LLM-Based Misconfiguration Detection for AWS Serverless Computing

JINFENG WEN, Beijing University of Posts and Telecommunications, China
ZHENPENG CHEN*, Nanyang Technological University, Singapore
ZIXI ZHU, Beijing University of Posts and Telecommunications, China
FEDERICA SARRO, University College London, United Kingdom
YI LIU, Advanced Institute of Big Data, China
HAODI PING*, Beijing University of Technology, China
SHANGGUANG WANG, Beijing University of Posts and Telecommunications, China

Serverless computing is a popular cloud computing paradigm that enables developers to build applications at the function level, known as serverless applications. The Serverless Application Model (AWS SAM) is the most widely adopted configuration schema. However, misconfigurations pose a significant challenge due to the complexity of serverless configurations and the limitations of traditional data-driven techniques. Recent advancements in Large Language Models (LLMs), pre-trained on large-scale public data, offer promising potential for identifying and explaining misconfigurations. In this paper, we present *SlsDetector*, the first framework that harnesses the capabilities of LLMs to perform static misconfiguration detection in serverless applications. *SlsDetector* utilizes effective prompt engineering with zero-shot prompting to identify configuration issues. It designs multi-dimensional constraints aligned with serverless configuration characteristics and leverages the Chain of Thought technique to enhance LLM inferences, alongside generating structured responses. We evaluate *SlsDetector* on a curated dataset of 110 configuration files, which includes correct configurations, real-world misconfigurations, and intentionally injected errors. Our results show that *SlsDetector*, based on ChatGPT-4o (one of the most representative LLMs), achieves a precision of 72.88%, recall of 88.18%, and F1-score of 79.75%, outperforming state-of-the-art data-driven methods by 53.82, 17.40, and 49.72 percentage points, respectively. We further investigate the generalization capability of *SlsDetector* across recent LLMs, including Llama 3.1 (405B) Instruct Turbo, Gemini 1.5 Pro, and DeepSeek V3, with consistently high effectiveness.

Additional Key Words and Phrases: Serverless computing, Software configuration, Large language model

ACM Reference Format:
Jinfeng Wen, Zhenpeng Chen, Zixi Zhu, Federica Sarro, Yi Liu, Haodi Ping, and Shangguang Wang. 2025. LLM-Based Misconfiguration Detection for AWS Serverless Computing. 1, 1 (June 2025), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

*Corresponding authors

Authors' addresses: Jinfeng Wen, Beijing University of Posts and Telecommunications, Beijing, China, jinfeng.wen@bupt.edu.cn; Zhenpeng Chen, Nanyang Technological University, Singapore, Singapore, zhenpeng.chen@ntu.edu.sg; Zixi Zhu, Beijing University of Posts and Telecommunications, Beijing, China, zhuzixi.zzx@gmail.com; Federica Sarro, University College London, London, United Kingdom, f.sarro@ucl.ac.uk; Yi Liu, Advanced Institute of Big Data, Beijing, China, liuyi14@pku.edu.cn; Haodi Ping, Beijing University of Technology, Beijing, China, haodi.ping@bjut.edu.cn; Shangguang Wang, Beijing University of Posts and Telecommunications, Beijing, China, sgwang@bupt.edu.cn.

XXXX-XXXX/2025/6-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Serverless computing is a popular cloud computing paradigm that allows developers to build and run applications, known as *serverless applications*, without managing underlying infrastructure tasks [60]. It has been widely adopted across diverse application domains [19, 53, 69], attracting growing interest from research communities, such as Software Engineering (SE) [60] and Systems [42], and from industry. To support the development and execution of serverless applications, leading cloud providers have introduced serverless platforms. Among these providers, Amazon Web Services (AWS) stands out as the leader in serverless computing [34, 60, 61].

Serverless computing includes two primary service models: Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [26, 32]. FaaS allows developers to build applications as small and event-driven functions (i.e., serverless functions). BaaS provides ready-to-use cloud services such as storage (e.g., AWS S3 [2]), database, and API gateway management. FaaS collaborates with BaaS to enable developers to create serverless applications efficiently. To configure and manage functions and required cloud resources for serverless applications, AWS provides the Serverless Application Model (AWS SAM) [5], the most widely adopted configuration schema in the serverless computing practice [1, 7, 15]. It can streamline the development process and reduce complexities associated with resource management in serverless applications.

However, misconfigurations have emerged as a major challenge in serverless application development [40, 55, 61], leading to serious security vulnerabilities and operational risks. For instance, as reported [17], a coronavirus testing company exposed over 50,000 scanned IDs and thousands of test results due to an AWS S3 bucket misconfiguration [17]. In another case, API misconfigurations within a serverless environment led to a breach affecting 4.9 million customers [12]. These incidents highlight that misconfigurations are systemic rather than isolated, underscoring the urgent need for effective detection mechanisms in serverless computing.

Misconfigurations have become one of the major causes of system software failures [66]. Despite the promise of existing data-driven methods for misconfiguration detection in other scenarios [50, 51, 72, 73], they have low effectiveness in serverless computing. Data-driven approaches, which rely on anomaly detection or pattern recognition based on training data, suffer from limitations such as incomplete or incorrect datasets [72, 73]. Additional strategies that incorporate extensive knowledge, such as predefined templates and official documentation, lack flexibility and adaptability. These problems make the data-driven approach not enough to detect configuration problems of serverless applications. Moreover, serverless application configurations involve intricate structures, including domain-specific languages, complex dependency relationships, and nested objects across over 800 cloud resource types, which further complicates their detection.

Recent advancements in Large Language Models (LLMs) offer a promising solution to this challenge. LLMs have demonstrated significant success in various SE tasks [44], such as code generation [29], code summarization [18], program repair [28], test generation [45], and log parsing [64]. Trained on large-scale public data, LLMs can potentially capture configuration patterns, best practices, and common pitfalls, making them well-suited for detecting misconfigurations in serverless applications.

In this paper, we present *SlsDetector*, the first LLM-based framework specifically designed to perform static misconfiguration detection in serverless applications. It does not rely on a large number of real-world training examples. Leveraging advanced prompt engineering in conjunction with zero-shot prompting, which requires no prior examples, *SlsDetector* efficiently identifies configuration problems with minimal effort. Given a serverless configuration file, it outputs detected misconfigurations along with detailed, structured explanations. *SlsDetector* features a prompt

generation component that dynamically integrates the configuration file, task description, multi-dimensional constraints, and customized response template. Multi-dimensional constraints are designed according to serverless configurations, incorporating resource types, configuration entries and values, as well as different levels of dependencies to provide context-aware guidance. Additionally, *SlsDetector* employs the Chain of Thought reasoning technique [10, 23] to enhance inference quality. The customized response template provides the content demand and format demand of LLM outputs, ensuring that responses are not only structured but also actionable answers aligned with detailed explanations. Particularly, *SlsDetector* targets misconfigurations within serverless application configuration files. It does not analyze application source code, nor does it detect misconfigurations related to external systems or environment-specific issues.

