RESTOR: Automated Test Oracle Generation for RESTful APIs via Reinforcement Learning

XUN ZHOU, Fudan University, China
ZHEN DONG*, Fudan University, China
MINGYU REN, Fudan University, China
QIANG LI, ByteDance, China
JUNJIE LI, ByteDance, China
SIFAN WANG, ByteDance, China
XIAOLONG YU, ByteDance, China
CHAOFENG SHA, Fudan University, China
XIN PENG, Fudan University, China

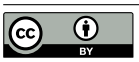
Modern REST API testing faces a critical challenge in defining reliable test oracles, particularly in agile industrial environments where formal specifications (e.g., OpenAPI) are frequently missing or outdated, and historical execution logs are unavailable for newly deployed endpoints. In this paper, we present RESTOR (**R**einforcement **E**nhanced **S**ingle-**T**raffic **O**racle generator for **R**EST APIs), a framework that generates executable test assertions from a single observed request–response pair in a black-box setting. Unlike existing approaches that rely on rule-based templates or massive training logs, RESTOR utilizes a novel data augmentation pipeline to fine-tune a lightweight Large Language Model (LLM) via Group Relative Policy Optimization (GRPO). This training process enables the model to internalize testing "common sense" by optimizing a reward function that jointly encourages: (i) the selection of stable, semantically meaningful fields for validation and the avoidance of dynamic noise (e.g., timestamps or trace IDs); (ii) the generation of robust assertions that withstand logic variations. We evaluate RESTOR on an industrial dataset comprising over 2,300 API traces across 246 real-world services. Comprehensive experiments demonstrate that RESTOR significantly outperforms prompt-engineered baselines and generalist models, achieving a superior F_1 score of 85.42% in key field identification and increasing the proportion of semantically accurate assertions. Furthermore, deployment in a production CI/CD workflow at ByteDance confirms its practical value: the system raised the adoption rate of automatically generated test cases from 74.1% to over 96%, substantially reducing manual Quality Assurance (QA) effort while ensuring high execution stability.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Information systems → RESTful web services**.

Additional Key Words and Phrases: REST API Testing, Oracle Generation, Deep Reinforcement Learning

*Corresponding Author.

Authors' Contact Information: Xun Zhou, Fudan University, Shanghai, China, xunzhou24@m.fudan.edu.cn; Zhen Dong, Fudan University, Shanghai, China, zhendong@fudan.edu.cn; Mingyu Ren, Fudan University, Shanghai, China, myren25@m.fudan.edu.cn; Qiang Li, ByteDance, Shanghai, China, liqiang.leo@bytedance.com; JunJie Li, ByteDance, Shanghai, China, wells.li@bytedance.com; Sifan Wang, ByteDance, Shanghai, China, wangsifan.28@bytedance.com; Xiaolong Yu, ByteDance, Shanghai, China, yuxiaolong.1@bytedance.com; Chaofeng Sha, Fudan University, Shanghai, China, cfsha@fudan.edu.cn; Xin Peng, Fudan University, Shanghai, China, pengxin@fudan.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA094
https://doi.org/10.1145/3832185

ACM Reference Format:
Xun Zhou, Zhen Dong, Mingyu Ren, Qiang Li, JunJie Li, Sifan Wang, Xiaolong Yu, Chaofeng Sha, and Xin Peng. 2026. RESTOR: Automated Test Oracle Generation for RESTful APIs via Reinforcement Learning. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA094 (October 2026), 22 pages. https://doi.org/10.1145/3832185

1 Introduction

In modern software architectures, REpresentational State Transfer (REST) has established itself as the *de facto* standard for designing networked applications and microservices [11, 17, 42]. These APIs facilitate communication between heterogeneous components via standard HTTP methods and lightweight formats such as JSON. As systems evolve toward complex, distributed ecosystems, the reliability of these interfaces becomes paramount. A single regression in a backend API can propagate cascading failures to front-end applications, resulting in critical service disruptions or revenue leakage. Consequently, automated API testing has become an indispensable component of Continuous Integration/Continuous Deployment (CI/CD) pipelines, aimed at detecting faults efficiently before system release.

However, the rapid iteration cycles characteristic of agile development impose severe constraints on Quality Assurance (QA). In many industrial settings, particularly within fast-paced domains like video editing or content delivery, developers frequently deploy new features or modify existing logic without updating formal documentation. This results in missing, incomplete, or outdated specifications (e.g., OpenAPI/Swagger [1]). Furthermore, for newly deployed interfaces, massive historical execution logs are often unavailable, rendering statistical analysis impossible. This creates a "cold-start" testing scenario: QA engineers sometimes have to write robust test oracles based solely on a single traffic sample (one request-response pair) captured during a manual smoke test, relying on domain knowledge to infer assertion logic.

Existing automated testing approaches fail to adequately address this zero-specification, single-sample constraint. Traditional test oracle generation tools [3, 4] typically depend on accurate specifications to generate schema-valid assertions; when the specification is absent, these tools are rendered ineffective. Similarly, dynamic invariant detection techniques (e.g., AGORA+ [2]) require large-scale historical logs to achieve statistical significance. Applying them to a single sample often yields fragile or trivial assertions that fail to capture complex business logic.

Recently, Large Language Models (LLMs) have shown promise in software testing tasks [26, 34, 40]. Yet, deploying general-purpose Large LLMs (e.g., DeepSeek-V3 [12]) for high-frequency API testing presents significant hurdles. First, the inference latency and financial cost of large parameter models are prohibitive for industrial CI/CD pipelines that execute thousands of tests daily. Second, without specific fine-tuning, general-purpose models often lack the necessary domain rigor; they tend to hallucinate testing logic or over-assert on fields unrelated to business logic, generating "flaky" tests that require constant maintenance. Using smaller, generic models alleviates cost but typically lacks the reasoning capability to generate assertions with high accuracy.

To bridge this gap, we present RESTOR, a novel framework designed to generate actionable, industrial-grade test assertions from a single API traffic sample without reliance on formal specifications or historical logs. Unlike standard Supervised Fine-Tuning (SFT) approaches that require a massive ground-truth corpus of perfect code, our approach leverages Reinforcement Learning (RL). Specifically, we employ Group Relative Policy Optimization (GRPO) [36] to fine-tune a lightweight LLM using an execution-feedback mechanism.

The core intuition behind RESTOR is to model the "common sense" of a human tester. We construct a rewarding pipeline where the agent is optimized for semantic accuracy rather than mere syntax mimicry. By rewarding the agent for correctly handling augmented traffic variations, the model learns to independently distinguish valid data fluctuations from actual errors. This process enables

the model to internalize the comprehensive boundaries of business logic and generate robust assertions, effectively bypassing the need for explicit rule definitions or extensive supervision.

We comprehensively evaluated RESTOR through offline controlled experiments, expert user studies, and online production deployment at ByteDance. On a curated test set of unseen API samples, RESTOR achieved an F_1 score of 85.42% in key field identification, outperforming the large-scale generalist model `DeepSeek-V3.1-Terminus` [13] by offering a superior balance of precision and recall. Regarding semantic correctness, RESTOR generated the highest number of *Exact Match* assertions (663) and minimized missed validations to only 28 cases, effectively solving the coverage issues observed in the base model. In qualitative user studies, domain experts confirmed that RESTOR produced more actionable assertions with significantly reduced noise, requiring fewer manual edits than baseline approaches. Most notably, in the real-world production environment, the deployment of RESTOR drove the Adoption Rate of generated test cases from 74.1% to a sustained level above **96%**, confirming its capability to meet strict industrial reliability standards.

The contributions of this paper are summarized as follows:

- We propose a Reinforcement Learning-based framework, RESTOR, that utilizes GRPO and a novel reward strategy to fine-tune a lightweight LLM. This allows the model to learn rigorous testing logic without requiring a ground-truth corpus of code.
- We provide a comprehensive evaluation and report on the large-scale industrial deployment of RESTOR at ByteDance. To our knowledge, this is the first study to demonstrate the practical efficacy of RL-fine-tuned LLMs for API oracle generation in a production CI/CD environment.

The remainder of this paper is organized as follows: Section 2 provides the background and a motivating example. Section 3 details the RESTOR approach, including dataset construction and model training. Section 4 describes the implementation details. Section 5 presents a comprehensive evaluation, including offline experiments, the qualitative user study, and industrial deployment results. Section 6 discusses related work. Section 7 outlines threats to validity, and Section 8 concludes the paper.

