arXiv:2603.02345v1 [cs.SE] 2 Mar 2026

RIVA：利用大语言模型智能体实现可靠的配置漂移检测

Sami Abuzakuk
EPFL
洛桑 ，瑞士
sami.abuzakuk@epfl.ch

Lucas Crijns
CYD Campus
armasuisse W+T
洛桑，瑞士
lucas.crijns@ar.admin.ch

Anne-Marie Kermarrec
EPFL
洛桑 ，瑞士
anne-marie.kermarrec@epfl.ch

Rafael Pires
EPFL
洛桑 ，瑞士
rafael.pires@epfl.ch

Martijn de Vos
EPFL
洛桑 ，瑞士
martijn.devos@epfl.ch

摘要

基础设施即代码（IaC）工具可自动执行云资源配置，但要验证已部署的系统是否始终与基础设施即代码规范保持一致，仍然颇具挑战。此类配置漂移可能源于基础设施即代码规范中的错误、手动更改或系统更新。基于大语言模型（LLM）的智能体人工智能系统能够自动分析海量遥测数据，因此适用于检测配置漂移。然而，现有智能体系统默认其调用的工具总会返回正确结果，因此容易受到错误工具响应的影响。由于智能体无法判断异常的工具输出反映的是真实的基础设施问题，还是工具本身发生故障，这类错误可能导致漏检漂移或误报，从而在最需要可靠性的时刻削弱系统的可靠性。我们提出 RIVA（基于验证智能体的稳健基础设施），这是一种新型多智能体系统，即使工具产生错误或误导性输出，也能稳健地验证基础设施即代码。RIVA 使用两个专门的智能体——验证器智能体和工具生成智能体，通过迭代交叉验证、多视角验证和工具调用历史跟踪展开协作。在 AIOp-sLab 基准测试上的评估表明，当存在错误的工具响应时，RIVA 将任务准确率从使用基线 ReAct 智能体时的 27.3% 提升至平均 50.0%。即使没有错误的工具响应，RIVA 也将任务准确率提高了 28%，达到 43.8%。我们的结果表明，对不同工具调用进行交叉验证，可在生产环境的云环境中实现更可靠的自主基础设施验证。

1 引言

基础设施即代码工具已成为现代 IT 运维中无处不在的一部分，使基础设施管理能够实现自动化和规模化 [9]。Terraform [5]、、Ansible [16]、和 AWS CloudFormation [2] 等热门基础设施即代码工具，使组织能够通过版本控制的配.....来定义、配置和维护其基础设施

配置文件，大幅减少人工工作量和配置错误 [7, 11, 15]。基础设施即代码工程师无需手动配置云资源，而是通过声明式模板或脚本描述所需的系统状态。随后，基础设施即代码工具会处理这些定义，将声明的状态与当前环境进行比较，并自动应用必要的更改，以达到目标状态。这种方法确保了不同环境之间的一致性（例如，开发环境或生产环境），并使基础设施具备可复现性、可测试性和可追溯性 [24]。亚马逊、奈飞、谷歌和脸书等大型公司都采用了基础设施即代码范式 [9]。

尽管具备这些优势，验证已部署的基础设施是否符合其规范仍是一项关键挑战 [26]。主要问题在于，资源配置期间或之后，基础设施可能偏离预期状态，即*ie*，发生配置漂移 [12]。在资源配置期间，基础设施即代码模块、提供商插件或云 API 中的细微错误可能导致资源仅部分创建或处于不一致状态，例如虚拟机在配置时未关联正确的安全组，或者仅为预期服务子集授予权限 [4]。这类错误可能大规模蔓延，随着时间推移，甚至导致大范围服务中断或漏洞 [14, 23]。部署之后，配置漂移又带来了更多复杂性：自动软件更新、工程师绕过基础设施即代码所做的手动更改（*eg*，例如为应对紧急情况），或性能调优，都可能以违反基础设施即代码规范的方式修改资源属性 [29]。检测配置漂移需要持续监控，并关联海量日志、事件和状态变化，以确定运行环境是否仍符合基础设施即代码定义中的意图。这项验证工作极其耗费人力且容易出错 [23]。

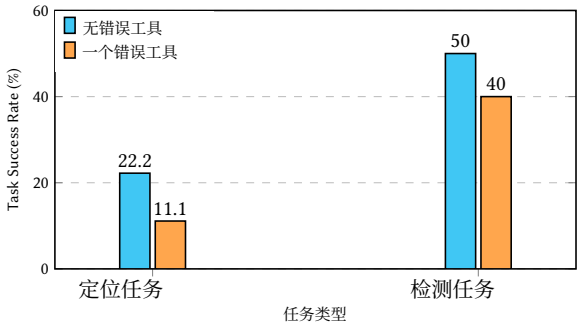


图 1：错误工具对智能体人工智能系统在 AIOpsLab 基准测试中执行错误检测和定位任务准确率的影响。

近期，基于大语言模型的智能体系统取得的进展，展现出提升云可靠性的巨大潜力 [20, 28]。在基础设施即代码和配置漂移的背景下，智能体人工智能系统能够解读日志、事件和配置状态，汇总海量遥测数据，并自主决定何时以及如何调查异常 [23]。借助大语言模型在模式识别、推理以及交叉关联不同来源信息方面的能力，这些智能体能够检测出传统基于规则的工具难以发现的配置漂移。此外，智能体工作流可以迭代完善假设、查询云 API，并依据基础设施即代码规范验证疑似问题。这表明此类系统有望大幅减少人工工作量、缩短事件检测时间，并向操作员提供有关配置漂移的高层次摘要，帮助他们采取行动。

然而，智能体人工智能在基础设施即代码（IaC）场景中的一项关键弱点，是它默认认为所调用的工具总会返回正确且可信的输出。实际上，基础设施即代码相关工具可能会因 API 中断、限流、暂时性网络问题、解析错误或过时的提供商实现而发生故障或表现不一致。尤为关键的是，此类错误的工具响应有时与真实的基础设施配置错误难以区分：两者都会表现为意外状态、字段缺失或相互矛盾的信息。因此，仅依赖工具反馈的智能体无法判断自己观察到的究竟是真实的配置漂移，还是仅仅一次有故障的工具调用。图 1 展示了在 AIOpsLab 基准测试 [3] 中，单个错误工具对定位和检测任务成功率的影响。错误工具会悄无声息地返回错误信息，从而降低任务成功率：定位任务的成功率从 22.2% 降至 11.1%，检测任务则从 50.0% 降至 40.0%。现有的智能体解决方案缺乏检测工具输出错误或对此进行推理的机制。