To evaluate *SlsDetector*, we curate an evaluation dataset of 110 configuration files, including 26 correctly configured files, 58 with real-world misconfigurations, and 26 with injected errors. Results show that *SlsDetector*, based on ChatGPT-4o (one of the most representative LLMs known for outstanding performance), achieves a precision of 72.88%, recall of 88.18%, and F1-score of 79.75%. It outperforms the state-of-the-art data-driven approach by 53.82, 17.40, and 49.72 percentage points, respectively. We further explore the generalization capability of *SlsDetector* using other representative LLMs, including Llama 3.1 (405B) Instruct Turbo, Gemini 1.5 Pro, and DeepSeek V3, with results demonstrating consistently high effectiveness across models.

In summary, this paper makes the following contributions:

- We present *SlsDetector*, the first LLM-based approach specifically designed for detecting misconfigurations in serverless computing. Its core is a carefully designed zero-shot prompt, incorporating novel multi-dimensional constraints that capture the configuration characteristics of serverless computing.
- We construct the first benchmark dataset for misconfiguration detection in serverless computing, releasing it alongside our scripts and results as a replication package [16].
- We conduct an empirical study using our benchmark dataset to evaluate the effectiveness of our misconfiguration detection approach, demonstrating that it outperforms baseline methods.

2 BACKGROUND

2.1 Serverless Computing

Applications developed within the serverless computing paradigm are referred to as serverless applications. These applications are built around event-driven serverless functions (i.e., FaaS), which represent the core business logic. Functions collaborate with associated cloud services that facilitate the integration of backend functionalities, i.e., BaaS. This combination streamlines the development process [26, 60]. During the development and deployment of serverless applications, developers define essential execution settings. These settings include the runtime environment, memory allocation, timeout duration, predefined event triggers, and required cloud resources for the serverless applications.

2.2 Serverless Application Configurations: AWS SAM

Developers leverage specified configuration files, such as YAML files, to define the execution settings of serverless applications. In serverless computing, serverless functions are inherently event-driven, meaning that the relationships between functions and predefined events are not explicitly detailed in the application code. Instead, these relationships are succinctly captured in the configuration file, which automates infrastructure provisioning. Thus, the configuration file plays a crucial role in the development process of serverless applications.

Among mainstream serverless platforms, AWS Lambda employs a widely used configuration schema [1, 15] known as the AWS Serverless Application Model (AWS SAM) [5]. AWS SAM enables developers to easily reuse proven configurations, streamlining the development and deployment of serverless applications. In contrast, other platforms, such as Google Cloud Functions [14] and Microsoft Azure Functions [9], lack a formal configuration schema. They rely on command-line interfaces or platform consoles to manually manage key settings and required resources. This manual way lacks standardization and the availability of configuration datasets for analysis. **Given AWS Lambda's widespread use and the advantages offered by AWS SAM's configuration schema, our paper focuses on analyzing the configurations of serverless applications built using AWS SAM.**

AWS SAM uses a YAML-based configuration file format with specialized template specifications. It builds upon and extends AWS CloudFormation [3], which is primarily used for provisioning and configuring non-serverless cloud resources. AWS SAM introduces a syntax specifically designed for defining and managing both serverless infrastructure (spanning nine categories [6]) and non-serverless infrastructure (covering over 800 categories [4]).

Serverless application configurations are complex and exhibit unique characteristics. Unlike the simple "flat" key-value pair format commonly seen in prior configuration studies [21, 50, 55, 58, 65], serverless application configurations feature intricate structures, including objects, lists, maps, and nested elements. Each cloud resource type is represented by custom-named objects, which contain specific configuration entries and their corresponding values. These values can be strings, lists, maps, or even nested objects representing other cloud resources. Additionally, serverless configurations introduce resource types specific to serverless environments (e.g., "AWS::Serverless::Function") and attributes unique to serverless applications (e.g., `Handler`, `MemorySize`, `Timeout`). This exhibits that AWS SAM YAML files function as domain-specific languages within the serverless computing domain, increasing the complexity of configurations.

2.3 Example of Configuration File

We provide a real-world configuration file example [13] from GitHub, a widely used platform for studying developer issues [35, 36], as shown in Fig. 1. In this example, the developer created a serverless function that responds to events from the AWS S3 storage service [2]. However, this configuration failed during deployment. Resolving this issue required nearly 20 rounds of communication involving 26 people and spanned almost five years before a correct solution was found. The root cause was the unsupported `Condition` entry mistakenly added on line 24. This example underscores the critical need for an effective approach to detect misconfigurations in serverless applications early. Such an approach would quickly pinpoint potential issues, reducing the time, effort, and communication overhead required to troubleshoot and resolve misconfigurations.

We explain this configuration file. The content mainly includes `Resources` section (lines 15-39). It defines the required execution settings through **resource types**. The "AWS::Serverless::Function" resource type (named "BucketEventConsumer") aims to configure a serverless function, while "AWS::S3::Bucket" (named "SomeBucket") represents an AWS S3 bucket, a non-serverless resource that frequently interacts with serverless functions. The "BucketEventConsumer" object includes **configuration entries** such as the handler function (line 19), runtime environment (line 20), code location (line 21), and predefined event (lines 22-33). These entries are allocated **values** that conform to the constraints. For example, `Runtime` is set to "python3.6" (line 20). The function is triggered when an S3 object is created (lines 27-28) and the object meets the filter rule specified as key-value pairs (lines 31-33). `Name` from line 32 and `Value` from line 33 need to appear together, indicating **entry dependencies**. Line 27 illustrates a relationship between `Bucket` value and the

```
1  AWSTemplateFormatVersion: '2010-09-09'
2  Transform: AWS::Serverless-2016-10-31
3  Description: Lambda that responds to S3 events
4  Parameters:
5      PreExistingBucket:
6          Description: "Does an existing bucket exist (not managed by cloudformation)"
7          Type: String
8          Default: 'no'
9          AllowedValues:
10             - 'yes'
11             - 'no'
12         ConstraintDescription: must specify yes or no.
13 Conditions:
14     NeedsSomeBucket: !Equals [!Ref PreExistingBucket, 'no']
15 Resources: // define objects with specific resource types (lines 15-39)
16     BucketEventConsumer:
17         Type: AWS::Serverless::Function
18         Properties:
19             Handler: BucketEventConsumer.main.lambda_handler // set configuration entry: the corresponding value
20             Runtime: python3.6 // set configuration entry: the corresponding value
21             CodeUri: bundle.zip
22             Events:
23                 CreateMetaEvent:
24                     # Condition: NeedsSomeBucket
25                     Type: S3
26                     Properties:
27                         Bucket: !Ref SomeBucket // rely on the resource object (line 34), representing value dependency
28                         Events: "s3:ObjectCreated:*"
29                         Filter:
30                             S3Key:
31                                 Rules:
32                                     - Name: suffix
33                                       Value: meta.json // rely on Name entry (line 32), representing entry dependency
34     SomeBucket:
35         Condition: NeedsSomeBucket
36         Type: AWS::S3::Bucket
37         Properties:
38             BucketName: 'some-bucket-somewhere'
39         DeletionPolicy: Retain
```

Fig. 1. An configuration file example of serverless applications.

