AgenticNorm: Multi-Agent Lightweight Anomaly Detection for Web Applications

ANONYMOUS AUTHOR(S)

Web applications support important areas such as finance, e-commerce, and healthcare, making their reliability and security of paramount importance. However, web frontends are inherently manipulable, and abnormal client behaviors may evade backend checks, creating exploitable vulnerabilities. Log analysis has emerged as an effective defense by capturing client-server interactions. Two main categories of log analysis methods exist: model-learning-based approaches that train predictive models from normal logs, and rule-learning-based approaches that mine logical constraints from logs. However, these two categories have their limitations. Model-learning-based methods detect anomalies by training predictive models on normal logs, but they suffer from poor interpretability, high false positive rates, and difficulty in capturing subtle attacks. Rule-learning-based methods provide stronger interpretability by extracting explicit constraints, as exemplified by WebNorm, but they rely heavily on program instrumentation, heavyweight proprietary models, and manually engineered prompts.

In this paper, we present AgenticNorm, a lightweight anomaly detection framework built upon lightweight, locally deployable LLMs. AgenticNorm avoids program-analysis dependence, removes reliance on heavyweight proprietary models, and mitigates prompt sensitivity through three innovations: (1) eliminating source-code dependence via frequency-based inter-API relation discovery, (2) reducing log complexity through field clustering into semantically coherent groups, and (3) iteratively refining prompts with adversarially generated attack logs. These components are integrated into a multi-agent workflow that progressively improves anomaly detection without extensive human intervention.

We implement and evaluate AgenticNorm on popular benchmarks, including TrainTicket and NiceFish. Results demonstrate that AgenticNorm achieves effective and interpretable anomaly detection while requiring significantly less contextual information compared to existing approaches.

CCS Concepts: • **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Web application security**; • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Web application security, Log anomaly detection, Rule learning

ACM Reference Format:
Anonymous Author(s). 2018. AgenticNorm: Multi-Agent Lightweight Anomaly Detection for Web Applications. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Web applications play a critical role in modern infrastructures, supporting domains such as finance [13, 44], e-commerce [37], and healthcare [22]. Their reliability and security are of paramount importance. Unfortunately, web frontends are inherently manipulable: attackers can alter client-side code or parameters to bypass validations, tamper with workflows, or inject attack behaviors.

In principle, backend applications implement authorization checks and other safeguards to prevent such manipulations. However, because attack behaviors are often difficult to exhaustively

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

covered through conventional frontend testing, certain attack actions may escape detection by the backend, leaving exploitable vulnerabilities. To mitigate such risks, log analysis has become an effective defense mechanism. Logs in web systems typically record detailed interactions between clients and servers, including API calls, request parameters, and response statuses. Analyzing these logs enables the identification of attack interaction patterns that may signal security threats [2, 49]. Current state-of-the-art log analysis methods can be broadly classified into two categories: (1) *model-learning-based approaches*, which train predictive models from normal logs and use them to detect anomalies [1–3, 6, 7, 20, 21, 25, 27, 32, 35, 40–42, 45, 46], and (2) *rule-learning-based approaches*, which mine logical constraints from logs and detect violations [23].

The first category, model-learning-based approaches, typically employ deep learning models to classify logs as normal or attack [8, 10]. While effective in many scenarios, these approaches suffer from limited interpretability: the output is often a binary decision without clear explanations of the underlying cause. Given the large volume of logs in practice, even a small false positive rate can result in overwhelming numbers of alerts, complicating deployment. Moreover, these methods often struggle to detect subtle anomalies [33]; when malicious modifications closely mimic normal behaviors, the models tend to misclassify them, leading to missed detections.

To address these limitations, rule-learning-based approaches attempt to extract explicit logical constraints from logs and use them for anomaly detection [23]. Existing rule-learning-based approaches, such as WebNorm [23], combine program analysis and LLM inference: they first analyze source code to identify how data flow across APIs, and then employ manually crafted prompts to guide the LLM in generating explicit constraints. For example, WebNorm can infer constraints like API1's input field A should match API2's output field B. This provides strong interpretability, as a violated constraint points to the concrete reason for detection.

- **Dependence on program analysis and source code**: it requires access to frontend/backend code and derives event relationships using program analysis by analyzing how data flow across API calls and execution traces, which is costly for deployment in new systems, and infeasible for closed-source or third-party components.
- **Reliance on heavyweight proprietary LLMs**: constraint confirmation and synthesis depend on heavyweight proprietary models, which introduce latency, cost, and compliance/privacy risks. Furthermore, real logs are long and deeply nested, often exceeding the context windows of lightweight models and forcing reliance on heavyweight remote services.
- **Prompt sensitivity**: generating correct constraints often requires carefully crafted prompts. This results in project-specific manual engineering with poor transferability and unstable outcomes.

In this paper, we propose AgenticNorm, a lightweight anomaly detection framework that retains the interpretability of rule-based methods while removing the barriers to deployment. Our design proceeds in three steps that directly address the above challenges. First, instead of relying on program analysis, we operate entirely on logs and uncover inter-API relationships through frequency-based analysis of co-occurring calls, deriving constraints directly from runtime behaviors. Second, to cope with the long and nested nature of real-world logs, we introduce **field clustering**, which expands nested entities into flattened fields and groups them into semantically coherent clusters. By invoking the LLM over one cluster at a time, the system significantly reduces input length while preserving meaningful comparisons, enabling lightweight models to handle large-scale logs. Third, to overcome prompt sensitivity, we employ an iterative loop of **prompt refinement via generated attacks**. Adversarial logs are synthesized to reveal missing constraints, and these counterexamples enrich the prompts with more explicit and diverse descriptions. As a result, refined prompts progressively capture a broader range of conditions, reduce noise, and improve recall without relying on manual engineering.

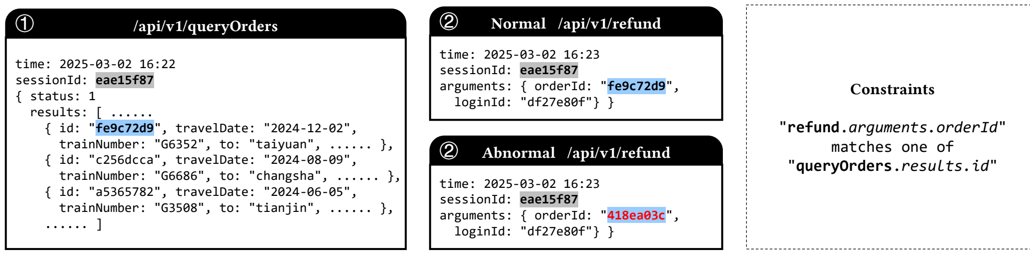


Fig. 1. A motivating example from the TrainTicket dataset illustrating a ticket refund workflow, where the user first retrieves a list of orders using `queryOrders` and then requests a refund through `refund`. The blue highlights indicate identifiers that should match across the two APIs, while the red highlight marks an attack case, where the `orderId` is replaced with a value not present in the queried list.

AgenticNorm integrates these components into a multi-agent workflow, where agents for constraint generation, attack generation, and prompt refinement cooperate iteratively. The system thus adapts and self-improves with minimal human intervention, enabling effective and interpretable anomaly detection using only lightweight, locally deployable LLMs.

This paper makes the following contributions:

- We introduce AgenticNorm, the first lightweight anomaly detection framework that derives semantic constraints directly from logs without relying on source code or program analysis, making it broadly applicable to closed-source and third-party systems.
- We propose a log-oriented prompt learning technique that leverages adversarial attack generation to iteratively refine prompts. This approach overcomes the limitations of manually crafted prompts, enhances stability across applications, and improves the transferability of lightweight LLMs.
- We implement AgenticNorm framework that integrates field clustering, attack generation, and prompt refinement into a multi-agent workflow. The source code and datasets are publicly available in our repository [4].
- We conduct extensive experiments on two widely used benchmarks, TrainTicket and NiceFish, covering over 220k log entries and 230 attack cases. Results show that AgenticNorm achieves state-of-the-art performance, attaining F1-scores of 0.92 and 0.85, and outperforming prior approaches such as WebNorm (0.88 and 0.75).

