 Java 字节码（Bytecode）转换为中间表示（Intermediate Representation），提供了一套用于分析和转换 Java 程序的 API。在云原生微服务场景下，Soot 能够直接处理编译后的 `.class` 文件或 JAR 包，无需源代码即可提取类结构、继承关系及成员变量定义，为构建服务间的依赖关系提供了底层技术支撑[59]。

### 2.3.1 Java 字节码与类文件解析

Java 字节码是运行在 Java 虚拟机（JVM）上的二进制指令集。根据 JVM 规范，类文件（Class File）存储了类的全限定名、父类引用、接口实现列表、字段表（Fields）及方法表（Methods）。Soot 框架首先通过 `SootResolver` 组件加载类路径下的所有类文件，将其解析为内存中的 `SootClass` 对象。这一过程能够完整保留类的元数据信息，包括修饰符（public/private）、注解（Annotations）以及泛型签名，是后续识别 AOP 切面标记或依赖注入注解的基础[60]。

### 2.3.2 Jimple 中间表示

为了克服原生字节码基于栈架构（Stack-based）难以分析的缺点，Soot 引入了 Jimple 中间表示。Jimple 是一种基于类型的三地址码（3-Address Code），它将复杂的栈操作指令转换为显式的赋值语句（如 `x = y + z`）。

Jimple 的核心优势在于简化了控制流分析。在 Jimple 中，所有的实例方法调用（Invokevirtual）和接口调用（Invokeinterface）都被显式保留，且变量的定义与使用关系

（Def-Use）清晰可见。这使得分析工具能够直接通过扫描 Jimple 语句（Statement）来识别代码中的方法调用行为，而无需模拟 JVM 的操作数栈行为[61]。

### 2.3.3 类层次与结构分析能力

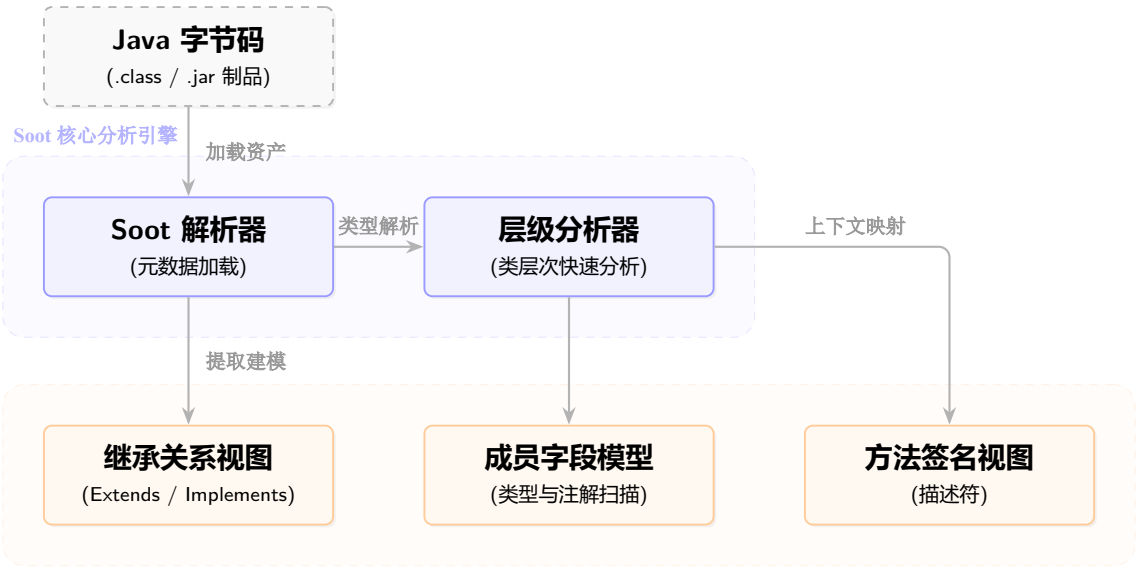


**图 2.2 Soot 框架类层次分析与程序结构提取流程示意图**

如图2.2所示，Soot 提供了强大的类层次分析（Class Hierarchy Analysis, CHA）与结构提取能力，这是构建类之间静态依赖关系的核心机制：

- 继承与实现关系分析：Soot 维护了一个全局的类层次结构树（Hierarchy），能够快速查询任意两个类是否存在继承（Extends）或接口实现（Implements）关系。这为识别基于多态的调用链路提供了必要的类型约束信息[62]。
- 成员变量与字段分析：通过 `SootField` 对象，Soot 允许分析人员遍历类中定义的所有成员变量及其类型。这在微服务分析中尤为重要，因为许多服务依赖是通过成员变量注入（如 `@Autowired` 或 `@Resource`）建立的。Soot 能够提取字段的声明类型，从而建立当前类与字段类型之间的关联边。
- 方法体与切面逻辑可见性：对于采用面向切面编程（AOP）的代码，编译后的字节码中往往包含由编译器织入（Weaving）的额外调用逻辑。Soot 对字节码的分析基于最终的编译产物，因此能够“看到”并解析这些由 AOP 框架自动生成的字节码指令，从而建立起切面类与目标类之间的关联。

综上所述，Soot 不仅是一个字节码转换工具，更提供了一个包含类结构、字段定义及方法签名的完整对象模型，为静态提取微服务内部复杂的类间依赖提供了完备的视图。

## 2.4 大语言模型技术

大语言模型（Large Language Model, LLM）是近年来人工智能领域的重要进展。在云原生安全领域，LLM 不仅需要理解单文件的代码逻辑，更需要具备跨文件的仓库级理解能力，以弥合“非结构化代码逻辑”与“结构化安全策略”之间的语义鸿沟[63]。

### 2.4.1 Transformer 架构与代码建模

现代大语言模型主要基于 Transformer 架构，其核心创新在于引入了自注意力（Self-Attention）机制。对于代码分析任务，Transformer 的优势在于能够捕捉长距离的依赖关系（Long-range Dependencies）。通过多头注意力（Multi-Head Attention）机制，模型能够从不同的语义子空间同时关注代码的语法结构、数据流向及控制流逻辑，从而在没有显式编译过程的情况下理解程序的业务意图[64]。

然而，通用 LLM 在处理代码任务时面临着显著的上下文窗口（Context Window）限制。微服务项目通常由数百个源文件组成，直接将整个仓库（Repository）输入模型会超出长度限制或导致“迷失中间（Lost-in-the-Middle）”现象，这使得模型难以捕捉跨文件的隐式依赖。

### 2.4.2 DeepWiki：仓库级依赖挖掘框架

为了解决大模型在全仓库理解上的瓶颈，DeepWiki 项目提出了一种基于 LLM 的仓库级依赖挖掘新范式。DeepWiki 的核心原理是将代码仓库视为一个巨大的知识库，通过“静态索引 + 动态检索”的方式实现对跨文件依赖的精准捕获[47]。

DeepWiki 的技术路径主要包含两个关键步骤：

1. 语义索引构建：利用静态分析工具提取代码中的实体（如类、方法、变量）及其拓扑关系（如继承、调用、引用），构建项目级的代码知识图谱（Code Knowledge

Graph）。同时，利用代码大模型生成每个实体的语义摘要（Summary），将其向量化后存入向量数据库。

2. 上下文感知检索：当需要分析某个微服务的对外调用时，DeepWiki 不再输入全部代码，而是根据当前分析的焦点（如一个 Controller 类），在知识图谱中游走检索其关联的依赖节点（如 Service 接口及其实现类）。

这种机制使得 LLM 能够在有限的上下文窗口内，获取到跨越多个文件的完整调用链路信息，从而准确识别出微服务间隐藏的远程调用逻辑。

### 2.4.3 思维链推理与结构化生成

在获取了完整的代码上下文后，利用思维链（Chain-of-Thought, CoT）技术可以进一步增强模型的推理鲁棒性。CoT 引导模型在生成最终的安全策略之前，显式地生成中间推理步骤（例如：“首先定位到 `OrderController`，发现其调用了 `PaymentService` 接口，经 DeepWiki 检索该接口实现类绑定了 `/api/pay` 路径”）。这种分步推理机制显著提高了策略生成的准确率，并缓解了深度学习模型的“黑盒”不可解释问题[48]。

## 2.5 本章小结

本章对支撑零信任微隔离策略生成系统的关键技术进行了梳理。首先，分析了容器与 Kubernetes 编排机制，指出原生网络策略在动态环境下的局限性，并确立了基于 Informer 的实时数据采集方案。其次，探讨了 Soot 静态分析框架，利用 Jimple 中间表示与类结构分析能力，为从字节码中提取确定的服务依赖提供了理论支撑。最后，介绍了大语言模型及 DeepWiki 框架，论证了其在跨文件依赖挖掘与策略语义增强方面的优势。上述技术共同构成了本文动静态协同分析框架的工程基础。

# 3 动静态分析的零信任网络策略生成系统

本章详细阐述动静态协同分析系统的总体架构与实现细节。针对云原生环境下微服务生命周期短、安全意图识别难等挑战，本系统设计了一套纵向贯穿基础设施到策略交付、横向融合动态运行时指纹与静态字节码逻辑的技术框架。


**图 3.1 动静态协同分析的零信任策略生成系统总体架构**

如图 3.1 所示，系统整体架构严格遵循论文的逻辑脉络，由以下四个核心层级组成：

1. 基础设施层 (Infrastructure Layer)：位于架构底部，是整个系统的数据源头。该层涵盖了处于运行态的 Kubernetes 集群以及待分析的 Java 字节码制品.

2. 动态感知与静态分析模块：承接基础设施层提供的数据流。

   - 动态感知模块：利用运行时探针 (Probe) 实时捕获集群事件，并执行动态指纹提取，以获取环境变量与注册中心配置。
   - 静态分析模块：通过三级过滤漏斗剔除冗余依赖，随后驱动 Soot 分析引擎对字节码进行调用提取与语义分析。

3. 微服务依赖图构建引擎 (Dependency Graph Builder)：作为系统的核心中枢，负责将动态感知的运行拓扑 $G_{runtime}$ 与静态分析得到的调用链 $G_{static}$ 进行深度融合。该引擎消除了动静态信息之间的“语义盲区”，构建出完整的服务依赖全景图。

4. 策略交付层 (Policy Layer)：处于架构顶端，基于融合后的依赖图执行确定性策略生成。通过标签映射与白名单合成，并结合策略冲突消解模块，最终输出可落地的 Kubernetes NetworkPolicy。

本章后续小节将围绕上述模块，依次论述各关键技术的具体设计思路与算法实现。

## 3.1 运行时数据采集

零信任微隔离策略的精确生成，依赖于对云原生集群运行状态的全面、真实刻画。仅依靠静态配置或离线代码分析，往往难以覆盖动态调度、弹性伸缩以及运行期配置变更等复杂场景，进而导致策略推导结果与实际运行环境存在偏差。为弥补这一不足，本文从运行时视角出发，构建了一套面向云原生环境的状态采集机制，为后续动静态联合分析提供可靠的环境上下文支撑。

针对上述需求，本文设计并实现了一个轻量级运行时采集探针（Collector），以独立服务的形式部署于 Kubernetes 集群内部。该探针不参与业务请求处理路径，仅通过标准接口感知集群运行态信息，其整体设计遵循“无侵入、低开销、强一致”的原则，确保在不影响业务稳定性的前提下，持续获取高可信度的运行数据。

运行时采集探针主要聚焦于以下四类关键信息，这些信息共同构成零信任策略生成所需的基础输入空间：

1. 编排元数据：采集 Pod、Service 等 Kubernetes 原生资源对象的生命周期状态、命名空间归属及标签信息，用于构建微服务之间的基础网络拓扑与身份映射关系；

2. 容器运行时信息：获取容器进程的环境变量、启动参数以及文件系统挂载结构，用以还原应用实例在运行阶段的真实配置上下文；

3. 应用资产（Artifacts）：提取运行态容器内实际加载的 Java 字节码制品（如 JAR、WAR 文件），作为后续静态依赖分析与调用关系解析的直接输入；

4. 服务注册信息：采集 Nacos、Eureka 等注册中心维护的服务实例与地址映射数据，用于解析微服务架构中逻辑服务名到物理网络地址之间的动态对应关系。

通过对上述多源运行态信息的统一采集与建模，系统能够在策略生成阶段获得与真实集群状态高度一致的环境视图，从而为后续的静态分析、合规性审计以及策略正向生成提供确定性的输入基础。

### 3.1.1 Kubernetes 资源

在云原生环境中，Kubernetes 作为事实上的容器编排标准，其资源对象承载了集群运行状态与服务组织结构的核心信息。Pod、Service、Deployment 等资源不仅描述了应用的部署形态，还隐含了服务实例之间的潜在通信边界与身份语义。因此，准确、持续地获取 Kubernetes 资源状态，是构建零信任网络策略的重要前提。

与传统静态配置文件不同，Kubernetes 资源具有高度动态性。服务实例可能因弹性伸缩、滚动更新或故障迁移而频繁变化，若仅在策略生成初期进行一次性资源快照采集，将难以反映真实的运行环境，进而导致生成的网络策略在时效性和准确性上存在明显不足。针对这一问题，本文从“持续一致性”的角度出发，对 Kubernetes 资源的采集机制进行了专门设计。

Kubernetes 集群中的资源对象处于高频变更状态，Pod 的扩缩容、故障迁移（Failover）以及 Service 定义的更新都会直接影响微隔离策略的有效性。为了实现对集群资源拓扑的毫秒级感知，本系统摒弃了高延迟的 API 轮询（Polling）模式，转而采用基于 Kubernetes 官方客户端库 client-go 的 SharedInformer 事件驱动机制[65]。

#### 3.1.1.1 基于 List-Watch 的增量同步机制

为了在保证数据完整性的同时降低采集开销，本文采用 Kubernetes 提供的 List-Watch 机制，实现对资源对象的增量同步。List-Watch 是 Kubernetes API Server 提供的一

种事件驱动式资源监听模型，其核心思想是通过初始全量查询（List）与后续事件订阅（Watch）的组合，持续感知资源状态的变化过程。

在系统启动阶段，采集探针首先通过 List 接口获取目标资源类型的完整快照，以建立本地的初始状态视图。该阶段确保系统在任意时间点接入集群时，均能够获得一致且完整的资源基线。随后，探针通过 Watch 接口订阅资源变更事件，包括资源的创建、更新与删除操作。每当集群中相关资源状态发生变化时，API Server 会以事件流的形式将变更信息推送至探针侧，从而避免了周期性轮询带来的额外开销。

相较于基于定时扫描的采集方式，List-Watch 机制具有以下优势：一方面，其事件驱动特性显著降低了不必要的 API 请求数量，减少了对控制面的压力；另一方面，资源状态的变化能够被近实时感知，有助于保持本地视图与集群实际状态之间的一致性。这种增量同步能力对于后续基于运行态事实生成网络策略尤为关键。

在具体实现中，本文通过为不同类型的 Kubernetes 资源分别维护独立的 Watch 通道，并结合资源版本号（ResourceVersion）进行一致性校验，从而在网络波动或探针重启等异常情况下，避免资源状态丢失或重复处理的问题。通过上述机制，系统能够在复杂运行环境下稳定、持续地获取 Kubernetes 资源变化信息。

#### 3.1.1.2 关键资源对象与采集维度

在完成资源同步机制设计的基础上，本文进一步明确了 Kubernetes 资源采集的对象范围与具体维度，以确保后续分析阶段具备充分且必要的上下文信息。针对零信任网络策略生成的需求，如表3.1，本文重点关注以下几类资源对象及其属性。

对于 Pod 资源，采集内容包括其所属 Namespace、Label 集合、节点调度信息以及容器规格等元数据。这些信息用于刻画服务实例的运行位置与逻辑身份，为后续 Pod 级访问控制策略的生成提供基础依据。特别地，Label 信息在 Kubernetes 中承担着逻辑分组与身份标识的双重角色，是实现细粒度策略匹配的重要锚点。

对于 Service 资源，系统重点采集其 Selector 定义、端口映射关系以及服务类型等属性，并同步引入对 Endpoint 资源的实时监听机制。通过解析 Selector 与 Pod Label 之间的静态对应关系，并结合 Endpoint 对象中实时更新的后端 Pod IP 列表，系统建立起逻辑服务到具体实例集合的动态映射模型。这种机制不仅明确了服务级与实例级的通信边界，更确保了当应用发生扩缩容或节点漂移时，系统能够即时感知后端具体地址的变

化，从而消除静态策略与动态环境之间的滞后性。

此外，Namespace 作为 Kubernetes 原生的隔离单元，其名称与关联策略信息同样被纳入采集范围。通过综合 Namespace 与 Label 两个维度，系统能够在保持策略表达灵活性的同时，避免因过度细化而导致的策略碎片化问题。

通过对上述关键资源对象及其多维属性的系统化采集，运行时数据采集模块，如图3.2，为后续的静态分析结果校验、访问意图推理以及零信任网络策略的正向生成奠定了统一且可靠的基础。

**表 3.1 Kubernetes 核心资源对象与策略生成维度的映射关系**

| 资源对象 | 关键字段 | 语义映射 | 策略生成用途 |
|---|---|---|---|
| Pod | `metadata.labels` | 工作负载身份标识 | `podSelector` |
| | `spec.nodeName` | 物理拓扑位置 | 辅助故障域分析 |
| Service | `spec.selector` | 逻辑服务分组 | 确定流量目标 |
| | `spec.clusterIP` | 虚拟网络入口 | VIP 地址解析 |
| EndpointSlice | `endpoints.addresses` | 实际后端 IP 列表 | 解析无头服务 (Headless) |
| ConfigMap | `data` (Key-Value) | 应用配置上下文 | 提取数据库/中间件地址 |

所有采集到的资源数据被封装为统一的 JSON 结构，并附带采集时间戳与集群标识（Cluster ID），通过消息队列异步推送至后端分析引擎，为构建全集群的服务依赖拓扑图提供元数据支撑。

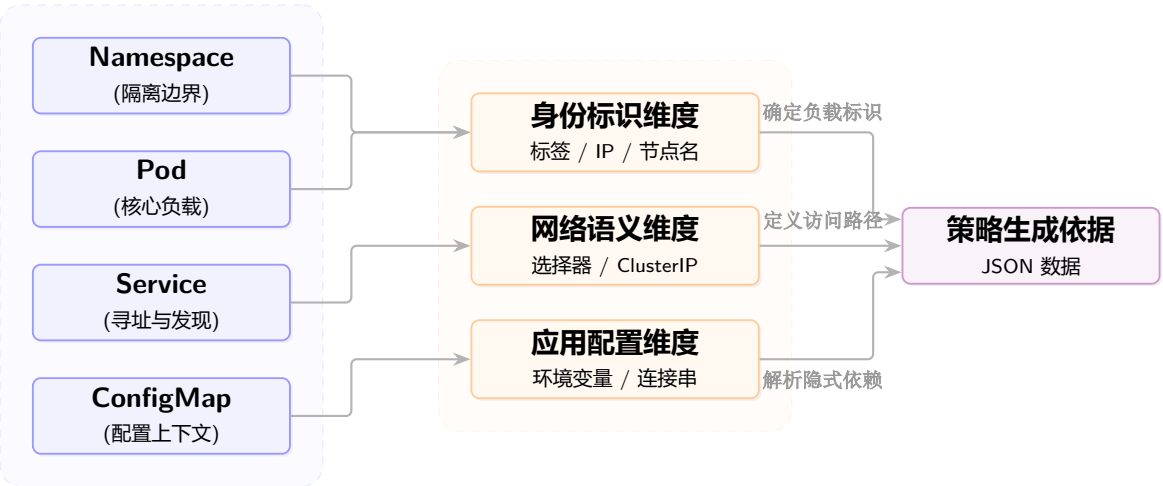


**图 3.2 Kubernetes 关键资源对象与策略采集维度映射示意图**

### 3.1.2 容器资源

容器作为微服务代码的运行时载体，其文件系统中往往冗余了大量与核心业务无关的基础设施组件（如 Shell 工具、包管理器、调试工具等）。若直接对静态镜像进行全量分析，不仅会引入巨大的计算开销，更会导致微隔离策略生成算法产生大量误报（例如误将调试工具的网络行为纳入业务白名单）。

借鉴基于运行时分析的沙箱挖掘思想[25]，本系统设计了一套“活跃资产锁定（Active Artifact Locking）”机制。该机制通过深度交互容器运行时接口（CRI）与 Linux 内核数据结构，精准识别并提取“积极参与业务逻辑”的进程与文件，从而收敛分析边界。

#### 3.1.2.1 基于 CRI 的容器定位与元数据映射

在 Kubernetes 体系中，容器的具体运行状态由底层容器运行时负责管理，而 Kubernetes 本身并不直接维护容器级别的详细信息。为实现对不同运行时实现的统一适配，Kubernetes 提供了容器运行时接口（Container Runtime Interface，CRI），用于规范编排系统与容器运行时之间的交互方式。

本文基于 CRI 接口实现对容器实例的统一定位与管理。通过 CRI 提供的接口，系统能够获取集群中运行容器的唯一标识、所属 Pod 信息以及对应的运行时元数据，从而建立 Kubernetes 资源对象与具体容器实例之间的映射关系。该映射关系是后续从容器层面提取运行态信息的基础。

在具体实现中，采集模块通过关联 Pod UID、容器名称与运行时返回的容器 ID，实现对容器实例的精确定位。同时，为应对集群中可能存在的多种容器运行时实现（如 containerd、CRI-O 等），系统通过 CRI 抽象接口屏蔽底层差异，确保采集逻辑在不同运行环境下具有一致性与可移植性。

通过上述机制，系统能够在不依赖特定运行时实现细节的前提下，稳定获取容器级运行态信息，为后续的配置解析与应用制品提取奠定基础。

#### 3.1.2.2 基于内存映射的活跃二进制集构建

为了剔除镜像中“存在但未运行”的僵尸文件，如图3.3，本系统利用 Linux 内核的内存管理机制来识别当前活跃的业务代码。具体而言，采集器通过 setns 系统调用进入

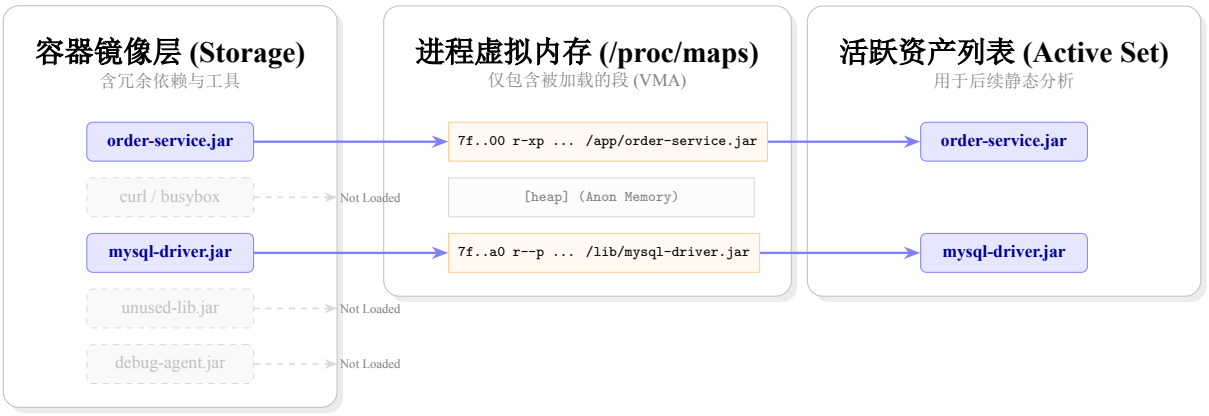


**图 3.3 基于内核内存映射的容器活跃资产锁定与提取原理**

目标容器的 PID Namespace，读取目标进程的 /proc/<PID>/maps 内存映射文件。

该文件记录了进程虚拟地址空间中加载的所有可执行文件与动态链接库（Shared Libraries）。系统通过解析这些内存段，构建出当前容器的“活跃二进制集（Active Binary Set）”：