"AWS::S3::Bucket" resource in line 34, showing that the **value dependencies** of one configuration value depend on other values.

In addition to the core sections, other parts of the configuration file are also important. The `AWSTemplateFormatVersion` section (line 1) specifies the template's capabilities, with the current valid format version being "2010-09-09" [8]. The `Transform` section (line 2) identifies the file as an AWS SAM template with the value "AWS::Serverless-2016-10-31." The `Description` section (line 3) provides a textual description of the template. The `Parameters` section (lines 4-12) defines values that are passed to the template at runtime. The "PreExistingBucket" parameter accepts either "yes" or "no" as values. The `Conditions` section (lines 13-14) controls resource creation or property assignment based on the value of a parameter. The "NeedsSomeBucket" condition checks if the "PreExistingBucket" parameter is set to "no". If true, the condition evaluates to true, otherwise, it evaluates to false.

2.4 Prompt Engineering of LLMs

Recent studies [43, 71] have demonstrated the effectiveness of prompt engineering in enhancing LLM performance across various tasks. Prompt engineering involves designing task-specific instructions, known as prompts, to guide LLM behavior without modifying the underlying model parameters. This approach allows LLMs to adapt seamlessly to different tasks based solely on carefully crafted input prompts. We introduce commonly used prompt engineering techniques:

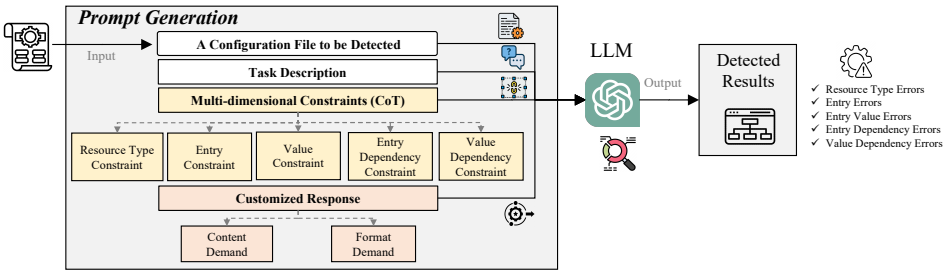

Fig. 2. The overview of our approach *SlsDetector*.

- **Zero-shot prompting**: The LLM performs a task without any prior training or examples, relying entirely on the prompt's instructions to generate the expected output. This approach tests the model's ability to generalize knowledge directly from its pre-trained data.
- **Few-shot prompting**: The LLM is provided with a limited set of examples within the prompt to infer patterns and apply learned knowledge to similar tasks. This technique improves response accuracy by offering contextual cues while still requiring minimal training data.
- **Chain-of-Thought (CoT) prompting**: The LLM enhances its reasoning capabilities by breaking down complex tasks into a sequence of intermediate natural language reasoning steps. This structured approach helps the model generate more logical, coherent, and interpretable responses.

This paper leverages prompt engineering techniques to design specialized prompts for detecting misconfigurations in serverless applications, ensuring effective misconfiguration identification.

3 OUR APPROACH: *SLSDETECTOR*

We present *SlsDetector*, an LLM-based framework designed to detect misconfigurations in serverless applications. *SlsDetector* takes a configuration file of the serverless application to be detected as input and outputs structured results, providing a list of detected misconfigurations along with detailed explanations for each issue. The framework is adaptable and supports various LLMs.

3.1 Overview

Fig. 2 shows an overview of *SlsDetector*. It converts a misconfiguration detection request into a meticulously constructed prompt for LLMs. We employ zero-shot prompting to minimize reliance on external sample configurations. This technique, which requires no prior examples, is a popular optimization technique [33, 62, 71]. While many studies [43, 64, 68] have utilized few-shot prompting to improve effectiveness by learning from examples during inference, they rely heavily on the quality and selection of labeled samples. In contrast, zero-shot learning avoids the cost and effort associated with sample collection and curation, making it the preferred technique for our framework.

In *SlsDetector*, we design a prompt generation component to construct a tailored prompt focused on the objective of detecting misconfiguration in serverless applications. This prompt is structured into four parts, where multi-dimensional constraints are the core of *SlsDetector* and highly context-aware, shown in Fig. 2. In particular, this paper novelly introduces multi-dimensional constraints based on configuration characteristics of our scenario, rather than previous work. Once the prompt is constructed, it is sent to the LLM, which generates the final output. Next, we introduce the prompt generation component in detail.

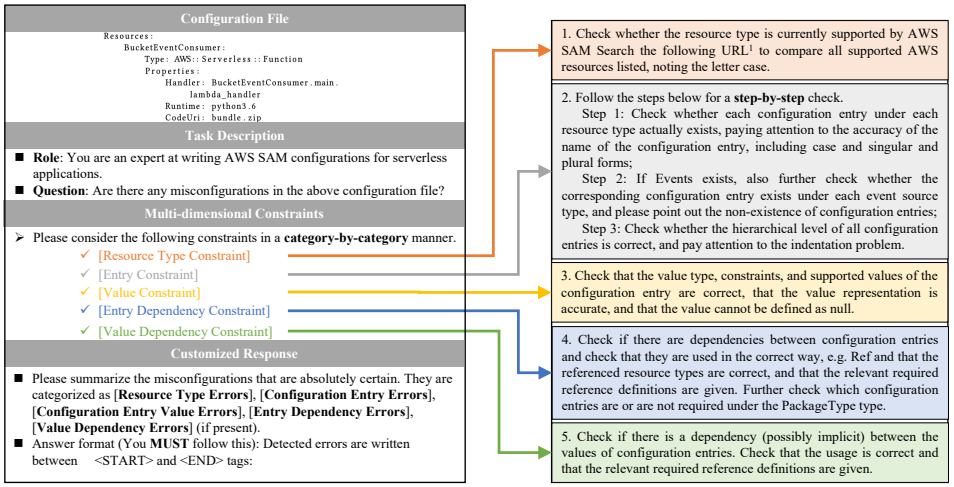

Fig. 3. The prompt structure of *SlsDetector*.

3.2 Prompt Generation

We present the prompt content generated by the prompt generation component, which includes: (i) the configuration file to be analyzed, (ii) a task description for the LLMs, (iii) detailed multi-dimensional constraints, and (iv) a customized response. Fig. 3 illustrates our prompt structure.

3.2.1 Task Description. The task description includes the following elements: (i) a role-playing instruction designed to enhance the LLM's ability to detect misconfigurations, which is a common prompt optimization technique [25, 71]; and (ii) a task description instruction. In our scenario, the role is designed as "You are an expert at writing AWS SAM configurations for serverless applications", while the task description asks, "Are there any misconfigurations in the above configuration file?". These elements are carefully crafted to clearly outline the tasks the LLM needs to complete within the assigned role.

