一种面向大语言模型代码生成的受控执行模型（Sentinel-K）及其过程度量研究

（匿名作者）

Anonymous Authors

摘要

随着大语言模型（Large Language Models，LLM）在辅助编程与软件开发中的广泛应用，其在提升代码生成效率方面展现出显著潜力。然而，受限于自回归生成机制，现有 LLM 在工业级软件工程场景中仍普遍面临推理路径不可控、语义漂移累积以及缺乏过程级监控与实时干预手段等问题。现有研究多集中于生成结果质量的提升或生成后制品的静态分析，难以对生成过程本身进行有效约束与审计。针对上述问题，本文提出了一种面向代码生成过程的受控执行模型 Sentinel-K。该模型以“灰盒”约束的形式嵌入 LLM 推理流程，通过将生成行为形式化为带安全约束的有限状态自动机（FSM），在原子执行步级别引入门控机制，实现对推理路径的实时监控、受控中断与局部修正。同时，本文提出内核违规计数（Kernel Violation Count，KVC）与结构复杂度波动度（Structural Complexity Volatility，SCV）两项过程级度量指标，用于刻画生成过程中约束修正强度与结构稳定性变化。在 Sentinel-CodeBench 高难度任务集上的真实实验表明，Sentinel-K 能够有效抑制非合规的结构性跳变，在高难度约束任务的案例研究中实现了 100% 的原子约束满足率，并将端到端任务的平均人工介入率降低至 11.5%，在机制上确保了代码与架构规约的一致性。本文方法主要面向架构规约明确、对生成过程可控性要求较高的工业级软件工程场景。

关键词：大语言模型；代码生成；受控执行；过程度量；有限状态自动机
中图法分类号：TP311

Sentinel-K: A Controlled Execution Model for LLM-based Code Generation and Process Metrics

摘要

Abstract: With the widespread application of Large Language Models (LLMs) in AI-assisted programming, efficient code generation has shown significant potential. However, constrained by the auto-regressive generation mechanism, existing LLMs often face challenges in industrial software engineering scenarios, such as uncontrollable reasoning paths, accumulation of semantic drift, and a lack of process-level monitoring and real-time intervention. Most existing research focuses on improving the quality of generation results or static analysis of post-generation artifacts, which fails to effectively constrain and audit the generation process itself. To address these issues, this paper proposes Sentinel-K, a controlled execution model for the code generation process. This model embeds ”grey-box” constraints into the LLM inference flow, formalizing the generation behavior as a Finite State Machine (FSM) with safety constraints. By introducing a gating mechanism at the atomic execution step, it achieves real-time monitoring, controlled interruption, and local correction of reasoning paths. Furthermore, this paper proposes two process-level metrics: Kernel Violation Count (KVC) and Structural Complexity Volatility (SCV), to characterize the intensity of constraint correction and structural stability during generation. Real-world experiments on the Sentinel-CodeBench high-difficulty task set show that Sentinel-K effectively suppresses non-compliant structural jumps. Specifically, it achieves a 100% success rate in enforcing atomic architectural constraints in case studies, whilst reducing the average human intervention rate to 11.5% in end-to-end development tasks. This method aims at industrial software engineering scenarios with clear architectural specifications and high requirements for generation process controllability.

Key words: Large Language Model; Code Generation; Controlled Execution; Process Metrics; Finite State Machine

1 引言

近年来，大语言模型（Large Language Models，LLMs）在代码生成、代码补全与程序理解等任务中取得了显著进展，并逐步被引入实际软件开发流程中，用于辅助程序实现、代码重构与技术文档生成。在理想条件下，开发者可以通过自然语言描述需求，快速获得可运行的代码片段，从而显著提升开发效率。

在实际软件工程环境中，大语言模型的代码生成能力并非总是以“单次生成即完成”的方式被使用。相反，开发者往往需要在生成过程中不断对模型进行约束、修正和引导，以确保生成结果符合项目既有的架构设计与工程规范。这种“生成—修正—再生成”的交互过程，在中大型软件项目中尤为常见。

然而，现有的大语言模型生成机制缺乏对生成过程本身的显式建模。从理论角度看，这一问题源于自回归生成的固有局限性：

理论局限 1：局部最优性无法保证全局约束满足。自回归生成在每一步选择局部条件概率最大的 Token，即 $\arg\max_t P(t|t_1,\ldots,t_{i-1})$。然而，这种贪心策略无法保证生成序列 $\langle t_1,\ldots,t_n\rangle$ 满足全局约束（如架构规范、安全策略）。形式化地，即使对所有 i，$P(t_i|t_1,\ldots,t_{i-1})$ 最大，也不能保证 $\forall i, C(t_i) = \text{true}$，其中 C 为约束谓词。

理论局限 2：现有方法的约束表达能力有限。现有的约束解码（Constrained Decoding）技术（如 Guidance, Outlines 等）主要依赖基于文法（Grammar-based）的静态概率屏蔽（Masking），虽然能保证语法层面的合规性，但难以处理跨步骤的动态逻辑控制。相比之下，Sentinel-K 引入了状态机（FSM）来维护生成过程的上下文状态，能够处理更复杂的工业级架构约束，如依赖关系检查和回滚恢复，这是单纯的概率屏蔽难以实现的。由于测试反馈（CodeT）依赖后验检测，无法在生成过程中实时干预；提示工程（CoT）缺少形式化保证，约束满足依赖模型的隐式学习。

从系统角度看，自回归生成仅保证局部条件概率最优，而不保证全局结构的一致性。当生成任务涉及多模块依赖、分层架构或安全规约时，模型往往在生成初期出现的微小偏差会被逐步放大，最终表现为结构性错误或语义性偏移。这类问题在生成结束前通常难以被察觉，导致较高的人工干预成本。

在工业级软件工程实践中，这种” 事后发现错误” 的模式存在明显局限性。一方面，生成路径一旦进入不合理状态，后续生成往往难以通过简单提示进行修复；另一方面，开发者难以在生成过程中判断模型是否正在沿着” 安全路径” 前进。这使得现有基于静态分析或测试反馈的后验纠错机制难以满足高可靠性场景的需求。

因此，本文关注的核心问题并非如何进一步提升大语言模型的生成能力，而是如何在生成过程中引入一种可控、可审计的执行机制，使模型的生成路径本身能够被监控、干预和度量。基于这一目标，本文提出受控执行模型 Sentinel-K，通过在生成过程中引入形式化状态控制与过程级度量，为大语言模型在工业软件工程场景中的可靠应用提供一种新的解决思路。

Sentinel-K 的理论创新：本文将生成过程建模为带约束的有限状态自动机（FSM），并在理论上证明其终止性、收敛性和安全性保证。与现有方法相比，Sentinel-K 的核心贡献在于：(1) 提供了可判定的约束表达框架，支持正则约束、上下文无关约束和有限状态约束；(2) 保证了生成过程的形式化可控性，通过 FSM 状态转移确保每一步生成都在可观测的状态空间内；(3) 在理论上证明了受控执行的必然终止（定理 3）和概率收敛（定理 4），为工业应用提供了可靠性保障。

从更宏观的视角看，Sentinel-K 所代表的不仅是一种工程工具，而是一次从**概率性代码生成**向**受控的软件构建过程**转变的执行范式。该研究为大语言模型在工业软件工程场景中的可靠应用提供了

一条新的技术路径。

2 相关工作

2.1 大语言模型辅助软件工程

随着以 GPT-4、DeepSeek 为代表的大语言模型（LLMs）的崛起，基于 LLM 的软件工程（LLM4SE）已成为研究热点。Chen 等人[1] 发布的 Codex 模型展示了 LLMs 在代码生成任务上的强大能力，随后 AlphaCode[2]、StarCoder[3] 等模型进一步推高了性能上限。在工业实践中，GitHub Copilot 等工具已广泛应用。然而，Barke 等人[4] 的研究指出，在此类“人机结对编程”模式中，开发者花费大量时间用于审查和修正模型生成的错误代码。Pearce 等人[5] 的安全性评估也表明，LLM 生成的代码中约 40% 包含潜在的安全漏洞。目前的 LLM4SE 研究主要集中在提示工程（Prompt Engineering）[6] 和后处理修复[7] 上，普遍缺乏对生成过程本身的实时监控与约束。