2 Background and Motivating Example

Consider the API `POST /api/subscription/plan_list`, designed to retrieve user subscription details. Figure 1 (left) presents a typical response body containing nested business objects. The response explicitly details the user's status through fields such as `cloud_info` (cloud storage plans) and `vip_info` (premium membership). Key attributes include `is_using`, which indicates whether a specific plan is active; `subscribe_type`, which defines the renewal policy (e.g., "un-auto" for non-renewing or "auto" for recurring billing); and temporal constraints defined by `begin_time` and `end_time`.

In a rapid iterative development environment, server-side code modifications occur frequently, introducing the risk of unintended side effects or regressions. For instance, if `subscribe_type` returns an unrecognized value (e.g., a typo or NULL), the billing system may fail to execute the renewal, directly leading to revenue leakage. Conversely, erroneous formatting could disrupt the front-end display. Consequently, Quality Assurance (QA) engineers are tasked with maintaining a robust suite of test cases to verify that the API functions correctly and that the data integrity involves strict semantic compliance.

To ensure reliability, test cases must go beyond simple HTTP status checks. Figure 1 (right) illustrates a comprehensive test script for the aforementioned API. The assertions verify not only the presence of the `cloud_info` structure (Line 7) but also enforce specific business logic. Crucially, writing these assertions often relies on the QA engineer's "common sense" and domain knowledge. For example, an engineer can infer that `subscribe_type` should belong to a closed set `["auto", "un-auto"]`

```
1 {
2  "ret": "0",
3  "errmsg": "success",
4  "systime": "1751945691302",
5  "log_id": "...",
6  "data": {
7   "cloud_info": {
8    "subscribe_type": "un-auto",
9    "plans": [
10    {
11     "package_name": "1024GB/Monthly ⟷
        Subscription",
12     "begin_time": 1750832254,
13     "end_time": 1753424254,
14     "product_id": "pro_monthly",
15     "is_using": true
16    }
17   ]
18  },
19  "vip_info": {
20   "subscribe_type": "auto",
21   "plans": [...]
22  },
23  ...
24 }
25 }
```

```
1 ... # Test prefix
2
3 # Assume the response body is `resp`
4 # API status assertions
5 assert resp["ret"] == "0"
6 assert resp["errmsg"] == "success"
7 assert "cloud_info" in resp["data"]
8 cloud_info = resp["data"]["cloud_info"]
9 # Common sense: validating enum constraints
10 assert cloud_info["subscribe_type"] in ["auto", ⟷
       "un-auto"]
11
12 for plan in cloud_info["plans"]:
13   # Common sense: validating temporal logic
14   assert 0 < plan["begin_time"] < plan["end_time⟷
       "]
15   assert len(plan["package_name"]) > 0
16   assert len(plan["product_id"]) > 0
17   assert isinstance(plan["is_using"], bool)
18
19 # Similar assertions for `vip_info`
20 assert "vip_info" in resp["data"]
21 vip_info = resp["data"]["vip_info"]
22
23 for plan in vip_info["plans"]:
24   ...
```

Fig. 1. Motivating Example. Left: Response example of `POST /subscription/plan_list`. Right: A manually written test case where QA engineers typically infer implicit constraints (e.g., temporal order or enumerations) relying on domain "common sense".

(Line 10) by observing different parts of the response, and intuitively asserts that a start timestamp should strictly precede an end timestamp (Line 14), even without explicit documentation.

However, manually writing such detailed assertions is labor-intensive and expensive, particularly given the scale of thousands of interfaces, each often containing numerous fields. While automated testing solutions exist, they face significant challenges in our industrial setting. Traditional schema-based tools require formal specifications (e.g., OpenAPI/Swagger), which are often absent or outdated in agile environments. Statistical approaches require massive historical traffic logs to infer patterns. In our scenario, we face a "cold start" problem: the testing platform operates in a black-box manner, relying on a single captured traffic sample (one request-response pair) without access to source code or database schemas.

The heavy reliance on "common sense" for generating meaningful assertions suggests that Large Language Models (LLMs) could be a viable solution, given their semantic reasoning capabilities. Nevertheless, deploying general-purpose Large LLMs (e.g., DeepSeek V3) within a high-frequency CI/CD pipeline is impractical due to high deployment costs and latency. Conversely, smaller, general-purpose models often lack the specific reasoning capabilities required to generate accurate assertions from limited context. To address this trade-off, we propose fine-tuning a lightweight model. By training a smaller model to internalize this testing-specific "common sense," we aim to achieve high-quality assertion generation that satisfies both accuracy requirements and operational constraints (cost and speed).

3 Approach

We present Restor, a framework based on Reinforcement Learning (RL) designed to autonomously generate executable test assertions from isolated API traffic samples, even in the absence of explicit specifications (e.g., OpenAPI) or historical execution logs.

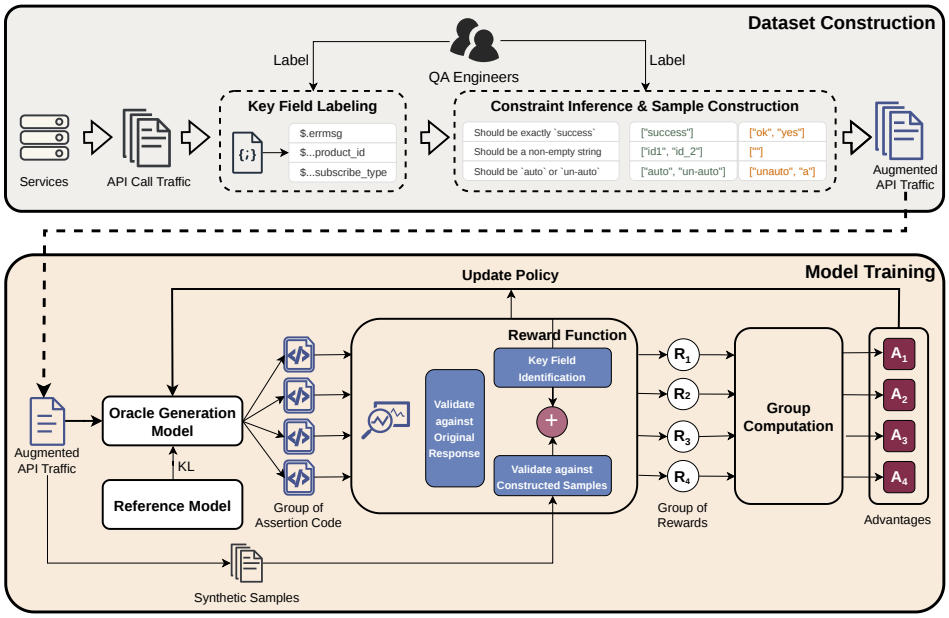


Fig. 2. Overview of the RESTOR framework. The workflow comprises two primary phases: (1) *Dataset Construction* (top) and (2) *Model Training* (bottom).

As illustrated in Figure 2, RESTOR addresses the challenge of creating robust oracles from limited context through two coordinated phases. We first facilitate a data augmentation pipeline designed to enrich raw traffic by identifying key business fields and generating positive and negative samples based on inferred constraints.

Subsequently, we employ an RL-based training process where a Large Language Model (LLM) functions as the policy network. We adopt the Reinforcement Learning paradigm rather than traditional Supervised Fine-Tuning (SFT) for two primary reasons. First, test assertions lack a unique, canonical "ground truth"; the same semantic constraint can be validated through diverse syntactic structures, making token-level imitation inefficient. Second, implicit field constraints are difficult to annotate precisely in a static dataset. RL allows the model to explore the solution space effectively, optimizing for functional correctness and logic coverage rather than strict syntactic matching.

Specifically, we utilize **Group Relative Policy Optimization (GRPO)** [36] to fine-tune the model. As a state-of-the-art on-policy algorithm, GRPO optimizes the policy using group-based relative rewards without the need for an additional value network. This approach significantly reduces the memory footprint and computational overhead compared to traditional Actor-Critic methods (e.g., PPO), thereby providing a cost-effective solution for industrial-scale reasoning tasks while maintaining high generation performance.

3.1 Dataset Construction

To facilitate the training of the agent, we construct an augmented dataset $\mathcal{D}_{\text{aug}}$ derived from real-world production traffic. Each training instance $T^{\text{aug}} \in \mathcal{D}_{\text{aug}}$ is represented as a tuple that encapsulates not only the raw API call traffic but also the semantic metadata required for rigorous

assessment:

$$T^{\text{aug}} = \langle \text{API}, \text{req}, \text{resp}, \mathbb{K}, \mathcal{S}_{\text{pos}}, \mathcal{S}_{\text{neg}} \rangle$$

where:

- $\langle \text{API}, \text{req}, \text{resp} \rangle$ represents the original captured traffic, containing the endpoint identifier, request payload, and the raw server response.
- $\mathbb{K}$ denotes the set of *Key Fields* extracted from the response, which serve as the focal point for assertion generation.
- $\mathcal{S}_{\text{pos}}$ and $\mathcal{S}_{\text{neg}}$ are sets of generated response bodies representing valid (positive) and invalid (negative) variations of the original data, respectively.