1. 主执行体定位：识别映射具有 r-xp（可读可执行）权限且路径指向文件系统的主程序，过滤掉由 entrypoint.sh 或 tini 启动的父进程包装器；

2. 解释器透视：对于 Java 应用，系统识别出 JVM 进程（如 java 二进制文件），并进一步扫描其打开的文件描述符（/proc/<PID>/fd），从而精准定位到被加载到内存中的 JAR 包或类文件，而非扫描整个 /app 目录；

这种基于运行时内存视图的采集方法，能够从根本上排除“镜像内打包了 curl 但业务从未调用”带来的策略噪声，确保后续分析仅聚焦于真实的业务实体。

#### 3.1.2.3 配置上下文与分层文件系统解析

除应用代码本身外，运行期配置同样对服务行为与通信方式产生重要影响。环境变量、配置文件以及启动参数等信息，往往决定了服务对外暴露的端口、依赖的外部服务地址以及启用的功能模块。因此，在生成零信任网络策略时，有必要对容器的配置上下文进行系统化采集与解析。

本文结合容器分层文件系统的特性，对运行期配置进行有针对性的解析。采集器通过以只读方式访问读容器进程的 /proc/<PID>/environ 文件，无侵入地提取全量环境变量键值对。同时，针对容器镜像采用的联合文件系统（UnionFS, 如 OverlayFS），系统解析

/proc/mounts 信息，区分 LowerDir（只读镜像层）与 UpperDir （读写容器层）。这使得分析引擎能够优先处理运行时产生的文件修改（Copy-on-Write），不仅还原了应用启动时的完整配置上下文，也确保了对热更新（Hot-Swap）代码的实时感知[66]。

通过将配置上下文与前述活跃二进制集进行关联，系统能够在后续分析阶段更准确地理解代码行为与运行环境之间的关系，为策略生成提供更加完整的上下文支撑。

### 3.1.3 Java 字节码制品提取

在微服务架构中，服务间通信关系最终体现在应用代码的实际执行路径之中。相较于源码级分析，直接面向运行环境中加载的 Java 字节码制品进行分析，能够有效规避源码缺失、版本不一致以及构建环境差异等问题。因此，本文选择以运行态 Java 字节码制品作为静态分析阶段的主要输入。

需要注意的是，在容器化部署场景下，镜像中包含的字节码文件并不一定全部参与实际运行。一方面，部分依赖可能因配置条件限制而未被加载；另一方面，应用在运行过程中可能通过插件机制或动态加载方式引入额外制品。若不加区分地对文件系统进行全量提取，容易引入大量与实际执行和服务调用无关的分析对象，进而影响后续调用关系与通信语义建模的准确性。

基于上述考虑，本文从运行态视角出发，设计了一套面向 Java 应用的字节码制品提取方法，重点关注“实际被加载，参与执行，服务调用相关”的代码集合，为后续静态分析提供精简且可信的输入。

#### 3.1.3.1 基于内存的活跃依赖识别

在 Java 应用的实际运行过程中，并非所有被打包进镜像或声明在 ClassPath 中的依赖都会参与执行。JVM 采用按需加载机制，仅在代码路径真正触发时才加载对应的类，并将其字节码映射到进程的虚拟地址空间中。因此，从运行态的角度来看，进程当前的内存映射情况能够较为真实地反映应用在该实例中实际生效的代码依赖。

基于上述观察，本文引入了一种运行态依赖采集方式，直接对容器内 Java 进程的内存映射信息进行分析。具体做法是通过 Linux 提供的进程文件系统（procfs），读取目标进程的 /proc/<PID>/maps 文件。该文件记录了进程虚拟地址空间中已映射的所有文件对象，包括由 JVM 在运行过程中加载的各类 JAR 包。

在实现过程中，采集器对映射结果进行筛选，仅保留路径以.jar 结尾的映射项，并过滤掉明显与业务逻辑无关的系统库及 JVM 自身组件。与仅基于镜像内容或构建配置进行静态分析相比，这种方式能够避免将“理论上存在、但实际未使用”的依赖纳入分析范围，从而更加贴近当前实例的真实运行状态。

该方法在实践中表现出几项较为明显的优势。首先，对于磁盘中存在但未被加载的依赖（例如某些备用数据库驱动或测试相关库），内存映射中不会出现对应记录，可以自然地将这类冗余依赖排除。其次，在使用自定义 ClassLoader 或插件机制的场景下，部分 JAR 包往往被加载至非标准路径，单纯依赖固定目录扫描容易遗漏，而内存映射能够直接反映其真实加载位置。此外，通过解析映射条目的真实文件路径，还可以消除符号链接带来的路径歧义问题，这在容器环境中较为常见[25]。

总体来看，基于进程内存映射的依赖识别方式无需对应用代码或运行流程进行侵入式改造，即可获得较为准确的运行态依赖信息，为后续的依赖分析和策略生成提供了可靠的基础。

#### 3.1.3.2 无侵入式流式传输通道

为了将锁定的活跃 JAR 包提取至后端分析引擎，系统设计了基于 Kubernetes API 的流式传输通道。采集器不依赖 `docker cp` 等宿主机特权命令，而是与 API Server 建立 HTTP/2 长连接。

通过调用 SKBN，采集器在目标容器内执行轻量级的拷贝命令，将筛选出的活跃 JAR 文件列表打包为二进制流，并通过标准输出（Stdout）管道实时传输。该过程采用了“零落地（Fileless）”策略，数据流直接在内存中转发，避免了在业务容器内创建临时文件，从而消除了对业务磁盘 I/O 的干扰及潜在的残留风险[67]。该方式在控制单次资源占用的同时，降低了对容器 I/O 与内存的瞬时压力。

通过将制品提取与传输过程解耦，系统能够在不影响业务请求处理的前提下，完成字节码数据的采集，为后续分析阶段提供稳定输入。

#### 3.1.3.3 增量指纹比对与传输优化

在微服务集群中，容器实例可能因滚动更新或弹性伸缩而频繁创建和销毁，同一服务的多个副本（Replicas）共享完全相同的代码层，且不同服务之间往往存在公共的基

础架构组件（如 Spring Cloud 全家桶）。若在每次实例启动后均对全部字节码制品进行完整传输，将带来不必要的网络与存储开销。为此，本文在字节码制品传输阶段引入增量指纹比对机制，以减少重复数据的传输。

具体而言，系统引入了基于 SHA-256 的增量指纹比对机制。在传输前，采集器在容器内计算目标文件的哈希摘要（Digest），并利用布隆过滤器（Bloom Filter）快速判断该哈希值是否已存在于分析引擎的制品库中。仅当检测到全新的哈希值（对应版本更新或新引入的依赖）时，才触发实际的数据传输。

该增量优化机制在不影响分析准确性的前提下，有效降低了数据传输规模，使字节码制品提取过程能够更好地适应云原生环境中实例频繁变动的运行特征。

### 3.1.4 注册中心配置

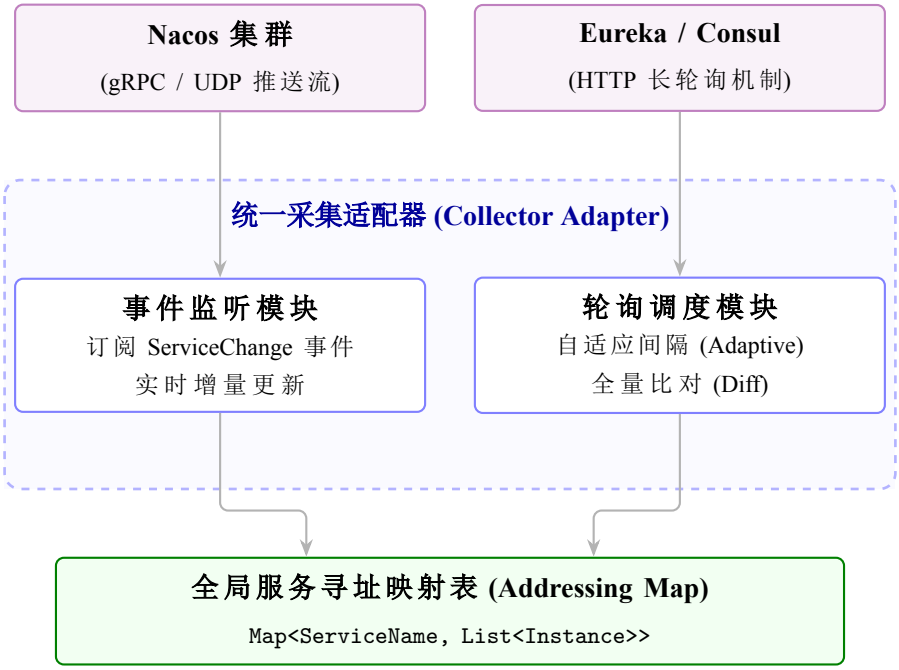


**图 3.4 异构注册中心数据的统一采集与映射机制**

在基于微服务的云原生系统中，服务实例的网络可达性有时并非通过静态配置直接确定，而是由注册中心在运行期间动态维护。应用在启动或状态变更时，可以将自身实例信息注册至注册中心；调用方则通过逻辑服务名完成服务发现。由此，注册中心成为连接应用代码与实际网络通信路径的重要中介，其配置状态直接影响服务之间的可达关系。

如果仅依据 Kubernetes 层面的 Service 或 Pod 信息进行分析，往往难以准确刻画逻辑服务名与真实网络地址之间的映射关系，尤其是在多注册中心并存或注册信息频繁变更的场景下。静态代码分析虽然能够提取出微服务间的逻辑调用关系（例如代码中硬编码的”http://payment-service/pay”），但无法直接推导出 Kubernetes NetworkPolicy 所需的确切网络五元组（IP Block 或 Pod Selector）。为了建立“逻辑服务标识”与“物理网络实体”之间的映射视图，如图3.4, 本系统设计了针对服务注册中心（Service Registry）的实时采集模块[8]，以弥补编排层信息在服务发现语义上的不足。

#### 3.1.4.1 异构注册中心适配器设计

在实际工程环境中，微服务系统可能同时采用多种注册中心实现，例如 Nacos、Eureka 或基于自研组件的服务发现机制。不同注册中心在接口形式、数据模型以及一致性语义等方面均存在差异，若针对每一种实现单独设计采集逻辑，将显著增加系统复杂度与维护成本。

因此，采集器采用了适配器模式（Adapter Pattern）构建统一的接入层，适配器对外提供一致的数据访问模型，对内负责将具体注册中心返回的信息转换为标准化表示。该层屏蔽了底层异构协议（如 Nacos 的 gRPC/UDP 推送、Eureka 的 HTTP 长轮询）的差异，向上层分析引擎提供标准化的 ServiceInstance 模型。在适配器实现过程中，系统重点关注服务实例的逻辑服务名、实例地址、端口信息以及健康状态等关键字段。这些字段直接决定了服务发现的结果，也是后续解析服务间实际通信路径所需的核心输入。

系统目前已内置支持云原生领域主流的服务发现协议：

- Nacos：作为阿里巴巴开源的云原生服务发现平台，Nacos 支持基于 UDP 的服务变更推送。采集器通过模拟 Nacos Client 行为，订阅指定命名空间下的服务列表变更事件，能够实现毫秒级的实例上下线感知。

- Eureka/Consul：针对基于 HTTP 的注册中心，采集器实现了自适应轮询机制（Adaptive Polling）。通过分析服务变动的历史频率，动态调整拉取间隔（例如在发布窗口期加密轮询，在稳定期降低频率），在保证数据新鲜度的同时降低控制面负载[68]。

#### 3.1.4.2 逻辑服务名到物理地址的动态映射

在注册中心机制下，服务调用通常以逻辑服务名作为入口，具体请求在运行时被路由至一个或多个实际服务实例。因此，从零信任网络策略的角度来看，仅限制逻辑层面的服务访问并不足以形成可执行的网络控制规则，还需要进一步解析其对应的物理网络地址范围。

本文通过对注册中心中维护的实例信息进行解析，构建逻辑服务名与物理地址之间的动态映射关系。该映射关系能够随着服务实例的注册、下线或状态变更而实时更新，从而反映集群运行阶段的真实通信可能性。

采集器将从各注册中心获取的数据聚合为一张全局的“服务寻址映射表（Service Addressing Map）”。该表维护了 Service Name→ List<Instance Address, Port, Metadata> 的实时对应关系，其中 Address 既可以表现为 IP 地址，也可以表现为 Pod Selector。

在后续的策略生成阶段，该映射表发挥着关键的“桥梁”作用：例如当静态分析引擎识别到 Java 代码中发起对 user-service 的 OpenFeign 调用时，系统查询该表即可锁定目标服务在当前集群中所有活跃实例的地址集合。

## 3.2 字节码静态分析

在完成运行时数据采集后，系统已经获得了与实际部署环境高度一致的应用制品集合。然而，仅依赖运行态信息仍不足以全面刻画服务之间的潜在通信关系。一方面，部分调用路径可能因条件分支或配置差异在当前运行实例中尚未触发；另一方面，零信任网络策略的生成需要对“可能发生的访问行为”进行约束，而不仅限于已观测到的通信事实。

基于上述考虑，本文引入字节码层面的静态分析方法，对 Java 应用的调用结构与通信语义进行系统建模。相较于源码分析，字节码分析不依赖于源码可得性，且能够直接反映编译后的真实执行逻辑，更适用于云原生环境中多版本、多构建产物并存的实际情况。

静态分析模块以 3.1 节提取的活跃字节码制品为输入，围绕方法调用关系、控制流与数据流传播等关键要素展开分析，旨在从代码层面还原服务对外通信的潜在意图，为后续的合规性审计与策略正向生成提供结构化依据。然而，在实际工程环境中，微服务

应用通常会引入数量庞大的第三方依赖库，其规模往往远超业务逻辑代码本身。若直接对全量字节码构建调用关系，不仅会带来显著的计算开销，还容易引入大量与业务语义无关的调用路径，例如日志框架或通用工具库内部的网络行为，从而干扰后续依赖关系的判定。除了第三方依赖库，在剩下的业务 JAR 包中，仍然存在大量不直接参与远程调用的类，例如仅用于数据封装的 POJO、通用工具类或配置类。这类代码虽然在程序运行中不可或缺，但其本身并不承载服务间通信语义。

为此，本节设计了一个“三级过滤漏斗”模型，依次执行第三方与非业务类剔除、自定义类过滤以及核心调用链提取，确保静态分析仅聚焦于能够产生服务间交互的关键代码片段。

### 3.2.1 第三方依赖包过滤

在实际微服务应用中，业务代码通常依赖大量第三方库以实现基础功能支持，例如日志记录、序列化、配置管理及网络通信封装等。从零信任网络策略生成的角度来看，第三方依赖几乎不具备分析价值。多数通用依赖库内部虽可能包含网络相关实现，但其行为通常由业务代码间接触发，且不直接反映服务间的逻辑依赖关系。因此，本文在字节码静态分析阶段引入第三方依赖包过滤机制，将分析重点聚焦于更可能承载业务通信语义的代码范围。

为了解决第三方库带来的 Java 应用普遍存在的代码膨胀（Bloatware）问题，系统首先在 JAR 包粒度进行粗筛。该过程基于软件成分分析（SCA）思想，构建了一个包含 Maven Central 主流组件的“指纹黑名单库”[69]。

如图3.5，具体算法流程如下：

1. 指纹计算：对输入的 JAR 包计算 SHA-256 哈希值；
2. 黑名单匹配：将哈希值与指纹库进行 $O(1)$ 匹配。指纹库预置了 Spring Boot Starter、Netty、Jackson 等基础设施组件的哈希特征。
3. 包名启发式过滤：对于未命中的 JAR 包，进一步检查其包名（Package Name）。若包名匹配 org.apache.*、org.springframework.* 或 io.kubernetes.* 等标准前缀，且不包含特定业务关键词，则将其标记为非业务依赖进行剔除。

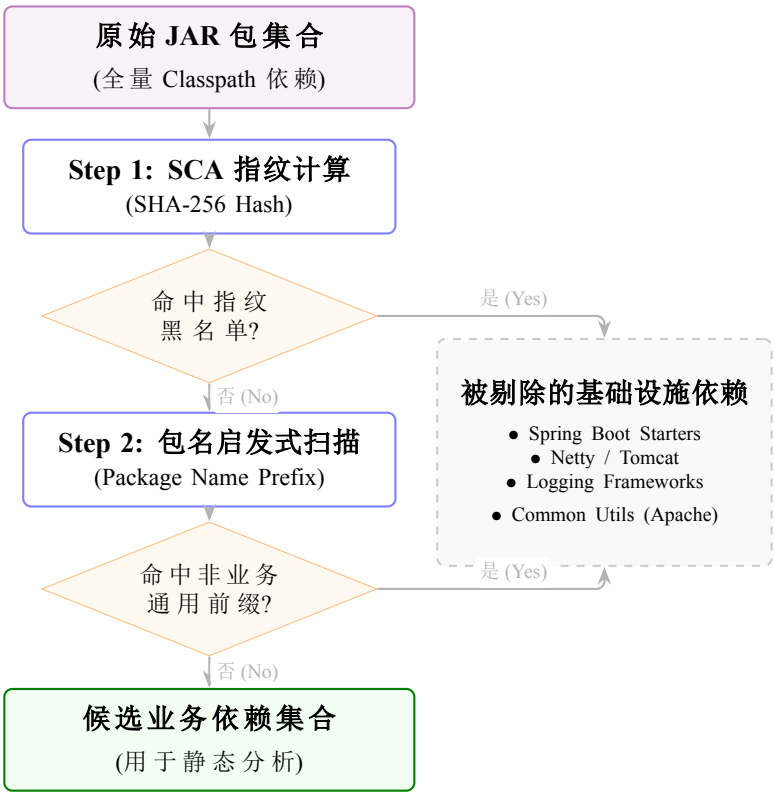


**图 3.5 基于指纹与启发式规则的第三方依赖过滤流程**

通过该步骤，分析引擎能够过滤掉基础设施代码，显著降低了后续 Soot 分析的内存压力。

### 3.2.2 自定义类的过滤

在完成第三方依赖包过滤后，剩余待分析的字节码主要由业务自身定义的类构成。然而，即便在业务代码范围内，仍然存在大量不直接参与服务通信的类，例如数据传输对象、配置封装类以及通用工具类等。若对这类自定义类进行同等粒度的静态分析，同样会引入冗余调用路径，影响分析结果的清晰度。

针对上述问题，本文在静态分析流程中进一步引入自定义类的过滤策略，以在不牺牲通信语义完整性的前提下，减少无关类对分析过程的干扰。该策略并不依赖完整的控制流或数据流分析，而是基于字节码层面的轻量级特征进行快速判断，适用于分析初期的规模裁剪阶段。

具体而言，系统通过检查类的结构特征与常量池信息。系统仅保留满足以下特征之一的“活跃类”：

- 入口特征：包含 @Controller、@RestController 或 @Scheduled 等注解的类，这通常是服务调用的起点；
- 出口特征：包含 @FeignClient、@DubboReference 等注解的接口或字段，这代表了明确的远程调用意图；
- 行为特征：常量池中包含”http://”、”https://” 或常见服务名前缀字符串的类，提示可能存在编程式的网络请求。

需要指出的是，启发式过滤策略本身并不保证绝对精确，其设计目标在于以较低成本显著缩小分析范围。因此，本文在过滤过程中采取保守策略，即在无法明确判断类是否参与通信行为时，优先将其保留进入后续分析阶段，以避免因过度过滤而遗漏潜在通信路径。

通过引入自定义类的启发式过滤机制，系统在分析效率与语义覆盖之间取得了较为平衡的效果，为后续调用链构建与服务依赖关系推断提供了更加聚焦的分析输入。

### 3.2.3 基于类的调用信息提取

在完成第三方依赖包与自定义类的多级过滤后，字节码静态分析的对象范围已被显著收敛。此时，分析重点由“缩小分析规模”转向“提取有意义的调用关系”，以刻画业务代码中可能存在的服务间通信路径。考虑到方法级调用分析在复杂工程环境中易产生规模膨胀问题，本文选择以类为基本分析粒度，对调用信息进行抽象建模。

基于类的调用信息提取旨在识别不同类之间的调用依赖关系，而不深入展开类内部的具体方法实现细节。通过这种方式，系统能够在保持调用关系整体结构的同时，有效控制分析复杂度，避免在早期阶段引入过多细粒度信息。

在具体实现过程中，系统通过解析字节码中的方法调用指令，统计类之间的调用关系，并据此构建类级调用依赖集合。本节针对云原生环境中最常见的两种调用模式，设计了专用的提取器（Extractor）。

#### 3.2.3.1 声明式调用提取

对于基于接口代理的声明式调用，分析引擎重点解析接口上的元数据注解。以 OpenFeign 为例，系统扫描所有带有 @FeignClient 注解的接口，提取 name 或 value 属性作为

目标服务标识，提取 path 属性作为基础路径。同时，解析接口方法上的 @RequestMapping 注解，获取具体的 HTTP 动词与子路径。这些信息组合后，构成了确定的 <Caller, CalleeService, Path> 三元组。