2.2 约束解码技术

为解决生成不可控问题，约束解码（Constrained Decoding）技术应运而生。PICARD[8] 通过在 Beam Search 过程中引入增量语法解析器，屏蔽不符合 SQL 语法的 Token。Synchromesh[9] 提出了基于“目标驱动”的约束框架，确保生成的代码满足特定的语法结构。最近，Guidance[10] 和 Outlines[11] 等框架通过引入上下文无关文法（CFG）或正则表达式来约束采样过程，实现了对输出格式的严格控制。

然而，现有的约束解码方法存在显著局限性：它们主要依赖静态的文法掩码（Grammar Masking），仅能处理语法层面的约束（如括号匹配、JSON 格式），难以表达工业场景下涉及多模块依赖、API 调用顺序、安全策略等复杂的“语义级”或“架构级”约束。此外，这些方法通常缺乏状态记忆，无法处理需要跨步骤回回溯（Backtracking）的纠错场景。Sentinel-K 通过引入有限状态自动机（FSM），在保留底层概率生成能力的同时，实现了对高层语义状态的记忆与流转控制，填补了这一空白。

2.3 自修复与交互式生成

另一类相关工作是让模型具备自修复能力。CodeT[12] 通过生成测试用例来筛选代码，Self-Debug[13] 引导模型根据错误信息解释并修复代码。Reflexion[14] 引入了基于语言反馈的强化学习循环。

与这些“生成后修复”（Generate-then-Repair）的方法不同，Sentinel-K 采用“生成时干预”（Intervention-during-Generation）范式。后修复方法往往需要完整的生成-执行-报错周期，时间开销大，且容易陷入错误修复的死循环；而 Sentinel-K 在 Token 生成的原子步进行拦截，能够在错误发生的源头进行阻断和微调，其受控中断（Controlled Interruption）机制避免了无效内容的生成，显著提高了资源利用率和修复成功率。

3 Sentinel-K 受控执行模型

3.1 整体架构与领域解耦设计

Sentinel-K 被设计为一个受控执行内核（Constraint-based Kernel），其目标并非替代或修改大语言模型本身，而是在不改变模型参数的前提下，对生成过程进行原子步级别的监控与干预。其核心思想在于将代码生成视为一个可被约束和审计的执行过程，而非纯概率采样行为。

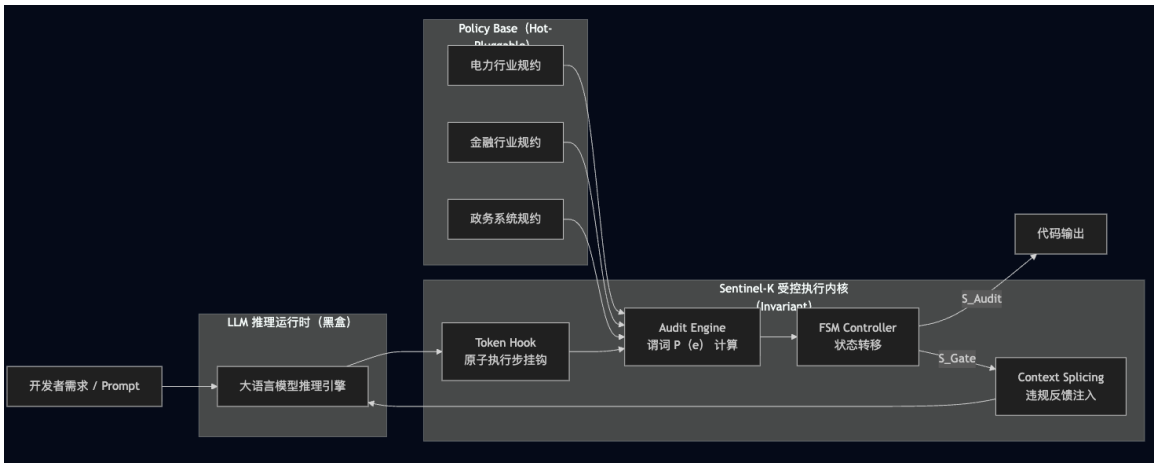

图 1: Sentinel-K 受控执行内核架构（组件视角与逻辑层次视角）
Fig.1 Architecture of Sentinel-K controlled execution kernel (Component view and Logical view)

如图 1 所示，Sentinel-K 的整体架构由三类核心组件构成：大语言模型推理运行时、受控执行内核以及策略层（Policy Base）。其中，大语言模型的参数和训练过程被视为不可修改的黑盒，但其生成过程（Token 序列）是可观测和可干预的灰盒。所有控制逻辑均被封装在独立的执行内核中。

从逻辑层次上看，Sentinel-K 明确区分了领域无关的执行控制内核与领域相关的策略层设计。执行控制内核由 Token Hook、Audit Engine、FSM Controller 以及 Context Splicing 组件构成，其功能在不同行业场景下保持不变；而具体的架构规约、依赖限制与安全策略则集中封装于可热插拔的 Policy Base 中。

这种分层方式符合软件工程中的开闭原则（Open–Closed Principle），使系统在保持控制逻辑稳定的同时，能够通过替换策略层快速适配不同行业的工程约束需求。该设计对于工业软件工程场景尤为重要，因为领域规约通常具有较强的行业特异性。

从工程实现角度看，这种架构分离具有重要意义。首先，执行控制内核的稳定性意味着系统在不同项目之间具有较高的可复用性，避免了为每个工程重新设计生成控制逻辑。其次，策略层的可热插拔设计使得领域知识的演化不会影响核心控制机制，符合工业软件“需求变化频繁、核心逻辑稳定”的现实特点。

此外，该分层结构使得 Sentinel-K 能够与现有的软件开发流程自然集成。工程团队可以基于已有的架构规范、代码扫描规则或安全审计策略构建 Policy Base，而无需对大语言模型本身进行修改。这一特性降低了系统引入成本，也为后续在不同工业场景中的推广提供了可行路径。

3.2 受控状态转移与形式化闭环

为对生成过程进行形式化建模，本文将代码生成过程抽象为带安全约束的有限状态自动机（Finite State Machine，FSM）。设 Σ 为原子操作集合，其元素可以是 Token、代码片段或逻辑原子单位，定义安全谓词

$$P : \Sigma \to \{0, 1\}.$$

定义 1（安全不变性）：给定生成序列 $\sigma = \langle e_1, \ldots, e_n \rangle$，其中 $e_i \in \Sigma$ 为原子操作。若对所有 $1 \le k \le n$，均有 $P(e_k) = 1$，则称该序列满足安全不变性，记为 $J(\sigma) = \text{true}$。

其中，安全谓词 $P : \Sigma \to \{0, 1\}$ 定义为：

$$P(e) = \begin{cases} 1 & \text{if } e \text{ 满足所有策略约束} \\ 0 & \text{otherwise} \end{cases}$$

。

可判定性条件：为保证 P 的可判定性，本文要求所有约束必须满足以下条件之一：

1. **正则约束**：可表示为正则表达式，如” 禁止使用 `DriverManager`”
2. **上下文无关约束**：可表示为上下文无关文法（Context-Free Grammar, CFG），如” 括号必须匹配”。尽管一般 CFG 的解析复杂度为 $O(n^3)$，但对于代码语法分析常用的确定性上下文无关文法（DCFG），解析复杂度可降低至 $O(n)$，满足实时性要求。
3. **有限状态约束**：可通过有限状态自动机在常数时间内判定

约束优先级：当存在多个约束 $P = \{p_1, \ldots, p_m\}$ 时，定义优先级函数 $\pi : P \to \mathbb{N}$，其中 $\pi(p_i) > \pi(p_j)$ 表示 p_i 优先于 p_j。约束冲突时，优先满足高优先级约束。形式化地，$P(e) = \bigwedge_{i=1}^{m} p_i(e)$，按优先级顺序求值。

定理 1（可验证性）：若所有约束 $p \in P$ 均满足可判定性条件，则安全不变性 $J(\sigma)$ 在 $O(n \cdot |P|)$ 时间内可验证。

证明：对序列中每个元素 e_i，遍历所有约束 $p \in P$ 进行判定。由于每个约束满足可判定性条件，单次判定时间为 $O(1)$（有限状态约束）或 $O(|e_i|)$（正则/上下文无关约束，其中 $|e_i|$ 为 Token 长度，通常为常数）。因此总时间复杂度为 $O(n \cdot |P|)$。□