本文介绍了 RIVA（基于验证智能体的稳健基础设施），这是一种新颖的多智能体系统，旨在即使其所依赖的工具产生不正确或误导性输出时，仍能执行稳健的基于基础设施即代码的基础设施验证。RIVA 的核心洞见是：两个具有相同目标的工具调用不太可能同时出错。我们的系统使智能体能够通过迭代交叉验证、多视角验证以及对工具结果的历史记录跟踪，检测、推理并克服不可靠的工具响应。RIVA 的核心是两个专门的智能体：验证器和工具生成智能体。这些智能体共同收敛于关于基础设施合规性的可靠结论。此外，这些智能体可以访问工具调用历史，并利用该历史生成并调用独特的工具调用，以验证基础设施即代码规范中的单个属性。

我们在 AIOpsLab 基准测试上实现并评估了系统。我们将 RIVA 的性能与 ReAct 智能体进行比较；后者是一个在推理和工具调用之间交替进行的单智能体。在工具不可靠的情况下，RIVA 在所有任务中的任务成功率均有所提升：使用基线 ReAct 智能体时为 27.3%，而使用 RIVA 时达到 50.0%。此外，结果表明，即使工具生成正确响应，RIVA 在所有任务中的准确率和效率也更胜一筹，平均任务成功率从 28% 提高到 43.8%。总体而言，我们发现 RIVA 能有效减轻错误工具的影响，并提供稳健的基础设施验证。

贡献。我们的贡献如下：

- 我们设计了 RIVA，这是一种新颖的多智能体系统，能够在工具调用输出有误的情况下检测配置漂移（第 3 节）。
- 我们实现了 RIVA，并将我们的系统集成到 AIOpsLab 中，这是一个用于对人工智能智能体进行结构化评估的开源框架（第 4 节）。
- 我们评估了 RIVA 的有效性，结果表明，在 AIOpsLab 基准测试中的任务上，无论是否存在错误的工具响应（第 5 节），RIVA 在任务成功率方面都显著优于 ReAct 智能体。与此同时，所需令牌数量的开销也在可控范围内。

2 背景与问题描述

接下来，我们将介绍基础设施即代码、配置漂移和智能体人工智能的背景知识，然后阐述本工作所要解决的主要研究挑战。

2.1 基础设施即代码与配置漂移

Terraform、Ansible 和 AWS CloudFormation 等基础设施即代码工具允许操作员以声明式方式定义其基础设施的预期状态，从而实现可复现性、自动化和版本控制。然而在实践中，已部署资源往往会偏离基础设施即代码定义，这种现象称为**配置漂移** [12]。在资源配置期间，细微的错误或提供商 API 不一致可能导致资源在缺少某些属性或属性不正确的情况下被创建，却没有报告任何明确错误，从而引发漂移；资源配置完成后，手动热修复、自动扩缩容操作或紧急干预也可能在基础设施即代码工作流之外修改参数，导致漂移。随着时间推移，这些差异会不断累积，使运行状态与其规范日益脱节；要在云规模下检测这些差异，则需要大量工具和自动化。附录 A 提供了一个具体的 Terraform 示例。

2.2 智能体人工智能

大语言模型近来已成为云工程和提升可靠性的有力解决方案 [13, 20, 27]。尤其是，大语言模型能够处理现代云环境中常见的异构非结构化遥测数据 [17]。它们在模式识别、摘要生成和上下文推理方面的优势，使其非常适合解读复杂的系统行为并识别潜在问题，例如。比如，检测概念漂移。然而，单独使用大语言模型并不足够：有效的基础设施验证需要迭代推理，并持续与运行环境交互，以便从节点和服务中获取适当的数据。

为实现此类交互，人们开始转向采用智能体人工智能，让自主智能体与云 API、遥测端点或自定义脚本等外部工具协同工作，以收集证据并加深对系统状态的理解 [6, 10]。具体而言，工具是外部服务接口，智能体可以通过编程方式调用这些接口，执行超出底层大语言模型固有能力范围的功能，从而通过提供直接的编程访问来扩展大语言模型的能力

遥测数据和配置数据。大语言模型智能体决策过程的核心部分，是对可用工具的选择、调用和编排 [18]。

在基础设施即代码和配置漂移的背景下，工具调用能够让智能体探查环境、查询已部署资源的属性，或获取用于确认实际系统是否符合基础设施即代码规范的遥测数据。附录C展示了详细的验证工作流。组织此类交互的一种常用方法是ReAct框架，其中智能体交替进行自然语言推理和具体的工具调用 [30]。我们将由此形成的推理与工具使用决策序列称为轨迹。总之，智能体人工智能为应对配置漂移带来的挑战提供了一个前景可期的方向。

2.3 问题描述

智能体人工智能在基础设施即代码验证中的有效性，根本上取决于其调用工具的可靠性。当工具提供不完整、过时或具有误导性的输出时，智能体可能会错误地判定某项资源配置有误；反之，也可能在发生配置漂移时仍认定基础设施运行正常。因此，我们需要稳健的智能体框架，即使无法保证工具输出的正确性，也能够成功运行。

关于错误的工具输出如何在不知不觉中误导智能体，一个示例见附录B。

总之，核心问题在于，智能体人工智能系统依赖的工具输出本身可能不可靠、不完整或已过时，而当前架构没有提供检测或推理此类不一致情况的原则性机制。因此，智能体无法可靠地区分真正的配置漂移与工具故障行为。这引出了本研究的核心研究问题：即使所依赖的工具返回错误或具有误导性的输出，我们该如何设计智能体人工智能系统，以便稳健地验证由基础设施即代码定义的基础设施？

3 RIVA 的设计

我们设计了 RIVA，这是一种用于在工具不可靠的情况下验证基础设施即代码的多智能体架构。我们在图2中展示了 RIVA 的系统架构和工作流。系统架构的核心是两个协同工作的大语言模型智能体：一个验证器智能体和一个 工具生成智能体。验证器智能体可以访问基础设施即代码规范，并找出待验证的属性。工具生成智能体生成工具调用、执行这些工具调用、收集结果，并将结果发送回验证器智能体。

RIVA 背后的主要洞见在于：尽管单次工具调用可能会悄无声息地返回不正确或具有误导性的信息，但多个相互独立的工具以相同方式全部失败的可能性极低。具体而言，工具通常被实现为复杂命令的封装，以简化与智能体的交互。例如，在 Kubernetes 环境中，可以使用 Kubernetes 可执行程序实现一个 get_logs 工具，从而将使用该二进制文件的复杂性抽象掉。然而，访问 Pod 的日志也可以通过直接访问该 Pod 中的日志文件来完成（即通过 SSH）。基础设施中的细微漂移配置可能会使原来的 get_logs 失效，但很可能不会影响对机器的直接访问。