2 Motivating Example

To illustrate the challenges of detecting anomalies from API logs, we examine a case from the TrainTicket dataset [31] involving a compromised ticket refund process. In a normal workflow, refund requires two steps. First, the user retrieves a list of refundable tickets. This list is displayed on the frontend, allowing the user to select which ticket to refund. Second, the user selects one of these tickets and submits a refund request.

APIs serve as the communication interface between frontend and backend components, typically defined by a request path and a payload format. Logs record the actual data exchanged through these APIs, usually including the request path and a request body and a response body.

Figure 1 shows a simplified version of the relevant log entries, while the full logs are available in our anonymous artifact [4]. In our example, when the user retrieves refundable tickets, the frontend calls the `/api/v1/queryOrders` (abbreviated as `queryOrders`) API (① step in Figure 1), which returns a list of ticket orders. When the user selects a ticket to refund, the frontend

Table 1. Number of Input Tokens for WebNorm

Dataset	Mean	GeoMean	Median
TrainTicket	2.4×10^5	4.3×10^4	2.4×10^4
NiceFish	1.5×10^4	6.9×10^3	4.5×10^3

calls the `/api/v1/refund` (abbreviated as `refund`) API (② step in Figure 1). During invoking the `queryOrders` API, the backend returns a list of refundable tickets in the `results` field, each identified by a unique `id`. When invoking the `refund` API, the frontend provides an `orderId` in the request arguments to specify which ticket to refund.

In normal operation, the frontend only shows tickets that can be legitimately refunded, and the user can only select from this list. So the `orderId` provided in the `refund` request must match one of the `id` values returned by the preceding `queryOrders` call. So there is a consistency constraint between these two APIs:

`refund.arguments.orderId` must appear in `queryOrders.results[].id`.

However, because frontend code executes entirely on the client side, a malicious user can tamper with browser data and forge unauthorized requests. For example, the attacker may replace the valid `orderId` with an arbitrary identifier not returned by `queryOrders` (e.g., `418ea03c`). This manipulation allows the attacker to refund a ticket they do not own or to repeat a refund that should not be permitted. Such behavior can cause duplicate refunds and financial losses, creating significant risks to system security and integrity.

2.1 Existing Approaches

Existing log-based anomaly detection methods fall into two categories: (1) **Model-learning-based approaches**, which learn embeddings or features from normal and attack logs to classify anomalies; and (2) **Rule-learning-based approaches**, which derive semantic constraints from logs and flag violations.

Model-learning-based approaches struggle in this scenario because normal and attack logs differ in only a few fields, making them nearly indistinguishable in embedding space. Furthermore, anomalies may be buried within long sequences of interleaved events, diluting the anomaly signal.

The rule-learning-based approach WebNorm [23] addresses this limitation by leveraging LLMs to infer semantic constraints from logs. For instance, it can derive constraints such as requiring `arguments.orderId` to be a valid UUID. By mapping code-level data flows to log fields and validating them as constraints, WebNorm can flag any violations as anomalies. However, due to its prompt design and the lengthy, nested nature of input logs, the authors acknowledge that WebNorm fails to capture the some cross-API constraint illustrated in Figure 1, leaving such attacks undetected [23]. Despite its strengths, WebNorm still suffers from three key limitations:

- **Dependence on program analysis and source code**: WebNorm requires access to and instrumentation of frontend/backend code, which is costly, fragile under rapid iteration, and infeasible for closed-source or third-party systems.
- **Reliance on heavyweight proprietary LLMs**: constraint synthesis depends on remote, proprietary LLMs, leading to cost and privacy risks. However, if directly replaced with a local deployable model, the long and nested logs often exceed the ability of small models, reducing effectiveness. Table 1 shows the number of input tokens for two datasets. The mean input length for TrainTicket is 2.40×10^5 tokens, far exceeding the context window of lightweight models

```
Based on the logs, infer the valid values for each field by
referencing these common types of data validation:
1. Data Type Check: ......
2. Range Check: does the value fall within a logical numerical
range? (e.g., temperature, latitude, price).
3. Format Check: ......
```

(a) Prompt

```
# Datatype check
if not isinstance(instance['orderId'], str):
    raise ValueError("orderId must be a str")
if not isinstance(instance['userId'], str):
    raise ValueError("userId must be a str")
if not isinstance(instance['price'], float):
    raise ValueError("price must be a str")

# Range Check
price_value = float(instance['price'])
if price_value <= 0:
    raise ValueError("price must be a positive number")
......
```

(b) Corresponding Generated Constraints

Fig. 2. An illustration of relation between prompt and generated constraints in WebNorm. The left panel shows the prompt, which relies on manually crafted instructions and examples. Adjusting such prompts is labor-intensive and they often lack generalizability across diverse scenarios. The right panel illustrates the corresponding constraints generated from the prompt. The highlighted regions reveal the direct mapping between the prompt instructions and the generated constraints. Importantly, the generated constraints strictly adhere to the specified prompt and are limited to the constraint types explicitly mentioned.

due to very lengthy logs. Even for NiceFish, which has shorter logs, the mean input length is 1.50×10^4 tokens, still too long for lightweight models.

- **Prompt sensitivity**: correct constraints often appear only with carefully engineered prompts, making the process labor-intensive, project-specific, and difficult to generalize. Figure 2 illustrates the direct mapping between prompt instructions and generated constraints. The generated constraints strictly adhere to the specified prompt and are limited to the constraint types explicitly mentioned. As a result, WebNorm may miss important constraints not covered by the prompt, leading to undetected anomalies.

2.2 Our Approach

We aim to preserve WebNorm's strength in capturing *consistency constraints* while addressing its limitations. Unlike WebNorm, our approach relies only on logs (without program analysis), operates primarily with lightweight, locally deployable LLMs, and mitigates prompt sensitivity by automatically refining prompts through generated attack logs.

AgenticNorm directly infers constraints from raw logs. AgenticNorm discovers inter-API relationships using frequency-based analysis of co-occurring API logs and derives constraints. To handle long and nested logs, AgenticNorm employs **Field Clustering** to reduce input length while retaining meaningful comparisons. To mitigate prompt sensitivity, AgenticNorm introduces **Prompt Refinement via Generated Attacks**, to iteratively improve prompts using adversarially generated logs that reveal missing constraints.

Field Clustering. To efficiently adapt logs for lightweight models, AgenticNorm expands each log entity into individual fields and then groups comparable ones into clusters. Figure 3 illustrates this process. Each cluster is given as a separate input to the LLM, ensuring that related fields are explicitly compared while avoiding unnecessary context.