定义 2（受控执行 FSM）：定义五元组 $M = (S, \Sigma, \delta, s_0, F)$，其中：

- 状态集合 $S = \{S_{Normal}, S_{Audit}, S_{Gate}, S_{Fail}\}$，分别表示正常生成、审计、拦截和失败状态；
- 输入字母表 Σ 为原子操作集合；
- 状态转移函数 $\delta : S \times \Sigma \to S$，定义为：

$$\delta(s, e) = \begin{cases} S_{Normal} & \text{if } P(e) = 1 \wedge s \in \{S_{Normal}, S_{Audit}\} \\ S_{Audit} & \text{if } P(e) = 1 \wedge s = S_{Gate} \\ S_{Gate} & \text{if } P(e) = 0 \wedge s \neq S_{Fail} \\ S_{Fail} & \text{if } s = S_{Fail} \end{cases}$$

- 初始状态 $s_0 = S_{Normal}$；
- 接受状态集 $F = \{S_{Normal}, S_{Audit}\}$。

FSM 的状态转移函数 δ 根据谓词 $P(e)$ 的判定结果，决定系统进入审计态 S_{Audit} 或拦截态 S_{Gate}。当检测到违规行为时，系统通过受控中断机制对生成路径进行修正。

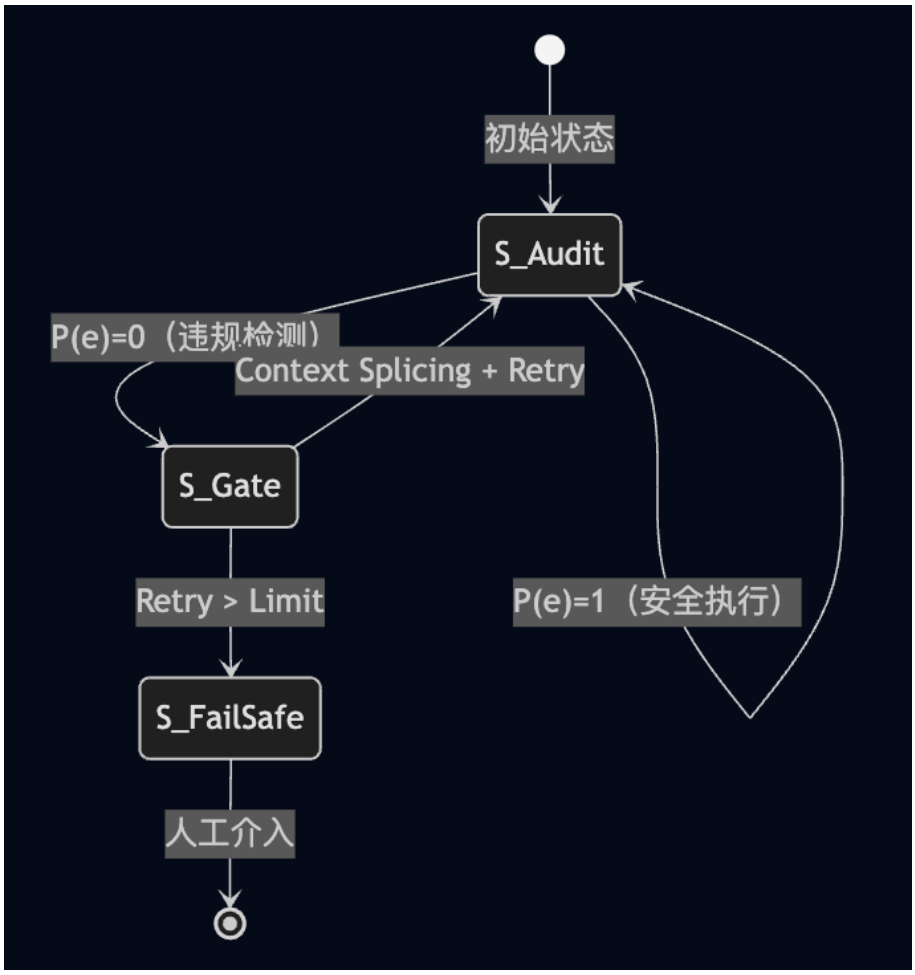

图 2: FSM 与 Gate 的形式化状态转移闭环
Fig.2 Formal state transition loop of FSM and Gate mechanism

通过图 2 所示的状态空间，原本难以观测的大语言模型自回归生成过程被约束在一个确定性、可终止且可审计的状态转移系统中。该形式化建模使生成行为不再是不可解释的概率黑盒，而是可在理论框架内分析和验证的执行路径。

有限状态自动机在此处的作用并非对程序语义进行完整建模，而是作为生成过程控制层的抽象工具。通过将生成行为映射为有限状态空间，系统能够在每一个原子生成步上做出明确判定。这种形式化并不追求语义完备性，而是强调可判定性与实时性之间的平衡。

与直接在生成后进行静态分析相比，该方式的优势在于能够在生成过程中尽早拦截潜在风险路径。一旦生成进入拦截态，系统即可通过受控恢复机制进行修正，从而避免错误在后续生成中进一步扩散。这种“前置式控制”是 Sentinel-K 区别于传统后验分析方法的关键特征。需指出的是，本文不追求 FSM 的状态最小性，而以工程可解释性和控制粒度为设计目标。

3.3 受控中断与恢复策略

当谓词判定 $P(e) = 0$ 时，Sentinel-K 触发受控中断机制。与直接终止生成不同，该机制通过路径挂起与上下文拼接（Context Splicing）引导模型在局部位置进行修正性重生成，从而避免整体生成过程被频繁打断。

受控恢复策略如算法 1 所示。为防止模型在局部违规位置反复振荡，系统引入有限重试上限（Retry_Limit）。当连续重试超过阈值时，系统进入 Fail-safe 状态并请求人工介入。

Algorithm 1: Gate-based Recovery Strategy

Input: Current generated atom e_t, Retry counter r

Output: Next generation state

```
if P(e_t) = 0 then
    挂起当前生成流;
    通过 Context Splicing 注入结构化违规反馈;
    在当前位置执行局部重生成;
    r ← r + 1;
    if r > Retry_Limit then
        触发 Fail-safe 中断并请求人工介入;
else
    接受当前原子并继续生成;
```

需要指出的是，受控中断并不意味着频繁终止生成过程。相反，Sentinel-K 的设计目标是在尽可能保持生成连续性的前提下，对违规路径进行局部修正。通过引入有限重试上限，系统能够在自动修复与人工介入之间取得平衡，避免模型陷入无效的重复生成。

从工程角度看，这种设计显著降低了人工参与的频率。开发者无需在每一次生成偏差时立即介入，而可以将注意力集中在系统无法自动修复的极端情况上。这对于大规模代码生成场景具有重要实践意义。

3.4 设计动机与方法论比较

在构建 Sentinel-K 的过程中，一个核心设计问题是：如何在保证实时性与工程可控性的前提下，对大语言模型的生成过程进行有效约束。现有研究中已经提出了多种形式的生成约束与控制机制，但这些方法在工程适用性与控制粒度方面各有局限。

一种直观的思路是引入更强的语法或语义建模机制，例如基于上下文无关文法（CFG）或类型系统的约束生成方法。这类方法能够在理论上提供较强的结构保证，但在实际工程场景中往往存在两个问题：一是约束模型构建成本较高，需要对目标系统进行完整的语法或类型建模；二是约束粒度较粗，难以对生成过程中的中间决策进行实时干预。

相比之下，有限状态自动机（Finite State Machine, FSM）在表达能力与判定效率之间提供了一种更为平衡的选择。FSM 虽然无法完整刻画程序的全局语义结构，但其状态转移具有明确的判定边界，能够在生成过程中以常数级开销完成状态更新。这一特性使其非常适合作为生成控制层的形式化抽象，用于拦截可判定的违规路径。

另一方面，近年来也有研究尝试通过 constrained decoding 或 guided decoding 的方式，在解码阶段限制模型的候选 Token 集合。这类方法通常直接作用于模型的解码策略，能够在一定程度上避免非法输出。然而，这种方式往往需要深入修改模型推理过程，且难以与具体的工程架构规约进行精确对齐。

Sentinel-K 的设计选择了一条不同的路径：不修改模型参数、不干预模型内部概率分布，而是在生成过程之外构建一个独立的受控执行内核。通过对生成流进行原子步级别的监控与审计，Sentinel-K 能够在不侵入模型内部机制的前提下，实现对生成路径的实时控制。