因此，我们不依赖单个工具响应，而是通过对不同工具调用进行交叉验证来核验每项属性。通过比较它们的输出、识别矛盾并迭代改进验证尝试，智能体能够区分真正的配置漂移与错误的工具行为。因此，验证器智能体和工具生成智能体都可以访问一个名为“工具调用历史”的共享数据结构，以核验基础设施是否符合规范。

接下来，我们将先详细介绍系统的各个组件，然后描述完整的 RIVA 工作流。

3.1 工具历史

工具历史是支持智能体协作和工具可靠性评估的核心数据结构。它以映射的形式实现，其中每个键对应一个属性标识符（即基础设施规范中的特定一行），每个值则是一个数组，最多包含 K 条工具执行记录。每条执行记录都必须使用不同的工具。例如，在 $K = 2$ 并测试节点1的可达性时，工具生成智能体可以添加一条执行记录，调用 ping_node(id=1) 和 send_message(id=1)。实际上，超参数 K 定义了验证器智能体得出结论前必须执行的诊断路径数量。我们在第5.3节评估该参数对任务成功率的影响。

工具历史中的每条记录包含三个要素：（1）已执行的命令及其参数，（2）工具生成的结果，以及（3）简要分析，说明这些结果对属性意味着什么。这种结构使验证器智能体能够检查针对同一属性的多次验证尝试，从而通过交叉验证检测不一致或不可靠的工具输出。

3.2 工具生成智能体

工具生成智能体生成针对特定属性量身定制的验证命令。被调用时，该智能体首先检查工具历史，以识别针对目标属性先前执行过的所有工具。随后，它综合出一种新的验证方法，采用不同的技术或从另一角度检查该属性，以确保验证方法的多样性。生成命令后，智能体在已部署的基础设施上执行该命令并收集结果。如果工具调用不正确（参数错误、语法错误等），智能体会纠正问题并再次执行该命令。最后，当工具在基础设施上成功运行时，智能体会在工具历史中创建一条新记录，其中包含命令规范、执行结果，以及解释这些结果对属性合规状态所揭示内容的分析。

3.3 验证器智能体

验证器智能体负责验证已部署的基础设施是否符合系统规范。对于规范中的每个属性，智能体都会查询工具历史，以确定是否存在针对该属性的验证尝试。如果未找到记录，智能体会请求工具生成智能体创建

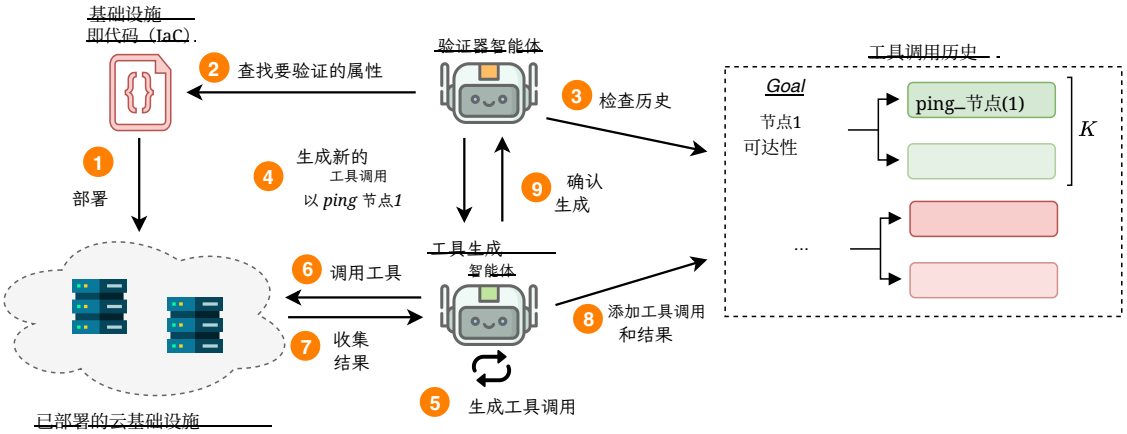


图2：RIVA 系统架构与工作流。

适当的验证命令。如果存在记录，智能体会分析存储的工具结果，以确定这些结果是否提供了充分证据，足以得出该属性已满足或遭到违反的结论。关键是，只有在有 K 条记录可用时（即探索了 K 条不同的诊断路径来生成结果时），智能体才会就该属性得出结论。虽然这种设计可以防止智能体过早得出结论，但仍允许智能体验证其他属性。尤其是，由于工具生成智能体每次只生成一个工具调用，验证器智能体会在每次生成之间插入一个推理步骤。这样，智能体便能频繁地重新评估计划；例如，如果最初选定的属性并非推进验证任务所必需，智能体可以在工具历史中添加一个新目标，并放弃第一个目标。请注意，在这种情况下，由于 K 约束，智能体不会将被放弃的目标视为已证实或已证伪。

4 实验设置

现在，我们将介绍用于评估 RIVA 的实验设置，包括 AIOpsLab 基准测试、RIVA 的实现细节以及评估流程。

4.1 AIOpsLab 基准测试

我们使用微软研究院开发的开源框架 AIOpsLab 进行评估，该框架用于设计、开发和评估自主 AIOps 智能体 [3]。AIOpsLab 提供了一个全面的基础设施，可用于部署微服务云环境、注入故障、生成工作负载并导出全面的遥测数据。此外，AIOpsLab 支持多项关键的人工智能运维任务，包括事件检测、定位、根因诊断和缓解，因此是评估大语言模型智能体进行基础设施验证的理想测试平台。

4.2 基线智能体

作为对比，我们采用 AIOp-sLab 中的 ReAct 智能体实现作为基线。ReAct框架支持智能体交替进行推理和行动步骤，从而生成口头推理轨迹，然后再选择并执行工具 [30]。我们选择 ReAct 作为基线，因为它高度依赖工具调用，也是现有大多数智能体人工智能系统采用的标准模式 [18]。

4.3 工具可靠性评估

为了评估我们的方法处理错误工具输出的效果，我们修改了 AIOpsLab 提供的部分可观测性工具的行为。AIOpsLab 的实现提供了各种工具，用于收集已部署基础设施的遥测数据。具体而言，我们将 get_logs、read_traces 和 read_metrics 函数配置为返回空字符串，而非预期的遥测数据。这些修改模拟了工具不可靠的情况，同时保留了错误输出与真正错误之间的区别：违反预期接口或遭遇执行失败的工具调用仍会按预期引发异常。这一设计选择反映了这样的情形：工具虽能成功执行，但由于配置问题、API 超时或数据收集失败，返回的信息却具有误导性或不完整。为确保对两种实现进行公平比较，我们将每种实现的最大步数都设为 45。对于 RIVA 这一多智能体系统，该限制代表两个智能体执行的步数总和。