3.2.2 Multi-dimensional Constraints. Multi-dimensional constraints are designed based on the distinct hierarchical structure of serverless application configurations, as described in Section 2.3. We systematically define five key constraint dimensions: **resource types, configuration entries, values of configuration entries, entry dependencies, and value dependencies**. These elements represent critical and unique configuration components due to the event-driven, distributed, and auto-scaling nature of serverless computing. While these components enable flexible and scalable resource management, they also introduce complexity and specific challenges in configuring and managing resources. Thus, emphasizing these components in the detection process enhances the LLM's ability to effectively identify misconfigurations.

• *Resource Types*: Serverless applications often comprise diverse resource types, including serverless functions, event sources such as API Gateway, S3, and DynamoDB, and resources that support scalability, such as step functions or queues. In AWS SAM, defining these resource types is critical because serverless applications are inherently composed of multiple resources that must work together seamlessly. Different from traditional server-based applications, where resources are fixed and managed manually, serverless applications dynamically scale and interact based on event triggers. This makes resource type configuration especially important. Fig. 1 shows some resource types in lines 17 and 36. For instance, custom names such as "BucketEventConsumer" (line 16) are

assigned to objects tied to specific resource types, such as "AWS::Serverless::Function". Moreover, resource type names are case-sensitive.

• *Configuration Entries*: Serverless applications require specific configuration entries to set execution parameters, such as the runtime environment, required memory, and event sources. These entries determine the operational behavior of serverless applications. For example, the `Runtime` entry defines the runtime environment for a serverless function, and the `Events` entry specifies the event triggers (e.g., API Gateway, S3 uploads) for the serverless function. In Fig. 1, the `Runtime` (line 20) and `Events` (line 22) entries are key configuration parameters that define how the serverless function is executed and what triggers it.

• *Values of Configuration Entries*: Configuration entries are assigned specific values that are often constrained by predefined sets. For example, memory size, timeout duration, and other trigger-related values must be accurately set. In Fig. 1, the `Runtime` entry in a serverless function configuration only allows certain runtime environments, such as "python3.6" and "nodejs16.x". The `Bucket` entry (line 27) might only accept references to other AWS objects or resources. These values must be accurate to ensure the serverless application behaves as intended.

• *Entry Dependencies*: Serverless applications often involve multiple interconnected services, such as event triggers or communication between functions. These dependencies must be accurately configured to ensure the correct flow of events and data. For instance, a serverless function might be triggered by an API Gateway, which itself needs proper configuration in terms of timeouts, integration, and permissions. The need for precise management of dependencies between these services is a defining feature of serverless environments, where different functions and services interact based on events. In this situation, some configuration entries are dependent on others to work correctly. For example, `Name` from line 32 and `Value` from line 33 need to be configured together for the configuration to be valid. These dependencies are generally implicit and can be discovered through documentation or inferred by understanding how AWS services interact.

• *Value Dependencies*: In serverless computing, values from one resource can influence the configuration of another. These cross-resource value dependencies are unique to serverless applications, where each resource's configuration may directly affect the efficiency of others. For example, the `RestApiId` entry, which is required for API Gateway event triggers, depends on the object name of the corresponding "AWS::Serverless::Api" resource. This shows how values can be linked across different resource types, reflecting the tight integration between Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) resources in serverless environments. This level of interdependence requires careful validation to avoid conflicts or inefficiencies.

Leveraging these configuration characteristics, we design serverless-specific multi-dimensional constraints to enhance the LLM's ability to identify serverless application misconfigurations. These constraints include: **resource type constraint**, **entry constraint**, **value constraint**, **entry dependency constraint**, and **value dependency constraint**. Importantly, these constraints are internally designed and do not require input from developers or generation by other tools. Fig. 3 shows their details.

Before explaining constraints, we apply the CoT technique [10, 23, 39]. CoT is a reasoning strategy to guide the problem-solving process toward more accurate and logical conclusions. Based on its principle, we design our CoT strategy for detecting misconfigurations of serverless applications by guiding LLMs to consider constraints in a "category-by-category" manner.

For **resource type constraint**, we describe it as follows: "Check whether the resource type is currently supported by AWS SAM. Search the following URL[1] to compare all supported AWS resources listed, noting the letter case." By providing a direct link to the official documentation, we

[1]Supported resource types: https://docs.aws.amazon.com/serverlessrepo/latest/devguide/list-supported-resources.html

enable *SlsDetector* to effectively identify and compare resource type names, with a particular focus on case sensitivity, a critical aspect in AWS SAM configurations.

For **entry constraint**, we adopt a structured three-step reasoning process that follows a coarse-to-fine validation design: (1) Large-scale constraint validation verifies whether each configuration entry exists under the corresponding resource type, ensuring structural integrity. (2) Fine-grained entry validation applies additional checks to essential entries in serverless applications, particularly event-related entries, which are critical for function execution. (3) Format validation ensures correct entry indentation in YAML-based configuration files, as improper formatting can lead to misconfigurations. This structured reasoning process refines misconfiguration detection, enabling a more effective evaluation. This structural design systematically refines constraint validation, moving from broad checks to fine-grained validation. By progressively narrowing the validation scope, its design helps the LLM focus on progressively more specific misconfiguration risks. *SlsDetector* applies these checks using the CoT technique, following a "step-by-step" process, as shown in Fig. 3. *Step 1: SlsDetector* checks that each configuration entry exists under its respective resource type. *Step 2:* For event-related entries, *SlsDetector* checks that configuration entries corresponding to each event source type are present. *Step 3: SlsDetector* checks the correct hierarchical structure of all configuration entries, with special attention to indentation. Misplaced or improperly indented entries may lead to errors, as they will not be recognized under the expected resource type. This three-step validation process allows *SlsDetector* to systematically detect errors, ensuring comprehensive and accurate checks for configuration entries.

For **value constraint**, *SlsDetector* validates that each configuration entry has the correct value type, satisfies the defined constraints and supported values, maintains an accurate value representation, and is not assigned a null value. These constraints ensure that all values adhere to the required specifications.

For **entry dependency constraint**, *SlsDetector* checks whether dependencies exist between configuration entries and verifies that they are correctly used. We also provide guidelines for validating dependencies, such as checking the accuracy of referenced resource types, ensuring required reference definitions are present, and confirming that required entries are properly configured.

For **value dependency constraint**, *SlsDetector* checks whether there are dependencies (including implicit ones) between the values of configuration entries, verifies their correct usage, and ensures that all required reference definitions are provided. This constraint helps to maintain consistency and correctness in how values interact and depend on each other within the configurations.

3.2.3 Customized Response. We customize the LLMs' output by specifying both the content and format requirements for the responses, ensuring their effectiveness and relevance. For the content demand, we aim to avoid receiving vague or uncertain answers that fail to explicitly identify configuration errors. To achieve it, we instruct the model with the directive: "Please summarize the misconfigurations that are absolutely certain". This ensures that only clear, deterministic errors are returned. Additionally, when applicable, we categorize the detected misconfigurations into specific groups, including "Resource Type Errors," "Configuration Entry Errors," "Configuration Entry Value Errors," "Entry Dependency Errors," and "Value Dependency Errors".