Prompt Refinement via Generated Attacks. Figure 4 (left) shows the original WebNorm prompt, which depends on handcrafted instructions and examples and cannot cover all possible constraints. Figure 4 (middle) shows the refined prompt in AgenticNorm after several iterations of attack generation and prompt refinement. The refined prompt includes more explicit instructions, e.g., specifying that fields with similar names should be compared. Figure 4 (right) shows the generated

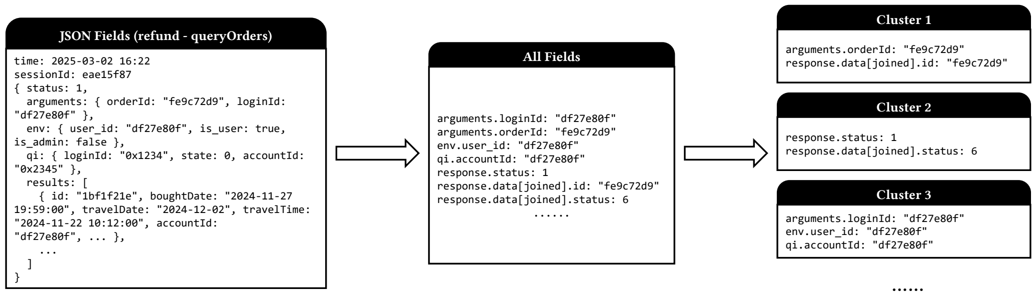


Fig. 3. An illustration of *Field Clustering* on the motivating example from the TrainTicket dataset. The original log entity is decomposed into flattened fields and then clustered into clusters, each containing only related and comparable fields.

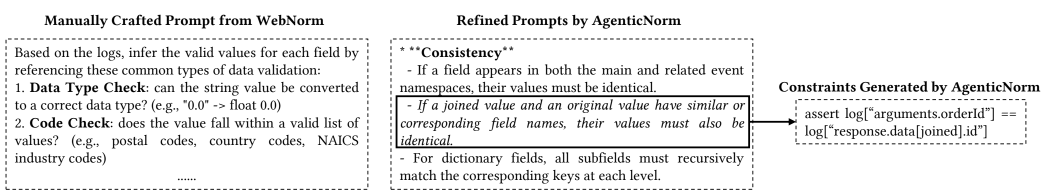


Fig. 4. An illustration of different prompts. The left panel shows the original prompt used in WebNorm, which relies on manually crafted instructions and examples, making prompt adjustment highly labor-intensive. The middle panel shows the refined prompt by AgenticNorm. The right panel presents new learned constraints guided by the new instructions.

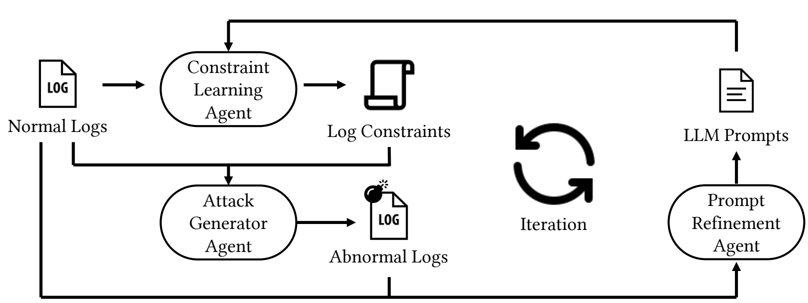


Fig. 5. Method Overview

constraints that correctly capture the necessary field-level dependencies, by one-of the instructions in refined prompt, which WebNorm's prompt failed to identify.

3 Method

In this section, we present AgenticNorm, a lightweight multi-agent anomaly detection framework for web applications. AgenticNorm is designed to overcome the limitations of prior solutions such as WebNorm, namely the reliance on heavyweight proprietary models, sensitivity to prompt

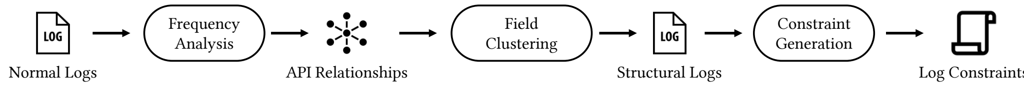


Fig. 6. Constraint Learning

engineering, and difficulty in handling long log contexts. Figure 5 provides an overview of the workflow.

3.1 Workflow

A central challenge in log-based anomaly detection is that prompt quality strongly influences detection results. Fixed prompts are brittle and may fail to capture certain constraints, leading to missed anomalies. To address this limitation, we propose an iterative loop in which attack generation and prompt adjustment are tightly coupled. The loop continuously strengthens prompts by exposing them to adversarial scenarios that exploit their current weaknesses. This process consists of three main modules, forming an iterative loop (Figure 5):

AgenticNorm consists of three main modules:

- **Constraint Learning Agent**: derives constraints from normal logs.
- **Attack Generation Agent**: synthesizes attack logs that break or bypass the learned constraints.
- **Prompt Refinement Agent**: updates LLM prompts using feedback from undetected attacks.

AgenticNorm begins by deriving constraints from normal logs using an initial prompt in the **Constraint Learning** module. Then, the **Attack Generation** module synthesizes attack logs that break or bypass the learned constraints. Finally, the **Prompt Refinement** module updates LLM prompts using feedback from undetected attacks.

This adversarial loop allows prompts to evolve dynamically. Each cycle expands the attack space by introducing logs that specifically target the weaknesses of the current constraints, and in turn strengthens the prompts by incorporating counterexamples. Over time, this reduces reliance on manual intervention and improves robustness against both known and novel attacks.

Next, we break down each module in detail.

3.2 Constraint Learning

AgenticNorm generally follows the idea of WebNorm, but differs in that it does not rely on source code or data-flow analysis. This requires us to replace several of its original components. Figure 6 illustrates the process of constraint learning. First, AgenticNorm discovers relationships between APIs through frequency-based analysis. Next, to adapt to lightweight LLMs, AgenticNorm applies *Field Clustering*, which reduces the length of the input context per query, thereby lowering the workload of the model while improving its ability to identify constraints. Finally, AgenticNorm adopts a similar approach to WebNorm for detecting both intra-API and inter-API constraints, using an LLM to extract constraints and generate corresponding Python checking code. Unlike WebNorm, however, the prompts employed here are not manually designed; instead, they are obtained from the iterative refinement process described later, making them better suited for lightweight LLMs.

3.2.1 Frequency Analysis. AgenticNorm employs a frequency-based method to identify related APIs, eliminating the need for program analysis. Specifically, for a given API, it scans the surrounding window of log entries and counts the frequency of co-occurring API calls. The top-K most frequent

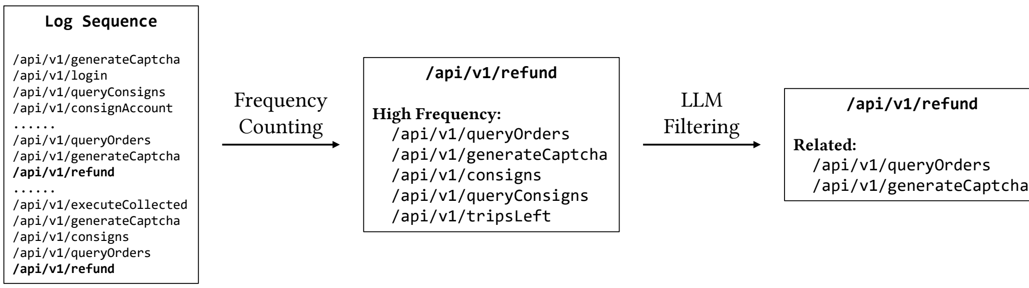


Fig. 7. An example of frequency-based analysis.

$$
\begin{aligned}
\text{Data} := \;& \texttt{unknown} \\
\mid\;& \texttt{bool} \\
\mid\;& \texttt{number} \\
\mid\;& \texttt{string} \\
\mid\;& \texttt{dict}[key_1 : \text{Data}, key_2 : \text{Data}, \ldots] \\
\mid\;& \texttt{array}[\text{Data}]
\end{aligned}
$$

Fig. 8. Grammar for Log Data Types

co-occurrences are considered related APIs, thus establishing inter-API relations. After this step, we utilize an LLM to verify and filter out spurious relations.

Figure 7 shows an example of frequency-based analysis on the TrainTicket dataset. Given a list of API calls, we slide a window of size K and count the frequency of co-occurring APIs. For instance, in this case, `CreateOrder` and `AddPassenger` frequently appear together, indicating a potential relationship. Then, the LLM is used to verify and filter out spurious relations.