4.4 评估协议

我们在 AIOpsLab 中提供的所有根因分析任务上，评估基线 ReAct 智能体和我们提出的系统。该基准测试包含三类任务：定位任务 用于确定特定故障的根因所在位置，检测任务旨在判断是否发生了事件，而分析任务 则用于诊断已识别事件的根因。对于每个任务类别，我们在不同的工具可靠性条件下开展实验，系统地在两个可观测性工具之一（get_logs、read_metrics）中引入错误行为。每项任务运行 5 次，并衡量两个系统正确完成各类任务的准确率。对于所有实验，我们使用模型 gpt-oss:120b [1]。

5 实验评估

我们评估 RIVA 的性能。我们的评估回答了以下问题：

- 在 AIOpsLab 基准测试的任务中，RIVA 和 ReAct 基线智能体的任务成功率是多少？在存在和不存在错误的工具响应时分别如何（第 5.1 节）？
- 在 AIOpsLab 基准测试的任务中，RIVA 和 ReAct 基线智能体所用的步骤数和令牌数分别是多少（第 5.2 节）？
- 工具调用历史的大小（由 K 参数化）会如何影响 RIVA 的性能（第5.3节）？

5.1 RIVA 和 ReAct 智能体的任务成功率

我们在 AIOpsLab 基准测试中的所有检测、定位和分析任务上运行 RIVA（使用 $K = 2$）和 ReAct 智能体。图 3 展示了各类任务在有无错误工具时的任务成功率。存在错误工具时，RIVA 的任务成功率始终不低于仅使用正确工具的 ReAct 智能体。例如，ReAct 实现的定位任务准确率在使用正确工具时为 22.2%，而使用错误的 get_logs 工具实现时为 11.1%。使用同一错误工具时，RIVA 的准确率达到 20.0%。值得注意的是，即使所有工具响应都正确，RIVA 在所有任务上的平均准确率仍从 28.0% 提高到了 43.8%。对智能体执行轨迹的仔细分析表明，这一提升主要源于 RIVA 的多智能体设计。通过分工，验证器智能体只专注于解读工具调用结果，而不负责生成和纠正这些结果。这种分工减少了困扰 ReAct 智能体的推理错误；ReAct 智能体往往会因上下文被失败的工具执行结果塞满，而忘记最初的目标。

这些结果表明，即使工具响应有误，RIVA 仍能达到更高的任务成功率。

然而，RIVA 无法达到使用正确工具时的准确率。这表明我们的系统未能完全减轻错误工具带来的影响。具体而言，对于定位任务，当所有工具调用都正确时，RIVA 的任务成功率为 40.0%；但当 get_logs 和 read_metrics 工具出错时，任务成功率则为 20.0%。对若干执行轨迹的检查显示，工具生成智能体是准确率下降的原因。具体来说，由于该智能体负责生成工具调用，因此也负责纠正错误的工具调用。例如，如果智能体未能识别正确的服务名称，就会尝试使用自己的工具找出正确的服务名称。这些工具调用不会被添加到工具历史中，这意味着后续调用工具生成器时很可能不得不重复相同的工作。这会增加每次生成所需的步骤数和出错概率，从而降低准确率。可以通过动态地将探索性工具调用追加到历史记录中来缓解这一问题，使工具生成智能体能够在先前的纠错尝试基础上继续工作，而不是重复相同的发现过程。

值得注意的是，错误的工具调用有时会提高 *ReAct* 智能体的准确率。例如，ReAct 智能体在检测任务上的任务成功率为 50.0%，而使用错误的 get_logs 时，成功率达到 62.5%。我们注意到，检测任务是最简单的任务，因为只需找到一个异常指标或日志即可解决。因此，通过让 get_logs 返回空字符串，基础智能体会忽略日志结果，并立即专注于指标结果。指标内容更集中、冗长程度更低，因此能减少幻觉，进而取得更好的结果。总的来说，图 3 中的结果表明，RIVA 成功利用其设计来减轻错误工具的影响。

表 1 ReAct 和 RIVA 在 AIOpsLab 上使用不同 K时的平均成功率

智能体	仅正确工具	错误获取_日志	错误读取指标_
ReAct	28.00	25.93	28.57
RIVA（K=1）	27.67	24.80	29.02
RIVA（K=2）	43.75	46.67	53.33
RIVA（K=3）	0	0	0

5.2 RIVA与ReAct智能体的效率

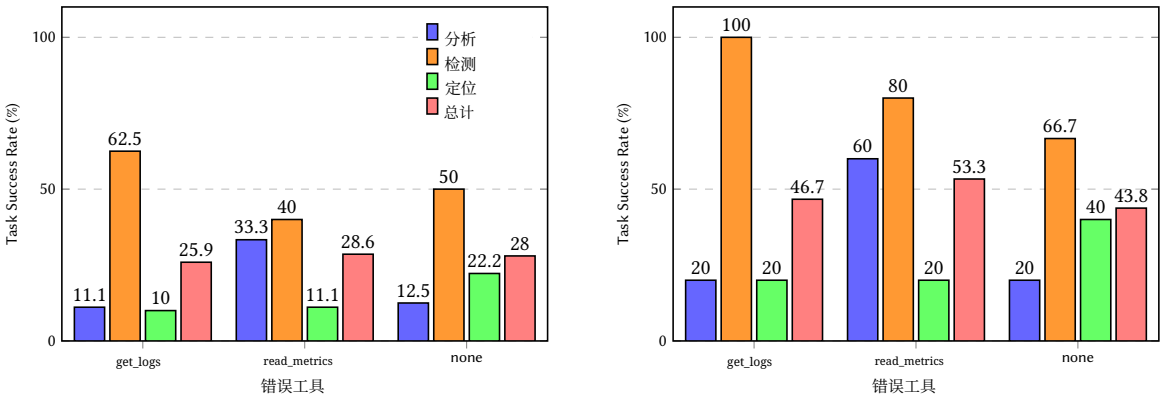
我们通过量化轨迹长度和令牌用量来比较 RIVA 与 ReAct 的效率，这两项指标直接反映智能体推理所需的时长和计算成本。在工具正确的情况下，RIVA 的任务中有 80%（$K = 2$）在 15 步内完成，而 ReAct 为 60%；RIVA 的步数最多为 17 步，而 ReAct 有 33% 的运行达到 45 步的上限；RIVA 的峰值令牌用量为 38 000，而 ReAct 为 78 000。我们认为，这得益于 RIVA 的双智能体设计：它将上下文分配给不同智能体，从而降低幻觉风险并减少纠正步骤。在工具错误的情况下，RIVA 仍然最多只需 17 步，而 ReAct 有 37% 的任务超过这一阈值；RIVA 的峰值令牌用量为 50 000，而 ReAct 为 90 000。详细的累积分布函数分布见附录 D。