The construction of T^{aug} involves two primary phases: identifying which fields require testing ($\mathbb{K}$) and generating data variations ($\mathcal{S}_{\text{pos}}, \mathcal{S}_{\text{neg}}$) to evaluate the quality of the generated assertions.

3.1.1 Collecting Traffic Data. Initially, we collect production traffic records from an internal API recording platform. Each record, denoted as T, encapsulates a single API interaction, comprising the API endpoint identifier (API), the client request payload (req), and the server response body (resp):

$$T = \langle \text{API}, \text{req}, \text{resp} \rangle$$

RESTOR primarily focuses on analyzing the resp component, which is typically a semi-structured document (e.g., JSON) representing a hierarchical collection of field-value pairs.

3.1.2 Key Field Labeling. Raw API responses frequently contain noise, such as transient diagnostic data or tracing identifiers, which are irrelevant to functional correctness. To ensure that the model focuses its attention on business-critical logic rather than structural noise, we filter the response fields to obtain the set $\mathbb{K}$.

To guarantee the reliability and precision of the dataset, the labeling process was performed through manual annotation by three professional QA engineers from ByteDance. To mitigate subjective bias, a majority voting mechanism was implemented to determine the final composition of $\mathbb{K}$. Specifically, each field was independently reviewed by the experts, and only those selected by at least two out of the three annotators were retained. Guided by deep domain expertise, this selection process adheres to three governing principles:

(1) **Entity Identification:** Fields that uniquely identify business entities (e.g., `product_id`) are included to ensure data integrity.
(2) **Domain Logic Constraints:** Experts label fields conveying essential domain logic based on their experience. For example, time intervals (e.g., `begin_time`, `end_time`) and some enumeration fields (e.g., `subscribe_type`), are selected to verify semantic compliance.
(3) **Operational Status:** Fields indicating the outcome of the request (e.g., `ret` codes, `errmsg`) are retained to validate the execution state.

Conversely, dynamic fields such as `log_id` or server timestamps (e.g., `systime`) are explicitly excluded to maintain deterministic testing conditions.

3.1.3 Semantic Constraint Inference and Evaluation Sample Construction. Since the dataset lacks formal specifications (e.g., OpenAPI), we cannot directly determine whether a generated assertion is factually correct. To address this, we construct positive and negative samples to serve as a testbed for evaluating the quality of model-generated assertions. The underlying premise is that a high-quality assertion should pass validation against all positive samples ($\mathcal{S}_{\text{pos}}$) while effectively flagging all negative samples ($\mathcal{S}_{\text{neg}}$).

To ensure the accuracy of the inferred semantic constraints and the representativeness of the constructed data, we employed the same rigorous annotation protocol described in the previous section. Specifically, the three expert QA engineers independently formulated the semantic constraints and designed the corresponding sample variations, with final decisions determined via a majority voting mechanism. For each field in $\mathbb{K}$, this process yields an enriched Natural Language (NL) constraint based on the field's name and its value in the original response. For example, as shown in Table 1, the confirmed constraint for `begin_time` is "Should be a non-negative Unix timestamp." Guided by these verified NL constraints, we then construct two sets of samples:

- **Positive Samples ($\mathcal{S}_{\text{pos}}$):** We construct response bodies where the values of key fields are replaced but remain compliant with the inferred constraints. For instance, replacing the timestamp `1750832254` with `1750832255` creates a valid variation. These samples ensure the generated assertions are not overfitting to the specific values of the single captured traffic.
- **Negative Samples ($\mathcal{S}_{\text{neg}}$):** We construct response bodies containing deliberate violations of the constraints. For example, injecting `-1` into a timestamp field or a string into a boolean field (see Table 1). These samples are crucial for verifying that the generated assertions possess sufficient strictness to detect data anomalies.

For each identified key field, we replace the original field value with three valid ones and three invalid ones to construct response variants. Some fields such as status code may have only one correct value, and we do not construct valid samples for them.

Table 1. Augmentation Examples for Key Fields in `POST /api/subscription/plan_list`

Field	Inferred Constraint	Constructed Positive Value	Constructed Negative Value
`ret`	Should be '0' for success.	`"0"`	`"1"`
`errmsg`	Should be 'success'.	`"success"`	`"internal error"`
`...product_id`	Should be a non-empty string.	`"another_id"`	`""`
`...begin_time`	Should be a non-negative Unix timestamp.	`1750832255`	`-1`
`...subscribe_type`	Should be one of "auto" or "un-auto".	`"auto"`	`"manual"`
`...is_using`	Should be a boolean value.	`false`	`"true"`

3.2 Model Training

3.2.1 Oracle Generation via Group Relative Policy Optimization. We formulate oracle generation as a reinforcement learning task where the policy π_θ generates assertion code conditioned on the API context. To efficiently align the model with test correctness objectives without the computational cost of training a value function, we employ **Group Relative Policy Optimization (GRPO)** [36].

For an API context q sampled from distribution μ, GRPO samples a group of outputs $\{o_1, o_2, \ldots, o_G\}$ from the old policy $\pi_{\theta_{\text{old}}}(\cdot \mid q)$. Let $o_i = (o_{i,1}, \ldots, o_{i,|o_i|})$ denote the token sequence of the i-th output.

The policy is optimized by maximizing the following objective:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{q\sim\mu,\,\{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|q)} \left[\frac{1}{G}\sum_{i=1}^{G} \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \left\{ \min \left[\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})} \hat{A}_{i,t}, \right. \right. \right. \\ \left. \left. \left. \text{clip}\left(\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})}, 1-\epsilon, 1+\epsilon \right) \hat{A}_{i,t} \right] - \beta \mathbb{D}_{KL}\left[\pi_\theta \| \pi_{\text{ref}}\right] \right\} \right]. \tag{1}$$

Here, ϵ is the clipping hyperparameter, β controls the KL-divergence penalty, and $\hat{A}_{i,t}$ is the advantage assigned to the t-th token of output o_i. Unlike PPO, GRPO estimates the advantage from the relative rewards of outputs within the same group and therefore does not require a separate value model.

Following GRPO, the token-level KL divergence between the current policy and the reference policy is estimated as

$$\mathbb{D}_{KL}\left[\pi_\theta \| \pi_{\text{ref}}\right] = \frac{\pi_{\text{ref}}(o_{i,t} \mid q, o_{i,<t})}{\pi_\theta(o_{i,t} \mid q, o_{i,<t})} - \log \frac{\pi_{\text{ref}}(o_{i,t} \mid q, o_{i,<t})}{\pi_\theta(o_{i,t} \mid q, o_{i,<t})} - 1. \tag{2}$$

In our setting, each generated assertion o_i receives an outcome-level reward r_i. The rewards within a group are normalized by subtracting their mean and dividing by their standard deviation. The resulting normalized reward is assigned as the advantage of every token in the corresponding output:

$$\hat{A}_{i,t} = \frac{r_i - \text{mean}(r_1, \ldots, r_G)}{\text{std}(r_1, \ldots, r_G)}, \qquad t = 1, \ldots, |o_i|. \tag{3}$$

This group-relative normalization serves as a baseline and captures the relative quality of each generated assertion without relying on a separate critic network.

Reward Definitions. The scalar reward r_i in Equation (3) evaluates the assertion's executability, key-field coverage, and semantic accuracy. Validating these properties requires interacting with the environment's augmented data samples. We detail the specific reward design in the following part.