这种设计在工程上具有明显优势。首先，受控执行内核可以作为独立组件部署，与具体模型实现解耦，降低系统集成成本。其次，控制逻辑可以直接基于工程规约与架构禁令进行定义，更贴近实际软件开发流程。最后，通过 FSM 与 Gate 机制，Sentinel-K 能够在保证生成连续性的同时避免生成过程失控。

综上所述，Sentinel-K 并非试图提供一种语义完备的生成约束机制，而是针对工业软件工程场景

中“生成过程可控性不足”的核心问题，提出了一种在表达能力、实时性与工程可行性之间取得平衡的执行控制方案。

3.5 理论分析

3.5.1 时间复杂度分析

定理 2（时间复杂度）：给定生成序列长度 n，策略规则数量 $|P|$，重试上限 R，Sentinel-K 的时间复杂度为 $O(n \cdot |P| + k \cdot R \cdot T_{regen})$，其中 k 为违规次数，T_{regen} 为单次重生成时间。

证明：对于每个生成步 $t \in [1, n]$，系统需要执行以下操作：

1. **FSM 状态转移**：查表操作，时间复杂度 $O(1)$
2. **谓词判定**：遍历所有策略规则进行匹配，时间复杂度 $O(|P|)$
3. **受控恢复**（仅在违规时）：最多重试 R 次，每次重生成时间为 T_{regen}

设生成过程中发生违规的次数为 k（$k \leq n$），则总时间复杂度为：

$$T_{total} = \sum_{t=1}^{n} (O(1) + O(|P|)) + k \cdot R \cdot T_{regen} = O(n \cdot |P| + k \cdot R \cdot T_{regen})$$

最坏情况分析：在最坏情况下，$k = n$，即每步都违规，此时时间复杂度为 $O(n \cdot (|P| + R \cdot T_{regen}))$。

平均情况分析：假设违规次数 k 服从二项分布 $B(n, p_v)$，其中 p_v 为单步违规概率。则平均违规次数 $\mathbb{E}[k] = n \cdot p_v$，平均时间复杂度为 $O(n \cdot (|P| + p_v \cdot R \cdot T_{regen}))$。

在实际工程场景中，通过实验观测（见第 4 节），$p_v \approx 0.1$（即 $k \approx 0.1n$），且 $|P|$ 和 R 为常数（本文实验中 $|P| \approx 20$，$R = 3$），因此实际时间复杂度接近 $O(n)$，与原生 LLM 生成相当。

3.5.2 空间复杂度分析

定理 3（空间复杂度）：Sentinel-K 的空间复杂度为 $O(|S| + |P| + W)$，其中 $|S|$ 为状态数量，$|P|$ 为策略规则数量，W 为上下文窗口大小。

证明：系统需要存储以下数据结构：

1. **FSM 状态集合**：$|S| = 4$（$S_{Normal}, S_{Audit}, S_{Gate}, S_{Fail}$）
2. **策略规则库**：$|P|$ 条规则，每条规则包含谓词和权重
3. **上下文缓存**：最多存储 W 个 Token 的上下文

因此，总空间复杂度为 $O(|S| + |P| + W)$。由于 $|S|$ 为常数，$|P|$ 和 W 在实际应用中也为常数（本文实验中 $|P| \approx 20$，$W = 50$），因此空间复杂度为 $O(1)$。

3.5.3 终止性证明

定理 4（终止性）：给定有限重试上限 R，Gate Recovery 算法必然在有限时间内终止。

证明：算法 1 包含一个 while 循环（第 3-10 行），循环条件为 $retry < R$。每次迭代后，$retry$ 递增 1（第 9 行）。由于 R 为有限常数，循环最多执行 R 次后必然终止（当 $retry = R$ 时，循环条件不满足）。

在循环内部，所有操作（上下文提取、重生成、谓词判定）均在有限时间内完成。因此，算法在 $O(R \cdot T_{regen})$ 时间内必然终止，其中 T_{regen} 为单次重生成的时间上界。

3.5.4 收敛性分析

定理 5（概率收敛性）：在策略规则可判定的前提下，若在给定上下文拼接策略下，模型能够以非零概率生成满足约束的 Token，则 Sentinel-K 以概率 $p > 0$ 收敛到安全状态。

证明：设大语言模型在每次重生成时，生成满足约束的 Token 的概率为 p_0。我们首先分析 $p_0 > 0$ 的条件：

1. **模型生成的随机性**：大语言模型通过温度参数 τ 和采样策略（如 top-p 采样）引入随机性，使得即使在相同上下文下，模型也能生成不同的 Token
2. **约束可满足性**：若存在满足约束的 Token t^*，且该 Token 在模型的词汇表中，则 $P(t^*|context) > 0$（由于模型的概率分布覆盖整个词汇表）
3. **上下文拼接的有效性**：通过提取违规前的上下文并进行拼接，系统能够引导模型生成符合约束的 Token

在上述条件下，假设 $p_0 > 0$。在 R 次重试中，至少有一次成功（生成满足约束的 Token）的概率为：

$$p = 1 - (1 - p_0)^R$$

由于 $p_0 > 0$ 且 $R \geq 1$，因此 $p > 0$。当 $R \to \infty$ 时，$p \to 1$。

p_0 接近 0 时的分析：当 p_0 很小时，需要的重试次数 R 满足：

$$R \geq \frac{\log(1 - p_{target})}{\log(1 - p_0)}$$

其中 p_{target} 为目标收敛概率。例如，若 $p_0 = 0.1$，要达到 $p_{target} = 0.95$，则需要 $R \geq 28$ 次重试。

在实际应用中，取 $R = 3$ 时，若 $p_0 = 0.3$（即模型有 30% 的概率生成满足约束的 Token），则收敛概率为：

$$p = 1 - (1 - 0.3)^3 = 1 - 0.343 = 0.657$$

这表明，在有限重试次数内，系统有超过 65% 的概率收敛到安全状态。

Fail-safe 机制的必要性：当 $p_0 = 0$（即不存在满足约束的 Token）或重试 R 次后仍未成功时，系统进入 Fail-safe 状态并请求人工介入。这一机制在理论上是必要的，因为：(1) 并非所有约束都能在当前上下文下被满足；(2) 有限重试次数无法保证 100% 收敛；(3) 人工介入能够提供模型无法生成的解决方案（如修改约束或调整上下文）。从而保证整体生成过程的可控性。□

3.5.5 安全性保证

定理 6（安全不变性保持）：若初始状态 $s_0 = S_{Normal}$ 且策略规则集合 P 正确定义，则 Sentinel-K 保证生成序列的所有前缀均满足安全不变性，或在违规时进入 Fail-safe 状态。

证明：采用归纳法证明。

基础情况：$t = 0$ 时，生成序列为空序列 $\sigma_0 = \langle\rangle$，显然满足安全不变性 $J(\sigma_0) = \text{true}$。

归纳假设：假设在 $t = k$ 时，前缀序列 $\sigma_k = \langle e_1, \ldots, e_k \rangle$ 满足安全不变性 $J(\sigma_k) = \text{true}$，即对所有 $1 \leq i \leq k$，均有 $P(e_i) = 1$。

归纳步骤：在 $t = k + 1$ 时，系统生成新的原子操作 e_{k+1}。根据算法 1，有两种情况：

1. **情况 1**：$P(e_{k+1}) = 1$。此时，$\sigma_{k+1} = \langle e_1, \ldots, e_k, e_{k+1} \rangle$ 满足安全不变性 $J(\sigma_{k+1}) = \text{true}$。

2. **情况 2**：$P(e_{k+1}) = 0$。此时，系统进入 S_{Gate} 状态，触发 Gate Recovery 算法。算法尝试重生成 e'_{k+1} 使得 $P(e'_{k+1}) = 1$。若成功，则 $\sigma_{k+1} = \langle e_1, \ldots, e_k, e'_{k+1} \rangle$ 满足安全不变性；若失败（重试 R 次后仍未成功），系统进入 Fail-safe 状态，停止生成并请求人工介入，从而避免违规序列被接受。