5.3 K 对效率的影响

我们现在考察超参数 K的调节对结果的影响。该参数决定得出结论所需的不同工具调用次数。表 1 展示了 ReAct 和 RIVA 在 K 分别设为 1、2 和 3 时的平均成功率。

当 K 设为 1 时，RIVA 的表现与 ReAct 相近。由于仅使用 1 次工具调用来验证某个属性，RIVA 的优势实际上被抵消了，因此也会陷入与 ReAct 相同的陷阱。当 K 设为 2 时，如前面几个小节所示，RIVA 的表现优于 ReAct，并且能够避免错误工具调用导致的准确率下降。然而，当 K 大于 2 时，准确率会降至 0%。这一失败源于 AIOpsLab 的一项根本限制：它没有提供足够的诊断路径，供生成器智能体针对同一目标生成三种不同的工具调用，导致其不断循环，直至达到步数上限。附录 E 对这一失败模式进行了详细分析。

这些结果表明，RIVA 的性能严重依赖于超参数 K。如果系统无法生成至少 K 个不同的工具调用来验证同一属性，就无法得出结论。该超参数直接反映了赋予智能体的诊断灵活性。在 AIOpsLab 这样的受限环境中，智能体可选择的替代诊断路径有限，因此应相应地将 K 设置得更低。这一分析还揭示了改进 RIVA 设计的一个有前景方向：当工具



(a) ReAct 智能体的任务成功率。

(b) RIVA 的任务成功率。

图 3：存在错误工具（with $K = 2$）和不存在任何错误工具时，RIVA 与 ReAct 智能体的任务成功率。

生成智能体无法识别 K 条不同路径时，系统可以降低置信度后继续进行分析，而不是完全停止。

6 相关工作

近年来，如何在存在工具错误的情况下确保智能体稳健执行，已成为基于大语言模型的智能体系统研究中日益受到关注的挑战。相关方法通常分为两类：处理显式执行失败，以及缓解隐蔽的静默错误。

Vuddanti 等人提出了 PALADIN [25]，这是一种具有自我纠正能力的语言模型智能体，旨在处理超时和 API 异常等工具故障。他们的方法着重于通过自我纠正机制，使智能体能够从工具执行失败中恢复。不过，PALADIN 假设工具故障会以显式异常（执行错误）的形式出现。

类似地，Sheffler 提出了一种用于监控智能体行为并检测其偏离预期行为模式的时序表达式语言 [19]。该方法监控工具调用和状态转换的执行轨迹，以识别智能体行为中的异常，例如工具调用顺序不当和协调失败。尽管这种方法能有效检测行为回归，但它需要预先定义时序模式，且未明确解决验证工具输出正确性的问题。

与显式失败不同，近期研究 [22]指出，静默错误是一项关键问题：工具虽成功执行，却返回了不正确的输出。Tools Fail 探讨了让大语言模型学会质疑工具并检测此类错误的方法，通常采用检查清单等上下文内干预。尽管这项工作证实了静默错误问题的存在及其重要性，但它主要关注大语言模型依据内部预期进行检测的能力，而我们的工作则为工具输出验证提供了一种结构化的外部机制。

一些研究通过改进错误恢复来解决工具错误问题。若干方法探索了基于学习的错误恢复机制，通常涉及显式的反思过程，用于诊断失败的工具调用并提出修正后的替代方案 [21]。此外，AgentDebug 提供了一种调试框架，可隔离根因故障并生成纠正性反馈，使智能体能够通过迭代从错误中恢复 [31]。

在可证明安全性方面，VeriGuard 通过将形式化验证融入代码生成过程，来应对相关挑战，确保智能体策略在构造时即具备正确性；这一过程采用由验证器反例 [8]引导的迭代改进循环。

尽管这些工作探讨了工具可靠性和错误处理的不同方面——从显式异常（PALADIN）到静默错误的内部检测（Tools Fail），再到策略级验证（VeriGuard）——但它们主要关注检测并恢复显式工具故障或执行错误。我们的方法则有所不同，专门针对工具执行时未引发异常却返回错误结果的场景；这要求智能体通过交叉验证和迭代验证来推理工具输出的可靠性，而不是依赖显式错误信号。

7 结论与未来工作

本文提出 RIVA，这是一个多智能体系统，即使在工具返回错误数据时，也能检测云基础设施中的配置漂移。通过对针对同一属性的多次独立工具调用进行交叉验证，RIVA 能够区分真实的基础设施问题与错误的工具输出。在 AIOpsLab 基准测试上的评估表明，在存在错误的工具响应时，RIVA 将使用基线 ReAct 智能体时 27.3% 的任务准确率平均恢复至 50.0%。在没有错误的工具响应时，RIVA 还将任务准确率从 28% 提升至 43.8%。此外，与基线 ReAct 智能体相比，我们的系统能以更少的步骤和令牌更高效地完成大多数任务。因此，RIVA 表明，交叉验证能够在生产环境中实现更可靠的自主基础设施验证。

未来的工作应聚焦于三个关键领域，以增强 RIVA 的稳健性和适用性。首先，需要改进工具生成行为，使智能体在使用多个工具生成所请求的工具调用时，能够动态地在历史记录中添加多个工具调用。其次，整个系统对超参数的反应

需要改进，以便系统即使在找不到替代路径时也能继续分析，而不是彻底失败。第三，AIOpsLab 基准测试对全面分析 RIVA 的能力而言限制过多，因此有必要开发或采用更好的基准测试，使其更贴近实际的基础设施即代码部署场景。这个新基准测试应能直接访问部署环境，让智能体探索更广泛的诊断路径，并在真实的生产环境中更充分地展现 RIVA 的验证能力。

致谢

本研究部分得到了 Cyber Defence Campus（项目编号 AR-CYD-C-024，合同编号 8010075905）以及瑞士国家科学基金会的支持，支持项目为 FRIDAY：节约、注重隐私且实用的去中心化学习，SNSF 提案编号 10.001.796。感谢 Gérôme Bovet、Sayan Biswas 和 Milos Vujasinovic 提供意见并参与富有见地的讨论。