3.2.2 Field Clustering. Lightweight LLMs are constrained by limited context windows, making it infeasible to directly process lengthy and complex logs. To address this limitation, we introduce *field clustering*, a technique that decomposes log entries into semantically related groups. This allows constraints to be extracted while ensuring that the input remains within the restricted context length.

To this end, AgenticNorm first analyzes the structure of logs, which often contain nested dictionaries and arrays. It then applies a set of expansion rules to flatten these structures into atomic fields. Finally, it employs an LLM to cluster the expanded fields based on semantic relatedness, forming lightweight groups that can be processed within the context limits.

Figure 3 provides an example of the field clustering process. The original log contains nested dicts (e.g., `arguments`, `qi`, etc.) and arrays (e.g., `results`). These structures are first expanded into flat fields (e.g., `arguments.loginId`, `arguments.orderId`, etc.). Finally, the expanded fields are clustered into semantically related groups (e.g., the cluster of `arguments.loginId`, `env.userId`, `qi.accountId` represents user identifiers).

Formally, we show the field clustering process in three steps: Log Structure Discovery, Expansion, and Clustering. Log Structure Discovery identifies the schema of logs and their data types. Expansion applies a set of rules to flatten nested structures into atomic fields. Clustering groups the expanded fields into semantically related clusters using an LLM.

Log Structure Discovery. Each API may produce logs with diverse structures, including nested dictionaries and arrays. We first parse the logs to uncover their structural schema and data types. For each API, AgenticNorm analyzes all log entries and infers a unified schema that captures the common structure, represented as fields and their associated types. Formally, we define a recursive grammar for data types shown in Figure 8.

Here, `unknown` denotes cases where the log structure cannot be precisely determined (e.g., due to ambiguity or inconsistency). By aggregating logs across APIs, AgenticNorm derives unified schemas that reconcile structural variations.

Expansion. After schema discovery, we apply a set of expansion rules to transform nested structures into flat fields. This ensures that all relevant information is explicitly represented, thereby facilitating clustering and constraint generation.

The rules are as follows:

- **Dict Expansion**: For a dictionary value, each key is concatenated with its parent field using a dot "." separator. Formally, `d: dict[key: value]` is expanded into `"d.key": value`.
- **Array of Dict Expansion**: For an array of dictionaries, each dictionary key is expanded to a new array field. Formally, `a: array[dict[key: value]]` is expanded into `"a[].key": array[value]`.
- **Field Joining**: In certain cases, meaningful semantics emerge when fields from different structural levels are *joined*. Specifically, if an outer field and an inner field share a common key (e.g., an identifier), we match the entry and promote it as a new joined field.

Here are some examples of the expansion rules in motivating example:

- **Dict Expansion**: `arguments: dict[loginId: string, orderId: string]` is expanded into `"arguments.loginId": string, "arguments.orderId": string`.
- **Array of Dict Expansion**: `results: array[dict[id: number, status: string]]` is expanded into `"results[].id": array[number], "results[].status": array[string]`.
- **Field Joining**: `arguments.orderId` can be joined with the elements of `results` via the `id` field. The result is a newly joined field: `"results['joined']": dict[id: number, status: string]`.

Clustering. The expansion step yields a large set of atomic fields, which are then organized into semantically coherent groups. To manage context length effectively, we cluster fields based on semantic relatedness. Instead of relying on hand-crafted heuristics, we employ an LLM to partition the expanded fields into clusters. For example, identifiers such as `user_id`, `session_id`, and joined fields with matching IDs form one cluster, while numerical values such as `price`, `amount`, and `discount` form another. This LLM-based clustering leverages semantic knowledge to generate meaningful and task-relevant partitions.

Through this pipeline, lengthy and complex logs are transformed into compact, semantically organized structures, enabling lightweight LLMs to effectively generate constraints without exceeding context limitations.

3.2.3 Constraint Generation. The constraint generation process of AgenticNorm closely resembles that of WebNorm, with the key distinction that its prompts are not manually crafted but automatically derived through the subsequent attack-generation and prompt-refinement loop, making them more suitable for lightweight LLMs. Given the structured logs, AgenticNorm first instructs the LLM to produce candidate constraints in the form of executable rules that capture constraints across different fields. These candidates are then iteratively evaluated against training logs, and any violations on normal cases are fed back to the LLM along with contextual information, prompting it to revise or discard the problematic constraints. Through this feedback loop, the system gradually

Table 2. OWASP API Security Top 10 categories used in our framework. We exclude API4 (Unrestricted Resource Consumption) and API9 (Improper Inventory Management) because both rely on frequency or large-scale traffic/endpoint modeling, which is beyond the scope of our log-based analysis

Broken Object Level Authorization	Broken Authentication
Broken Object Property Level Authorization	Broken Function Level Authorization
Unrestricted Access to Sensitive Business Flows	Server-Side Request Forgery
Security Misconfiguration	Unsafe Consumption of APIs

** *Identify* ** You are a security testing expert. To ensure system security, your task is to generate attack logs for a given API or API pair based on the OWASP API Security Top 10 categories. *[Attack Strategies from OWASP] [Input/Output Format] [Example]*

** *Input* ** *[Normal Log Entries] [Constraints Conditions]*

** *Output* ** *[Attack Log Entries]*

Fig. 9. Abbreviated version of the LLM prompt for attack generation. Attack strategies and detailed examples are omitted here for brevity; the complete prompt is available in our artifact repository [4].

converges to a compact and reliable set of constraints that preserve both structural correctness and semantic consistency.

3.3 Attack Generation

Based on the extracted constraints and a pool of normal logs, we deliberately synthesize attack log entries that are difficult for the current constraints to capture. The attack generation process is anchored in the *OWASP API Security Top 10*, one of the most authoritative industry standards for categorizing API vulnerabilities. To align with our log-based setting, we exclude categories that depend primarily on traffic volume or usage frequency (e.g., excessive resource consumption).

The *OWASP API Security Top 10*, maintained by the Open Worldwide Application Security Project (OWASP), serves as the de facto reference for identifying and evaluating API vulnerabilities. It is widely adopted by practitioners, penetration testers, and auditors as a standard checklist for assessing the security of modern web APIs. Its categories are derived from extensive industry data and community feedback, collectively covering the vast majority of real-world API attacks observed in practice.

In our framework, we adopt the OWASP API Security Top 10 as the foundation for guiding attack synthesis. Because our anomaly detection operates at the log level rather than the traffic level, frequency-dependent categories (e.g., rate limiting and resource exhaustion) are excluded. For the remaining categories, we refine them into finer-grained subcategories using LLM-based analysis, ensuring that each synthesized attack corresponds to the log semantics of the target system. Table 2 summarizes the OWASP API Security Top 10 categories and indicates their usage in our pipeline. Due to space limit, detailed descriptions can be found in our artifact repository [4].

Concretely, for each API and each API pair, we first sample a set of normal log entries. Guided by the OWASP classification and the descriptions of each attack category, we then prompt an LLM to generate corresponding attack log entries. The generated logs are required to bypass the existing constraints whenever possible. These attack entries, together with the sampled normal logs, form a labeled dataset that is subsequently used for prompt refinement. The prompt used for attack

Algorithm 1 Prompt Refinement via Log-Guided Feedback

Require: Dataset D containing pairs of normal logs N and attack logs A
Require: Original prompt P
Ensure: Refined prompt P'

$M_s \leftarrow [\ ]$ {Initialize list of modification suggestions}
for each $(N, A) \in D$ **do**
  $M \leftarrow$ LLM("Generate modification suggestion", N, A, P) {Generate modification suggestions based on a normal-attack log pair}
  M_s.append(M)
end for
$P' \leftarrow$ LLM("Refine prompt", P, M_s) {Refine the original prompt by incorporating aggregated suggestions}
return P'

generation is shown in Figure 9, with detailed attack strategies and input/output examples provided in our artifact repository [4].