For the format demand, to eliminate redundant content that does not reveal specific misconfigurations from the raw output, we use delimiters: "<START>" and "<END>", to mark the required portion of the response. In *SlsDetector*, the desired output is enclosed within these markers, for example: "<START> Resource Type Errors: ..., Value Dependency Errors: ... <END>". This structured way ensures that only the relevant content is captured. During post-processing, *SlsDetector* employs regular expressions to extract the information between these markers efficiently. Although the

model might generate additional text beyond the expected response, the use of locators allows for the seamless extraction of relevant content while discarding unnecessary text.

3.3 Prompt Discussion

The prompt generation process is designed iteratively through systematic trial and error, aiming to maximize misconfiguration detection effectiveness. The process begins with a minimal prompt that encapsulates only a generic misconfiguration detection objective, as shown in Fig. 4. However, early evaluations reveal its limited capability, consistent with the results of the basic LLM method presented in Section 5.2. To enhance detection effectiveness, we hypothesize that enriching the prompt with configuration-specific features would be beneficial. Guided by this insight, we analyze the structural and semantic characteristics of configuration files, focusing on key elements such as resource types, configuration entries, and their values, as well as dependencies among different elements. Based on this analysis, we iteratively introduce a series of constraints that capture these essential dimensions. Each iteration involved empirical evaluation of configuration examples, assessing whether the modified prompt improved the detection of ground-truth misconfigurations. This process included refinements such as progressing from coarse-grained to fine-grained validation, and incrementally supplementing constraints to better model the implicit rules governing configuration correctness. Through this iterative refinement, we arrive at the final optimized prompt shown in Fig. 3, which achieves a practical balance between detection accuracy and description generalizability. Its effectiveness is further validated by large-scale evaluation results presented in RQ2, which is attributable to the incorporation of multi-dimensional configuration constraints.

4 EXPERIMENTAL EVALUATION

To evaluate the effectiveness of *SlsDetector* in identifying misconfigurations within serverless applications, we present four research questions (Section 4.1). To answer these questions, we detail the evaluation metrics (Section 4.2), baselines for comparison (Section 4.3), evaluation dataset (Section 4.4), and experimental settings (Section 4.5).

4.1 Research Questions

- **RQ1:** How does the effectiveness of *SlsDetector* compare to traditional data-driven methods?
- **RQ2:** How effective is *SlsDetector* without considering our multi-dimensional constraints?
- **RQ3:** How does the non-determinism of LLMs influence the effectiveness of *SlsDetector*?
- **RQ4:** How well does the generalization capability of *SlsDetector* perform when using different LLMs?

4.2 Evaluation Metrics

We use *precision*, *recall*, and *F1-score* as evaluation metrics to compare *SlsDetector* against the baseline methods at the configuration parameter level, i.e., configuration entries or values. We check whether the detection approach can accurately determine the validity of each configuration parameter within the configuration file. *precision* measures the proportion of correctly identified misconfigured parameters among all parameters flagged as misconfigured. *recall* quantifies the ability of the approach to detect actual misconfigurations by calculating the proportion of true misconfigured parameters that are correctly identified. *F1-score* provides a balanced measure that accounts for the significance of both false positives and false negatives. These metrics are calculated through True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN), explained in Table 1. $precision = \frac{TP}{TP+FP}$, $recall = \frac{TP}{TP+FN}$, and $F1\text{-}score = 2 \times \frac{precision \times recall}{precision+recall}$. Values range from 0% to 100%, with scores closer to 100% indicating greater effectiveness.

Table 1. The explanation of TP, FP, TN, and FN in our scenario.

TP	A misconfigured parameter correctly identified as misconfigured
FP	A correctly configured parameter mistakenly flagged as misconfigured
TN	A correctly configured parameter accurately recognized as valid
FN	A misconfigured parameter that is overlooked or incorrectly classified as valid

4.3 Baseline Methods

We implement three types of baselines to evaluate effectiveness. Given the lack of approaches specifically tailored for detecting misconfigurations in serverless computing, we first draw on principles from established data-driven techniques used in prior configuration studies [50, 51, 72, 73]. By adapting these methods, we create a data-driven baseline suited to the characteristics of serverless applications. Additionally, we introduce LLM-based baselines as comparisons.

• **Baseline 1:** Data-driven method. We implement a data-driven approach for serverless applications by learning configuration patterns from a dataset of configuration files. As no existing dataset specifically focuses on serverless application configurations, we collect our data from the AWS Serverless Application Repository (SAR) [7], an official repository for serverless applications where each application is packaged with an AWS SAM template and links to relevant configuration files. We include all configuration files associated with serverless applications that have been successfully deployed at least once. This results in a collection of 701 configuration files across 658 serverless applications, with some links providing multiple configuration files representing distinct configurations. Given the correctness of ensuring the dataset, we conduct a careful manual review of the configuration files. This review was performed by the first two authors, who have a background in cloud computing. Identified issues were discussed and resolved with consensus among the authors. To assess the consistency of independent labeling, we employ Cohen's Kappa (κ) [24], a widely used metric for measuring inter-rater agreement. The resulting κ value of 0.916 indicates an almost perfect agreement and a reliable labeling procedure [37].

Using this dataset, we learn configuration patterns, focusing on common resource types, configuration entries, values, and dependencies among entries and values. We first standardize the configuration files into a uniform representation. Object names for various resource types are identified, with object names replaced by standardized labels (e.g., a placeholder "PH+resource type") for consistency across configuration entries and values. We then extract the used resource types, entries, and values. To detect dependencies among both entries and values, we apply association rule mining techniques [67, 70]. Specifically, we use the FP-Growth algorithm [31], which is known for its scalability. We set a support threshold for frequent itemsets using the formula $\alpha \times len$, where len represents the total number of configuration files, a deterministic value, and α is a percentage that indicates the desired mining granularity. Leveraging mined frequent itemsets, we generate association rules by utilizing traversal way and dividing items into left and right sets, where items in the right set must appear if those in the left set are present. These rules reveal the configuration dependencies. If the tested file contains all items in a left set, this approach checks whether it includes the corresponding items in the right set. If any items are missing, it reports them.

• **Baseline 2**: Basic LLM method. It is designed using a straightforward prompt that does not take our multi-dimensional constraints into account. This prompt contains the configuration file content followed by a task description. Similarly, the output is enclosed within a locator pair, "<START>" and "<END>", to delimit the required response. This prompt is shown in Fig. 4.