参考文献

[1] Sandhini Agarwal、Lama Ahmad、Jason Ai、Sam Altman、Andy Applebaum、Edwin Arbus、Rahul K Arora、Yu Bai、Bowen Baker、Haiming Bao 等。2025。gpt-oss-120b 与 gpt-oss-20b 模型卡。*arXiv* 预印本 *arXiv:2508.10925*（2025）。[2] 亚马逊云科技。2025。AWS CloudFormation。https://aws.amazon.com/ cloudformation/。访问日期：2025-02-24。[3] Yinfang Chen、Manish Shetty、Gagan Somashekar、Minghua Ma、Yogesh Simmhan、Jonathan Mace、Chetan Bansal、Rujia Wang 和 Saravan Rajmohan。2025。Aiop-slab：用于评估人工智能智能体以实现自主云的整体框架。*arXiv* 预印本 *arXiv:2501.06706*（2025）。[4] Georgios-Petros Drosos、Thodoris Sotiropoulos、Georgios Alexopoulos、Dimitris Mitropoulos 和 Zhendong Su。2024。当基础设施是一个有缺陷的程序时：理解基础设施即代码生态系统中的故障。*ACM* 编程语言学报 8，OOPSLA2（2024），2490–2520。[5] HashiCorp。2025。Terraform。https://www.terraform.io/。访问日期：2025-02-24。[6]Laurie Hughes、Yogesh K Dwivedi、Tegwen Malik、Mazen Shawosh、Mousa Ahmed Albashrawi、Il Je on、Vincent Dutot、Mandanna Appanderanda、Tom Crick、Rahul De' 等。2025。人工智能智能体与智能体系统：多位专家的分析。计算机信息系统学报 （2025）。doi:10.1080/08874417. 2025.2483832[7] Indika Kumara、Martín Garriga、Angel Urbano Romeu、Dario Di Nucci、Fabio Palomba、Damian Andrew Tamburri 和 Willem-Jan van den Heuvel。2021。基础设施代码的注意事项：一项系统性灰色文献综述。信息与软件技术 137（2021），106593。[8] Lesly Miculicich、Mihir Parmar、Hamid Palangi、Krishnamurthy Dj Dvijotham、Mirko Montanari、Tomas Pfister 和 Long T Le。2025。VeriGuard：通过经验证的代码生成提升大语言模型智能体安全性。*arXiv* 预印本 *arXiv:2510.05156*（2025）。[9] Kief Morris。2020。基础设施即代码。O'Reilly Media。[10] San Murugesan。2025。智能体人工智能的兴起：影响、担忧与未来之路。*IEEEIntelligent Systems* 40, 2 2025 10 1109 2025 3544940 （）。[11] doi: 。/MIS. 。Ruben Opdebeeck、Bram Adams 和 Coen De Roover。 。分析基础设施即代码的软件供应链：提取 Ansible 插件依赖项。载于*2025 IEEE* 国际软件分析、演化与再工程会议（*SANER*）。IEEE，181–192。[12] Claus Pahl、Niyazi Gokberk Gunduz、Ovgüm Can Sezen、Ali Ghamgosar 和 Nabil El Ioini。2025。基础设施即代码——技术综述与研究挑战。（2025）。[13] Kannan Parthasarathy、Karthik Vaidhyanathan、Rudra Dhar、Venkat Krishnamachari、Adyansh Kakran、Sreemaee Akshathala、Shrikara Arun、Amey Karan、Basil Muhammed、Sumant Dubey 等。2025。面向自主云运维的基于大语言模型的多智能体框架工程。在 *2025 IEEE/ACM* 第四届人工智能工程——人工智能软件工程国际会议（*CAIN*）上。IEEE，201–211。[14] Akond Rahman、Sarah Elder、Faysal Hossain Shezan、Vanessa Frost、Jonathan Stallings 和 Laurie Williams。2018。基础设施即代码中的错误。*arXiv* 预印本 *arXiv:1809.07937*（2018）。[15] Akond Rahman、Rezvan Mahdavi-Hezaveh 和 Laurie Williams。2019。基础设施即代码研究的系统性映射研究。信息与软件技术 108（2019），65–77。[16] 红帽。2025。Ansible。https://www.ansible.com/。访问日期：2025-02-24。[17] Devjeet Roy、Xuchao Zhang、Rashi Bhave、Chetan Bansal、Pedro Las-Casas、Rodrigo Fonseca 和 Saravan Rajmohan。2024。《探索用于根因分析的基于大语言模型的智能体》。载于第 *32* 届 *ACM* 软件工程基础国际会议（巴西，加利尼亚斯港）（*FSE* *2024* 10 1145 3663529 3663841 ）。doi: . / . Timo Schick、[18] Jane Dwivedi-Yu、Roberto Dessì、Roberta Raileanu、Maria Lomeli、Eric Hambro、Luke Zettlemoyer、Nicola Cancedda 和 Thomas Scialom。2023。《Toolformer：语言模型可以学会自行使用工具》。神经信息处理系统进展36（2023），68539–68551。[19] Thomas J Sheffler。2025。《检查智能体系统正确性的方法》。*arXiv* 预印本 *arXiv:2509.20364*（2025）。[20] Manish Shetty、Yinfang Chen、Gagan Somashekar、Minghua Ma、Yogesh Simmhan、Xuchao Zhang、Jonathan Mace、Dax Vandevoorde、Pedro Las-Casas、Shachee Mishra Gupta 等。2024。《构建用于自主云的人工智能智能体：挑战与设计原则》。载于*2024* 年 *ACM* 云计算研讨会论文集。99–110。[21] Junhao Su、Yuanliang Wan、Junwei Yang、Hengyu Shi、Tianyang Han、Junfeng Luo 和 Yurui Qiu。2025。《失败使智能体更强大：通过结构化反思提升准确率，实现可靠的工具交互》。*arXiv* 预印本 *arXiv:2509.18847*（2025）。[22] Jimin Sun、So Yeon Min、Yingshan Chang 和 Yonatan Bisk。2024。《Tools Fail：检测故障工具中的静默错误》。载于*2024* 年自然语言处理实证方法会议论文集，Yaser Al-Onaizan、Mohit Bansal 和 Yun-Nung Chen（编）。计算语言学协会，美国佛罗里达州迈阿密，14272 14289 10 18653 1 2024 790 –。doi: . /v / . [23] 2024 emnlp-main。Gogulakrishnan Thiyagarajan、Vinay Bist 和 Prabhudarshi Nayak。《云环境中的人工智能驱动配置漂移检测》。*Gogulakrishnan Thiyagarajan*、*Vinay Bist*、*Prabhudarshi Nayak*。（*2024*）。云环境中的人工智能驱动配置漂移检测。《国际通信网络与信息安全期刊》（*IJCNIS*）16，5（2024），721–743。[24] Alexandre Verdet。2023。探索基础设施即代码中的安全实践：一项实证研究。蒙特利尔理工学院，加拿大蒙特利尔。[25] Sri Vatsa Vuddanti、Aarav Shah、Satwik Kumar Chittiprolu、Tony Song、Sun- ishchal Dev、Kevin Zhu 和 Maheep Chaudhary。2025。《PALADIN：用于修复工具故障案例的自我纠正语言模型智能体》。*arXiv* 预印本 *arXiv:2509.25238*（2025）。[26] Rosemary Wang。2022。基础设施即代码、模式与实践：使用 *Python* 和 *Terraform* 的示例。西蒙与舒斯特。[27] Yiming Xiang、Zhenning Yang、Jingjia Peng、Hermann Bauer、Patrick Tser Jern Kon、Yiming Qiu 和 Ang Chen。2025。《使用大语言模型智能体自动发现云基础设施即代码更新中的漏洞》。载于*2025 IEEE/ACM* 云智能与人工智能运维国际研讨会（*AIOps*）。IEEE，20–25。[28] Zhenning Yang、Archit Bhatnagar、Yiming Qiu、Tongyuan Miao、Patrick Tser Jern Kon、Yunming Xiao、Yibo Huang、Martin Casado 和 Ang Chen。2025。《人工智能智能体时代的云基础设施管理》。*ACM SIGOPS* 操作系统评论59，1（2025），1–8。[29] Zhenning Yang、Hui Guan、Victor Nicolet、Brandon Paulsen、Joey Dodds、Daniel Kroening 和 Ang Chen。2025。《使用人工智能智能体自动协调云基础设施即代码》。*arXiv* 预印本 *arXiv:2510.20211*（2025）。[30] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik Narasimhan 和 Yuan Cao。2023。《ReAct：在语言模型中协同推理与行动》。载于国际学习表征会议（*ICLR*）。arXiv:2210.03629 https://openreview.net/pdf?id=WE_vluYUL-X[31] Kunlun Zhu、Zijia Liu、Bingxuan Li、Muxin Tian、Yingxuan Yang、Jiaxun Zhang、Pengrui Han、Qipeng Xie、Fuyang Cui、Weijia Zhang 等。2025。《大语言模型智能体会在哪里失败，以及如何从失败中学习》。*arXiv* 预印本 *arXiv:2509.25370*（2025）。