By grounding attack generation in this taxonomy, our framework inherits both breadth and credibility: it covers a wide spectrum of realistic API threats while remaining fully compatible with our log-based constraints detection setting.

3.4 Prompt Refinement

After attack generation, we obtain a labeled dataset consisting of both normal and attack logs. Our next task is to refine the prompts used in constraint generation, so that they can better capture the constraints needed to detect the synthesized attacks. The refinement process is similar to learning a model from labeled data, where the input dataset is the logs and the labels are whether each log is normal or attack. The difference is that instead of adjusting model parameters by policy gradient or backpropagation, we update the prompt text itself using an LLM.

Algorithm 1 outlines the prompt refinement process. For each normal-attack log pair in the dataset, we feed it into an LLM along with the current prompt, asking it to generate a modification suggestion. The LLM analyzes the pair and identifies what changes to the prompt could help distinguish between the normal and attack cases. This may involve adding new clauses, modifying existing ones, or removing irrelevant parts. Figure 10 shows an abbreviated version of the prompt used for refinement, with the complete version available in our artifact repository [4].

3.5 Implementation Details

LLMs Used. AgenticNorm is designed to work with lightweight, locally deployable LLMs. In our study, we observed that the tasks of *Attack Generation* and *Prompt Refinement* place heavier demands on the neural models, as they require more complex reasoning and creative generation. Therefore, we employ larger-scale models for these two tasks, specifically the open-source DeepSeek-V3. For *Constraint Learning*, the requirements are relatively lower, and we adopt smaller models to balance efficiency and effectiveness. In this work, we experimented with multiple models for constraint learning, including `GPT-OSS-120B`, `GPT-OSS-20B`, `Gemma-3-4B`, and `DeepSeek-V3`. This hybrid strategy allows us to maintain strong performance while reducing overall system resource consumption and deployment complexity.

Hyperparameters. For frequency-based API relation extraction, we set the sliding window size to 20 and select the top-5 most frequent APIs as related APIs. In field clustering, the maximum

** *Identify* ** You are an expert in prompt engineering and constraints design for API logs. Your role is to iteratively refine prompts so they generate stronger constraints and corresponding Python detection functions. Your should output modification suggestions for the current prompt.
[Input/Output Format] [Example]
** *Input* ** *[Normal Log Entries] [Attack Log Entries] [Current Prompt]*
** *Output* ** *[Modification Suggestions]*

(a) Modification Suggestions

** *Identify* ** You are an expert in prompt engineering and constraints design for API logs. Your role is to iteratively refine prompts so they generate stronger constraints and corresponding Python detection functions. You should apply the suggested modifications to the current prompt.
[Input/Output Format] [Example]
** *Input* ** *[Current Prompt] [Modification Suggestions]*
** *Output* ** *[Refined Prompt]*

(b) Refined Prompt

Fig. 10. Abbreviated version of the LLM prompt for prompt refinement. The complete prompt is available in our artifact repository [4].

expansion depth for nested dictionaries is limited to 3, in order to avoid field explosion from excessive expansion. For each API, we generate up to 10 normal logs and 10 attack logs for use in prompt refinement. Prompt refinement is iterated for 10 rounds to ensure that the prompts sufficiently adapt to the synthesized attack scenarios. Further experimental details can be found in our code repository [4].

4 Experiments

We focus on the following research questions.

- **RQ1: Overall Performance.** How effective is AgenticNorm in detecting web tamper attacks compared to state-of-the-art baselines and WebNorm? We evaluate its precision, recall, and F1-score on standard benchmarks.
- **RQ2: Ablation Study.** How do the core components of AgenticNorm contribute to its performance? We conduct ablation experiments on field clustering, attack generation, and prompt refinement to measure their individual impact.
- **RQ3: Model Scalability.** How does AgenticNorm perform when deployed with different scales of lightweight, locally deployable LLMs? We assess the trade-offs between detection accuracy, efficiency, and resource consumption across small, medium, and larger models.
- **RQ4: Direct Substitution.** What happens if WebNorm is directly replaced with a smaller LLM without architectural modifications? This comparison highlights the necessity of our proposed techniques over naïve model substitution.

4.1 Experimental Setup

Benchmarks. We evaluate our approach on two widely-used benchmarks of web application logs: TrainTicket and NiceFish. Both datasets contain normal and attack traces derived from real-world systems, with injected tampering behaviors that allow controlled evaluation.

Table 3. Overall evaluation of AgenticNorm

Model	**TrainTicket**			**NiceFish**		
	Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**
LogRobust [51]	0.12	0.65	0.20	0.21	0.54	0.30
LogFormer [15]	0.27	0.76	0.40	0.30	0.70	0.42
RAPID [33]	0.11	0.90	0.20	0.04	1.00	0.08
FastLogAD [24]	0.04	0.20	0.07	0.01	0.05	0.01
WebNorm [23]	1.00	0.80	0.88	1.00	0.75	0.86
AgenticNorm (Ours)	1.00	0.86	**0.92**	1.00	0.92	**0.95**

Baselines. To demonstrate the effectiveness of AgenticNorm, we compare against three categories of methods: (1) *model-learning-based baselines*, including LogRobust [51], LogFormer [15], and RAPID FastLogAD [24], which rely on supervised or semi-supervised learning of log sequences; (2) *rule-based approaches*, represented by WebNorm [23], the current state-of-the-art interpretable system for normality modeling. These baselines cover both predictive and rule-driven paradigms in log anomaly detection.

Evaluation Metrics. We adopt precision and recall as the primary metrics. For windows of normal logs, if any attack is incorrectly flagged, the window is counted as a false positive (FP); otherwise it is a true negative (TN). For attack-containing windows, the detection of any injected attack is considered a true positive (TP), otherwise it is a false negative (FN). Formally, precision, recall, and F1-score are computed as

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}, \quad \text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

4.2 Results and Analysis

RQ1: Overall Performance. Table 3 summarizes the overall comparison[1]. AgenticNorm achieves the highest F1-scores on both TrainTicket (0.92) and NiceFish (0.95). Some baselines, such as LogFormer and RAPID, obtain relatively high recall (e.g., 0.90 on TrainTicket and 1.00 on NiceFish for RAPID), but this comes at the cost of extremely low precision (0.11 / 0.04), leading to many false alarms. LogFormer offers a more balanced trade-off, but its F1-scores (0.40 / 0.42) remain far lower than ours. By contrast, WebNorm achieves perfect precision (1.00) but suffers from lower recall (0.80 on TrainTicket and 0.75 on NiceFish), missing many real attacks due to its reliance on fixed rules. AgenticNorm preserves the perfect precision of WebNorm while substantially improving recall (0.86 / 0.92), thus delivering the strongest overall detection performance.

> **RQ1:** AgenticNorm surpasses state-of-the-art baselines, achieving the best F1-scores across both benchmarks.

RQ2: Ablation Study. Table 4 reports the impact of removing each component. All three modules contribute to performance improvements, but their effects differ in magnitude. Field clustering proves most critical: removing it reduces the F1-score from 0.92 to 0.60 on TrainTicket and from 0.95 to 0.83 on NiceFish. Prompt refinement has a comparable impact, with F1 dropping to 0.61 and

[1] We note that our reproduced results of WebNorm differ slightly from those reported in the original paper. After contacting the authors, we confirmed that they updated their dataset, which led to improved precision but reduced recall. The results shown here reflect this corrected version.