• **Alternative Baselines**: We design other LLM-based methods: (1) splitting our multi-dimensional constraints into separate prompts and integrating their results, denoted as Separated LLM method, and (2) employing few-shot prompting with a small sample set, denoted as Few-shot LLM method. For the first method, we divide the constraints into five separate prompts, each corresponding to a

Configuration File	Task Description	Response
Resources: BucketEventConsumer: Type: AWS::Serverless::Function Properties: Handler: BucketEventConsumer.main. lambda_handler Runtime: python3.8 CodeUri: bundle.zip	■ **Question**: Are there any misconfigurations in the above configuration file?	➢ Answer format (You **MUST** follow this): Detected errors are written between <START> and <END> tags:

Fig. 4. The prompt of basic LLM method.

specific constraint type. We then aggregate the results (enclosed within a locator pair, "<START>" and "<END>") from these prompts. For the second method, we design a three-shot learning approach inspired by common comparison practices in software engineering research [46, 48]. We randomly select three configuration files from the 701 correctly configured samples. Two of these files are injected with misconfigurations, while the third maintains correct configurations without injected errors. We provide the corresponding detection results for these examples to guide the LLMs.

4.4 Evaluation Dataset

We conduct experimental evaluations on a dataset comprising three types of configurations. The first type includes error-free configurations, enabling us to evaluate true negatives and false positives in detection. The second type contains configurations with real-world errors, allowing for the assessment of true positives and false negatives. Although this second type is somewhat free of data leakage concerns of LLMs, we include a third type to strengthen the validity of our conclusions. The third type consists of configurations with injected errors, which are not exposed to LLMs during training, thereby eliminating data leakage concerns. By utilizing these diverse configurations, we can achieve a valid and comprehensive evaluation.

• *Configurations without Errors (26).* We manually collect configuration files that have been successfully executed without errors. This data is separate from the one used to mine configuration patterns in the data-driven approach.

We collect real-world configuration cases from GitHub. GitHub issues provide rich information, including developer discussions and related code or configuration fragments. We conduct the following steps. First, on July 2, 2024, the date we collected this data, we searched GitHub using the keywords "AWS," "serverless," and "configuration," which yielded more than 8,000 relevant configuration-related issues. We then manually reviewed these issues to extract correct configuration fragments from the problematic cases—a time-consuming and challenging process. To facilitate this task, the first two authors jointly review the configurations. Initially, they filter through the configuration fragments by searching for terms including "successful," "successfully," and "it works" within the issues to identify correct configurations. For the fragments that matched, they conducted a manual verification process to ensure that the configurations were indeed error-free. Over two months, the two authors identified 52 configuration fragments that met our criteria. These error-free real-world configuration fragments are divided into two sets: 26 (naming from case 1 to case 26) are used to evaluate error-free configurations, while the remaining 26 (naming from case 27 to case 52) are reserved for generating configurations with injected errors, which is explained in detail later.

• *Real-world Misconfigurations (58).* To evaluate the effectiveness of approaches in identifying real-world misconfigurations in serverless applications, we construct a relevant dataset by mining real-world configuration issues from GitHub. These issues need to contain clearly identified root causes as ground truths, enabling us to accurately assess the effectiveness of detection results.

The selection process is as follows: First, we use the same keywords (i.e., "AWS," "serverless," and "configuration") to search for relevant issues on GitHub on July 2, 2024. Next, we identify satisfied issues based on the following criteria: (i) the issue is marked as closed, indicating that it has been resolved; (ii) the issue includes a configuration fragment based on AWS SAM for analysis; and (iii) the discussion concludes with a clearly identified root cause of the problem. Using these criteria,

Table 2. Misconfiguration generation rules (we use generation rules from previous work [40, 41, 55, 65] and customize them in our scenario.)

Category	Subcategory	Specification	Generation Rules
Syntax	Resource type	Value set = {AWS::Serverless::Function, AWS::Serverless::Api, ...}	Generate a resource type that does not belong to the value set
	Entry	Value set = {entry1, entry2, ...}, specific entries are used in a certain resource type	Generate an invalid entry for a resource type
Range	Basic numeric	Valid range constrained by data type	Generate values outside the valid range (e.g., max value+1)
	Enum	Options, value set = {enum1, enum2, ...}, specific values are used in a certain configuration entry	Generate a value that does not belong to set
Dependency	Entry relationship	$(P_1, V, \diamond) \mapsto P_2, \diamond \in \{>, \geq, =, \neq, <, \leq, occurrence\}$	Generate invalid entry relationships for configuration entries $(P_1, V, \neg\diamond)$
	Value relationship	$(P_1, P_2, \diamond), \diamond \in \{>, \geq, =, \neq, <, \leq, occurrence\}$	Generate invalid value relationship for configuration entry values $(P_1, P_2, \neg\diamond)$

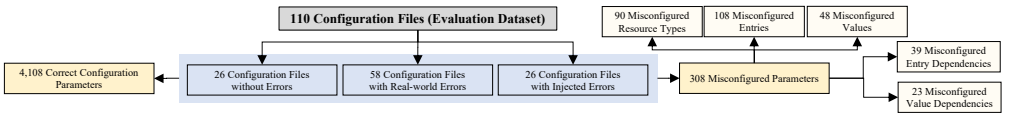

Fig. 5. The Details of Evaluation Dataset.

we select 58 real-world configuration problems encountered in serverless applications, surpassing the scale of prior studies on configuration-related research [70, 72].

To ensure the accuracy of the configuration errors to be detected, we meticulously review each real-world configuration file in conjunction with its identified root cause. During this process, we also manually identify and address any potential configuration issues (e.g., outdated runtime) that could influence the evaluation. Details of the modifications are available on our GitHub [16].

• *Injected Misconfigurations (26).* We construct injected misconfigurations by generating various errors in the correct configuration files. To achieve this, we use 26 error-free configuration files named from case 27 to case 52. Misconfigurations of different types are then generated, following misconfiguration generation rules from prior studies [40, 41, 43, 55, 65]. Prior studies [40, 43] showed that these rules can cover most configurations. In addition to utilizing existing rules, we extend specific misconfiguration generation rules tailored to serverless application configurations, as outlined in Table 2. For each selected configuration file, we randomly sample a configuration parameter that aligns with the subcategories in Table 2 and generate invalid configurations, creating a new erroneous configuration file for detection. In total, we generate 26 configuration files with injected misconfigurations for evaluation. Detailed changes are provided on our GitHub [16].

Our evaluation dataset contains 110 configuration files with corresponding ground-truth answers. Fig. 5 shows its details. Of these, 26 are error-free configuration files, 58 contain real-world errors, and 26 have injected errors. Across all configuration parameters, there are 4,108 correct configuration parameters and 308 misconfigured ones. Among the misconfigured parameters, 90 involve incorrect resource types, 108 have misconfigured entries, 48 contain incorrect values, 39 exhibit entry dependency issues, and 23 have value dependency issues. We analyze the detection results across all configuration parameters to obtain TP, FP, TN, and FN. We then calculate *precision*, *recall*, and *F1-score* to evaluate the effectiveness of the detection.

Table 3. RQ1: Results about *SlsDetector* and the data-driven method.

Methods	*precision*	*recall*	*F1-score*
Data-driven method with 5% threshold (default)	19.06%	70.78%	30.03%
SlsDetector (vs data-driven method with 5% threshold)	72.88% (↑ 53.82%)	88.18% (↑ 17.40%)	79.75% (↑ 49.72%)
Data-driven method with 10% threshold	17.70%	64.61%	27.79%
Data-driven method with 3% threshold	18.83%	70.13%	29.69%
Data-driven method with 1% threshold	18.85%	70.45%	29.75%

4.5 Experimental Settings

We introduce our parameter settings, experimental repetitions, and environment.