3.2.2 Reward Function. To guide the reinforcement learning agent toward generating test oracles that are syntactically correct, semantically precise, and functionally robust, we design a composite reward mechanism. The final reward $\mathcal{R}$ for a generated assertion $\mathcal{A}$ is formulated as a conditional function that first penalizes execution failures and then linearly combines field identification and semantic accuracy, clipped to a normalized range:

$$\mathcal{R}(\mathcal{A}) = \begin{cases} \rho_{\text{fail}} & \text{if Exec}(\mathcal{A}, \text{resp}) = \text{False} \\ \text{clip}\left(\alpha \cdot \mathcal{R}_{\text{ident}}(\mathcal{A}) + \beta \cdot \mathcal{R}_{\text{sem}}(\mathcal{A}), -1, 1\right) & \text{otherwise} \end{cases} \tag{4}$$

where $\text{Exec}(\mathcal{A}, \text{resp})$ denotes the execution result on the original ground-truth response (returning True only if the assertion passes without error), and ρ_{fail} is a fixed penalty (set to -1) for invalid assertions. For executable assertions, the reward is derived from two sub-components: Key Field Identification ($\mathcal{R}_{\text{ident}}$) and Semantic Accuracy ($\mathcal{R}_{\text{sem}}$), balanced by coefficients α and β. The final score is clipped to the interval $[-1, 1]$ to ensure optimization stability during the GRPO training phase. The components are defined as follows:

1. Validity Check (The Gatekeeper). The most fundamental requirement is that the generated assertion must validate the original captured API response successfully. If the assertion $\mathcal{A}$ raises an exception or asserts false on the observed data resp, it is fundamentally flawed. In such cases, the evaluation terminates immediately with the penalty ρ_{fail}, discouraging the model from generating hallucinatory or syntactically invalid code.

2. Key Field Identification ($\mathcal{R}_{ident}$). To ensure the assertion focuses on business-critical logic rather than irrelevant noise (e.g., log IDs), we calculate the overlap between the fields accessed by the assertion code, denoted as $\mathcal{F}(\mathcal{A})$, and the pre-identified key field set $\mathbb{K}$. We quantify relevance using precision and recall:

$$\text{precision} = \frac{|\mathcal{F}(\mathcal{A}) \cap \mathbb{K}|}{|\mathcal{F}(\mathcal{A})|}, \quad \text{recall} = \frac{|\mathcal{F}(\mathcal{A}) \cap \mathbb{K}|}{|\mathbb{K}|}$$

The relevance reward is a weighted sum of precision and recall:

$$R_{\text{ident}}(\mathcal{A}) = w_{\text{prec}} \cdot \text{precision} + w_{\text{rec}} \cdot \text{recall}$$

where w_{prec} and w_{rec} balance the trade-off between avoiding noise and ensuring comprehensive coverage.

3. Semantic Accuracy ($\mathcal{R}_{sem}$). A high-quality oracle must not only accept the observed traffic but also correctly distinguish between valid and invalid data distributions. We evaluate this using the augmented datasets generated in Section 3.1: the positive samples $\mathcal{S}_{\text{pos}}$ (valid variations) and negative samples $\mathcal{S}_{\text{neg}}$ (constraint violations).

Let $KillRate(\mathcal{A}, \mathcal{S}) \in [0, 1]$ represent the **Kill Rate** of assertion $\mathcal{A}$ over a set of samples $\mathcal{S}$, defined as the proportion of samples for which the assertion evaluates to false. This reward is defined as the differential behavior between these two sets:

$$\mathcal{R}_{\text{sem}}(\mathcal{A}) = w_{\text{neg}} \cdot KillRate(\mathcal{A}, \mathcal{S}_{\text{neg}}) - w_{\text{pos}} \cdot KillRate(\mathcal{A}, \mathcal{S}_{\text{pos}}) \tag{5}$$

where w_{neg} and w_{pos} are hyperparameters.

Intuitively, this function rewards the model for maximizing the acceptance of valid positive samples (avoiding false positives/overfitting) while simultaneously maximizing the rejection of invalid negative samples (ensuring strictness/detection capability). An ideal assertion yields a pass rate of 1.0 for $\mathcal{S}_{\text{pos}}$ and 0.0 for $\mathcal{S}_{\text{neg}}$, resulting in a maximal component reward.

4 Implementation

The policy network of RESTOR, responsible for generating assertion code token-by-token, is instantiated using `Doubao-Seed-1.6-flash` [6]. This choice is strategic; Doubao-Seed-1.6-flash is a lightweight and fast large language model developed internally at ByteDance, enabling efficient iteration during training and deployment. To optimize the policy network, we employ GRPO [36] as the fine-tuning algorithm, directly maximizing the expected reward objectives defined in the previous section.

All model training and experimentation were conducted on the internal cloud AI service platform at ByteDance, utilizing standard computational resources for large-scale model development. Regarding the hyperparameter settings for GRPO, the model was fine-tuned for 2 epochs with a learning rate of 1×10^{-6}. To ensure training stability, the KL divergence coefficient was set to 0.001, and the policy clip ratio was fixed at 0.2. During the rollout phase, the number of generations was set to 8, meaning the policy sampled 8 distinct outputs for each training prompt to calculate the group-relative advantage. Due to the substantial computational resources and high financial cost required for reinforcement learning fine-tuning on Large Language Models, the training and evaluation processes were conducted in a single run.

5 Evaluation

To comprehensively evaluate the performance of RESTOR, we investigated the following three research questions:

- RQ1 (**Effectiveness**): How effective is RESTOR in generating test oracles for REST APIs compared to baseline models?
- RQ2 (**Expert Evaluation of Usefulness and Reliability**): How do software testing experts evaluate usefulness, and reliability of the generated assertions?
- RQ3 (**Industrial Application**): How does RESTOR perform when deployed in production environment at ByteDance?

5.1 Experimental Setup

5.1.1 Test Generation Platform. RESTOR is integrated into our internal automated test case generation platform deployed at ByteDance. This platform facilitates a workflow: QA engineers upload anonymized API traffic captured from the production or test environment, and the system generates a complete Python test script containing assertions. Engineers then review the generated code, modifying or accepting it for commit to the version control system. This infrastructure serves as the deployment environment for assessing industrial applicability.

5.1.2 Dataset Construction. We constructed a large-scale dataset of production API traffic sourced from an internal recording platform at ByteDance. The dataset comprises over 2,300 unique traffic samples derived from over 1,500 distinct REST APIs (with each API contributing 1 to 3 traces). This spans 246 distinct services and 15 business lines, ensuring a diverse representation of industrial scenarios ranging from content management to billing systems. To protect privacy and confidentiality, all recorded traffic was anonymized prior to analysis, removing or irreversibly transforming any personally identifiable information and other sensitive identifiers to prevent disclosure of individual users.

The dataset was randomly partitioned into training, validation, and testing sets in an 8:1:1 ratio. The main evaluation (RQ1) is conducted on the held-out test set, consisting of 229 unseen API samples. As detailed in Section 3, ground truth was established through expert annotation, where key fields were explicitly labeled and logical constraints were described in natural language. We utilized a voting consensus mechanism among three QA engineers to validate the semantic alignment of the ground truth. This manual annotation process required substantial effort: the identification of key fields took approximately 4 days, while the definition of semantic constraints and the construction of positive and negative samples required approximately 2 weeks of effort from the annotators.

5.1.3 Baselines. We compare RESTOR against two baseline models that rely solely on prompt engineering without reinforcement learning fine-tuning. To ensure a fair comparison, all models utilize the exact same system prompt and input format, the full details of which are provided in Appendix A. Furthermore, both baseline models operate in standard generation mode with reasoning capabilities disabled.

- **Doubao-Seed-1.6-flash:** We evaluate the original, pre-trained version of the base model. Since this model serves as the initialization for our policy network, comparing against it quantifies the specific performance gains attributable to the GRPO fine-tuning process.
- **DeepSeek-V3.1-Terminus [13]:** We include DeepSeek-V3.1-Terminus (hereafter referred to as DeepSeek), one of the best open-source LLMs, derived from the `DeepSeek-V3` [12] architecture. As a large-parameter foundation model, this baseline represents the capabilities

of a powerful, general-purpose LLM. This comparison evaluates the trade-offs between a specialized lightweight model and a large-scale generalist model.

5.1.4 Evaluation Metrics. We assess model performance across two primary dimensions:

1) Key Field Identification. This metric measures the model's ability to identify business-critical fields ($\mathbb{K}$) while filtering out structural noise (e.g., dynamic timestamps, request IDs). We report *precision*, *recall*, and F_1 *score*.

2) Assertion Accuracy. We evaluate the semantic correctness of the generated assertions. Based on expert review, each generated assertion is categorized as:

- *Exact Match:* The logic is semantically correct and aligns with domain requirements.
- *Overly General:* The assertion is valid but too loose (e.g., checking for non-null instead of a specific enum value).
- *Partial Match:* The code covers some semantic conditions but misses boundary details.
- *Incorrect:* The generated logic is wrong (false positive).
- *Not Validated:* The model failed to generate an assertion for a requisite key field.