Table 4. Ablation Study

Model	TrainTicket	NiceFish
Original (AgenticNorm)	**0.92**	**0.95**
w/o API relation prediction (in Constraint Learning)	0.67	0.50
w/o field clustering	0.60	0.83
w/o prompt refinement	0.61	0.83

Table 5. Comparison between Number of Tokens with and without Clustering

	With Clustering	Without Clustering
TrainTicket	7.2×10^3	2.4×10^5
NiceFish	6.0×10^3	1.5×10^4

Table 6. Comparison of Different LLMs

	TrainTicket		NiceFish	
	Precision	Recall	Precision	Recall
DeepSeek-V3	1.00	0.86	1.00	0.95
Gemma-3-4B	1.00	0.86	1.00	0.95
GPT-OSS-20B	1.00	0.83	1.00	0.90
GPT-OSS-120B	1.00	0.83	1.00	0.95

0.83, respectively. By contrast, removing API relation prediction (in Constraint Learning) leads to smaller but still notable degradation (0.92 ⟶ 0.67 on TrainTicket and 0.95 ⟶ 0.50 on NiceFish), showing that it provides complementary benefits.

To further evaluate the effectiveness of field clustering, we analyze the total number of input tokens in the prompts, comparing settings with and without clustering. Table 5 reports the token counts for each constraint generation task. The reduction is particularly pronounced on TrainTicket, as its logs contain more fields, and clustering eliminates a larger portion of redundancy. By shortening the token length, the model can process inputs more efficiently, which in turn leads to higher-quality constraints.

> **RQ2:** Each component of AgenticNorm improves performance, with field clustering and prompt refinement being especially crucial.

RQ3: Comparison between Different LLMs. Table 6 shows results when varying the LLM used for the *Constraint Learning* module, while keeping *Attack Generation* and *Prompt Refinement* fixed to DeepSeek-V3. Across all four models (`DeepSeek-V3`, `Gemma-3-4B`, `GPT-OSS-20B`, `GPT-OSS-120B`), precision remains consistently perfect (1.00), and recall varies only slightly (0.83–0.86 on TrainTicket and 0.90–0.95 on NiceFish). This indicates that the effectiveness of AgenticNorm is not tied to a specific model scale in the constraint learning stage. The constraints derived through clustering and refinement are robust across models, demonstrating that smaller and more efficient LLMs can be deployed in practice without sacrificing detection accuracy.

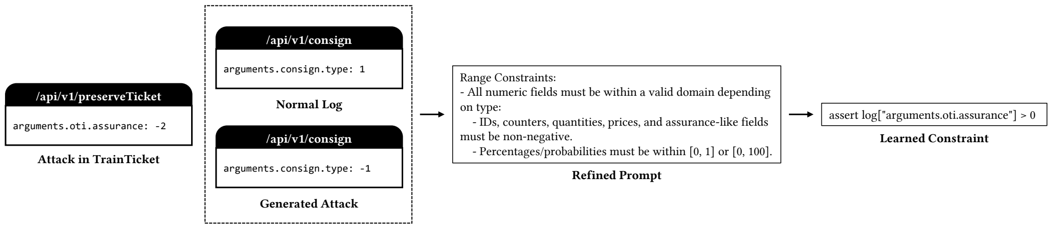


Fig. 11. An example from TrainTicket demonstrating how prompt refinement enhances the quality of generated constraints. The left panel illustrates a simplified attack case. The two middle panels show the generated attacks and the refined prompt, which incorporates additional range constraints compared to the original version. The right panel presents the resulting constraints.

> **RQ3:** AgenticNorm maintains high performance across different LLMs, confirming its adaptability to smaller, locally deployable models.

RQ4: Direct Substitution. To further validate our design, we directly substitute WebNorm's backbone with an open-source LLM (e.g., `DeepSeek-V3`), without applying any of our proposed architectural modifications. Performance drops sharply in recall: on TrainTicket, recall falls from 0.92 to 0.60, and on NiceFish, from 0.95 to 0.50. This experiment shows that simply replacing heavyweight proprietary models with smaller ones is insufficient. While WebNorm functions well with powerful heavyweight LLMs, its constraints are too brittle when scaled down. By contrast, our techniques—field clustering, attack generation, and prompt refinement—enable small models to remain competitive, supporting practical local deployment.

> **RQ4:** Simply substituting smaller LLMs into WebNorm leads to severe performance degradation, highlighting the necessity of our architectural innovations for making lightweight deployment viable.

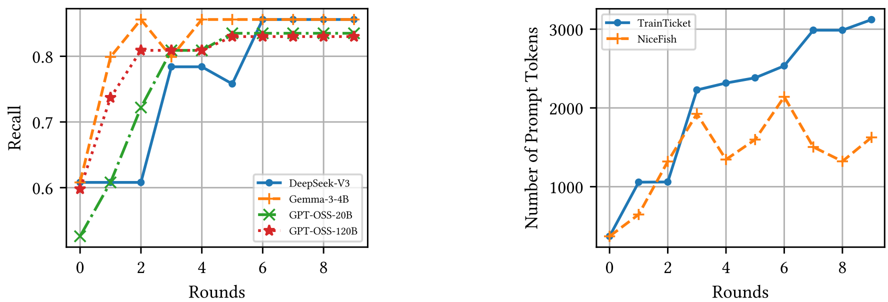
5 Discussion

5.1 Impact of Adversarial Attacks on Prompt Refinement

Adversarial attacks play a crucial role in refining the prompts used for constraint generation. At the initial stage (Round 0), the prompt may fail to capture critical constraints, leading to missed detections for certain types of tamper attacks. However, when we introduce adversarial attacks that exploit these weaknesses, the system is forced to adapt: the failure cases serve as concrete counterexamples that guide the prompt-refinement process. After just one refinement iteration (Round 1), the updated prompt can successfully detect the previously missed anomaly.

For example, as shown in Figure 11, in Round 0, AgenticNorm fails to detect an attack where `arguments.oti.assurance` is assigned a negative value (`-2`). In the first iteration of attack generation, AgenticNorm produces an attack in the *Unsafe Consumption of APIs* category, which involves assigning out-of-range values to fields. The refined prompt introduces explicit range constraints on numeric fields, resulting in the learned constraint. This new constraint allows the system to correctly identify the anomaly that was previously missed.

This self-improving loop demonstrates how adversarial attacks not only test the robustness of the system but also actively drive the enhancement of detection accuracy, ultimately reducing the need for manual intervention and improving efficiency in practice.



(a) Detection Performance (b) Number of Tokens in Prompts

Fig. 12. Different Rounds of Prompt Refinement

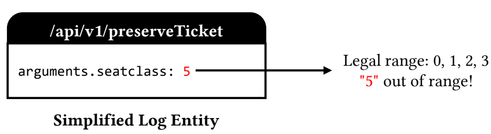


Fig. 13. An example from TrainTicket that cannot be detected by AgenticNorm. In this example, the range of a given field should be 0, 1, 2, 3 according to the system documentation. However, AgenticNorm does not know this information and thus cannot detect the tampering when the field is set to 5.

5.2 Effect of Iterative Prompt Refinement

Figure 12a presents the effect of iterative prompt refinement on detection performance across different models. We observe that recall improves steadily during the early rounds, as each iteration introduces additional constraints that the models can leverage for anomaly detection. Most models reach their highest performance by round 5-6, after which additional refinements yield diminishing returns and performance stabilizes. This pattern suggests that iterative refinement is highly effective in the initial phase, where previously missing constraints are incrementally uncovered, but exhibits saturation once the critical set of constraints has been captured.

This trend aligns with Figure 12b, which shows that the number of tokens in the refined prompts continues to increase with each round. While longer prompts provide more detailed constraints, they eventually add little marginal benefit, indicating that the models have already captured the critical constraints needed for detection. Beyond this stage, further refinement mainly increases prompt complexity without improving effectiveness.