#### 3.2.3.2 编程式调用提取

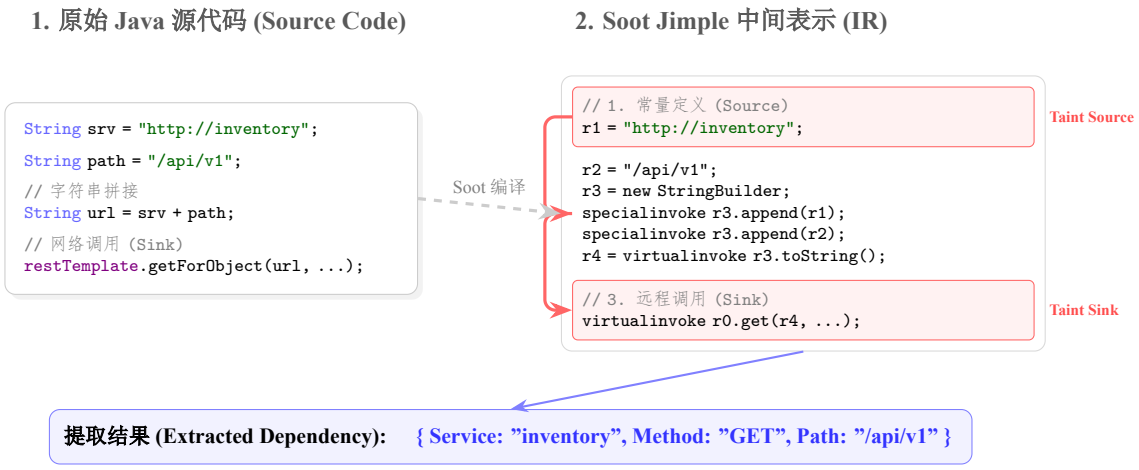


**图 3.6 静态污点分析与调用提取示例**

对于直接使用 HTTP 客户端的编程式调用，如图3.6，系统采用基于 Jimple 的静态污点分析（Static Taint Analysis）技术。

1. Source 识别：将 URL 字符串常量或配置项标记为污点源；
2. Sink 识别：将 `RestTemplate.getForObject()` 等网络请求方法标记为污点汇聚点；
3. 数据流追踪：利用 Soot 的 `SmartLocalDefs` 分析器，逆向追踪网络请求方法的 URL 参数定义点。如果 URL 是由字符串拼接而成（如 `"http://" + serviceName + "/api"`），系统通过常量传播分析尝试还原其静态部分，并将动态部分标记为需在运行时解析的变量[70]。

这一机制使得系统能够从非结构化的代码中还原出隐含的服务依赖关系。

## 3.3 调用关系图构建

在完成类级调用信息提取后，静态分析阶段已经获得了一组离散的调用依赖关系。然而，这些依赖关系在形式上仍然是相互独立的调用对，尚不足以反映业务逻辑中更高

层次的调用结构。为进一步刻画服务内部及服务之间可能存在的调用路径，有必要对上述调用信息进行系统化组织，构建统一的调用关系图。

调用关系图以类作为基本节点，以类之间的调用依赖作为有向边，用于描述程序中可能存在的调用传播路径。通过将离散的调用信息整合为图结构，系统能够从整体视角分析调用关系的连通性与层次性，为后续识别完整调用链提供基础支撑。相较于线性调用列表，图结构在表达复杂依赖关系时更具灵活性，也更适合进行后续的聚合与裁剪操作。

为了推导微服务间的全局依赖拓扑，本节提出了一种两阶段图构建算法：首先构建细粒度的类级调用图，以此为基础，结合微服务部署元数据，聚合生成粗粒度的微服务依赖图。

### 3.3.1 基于类的调用关系图

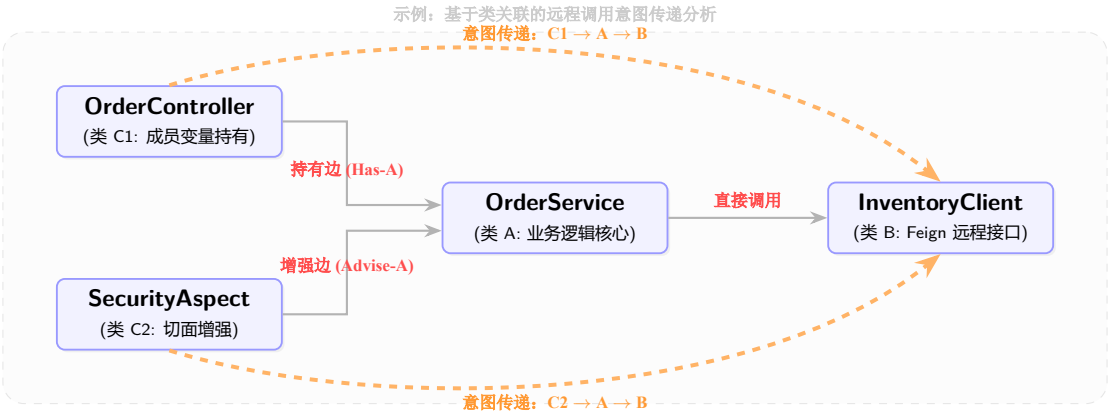


**图 3.7 复杂场景下类级调用关系图的关联传递示例**

在 Spring Cloud 等现代微服务框架中，服务间的调用往往被封装在复杂的对象关系之中。为了还原这些隐式联系，分析引擎构建了一个有向图 $G_{class} = (V_c, E_c)$，其中节点 $V_c$ 代表业务类，边 $E_c$ 代表类与类之间的关联关系。

不同于传统仅基于函数调用的构图方法，本文针对云原生应用的开发范式，如图3.7,设计了以下三种特定类型的边生成规则，以解决依赖注入（DI）与面向切面编程（AOP）带来的断链问题：

3.3.1.1 基于成员变量注入的关联边

在 Spring 容器中，服务类通常通过 `@Autowired` 或 `@Resource` 注解将其他组件（如 `FeignClient` 接口）注入为成员变量。Soot 的类结构分析能力使得系统能够扫描所有字段定义。若类 $A$ 包含一个类型为 $B$ 的成员变量，且该变量带有注入注解，则建立一条 $A \to B$ 的“持有边（Has-A）”。这种边揭示了潜在的调用能力，即使代码中未显式出现 `new` 关键字[71]。

3.3.1.2 基于继承与实现的泛化边

为了处理多态调用，系统利用 Soot 的类层次分析（CHA）算法处理继承关系。若类 $A$ 继承自父类 $B$ 或实现了接口 $I$，则建立 $A \to B$ 或 $A \to I$ 的**“泛化边（Is-A）”**。在后续的路径搜索中，若遇到针对接口 $I$ 的调用，算法将自动扩展至其所有具体实现类 $A$，从而覆盖多态场景下的动态分派路径。

3.3.1.3 基于切面的增强边

微服务架构广泛使用 AOP 技术实现熔断、限流或日志记录。切面类（Aspect）通常通过 `@Before`、`@Around` 等注解绑定到特定的目标方法上，导致控制流在运行时发生隐式跳转。系统通过解析切面表达式（Pointcut Expression），识别出切面类 $A$ 与被增强类 $B$ 之间的绑定关系，建立 $A \to B$ 的“增强边（Advise-A）”。这确保了在分析调用链时，能够正确穿越由中间件引入的代理逻辑。

### 3.3.2 基于微服务的调用关系图

类级调用图虽然精确，但其粒度过细，无法直接用于 Kubernetes NetworkPolicy 的生成。为此，如图3.8系统执行图聚合操作，构建以微服务为节点的有向图 $G_{service} = (V_s, E_s)$。

3.3.2.1 节点映射与聚合

首先，利用 3.1 节采集的 Kubernetes Pod 标签信息，建立“类 $\to$ 微服务”的归属映射函数 $M(c) = s$。对于 $G_{class}$ 中的每一个节点 $c \in V_c$，将其映射到对应的微服务标识 $s$。所有映射到同一 $s$ 的类节点被合并为一个微服务节点 $v_s \in V_s$。

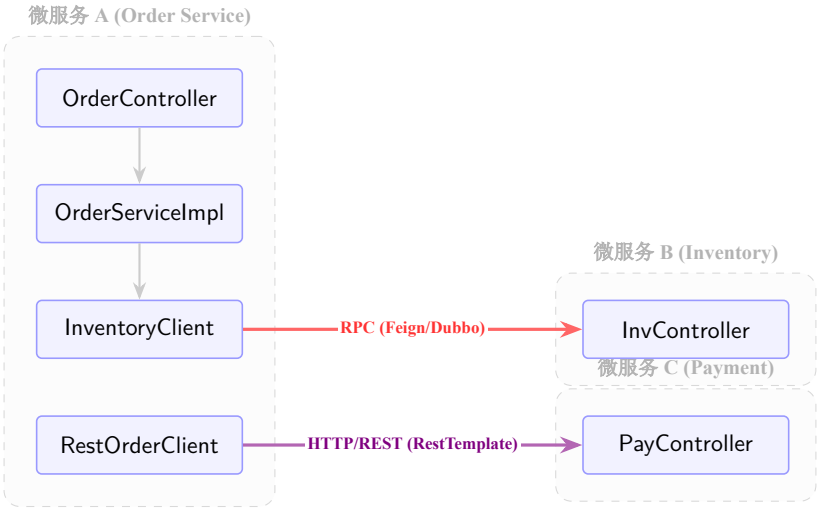


**图 3.8 基于类级关联聚合的微服务调用关系图构建示例**

3.3.2.2 跨服务边的解析与生成

遍历 $G_{class}$ 中的所有边 $(u, v)$，若 $M(u) \neq M(v)$，则表明存在潜在的跨服务调用。此时，分析引擎需结合 3.1.4 节采集的注册中心数据进行地址解析：

- 声明式调用解析：若节点 $v$ 是一个被标记为 `@FeignClient("order-service")` 的接口，系统查询服务寻址映射表，找到 `order-service` 对应的微服务节点 $v_{target}$，并在 $M(u)$ 与 $v_{target}$ 之间添加一条有向边。
- 编程式调用解析：若节点 $u$ 中明确包含了指向 `"http://inventory-service"` 的 `RestTemplate` 调用（经 3.2.3 节污点分析提取），系统同样通过服务发现机制解析目标服务，并在图中建立依赖关系。

最终生成的 $G_{service}$ 清晰描述了集群内所有微服务之间的“谁调用谁”的拓扑结构，且每一条边都附带了具体的调用端口与协议元数据，为后续生成最小权限的 ACL 规则提供了完备的决策依据[10]。

## 3.4 网络策略生成

微服务依赖图 $G_{service}$ 描述了系统内部合法的通信链路。为了将这些抽象的拓扑关系转化为 Kubernetes 集群可执行的访问控制规则，本节设计了一套自动化的策略生成算

法。该算法遵循零信任架构的“最小权限（Least Privilege）”原则，通过标签映射、白名单合成及冲突消解三个步骤，生成确定性的 NetworkPolicy 资源对象。

### 3.4.1 Pod 标签与服务拓扑映射机制

Kubernetes NetworkPolicy 的核心机制是基于标签选择器（Label Selector）来界定策略的作用域（Target）与对端（Peer）。因此，策略生成的首要任务是将 $G_{service}$ 中的节点 $v_s$ 映射回集群运行时的 Pod 标签集合。

利用 3.1 节采集的 Kubernetes 编排元数据，系统建立了一个双向索引表：

$$Map : ServiceName \rightarrow \{\langle Key, Value\rangle\}_{labels} \tag{3-1}$$

映射逻辑如下：

1. 服务发现映射：对于通过 Service 暴露的微服务，分析系统会解析 Service 定义中的 `spec.selector` 字段，将其作为该服务对应 Pod 的标识标签。

2. 无头服务映射：对于 Headless Service 或未定义 Selector 的外部服务，系统通过解析 EndpointSlice 资源，获取后端 Pod 的实际 IP，若无法关联标签，则生成基于 `ipBlock` 的 CIDR 规则。

3. 命名空间隔离：在映射过程中，严格保留命名空间（Namespace）上下文。若依赖图中存在跨命名空间的调用边，系统将在生成的规则中自动注入 `namespaceSelector`，以确保多租户环境下的隔离性[72]。

### 3.4.2 最小化白名单生成

基于“永不信任，始终验证”的零信任理念，本系统采用“默认拒绝 + 显式允许”的策略生成模式。对于依赖图中的每一个微服务节点 $v$，如图3.9, 生成策略流程如下：

#### 3.4.2.1 基线策略构建

首先为目标工作负载生成一条“隔离基线（Isolation Baseline）”策略。该策略将 `policyTypes` 设置为 `["Ingress", "Egress"]`，但不包含任何允许规则。一旦应用该策略，Kubernetes CNI 插件将立即阻断进出该 Pod 的所有非授权流量，从而收敛攻击面。

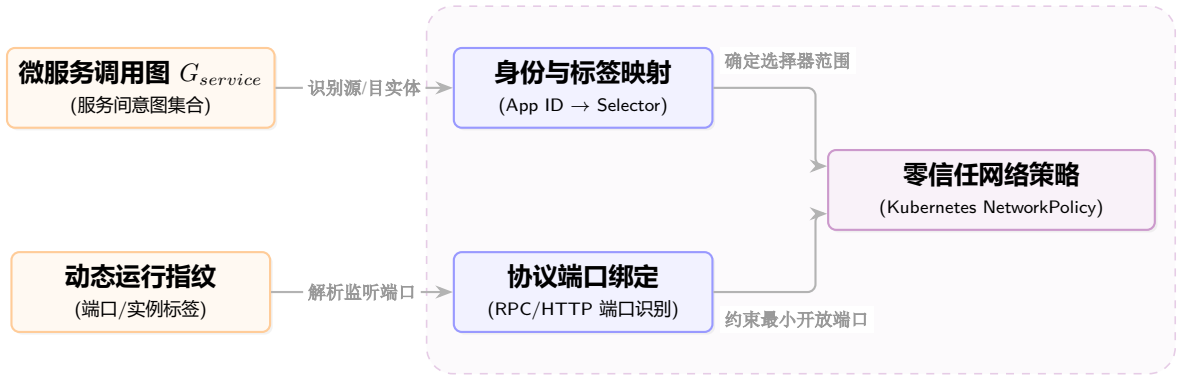


**图 3.9 最小化白名单策略生成流程示意图**

#### 3.4.2.2 白名单规则合成

遍历图 $G_{service}$ 中与节点 $v$ 相关联的所有边，生成细粒度的放行规则：

- 入站规则（Ingress）：对于所有指向 $v$ 的入边 $(u, v) \in E_s$，在 $v$ 的策略中添加一条 Ingress 规则，将来源设置为 $u$ 的标签选择器（`from: podSelector`），并严格限定目标端口（`ports`）为 $v$ 的监听端口。
- 出站规则（Egress）：对于所有由 $v$ 发出的出边 $(v, w) \in E_s$，在 $v$ 的策略中添加一条 Egress 规则，将目标设置为 $w$ 的标签选择器或 IP 段（`to: podSelector/ipBlock`）。
- 基础设施工单：自动注入对集群 DNS 服务（UDP 53 端口）的放行规则，保障微服务的基础解析能力。

### 3.4.3 策略冗余消除与冲突检测

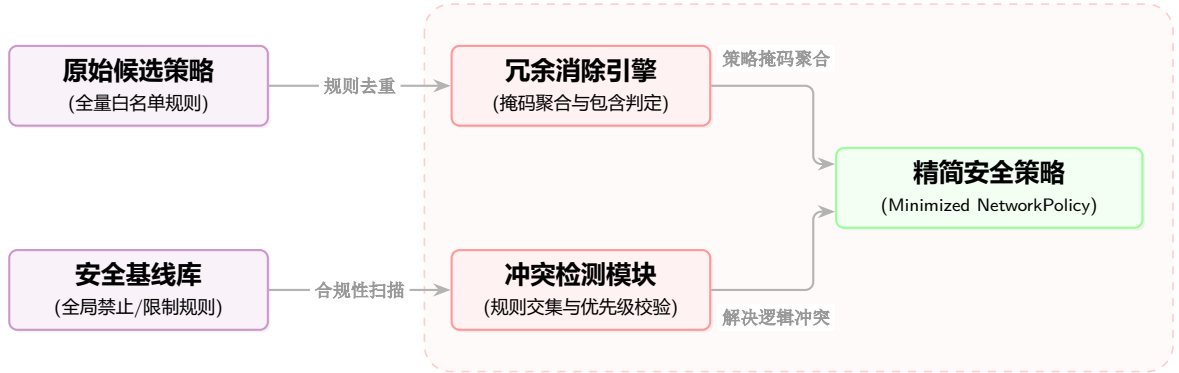


**图 3.10 策略冗余消除与冲突检测流程示意图**

在大规模集群中，自动化生成的策略可能与运维人员手动配置的规则产生重叠或冲突。为了保证策略集的有效性与性能，如图3.10所示，系统在下发前执行基于集合论的优化算法。

#### 3.4.3.1 CIDR 聚合与冗余消除

对于基于 IP 地址的 `ipBlock` 规则，系统采用区间合并算法（Interval Merging）对重叠的 CIDR 网段进行聚合。例如，规则 $R_1$：192.168.1.0/25 与 $R_2$：192.168.1.128/25 将被合并为 192.168.1.0/24。这不仅减少了底层 iptables/eBPF 的条目数量，也降低了策略匹配的计算开销。

**表 3.2 微隔离策略冲突类型定义与系统消解动作**

| 冲突类型 | 逻辑定义 (Description) | 系统消解动作 (Action) |
| --- | --- | --- |
| 冗余冲突 (Redundancy) | 新生成的规则 $R_{new}$ 是现有规则 $R_{exist}$ 的真子集 ($R_{new} \subset R_{exist}$)，且动作一致。 | **丢弃**：保留现有宽泛规则，减少规则数量。 |
| 遮蔽冲突 (Shadowing) | 现有规则 $R_{exist}$ 优先级更高且范围覆盖 $R_{new}$，但动作相反（如 $R_{exist}$ 为 Deny）。 | **报警**：提示策略无效，需人工介入调整优先级。 |
| 相关冲突 (Correlation) | $R_{new}$ 与 $R_{exist}$ 作用域部分重叠，且动作不同，导致重叠区域行为不确定。 | **拆分**：将重叠区域提取为独立的高优先级规则。 |
| 网段聚合 (Aggregation) | 多个 CIDR 规则连续（如 /24 相邻网段）且动作一致。 | **合并**：聚合为更大的 CIDR 块 (Supernetting)。 |

#### 3.4.3.2 逻辑冲突检测

系统定义了策略间的遮蔽（Shadowing）与泛化（Generalization）关系。在生成新策略前，系统会与集群现存策略进行集合包含关系检查：

$$Conflict(R_{new}, R_{exist}) \iff (Action_{new} \neq Action_{exist}) \wedge (Scope_{new} \cap Scope_{exist} \neq \emptyset) \tag{3-2}$$

若检测到新生成的“拒绝”规则被现有的“宽泛允许”规则（如 `allow-all`）所遮蔽，系统将生成报告，提示存在潜在的安全敞口，而非盲目覆盖，从而确保了生产环境的稳定性[73]。。

## 3.5 本章小结

本章构建了基于动静态协同分析的零信任微隔离策略生成底座。首先，设计了基于 SharedInformer 与 CRI 的运行时探针，实现了对 Kubernetes 元数据、容器进程上下文及业务字节码的无侵入采集；其次，利用指纹去重与 Soot 静态分析框架，精准提取了微服务间的远程调用链路，并结合依赖注入与切面特征构建了服务级依赖拓扑；最后，遵循零信任最小权限原则，建立了从服务拓扑到 NetworkPolicy 的自动化映射与冲突消解机制，为系统提供了确定性的安全规则保障。

# 4 基于大模型代码分析的零信任微隔离策略双向智能生成

本章提出了一种基于大语言模型（LLM）与 DeepWiki 代码知识库的双向智能生成框架。该框架旨在引入外部语义知识，弥合底层基础设施与上层业务逻辑之间的鸿沟。

系统的总体架构如图 4.1 所示，核心逻辑包含以下三个层次：

1. 多源信息关联层 (Context Layer)：针对云原生环境下容器运行时与静态代码割裂的痛点，本层并不重复底层的全量采集工作，而是聚焦于“关联”。通过引入 LLM 仓库推断 (Repo Inference) 模块，系统利用运行时捕获的环境变量、启动指令等指纹特征，智能推导并锁定目标容器对应的源代码仓库地址，从而建立起“运行时实例 ↔ 静态代码库”的语义索引，为后续分析提供物理入口。

2. 语义推理层 (Semantic Reasoning Layer)：该层是本章系统的智能中枢，由两大核心组件构成。DeepWiki 引擎负责对代码仓库进行深度扫描，挖掘跨文件的隐式依赖与配置文件路径，构建代码知识图谱；LLM 推理代理 (Reasoning Agent) 则基于 DeepWiki 提供的知识上下文，利用思维链 (Chain-of-Thought, CoT) 技术执行复杂的逻辑推演，负责构建合规性证据链与意图识别。

3. 双向生成与应用层 (Application Layer)：基于推理层的语义理解能力，系统向上提供两种截然不同的工作模式，以覆盖微隔离的全生命周期需求：

    - 逆向解释 (Inverse Explanation)：面向存量系统的审计场景。系统接收现有的 NetworkPolicy，回溯代码层面的业务证据，生成包含因果归因的可解释性审计报告，解决“策略为何存在”的可解释性难题。
    - 正向生成 (Forward Generation)：面向新上线服务的防护场景。针对无历史流量的新服务，系统直接从代码逻辑中提取通信意图，生成默认安全的微隔离策略，解决“冷启动”阶段的防护难题。