3) Industrial Adoption Rate. To quantify the practical utility of RESTOR in the production environment (RQ3), we utilize the **Adoption Rate**, derived from the platform described in Section 5.1. This metric is defined as the ratio of generated test cases that are explicitly accepted and committed by QA engineers to the repository versus the total number of generation tasks. A high adoption rate indicates that the generated assertions meet strict industrial standards with minimal need for manual modification.

5.2 Analysis of RQ1: Effectiveness

In this section, we analyze the quality of generated assertions on the curated test set, focusing on two key dimensions: the ability to identify business-critical fields and the semantic correctness of the generated logic.

5.2.1 Key Field Identification. We first evaluate the capability of the models to correctly identify the set of business-critical fields $\mathbb{K}$, distinguishing them from dynamic noise (e.g., `log_id`, timestamps). Figure 3 visualizes the distribution of precision, recall, and F_1 score across the test set.

As indicated by the distributions, RESTOR demonstrates a superior balance between coverage and selectivity. By leveraging GRPO to internalize the distinction between signal and noise, RESTOR achieves a precision of 81.30% and maintains a competitive recall of 96.15%, consequently attaining the highest F_1 score of 85.42%. The density distribution in Figure 3 corroborates this stability, as the F_1 score violin for RESTOR is structurally tighter and concentrated at higher values compared to the dispersed distributions of the baseline models.

In comparison, the baseline models exhibit distinct performance trade-offs. As indicated by the recall distribution, the large-scale model DeepSeek achieves the highest average recall of 98.64%, reflecting its broad semantic coverage; however, its corresponding precision is notably limited at 67.57%, indicating a tendency to over-generate assertions for irrelevant fields. Meanwhile, the base Doubao-flash model performs worse, exhibiting low precision (68.54%) and a moderate F_1 score (77.29%).

To rigorously validate these comparisons, we conducted a one-tailed Wilcoxon signed-rank test ($N = 229$, confidence level 0.95). The statistical results confirm that RESTOR achieves significant improvements in both precision and F_1 score when compared to Doubao-flash ($p < 0.001$) and DeepSeek ($p < 0.001$). While DeepSeek exhibits a slightly higher average recall, its significantly lower precision ultimately yields a statistically inferior F_1 score.

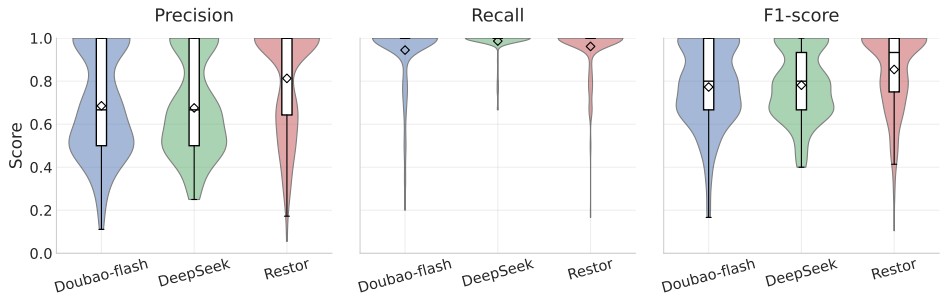


Fig. 3. Distribution of Key Field Identification Performance. The visualization integrates violin plots to show data distribution density, box plots to indicate quartiles, and white diamond markers to denote the mean score.

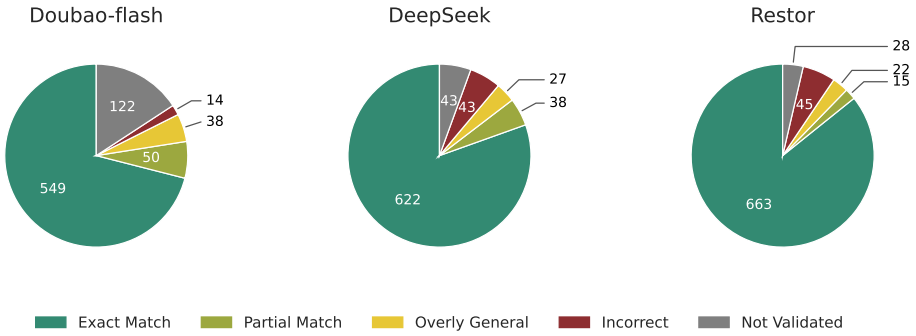


Fig. 4. Comparative Analysis of Assertion Accuracy. This distribution quantifies the semantic correctness of the generated test code, highlighting the volume of exact matches versus missed validations.

5.2.2 Assertion Accuracy. Figure 4 presents the breakdown of assertion quality into five categories as defined in the experimental setup.

RESTOR exhibits the highest efficacy, generating **663** *Exact Match* assertions, surpassing both DeepSeek (622) and Doubao-flash (549). Critically, RESTOR maximizes coverage completeness by minimizing the *Not Validated* category. While Doubao-flash failed to validate 122 requisite fields and DeepSeek missed 43, RESTOR reduced this figure to just **28**. This indicates that the RL fine-tuning process successfully aligned the output with the strict logical boundaries required for automated testing, reducing the need for manual correction.

Handling False Positives and Non-determinism. A critical factor in the practical usefulness of automated test generation is the rate of false positives (categorized here as *Incorrect* assertions). Our evaluation confirms that such occurrences are exceptionally rare with RESTOR. Furthermore, dynamic and non-deterministic fields (e.g., system timestamps or request IDs) are explicitly filtered during training and inference, preventing the generation of flaky assertions. In the production environment, any residual false positives are effectively mitigated through a "human-in-the-loop" mechanism: QA engineers review and refine the generated test cases prior to committing them to the code repository, ensuring that flawed assertions never disrupt automated CI/CD pipelines.

Table 2. Comparison of Generated Assertions for Key Fields. For clarity, the `assert` keyword is omitted, and specific hierarchical field paths (e.g., `resp['data']['space_info']['quota']`) are abstracted as `val`.

Target Field	Ground Truth Constraint	Doubao-flash	DeepSeek	RESTOR
`errmsg`	Should be 'success'.	**Exact Match:** `val == 'success'`	**Exact Match:** `val == 'success'`	**Exact Match:** `val == 'success'`
`subscribe_type`	Should be one of 'auto' or 'un-auto'.	*(Not Validated)*	**Overly General:** `len(val) > 0`	**Exact Match:** `val in ['un-auto', 'auto']`
`space_info.quota`	Should be a string representing a non-negative integer.	**Overly General:** `'quota' in space_info`	**Partial Match:** `int(val) >= 0`	**Exact Match:** `isinstance(str) and int(val) >= 0`
`plan.begin_time`	Should be a non-negative Unix timestamp.	**Overly General:** `'begin_time' in plan`	**Partial Match:** `int(val) >= 0`	**Partial Match:** `val >= 0`
`systime` / `log_id`	***Noise:*** *Dynamic system fields; should be ignored.*	*(Ignored)*	**False Positive:** `int(val) > 0` `len(val) > 0`	*(Ignored)*

5.2.3 Case Study. To qualitatively assess model performance, we analyze assertions generated for the API `POST /api/subscription/plan_list`. Table 2 presents a comparative analysis of how RESTOR and the baselines handle specific critical fields and dynamic noise.

The baseline models exhibit distinct failure patterns. Doubao-flash predominantly relies on shallow structural checks, often validating only the existence of keys (e.g., `'quota' in space_info`) rather than their values, which limits the depth of testing. Conversely, DeepSeek, creates two issues despite its reasoning capability: (1) Generic Constraints, where it defaults to weak checks (e.g., `len > 0`) instead of inferring Enums or value ranges, and (2) Noise Sensitivity, where it generates assertions for unstable system fields (e.g., `log_id`), leading to flaky tests in production.

In contrast, RESTOR demonstrates the ability to infer strict semantic boundaries. It correctly identifies enumeration constraints (e.g., `subscribe_type`), enforces semantic type validation on string-encoded numbers, and successfully filters out volatile system fields, ensuring the assertions are both rigorous and stable.

Answer to RQ1

RESTOR significantly improves upon prompt-based baselines in automated assertion generation. It achieves the highest F_1 score (**85.42%**) in identifying business-critical fields, effectively filtering dynamic noise that plagues large general-purpose models. Furthermore, RESTOR produces the highest volume of *Exact Match* assertions while minimizing missed validations, demonstrating that GRPO fine-tuning enables a lightweight model to generate stricter, more complete, and production-ready test oracles.