5.3 False Negatives

Although AgenticNorm achieves consistently high precision and recall, some false negatives inevitably remain. Figure 13 illustrates one such case from TrainTicket where AgenticNorm fails to detect the anomaly. In this example, the field `arguments.oti.assurance` is documented to only take values in the range {0, 1, 2, 3}. When the field is tampered with and set to an out-of-range value (e.g., 5), the violation should be flagged as anomalous. However, because AgenticNorm does not have direct access to external domain knowledge such as system documentation or API specifications, it cannot infer that the injected value is invalid. This limitation highlights an important trade-off in our design: the reliance on constraints inferred purely from runtime logs ensures broad applicability in scenarios where source code or documentation is unavailable, but it also means that certain domain-specific invariants may not be captured.

5.4 Limitations

Despite the promising results of AgenticNorm, several limitations remain.

Benchmark Selection. Our evaluation is restricted to two benchmarks (TrainTicket and NiceFish). While these datasets are widely used in prior work, they may not fully reflect the diversity of real-world applications, especially in large-scale industrial or domain-specific systems. Moreover, our attack synthesis is guided by the OWASP API Security Top 10, which provides strong coverage of common vulnerabilities but may overlook emerging or highly specialized attack patterns. This suggests that future work should consider integrating adaptive or domain-specific attack taxonomies.

Model Dependencies. Although AgenticNorm is designed for lightweight deployment, certain modules (e.g., attack generation and prompt refinement) still rely on larger open-source models. This hybrid strategy ensures effectiveness but may limit applicability in environments with strict computational or resource constraints. In addition, the iterative prompt refinement process increases prompt length and complexity, which may reduce efficiency and introduce difficulties in maintaining refined prompts at scale.

6 Related Work

Log Anomaly Detection. Early attempts at anomaly detection from logs were largely based on analyzing execution traces of systems or individual APIs, with the goal of spotting deviations from expected behaviors [48]. Classical approaches mostly relied on manually written rules or statistical thresholds [18, 34, 36, 38, 39, 47, 49]. While useful in certain domains, these methods require expert-crafted specifications and often fail to generalize across applications.

With the development of machine learning, researchers began to explore learning-based approaches that automatically infer normal patterns from logs. Such methods can be broadly categorized into two groups: (1) *model-learning-based approaches*, which train predictive models from normal logs and use them to identify anomalies, and (2) *rule-learning-based approaches*, which mine logical constraints from normal logs and detect violations against them.

Model-learning-based approaches emerged with the advent of data-driven methods, where models are trained to capture behavioral patterns directly from log data [1–3, 6, 7, 20, 21, 25, 27, 32, 35, 40–42, 45, 46]. A large body of work in this direction adopts deep learning models to classify log sequences as normal or anomalous. Recurrent architectures have been widely used to capture sequential dependencies [8, 10], while CNN-based solutions exploit local contextual signals [14, 26]. Recent advances leverage Transformers [15, 19], graph neural networks [50], or pretrained language models tailored for log data [16, 17]. In addition, training-free retrieval methods have been proposed to exploit pre-trained models without fine-tuning, thereby reducing training costs and emphasizing token-level semantics [33]. Other works focus on improving data efficiency, for example by employing pseudo anomaly generation to augment scarce training data [24] or by integrating active learning into retrieval-augmented generation frameworks for anomaly detection [12]. Despite their predictive strength, these neural methods often act as black boxes, providing little insight into the root cause of anomalies and occasionally overlooking subtle yet critical deviations.

To address the limitations of black-box models, a complementary research direction emphasizes *explainability* through rule-learning-based methods. This line of work constructs explicit normality constraints that enable both anomaly detection and interpretable explanations. The most prominent example is WebNorm [23], which encodes behavioral constraints of web systems as first-order logic rules derived from logs. While effective, WebNorm relies heavily on source code analysis and large proprietary models, and it does not explicitly enforce the consistency between logs and their underlying data sources. Our work builds on this direction, proposing techniques that refine normality inference with lightweight, deployable models and enhanced relational reasoning.

RESTful API Security. RESTful APIs, by virtue of their statelessness, simplicity, and ubiquity, have become a critical component of modern web systems as well as an essential surface for attacks. Prior research has explored automated vulnerability discovery through API testing and fuzzing [5, 9, 11, 29, 30, 43], where request sequences or payloads are systematically mutated to reveal abnormal behaviors. These methods are often guided by API specifications, dependency constraints [30, 43], or machine learning predictions [28], reducing the need for manual rule design. More recent approaches further refine fuzzing strategies to target specific classes of vulnerabilities, such as injection attacks and cross-site scripting [9, 11], thereby improving the precision and effectiveness of automated security testing.

While fuzzing uncovers flaws through active probing, our approach takes a complementary perspective: we aim to strengthen API security by learning constraints that characterize normal interaction patterns. By detecting deviations from these learned normalities, we provide a systematic way to capture tampering behaviors that bypass conventional fuzzing-based detection.

Comparison to Our Work. Existing solutions for log anomaly detection and RESTful API security either emphasize predictive accuracy through deep learning or rely on fuzzing techniques to expose vulnerabilities. While effective to some extent, these approaches face notable limitations: deep neural methods lack interpretability and often miss subtle constraints, whereas fuzzing uncovers vulnerabilities opportunistically but does not generalize to unseen tampering strategies. WebNorm represents an important step toward explainable anomaly detection, but its reliance on heavyweight, proprietary models and program source code restricts its applicability in practice.

In contrast, our approach focuses on lightweight, locally deployable models integrated into a multi-agent framework. By introducing field clustering, we address the long-context challenge inherent in lightweight models, and by leveraging an iterative attack-driven prompt refinement loop, we enable the system to self-improve without extensive manual intervention. This combination not only preserves explainability but also ensures that detection can be deployed securely and efficiently in real-world settings.

7 Conclusion

In this paper, we presented AgenticNorm, a lightweight and locally deployable framework for detecting web tamper attacks from logs. By combining field clustering, adversarial attack generation, and iterative prompt refinement in a multi-agent workflow, AgenticNorm addresses key limitations of prior approaches, including dependence on source code, reliance on heavyweight proprietary models, and sensitivity to prompt design. Our evaluation on TrainTicket and NiceFish demonstrates that AgenticNorm achieves state-of-the-art performance while remaining robust across different LLM scales.

Looking ahead, we aim to enable AgenticNorm to adapt continuously as web applications evolve, allowing it to handle changing APIs and attack patterns with minimal retraining. We also plan to explore transfer and meta-learning techniques so that constraints and refined prompts learned from one system can be effectively reused in new applications, improving both efficiency and generalization across domains.

Data Availability

All source code, illustrative examples, detailed prompts, datasets, and experimental results related to AgenticNorm are publicly available at https://sites.google.com/view/agenticnorm/home [4].

References

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/FSE*. https://doi.org/10.1145/1287624.1287630

[2] Md Rakibul Alam, Ilias Gerostathopoulos, Christian Prehofer, Alessandro Attanasi, and Tomas Bures. 2019. A framework for tunable anomaly detection. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 201–210.

[3] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *ESEC/FSE*. https://doi.org/10.1145/3236024.3236069

[4] Anonymous Authors. 2025. *Details of AgenticNorm*. https://sites.google.com/view/agenticnorm/home

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful rest api fuzzing. In *ICSE*. https://doi.org/10.1109/ICSE.2019.00083

[6] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*. https://doi.org/10.1145/2025113.2025151

[7] Jakub Breier and Jana Branišová. 2015. Anomaly detection from log files using data mining techniques. In *Information Science and Applications*. 449–457. https://doi.org/10.1007/978-3-662-46578-3_53

[8] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *MLCS*. https://doi.org/10.1145/3217871.3217872

[9] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. 2023. NAUTILUS: Automated RESTful API Vulnerability Detection. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity23/presentation/deng-gelei

[10] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*. https://doi.org/10.1145/3133956.3134015

[11] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented testing for restful apis. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/du

[12] Chiming Duan, Tong Jia, Yong Yang, Guiyang Liu, Jinbu Liu, Huxing Zhang, Qi Zhou, Ying Li, and Gang Huang. 2025. EagerLog: Active Learning Enhanced Retrieval Augmented Generation for Log-based Anomaly Detection. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1–5.