Parameter Settings. For **RQ1**, the compared data-driven method needs to specify a frequent threshold, α. We experiment with various threshold levels: low (1%), medium (3% and 5%), and high (10%). A lower threshold corresponds to a lower support value, enabling the discovery of more dependencies. For comparisons with *SlsDetector*, we use a default α value of 5%. Experimental results also show that 5% is optimal for achieving the best effectiveness results in the data-driven method. We also report results for both *SlsDetector* and the data-driven method across other thresholds. For **RQ2**, we compare *SlsDetector* with the basic LLM method and other alternative methods, all of which leverage LLMs. We select ChatGPT-4o as the default LLM due to the widespread use and outstanding performance of ChatGPT in recent research [43, 71]. A crucial parameter of LLMs is the temperature, which controls the level of randomness in the generated responses. To ensure reproducibility and consistency, we follow the previous work [22, 30, 64, 68] to set the temperature to 0 for all identical queries. For **RQ3**, we set the temperature to 0 by default to conduct a detailed analysis of the non-determinism of LLMs. Additionally, we also set the temperature to 0.2 and 0.5 to evaluate the robustness of *SlsDetector* in real-world scenarios. For **RQ4**, we evaluate the generalization capability of *SlsDetector* across various LLMs, excluding ChatGPT-4o. Specifically, we utilize an open-source model, Llama 3.1 (405B) Instruct Turbo, and a proprietary model, Gemini 1.5 Pro. These models are among the top-ranked LLMs [11]. We also leverage DeepSeek V3, a recently emerging and widely recognized model. As with RQ2, we set the temperature of LLMs to 0 to maintain consistent outputs across repeated queries.

Experimental Repetitions. For experiments involving stochastic processes, we follow established best practices [30, 64], repeating each experiment five times and reporting the mean evaluation metrics to reduce the impact of random variations.

Experimental Environment. Our experiments were conducted on an Ubuntu 18.04.4 LTS server with an Intel Xeon (R) 4-core processor and 24 GiB of memory. The LLMs were accessed through their respective APIs. While all methods are implemented in Python, their misconfiguration detection capabilities are independent of the underlying programming language.

5 EVALUATION RESULTS

This section presents the results of each research question.

5.1 RQ1: Effectiveness of *SlsDetector* and Data-Driven Method

This section explores the effectiveness of *SlsDetector* in comparison to the data-driven method. *SlsDetector* has a significant advantage in the effectiveness aspect. Table 3 presents their results in detecting misconfigurations in serverless applications. Specifically, *SlsDetector* achieves a *precision* of 72.88%, *recall* of 88.18%, and *F1-score* of 79.75%. In contrast, the data-driven method, with its default threshold of 5%, only reaches a *precision* of 19.06%, *recall* of 70.78%, and *F1-score* of 30.03%. *SlsDetector* outperforms the data-driven method, increasing *precision* by 53.82 percentage points,

Table 4. RQ1: Results* of TP, FN, FP, and TN for data-driven method and *SlsDetector*.

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
Data-driven method (default)	218 (70.78%)	90 (29.22%)	**926 (22.54%)**	3,182 (77.46%)
SlsDetector (default)	272 (88.31%) ✓	36 (11.69%) ✓	102 (2.48%) ✓	4,006 (97.52%) ✓

* Higher TP and TN are preferable, while lower FN and FP are desired.

Table 5. RQ1: Results of TP, FN, FP, and TN for the data-driven method with different thresholds α.

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
Data-driven method with 10% threshold	199	109	925	3,183
Data-driven method with 3% threshold	216	92	931	3,177
Data-driven method with 1% threshold	217	91	934	3,174

recall by 17.40 percentage points, and *F1-score* by 49.72 percentage points, showing its superior effectiveness.

We investigate why the data-driven method produces less effective results. One major issue is its low *precision* (19.06%) and *F1-score* (30.03%). We further observe TP, FN, FP, and TN values obtained by the data-driven method across all configuration parameters, as shown in Table 4. Results show that the FP value is 926, indicating that 22.54% of the 4,108 correct configuration parameters are mistakenly flagged as misconfigurations. In contrast, on average, *SlsDetector* misclassifies only 2.48% of correct configuration parameters as misconfigurations. Thus, the low effectiveness of the data-driven method is attributed to high false positives. The data-driven method learns configuration patterns based on historical data, which mainly includes previously used configurations. This reliance makes it difficult to accurately identify configurations that are either rare or newly supported, resulting in numerous false positives. Thus, the data-driven method fails to detect some valid configurations that are indeed supported, leading to its low *precision* and *F1-score*.

We also compare the effectiveness of the data-driven method under different thresholds α: 10%, 3%, and 1%, with the results presented in Table 3. As α decreases from 10% to 1%, the evaluation metrics show improvement. Specifically, *precision* increases from 17.70% to 18.85%, *recall* rises from 64.61% to 70.45%, and *F1-score* improves from 27.79% to 29.75%. To further explore the reasons for their changes, we give TP, FN, FP, and TN results of the data-driven method under different thresholds, as shown in Table 5. The primary reason for improvements is that lower α mines more dependencies among entries or values. This enables the accurate identification of a larger number of misconfigured parameters. Specifically, the TP value for the data-driven method at a 10% threshold is 199, whereas at a 1% threshold, it increases to 217. This improvement leads to a higher *recall*, increasing from 64.61% to 70.45%. However, a lower α also increases the risk of generating potentially invalid dependencies, resulting in correctly configured parameters being mistakenly flagged as misconfigurations. This is evident from the FP values: the FP value for the data-driven method at a 10% threshold is 925, while at a 1% threshold, it increases to 934. As a result, *precision* shows only a modest improvement, from 17.70% to 18.85%. For *F1-score*, lowering α enhances the effectiveness of the data-driven method, reaching a value of 29.75%. However, it still significantly lags behind the 79.75% achieved by *SlsDetector*.

In addition, we observe that a threshold of 5% for the data-driven method yields superior results compared to 1%, 3%, and 10%, suggesting that 5% is an optimal threshold for the data-driven method in this scenario. In the threshold of 5%, the FP-growth algorithm can effectively mine relationships without losing valid dependencies or generating an excessive number of invalid dependencies. However, even at 5%, the effectiveness of the data-driven method remains significantly lower than that of *SlsDetector*, with particularly low *precision* and *F1-score*.

Table 6. RQ1: The number of misconfigured parameters correctly identified as misconfigured across different categories for the data-driven method and our approach.