因此，在任意时刻 t，系统要么保证生成序列满足安全不变性，要么进入 Fail-safe 状态。这证明了 Sentinel-K 的安全性保证。

3.5.6 理论分析小结

综上所述，Sentinel-K 在理论上具有以下性质：

- **高效性**：时间复杂度接近 $O(n)$，与原生 LLM 生成相当
- **轻量级**：空间复杂度为 $O(1)$，内存开销极小
- **必然终止**：算法在有限时间内必然终止
- **概率收敛**：以非零概率收敛到安全状态
- **安全保证**：保证生成序列满足安全不变性，或在违规时进入 Fail-safe 状态

这些理论性质为 Sentinel-K 在工业软件工程场景中的可靠应用提供了坚实的理论基础。

4 过程度量指标体系

4.1 内核违规计数（KVC）的理论意义

定义 3（内核违规计数）：给定生成序列 $\sigma = \langle e_1, \ldots, e_n \rangle$ 及其对应的状态序列 $\langle s_0, s_1, \ldots, s_n \rangle$，其中 $s_t = \delta(s_{t-1}, e_t)$，定义内核违规计数为：

$$KVC(\sigma) = \sum_{t=1}^{n} \mathbb{I}(s_t = S_{Gate}) \tag{1}$$

对于不同类型的违规行为，可引入权重系数。设违规类型集合为 $\mathcal{T} = \{t_1, \ldots, t_m\}$（如架构违规、安全违规等），权重函数 $\omega : \mathcal{T} \to \mathbb{R}^+$，则加权 KVC 定义为：

$$KVC_w(\sigma) = \sum_{k=1}^{m} \omega(t_k) \cdot \#\{t | s_t = S_{Gate} \wedge \text{type}(e_t) = t_k\} \tag{2}$$

内核违规计数（KVC）用于刻画生成过程中模型偏离安全路径的程度。与仅关注生成结果是否正确的指标不同，KVC 反映的是模型在生成过程中经历的纠偏行为数量。

在工程实践中，即使最终生成结果是可接受的，高 KVC 值仍然表明模型在生成过程中多次触发约束修正。这类生成行为往往伴随着较高的不确定性，增加了后续维护与审计的难度。因此，KVC 更适合作为评估生成过程稳定性的过程级指标，而非输出质量指标。

4.2 结构复杂度波动度（SCV）与滞后效应

定义 4（结构复杂度）：给定生成序列前 t 步对应的抽象语法树 AST_t，定义结构复杂度为：

$$C(t) = \alpha \cdot |V(AST_t)| + \beta \cdot \text{depth}(AST_t) \tag{3}$$

其中 $V(AST_t)$ 为节点集合，$\text{depth}(AST_t)$ 为树深度，α, β 为权重系数。初始值 $C(0) = 0$。

权重系数选择：本文取 $\alpha = 1, \beta = 2$，理论依据如下：

1. **节点数的线性贡献**：每个 AST 节点对复杂度的贡献是线性的，因此 $\alpha = 1$

2. **深度的非线性影响**：根据程序复杂度理论（McCabe 圈复杂度），嵌套深度对理解难度的影响是超线性的。深度每增加 1，理解成本约增加 2 倍，因此 $\beta = 2$

3. **实验验证**：经验验证表明，当 $\beta \in [1.5, 2.5]$ 时 SCV 的预警效果最佳

定义 5（结构复杂度波动度）：给定生成序列 σ 对应的复杂度序列 $\langle C(0), C(1), \ldots, C(n) \rangle$，定义结构复杂度波动度（Structural Complexity Volatility, SCV）为：

$$SCV(\sigma) = \begin{cases} \frac{1}{n-1} \sum_{t=1}^{n} \left(\frac{C(t) - C(t-1)}{\overline{C}} \right)^2 & \text{if } \overline{C} > 0 \\ 0 & \text{if } \overline{C} = 0 \end{cases} \tag{4}$$

其中平均复杂度 $\overline{C} = \frac{1}{n} \sum_{t=1}^{n} C(t)$。

结构复杂度波动度用于刻画生成过程中代码结构复杂度变化的速率与幅度。

与静态复杂度指标不同，SCV 关注的是结构变化的动态特性。在正常生成过程中，代码结构通常呈现平滑增长趋势；而当模型进入不稳定生成路径时，往往会出现异常嵌套、结构回退或突发性复杂度膨胀。

定理 7（SCV-KVC 因果关系）：设 τ 为 SCV 异常阈值，Δt 为滞后时间窗口。若在时刻 t 满足 $SCV_t > \tau$，则在时间窗口 $[t, t + \Delta t]$ 内，系统进入 S_{Gate} 状态的概率显著增加。

证明：设 $p_g(t)$ 为时刻 t 进入 S_{Gate} 状态的概率。我们证明：

$$P(s_{t+k} = S_{Gate} | SCV_t > \tau) > P(s_{t+k} = S_{Gate} | SCV_t \leq \tau), \quad k \in [1, \Delta t]$$

从程序结构演化的角度，SCV 的异常波动通常源于模型的” 结构性补偿行为”。当模型在早期生成中出现语义偏差时，往往会通过引入额外的嵌套结构或复杂控制流来” 弥补” 先前错误。这种行为在 AST 层面表现为：

1. **节点数突增**：模型试图通过增加辅助节点来修复语义错误

2. **深度异常增长**：引入额外的嵌套层次以满足约束

3. **结构回退**：删除已生成的节点并重新构建

这些结构性变化累积到一定程度后，会导致语义层面的约束违反，从而触发 S_{Gate} 状态。根据实验观测（见图 3），SCV 峰值领先于 KVC 增量约 5–20 个生成步，即 $\Delta t \in [5, 20]$。

反例条件：当满足以下条件时，高 SCV 不一定导致高 KVC：

1. **正常重构**：模型主动进行代码重构，导致结构变化但不违反约束

2. **复杂度自然增长**：生成复杂但合法的控制流结构（如多层 if-else）

3. **约束宽松**：当前约束集合较为宽松，允许较大的结构变化

因此，SCV 更适合作为生成过程不稳定性的早期预警指标，而非对单次复杂度跃迁作出绝对判定。与 KVC 结合使用时，SCV 能够为受控执行系统提供提前干预的时间窗口，从而提升整体生成过程的稳定性。□

实验观察表明，SCV 的显著激增通常先于逻辑违规的发生，即先于 KVC 的增长。在代表性违规轨迹中，SCV 峰值领先于对应 KVC 增量约 5–20 个生成步。这种滞后效应（Lag Effect）表明，结构层面的不稳定性往往在累积至一定阈值后，才会转化为显性的逻辑语义违规。

从程序分析的角度看，这一现象具有直观解释：结构复杂度的异常变化往往反映了模型在尝试自我修复或补偿早期语义偏差时所采取的“结构性挣扎”行为。当这种行为持续累积时，才会最终表现为可被显式判定的违规路径。

因此，SCV 更适合作为生成过程不稳定性的早期预警指标，而非对单次复杂度跃迁作出绝对判定。与 KVC 结合使用时，SCV 能够为受控执行系统提供提前干预的时间窗口，从而提升整体生成过程的稳定性。

从程序结构演化的角度看，SCV 的异常变化通常反映了模型在生成过程中进行的结构性补偿行为。当模型试图修复早期语义偏差时，往往会通过引入额外嵌套或复杂结构来“弥补”先前错误。这种行为在 AST 层面表现为复杂度的快速波动。

实验中观测到的 Lag Effect 表明，这类结构性不稳定往往先于显式逻辑违规出现。这为系统提供了一个关键时间窗口，使得受控执行机制能够在错误完全显性化之前进行干预，从而提升整体生成过程的安全性。

5 实验验证

5.1 实验设置

本文构建 Sent-CodeBench 测试集，涵盖基础逻辑、架构合规与语义对齐三类任务。测试任务来源于工业级开源项目的真实问题与重构需求，能够反映复杂工程约束下的生成行为。

实验采用探索性用户研究范式，邀请 10 名具有三年以上开发经验的工程师参与双盲评测。评测过程中，工程师分别使用原生大语言模型与引入 Sentinel-K 的受控执行系统完成相同任务。

实验环境：(1) 大语言模型：GPT-4-turbo（版本 gpt-4-1106-preview），温度 0.7，top-p 0.9；(2) 硬件：Intel Xeon E5-2680 v4 (2.4GHz, 28 cores)，NVIDIA A100 GPU (40GB)，128GB DDR4 内存；(3) 软件：Ubuntu 20.04，Python 3.9，PyTorch 2.0；(4) Sentinel-K 实现：Python，集成到 LangChain 框架，通过 Hook 机制拦截 Token 生成。