通过这一架构，系统不仅弥补了确定性分析的语义盲区，更通过双向闭环机制，实现了从“被动防御”到“意图驱动”的智能化升级。

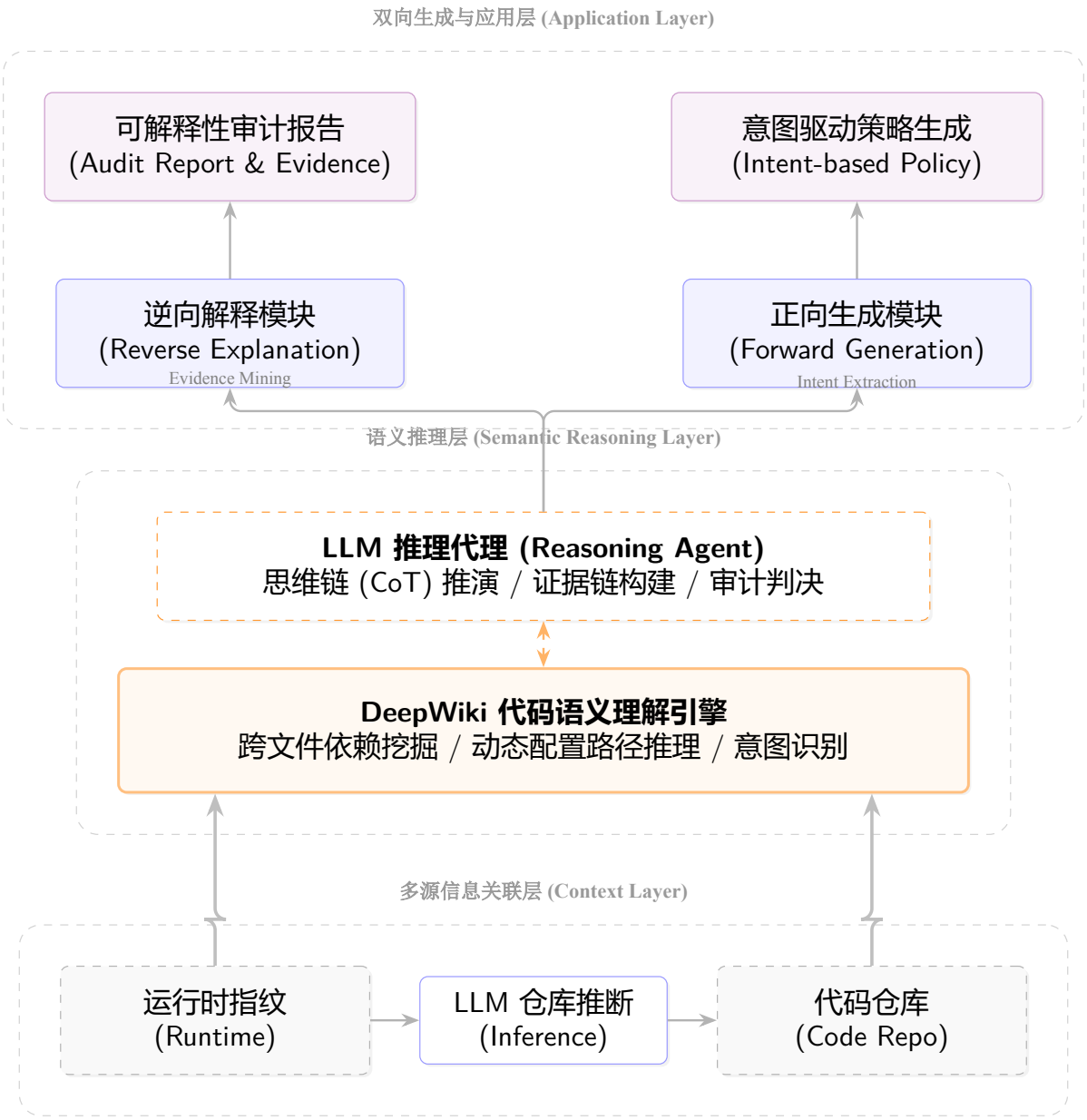


**图 4.1 基于大模型语义分析的双向策略智能生成系统架构**

## 4.1 多源信息采集与代码仓库关联机制

在上一章中，系统结合运行时探针与字节码静态分析手段，从物理执行层面对微服务集群的通信关系进行了还原，较为准确地解决了服务之间“谁与谁发生了通信”的问题。然而，实践表明，仅依赖二进制制品（Artifacts）和底层网络状态的确定性分析方式仍然存在明显局限。一方面，编译后的字节码虽然完整保留了程序的控制流和调用关系，但在编译过程中不可避免地丢失了大量由开发者在编码阶段引入的语义信息，例如业务注释、架构说明文档、具有业务含义的变量命名以及代码组织方式等。这类信息往往直接反映了服务的设计意图和业务边界，其缺失使得即便引入通用大语言模型，也难以仅凭反编译代码或底层元数据开展有效的深层逻辑推理。另一方面，字节码静态分析本身对语言体系具有较强依赖性，在多语言混合的微服务环境中，往往只能覆盖特定语言的运行时，难以形成统一的分析视角。

针对上述问题，本文尝试从系统层面弥合运行态行为与开发态语义之间的差距，提出了一种贯通“运行时（Runtime）”与“开发态（Development）”的全链路溯源机制。该机制不再局限于容器内部的执行环境，而是引入大模型在知识检索与语义推理方面的能力，将线上运行的 Pod 实例与线下静态代码仓库（Git Repository）建立起一一对应的映射关系。通过这一过程，原本分散、零碎的运行时指纹得以回溯至完整的源代码上下文，使系统能够重新获得开发者在设计阶段所表达的业务语义与架构意图，从而为后续基于代码语义的策略生成提供更加充分和可靠的知识基础。

### 4.1.1 运行时通用指纹特征的采集

在缺乏显式仓库标识的情况下，运行时环境中仍然保留了大量可用于识别代码来源的隐含特征。本文将这类特征统称为运行时通用指纹特征，其本质是能够在不同部署实例之间保持相对稳定、且与代码仓库存在潜在关联的信息集合。

为了在异构的多语言环境中准确识别服务归属，系统首先需要从容器运行时中提取具有普适性的“特征指纹”。这一过程的核心在于获取容器内部的进程执行上下文，而非依赖静态的镜像元数据。

采集模块主要关注以下三类关键运行时数据：

- 关键环境变量：环境变量是云原生应用配置注入的主要载体。系统重点提取包含

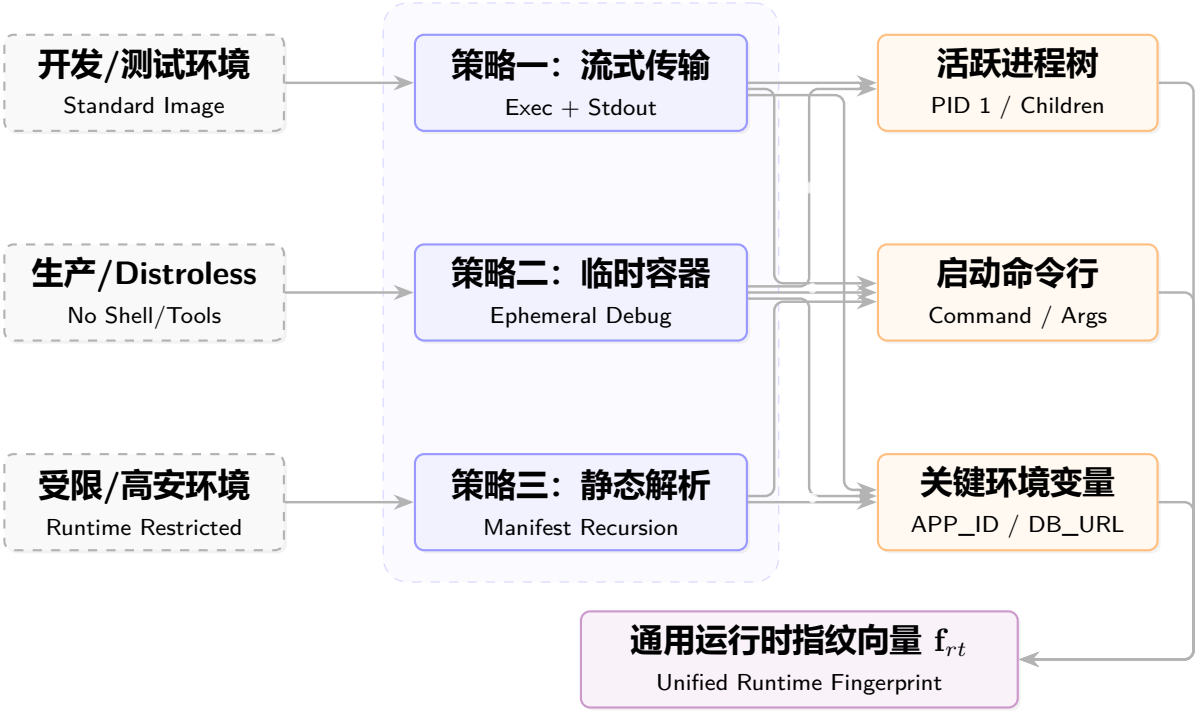


**图 4.2 指纹采集逻辑图**

业务语义的变量，例如服务名称（APP_NAME）、应用 ID（APP_ID）或配置中心地址。这些变量通常在部署清单（Deployment YAML）中定义，是识别服务身份的强特征。

- 启动命令行：解析容器主进程（PID 1）的完整启动命令（Command）及其参数（Args）。无论是 Java 的 -jar 参数、Python 的入口脚本路径，还是 Go 二进制文件的名称，都直接暴露了程序的入口点信息。
- 活跃进程列表：扫描容器内当前运行的所有进程树信息，用于辅助判断容器的实际运行状态和技术栈类型（例如是否存在 node、python 或 java 进程）。

在数据采集的工程实现上，针对不同的镜像类型，本系统设计了三种自适应的采集策略。

#### 4.1.1.1 策略一：基于流式传输的数据采集

该方案通过与 Kubernetes API Server 建立持续连接，基于 HTTP/2 协议维持一条稳定的流式通信通道，用于采集容器在运行过程中的相关信息。具体实现中，采集探针借

助 API Server 向目标容器发起标准化的 Shell 指令执行请求，并对其标准输出（Stdout）进行持续读取，从而逐步获取所需的运行时数据。

与传统的轮询式采集方式相比，流式传输在实现复杂度基本不增加的情况下，能够有效降低数据获取的时延，同时避免因频繁建立和释放连接而产生的额外开销。此外，该方案完全基于 Kubernetes 提供的原生接口完成，不需要在宿主机或集群节点侧额外部署代理组件，对现有集群环境的侵入性较低，部署和维护成本也相对可控。

需要说明的是，该方法依赖于目标容器中保留基本的 Shell 及命令执行能力。因此，其适用场景主要集中在开发测试环境，或尚未进行严格镜像裁剪的传统微服务应用中。在采用 Distroless 等极简镜像构建的生产环境下，由于缺乏必要的执行环境，该方案的可用性将受到一定限制。

#### 4.1.1.2 策略二：基于临时容器的无侵入透视

在对安全性和攻击面控制要求较高的生产环境中，微服务应用往往会采用 Distroless 或 Scratch 等极简镜像进行部署。这类镜像在构建阶段主动移除了 Shell 及常用的基础工具，使容器内部仅保留业务程序及其最小运行时依赖。在这种情况下，依赖 `exec` 机制向容器内注入命令的运行时采集方式将无法执行，传统的命令驱动式采集策略也随之失效。

为应对上述限制，本文在系统中引入了一种基于临时容器（Ephemeral Container）的运行态透视机制。该机制利用 Kubernetes 提供的调试能力，在不重启业务 Pod、也不改动原有镜像内容的前提下，向目标 Pod 动态注入一个独立的临时容器。该临时容器包含完整的基础工具集，并被配置为与业务容器共享同一进程命名空间（Process Namespace），从而具备观察业务进程运行状态的能力。

在此基础上，采集探针无需进入业务容器本身，而是从临时容器一侧直接访问共享的 `/proc` 文件系统，读取业务容器中主进程（通常为 PID 1）对应的 `/proc/<PID>/environ`、`/proc/<PID>/cmdline` 等关键信息。通过这种方式，可以在不依赖 Shell 工具的情况下获取完整的运行时环境信息，有效规避极简镜像带来的功能缺失问题[51,74]。由于整个过程不涉及对业务容器镜像或文件系统的任何修改，该方案符合“不可变基础设施”的设计理念，因而更适合应用于安全约束较为严格的生产部署场景。

#### 4.1.1.3 策略三：基于元数据定义的静态解析

在部分安全策略要求极为严格的部署环境中，集群可能会明确禁止任何形式的运行时注入操作，包括基于 exec 的命令执行以及临时容器的引入。在这类运行时接口完全受限的场景下，系统无法从容器执行层面获取信息，因此需要回退至不依赖运行时能力的采集方式，以保证基础分析能力的可用性。

针对上述情况，系统采用了一种基于 Kubernetes 元数据的静态解析策略。该策略不与容器运行时直接交互，而是从声明式配置入手，通过解析 Pod 的定义信息还原容器在启动阶段的关键运行参数。采集器基于 Kubernetes Informer 机制持续监听 Pod 对象的变更事件，并在检测到相关更新后触发对应的解析流程。

在具体实现中，系统首先对 spec.containers.env 字段进行解析，并进一步递归处理 envFrom 中引用的 ConfigMap 与 Secret 资源。解析过程中，系统遵循 Kubernetes 对环境变量的覆盖与合并规则，最终构建出容器启动时实际生效的环境变量集合。同时，采集器遍历 spec.volumes 的定义，识别以文件形式挂载的 ConfigMap，并提取其中的关键配置内容，例如 application.properties 等常见配置文件，从而补充环境变量无法直接表达的配置信息。

需要指出的是，该方式无法感知运行过程中动态生成的临时变量或配置变更。但作为一种兜底手段，在运行时接口受限的情况下，该策略仍然能够获取服务注册中心地址、数据库连接串等与服务拓扑直接相关的核心信息，为后续依赖分析与策略推导提供必要的基础支撑。

### 4.1.2 基于大模型的仓库推断

在获取容器运行时的通用指纹特征后，系统面临的核心问题在于如何将这些来源多样、形式非结构化的运行时“痕迹”，准确映射回确定性的静态代码仓库地址。由于云原生应用生态中广泛混合部署了通用开源基础设施（如 MySQL、Redis）与具体业务微服务组件（如 DevLake、TiDB 的子模块），单纯依赖人工规则或特征匹配的方法难以覆盖复杂多样的组件类型，且维护成本较高。

为此，系统引入基于大语言模型（Large Language Model, LLM）的仓库推断机制，利用其在开源代码生态与软件工程语义方面的知识记忆能力，实现从运行时指纹到 GitHub

仓库地址的零样本推断。该机制的核心在于构建具备明确推理约束的 Prompt 模板（即 github_code_prompt），通过显式引导模型的分析路径，使其能够在无需人工规则库支持的情况下完成可靠推断。

#### 4.1.2.1 多视图证据链构建

为提高推断结果的稳定性与鲁棒性，系统并未依赖单一类型的运行时特征，而是引导大模型从多个语义视角综合分析容器的运行环境信息，构建完整的证据链。首先，模型被提示优先检查 Kubernetes 编排层面的元数据特征，尤其是由 Helm Chart 或部署模板注入的标准标签（Labels），例如 app.kubernetes.io/name，以及项目特有的组件标识标签。这类信息通常直接反映了工作负载在系统中的逻辑身份，是识别项目归属的强信号来源。

当编排元数据不足以形成明确判断时，模型进一步分析容器环境变量中所蕴含的配置语义特征。一方面，通过识别服务端点相关变量命名模式（如 *_URL、*_SERVICE_*），模型可以推断该组件在系统中的交互对象与功能角色；另一方面，环境变量中的项目前缀或命名约定（如 MYSQL_ROOT_PASSWORD、MY_RELEASE_DEVLAKE）往往直接暴露其所属的软件家族或上层项目。

在此基础上，模型还会结合容器的启动命令及参数信息，对实际执行语义进行补充判断。例如，通过识别启动的二进制文件名称（如 mysqld、nginx）或特定参数标志，进一步确认容器内主进程的类型与用途，从而完善整体推理依据。

#### 4.1.2.2 标准组件与业务项目的消歧

在综合多视图证据后，大模型仍需面对一个关键问题，即如何区分通用标准软件与隶属于特定业务项目的组件。为此，系统在 Prompt 中显式注入消歧逻辑，引导模型在推断过程中优先判断组件的“归属层级”。

对于被识别为通用基础设施的软件组件，例如以 mysqld 作为主进程的数据库容器，模型被要求将其映射至对应的官方维护仓库（如 https://github.com/mysql/mysql-server）。相对地，对于属于大型微服务系统的子组件，即便其运行于通用语言基础镜像之上，模型也应优先将其归属到项目级主仓库（如 https://github.com/apache/incubator-devlake），而非语言或运行时本身的仓库。这一策略有效避免了因运行环境相似而导致的误判问

题。

#### 4.1.2.3 结构化输出约束

为确保仓库推断结果能够被后续的 DeepWiki 引擎直接消费，系统在 Prompt 设计中引入了严格的结构化输出约束。通过指令微调（Instruction Tuning）的方式，模型被强制要求仅返回标准的 HTTPS 克隆地址格式，例如 https://github.com/owner/repo，并自动剥离任何解释性或冗余文本。

通过“多视图信息组装—显式逻辑引导—输出格式约束”的协同设计，系统充分发挥了大语言模型在代码生态领域的知识泛化能力，在无需人工维护规则集的前提下，实现了对多样化开源组件代码仓库的精准定位[49]。

### 4.1.3 基于 DeepWiki 的配置文件路径推理与采集

在通过第 4.1.2 节完成对微服务源代码仓库的精准定位后，系统进一步面临的关键任务是解析服务内部所依赖的隐式网络配置，例如数据库连接地址、服务注册中心位置等。这类信息通常存储于配置文件中，是理解微服务通信关系与依赖拓扑的重要基础。然而，不同项目在配置文件的组织方式上差异显著，其存储路径往往受到构建脚本、启动参数以及环境变量的共同影响，难以通过静态规则直接确定。

针对上述问题，系统并未简单依赖通用大语言模型对配置路径进行猜测，而是引入 DeepWiki 代码理解引擎，将第 4.1.1 节采集的运行时上下文信息与代码仓库的静态结构相结合，执行基于证据约束的配置文件路径推理。通过这一方式，路径推断过程不再是孤立的语义推测，而是建立在代码事实与运行环境之上的可解释推理过程。

#### 4.1.3.1 跨模态路径推理机制

为实现对配置文件路径的准确定位，系统构建了推理提示词（config_file_prompt），引导 DeepWiki 在代码仓库中执行“运行态信息驱动、静态结构验证”的联合分析流程。首先，DeepWiki 会结合容器运行时的主进程启动命令及其参数，在代码仓库中定位对应的程序执行入口。例如，通过分析 Dockerfile 中的 ENTRYPOINT 指令或启动脚本（如 docker-entrypoint.sh），确定应用在启动阶段加载配置的具体逻辑位置。

在此基础上，DeepWiki 进一步分析代码与启动脚本中对环境变量的读取行为，将运行时采集到的环境变量与代码中的使用位置进行映射。通过判断诸如 MYSQL_HOME、NACOS_APPLICATION_CONF 等变量是否参与配置路径的重定义，系统能够推断默认配置加载路径是否被显式覆盖。

当运行时参数与环境变量均未提供明确指向时，推理过程将回溯至镜像的构建阶段。DeepWiki 会解析 Dockerfile 中的 COPY 或 ADD 指令，追踪配置文件在构建过程中被放置的位置，从而直接推导其在容器文件系统中的绝对路径。这一方式避免了依赖经验性“惯例路径”的不确定性，使推断结果具备更高的确定性。

#### 4.1.3.2 输出约束与运行时验证

为保证推理结果能够稳定地被自动化采集流程使用，系统对 DeepWiki 的输出施加了严格约束。模型仅被允许返回唯一的配置文件绝对路径（例如 /etc/my.cnf），或在无法确定时返回 None，以避免歧义性结果进入后续流程。

在获得候选路径后，系统调用第 4.1.1 节所述的运行时采集探针，对目标容器内的对应文件进行实地验证与读取。通过这一机制，DeepWiki 的代码理解能力用于解决“配置文件位于何处”的问题，而运行时探针则负责完成“配置内容如何获取”的实际操作，从而形成完整的闭环。

这种“静态路径推导与动态运行时验证相结合”的方法，有效弥补了单一静态分析难以覆盖运行时差异、单一动态分析难以准确定位文件路径的不足，使系统能够在复杂多变的云原生环境中，实现对微服务关键配置的稳定、精准采集。

## 4.2 基于代码上下文的策略逆向解释机制

通过 4.1 节构建的多源采集与关联机制，系统已成功打破了容器运行时的黑盒边界，建立了从动态 Pod 到静态代码仓库的确定性索引。然而，在零信任微隔离的实际运维场景中，仅拥有代码链接尚不足以解决“策略可读性差”的核心痛点。现有的 Kubernetes NetworkPolicy 资源对象通常以 YAML 格式存储，其描述语言仅包含抽象的标签选择器（Label Selector）与端口号（如 allow port 8080）。这种描述方式虽然满足了底层 CNI 插件的执行需求，却完全剥离了策略背后的业务语义。

对于安全运维与审计人员而言，面对成百上千条由 IP 段和端口构成的规则，往往陷入“知其然而不知其所以然”的困境：无法判断某条规则是承载了核心支付业务，还是仅仅为了通过健康检查；也难以确认该端口的开放是否符合代码层面的最小权限设计。这种“语义鸿沟（Semantic Gap）”不仅增加了误配置的风险，更严重阻碍了自动化策略在严格合规环境下的落地。

为此，本节提出了一种基于代码上下文的策略逆向解释（Reverse Explanation）机制。该机制视“代码”为解释网络行为的唯一真值来源（Source of Truth），通过将运行时状态完整输入 DeepWiki 引擎，从源代码层面反向推导策略的“调用意图（Intent）”与“支撑证据（Evidence）”，旨在将晦涩的底层 ACL 规则转化为具备可解释性、可审计性的自然语言报告。

### 4.2.1　策略与工作负载信息的级联采集机制

要对一条抽象的网络策略进行语义解释，系统的首要任务是将其“锚定”到具体的业务实体上。Kubernetes NetworkPolicy 本质上是面向标签（Label-oriented）的声明式规则，它仅通过 `podSelector` 定义作用域，而不直接绑定具体的容器实例。因此，在调用 DeepWiki 引擎之前，系统必须执行一个逆向的级联采集流程，从静态的 YAML 规则出发，层层回溯至动态的 Pod 实例，最终锁定到静态的代码仓库，构建出完整的“策略-实例-代码”证据链。

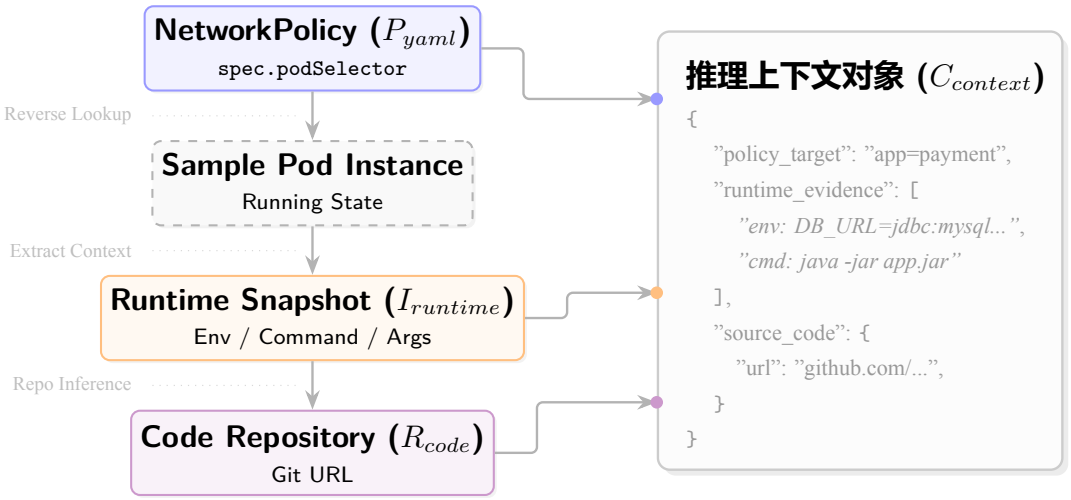


**图 4.3 策略与工作负载信息的级联采集机制**

这一级联采集过程如图4.3, 包含以下三个关键步骤：

#### 4.2.1.1 基于选择器的策略-实例反向锚定

系统通过 Kubernetes Informer 机制持续监听集群中的 NetworkPolicy 资源。当触发对某条策略的解释请求时，解析流程首先从策略对象中提取其 `spec.podSelector` 字段，用以确定策略所作用的目标 Pod 范围，同时读取 `policyTypes` 字段以明确该策略约束的通信方向。