5.3 Analysis of RQ2: Expert Evaluation of Usefulness and Reliability

This section evaluates the practical utility of Restor in real-world testing workflows. We conducted a study on 48 cases randomly sampled from our internal automated test case generation platform. Users uploaded API traffic and used our platform to generate these cases, which exhibit higher structural variance and noise than the curated dataset. Different from the curated setting in RQ1, we collect both (i) quantitative evaluation results and (ii) qualitative feedback from users, including their perceived key fields, preferred assertion strictness, and additional usability concerns such as generation latency, code style, and maintainability. The goal is to determine whether the generated oracles are useful for our users.

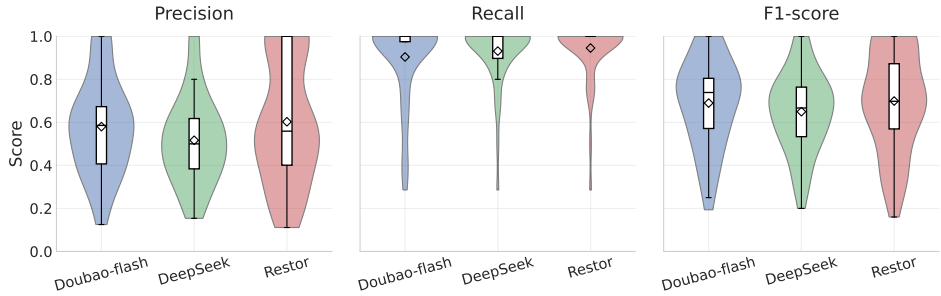


Fig. 5. Distribution of Key Field Identification Metrics in User Study. The figure integrates violin plots (density), box plots (quartiles), and diamond markers (mean) to visualize model performance variance on real-world data.

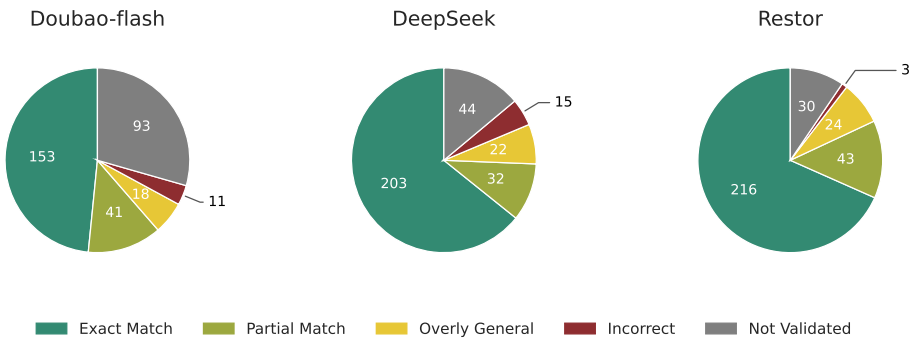


Fig. 6. Distribution of Assertion Accuracy Categories. The chart quantifies the semantic correctness of the generated logic, highlighting the volume of usable assertions (Exact Match) versus errors (Incorrect) and omissions (Not Validated).

5.3.1 Key Field Identification. Similar to RQ1, we evaluate key field identification using precision, recall, and F_1 score against expert-reviewed key field labels.

Figure 5 presents the detailed distribution of these metrics. Specifically, Restor achieves a precision of 60.27%, a recall of 94.54%, and an F_1 score of 69.91%, demonstrating a concentrated distribution that suggests improved robustness on noisy industrial traffic. In contrast, DeepSeek yields a mean recall of 93.13% but a lower precision of 51.67%. These results indicate that while DeepSeek

adopts an aggressive selection strategy that frequently includes non-deterministic fields requiring manual filtering, RESTOR achieves the best overall balance between coverage and selectivity.

5.3.2 Assertion Accuracy. Beyond field selection, we evaluate whether the generated assertions match expert expectations. Figure 6 summarizes the semantic quality of assertions using the same five categories defined in Section 5.1.

RESTOR generates the largest number of *Exact Match* assertions (216), exceeding DeepSeek (203) and Doubao-flash (153). Moreover, RESTOR exhibits high reliability, producing only 3 *Incorrect* assertions across all cases, while DeepSeek generates 15 incorrect assertions. These results indicate that RESTOR yields more directly usable test logic and reduces the risk of false positives that can interrupt CI/CD pipelines.

5.3.3 Categorized Analysis of Expert Feedback. To complement the quantitative findings, we conducted a qualitative user study involving 13 QA engineers across 11 distinct business lines. Participants were asked to evaluate the generated oracles based on four specific criteria: (1) *Relevance* (whether the oracles focus on the critical fields actually monitored in production), (2) *Accuracy* (the factual correctness of the assertions), (3) *Experience* (satisfaction with generation speed and system latency), and (4) *Open Feedback*. We categorized their insights into four recurring core themes regarding operational utility and boundary cases:

- **Actionable Oracle Generation:** Experts noted that baseline models often rely on shallow structural checks, such as generic existence or type validation. In contrast, RESTOR significantly reduces manual post-editing effort by producing deeper, semantically rigorous assertions.
- **High Execution Efficiency:** Most users highlighted a substantial improvement in generation speed compared to large-scale general LLMs, particularly when generating comprehensive test cases with numerous assertions. This reduction in latency enhances developer experience and facilitates seamless iterative testing within daily workflows.
- **Payload Scale Boundaries:** In a few extreme instances (3/48), the generated assertions became relatively verbose when processing complex response bodies with exceptional field cardinality (> 400 fields), which outstrips the typical training distribution (< 100 fields). Under such sparse edge cases, the assertion density increases, occasionally requiring minor manual refactoring to optimize long-term maintainability.
- **Context-Dependent Domain Constraints:** Certain highly specialized business rules (e.g., proprietary, complex billing calculations) could not be entirely inferred due to the deterministic black-box nature. Future work will explore incorporating lightweight domain-specific prompts during inference to address this without retraining.

Answer to RQ2

Expert evaluation confirms that RESTOR outperforms baselines in both reliability and utility for real-world testing. Quantitatively, it achieves the highest semantic accuracy with minimal errors and a superior balance in key field identification. Qualitatively, users report significantly reduced manual editing efforts and improved generation efficiency, validating the model's effectiveness for industrial workflows despite minor limitations with extreme payload sizes.

5.4 Analysis of RQ3: Industrial Application

To evaluate the performance of RESTOR in the production environment, we monitored the system's usage on the test generation platform (introduced in Section 5.1) starting from early November

2025. We analyze the **Adoption Rate** across two dimensions: temporal trends over five bi-weekly intervals and distribution across distinct business lines.

5.4.1 Temporal Trend of Adoption Rate. To assess the impact of RESTOR on testing efficiency over time, we analyzed the adoption trends across five bi-weekly intervals from mid-October 2025 to mid-December 2025. Figure 7 illustrates the trajectory of user acceptance before and after the model deployment.

As illustrated in Figure 7, the adoption rate stood at 74.1% prior to the full deployment of the fine-tuned model (before November 8). Following the rollout, the system demonstrated an immediate and substantial performance leap, with the adoption rate surging to 92.6% in the initial post-deployment interval. This upward trajectory continued steadily, stabilizing above 96% throughout December. This sustained high acceptance rate confirms the effectiveness of the GRPO fine-tuning, as the model consistently generates assertions that meet strict production standards, thereby minimizing the need for manual intervention.

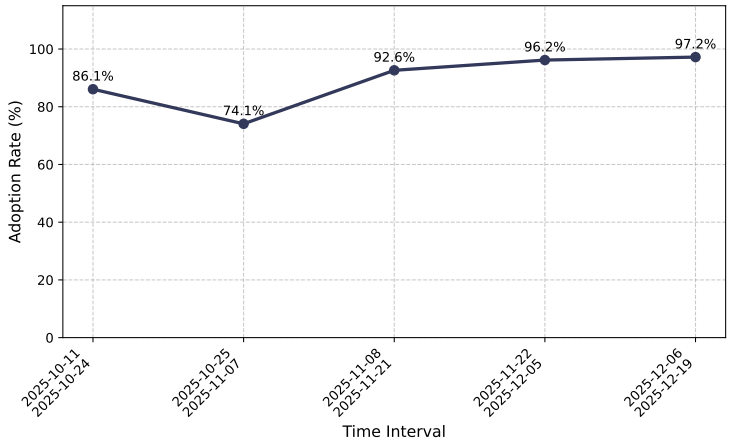


Fig. 7. Temporal trends of adoption rate. The deployment of the fine-tuned model (starting early November) correlates with a sustained improvement in adoption, stabilizing above 97% in the final interval.