[13] Erik Feyen, Jon Frost, Leonardo Gambacorta, Harish Natarajan, and Matthew Saal. 2021. Fintech and the digital transformation of financial services: implications for market structure and public policy. *BIS papers* (2021).

[14] Yuanyuan Fu, Kun Liang, and Jian Xu. 2023. Mlog: Mogrifier lstm-based log anomaly detection approach using semantic representation. *IEEE Transactions on Services Computing* 16, 5 (2023), 3537–3549. https://doi.org/10.1109/TSC.2023.3289488

[15] Hongcheng Guo, Jian Yang, Jiaheng Liu, Jiaqi Bai, Boyang Wang, Zhoujun Li, Tieqiao Zheng, Bo Zhang, Junran Peng, and Qi Tian. 2024. Logformer: A pre-train and tuning pipeline for log anomaly detection. In *AAAI*. https://doi.org/10.1609/aaai.v38i1.27764

[16] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. Logbert: Log anomaly detection via bert. In *IJCNN*. https://doi.org/10.1109/IJCNN52387.2021.9534113

[17] Xiao Han, Shuhan Yuan, and Mohamed Trabelsi. 2023. LogGPT: Log anomaly detection via GPT. In *BigData*. https://doi.org/10.1109/BigData59044.2023.10386543

[18] Stephen E. Hansen and E. Todd Atkins. 1993. Automated System Monitoring and Notification With Swatch. In *LISA*. https://dl.acm.org/doi/10.5555/1024753.1024780

[19] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. HitAnomaly: Hierarchical transformers for anomaly detection in system log. *IEEE transactions on network and service management* 17, 4 (2020), 2064–2076. https://doi.org/10.1109/TNSM.2020.3034647

[20] Qiao Kang, Ankit Agrawal, Alok Choudhary, Alex Sim, Kesheng Wu, Rajkumar Kettimuthu, Peter H Beckman, Zhengchun Liu, and Wei-keng Liao. 2019. Spatiotemporal real-time anomaly detection for supercomputing systems. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 4381–4389.

[21] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *FSE*. https://doi.org/10.1145/2635868.2635890

[22] Athina A Lazakidou. 2009. *Web-based applications in healthcare and biomedicine*. Vol. 7. Springer Science & Business Media.

[23] Yifan Liao, Ming Xu, Yun Lin, Xiwen Teoh, Xiaofei Xie, Ruitao Feng, Frank Liaw, Hongyu Zhang, and Jin Song Dong. 2024. Detecting and Explaining Anomalies Caused by Web Tamper Attacks via Building Consistency-based Normality. In *ASE*. https://doi.org/10.1145/3691620.3695024

[24] Yifei Lin, Hanqiu Deng, and Xingyu Li. 2024. Fastlogad: log anomaly detection with mask-guided pseudo anomaly generation and discrimination. *arXiv preprint arXiv:2404.08750* (2024).

[25] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *ICSE*. https://doi.org/10.1145/1368088.1368157

[26] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. 2018. Detecting anomaly in big data system logs using convolutional neural network. In *DASC*. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037

[27] Scott Lupton, Hironori Washizaki, Nobukazu Yoshioka, and Yoshiaki Fukazawa. 2021. Literature review on log anomaly detection approaches utilizing online parsing methodology. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 559–563.

[28] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. 2023. MINER: A Hybrid Data-Driven Approach for RESTAPI Fuzzing. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity23/presentation/lyu

[29] Alberto Martin-Lopez. 2020. Automated analysis of inter-parameter dependencies in web APIs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 140–142.

[30] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In *Service-Oriented Computing: 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings 18*. Springer, 459–475. https://doi.org/10.1007/978-3-030-65310-1_33

[31] Microservice.System.Benchmark. 2018. *TrainTicket*. https://github.com/FudanSELab/train-ticket/

[32] Anthonia Njoku, Heng Li, and Foutse Khomh. 2025. Kernel-Level Event-Based Performance Anomaly Detection in Software Systems under Varying Load Conditions. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 26–30.

[33] Gunho No, Yukyung Lee, Hyeongwon Kang, and Pilsung Kang. 2023. RAPID: Training-free Retrieval-based Log Anomaly Detection with PLM considering Token-level information. *arXiv preprint arXiv:2311.05160* (2023).

[34] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *DSN*. https://doi.org/10.1109/DSN.2015.14

[35] Michael Pradel and Thomas R Gross. 2009. Automatic generation of object usage specifications from large method traces. In *ASE*. https://doi.org/10.1109/ASE.2009.60

[36] James E Prewett. 2003. Analyzing cluster log files using logsurfer. In *Proceedings of the 4th Annual Conference on Linux Clusters*. Citeseer State College, PA, USA, 1–12.

[37] Shati Sarmin Rahman and Sreekanth Dekkati. 2022. Revolutionizing commerce: The dynamics and future of E-commerce web applications. *Asian Journal of Applied Science and Engineering* 11, 1 (2022), 65–73.

[38] John P Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC). In *LISA*. https://dl.acm.org/doi/10.5555/1052676.1052694

[39] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. PerfAugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE*. https://doi.org/10.1109/ICDE.2015.7113365

[40] Vilc Queupe Rufino, Mateus Schulz Nogueira, Alberto Avritzer, Daniel Sadoc Menasche, Barbara Russo, Andrea Janes, Vincenzo Ferme, Andre Van Hoorn, Henning Schulz, and Cabral Lima. 2020. Improving predictability of user-affecting metrics to support anomaly detection in cloud services. *IEEE Access* 8 (2020), 198152–198167.

[41] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D Ernst, and Yuriy Brun. 2010. Synoptic: Summarizing system logs with refinement. In *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML 10)*.

[42] Andrea Stocco and Paolo Tonella. 2020. Towards anomaly detectors that learn continuously. In *2020 IEEE international symposium on software reliability engineering workshops (ISSREW)*. IEEE, 201–208.

[43] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Resttestgen: automated black-box testing of restful apis. In *ICST*. https://doi.org/10.1109/ICST46399.2020.00024

[44] Darko B Vuković, Senanu Dekpo-Adza, and Stefana Matović. 2025. AI integration in financial services: a systematic review of trends and regulatory challenges. *Humanities and Social Sciences Communications* 12, 1 (2025), 1–29.

[45] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *ASE*. https://doi.org/10.1109/ASE.2008.35

[46] Xingfang Wu, Heng Li, and Foutse Khomh. 2023. On the effectiveness of log representation for log-based anomaly detection. *Empirical Software Engineering* 28, 6 (2023), 137.

[47] Kenji Yamanishi and Yuko Maruyama. 2005. Dynamic syslog mining for network failure monitoring. In *KDD*. https://doi.org/10.1145/1081870.1081927

[48] Wenqian Ye, Guangtao Zheng, Xu Cao, Yunsheng Ma, and Aidong Zhang. 2024. Spurious correlations in machine learning: A survey. *arXiv preprint arXiv:2402.12715* (2024). https://arxiv.org/abs/2402.12715

[49] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *ACSAC*. https://doi.org/10.1145/2523649.2523670

[50] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *ICSE*. https://doi.org/10.1145/3510003.3510180

[51] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *ESEC/FSE*. https://doi.org/10.1145/3338906.3338931

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009