在实际集群中，标签选择器往往会命中多个 Pod 副本。例如，一条以 `app=payment` 为选择条件的策略，可能同时作用于多个无状态副本。若对所有匹配实例逐一分析，不仅会带来额外开销，也难以在语义层面提供更多有效信息。基于这一考虑，系统通过调用 Kubernetes API 的 `List` 接口，在策略所属的命名空间内反向查询所有标签匹配且处于 Running 状态的 Pod 实例，用以反映当前策略在真实工作负载上的作用情况。通过这一处理过程，原本抽象的策略选择条件被具体映射到可观测的运行实体，从而为后续基于运行态与代码语义的分析提供了明确的物理载体。

#### 4.2.1.2 面向策略验证的运行时上下文构建

在锁定目标 Pod 之后，系统通过第 4.1 节中已经介绍过的底层采集流程，构建了一份用于语义分析的运行时快照（Runtime Snapshot），以支持后续对策略合理性的判断。

其中，首先关注的是与网络通信行为密切相关的配置上下文。系统提取容器环境变量中涉及网络连接的关键字段，例如 DB_HOST、Consul_ADDR、PORT 等。这类变量通常作为代码中网络访问逻辑的配置入口，决定了服务在运行时实际连接的目标地址与端口范围。在策略解释阶段，这些信息可直接用于校验端口放行与依赖关系是否与代码意图一致，是 DeepWiki 进行端口合理性分析的重要依据。

此外，系统还强调容器主进程的启动命令及其参数信息。该部分上下文反映了容器在启动阶段加载的具体业务模块及运行模式，有助于明确程序的实际入口点。通过结合启动命令与参数，DeepWiki 能够在代码层面更准确地定位与当前运行实例对应的执行路径，从而减少因多模块或多启动模式带来的语义歧义。

4.2.1.3 从实例到代码仓库的索引闭环

最后，系统利用 4.1.2 节建立的仓库推断能力，将上述运行时指纹映射为确定的源代码仓库地址（Git URL）。

至此，系统成功构建了一个结构化的推理上下文对象（Inference Context Object），其包含：

$$C_{context} = \{P_{yaml},\ I_{runtime},\ R_{code}\} \tag{4-1}$$

其中 $P_{yaml}$ 为待解释的网络策略，$I_{runtime}$ 为 Pod 运行时快照，$R_{code}$ 为代码仓库索引。这一上下文对象打破了网络层与应用层的隔离，为后续 DeepWiki 引擎深入代码内部挖掘“该策略为何存在”提供了唯一的物理入口。

### 4.2.2 容器级网络行为的局部推理机制

在 Kubernetes 的实际部署中，Pod 往往采用多容器协同模式（如 Istio 的 Sidecar 代理、日志采集器与业务容器共存）。为了精准归因网络流量的产生源头，系统采用了“分治”策略，首先对 Pod 内的每一个独立容器执行局部语义推理。

系统构建了专用的容器分析提示词（container_context_prompt_for_explain），设定大模型扮演“Kubernetes 网络安全专家”的角色，输入单一容器上下文与当前 NetworkPolicy 摘要，执行以下标准化的推理流程。

4.2.2.1 合理性优先的推理逻辑

不同于常规的漏洞扫描工具侧重于“找茬”，本模块的设计目标是解决“可解释性”问题，流程如图4.4。因此，提示词中明确植入了“合理性优先（Rationality-First）”的分析原则：即假设当前运行中的策略是满足业务可用性的，模型的首要任务是挖掘支撑该策略存在的“合法性证据”，仅在发现明显的端口错配或协议冲突时才标记为异常。

该逻辑包含四个递进阶段：

1. 硬性约束扫描：模型首先提取容器描述文件（Dockerfile/YAML）中显式声明的 EXPOSE 端口与协议，建立基础的网络指纹。

2. 多源证据挖掘：提示 DeepWiki 深入扫描代码层面的隐式证据（如 Node.js 中的

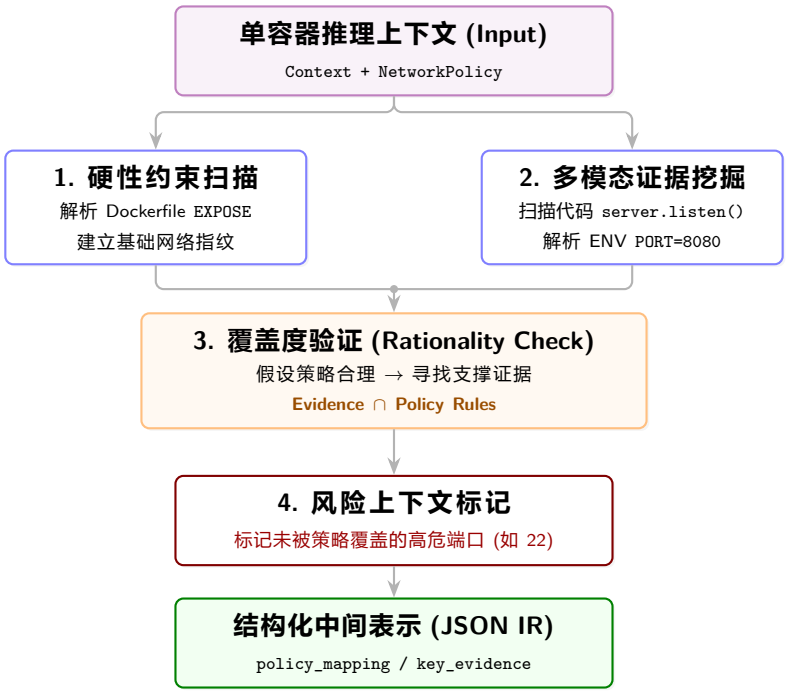


**图 4.4 容器级网络行为的推理流程**

server.listen()）、环境变量层面的配置证据（如 PORT=8080）以及配置文件中的绑定地址。这一步确保了“没有任何一条规则是凭空臆造的”。

3. 覆盖度验证：将上述证据与 NetworkPolicy 的 Ingress/Egress 规则进行比对。例如，验证策略放行的 TCP 3306 是否确实对应了容器内的 MySQL 进程，并生成自然语言解释：“MySQL 服务器监听 TCP 3306，客户端需通过此端口执行事务”。

4. 风险上下文标记：若发现容器内有未被策略覆盖的高危端口（如开放了 22 端口但策略未放行），模型将在输出中标记潜在的阻塞风险，为后续的聚合分析提供警示。

#### 4.2.2.2 结构化中间表示（IR）

为了支持大规模自动化分析，该环节的输出被严格约束为 JSON 格式的中间表示（Intermediate Representation, IR）。输出结构包含三个核心字段：

- policy_mapping：布尔值的匹配结果与对应的自然语言解释，直接回答“策略是否覆盖了需求”。
- key_evidence：提取的代码片段或配置项列表，作为可追溯的审计证据。

- pod_context：高度凝练的容器网络画像（如“Ingress TCP 8080 + Egress HTTPS API”），用于后续 Pod 级别的上下文聚合。

这种标准化的 JSON 输出屏蔽了不同大模型的行文风格差异，使得上层模块能够像处理数据库记录一样处理非结构化的语义解释。

### 4.2.3 Pod 级全局上下文聚合与审计机制

Kubernetes 的网络模型以 Pod 为最小调度单元，所有共享同一 Network Namespace 的容器（含初始化容器、Sidecar 代理及业务容器）共同对外呈现统一的 IP 地址与端口视图。然而，4.2.3 节的推理是基于单一容器视角的。为了应对 Sidecar 模式下的复杂流量特征（例如 Istio Proxy 接管流量、Prometheus Exporter 暴露指标），系统需要执行最后一步的全局上下文聚合与最终审计。

系统构建了聚合层提示词（pod_aggregation_prompt），将 4.2.3 节输出的所有容器级 JSON 中间表示组装成数组，连同完整的 NetworkPolicy YAML 一并输入大模型。模型被设定为“审计专家（Auditor）”，执行以下三个维度的全局推理：

#### 4.2.3.1 多源需求的并集聚合

首先，模型需要应对的是“需求碎片化”问题。在实际部署中，一个 Pod 往往包含多个容器，而一条合法的 Pod 级 NetworkPolicy 必须同时满足 Pod 内所有容器的通信需求，而不能仅依据某一个业务容器进行推断。

在生成策略时，模型会遍历该 Pod 中所有容器对应的 container_summary 信息，将各容器的入站需求进行统一整理。例如，业务容器对外提供的 8080/TCP 服务端口，需要与 Sidecar 容器用于健康检查的 15020/TCP 一并纳入 Ingress 需求集合，最终形成覆盖整个 Pod 的统一规则列表，而非分散的容器级判断。

在少数情况下，不同容器可能会对同一端口给出用途描述不一致的证据。针对这类冲突，模型不会简单地进行覆盖或取并集处理，而是结合证据来源的可靠性进行消解。具体而言，模型会对不同证据赋予权重，并优先采信能够在代码层面明确定位的绑定关系，例如通过 server.listen 等显式配置所确认的端口用途，从而避免因语义歧义导致策略生成偏离实际行为。

#### 4.2.3.2 基于证据的合规性裁决

在构建了完整的 Pod 需求视图后，模型对比输入的 NetworkPolicy，进行最终的“Accept/Reject”二元裁决。为了保证工程落地的稳定性，系统在提示词中植入了严格的“保守拒绝原则（Conservative Rejection Principle）”：

- **通过标准（Accept）**：只要策略覆盖了聚合后的所有核心业务端口，且未出现明显的协议冲突（如需求是 UDP 但策略仅允许 TCP），即判定为通过。此时，模型生成的解释报告会综合引用各容器的代码证据（如“Ingress 允许 3306，因为容器 A 的代码中 MySQL 监听了该端口”），形成完整的合理性论述。
- **拒绝标准（Reject）**：仅在发现“阻塞必需流量”或“高危配置冲突”等确凿错误时，才触发拒绝动作。模型必须在 修复建议字段中输出具体的 YAML 修正片段，并严格遵循最小权限原则，不得臆造不存在的规则。

#### 4.2.3.3 标准化审计报告生成

最终，系统将大模型的推理结果格式化为人类可读的 Markdown 报告。该报告包含四个标准板块：评估结果、Pod 网络摘要、策略解释以及问题评估/修复建议。

这种“分层推理（局部容器 → 全局 Pod）”的设计，不仅完美适配了 Kubernetes 的 Sidecar 模式，更通过最终的聚合校验，确保了生成的微隔离策略在保障安全的同时，绝对不会破坏复杂微服务架构下的业务可用性。[15]。

## 4.3 基于代码上下文的策略正向生成机制

通过前两节的分析，系统已经打通了从运行时实例到静态代码仓库之间的语义映射路径，并在此基础上实现了对既有 NetworkPolicy 的深度审计与可解释性分析。然而，在零信任架构（Zero Trust Architecture）的完整治理流程中，仅具备事后审计能力仍然难以满足云原生应用高频迭代的实际需求。相比发现问题本身，运维人员更关注的是如何快速完成策略的构建,

前一章提出的动静态分析框架虽然在 Java 生态中具有较强的分析能力，但其适用范围受限于 JVM 语言体系，难以覆盖当前云原生环境中广泛存在的多语言微服务场景。

相比之下，基于代码语义与仓库级上下文的大模型分析能够跨越语言边界，对不同技术栈的服务采用统一的意图抽取方式，从而为异构系统提供一致的策略生成能力。

为了克服上述缺陷，本节提出了一种“大语言模型理解代码意图（Intent-based）”的策略生成范式。该机制视源代码为网络行为的“第一性原理”，利用 DeepWiki 引擎从代码逻辑中直接提取确定的通信意图（如监听端口、上游依赖），从而在无流量基线的情况下生成精准的预定义策略，并结合大模型的推理能力实现审计问题的自动化修复。

### 4.3.1 基于代码意图的正向策略生成机制

本系统提出了一种基于代码意图（Intent-based）的正向策略生成机制，该机制视源代码与运行时配置为网络行为的“第一性原理”，通过 DeepWiki 引擎执行严格的静态与动态证据挖掘，分别推导微服务的入站（Ingress）服务面与出站（Egress）依赖面，从而在无流量基线的情况下构建“默认安全”的白名单策略。

**表 4.1 流量意图类型与 NetworkPolicy 规则合成逻辑对照表**

| 流量意图类型 | DeepWiki 检测逻辑 | 规则合成逻辑 (Rule Synthesis) |
|---|---|---|
| 集群内服务调用 | 识别 K8s Service 名称或内部 DNS 域名 (e.g., `user-svc`) | 生成 `podSelector`，匹配目标服务 Labels。自动处理 `namespaceSelector`。 |
| 集群外 API 访问 | 识别公网域名 (e.g., `api.stripe.com`) | 解析 DNS 获取 IP 列表，生成 `ipBlock`(CIDR)。建议配合 Egress Gateway。 |
| 基础设施依赖 | 识别数据库/中间件连接 (e.g., `jdbc:mysql://...`) | 优先匹配集群内 StatefulSet; 若为外部 RDS，则生成固定 IP 白名单。 |
| 基础网络服务 | 识别 DNS 解析请求 (UDP 53) | 自动注入 `kube-dns` 放行规则 (UDP/TCP 53)。 |

#### 4.3.1.1 服务端点挖掘与 Ingress 规则构建

针对入站流量控制，系统首先需要准确识别微服务实际对外暴露的监听端口。为此，DeepWiki 引擎并未停留在对 Dockerfile 中 EXPOSE 指令的表层扫描，而是深入分析应用代码中的端口绑定逻辑，从源头还原服务真实的对外接口。

在具体实现上，对于 Kafka、Zookeeper 等端口由配置动态决定的组件，系统沿着代码中的配置读取路径进行追踪，例如定位 config.get(”listeners”) 等关键调用点，并结合 4.1.3 节中采集到的运行时配置取值，推导出实例中实际生效的监听地址与端口范围。通过这种方式，可以有效避免由于默认配置与运行时覆盖不一致而导致的策略缺失或误放通问题。

此外，系统还会分析应用内部是否存在显式的访问控制逻辑。当代码中包含基于源地址的白名单校验（如对 remote_addr 的 CIDR 判断）时，DeepWiki 将其视为一种已经在应用层实现的安全约束，并自动将该约束映射为 NetworkPolicy 中的 ipBlock 规则，从而将原本分散在代码中的访问控制语义下沉到网络层执行。该过程不仅减少了策略配置的随意性，也提升了应用层与基础设施层安全策略之间的一致性。

#### 4.3.1.2 服务依赖拓扑与 Egress 规则构建

针对出站流量控制，系统围绕容器级依赖识别设计了专用的依赖分析提示模板（container_egress_prompt，用于约束 DeepWiki 在分析过程中遵循严格的证据驱动原则。分析过程中不允许基于经验或常识进行推断，只有在代码或配置中存在明确依据时，系统才会生成对应的出站访问规则，从而避免策略过度放宽。

整体分析流程从三个互补的维度展开。

首先，系统对代码中的客户端调用模式进行系统性扫描，覆盖常见的网络访问形式，包括数据库连接（如 mysql.connect()）、HTTP 请求（如 axios.get()）、RPC 框架调用（如 grpc.Dial()）以及消息中间件交互（如 kafka.Producer()）。在识别相关调用点后，系统进一步解析其参数来源，提取目标主机、端口及协议等关键信息，以此作为出站依赖的直接证据。

其次，系统对环境变量与配置文件中的网络相关配置进行语义对齐分析。通过解析诸如 DATABASE_URL、application.yml 等配置项中的地址信息，系统能够补全代码中

以变量形式引用的外部依赖。同时，DeepWiki 在分析过程中会区分服务自身的监听端口与作为客户端访问目标时使用的端口，避免将入站行为误判为出站依赖，从而确保策略方向的正确性。

最后，为了生成可直接应用于 Kubernetes 的精确选择器，系统会对识别出的目标地址进行内外部服务分类处理。对于集群内部依赖，系统通过识别.svc.cluster.local 后缀或短服务名（如 redis），将其解析为对应的命名空间与 Pod 选择条件，使生成的 NetworkPolicy 能够随工作负载实例的动态变化自动适配。对于集群外部依赖，则识别公网域名或固定 IP 地址，并结合 DNS 解析结果生成精确的 CIDR 范围，构造基于 ipBlock 的出站规则。

#### 4.3.1.3 证据链构建与策略合成

为了保证自动生成策略的可审计性，系统要求 DeepWiki 输出严格的 JSON 格式证据链（Evidence Chain）。对于每一个生成的网络规则，系统都附带了以下元数据：

- 代码证据:具体的函数调用行号或配置项路径（例如 env: MYSQL_HOST=10.0.1.5）。
- 业务目的：基于代码上下文推断的流量用途（如“用户数据持久化”或“第三方支付 API 调用”）。

最终，系统将上述 Ingress 与 Egress 需求合并，转换为标准的 Kubernetes NetworkPolicy YAML 文件。这种机制确保了每一条生成的规则都有据可查，既实现了最小权限控制，又彻底消除了运维人员对“黑盒”自动生成规则的信任危机。

### 4.3.2 基于 DeepWiki 的自适应策略生成与验证

在完成正向的意图提取后，系统的最后一步是将这些半结构化的依赖数据转化为标准、可执行且严谨的 Kubernetes NetworkPolicy 资源对象。为了确保生成策略的精确性并杜绝大模型的“幻觉”风险，系统设计了基于 DeepWiki 的自适应生成流水线。

该流水线包含三个核心处理阶段，严格遵循“无证据不生成（No Evidence, No Rule）”的原则。

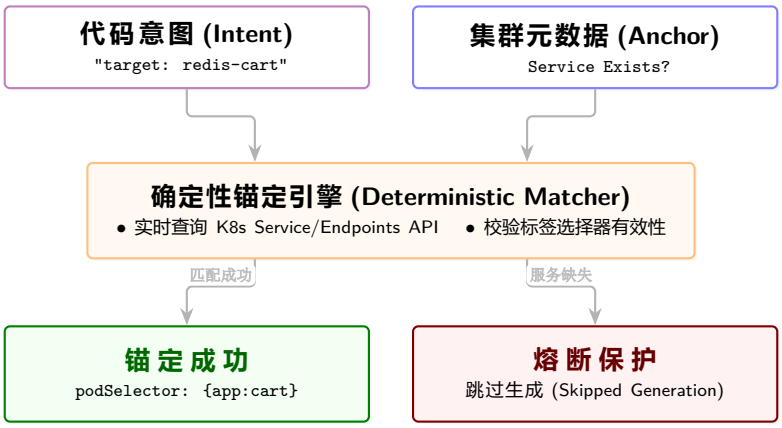


**图 4.5 确定性锚定与熔断机制**

#### 4.3.2.1 服务选择器的确定性锚定

NetworkPolicy 的核心控制机制基于标签选择器（Label Selector）对流量进行约束，因此，将代码中出现的服务标识（例如 redis-cart）准确映射为集群内实际生效的 Pod 标签，是策略自动生成过程中必须解决的关键问题。

为避免传统基于字符串相似度或经验规则的模糊匹配带来的不确定性，系统采用了一种确定性的锚定机制，其整体流程如图 4.5 所示。该机制以 Kubernetes 集群的实时元数据为唯一可信依据，确保生成结果具有可验证性和可追溯性。

在具体实现中，DeepWiki 引擎在解析代码中显式声明的外部依赖时，会同步查询 Kubernetes 的 Service 与 Endpoints 资源。只有当代码中识别出的目标服务名能够在当前集群中精确匹配到实际存在的 Service 对象，且该 Service 定义了有效的 matchLabels 选择条件时，系统才会据此构造对应的 podSelector，将抽象的代码依赖关系锚定到具体的工作负载集合。

当目标服务在集群中不存在，或其未绑定任何可用于选择 Pod 的标签（即 service_selector_labels 为空）时，策略生成模块将触发熔断机制，主动放弃该依赖项的规则生成。系统不会基于猜测或默认假设进行补全，并在生成的策略解释报告中明确标注“因缺乏有效集群元数据而跳过生成”，从而避免因标签映射错误在生产环境中引入非预期的流量阻断风险。

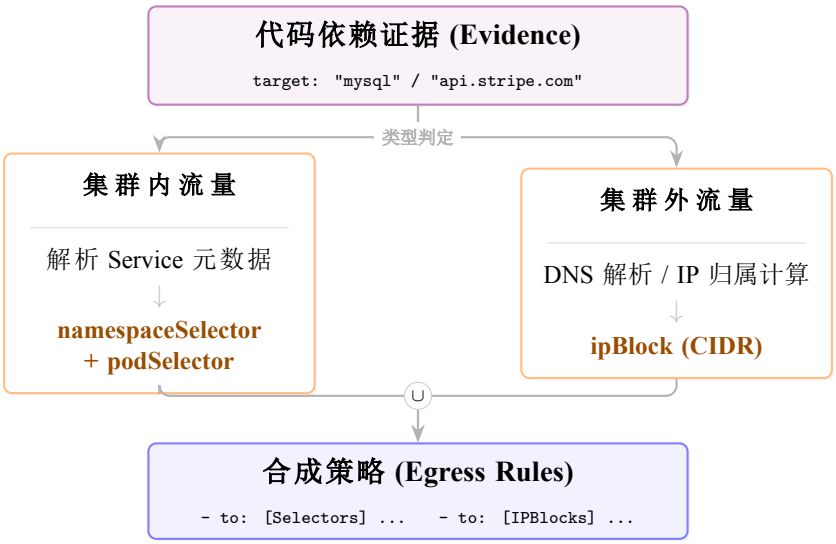


**图 4.6 基于证据的规则合成流程**

#### 4.3.2.2 基于证据的规则合成

在网络规则的最终合成阶段，如图 4.6 所示，系统对不同类型的通信需求执行严格的分类处理逻辑，确保生成的每一行 NetworkPolicy YAML 均能够追溯到明确的代码级或配置级证据来源，从而避免策略膨胀或隐式放行。

对于集群内的东西向流量（East-West Traffic），即被标记为 is_external: false 的内部服务依赖，策略生成器采用基于 Kubernetes 原生语义的选择器组合方式进行建模。具体而言，系统通过构建 namespaceSelector 与 podSelector 的联合约束，将代码中抽象的服务调用关系精确映射为集群中的目标工作负载集合。当目标服务与当前 Pod 不位于同一命名空间（例如访问 mysql.default）时，系统会自动解析目标命名空间的元数据标签（如 kubernetes.io/metadata.name: default），以保证跨命名空间访问在策略层面被显式、正确地放行，而非依赖隐含的全局默认行为。

对于集群外的南北向流量（North-South Traffic），即标记为 is_external: true 的外部依赖（如第三方 API 服务），生成器优先基于代码与配置中提取的域名信息执行 DNS 解析，并据此生成精确收敛的 ipBlock CIDR 规则，以最大程度缩小出站白名单的攻击面。当检测到目标服务使用动态 IP 机制（例如经由 CDN 分发）而无法获得稳定地址集合时，系统会启用安全兜底策略：生成带有明确风险说明注释的 0.0.0.0/0 出站规则，同时强制排除集群内部私有地址段（如 10.0.0.0/8、172.16.0.0/12 等），从策略层面避免因外部放行规则配置不当而引发的内部网络暴露风险。