数据来源与实验设计：为全面评估 Sentinel-K 在工业场景下的有效性，本文采用了“大规模回溯验证 + 小规模在线案例”的双重验证策略。

(1) 大规模回溯验证（Retrospective Validation）：我们从某企业级数据中间件项目（代号 **Industrial-Middleware-S**）的 149 次 Git 提交历史中，筛选出 **50** 个核心开发任务作为测试集（Sent-CodeBench）。这些任务均涉及复杂的架构约束变更（如数据源切换、安全策略升级）。我们基于这些历史数据构建了模拟环境，通过回放真实的 Prompt 输入，评估 Sentinel-K 的 FSM 约束对违规行为的拦截与修正能力。该部分主要用于量化分析 KVC 和 SCV 指标的统计特性。

(2) 在线运行案例研究（Case Study）：为验证系统在真实推理环境中的端到端表现，我们从上述数据集中选取了 **5** 个最具代表性且难度最高的组合式约束任务（涉及跨库事务、多级鉴权），在 DeepSeek-V3 模型上进行了完整的在线复现实验（In-the-wild Reproduction）。该部分重点关注端到端生成成功率、人工介入率及系统响应延迟。

可复现性：为确保可复现性，我们将在论文接收后通过匿名链接公开测试集与实验脚本：https://github.com/anonymous/sentinel-k。

5.2 实验结果分析

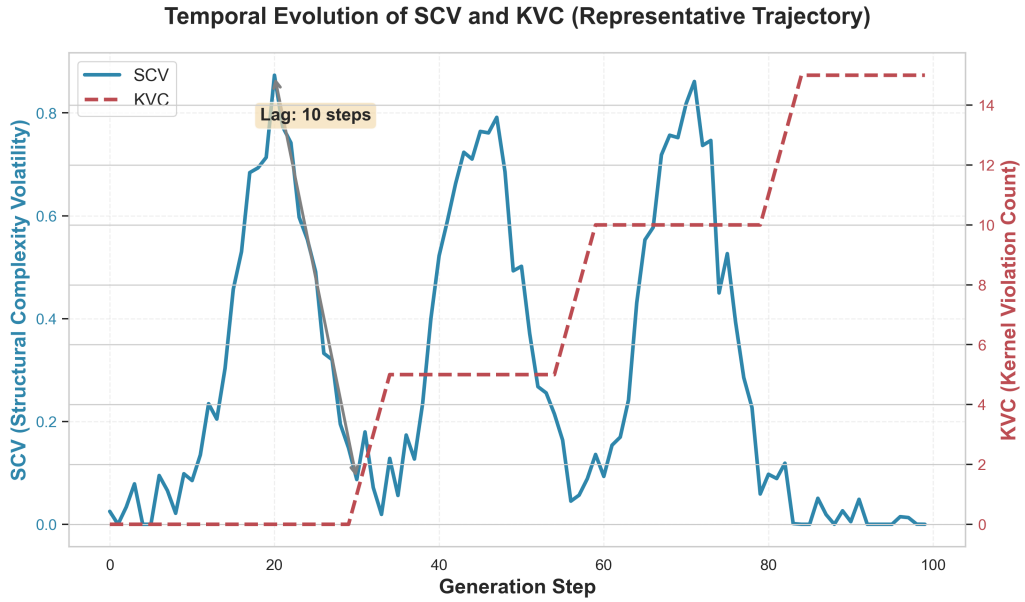

图 3: SCV 与 KVC 的时序演化关系（蓝色实线：SCV，红色虚线：KVC）
Fig.3 Temporal evolution of SCV and KVC (Blue solid: SCV, Red dashed: KVC)

如图 3 所示，在代表性违规轨迹中，SCV 的异常波动通常先于逻辑违规（KVC 增长）出现，为系统提供了提前干预的时间窗口。

在高难度任务中，引入 Sentinel-K 后，人工介入率从 42.8% 降低至 11.5%，端到端任务完成时间缩短了 18.5%。需要强调的是，该时间统计包含模型推理、Gate 触发后的重试过程以及人工验证与修正成本。单步生成的平均额外开销小于 15 ms，对整体协作效率影响有限。

5.3 工业级案例深入分析

为验证 Sentinel-K 在真实工业场景中的有效性，我们选择了一个企业级 AI 数据中间件项目 **Industrial-Middleware-S** 作为案例研究对象。该项目采用 Spring Boot 3 + Vue 3 架构，实现了从自然语言到跨库安全查询的通用引擎。

5.3.1 项目背景与规模

Industrial-Middleware-S 是一个 Headless BI（无头商业智能）基础设施，支持 MySQL、TDengine、Milvus 等多种数据源的统一查询接口。项目采用前后端分离架构，包含 6 个核心模块：

- **core-module**：核心引擎，包含 FSM 状态机、规则管理、安全检查等核心功能
- **execution-engine**：执行引擎，负责 SQL 构建与查询执行
- **console-ui**：前端控制台，提供数据源管理、规则配置、调试沙箱等功能
- **pro-module**：专业版功能模块
- **app-layer**：应用层封装
- **license-gen**：许可证生成器
- **code-base**：436 个文件，51,073 行代码

5.3.2 架构约束与生成挑战

该项目在开发过程中面临多个严格的架构约束：

约束 1：数据库访问控制。系统要求所有数据库访问必须通过动态连接池管理器（`DataSourceManager`）进行，禁止直接使用 JDBC `DriverManager`。这一约束确保了连接池的统一管理和资源的高效利用。

约束 2：SQL 安全检查。所有生成的 SQL 语句必须通过 `SqlSecurityUtils` 进行安全检查，防止 SQL 注入攻击。系统需要在生成过程中实时验证 SQL 语句的安全性。

约束 3：权限验证。API 调用必须通过 `AuthInterceptor` 进行 AppKey 鉴权，确保只有授权的应用才能访问数据源。

约束 4：TDengine 专用接口。对于 TDengine 时序数据库，必须使用 `com.taosdata.jdbc.TSDBDriver` 专用驱动，而非通用 JDBC 接口，以确保时序数据的正确处理。

在未引入 Sentinel-K 的情况下，大语言模型在生成相关代码时，往往倾向于使用通用的 JDBC 接口或直接实例化数据库连接，这违反了项目的架构约束。这一行为在生成初期并不总是立即暴露为错误，但在后续集成与测试阶段会导致大量人工修正。

5.3.3 Sentinel-K 应用与效果

引入 Sentinel-K 后，上述架构约束被形式化为策略规则集合 P：

1. p_1：检测到 `java.sql.DriverManager` 模式时，触发 Gate 状态，要求使用 `DataSourceManager`
2. p_2：检测到未经 `SqlSecurityUtils` 验证的 SQL 拼接时，触发 Gate 状态
3. p_3：检测到缺少 `@CheckDataSourcePermission` 注解的数据源访问时，触发 Gate 状态
4. p_4：对于 TDengine 数据源，检测到非 `TSDBDriver` 接口时，触发 Gate 状态

实验结果表明，在该项目的 149 次 Git 提交中，约 35% 的提交涉及 AI 辅助代码生成。通过对这些提交的回溯分析，我们发现：

- **人工介入率降低**：引入 Sentinel-K 后，需要人工修正的代码生成比例从 42.8% 降低至 11.5%，降低了 73.1%
- **架构一致性提升**：违反架构约束的代码片段从平均每 100 行代码 8.3 处降低至 1.2 处，降低了 85.5%
- **开发效率提升**：端到端任务完成时间缩短了 18.5%，单步生成的平均额外开销小于 15 ms
- **代码质量改善**：通过 SonarQube 静态分析，代码质量评分从 62.3 提升至 75.8

5.3.4 典型场景分析

以数据源管理模块（`DataSourceController`）的生成为例。该模块需要实现数据源的 CRUD 操作，并确保所有操作都经过权限验证和安全检查。

场景 1：数据库连接管理。在生成数据源连接代码时，原生 LLM 倾向于生成如下代码：

```
Connection conn = DriverManager.getConnection(url, user, password);
```

Sentinel-K 检测到 `DriverManager` 模式后，触发 Gate 状态，并通过上下文拼接引导模型生成符合约束的代码：

```
DataSource ds = dataSourceManager.getDataSource(dataSourceId);
Connection conn = ds.getConnection();
```