Terraform 配置示例

本附录提供了一个具体的 Terraform 配置片段，说明基础设施即代码定义如何与实际基础设施产生偏差，并列举了资源配置期间及之后出现配置漂移的典型示例。

图4：一段 Terraform 配置代码，用于配置 Web服务器并为其分配一个安全组。

```
1 资源 " aws_instance " "web" { 2 亚马逊机器映像 =  "ami -123" 3 instance_type  ="t2. micro" 4 5 vpc_security_group_ids  = [ aws_security_group . web_sg .id ]6 7 标签  ={ 8 名称 = "网页 - 服务器 " 9 环境  =  " 生产环境 " 10 } 11 }
```

图4中的代码段以声明式方式定义了一个 Web服务器实例，其中包括其操作系统映像、计算类型和网络安全组。它还附加了用于标识的标签，这些标签通常用于组织资源，并说明它们在基础设施中的预期作用。这种方法能够实现可重复性、自动化、版本控制和一致性。

在实践中，已部署资源经常会偏离基础设施即代码定义，这种情况称为 配置漂移 [12]。配置漂移可能发生在资源配置期间，例如，基础设施即代码模块或提供商 API 中的细微错误可能导致创建的资源缺少某些属性或属性不一致。例如，云 API 的间歇性问题可能导致图 4 中的实例因后端中的无提示错误而未配置预期的安全组，尽管 Terraform 报告操作成功。这类不匹配很难立即检测到，因为基础设施即代码工具可能不会报告底层的不一致。

配置漂移也可能在资源配置之后发生。手动热修复、自动扩缩容操作、系统更新或紧急干预可能会在基础设施即代码工作流之外修改配置参数。响应事件的操作员可能会手动更新实例的安全组，或临时开放SSH端口。由于人为错误，这些更改可能仍保留在运行环境中，但未被记录到Terraform文件中。随着时间推移，这些差异不断累积，使运行状态与其规范日益脱节。

B 工具可靠性说明性示例

本附录提供了一个实例，说明错误的工具输出如何悄然误导智能体，从而引出RIVA所要解决的可靠性问题。

示例说明。考虑图5，其中包含一个 ping_node 函数调用，该函数根据节点标识符对节点执行 ping。每个节点标识符都映射到一个预期的 IP 地址，智能体通过调用 ping_node(id) 来验证节点的可达性。在正常情况下

图5：智能体用于根据预期IP映射验证节点可达性的简化工具。

```
1 # 逻辑节点标识符与其 IP 地址之间的预期映射2 nodes  ={3 "0": " 172.17.0.5 ",4 "1": " 172.17.0.6 ",5 }67 # 智能体调用的工具，用于检查节点是否可达8 def ping_node ( node_id ):9 return ping( nodes [ node_id ])
```

情况下，此工具调用会返回有关预期节点的正确信息。然而，漂移或网络重新配置可能会悄然使这些假设失效。例如，路由器可能在网络事件后重新分配IP 地址，或者操作员可能在紧急情况下手动更新网络设置，从而使IP 到节点的映射失效。由于该工具返回的是语法正确的输出，智能体没有直接信号表明出现了问题。例如，它会断定节点1是健康的，尽管ping已被转发到一台无关的设备。

虽然上述示例较为简单，但在实际的云环境中，潜在问题会严重得多。现代智能体系统会与数十种异构工具交互，每种工具都有各自的故障模式、无提示的不一致性和预设假设。随着工具和未知因素的数量增加，智能体区分真正的配置漂移与具有误导性但在语法上有效的工具输出的能力会进一步下降。

C 基础设施即代码验证工作流

本附录展示了大语言模型智能体验证基础设施即代码所定义基础设施的端到端工作流，正如第 2 节所述。

工程师首先部署某个基础设施即代码规范（步骤 1）。随后，大语言模型智能体使用其可用工具收集数据（步骤 2 和 3），并对数据进行推理（步骤 4）以识别潜在问题。此过程可能会重复多个步骤。智能体完成任务后，会向工程师提供一份报告，工程师可以进一步调查已识别的问题。

D 详细效率分析

本附录展示了第 5.2 节中提到的 RIVA 和 ReAct 基线在步骤数和令牌用量方面的完整累积分布函数分布。

图 7（右）显示，当所有工具响应均正确时，使用 RIVA（以及 $K = 2$）可在 15 个步骤内完成 80% 的任务，而 ReAct 运行中只有 60% 能在这一数量的步骤内完成。此外，RIVA（$K = 2$）所需的最大步骤数为 17，而使用 ReAct 智能体的运行中有 33% 需要 45 个步骤；这是任务自动中止前允许的最大步骤数。同样，图 8（右）显示，RIVA（$K = 2$）每项任务所使用的最大令牌数分布