#### 4.3.2.3 零信任兜底与策略说明

在生成最终 NetworkPolicy YAML 文件的同时，系统会自动附带一份结构化的策略生成说明书（Policy Manifest），用于对策略的生成依据与潜在风险进行同步呈现，从而提升策略落地前的可审计性与可决策性。

说明书首先体现零信任微隔离中的默认拒绝原则。例如当代码分析结果表明某一容器不存在任何明确的出站通信需求，即其 egress_requirements 为空时，系统不会推断或补全潜在依赖，而是直接生成仅声明 policyTypes: [Egress] 且 egress 规则列表为空的 NetworkPolicy 对象。该策略在 Kubernetes 语义下等价于对该 Pod 启用“全出站阻断”，从基础设施层面实现严格的网络隔离，避免因隐式放行造成的攻击面扩大。

此外，策略说明书还承担了风险可视化与人工复核支撑的作用。系统会针对每一条生成的网络规则，明确列出其对应的证据来源（如代码位置、配置项或运行时指纹）、在代码路径中的使用频率，以及基于依赖类型与作用范围评估得到的风险等级。这些信息使安全管理员在策略正式应用前，能够快速判断规则的必要性与潜在安全影响，而无需回溯底层分析过程。

通过上述机制，系统将源代码中隐含的通信意图自动转化为具备完整解释能力的基础设施即代码（Infrastructure as Code, IaC）制品，形成从“代码语义”到“网络策略”再到“审计说明”的闭环流程，从而保障微隔离策略在整个生命周期内的准确性、可控性与合规性。

## 4.4 本章小结

本章详细论述了基于代码上下文的零信任微隔离策略生成技术。针对云原生环境下的语义缺失难题，本章首先利用大模型推理运行时指纹，建立了运行时容器与静态代码仓库的精准关联；进而通过 DeepWiki 引擎挖掘代码中的业务意图，实现了网络策略的逆向语义解释；最后提出了基于代码逻辑与配置解析的策略生成算法，通过将静态代码分析与动态配置数据相结合，在不依赖流量基线的情况下，生成了兼顾最小权限原则与长尾业务覆盖的自适应微隔离策略，有效解决了现有技术在准确性与可解释性方面的不足。

# 5 实验与分析

## 5.1 实验环境构建

### 5.1.1 硬件环境

本研究的实验验证体系构建于三个 Kubernetes 集群之上。通过构建多个集群，本文旨在模拟企业级生产环境中复杂的集群服务发现、动态容器动态及多节点下的安全策略下发场景，从而验证微隔离策略生成框架的正确性与自适应能力。

实验环境的具体节点角色与硬件配置参数如表 5-1 所示：

**表 5.1 实验环境多集群硬件配置表**

| 集群编号 | 主机名称 | 节点角色 | 主机 IP | CPU 配置 | 内存容量 |
|---|---|---|---|---|---|
| 集群 A | master-A | 控制平面 | 192.168.4.143 | 16 核心 | 30 GB |
| 集群 B | master-B | 控制平面 | 192.168.4.135 | 8 核心 | 32 GB |
| 集群 C | master-C | 控制平面 | 192.168.7.145 | 4 核心 | 16 GB |
| 集群 C | node-C | 工作节点 | 192.168.7.146 | 4 核心 | 16 GB |

### 5.1.2 软件环境

实验集群采用 CentOS 7 操作系统，底层计算资源通过 VMwareWorkstation 与 vSphere 虚拟化平台进行编排。为了实现深度的流量感知与细粒度的策略执行，集群内部署了 Calico 网络插件，支持标准的 NetworkPolicy 声明式语法。

系统关键软件栈及其在实验中的具体职能定义如表 5-2 所示：

## 5.2 基于动静态协同分析的策略生成实验

本节旨在量化评估动静态协同分析引擎在真实微服务项目中的依赖提取能力与策略生成准确性。实验的核心目标是验证该引擎能否在涵盖 Spring Cloud、Dubbo 等多重技术栈的复杂工程中，精准还原服务间的调用拓扑，并据此生成完备的微隔离策略。

**表 5.2 实验环境软件系统版本及用途**

| 软件名称 | 版本 / 技术路径 | 实验用途 |
| --- | --- | --- |
| Kubernetes | v1.23 | 编排能力与策略下发接口 |
| Calico | v3.21 | 负责流量阻断 |
| Soot | v4.6 | 对 Java 项目进行字节码分析 |
| DeepWiki | | 代码仓库级挖掘 |
| LLM APIs | GPT-4o / Deepseek-R1 / Claude-Sonnet-4.5 | 策略校核、推理与生成 |

### 5.2.1 数据集定义

为系统性评估动静态协同分析引擎在复杂微服务架构条件下的依赖识别精度及策略生成效果，本文构建了一套覆盖主流技术体系的实验测试数据集。该数据集旨在尽可能还原真实云原生生产环境中的服务组织形态与依赖模式，从而保证评测结论的可靠性与工程参考意义。

在数据来源方面，实验选取了在 GitHub 与 Gitee 社区中具有较高认可度的开源微服务项目。这些项目覆盖了企业级 ERP 系统、分布式电商平台、RPC 中间件以及包含非标准动态路由机制的复杂业务场景，能够较为全面地反映当前微服务架构在不同行业中的实际应用情况。相关项目的社区关注度普遍处于中高水平，其架构设计与实现方式在工业实践中具有一定代表性，从而为实验结果在存量系统中的推广提供了基础保障。

围绕上述项目中涉及的近百个微服务实例，本文对其进行了完整的静态代码扫描与特征信息抽取。为构建可信的评测基准，实验未依赖自动化工具生成参考答案，而是采用人工代码审查与运行时流量核对相结合的方式，对服务之间的 API 调用关系及 RPC 依赖进行逐一标注。最终形成的基准数据集包含了服务拓扑结构的完整真值信息，可作为评估分析引擎依赖覆盖能力与识别准确性的统一对照标准。

表 5.3 详细列出了这些核心测试项目。

**表 5.3 实验项目集：业务场景与部署规模统计**

| 序号 | 项目名称 | 运行的服务数量 | 业务类型与特征备注 |
|---|---|---|---|
| 1 | RuoYi-Cloud | 9 | 商业级后台管理 |
| 2 | Yudao-Cloud | 12 | 模块化 ERP 系统 |
| 3 | Alibaba-Demo | 5 | Dubbo RPC 官方示例 |
| 4 | SC-Huawei | 4 | 华为云架构示例 |
| 5 | vhr | 6 | 人力资源系统 |
| 6 | PaasCloud | 9 | 电商 PaaS 平台 |
| 7 | Online-Boutique | 11 | 云原生微服务标杆 |
| 8 | Reactive-Shopping | 5 | 响应式商城 Demo |
| 9 | Micro-Scaffold | 6 | 开发脚手架 |
| 10 | Cloud-Platform | 8 | 统一网关平台 |
| 11 | PiggyMetrics | 5 | 个人理财系统 |
| 12 | Mall-Swarm | 7 | 容器化电商 |
| 13 | NewBee-Mall | 7 | 轻量级商城 |
| 14 | JNPF-Cloud | 6 | 低代码平台 |
| 15 | Killbug | 3 | 缺陷管理 |

### 5.2.2 实验结果统计与性能分析

#### 5.2.2.1 实验结果统计

我们将动静态协同分析引擎应用于上述数据集，统计其提取到的依赖关系数量，并与真值矩阵 $M_{GT}$ 进行比对。实验定义策略覆盖率（Policy Coverage, PC）为核心评价指标，计算公式为：

$$PC = \frac{|E_{extracted} \cap E_{GT}|}{|E_{GT}|} \times 100\% \tag{5-1}$$

其中 $E_{extracted}$ 为系统自动生成的策略集合，$E_{GT}$ 为人工审计的真值集合。

基于表 5.4 的实验统计数据，本系统在 15 个测试项目中取得了 13 个 100% 覆盖的优异成绩。

#### 5.2.2.2 性能分析

在云原生架构下，系统的部署与更新均以 Pod（容器组）为最小单元。因此，评估静态分析引擎效能的关键指标并非整个代码仓库的总耗时，而是针对单个微服务单元（Per-Pod）的扫描效率。这直接决定了系统能否在 Pod 扩容或重启的瞬间完成“即时分析”。

图 5.1 统计了实验中不同功能类型的微服务 Pod 的平均分析耗时。

如图 5.1 所示，尽管不同微服务在代码规模与业务复杂度方面存在一定差异，但系

表 5.4 动静态协同分析引擎在开源微服务项目上的测试结果统计

| 序号 | 项目名称 | 核心技术栈 | LOC | 提取特征类型 | 覆盖率 |
|---|---|---|---|---|---|
| 1 | RuoYi-Cloud | Spring Cloud | 85k+ | Feign / RestTemplate | 100% |
| 2 | Yudao-Cloud | SC Alibaba | 120k+ | Feign / RPC | 100% |
| 3 | Alibaba-Demo | Dubbo / Nacos | 15k+ | Dubbo RPC 接口 | 100% |
| 4 | SC-Huawei | Spring Cloud | 12k+ | 注解驱动接口 | 100% |
| 5 | vhr | Spring Boot | 45k+ | 模块化内部接口 | 100% |
| 6 | PaasCloud | Spring Cloud | 68k+ | 动态服务发现 | 100% |
| 7 | Online-Boutique | Go / Java | 19k+ | 跨语言 gRPC 调用 | 0.0% |
| 8 | Reactive-shopping | WebFlux | 8k+ | 异步响应式流 | 13.6% |
| 9 | Micro-Scaffold | Sentinel | 18k+ | 认证调用链 | 100% |
| 10 | Cloud-Platform | Spring Cloud | 55k+ | 网关聚合接口 | 100% |
| 11 | PiggyMetrics | Spring Cloud | 32k+ | 分布式统计 | 100% |
| 12 | Mall-Swarm | Spring Cloud | 41k+ | 订单库存对接 | 100% |
| 13 | NewBee-Mall | Spring Cloud | 28k+ | 组件通信 | 100% |
| 14 | JNPF-Cloud | SC Alibaba | 35k+ | 低代码依赖 | 100% |
| 15 | Killbug | React / Boot | 5k+ | Dubbo RPC 接口 | 100% |

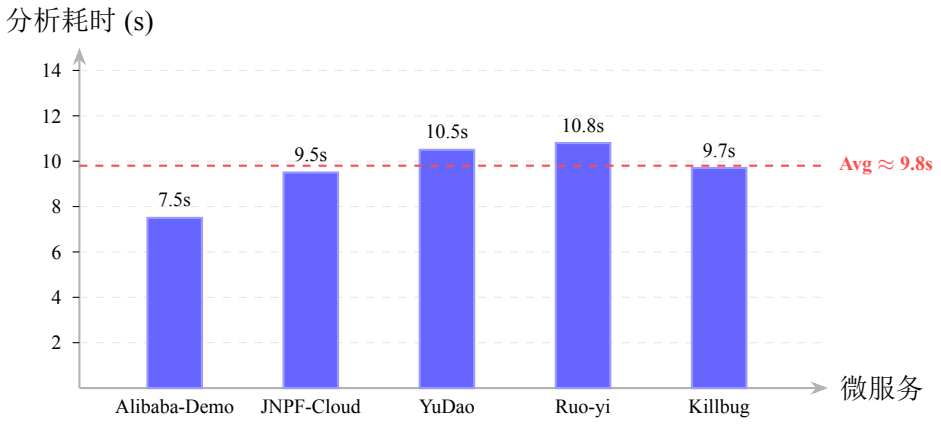


图 5.1 针对不同项目的微服务单元 (Pod) 平均耗时统计举例

统在单个 Pod 维度上的分析耗时整体表现出较高的稳定性，其平均耗时基本维持在 9.8 秒左右。该结果表明，分析过程对服务规模变化不敏感，具备较为稳定的时间开销特征。

从工程实践角度来看，上述时间特性具有重要意义。一方面，在 Kubernetes 等容器编排环境中，微服务实例通常以并行方式完成启动。由于本文系统将分析粒度细化至 Pod 级别，，从而保证了良好的并行扩展能力。另一方面，在典型的 Java 微服务冷启动过程中，JVM 初始化及 Spring 容器加载往往需要 15-30 秒不等。本系统约 9.8 秒的分析过程可以与启动流程并行执行，实现对服务运行过程的无侵入集成，在不额外增加上线时延的前提下完成安全与依赖分析任务。

### 5.2.3 实验分析

#### 5.2.3.1 Java 微服务生态的适配性

实验结果显示，本文提出的系统在 JVM 语言体系下的微服务架构中具备较强的通用适配能力。无论是基于 Spring Cloud、Dubbo 等主流框架构建的标准化微服务体系，还是引入动态配置与非固定路由机制的复杂业务场景，系统均能够完成服务特征的完整提取，未出现因架构差异导致的分析缺失问题。

在典型的标准微服务架构中，以 RuoYi-Cloud、Yudao-Cloud 等采用 OpenFeign、RestTemplate 及 Dubbo RPC 等主流技术栈的企业级项目为代表，系统能够稳定构建出完整且准确的服务调用拓扑。实验表明，即便在代码规模较大、业务逻辑复杂的场景下，分析结果的完整性与准确性仍未受到明显影响，说明系统对代码规模增长具有良好的鲁棒性。

针对部分项目中广泛存在的动态配置注入与非硬编码路由问题，如 Killbug 与 PaasCloud 等场景，系统通过强化静态配置文件解析流程，并结合常量传播等分析手段，对运行时服务地址进行了有效还原。相关结果表明，在无需引入额外外部推理或运行时探测机制的前提下，基于深度静态分析的方法已能够覆盖 Java 微服务环境中大多数由配置解耦与动态路由引入的复杂依赖关系。

#### 5.2.3.2 技术边界与失效场景归因

尽管前述实验结果表明系统在 Java 语言体系内具备较为稳健的分析能力，但对照组实验中个别测试场景进一步揭示了当前基于字节码静态分析方法所面临的技术边界。

这些边界并非实现层面的偶发缺陷，而是由分析范式本身的适用范围所决定。

首先，在跨语言微服务交互场景中，系统表现出明显的能力受限。在 Online-Boutique 项目中，部分服务通过 gRPC 与 Go 或 Node.js 实现的微服务进行通信。由于分析引擎以 JVM 字节码为核心对象，其分析能力天然受限于 Java 运行时环境，无法解析或推断异构语言服务内部的调用语义，最终导致跨语言调用链无法被完整识别。这一现象表明，仅依赖单语言静态分析难以覆盖多语言微服务体系中的全局依赖关系。

其次，异步与响应式编程模型对分析精度也构成了显著挑战。在 Reactive-Shopping 项目中，基于 Spring WebFlux 的响应式架构大量依赖 Reactor 框架所提供的异步回调与函数式组合机制。此类 Lambda 回调链在执行过程中会对传统控制流结构产生离散化影响，使得控制流图难以保持连续性，进而削弱了基于控制流与数据流联合建模的污点分析效果。实验结果显示，在该类场景下，系统对服务依赖关系的覆盖率出现明显下降，反映出静态分析方法在处理复杂异步上下文时的天然局限性。

## 5.3 大模型逆向解释与审计实验

### 5.3.1 数据集定义

为了进一步验证大模型（LLM）在复杂语义环境及多语言异构架构下的策略审计与逆向解释能力，本实验构建了一套包含“云原生基础设施组件”与“异构微服务标杆”的混合测试数据集。

为系统评估本文方法在复杂云原生环境中的适用性与有效性，实验构建了一套具有较高代表性的评测数据集，重点覆盖当前工业界常见的多语言微服务架构与高复杂度部署场景。

在数据集选取方面，本文从 GitHub 社区中筛选了具有较高关注度和广泛应用基础的开源项目。整体数据集由三类典型场景构成：其一是分布式数据库与工作流调度等基础设施级系统，如 TiDB 与 Airflow，这类项目在架构复杂性与组件耦合度方面具有较强代表性；其二是若干 Java 微服务系统，包括 Online-Boutique 与 RuoYi-Cloud 等，这些项目在前序动静态分析实验中表现出不同程度的优势或不足，有助于进一步验证方法在 Java 生态下的稳定性与边界；其三是以 Bank of Anthos、eShopOnContainers 为代表的多语言微服务示例项目，这类系统被广泛视为业界多语言架构设计的参考实现。上述项

**表 5.5 实验项目集：业务场景与部署规模统计**

| 序号 | 应用系统 | 运行的服务数量 | 业务类型与架构特征说明 |
|---|---|---|---|
| 1 | Airflow | 5 | Apache 分布式工作流调度平台 |
| 2 | Devlake | 7 | 研发效能数据聚合与分析平台 |
| 3 | TiDB | 8 | 兼容 MySQL 协议的分布式 HTAP 数据库 |
| 4 | Druid | 10 | 实时大数据分析数据库 |
| 5 | Pinot | 5 | 实时分布式 OLAP 数据存储 |
| 6 | JupyterHub | 7 | 多用户交互式计算环境 |
| 7 | RuoYi-Cloud | 9 | 基于 Spring Cloud 的商业级后台管理系统 |
| 8 | Yudao-Cloud | 12 | 模块化、多租户的数字化转型 ERP 系统 |
| 9 | Reactive-Shopping | 5 | 基于响应式编程 (Reactive) 的商城 Demo |
| 10 | Killbug | 3 | 极简架构的缺陷管理系统 (非标定义) |
| 11 | Online-Boutique | 11 | Google 云原生微服务标杆示例 (gRPC) |
| 12 | Bank of Anthos | 7 | Google 模拟金融交易系统 (Ledger/Balance) |
| 13 | Microservices-K8s | 4 | 影院购票系统 (Luke-Micro, 非标 YAML) |
| 14 | Robot Shop | 10 | Instana 提供的多语言 AI 观测性演示商城 |
| 15 | eShopOnContainers | 9 | Microsoft 官方.NET Core 微服务架构参考 |
| 16 | Sock Shop | 8 | 经典的容器化微服务电商 Demo |
| 17 | Milvus | 7 | 云原生向量数据库 |
| 18 | Vitess | 7 | MySQL 分布式数据库集群中间件 |

目的社区关注度普遍处于较高水平，能够反映当前云原生生产环境中最为复杂的系统形态与跨语言交互模式，从而为实验结论在大规模异构集群治理场景中的推广提供现实依据。

在样本构建与规模方面，本文针对上述项目对应的微服务部署场景，通过解析官方文档以及相关的 YAML 配置文件，构建了 140 条语义正确的 NetworkPolicy 作为正样本基准。在此基础上，进一步通过人工方式引入端口不一致、协议冲突以及标签选择器偏移等常见配置逻辑错误，生成了数量相同的负样本策略。最终形成的审计基准库共包含 280 条正负样本，所有样本均经过人工复核以确保其语义正确性与真值标注的可靠性。

表 5.5 详细列出了这些测试对象的业务类型与服务规模统计。

### 5.3.2 实验结果统计与性能分析

为了全面评估不同大模型在微服务策略审计任务中的表现，本实验将评价维度拆分为准确性（Accuracy）与效能成本（Performance & Cost）两个方面进行独立统计。实验选取了 GPT-4o、DeepSeek Reasoner (R1) 以及 Claude-Sonnet-4.5 三款代表性模型，在包含 280 条正负样本的基准数据集上进行了全量测试。

#### 5.3.2.1 审计准确性统计

准确性是安全审计系统的核心生命线，尤其是假阳性率（False Positive Rate, FPR），其高低直接决定了运维人员对系统输出结果的信任程度。为对模型的审计能力进行量化评估，本文基于二分类安全审计任务的混淆矩阵（Confusion Matrix），定义如下统计指标。

设：

- $TP$（True Positive）：模型校验成功，放行的正确策略；
- $FP$（False Positive）：模型校验失败，放行的错误策略；
- $TN$（True Negative）：模型校验成功，禁止的错误策略；
- $FN$（False Negative）：模型校验失败，禁止的正确策略。

在此基础上，真阳性率（True Positive Rate, TPR）与假阳性率（False Positive Rate, FPR）分别定义为：

$$\mathrm{TPR} = \frac{TP}{TP + FN} \tag{5-2}$$

$$\mathrm{FPR} = \frac{FP}{FP + TN} \tag{5-3}$$

其中，TPR 用于衡量模型对真实正确业务通信需求的识别能力；FPR 则刻画模型过度授权错误规则的倾向。各模型在上述指标上的统计结果如表 5.6 所示。

**表 5.6 不同模型在策略审计任务中的准确性指标对比**

| 模型名称 | 真阳性率 (TPR) | 假阳性率 (FPR) |
|---|---|---|
| GPT-4o | 98.6% | 0.7% |
| DeepSeek Reasoner (R1) | 98.6% | 0.0% |
| Claude-Sonnet-4.5 | 100.0% | 0.0% |

实验结果表明，不同模型在网络策略审计任务中的表现存在明显差异。Claude-Sonnet-4.5 在本次评测中取得了最为稳定的结果，其在全部测试样本上均给出了正确

判断，既未出现漏报，也未引入误报，整体审计效果达到完全覆盖。该结果说明该模型在规则一致性校验与逻辑约束推理方面具备较强能力，能够准确识别隐藏于配置细节中的潜在风险。

DeepSeek R1 的整体表现同样较为突出。依托其较强的推理能力，该模型在全部负样本中未产生任何误报，保持了零误报率，仅在少数结构复杂、语义较为隐蔽的漏洞场景下出现漏检。这表明其在审计结果可靠性方面具有较高稳定性，但在覆盖极端复杂配置时仍存在一定提升空间。

相比之下，GPT-4o 虽然在漏洞检出率方面保持了较高水平，但仍出现了少量误报。对相关错误样本进行回溯分析后发现，其主要失效模式表现为数值层面的逻辑不一致未被有效识别，即模型在推理过程中倾向于生成自洽但不符合实际配置约束的判断结果。例如，在容器实际监听端口为 9091，而 NetworkPolicy 误放行 8091 的情况下，模型仍将该策略判定为正确，反映出其在精细数值一致性校验方面的不足。

#### 5.3.2.2 性能消耗与成本效益评估

除了准确性，大规模审计任务的执行效率与经济成本也是工程落地的关键考量。表 5.7 详细统计了各模型处理单条策略的平均耗时、Token 消耗量及对应的资金成本。

**表 5.7 不同模型在单条策略审计任务中的性能消耗与成本评估**

| 模型名称 | 平均耗时 (s) | 平均输入 (Tokens) | 平均输出 (Tokens) | 平均成本 ($/条) |
|---|---|---|---|---|
| GPT-4o | 133.63 | 3,128.6 | 198.9 | 0.0098 |
| DeepSeek Reasoner (R1) | 294.68 | 3,242.0 | 1,761.4 | 0.005 |
| Claude-Sonnet-4.5 | 237.03 | 18,349.8 | 3,271.8 | 0.1041 |

注：成本计算基于官方 API 定价：GPT-4o ($2.5/$10), DeepSeek ($0.55/$2.19), Claude ($3/$15) per 1M tokens。