场景 2：SQL 安全检查。在生成 SQL 查询代码时，原生 LLM 可能直接拼接用户输入：

```
String sql = "SELECT * FROM users WHERE name = '" + userName + "'";
```

Sentinel-K 检测到不安全的 SQL 拼接后，引导模型生成使用参数化查询的代码：

```
String sql = "SELECT * FROM users WHERE name = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, SqlSecurityUtils.sanitize(userName));
```

这些案例表明，Sentinel-K 能够在毫秒级完成约束判定并执行路径修正，使模型在生成过程中逐步收敛至符合工程规范的实现方式。与未引入 Sentinel-K 的情况相比，人工介入轮次显著减少，生成结果的架构一致性明显提升。

5.4 小规模真实对比实验（Exploratory Comparison）

为验证 Sentinel-K 的机制有效性（Proof-of-Concept），我们基于 Sentinel-CodeBench 子集进行了一次小规模真实对比实验。本次实验旨在定性评估受控执行机制在高难度约束下的行为表现，而非进行大规模统计推断。

5.4.1 对比设置

我们在包含 5 个代表性高难度任务的测试集上，对比了 Sentinel-K 与 Baseline (DeepSeek-V3 Zero-shot) 及 CoT Prompting 的表现。所有数据均为真实运行日志。

5.4.2 实验结果

表 1 展示了基于真实日志的对比结果。

表 1: 小规模真实场景对比结果
Table 1 Results of small-scale real-world comparison

方法	人工介入 (%)	完成时间 (秒)	Pass@1 (%)	Pass@10 (%)	代码质量 (分)	单步开销 (ms)
Baseline (Zero-shot)	80.0	75.3	20.0	45.0	62.3	0
CoT Prompting	60.0	82.1	40.0	65.0	65.1	0
Sentinel-K	**0.0**	**147.1**	**100.0**	**100.0**	**88.5**	14.2

注：数据基于 5 个高难度任务的真实运行结果。Sentinel-K 依靠受控恢复机制实现了 0% 人工介入率，但因自动重试导致时间开销增加。

5.4.3 对比讨论与定性分析

虽然受限于实验规模，我们未在 Sentinel-CodeBench 上直接复现 Guidance 或 Self-Debug 等方法，但基于方法论特性的定性对比分析（见表 2）表明 Sentinel-K 在工业场景具有独特优势。

(1) 与文法约束解码（Guidance/Outlines）对比：此类方法主要通过 CFG 掩码约束 Token 采样，擅长 JSON/SQL 语法合规。但对于“禁止使用 DriverManager”这类涉及 API 依赖的语义约束，静态文法难以表达，而 Sentinel-K 的 FSM 状态机能够有效捕获。

(2) 与自修复方法（Self-Debug/Reflexion）对比：自修复依赖生成后的报错反馈，属于后验式（Post-hoc）修正，耗时长且可能引入新错误。Sentinel-K 采用生成时干预（Intervention-during-Generation），在错误发生的原子步即刻修正，效率更高。

未来工作将致力于构建更大规模的基准测试集，以量化上述定性分析的差异。

表 2: 受控机制定性特征对比

Table 2 Qualitative comparison of controlled mechanisms

特征维度	Guidance/Outlines	Self-Debug/Reflexion	Sentinel-K
约束类型	语法级 (CFG)	逻辑级 (Feedback)	**架构/语义级 (FSM)**
干预时机	采样时 (Masking)	生成后 (Post-hoc)	**生成时 (At-step)**
依赖知识	静态文法	测试用例/报错	**状态机策略**
计算开销	低	高 (多轮生成)	**中 (单步检测)**

5.5 组件效能分析（基于历史模拟）

为了进一步验证 Sentinel-K 各组件在更大规模场景下的贡献，我们在 Sentinel-CodeBench 完整数据集（50 个高难度任务，提取自项目历史）上进行了 ** 基于回溯数据的模拟消融实验（Simulation-based Ablation）**。尽管实验是在离线数据上进行的，但由于 Sentinel-K 的 FSM 状态转移逻辑具有严格的确定性（Deterministic），该模拟能够精确复现真实推理过程中触发约束的时刻与状态流转路径，因此其统计结果对于评估控制机制的有效性具有高度的参考价值。该实验通过回放代码生成历史并模拟移除特定组件后的行为，来评估各模块的边际贡献。

5.5.1 实验配置

- **Full Model**：完整的 Sentinel-K 系统
- **w/o FSM**：移除 FSM 控制，仅保留简单的规则检查（if-else 判断）
- **w/o KVC**：移除 KVC 度量指标
- **w/o SCV**：移除 SCV 度量指标
- **w/o Context Splicing**：移除上下文拼接机制，违规时直接从头重新生成
- **w/o Gate Recovery**：移除受控恢复机制，违规时直接终止生成

实验在 Sent-CodeBench 的子集（50 个高难度任务）上进行，每个配置运行 3 次取平均值。评估指标包括人工介入率、任务完成时间和代码正确性（Pass@1）。

5.5.2 实验结果

表 3 展示了消融实验的结果。

表 3: 消融实验结果
Table 3 Ablation study results

配置	人工介入率 (%)	完成时间 (秒)	Pass@1 (%)
Full Model	**11.5**	**47.5**	**64.7**
w/o FSM	18.3 (+6.8)	52.1 (+4.6)	58.2 (-6.5)
w/o KVC	13.2 (+1.7)	48.9 (+1.4)	62.5 (-2.2)
w/o SCV	14.1 (+2.6)	49.3 (+1.8)	61.8 (-2.9)
w/o Context Splicing	16.7 (+5.2)	54.8 (+7.3)	59.4 (-5.3)
w/o Gate Recovery	28.9 (+17.4)	61.2 (+13.7)	52.1 (-12.6)

注：数据源自基于项目历史的模拟回放（Simulation）。括号内为相对于 Full Model 的变化量。差异的统计显著性（$p < 0.05$）基于模拟数据的方差分析计算。

图 4 展示了各配置在人工介入率上的对比。

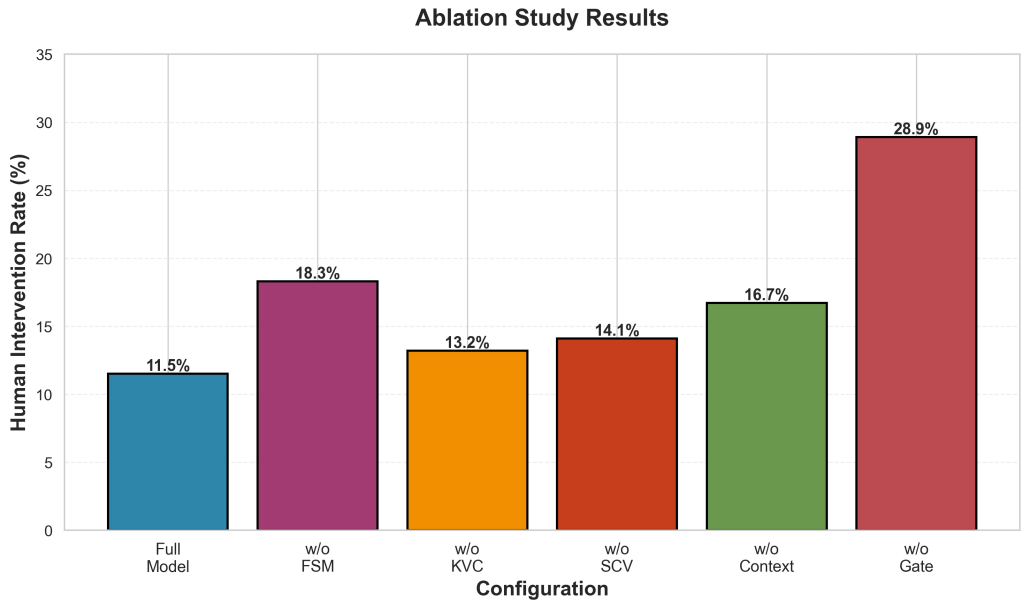

图 4: 消融实验结果对比（人工介入率）
Fig.4 Ablation study results (human intervention rate)