5.4.2 Adoption Rate Across Business Lines. We further decomposed usage metrics by business line to assess domain adaptability. Figure 8 presents the adoption rates and generation volumes across ten business lines (BL01–BL10). A 'task' represents a single test generation request, which is initiated when a QA engineer uploads a sampled request-response traffic trace to the platform.

The results demonstrate a usage pattern concentrated in core services. BL02 and BL03, which represent the platform's primary workload, achieved adoption rates of 97.0% (1,064 tasks) and 93.2% (117 tasks), respectively. This confirms that RESTOR successfully captures the domain logic for the system's most active users. Conversely, lines with low task volumes (BL04–BL10) exhibit higher variance. Analysis suggests this stems from two factors: specific lines with highly complex, context-heavy logic currently outside the model's scope, and new engineering teams still in the platform onboarding phase. Overall, the tool achieves high fidelity in the dominant production scenarios.

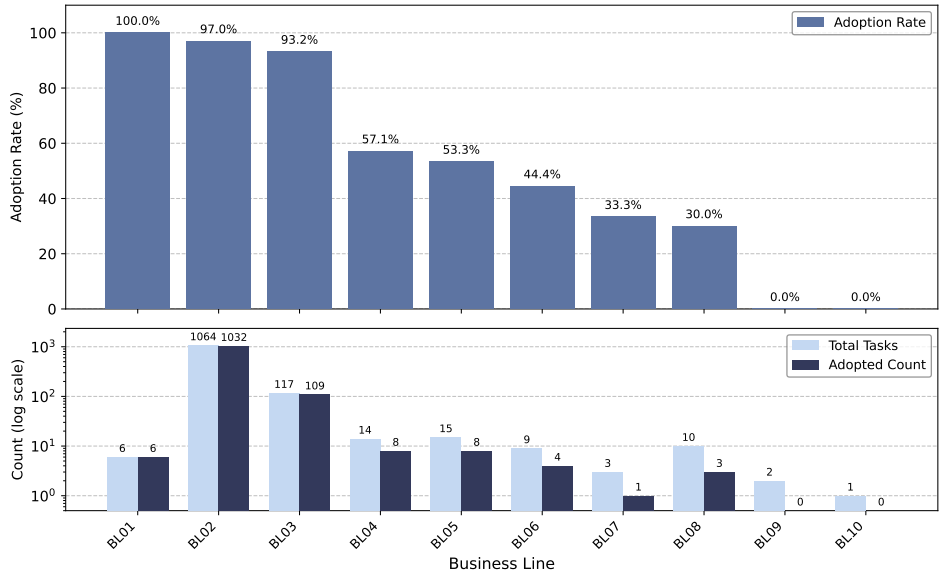


Fig. 8. Adoption rates (top) and task volumes (bottom) distributed by Business Lines. Usage is concentrated in core lines (BL02, BL03) which show high adoption (>93%).

Answer to RQ3

The industrial deployment confirms the practical value of RESTOR. Following the model's release, the global adoption rate improved from 74.1% to a stable level exceeding 96%. Furthermore, in core business lines that constitute the majority of production traffic, the tool achieved an acceptance rate greater than 93%. These results indicate that the fine-tuned model effectively delivering reliable and low-maintenance assertions for QA engineers.

6 Related Work

6.1 Automated Testing of REST APIs

Automated testing of REST APIs primarily adopts a black-box approach, generating HTTP request sequences to maximize code coverage or uncover faults [5, 11, 25, 42]. The majority of these techniques rely heavily on formal specifications, such as the OpenAPI Specification (OAS), to derive valid inputs and expected behaviors [1, 21, 29, 35]. In the absence of specifications, tools must infer schemas from traffic, often struggling with complex dependencies.

A critical challenge in this domain is the *oracle problem*. Traditional tools typically employ implicit oracles restricted to detecting crashes (e.g., 5XX status codes) or specification violations [20, 28], failing to verify domain-specific business logic. To bridge this gap, recent works utilize either static or dynamic inference. SATORI [3], a static approach, employs LLMs to deduce behavioral rules from OAS descriptions. Conversely, dynamic invariant detection methods like AGORA+ [2] mine patterns from massive execution logs. However, both paradigms have significant limitations: static methods depend on accurate, up-to-date specifications, while dynamic methods require extensive, diverse traffic to avoid overfitting. Our work targets a strictly colder start scenario—generating strict assertions from a single request-response sample without access to specifications or historical logs.

6.2 Test Oracle Generation

General automated oracle generation often relies on white-box or grey-box information, such as source code analysis [14, 23], method-level contracts [9, 41], or rich documentation [18]. For instance, many mature techniques operate at the method or unit level within specific programming languages (mainly Java), leveraging internal information such as dataflow, variable names, and code structure to derive assertions [9, 14, 23, 30, 38, 41]. While effective for unit testing, these approaches are inapplicable to black-box REST API testing where only inputs and outputs are accessible.

Dynamic invariant detection remains the closest predecessor to our work for black-box systems [2, 4]. Yet, as noted above, its reliance on large-scale execution history makes it unsuitable for validating new features with scarce data. Recently, Large Language Models have been applied to oracle generation [22, 31]. However, most LLM-based solutions rely on prompt engineering with large models, which incurs high latency and cost, or require code context not available in black-box settings. Ours is distinct in leveraging Reinforcement Learning to fine-tune a lightweight model. This enables the agent to internalize testing semantics and generate executable assertions efficiently, bypassing the need for huge contexts or expensive general-purpose models.

6.3 Reinforcement Learning in Test Generation

Reinforcement Learning (RL) has proven effective in software testing by training agents to maximize objectives like code coverage or crash diversity. RL-based test generation has been successfully applied across diverse domains, including the validation of complex fundamental systems (e.g., compiler testing [8], cyber-physical systems [43]), domain-specific applications such as autonomous driving and robotics [10, 16, 24, 27, 39], and general-purpose softwares like Android applications [7, 15, 19, 33, 37] and CI systems [32]. In these contexts, RL agents are typically trained to generate effective test inputs or environment configurations that maximize a calculated reward, such as crash probability [27] and coverage [24]. The central objective in these works remains optimizing the sequence or structure of the inputs fed to the system under test to achieve higher fault detection.

Critically, existing RL-based testing works focus almost exclusively on *input generation*. The application of RL to the *oracle problem*, i.e., verifying the correctness of the output, remains unexplored. To the best of our knowledge, our work represents the first application of RL specifically for generating test oracles for REST APIs in a black-box context. Instead of optimizing input fuzzing, we train the model to distinguish strict logic boundaries, transforming raw responses into high-quality, executable verification code.

7 Threats to Validity

We define potential threats to the validity of our study and discuss the mitigation strategies employed to address them.

7.1 Internal Validity

Human Annotation Bias. The reliance on human judgment for test oracle validation (RQ1 and RQ2) introduces risks of subjectivity and fatigue. To mitigate this, we employed three senior QA engineers from ByteDance. We implemented a strictly blinded consensus mechanism, where ground truth labels were established only upon majority agreement. This approach minimizes individual bias and ensures the evaluation aligns with objective industrial standards.

Data Leakage. To ensure the model learns generalized logic rather than memorizing training data, we adopted a two-tiered strategy. First, for RQ1, we enforced strict contamination control by physically excluding test set API endpoints from the training corpus. Second, the real-world

evaluation in RQ2 utilizes traffic from *newly developed* features not yet integrated into the traffic recording platform. Consequently, these samples are inherently disjoint from the training distribution, effectively precluding data leakage.

Validity of Adoption Rate Metric. In RQ3, we use adoption rate as a proxy for utility. A potential threat is that users might commit generated code without sufficient scrutiny. However, the deployment workflow at ByteDance mandates peer code review. Therefore, committed assertions represent code that has satisfied not only the author but also independent human reviewers, validating the practical quality of the generation.

7.2 External Validity

Generalizability of Source Data. Our dataset originates entirely from ByteDance's production environment, posing a risk of overfitting to specific corporate conventions (e.g., specific envelope structures or naming schemes) that may not apply to other organizations. To proactively mitigate this threat and ensure broader generalizability, we intentionally sampled data across 15 distinct business lines and 246 diverse services. This extensive variety encompasses a wide spectrum of RESTful design patterns, payload complexities, and data logic prevalent in the broader software industry. While the deployed model may inevitably learn certain internal formatting conventions, the underlying reinforcement learning methodology for internalizing oracle generation logic is highly generalizable. We assert that this framework can be readily transferred to other organizational contexts and API paradigms (such as RPC) with appropriate data collection and constraint definitions.