通过对模型审计效果、执行时延与调用成本等多维指标进行综合分析，可以观察到不同方案在性能与经济性方面呈现出较为鲜明的取舍关系。

首先，DeepSeek R1 在整体成本控制方面展现出显著优势。由于引入思维链推理机制，其单次审计过程中产生的输出规模明显高于其他模型，平均 Token 数达到 1,761.4，对应的推理耗时也相对较长，单条策略分析通常需要约 5 分钟。然而，受益于较低的接口定价，其单次策略审计的实际成本仅为 0.0056 美元，在所有对比模型中处于最低水

平。该成本约为 GPT-4o 的一半左右，也显著低于 Claude 系列模型。因此，在对时效性要求不高、但需要对大量策略进行离线批量审计的场景下，DeepSeek R1 更适合作为一种以时间换取成本优势的解决方案。

其次，Claude-Sonnet-4.5 的性能特征体现出典型的高成本与高精度权衡。实验数据显示，其单条审计的平均成本达到 0.1041 美元，显著高于其他模型。进一步分析发现，该开销主要来源于其在审计过程中引入了极为完整的上下文信息，平均输入规模接近 1.8 万 Token。通过覆盖更全面的代码与依赖关系，该模型实现了最为稳定且准确的审计结果，但相应的资源消耗也随之大幅增加。因此，该模型更适合部署于对安全性与准确性要求极高的关键业务场景，例如核心金融系统或高风险生产环境。

最后，GPT-4o 在推理效率方面表现突出。其平均审计耗时约为 133.63 秒，明显快于其他对比模型，同时单次调用成本保持在 0.0098 美元的中等水平。综合速度与成本因素，该模型更适合应用于对响应时间敏感的在线审计任务，或作为 CI/CD 流水线中的自动化质量控制节点。

综合上述分析可以得出结论，引入思维链推理机制或超长上下文关联能力，均能够在不同程度上提升审计准确性，但不可避免地会带来额外的时间或经济成本。在实际工程应用中，更为可行的方案是采用分级审计策略：在常规场景下利用成本较低的模型完成全量扫描，而针对识别出的高风险策略，再引入高精度模型进行深入复核，以在安全性、效率与成本之间实现平衡。

### 5.3.3 模型解释能力分析

除了准确率与响应时延等定量指标外，大语言模型在输出解释报告时的逻辑深度、证据颗粒度以及对云原生架构的洞察力，直接决定了其在“人在回路（Human-in-the-Loop）”审计场景下的可用性。本节特别选出开源项目 JupyterHub 中最为核心的 hub 组件网络策略这一例子作为典型案例，来对比分析了 GPT-4o、DeepSeek Reasoner (R1) 与 Claude-Sonnet-4.5 三种模型在执行逆向解释任务时的具体表现。

#### 5.3.3.1 案例背景与分析

该案例选取自 JupyterHub 项目的 hub 微服务 Pod。该组件负责管理用户生命周期及代理路由，其网络行为具有高度复杂性：既需要 Ingress 方向接收 Proxy 的 API 请求，又

需要 Egress 方向管理 singleuser-server（单用户容器）及连接外部数据库。

#### 5.3.3.2 解释能力的维度对比

该案例选取自 JupyterHub 项目的 hub 微服务 Pod。作为系统的控制中枢组件，hub 负责用户生命周期管理及代理路由控制，其网络行为呈现出显著的复杂性：在入站方向需要接收来自 Proxy 组件的 API 请求，在出站方向则既要调度 singleuser-server（单用户 Notebook 容器），又需要访问外部数据库与基础设施服务。这一特性使其成为检验模型策略解释能力的代表性样本。

为系统性比较不同模型的解释质量，本文从证据溯源的颗粒度、逻辑完备性以及架构层面的洞察力三个维度，对模型输出文本进行了语义层面的分析。

在证据溯源的颗粒度方面，三种模型呈现出明显的能力分层。GPT-4o 的解释方式以“直接映射”为主，能够将配置项与网络规则建立起一阶对应关系。例如，该模型正确识别了配置项 c.JupyterHub.hub_bind_url 与 Ingress 方向 8081 端口之间的绑定关系。然而，其证据链整体较为扁平，主要停留在配置存在性的层面，缺乏对配置来源、上下文以及多重验证路径的进一步挖掘。相比之下，DeepSeek Reasoner（R1）展现出更强的结构化溯源能力。其不仅引用了核心配置文件内容，还进一步关联了运行时环境变量（如 HUB_SERVICE_PORT=8081），通过配置与环境的交叉印证来支撑规则生成。这种多源证据并行的方式显著提升了策略解释的可核验性。Claude-Sonnet-4.5 则进一步上升到了语义层面的证据表达。除端口与配置匹配关系外，它还解释了标签选择器的合理性，例如明确指出 Proxy Pod 的标签为何能够被 Ingress 规则命中，并结合 KubeSpawner 的配置，论证 Hub 组件对 singleuser-server 发起出站访问的必然性，从而形成完整的语义闭环。

在逻辑完备性方面，不同模型在覆盖范围上的差异更加直观。GPT-4o 在该案例中出现了关键遗漏，其未能准确识别 Hub 组件对 singleuser-server（端口 8888）的出站访问需求，而是以较为笼统的“外部 IP 访问”进行概括。这一缺失直接导致解释结果在网络依赖维度上不完整，存在潜在的误导风险。相比之下，DeepSeek R1 与 Claude-Sonnet-4.5 均成功识别并覆盖了 Hub Pod 的全部出站需求，包括 Proxy API（8001）、singleuser-server（8888）、DNS（53）以及外部网络访问等四类通信模式。其中，DeepSeek 的优势体现在输出结构的规范性上，其对每一条 Egress 规则均明确标注了对应的功能组件，使得结果

更易被自动化工具或后续流程消费。

在架构洞察力与容错性这一更高层次的维度上，模型之间的差距进一步拉大。GPT-4o 在解释过程中曾出现“proxy 容器需要……”等表述，暴露出其对 Pod 内部组件边界的理解并不完全清晰，未能准确区分 Hub Pod 与独立 Proxy 组件之间的策略作用范围。这种混淆在实际运维中可能引发策略误用。Claude-Sonnet-4.5 则体现出了接近人工安全审计员的架构理解能力。它明确指出，尽管分析过程中提及 Proxy 相关行为，但 Proxy 容器本身并不受当前 NetworkPolicy 管控，这一事实反而表明应当为 Proxy 组件单独设计并部署独立的 NetworkPolicy。此类从整体架构层面出发的“防御性解释”，能够有效避免运维人员因误解策略边界而引入新的配置风险。

总体而言，该案例清晰地展示了不同模型在策略解释能力上的层级差异，也印证了高阶语义理解与架构感知能力在复杂微服务场景中的实际价值。

5.3.3.3 综合评价

通过对比分析针对之前全部策略的解释结果，表 5.8 总结了三种模型在对策略定性分析中的综合表现。

**表 5.8 不同大模型在微隔离策略解释任务中的定性能力对比**

| 评价维度 | GPT-4o | DeepSeek Reasoner (R1) | Claude-Sonnet-4.5 |
|---|---|---|---|
| **证据深度** | 浅层（配置） | 中层（配置 + 环境变量） | 深层（代码逻辑 + 架构语义） |
| **规则召回** | 存在遗漏 | 完全覆盖 | 完全覆盖 |
| **架构理解** | 局部视角 | 模块化视角 | 全局系统视角 |
| **输出风格** | 简洁摘要 | 结构化 JSON/List | 详尽的审计报告 |
| **适用场景** | 快速预览 | 自动化合规检查 | 复杂故障排查与人工审计 |

## 5.4 大模型正向生成实验

### 5.4.1 数据集定义

为验证第四章提出的大模型正向生成机制在复杂基础设施环境及隐式服务依赖场景中的有效性，本文在实验设计中继续采用第 5.3 节（表 5.5）所定义的数据集。该数据

集汇集了多个具有代表性的核心开源项目，能够较好地反映真实生产环境中基础设施组件之间的复杂交互关系。

选择该数据集用于正向生成实验，主要基于其在依赖表达形式上的复杂性与评测基准构建上的可控性。一方面，数据集中的多个关键组件之间存在较为隐蔽的依赖关系，例如 Airflow 中 scheduler 与 worker 之间的协作机制，以及 TiDB 体系中 PD 与 TiKV 的交互方式。这类系统广泛采用动态配置加载、自定义 RPC 通信协议以及非标准端口绑定策略，使得服务依赖并未以显式、固定的形式体现在配置文件或接口声明中，从而构成了传统确定性分析方法难以覆盖的语义盲区。正是由于这些特性，该数据集为检验基于大模型的 DeepWiki 引擎在深层语义理解与代码意图挖掘方面的能力提供了理想测试场景。

另一方面，为了对正向生成结果进行量化评估，实验在真值标准的构建上采用了严格的人工审计流程。不同于依赖运行时流量作为参考基线的评估方式，本文针对选定的服务组件，通过对白盒源代码以及官方架构文档的系统分析，人工编写了一组符合最小权限原则的 NetworkPolicy 作为“黄金标准”策略集。在实验过程中，系统依据代码语义自动生成的策略 YAML 文件将与该真值集合进行逐项比对，从而计算生成结果在精确率与召回率等指标上的表现。

综合上述设计，本实验的目标在于验证基于大模型的方法是否仍能够准确理解服务代码所含的完整通信意图，并生成满足最小权限约束的安全策略。

### 5.4.2 实验结果统计与性能分析

#### 5.4.2.1 实验结果统计

为了评估系统生成策略的质量，本实验摒弃了生成耗时等次要指标，聚焦于安全防护最核心的两个维度：策略覆盖率（衡量是否覆盖了所有必要的业务通信）与策略冗余度（衡量是否生成了多余的非必要规则）。

基于零信任微隔离“默认拒绝、最小授权”的安全设计原则，本文定义以下两个量化指标，用于评估大模型生成 NetworkPolicy 的有效性与安全性。

- 策略覆盖率（Policy Coverage Rate, PCR）：该指标用于衡量生成策略对真实业务通信需求的覆盖程度，反映模型在策略生成过程中的召回能力。设 $\mathcal{P}_{gt}$ 表示通过人

工审计得到的真实通信需求集合（Ground Truth），$\mathcal{P}_{gen}$ 表示模型自动生成的策略集合，则策略覆盖率定义为：

$$\mathrm{PCR} = \frac{|\mathcal{P}_{gen} \cap \mathcal{P}_{gt}|}{|\mathcal{P}_{gt}|} \tag{5-4}$$

PCR 越高，说明生成策略对实际业务依赖的覆盖越完整，由策略缺失导致业务被误阻断的风险越低。

- 策略冗余度（Policy Redundancy Rate, PRR）：该指标用于衡量生成策略中包含的非必要或过度宽松规则的比例，用以反映模型在策略生成过程中的精确性与收敛程度。在相同符号约定下，策略冗余度定义为：

$$\mathrm{PRR} = \frac{|\mathcal{P}_{gen} \setminus \mathcal{P}_{gt}|}{|\mathcal{P}_{gen}|} \tag{5-5}$$

PRR 越低，表示生成的网络白名单越精简，潜在攻击面越小，更符合最小授权原则下的安全设计目标。

通过对比了 GPT-4o、DeepSeek Reasoner (R1) 与 Claude-Sonnet-4.5 三款模型在上述两个核心指标上的表现。实验统计结果如表 5.9 所示。

从加权平均指标（Weighted Avg）来看，三种模型在整体表现上均体现出较高的工业可用性，其策略覆盖率（Policy Coverage Rate, PCR）均稳定维持在 98.8% 以上，说明所生成的安全策略在大多数场景下能够较为完整地刻画服务间的通信需求。其中，Claude-Sonnet-4.5 的综合表现最为突出，其在全部测试项目中实现了 100.0% 的覆盖率，同时冗余度仅为 1.2%。这一结果表明，该模型在处理复杂服务依赖关系时具备较强的全局推理能力，能够在保证“零漏报”的同时，将不必要的策略放行控制在较低水平。DeepSeek-R1 的整体表现与 Claude 接近，其加权平均 PCR 达到 99.4%，且在绝大多数项目中保持了相似的低冗余特征，显示出较为稳定的策略生成质量。相比之下，GPT-4o 的总体覆盖率同样处于较高水平（98.8%），但其生成策略的冗余度相对偏高，达到 2.6%，反映出该模型在决策过程中更倾向于通过放宽约束来避免漏检，从而呈现出一定的保守生成策略特征。

进一步从具体项目分布来看，在 Airflow、TiDB 等前十个以标准架构为主的系统中，三种模型均实现了 100% 的覆盖率，说明在依赖关系清晰、架构模式成熟的场景下，不同

**表 5.9 多模型在全量实验项目集上的正向策略生成指标统计 (PCR: 覆盖率 / PRR: 冗余度)**

| | | GPT-4o | | DeepSeek-R1 | | Claude-Sonnet-4.5 | |
|---|---|---|---|---|---|---|---|
| 序号 | 应用系统 | PCR | PRR | PCR | PRR | PCR | PRR |
| 1 | Airflow | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 2 | Devlake | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 3 | TiDB | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 4 | Druid | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 5 | Pinot | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 6 | JupyterHub | 100.0% | 13.3% | 100.0% | 0.0% | 100.0% | 0.0% |
| 7 | RuoYi-Cloud | 100.0% | 0.0% | 100.0% | 3.3% | 100.0% | 0.0% |
| 8 | Yudao-Cloud | 100.0% | 0.0% | 100.0% | 3.6% | 100.0% | 0.0% |
| 9 | Reactive-Shopping | 100.0% | 0.0% | 100.0% | 5.9% | 100.0% | 2.5% |
| 10 | Killbug | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 11 | Online-Boutique | 94.1% | 15.0% | 100.0% | 10.0% | 100.0% | 15.0% |
| 12 | Bank of Anthos | 95.7% | 0.0% | 95.7% | 0.0% | 100.0% | 0.0% |
| 13 | Microservices-K8s | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 14 | Robot Shop | 100.0% | 18.2% | 100.0% | 0.0% | 100.0% | 0.0% |
| 15 | eShopOnContainers | 97.5% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 16 | Sock Shop | 90.0% | 0.0% | 90.0% | 0.0% | 100.0% | 0.0% |
| 17 | Milvus | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 18 | Vitess | 100.0% | 0.0% | 100.0% | 0.0% | 100.0% | 0.0% |
| 总体平均 (Weighted Avg) | | 98.8% | 2.6% | 99.4% | 1.8% | 100.0% | 1.2% |

模型之间的性能差异并不显著。模型能力的区分主要体现在架构更为复杂、依赖表达更为隐式的微服务示例项目中。例如，在 Sock Shop 项目中，GPT-4o 与 DeepSeek-R1 的覆盖率均下降至 90.0%。经人工审计发现，遗漏部分主要集中在 front-end 服务对 catalogue 服务所采用的非标准 HTTP 调用路径上，该调用关系未以集中式配置形式显式呈现，增加了语义关联难度。相比之下，Claude-Sonnet-4.5 借助其更大的上下文建模能力，成功整合了分散于多个配置文件中的路由与依赖信息，从而弥补了这一分析盲区。

类似的差异也体现在 Online-Boutique 项目（编号 11）中。GPT-4o 在该场景下的覆盖率为 94.1%，未能达到完全覆盖。结合调试日志与生成结果分析可以发现，其主要问题在于未能正确解析 artservice 中基于 Redis Sentinel 的动态服务发现机制，导致部分从节点之间的通信规则未被纳入生成策略中。这一结果进一步说明，在涉及复杂中间件机制与动态依赖解析的场景下，模型对上下文关联与隐式语义的建模能力，将直接影响其策略生成的完整性。

冗余度指标揭示了模型在面对“不确定性”时的决策边界。

- Online-Boutique 关联性幻觉：这是本实验中最具代表性的案例，所有模型在该项目中都产生了冗余策略，这一现象揭示了大模型在处理多组件复杂上下文时典型的“关联性幻觉”，Online-Boutique 包含 10 个以上微服务组件，导致 Prompt 上下文中充斥着大量的跨服务环境变量与配置片段。大模型在面对如此高密度的信息时，未能有效隔离不同 Pod 的代码上下文边界。例如，它错误地将组件 redis-cart “过度关联”给了并未实际调用的组件，产生了一条并不存在的依赖规则。

- Robot Shop 的极端分化：这是本实验中最具代表性的案例。GPT-4o 在该项目中产生了高达 18.2% 的冗余策略，而 DeepSeek-R1 与 Claude-3.5-Sonnet 均为 0.0%。通过对比推理日志发现，Robot Shop 混合使用了 Python、Node.js 和 Java，且部分服务未显式声明端口。GPT-4o 在无法确信时，倾向于开放整个子网段或常用端口（如 8080/80）以保证连通性；而 DeepSeek-R1 利用思维链（Chain-of-Thought）进行了多轮否定推理，明确排除了非必要端口；Claude-Sonnet-4.5 则在 JSON 的 security_recommendations 字段中明确指出“未发现明确网络依赖，建议实施默认阻断”，从而避免了冗余规则的生成。

#### 5.4.2.2 性能分析

在云原生微隔离策略的工程化落地中，除了策略生成的准确性外，系统的响应时效性与经济成本效益比（Cost-Performance Ratio）同样是决定其能否大规模商用的关键指标。本节基于实验记录的详细日志，对 GPT-4o、DeepSeek Reasoner (R1) 与 Claude-Sonnet-4.5 三款模型在执行策略生成任务时的平均资源消耗进行了量化评估。统计结果如表 5.10 所示。

**表 5.10 多模型策略生成任务的单 Pod 平均资源消耗与成本统计**

| 模型名称 | 平均耗时 (s) | 平均输入 (Tokens) | 平均输出 (Tokens) | 平均成本 ($) |
|---|---|---|---|---|
| GPT-4o | 40.0 | 3,048 | 480 | 0.012 |
| DeepSeek-R1 | 299.3 | 2,959 | 2,034 | 0.006 |
| Claude-Sonnet-4.5 | 118.3 | 14,966 | 1,973 | 0.074 |

从平均耗时来看，GPT-4o 展现了显著的工程化优势。其单 Pod 平均处理时间仅为 40.0 秒，这意味着在并发处理能力充足的情况下，系统能够以分钟级的速度完成大规模集群的策略基线构建。这种快速响应能力使其非常适合应对紧急扩容或故障恢复场景下的策略下发。

相比之下，DeepSeek-R1 的平均耗时最长，达到 299.3 秒（约 5 分钟）。这与其内置的思维链（Chain-of-Thought）机制直接相关：模型在生成最终策略前，需要进行大量的逻辑推演。虽然牺牲了部分时效性，但前文实验证明这种“慢思考”机制有效提升了策略的收敛度。

在经济成本维度，各模型表现出了截然不同的定价策略与资源消耗模式：

DeepSeek-R1 的极致性价比：尽管 DeepSeek-R1 的平均输出 Token 量较高（2,034 Tokens，包含思维链内容），但得益于其极低的 API 单价，其单 Pod 生成成本仅为 0.006 美元（约合人民币 0.04 元）。这意味着企业仅需极低的预算即可实现全量微服务的精细化防护，具备极高的规模化推广潜力。

Claude 的“全上下文”策略代价：Claude-Sonnet-4.5 的单 Pod 成本最高，达到 0.074 美元，是 DeepSeek 的 12 倍以上。数据分析显示，这主要是由于其平均输入 Token 高达 14,966 个。为了追求 100% 的覆盖率，系统在调用 Claude 时加载了完整的代码仓库上下

文而非摘要。这种高成本投入换取了最高的准确率，适合应用于对安全要求极高的核心支付类业务。

GPT-4o 的均衡选择：GPT-4o 以 0.012 美元的适中成本提供了最快的生成速度。其输入输出比例最为精简，表明该模型能够通过较短的 Prompt 高效理解任务意图，是平衡性能与预算的通用选择。

综上所述，工程实践中建议采用分级治理策略：对于绝大多数非核心微服务，采用 DeepSeek-R1 进行低成本的夜间批量计算；对于核心金融业务，采用 Claude-Sonnet-4.5 进行高精度覆盖；而对于应急响应场景，则优先使用 GPT-4o 进行快速阻断。

综合准确率、时效性与成本三个维度，本文提出如下工程选型建议：

- 日常运维与冷启动：推荐使用 **DeepSeek-R1**。虽然耗时较长，但其 $0.006/Pod 的低廉成本与 98.6% 的高覆盖率，非常适合在夜间批量执行全量策略生成任务。
- 应急响应与扩容：推荐使用 **GPT-4o**。其分钟级的响应速度能迅速填补新上线服务的防护空白。
- 核心金融级审计：推荐使用 **Claude-Sonnet-4.5**。在不计成本追求 0 误报与 100% 覆盖的核心支付场景（如 `epay`, `anthos-bank`）中，其全上下文推理能力是不可替代的。

subsection 模型证据链分析

在零信任微隔离的工程落地中，策略的“可解释性（Explainability）”十分重要。运维人员通常面临“敢开通不敢阻断”的困境，根本原因在于缺乏足够的证据来证明“某条流量确实是不需要的”。

本节基于模型输出的证据链文件，重点对比了 GPT-4o、DeepSeek-R1 与 Claude-Sonnet-4.5 在策略生成过程中的证据溯源（Evidence Sourcing）能力。

#### 5.4.2.3 证据粒度分级评估

通过对全量实验日志的审计，我们将模型的证据能力划分为三个层级：表层配置级、结构化溯源级与代码语义级。各模型的典型表现如表 5.11 所示。

**表 5.11 多模型策略生成证据的质量与粒度分级对比**

| 模型 | 证据层级 | 典型证据模式 (Pattern) | 可信度评级 |
| --- | --- | --- | --- |
| GPT-4o | **表层配置级** (Surface) | 仅提取显式的环境变量键值对，缺乏上下文关联，无法解释隐式依赖。 | ⋆⋆ |
| DeepSeek-R1 | **结构化溯源级** (Structured) | 精确引用配置文件路径、JSON/YAML 结构片段，逻辑链路清晰，适合自动化验证。 | ⋆⋆⋆⋆ |
| Claude-3.5 | **代码语义级** (Semantic) | 深入代码实现细节，结合类名、库函数、架构模式（如“内存存储”）进行深度归因。 | ⋆⋆⋆⋆⋆ |

5.4.2.4 典型案例：从“配置读取”到“意图理解”的跨越

为更直观地刻画不同模型在证据层级与解释深度上的差异，本文以 Anthos Bank 项目中的两个代表性服务为例，从正向依赖与负向阻断两个方面展开微观对比分析。

在正向依赖场景中，以 userservice 对 accounts-db 的数据库访问需求为例，三种模型均能够生成语义正确的放行策略，但其所提供的证据链条厚度存在显著差别。GPT-4o 在解释过程中主要停留在配置层面，其给出的证据仅包含相关环境变量声明，例如 ACCOUNTS_DB_URI 的定义。这类证据只能表明配置项在运行环境中客观存在，却无法进一步证明业务代码是否真实使用了该配置，从而在因果链条上仍存在断层。相比之下，DeepSeek-R1 在证据组织上更进一步，不仅引用了环境变量本身，还补充了对应的配置文件路径与结构化片段，明确指出该配置是如何被注入至 userservice 容器之中的。这种以配置来源为核心的溯源方式，有助于下游审计或自动化校验工具对证据进行快速索引与验证。Claude-Sonnet-3.5 则展现出更高层次的语义理解能力，其解释直接上升至代码层面，明确指出具体的 Java 类在运行过程中读取 ACCOUNTS_DB_URI 并据此建立与 PostgreSQL 数据库的连接关系。通过将配置项与业务实现类进行绑定，该模型构建了一条从环境配置到代码行为的完整证据闭环，显著增强了审计结论的说服力。