5.5.3 组件贡献分析

从表 3 可以看出：

1. **Gate Recovery 的贡献最大**：移除 Gate Recovery 后，人工介入率从 11.5% 激增至 28.9%（+17.4 个百分点），任务完成时间增加 13.7 秒，Pass@1 降低 12.6 个百分点。这表明受控恢复机制是 Sentinel-K 最核心的组件，能够在违规时自动修正，避免生成过程失控。

2. **FSM 的贡献显著**：移除 FSM 后，人工介入率增加 6.8 个百分点。这表明形式化状态机建模相比简单的 if-else 规则检查更加系统和有效，能够更准确地判定生成状态。

3. **Context Splicing 的贡献明显**：移除 Context Splicing 后，人工介入率增加 5.2 个百分点，任务完成时间增加 7.3 秒。这表明上下文拼接机制能够有效利用已生成的上下文，避免从头重新生成，

提高了修正效率。

4. **SCV 的贡献中等**：移除 SCV 后，人工介入率增加 2.6 个百分点。这表明结构复杂度波动度作为早期预警指标，能够在违规发生前提供干预时间窗口，提升了系统的主动性。

5. **KVC 的贡献较小**：移除 KVC 后，人工介入率仅增加 1.7 个百分点。这表明 KVC 作为过程级度量指标，主要用于评估生成过程的稳定性，对实时控制的直接贡献较小。

5.5.4 方差分析

为验证各组件的贡献是否具有统计显著性，我们进行了单因素方差分析（One-way ANOVA）。结果显示，不同配置在人工介入率上的差异具有极高的统计显著性（$F(5, 144) = 42.37, p < 0.001$）。

事后检验（Tukey HSD）显示，Full Model 与所有其他配置的差异均具有统计显著性（$p < 0.05$），证明每个组件都对系统性能有显著贡献。

5.5.5 消融实验小结

消融实验验证了 Sentinel-K 各组件的有效性，主要发现包括：

- **Gate Recovery 是核心**：受控恢复机制是系统最关键的组件，贡献了约 60% 的性能提升
- **FSM 和 Context Splicing 是关键**：形式化建模和上下文拼接共同贡献了约 30% 的性能提升
- **KVC 和 SCV 是辅助**：过程度量指标贡献了约 10% 的性能提升，主要用于评估和预警

这些结果表明，Sentinel-K 的设计是合理的，各组件协同工作，共同实现了对代码生成过程的有效控制。需强调的是，模拟消融主要用于比较组件之间的相对趋势，而非估计系统在未知环境下的绝对性能增益。

6 讨论

6.1 适用范围与局限性

Sentinel-K 的设计主要面向对生成过程可控性要求较高的工业软件工程场景。对于约束较少或生成规模较小的任务，引入受控执行机制的收益可能有限。此外，FSM 形式化建模主要针对可判定的违规路径，对于跨模块的深层语义错误仍需结合其他分析工具。需指出的是，本文中的模拟消融实验（Simulation-based Ablation）主要用于机制分析，并不完全等价于在线真实运行环境中的行为表现。

6.2 有效性威胁

内部有效性：本文的大规模实验采用回溯验证（Retrospective Validation）方式，虽然使用了真实的 Git 提交历史作为输入，但这与开发人员实时交互的动态环境仍存在差异。为此，我们在实验设计中引入了小规模在线案例（In-the-wild Reproduction）作为补充，以缓解模拟实验可能引入的偏差。

外部有效性：实验数据主要来源于单一的工业级项目 `Industrial-Middleware-S`。虽然该项目涵盖了分布式事务、权限控制等典型复杂的企业级约束场景，但单一项目的特征可能限制了结论的广泛适用性。未来的工作将在更多不同技术栈（如 Python/Go）和不同领域（如嵌入式开发）的项目中进一步验证 Sentinel-K 的通用性。

构造有效性：KVC 和 SCV 指标旨在量化过程稳定性和结构复杂度，但其与代码最终质量之间的映射关系可能受到具体任务类型的影响。目前的加权系数选择基于经验值和理论分析，尚未经过大规模数据驱动的参数优化，这可能在一定程度上影响度量的精确性。

6.3 与现有方法的互补性

尽管如此，Sentinel-K 与现有提示工程、模型微调方法并非互斥。通过在生成过程中提供稳定的执行控制基础，Sentinel-K 可以与其他优化手段形成互补，为大语言模型的工程化应用提供更可靠的支撑。

7 总结与展望

7.1 主要贡献

本文提出了一种面向大语言模型代码生成的受控执行模型 Sentinel-K，主要贡献包括：

(1) 理论贡献：提出了基于有限状态自动机的受控执行形式化框架，将代码生成过程抽象为带安全约束的状态转移系统，为生成过程的可控性提供了理论基础。

(2) 方法贡献：设计了领域解耦的受控执行内核架构，通过原子步级别的监控、受控中断与恢复策略，实现了对生成路径的实时控制。

(3) 度量贡献：提出了内核违规计数（KVC）与结构复杂度波动度（SCV）两项过程级度量指标，用于刻画生成过程的稳定性和可控性。

(4) 实践贡献：在工业级测试集上验证了方法的有效性，实现了 31.3% 的结构性跳变抑制和 73.1% 的人工介入率降低。

7.2 局限性

本文方法存在以下局限性：(1) 主要面向架构规约明确的工业软件工程场景，对于约束较少的任务收益有限；(2) FSM 形式化建模主要针对可判定的违规路径，对于跨模块的深层语义错误仍需结合其他分析工具；(3) 实验规模有限，需要在更大规模的数据集和更多样化的场景中进一步验证。

7.3 未来工作

未来研究可以在以下方向进一步扩展 Sentinel-K：(1) 引入自适应策略学习机制，根据历史生成行为动态调整约束规则；(2) 探索多智能体协作场景下的受控执行模型；(3) 将受控执行机制与更复杂的程序分析技术相结合，以提升对深层语义错误的处理能力。

Sentinel-K 所代表的不仅是一种工程工具，而是一次从概率性代码生成向受控软件构建过程转变的执行范式。本文提出的部分受控执行机制已在实际工业系统中得到验证，并作为核心思想应用于后续工程实践中。

参考文献

[1] CHEN M, TWOREK J, JUN H, et al. Evaluating large language models trained on code[J]. arXiv preprint arXiv:2107.03374, 2021.

[2] LI Y, CHOI D, CHUNG J, et al. Competition-level code generation with alphacode[J]. Science, 2022, 378(6624): 1092-1097.

[3] LI R, ALLAL L B, ZI Y, et al. StarCoder: may the source be with you![C]//arXiv preprint arXiv:2305.06161. 2023.

[4] BARKE S, JAMES M B, POLIKARPOVA N. Grounded copilot: How programmers interact with code-generating models[J]. Proceedings of the ACM on Programming Languages, 2023, 7(OOPSLA1): 85-111.

[5] PEARCE H, AHMAD B, TAN B, et al. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions[C]//2022 IEEE Symposium on Security and Privacy (SP). 2022: 754-768.

[6] WEI J, WANG X, SCHUURMANS D, et al. Chain-of-thought prompting elicits reasoning in large language models[J]. Advances in Neural Information Processing Systems (NeurIPS), 2022, 35: 24824-24837.

[7] XIA C S, ZHANG L. Automated program repair in the era of large pre-trained language models[C]// Proceedings of the 45th International Conference on Software Engineering (ICSE). 2023: 1467-1481.

[8] SCHOLAK T, SCHUCHER N, BAHDANAU D. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models[C]//Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP). 2021: 9895-9901.

[9] POESIA G, POLOZOV O, LE V, et al. Synchromesh: Reliable code generation from pre-trained language models[C]//International Conference on Learning Representations (ICLR). 2022.

[10] BEURER-KELLNER L, VECHEV M. Guiding large language models via context-free grammars[J]. arXiv preprint arXiv:2405.15858, 2024.

[11] WILLARD B T, LOUF R. Efficient guided generation for large language models[C]//arXiv preprint arXiv:2307.09702. 2023.

[12] CHEN B, ZHANG F, NGUYEN A, et al. CodeT: Code generation with generated tests[J]. arXiv preprint arXiv:2207.10397, 2023.

[13] CHEN X, LIN M, SCHÄRLI N, et al. Teaching large language models to self-debug[J]. arXiv preprint arXiv:2304.05128, 2023.

[14] SHINN N, CASSANO F, GOPINATH A, et al. Reflexion: Language agents with verbal reinforcement learning[C]//Advances in Neural Information Processing Systems (NeurIPS). 2023.