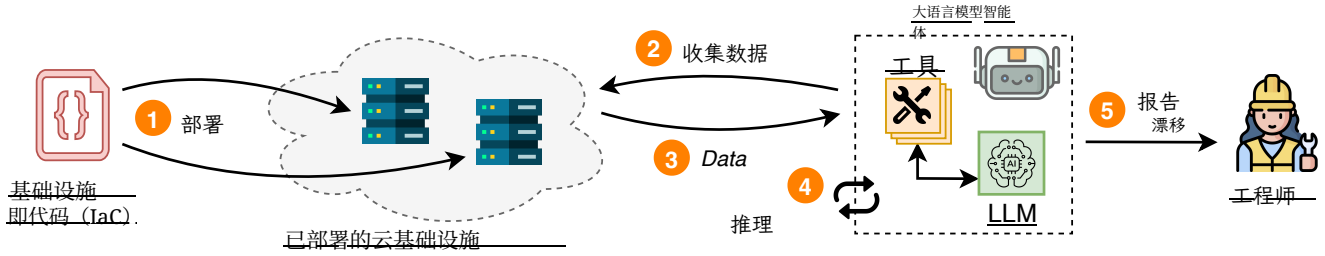


图 6：T　大语言模型智能体对由 Ia 部署的基础设施进行配置漂移检测的工作流　C.

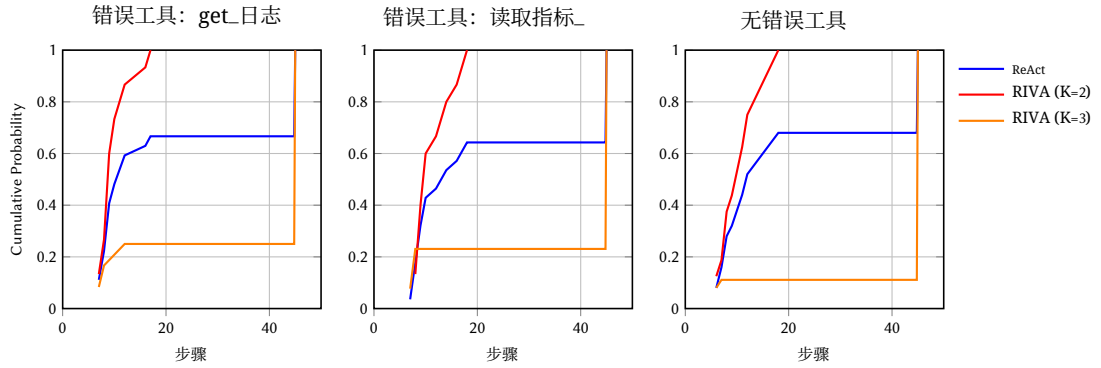


图 7：对于 RIVA 和 ReAct 智能体，在存在和不存在错误工具响应的情况下，完成任务所需步数的分布。

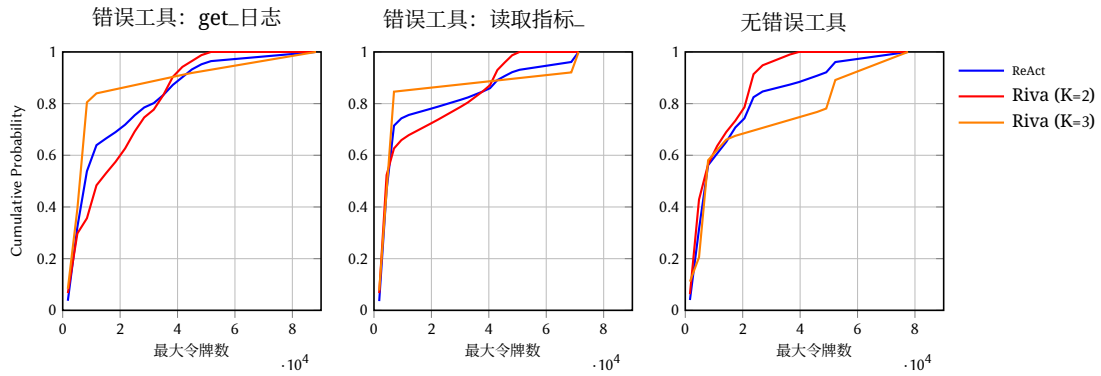


图 8：对于 RIVA 和 ReAct 智能体，在存在和不存在错误工具响应的情况下，完成任务所需最大令牌数的分布。

在 ReAct 智能体的范围内，不过 RIVA（$K = 2$）的最大令牌数为 38000 个令牌，而 ReAct 中一项任务最多需要 78 000 个令牌。我们将此归因于 RIVA 的双智能体设计，该设计有效地在两个智能体之间分配信息，从而减少上下文使用量，降低最大令牌数，并减少产生幻觉的可能性。RIVA 较小的上下文规模还减少了每个步骤中的错误工具调用次数，进而减少智能体为修复工具调用而执行的纠正步骤数量。

存在错误工具时，RIVA 平均比 ReAct 少用若干步骤即可完成任务（见图 7 左侧和中间）。

具体而言，面对错误工具时，RIVA 最多需要 17 步，而所有 ReAct 任务中有 37% 需要超过 17 步。然而，图 8（左侧和中间）显示，当工具有误时，RIVA 在每个任务中使用的令牌数上限更高；但超过 40000 个令牌后，趋势反转，因为 ReAct 运行中排名前 20% 的令牌用量超过了 RIVA 运行中排名前 20% 的用量。与仅使用正确工具的实验类似，RIVA 的最大值远低于 ReAct：RIVA 为 50000 个令牌，而 ReAct 为 90000 个令牌。进一步检查发现，与所有工具都提供正确输出时相比，存在错误工具时，RIVA 会开始生成更多命令，因此令牌用量更高，从而使系统能够避免

错误工具带来的负面影响。另一方面，ReAct 无法从错误结果中恢复，并耗尽所有步骤来尝试对这些结果进行推理。总体而言，RIVA 的设计使其能够比 ReAct 智能体更有效地利用资源，尤其是在解决任务所用的最大令牌数方面。

E K 超参数敏感性分析

本附录详细分析了 RIVA 在 $K = 3$时完全失效的原因，相关内容见第 5.3节。

图 7 和图 8 显示，$K = 3$ 会导致显著不同的结果：83% 的任务使用少于 15000 个令牌，而 77% 的运行达到 45 步，即评估中允许的最大步数。深入分析发现，大多数执行在生成第三条命令时失败。具体而言，AIOpsLab 没有为生成器智能体提供足够的灵活性，使其无法针对特定目标生成超过 2 种不同的工具调用。一旦验证器智能体要求生成器智能体生成第三条命令，生成器就会持续尝试，直至达到最大步数。此外，由于生成的工具调用全部不正确，它们只会产生少量令牌（来自基准测试的错误消息），这解释了观察到的较小令牌数上限。