在负向阻断场景中，问题则转变为如何为不存在依赖关系提供充分论证。以 adservice 这一不依赖外部数据库的纯计算型服务为例，GPT-4o 与 DeepSeek-R1 均给出了较为保守的解释方式，其结论主要体现为未发现任何出站依赖，或以空的 Egress 列表作

为佐证。这类表述在形式上虽不错误，但从运维视角来看，容易被理解为模型能力不足导致的未检出，而非对确实不存在依赖的正向确认。Claude-Sonnet-3.5 在该场景下则给出了更具解释力的负向证据，其在安全建议中明确指出广告数据存储于内存结构 ImmutableListMultimap 中，不涉及任何外部持久化数据库，同时结合服务职责说明该容器仅对外提供入站 gRPC 接口，并不会主动发起出站连接。通过识别具体的数据结构并推断其内存型存储特性，模型从实现原理层面对无外部依赖这一结论进行了论证，有效消除了潜在的隐式访问疑虑，为实施严格的默认拒绝网络策略提供了可靠依据。

#### 5.4.2.5 分析总结

实验结果表明，Claude-Sonnet-4.5 在当前对比模型中是唯一能够稳定提供代码级审计证据的模型，其分析能力已超越基于规则或模式匹配的浅层判断，进入对程序语义进行理解与验证的层面。该模型不仅给出策略结论，还能够通过具体实现细节对结论加以佐证，呈现出接近人工代码审计员的分析特征。

在实际工程场景中，这种能力具有明确且可感知的价值。例如，在数据库访问审计过程中，Claude-Sonnet-4.5 能够直接指出 UserDb 等具体 Java 类在运行时读取环境变量并建立数据库连接的事实，从而将网络策略与业务代码中的真实调用路径一一对应。这种以具体类名或实现位置为依据的解释方式，使运维与安全人员能够快速核实策略合理性，有效缓解因模型误判而带来的误报焦虑。

另一方面，在对计算型服务进行分析时，Claude-Sonnet-4.5 还能够识别出底层采用内存数据结构实现的架构模式。例如，在审计广告服务相关组件时，模型明确指出其核心数据存储依赖于 StackExchange.Redis 或内存型集合结构，而非外部持久化数据库，并进一步结合服务职责说明其仅对外提供入站接口、不主动发起出站通信。这类基于实现机制给出的负向证据，使安全团队能够在关键组件上更有把握地实施零出站（Zero Egress）等激进防御策略，而不必担心潜在的隐式依赖。

相比之下，DeepSeek-R1 在证据表达的语义深度上虽然不及 Claude-Sonnet-4.5，但其输出形式具有较强的工程友好性。该模型通常以结构化的 JSON 片段呈现配置来源、依赖关系与策略依据，字段定义明确且格式稳定，便于在持续集成与持续交付（CI/CD）流水线中作为中间分析模块使用，用于自动生成合规性检查结果或审计报告。

## 5.5 本章小结

本章通过在经典开源基础设施组件的异构 Kubernetes 集群上进行实验，全面验证了本文提出的零信任微隔离策略生成框架的有效性。实验结果表明，动静态协同分析引擎在标准场景下实现了极高的基础策略生成准确率，且处理时延满足云原生环境的实时性需求。同时，大模型驱动的智能化模块在策略审计与正向生成任务中表现卓越，不仅实现了极高的逻辑错误识别率与极低的假阳性率，更成功解决了服务冷启动阶段的防护真空问题，验证了该方案在提升微隔离策略准确性与可解释性方面的工程价值。

# 6 总结与展望

## 6.1 工作总结

本文针对云原生架构下微服务集群的安全治理难题，深入探讨了零信任微隔离策略生成的理论支撑与工程实践。在微服务架构日益普及的背景下，其极致的动态性与多语言异构性，使得传统基于边界防御和被动流量学习的防御体系逐渐失效。本研究的核心价值在于，提出了一套融合“确定性程序分析”与“生成式语义推理”的内生安全防御体系，通过两套互补的技术方案，有效解决了现有研究中普遍存在的策略生成滞后、覆盖率不足以及缺乏可解释性等关键问题。

本文首先针对微服务基础通信关系的构建，提出了基于图结构的动静态协同分析方法。这一方案主要适用于以 Java（如 Spring Cloud 生态）为代表的标准化微服务架构场景。相较于相关研究中主流的“被动流量学习”技术，本方法的显著优势在于其极高的执行效率与近乎零的边际成本——它能够在不消耗昂贵算力资源的前提下，快速完成集群的拓扑提取。通过将 Kubernetes 运行时的瞬时状态与静态字节码中的长效逻辑进行同构映射，该方法能够精准还原服务间的调用拓扑，为提供了高置信度的策略生成。然而，受限于静态分析技术的语言依赖性，该方法在面对非 Java 语言或高度动态的脚本语言环境时存在一定的适用性局限。

为了突破单一技术栈的限制并解决复杂异构场景下的语义理解难题，本文构建了基于大语言模型增强的策略生成与审计框架。这一方案作为动静态分析的有力补充，主要适用于多语言混合（如 Go、Python、Node.js 等）的复杂异构场景。虽然大模型的引入带来了相对较高的算力成本与推理时延，但其展现了卓越的通用性与语义理解能力。它不仅能够跨越编程语言的障碍，深入理解代码层面的业务意图与隐式约束，还能解析非标协议与复杂的架构模式。其核心价值在于实现了从“语法匹配”到“语义理解”的跨越，不仅填补了传统静态工具在多语言支持上的空白，还能提供包含代码级证据的可解释性溯源，解决了零信任落地中运维人员因“不敢阻断”而导致的工程僵局。

综上所述，本文通过“动静态协同分析”与“大模型语义增强”的双轨驱动，构建了完整的零信任微隔离技术闭环。前者作为“低成本、高时效”的基础设施，负责在 Java

等标准场景下提供坚实的确定性策略；后者作为“高智能、全覆盖”的增强引擎，负责在多语言异构场景下注入深度的语义理解。两者优势互补，有效地将微隔离策略的生成模式从“基于流量观测的被动适应”转变为“基于代码意图的主动构建”，为云原生环境下的零信任安全落地提供了切实可行的理论依据与技术方案。

## 6.2 未来展望

尽管本文在自动化策略生成领域取得了一定成果，但云原生安全防御的演进永无止境，未来的研究仍有广阔的探索空间。随着 eBPF 等内核技术的成熟与大模型逻辑推理能力的进一步提升，微隔离防护将从单纯的“流量管控”向“意图自治”方向深度迈进。未来的安全体系应当具备更深层次的协议洞察能力，能够穿越加密流量与复杂 RPC 框架，实现应用层语义的实时解构与风险感知。

策略的自愈能力将成为下一个研究重点。在持续集成与持续部署的流水线中，安全策略不应是静态的基线，而应是随代码变更而自动漂移的“流体”。探索如何利用大模型实时监听代码库的微小变动，并自动触发毫秒级的策略微调，将彻底解决安全配置滞后于业务迭代的顽疾。此外，跨云一致性与策略互操作性也是亟待解决的问题。在多云混合架构成为主流的趋势下，如何构建一套不依赖于特定网络插件的标准化策略语言，实现“一次意图提取，全球拓扑分发”，将具有极大的行业应用前景。

最后，大模型本身的性能开销与对抗性安全性亦不容忽视。在追求推理精度的同时，如何通过模型蒸馏与领域知识对齐，降低系统在生产环境中的资源占用，是工程化落地的关键。同时，利用人工智能技术对现有策略进行模拟攻防探测，构建基于“红队思维”的自适应加固机制，将进一步提升系统在零日攻击场景下的生存能力。我们相信，随着程序分析技术与人工智能的持续融合，云原生环境将真正实现安全与敏捷的深度平衡，构建起一个透明、智能且具备自我进化能力的零信任安全生态。

# 参考文献


[1] Cloud Native Computing Foundation. CNCF Annual Report 2024[R]. Annual Report. Available at: https://www.cncf.io/reports/. San Francisco, CA: The Linux Foundation, 2024.

[2] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, Omega, and Kubernetes[J/OL]. Communications of the ACM, 2016, 59: 50-57. https://api.semanticscholar.org/CorpusID:13246959.

[3] HILS A, KAUR R, D'HOINNE J. Market Guide for Microsegmentation[Z]. Document ID: 4435299. Available at: https://www.gartner.com/en/documents/4435299. 2023.

[4] WARD R, BEYER B. BeyondCorp: A New Approach to Enterprise Security[J/OL]. login Usenix Mag., 2014, 39. https://api.semanticscholar.org/CorpusID:56594397.

[5] ROSE S, BORCHERT O, MITCHELL S, et al. Zero Trust Architecture[C/OL]//. 2019. https://api.semanticscholar.org/CorpusID:212765583.

[6] LI W, LEMIEUX Y, GAO J, et al. Service Mesh: Challenges, State of the Art, and Future Research Opportunities[J/OL]. 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019: 122-1225. https://api.semanticscholar.org/CorpusID:146119227.

[7] CHANDRAMOULI R. Attribute-based Access Control for Microservices-based Applications Using a Service Mesh[C/OL]//NIST Special Publication 800-204B. 2021. https://api.semanticscholar.org/CorpusID:238984004.

[8] JAMSHIDI P, PAHL C, das CHAGAS MENDONÇA N, et al. Microservices: The Journey So Far and Challenges Ahead[J/OL]. IEEE Softw., 2018, 35: 24-35. https://api.semanticscholar.org/CorpusID:25437582.

[9] Isovalent Inc. eBPF-based Networking, Observability, and Security[R]. Whitepaper. Available at: https://isovalent.com/resource-library/. Isovalent, 2023.

[10] ZHOU X, PENG X, XIE T, et al. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study[J/OL]. IEEE Transactions on Software Engineering, 2018, 47: 243-260. https://api.semanticscholar.org/CorpusID:57462883.

[11] CHEN X, ZHANG M, MAO Z M, et al. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions[C/OL]//USENIX Symposium on Operating Systems Design and Implementation. 2008. https://api.semanticscholar.org/CorpusID:7600750.

[12] HE K, KIM D D, ASGHAR M R. Adversarial Machine Learning for Network Intrusion Detection Systems: A Comprehensive Survey[J/OL]. IEEE Communications Surveys & Tutorials, 2023, 25: 538-566. https://api.semanticscholar.org/CorpusID:255718190.

[13] Sysdig Threat Research Team. Sysdig 2024 Cloud-Native Security and Usage Report[R]. Industry Report. Available at: https://sysdig.com/blog/2024-cloud-native-security-and-usage-report/. Sysdig, 2024.

[14] GNUTTI A, VACA F D, FACCHI C. GCN-Based Encrypted Traffic Classification: A Representation Learning Approach[J/OL]. IEEE Transactions on Network and Service Management, 2022, 19(4): 5074-5086. DOI: 10.1109/TNSM.2022.3204369.

[15] COHEN O S, MALUL E, MEIDAN Y, et al. KubeGuard: LLM-Assisted Kubernetes Hardening via Configuration Files and Runtime Logs Analysis[J/OL]. ArXiv, 2025, abs/2509.04191. https://api.semanticscholar.org/CorpusID:281103555.

[16] KIM B, LEE S. KubeAegis: A Unified Security Policy Management Framework for Containerized Environments[J/OL]. IEEE Access, 2024, 12: 160636-160652. https://api.semanticscholar.org/CorpusID:273723001.

[17] SANDHU R S, COYNE E J, FEINSTEIN H L, et al. Role-Based Access Control Models[J/OL]. Computer, 1996, 29: 38-47. https://api.semanticscholar.org/CorpusID:1958270.

[18] FERRAIOLO D F, SANDHU R, GAVRILA S, et al. Proposed NIST Standard for Role-Based Access Control[J/OL]. ACM Transactions on Information and System Security (TISSEC), 2001, 4(3): 224-274. DOI: 10.1145/501978.501980.

[19] ROSTAMI G. Role-based Access Control (RBAC) Authorization in Kubernetes[J/OL]. J. ICT Stand., 2023, 11: 237-260. https://api.semanticscholar.org/CorpusID:261800733.

[20] KUHN D R, COYNE E J, WEIL T R. Adding Attributes to Role-Based Access Control[J/OL]. Computer, 2010, 43: 79-81. https://api.semanticscholar.org/CorpusID:17866775.
[21] Red Hat. The State of Kubernetes Security Report: 2024 Edition[R]. Industry Report. Available at: https://www.redhat.com/en/engage/state-kubernetes-security-report-2024. Red Hat, 2024.
[22] GU Y, TAN X, ZHANG Y, et al. EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications[J/OL]. 2025 IEEE Symposium on Security and Privacy (SP), 2025: 3199-3217. https://api.semanticscholar.org/CorpusID:274772137.
[23] HU V C, FERRAIOLO D F, KUHN R, et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations[C/OL]//. 2014. https://api.semanticscholar.org/CorpusID:168659974.
[24] LI X, CHEN Y, LIN Z, et al. Automatic policy generation for inter-service access control of microservices[C]//USENIX Security Symposium. 2021: 3971-3988.
[25] GHAVAMNIA S, PALIT T, BENAMEUR A, et al. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction[C/OL]//International Symposium on Recent Advances in Intrusion Detection. 2020. https://api.semanticscholar.org/CorpusID:220778345.
[26] CERNÝ T, TAIBI D. Microservice-Aware Static Analysis: Opportunities, Gaps, and Advancements [J/OL]. 2022. https://api.semanticscholar.org/CorpusID:266363693.
[27] ABDELFATTAH A S, CORDES K E, MEDINA A, et al. Semantic Dependency in Microservice Architecture[J/OL]. 2025 IEEE/ACM 22nd International Conference on Software and Systems Reuse (ICSR), 2025: 44-54. https://api.semanticscholar.org/CorpusID:275758371.
[28] Tigera. Zero Trust Network: Why It’s Important & Zero Trust for K8s[EB/OL]. 2024 [2025-11-28]. https://www.tigera.io/learn/guides/zero-trust/zero-trust-network/.
[29] RAHAMAN M S, ISLAM A, CERNÝ T, et al. Static-Analysis-Based Solutions to Security Challenges in Cloud-Native Systems: Systematic Mapping Study[J/OL]. Sensors (Basel, Switzerland), 2023, 23. https://api.semanticscholar.org/CorpusID:256647815.
[30] Cybersecurity and Infrastructure Security Agency. Zero Trust Maturity Model (Version 2.0)[R]. Guidance Document. Accessed: 2025-01-10. CISA, 2024.
[31] AccuKnox. KubeArmor: Runtime Security for Cloud-Native Workloads[Z]. AccuKnox Technical Whitepaper. Accessed: 2025-02-01. 2024.
[32] Palo Alto Networks. The State of Cloud-Native Security 2024[R]. Industry Report. Available at: https://www.paloaltonetworks.com/state-of-cloud-native-security. Palo Alto Networks, 2024.
[33] Cloud Native Computing Foundation. CNCF Annual Report 2023[R]. Annual Report. Published January 2024. Available at: https://www.cncf.io/reports/cncf-annual-report-2023/. San Francisco, CA: The Linux Foundation, 2024.
[34] Axiomatics. The State of Authorization 2022[Z]. Industry Survey Report: The Road to Zero Trust. Available at: https://www.axiomatics.com/resources/the-state-of-authorization-2022/. 2022.
[35] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing v4.0[R]. Guidance Document. Available at: https://cloudsecurityalliance.org/artifacts/security-guidance-v4. Seattle, WA: Cloud Security Alliance, 2017.
[36] 3GPP. Security architecture and procedures for 5G System (Release 17)[R]. Technical Specification TS 33.501. Available at: https://portal.3gpp.org/. 3rd Generation Partnership Project (3GPP), 2022.
[37] LI X, LENG X, CHEN Y. Securing Serverless Computing: Challenges, Solutions, and Opportunities [J/OL]. IEEE Network, 2021, 37: 166-173. https://api.semanticscholar.org/CorpusID:235195813.
[38] Palo Alto Networks. State of Cloud-Native Security 2023[R]. Industry Report. Available at: https://www.paloaltonetworks.com/state-of-cloud-native-security. Palo Alto Networks, 2023.
[39] Datadog. State of Cloud Security 2024[R]. Industry Report. Available at: https://www.datadoghq.com/state-of-cloud-security/. Datadog, 2024.
[40] CrowdStrike. Architecture Drift: What It Is and How It Leads to Breaches[Z]. CrowdStrike Cybersecurity Blog. Accessed: 2025-02-15. 2024.
[41] ANDRESINI G, PENDLEBURY F, PIERAZZI F, et al. INSOMNIA: Towards Concept-Drift Robustness in Network Intrusion Detection[J/OL]. Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security, 2021. https://api.semanticscholar.org/CorpusID:240001863.
[42] State Administration for Market Regulation of PRC. GB/T 22239-2019: Information Security Technology - Baseline for Classified Protection of Cybersecurity[Z]. National Standard of P.R. China.

2019.
[43] ROSE S, BORCHERT O, MITCHELL S, et al. Zero Trust Architecture[R/OL]. 800-207. National Institute of Standards, 2020. DOI: 10.6028/NIST.SP.800-207.
[44] YARYGINA T, BAGGE A H. Overcoming Security Challenges in Microservice Architectures[J/OL]. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018: 11-20. https://api.semanticscholar.org/CorpusID:21718508.
[45] HU V C, FERRAIOLO D, KUHN R, et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations[R/OL]. 800-162. National Institute of Standards, 2014. DOI: 10.6028/NIST.SP.800-162.
[46] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot - a Java Bytecode Optimization Framework[C]//Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. 1999.
[47] DU J, LIU Y, GUO H, et al. DependEval: Benchmarking LLMs for Repository Dependency Understanding[C/OL]//Annual Meeting of the Association for Computational Linguistics. 2025. https://api.semanticscholar.org/CorpusID:276903041.
[48] WEI J, WANG X, SCHUURMANS D, et al. Chain-of-thought prompting elicits reasoning in large language models[C]//Advances in Neural Information Processing Systems (NeurIPS): vol. 35. 2022: 24824-24837.
[49] CHEN M, TWOREK J, JUN H, et al. Evaluating Large Language Models Trained on Code[J/OL]. ArXiv, 2021, abs/2107.03374. https://api.semanticscholar.org/CorpusID:235755472.
[50] MERKEL D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux journal, 2014, 2014(239): 2.
[51] KERRISK M. Namespaces in operation, part 1: namespaces overview[J/OL]. LWN. net, 2013, 1. https://lwn.net/Articles/531114/.
[52] MENAGE P. Cgroups[C]//Proceedings of the Linux Symposium: vol. 1. 2007: 27-29.
[53] SULTAN S, AHMAD I, DIMITRIOU T. Container security: Issues, challenges, and the road ahead [J]. IEEE Access, 2019, 7: 52976-52996.
[54] BROWN N. OverlayFS: The Multi-Layer Filesystem[J/OL]. LWN.net, 2013. https://lwn.net/Articles/718131/.
[55] Cloud Native Computing Foundation. Container Network Interface Specification[Z]. https://github.com/containernetworking/cni. Accessed: 2024-03-20. 2023.
[56] BURNS B, GRANT B, OPPENHEIMER D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.
[57] SOUPPAYA M, MORELLO J, SCARFONE K. Application container security guide[R]. NIST Special Publication 800-190. National Institute of Standards, 2017.
[58] SHAMIM M S I, GIBSON J A, MORRISON P, et al. Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes[J/OL]. arXiv preprint arXiv:2211.07032, 2022. DOI: 10.48550/arXiv.2211.07032.
[59] VALLÉE-RAI R, CO P, GAGNON E, et al. Soot - a Java Bytecode Optimization Framework[C/OL]//Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. 1999: 13. DOI: 10.1145/781995.782008.
[60] LINDHOLM T, YELLIN F, BRACHA G, et al. The Java virtual machine specification[M]. Java SE 8 Edition. Addison-Wesley Professional, 2014.
[61] VALLÉE-RAI R, GAGNON E, HENDREN L, et al. Soot: a Java bytecode optimization framework [C]//CASCON First Decade High Impact Papers. 2010: 214-224.
[62] DEAN J, GROVE D, CHAMBERS C. Optimization of object-oriented programs using static class hierarchy analysis[C]//European Conference on Object-Oriented Programming. 1995: 77-101.
[63] BOMMASANI R, HUDSON D A, ADELI E, et al. On the opportunities and risks of foundation models [R]. arXiv preprint arXiv:2108.07258. Stanford University, 2021.
[64] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[C]//Advances in Neural Information Processing Systems: vol. 30. 2017: 5998-6008.
[65] DOBIES J, WOOD J. Kubernetes Operators: Automating the Container Orchestration Platform[M]. O'Reilly Media, 2020.

[66] Open Container Initiative. Open Container Initiative Runtime Specification[Z]. https://github.com/opencontainers/runtime-spec. v1.1.0. 2023.
[67] KERRISK M. The Linux programming interface: a Linux and UNIX system programming handbook [M]. No Starch Press, 2010.
[68] RICHARDSON C. Microservices patterns: with examples in Java[M]. Manning Publications, 2018.
[69] SOTO-VALERO C, HARRAND N, MONPERRUS M, et al. Comprehensive Analysis of Bloat in Java Applications[C/OL]//Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021: 1069-1081. DOI: 10.1109/ASE51524.2021.9678558.
[70] ARZT S, RASTHOFER S, FRITZ C G, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps[J/OL]. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014. https://api.semanticscholar.org/CorpusID:12083354.
[71] TIP F, PALSBERG J. Scalable propagation-based call graph construction algorithms[C/OL]// Conference on Object-Oriented Programming Systems, Languages, and Applications. 2000. https://api.semanticscholar.org/CorpusID:11748853.
[72] CHANDRAMOULI R. Attribute-based Access Control for Microservices-based Applications using a Service Mesh[R/OL]. NIST Special Publication 800-204B. Available at: https://csrc.nist.gov/pubs/sp/800/204/b/final. Gaithersburg, MD: National Institute of Standards, 2021. DOI: 10.6028/NIST.SP.800-204B.
[73] CUPPENS F, CUPPENS-BOULAHIA N, GARCIA-ALFARO J, et al. Detection of Configuration Vulnerabilities in Distributed Firewalls[J/OL]. IEEE Transactions on Network and Service Management, 2014, 11(3): 318-331. DOI: 10.1109/TNSM.2014.2336473.
[74] Kubernetes Documentation. Debugging with Ephemeral Containers[Z]. https://kubernetes.io/docs/tasks/debug/debug-application/debug-running-pod/. Official guide on using ephemeral containers for distroless debugging. 2024.