Language Specificity. The current implementation of Restor generates test assertions exclusively in Python. This limits the direct application of our fine-tuned model to environments using other technology stacks (e.g., Java/JUnit or JavaScript/Jest). However, the core contribution of this work lies in the reinforcement learning methodology that aligns LLMs with testing logic ("common sense"), rather than language syntax. We believe the proposed approach can be readily adapted to other programming languages by substituting the ground truth samples and execution feedback mechanisms in future work.

8 Conclusion

In this paper, we presented Restor, a reinforcement learning-driven framework designed to automate the generation of precise semantic test assertions for REST APIs. By adapting a lightweight Large Language Model via GRPO, our approach effectively internalizes domain-specific testing logic, enabling the system to identify business-critical fields and infer strict logical constraints from single traffic samples without relying on formal specifications. Extensive experiments demonstrate that Restor significantly outperforms both prompt-engineered baselines and large-scale general models (e.g., DeepSeek[13][12]) in terms of key field identification precision and assertion safety. Furthermore, the successful deployment of Restor within ByteDance's production testing platform—achieving a sustained adoption rate exceeding 90% across core business lines—validates its practical utility and robustness in accelerating industrial testing workflows.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant No. 2024YFB4505902.

A Prompt Templates

System Prompt Template for Test Oracle Generation

Role

You are a professional API testing expert. You are skilled at generating high-quality assertions based on limited API information with your professional knowledge.

Task

Your task is to analyze the relevant information of an API, combine it with some historical traffic, and generate assertions for this API.

Instructions

Please generate assertions implemented in Python. The overall requirements are as follows:
• Based on the API information and your understanding, select important fields from the response body to assert.
• Assertion types include value equality, value range checks, object field existence checks, etc.
• The assertions you generate should be applicable to all scenarios of this API, not limited to the specific response body provided in the example traffic.
• Generate Python assertion statements only.

Notes

For selecting assertion fields, it is recommended to:
• Focus on the core resources expected to be returned by the API, for example, `users` in the response body of `getUsers`.
• Choose fields with clear semantic constraints, such as URIs (format validation), counters (non-negative numbers), etc.
• DO NOT consider: timestamps or other random fields, meaningless fields.
For assertions on different types of fields, after understanding the field semantics and inferring their constraints:
• Perform format validation for strings, range validation for numbers.
• For enum-like fields, you may further perform set membership validation.
• If it is truly difficult to determine constraints, only provide type validation.

Input and Output Format

...

Data Availability

The research presented in this paper was conducted within ByteDance, utilizing proprietary industrial datasets and internal closed-source models. Due to strict corporate non-disclosure agreements and information security policies regarding user privacy and intellectual property, the source code, raw data, and trained models cannot be made publicly available.

References

[1] 2025. OpenAPI Specification. https://www.openapis.org. Accessed: November 2025.
[2] Juan C. Alonso, Michael D. Ernst, Sergio Segura, and Antonio Ruiz-Cortés. 2025. Test Oracle Generation for REST APIs. *ACM Trans. Softw. Eng. Methodol.* (March 2025). doi:10.1145/3726524 Just Accepted.
[3] Juan C Alonso, Alberto Martin-Lopez, Sergio Segura, Gabriele Bavota, and Antonio Ruiz-Cortés. 2025. SATORI: Static Test Oracle Generation for REST APIs. *arXiv preprint arXiv:2508.16318* (2025).
[4] Juan C Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: automated generation of test oracles for REST APIs. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1018–1030.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

[6] ByteDance. 2025. Doubao-Seed-1.6-flash Large Language Model. https://console.volcengine.com/ark/region:ark+cn-beijing/model/detail?Id=doubao-seed-1-6-flash. Accessed: 2026-01-13.

[7] Xiaobao Cai, Zhen Dong, Yongjiang Wang, Abhishek Tiwari, and Xin Peng. 2024. Reproducing Timing-dependent GUI Flaky Tests in Android Apps via A Single Event Delay. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[8] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler test-program generation via memoized configuration search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2035–2047.

[9] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting static analysis accuracy with instrumented test executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1154–1165.

[10] Tianyi Chen, Qidi Wang, Zhen Dong, Liwei Shen, and Xin Peng. 2023. Enhancing Robotic Program Synthesis Through Environmental Context. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*.

[11] Yiru Chen, Chenxi Zhang, Zhen Dong, Dingyu Yang, Xin Peng, Jiayu Ou, Hong Yang, Zheshun Wu, Xiaoun Qu, and Wei Li. 2023. Dynamic Graph Neural Networks-based Alert Link Prediction for Online Service Systems. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[12] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[13] DeepSeek-AI. 2025. DeepSeek-V3.1-Terminus: A Specialized Large Language Model. https://huggingface.co/deepseek-ai/DeepSeek-V3.1-Terminus. Version 3.1-Terminus.

[14] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[15] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering (ICSE)*.

[16] Andréa Doreste, Matteo Biagiola, and Paolo Tonella. 2024. Adversarial testing with reinforcement learning: A case study on autonomous driving. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 293–304.

[17] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures*. University of California, Irvine.

[18] Gregory Gay, Sanjai Rayadurgam, and Mats PE Heimdahl. 2014. Improving the accuracy of oracle verdicts through automated model steering. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 527–538.

[19] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and Fixing Data Loss Issues in Android Apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[20] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web api schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 345–346.

[21] Jie He, Ezio Bartocci, Dejan Ničković, Haris Isakovic, and Radu Grosu. 2022. Deepstl: from english requirements to signal temporal logic. In *Proceedings of the 44th International Conference on Software Engineering*. 610–622.

[22] Yibo He, Jiaming Huang, Hao Yu, and Tao Xie. 2024. An empirical study on focal methods in deep-learning-based approaches for assertion generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1750–1771.

[23] Soneya Binta Hossain and Matthew Dwyer. 2024. Togll: Correct and strong test oracle generation with llms. *arXiv preprint arXiv:2405.03786* (2024).

[24] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. 2024. Reinforcement learning informed evolutionary search for autonomous systems testing. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–45.

[25] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2025. Llamaresttest: Effective rest api testing with small language models. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 465–488.

[26] Jiageng Li, Zhen Dong, Chong Wang, Haozhen You, Cen Zhang, Yang Liu, and Xin Peng. 2025. LLM Based Input Space Partitioning Testing for Library APIs. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[27] Chengjie Lu, Yize Shi, Huihui Zhang, Man Zhang, Tiexin Wang, Tao Yue, and Shaukat Ali. 2022. Learning configurations of operating environment of autonomous vehicles to maximize their collisions. *IEEE Transactions on Software Engineering* 49, 1 (2022), 384–402.

[28] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In *International Conference on Service-Oriented Computing*. Springer, 459–475.

[29] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 682–685.

[30] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. Evospex: An evolutionary algorithm for learning postconditions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1223–1235.

[31] Davide Molinelli, Alberto Martin-Lopez, Elliott Zackrone, Beyza Eken, Michael D Ernst, and Mauro Pezzè. 2025. Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1887–1909.

[32] Paulina Stevia Nouwou Mindom, Amin Nikanjam, and Foutse Khomh. 2023. A comparison of reinforcement learning frameworks for software testing tasks. *Empirical Software Engineering* 28, 5 (2023), 111.

[33] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 153–164.

[34] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[35] Sergio Segura, Juan C Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. 2022. Automated generation of metamorphic relations for query-based systems. In *Proceedings of the 7th International Workshop on Metamorphic Testing*. 48–55.

[36] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[37] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[38] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1398–1409.

[39] Tianyi Wu, Liwei Shen, Zhen Dong, Xin Peng, and Wenyun Zhao. 2024. Synthesizing Programmatic Policy for Generalization within Task Domain. In *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI)*.

[40] Haozhen You, Jingjing Wang, Qiang Li, Xin Peng, and Zhen Dong. 2026. Industrial Practice of LLM-based Test Case Carving and Assertion Generation. In *Proceedings of the 35th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[41] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering*. 163–174.

[42] Chenxi Zhang, Zhen Dong, Xin Peng, Bicheng Zhang, and Miao Chen. 2024. Trace-based Multi-Dimensional Root Cause Localization of Performance Issues in Microservice Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*.

[43] Shaohua Zhang, Shuang Liu, Jun Sun, Yuqi Chen, Wenzhi Huang, Jinyi Liu, Jian Liu, and Jianye Hao. 2021. FIGCPS: Effective Failure-inducing Input Generation for Cyber-Physical Systems with Deep Reinforcement Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 555–567. doi:10.1109/ASE51524.2021.9678832

Received 2026-01-30; accepted 2026-04-16