Methods	Misconfigured resource types (90)	Misconfigured entries (108)	Misconfigured values (48)	Misconfigured entry dependencies (39)	Misconfigured value dependencies (23)
Data-driven method	89 (98.89%) ✓	79 (73.15%)	25 (52.08%)	12 (30.77%)	10 (43.48%)
SlsDetector	84 (93.33%)	93 (86.11%) ✓	43 (89.58%) ✓	38 (97.44%) ✓	19 (82.61%) ✓

To assess the ability of different methods to identify various types of misconfigurations, we analyze the number of misconfigured parameters correctly detected across different categories by our approach and the data-driven method. As shown in Table 6, the data-driven method demonstrates a higher detection rate for resource type errors (98.89%) compared to *SlsDetector* (93.33%). This suggests that the 701 historical configuration samples used by the data-driven method cover the most commonly applied resource types, leading to a stronger effectiveness in this category. Resource types are relatively high-level objects that encompass multiple configuration entries, making them less numerous and easier to match against historical data. However, the detection rate of *SlsDetector* in this category remains high, exceeding 90% with only a minimal gap. In addition, in most other misconfiguration categories, *SlsDetector* outperforms the data-driven method. Specifically, *SlsDetector* identifies a higher proportion of errors in configuration entries (86.11%), values (89.58%), entry dependencies (97.44%), and value dependencies (82.61%). The lower effectiveness of the data-driven method in these categories may be attributed to several factors. First, many configuration entries and values in real-world settings may not have been previously encountered in historical data, leading to gaps in coverage. Second, constraints governing values may be incomplete or underrepresented in the dataset, limiting the effectiveness of historical value matching. Finally, implicit dependencies—particularly those related to entry relationships and value relationships, which are complex and context-dependent, making them difficult to extract purely from past configurations. These results indicate the advantages of *SlsDetector* in detecting a wider range of misconfigurations, especially in cases where historical data is insufficient in the data-driven method to generalize to unseen configuration scenarios.

> **Ans. to RQ1:** *SlsDetector*, which does not rely on learning from a large number of real examples, achieves a *precision* of 72.88%, *recall* of 88.18%, and *F1-score* of 79.75%, surpassing data-driven methods across all metrics. It shows significant improvements, with increases of 53.82 percentage points in *precision*, 17.40 percentage points in *recall*, and 49.72 percentage points in *F1-score*. These results suggest the high effectiveness of *SlsDetector*.

5.2 RQ2: Effectiveness of *SlsDetector* and LLM-based Baselines

We explore the effectiveness of *SlsDetector* in comparison to the basic LLM method and other alternative methods (separated LLM method and few-shot LLM method) using the default ChatGPT-4o for detecting misconfigurations in serverless applications. Table 7 presents their results, showing that *SlsDetector* is more effective than the basic LLM method and other alternative methods. Specifically, *SlsDetector* achieves a *precision* of 72.88%, a *recall* of 88.18%, and an *F1-score* of 79.75%. In contrast, the basic LLM method achieves a *precision* of 51.65%, *recall* of 65.00%, and an *F1-score* of 57.55%. For the separated LLM method, the results indicate a *precision* of 48.20%, a *recall* of 87.73%, and an *F1-score* of 62.20%. The few-shot LLM method achieves a *precision* of 70.10%, a *recall* of 66.95%, and an *F1-score* of 68.43%. For specific analyses, *SlsDetector* significantly outperforms the basic LLM methods across all evaluation metrics, with improvements of 21.23 percentage points in *precision*, 23.18 percentage points in *recall*, and 22.20 percentage points in *F1-score*. For the

Table 7. RQ2: Results about *SlsDetector* and the compared LLM-based methods using the default LLM (ChatGPT-4o).

Baseline	*precision*	*recall*	*F1-score*	**Our Approach**	*precision*	*recall*	*F1-score*
Basic LLM method	51.65%	65.00%	57.55%	***SlsDetector*** (vs Basic LLM method)	72.88% (↑ 21.23%)	88.18% (↑ 23.18%)	79.75% (↑ 22.20%)
Separated LLM method	48.20%	87.73%	62.20%	***SlsDetector*** (vs Separated LLM method)	72.88% (↑ 24.67%)	88.18% (↑ 0.45%)	79.75% (↑ 17.55%)
Few-shot LLM method	70.10%	66.95%	68.43%	***SlsDetector*** (vs Few-shot LLM method)	72.88% (↑ 2.78%)	88.18% (↑ 21.23%)	79.75% (↑ 11.32%)

Table 8. RQ2: Results of TP, FN, FP, and TN for the compared LLM-based methods and *SlsDetector*, on average.

Methods	308 misconfigured parameters		4,108 correct configuration parameters	
	TP	**FN**	**FP**	**TN**
Basic LLM method (default)	200 (64.94%)	107 (34.74%)	188 (4.58%)	3,920 (95.42%)
Separated LLM method (default)	270 (87.66%) ✓	38 (12.34%) ✓	291 (7.08%)	3,817 (92.92%)
Few-shot LLM method (default)	207 (67.21%)	102 (33.12%)	88 (2.14%) ✓	4020 (97.86%) ✓
SlsDetector (default)	272 (88.31%) ✓	36 (11.69%) ✓	102 (2.48%) ✓	4,006 (97.52%) ✓

* Higher TP and TN are preferable, while lower FN and FP are desired.

separated LLM method, *SlsDetector* also shows significant improvements in *precision* and *F1-score*, with increases of 24.67 and 17.55 percentage points, respectively. Compared to the few-shot LLM method, *SlsDetector* significantly achieves gains of 21.23 and 11.32 percentage points on the metrics of *recall* and *F1-score*, respectively. Overall, these results indicate that the compared methods are less effective than our approach across all evaluation metrics.

We analyze the factors contributing to the lower effectiveness of the compared methods. Table 8 presents TP, FN, FP, and TN values obtained by each method across all configuration parameters. The results reveal that the basic LLM and few-shot LLM methods exhibit low TP values of 200 and 207, correctly identifying only 64.94% and 67.21% of the 308 misconfigured parameters, respectively. In contrast, *SlsDetector* and the separated LLM method accurately detect an average of 272 (88.31%) and 270 (87.66%) misconfigured parameters, respectively. Although the separated LLM method achieves a *recall* comparable to *SlsDetector*, indicating that most true misconfigured parameters are correctly identified, it suffers from a lower *precision* (48.20%) and *F1-score* (62.20%). This suggests a higher proportion of correctly configured parameters being mistakenly flagged as misconfigured, as reflected by its high FP value (291, 7.08%) in Table 8. For the few-shot method, while it maintains a low FP value (2.14%), it struggles with low TP (67.21%), leading to a lower *recall* (66.95%) and *F1-score* (68.43%) due to its reduced ability to correctly identify true misconfigured parameters. Overall, these analyses demonstrate that *SlsDetector* outperforms the compared methods. By leveraging multi-dimensional constraints, our approach achieves a better balance between *precision* and *recall*, effectively reducing false positives while maintaining high misconfiguration detection accuracy.

To further examine the factors influencing detection effectiveness, we analyze the average number of correctly identified misconfigured parameters across different categories for both the baselines and our approach. As shown in Table 9, the basic LLM and few-shot methods consistently underperform compared to *SlsDetector* across all categories, detecting fewer errors in resource types (68.89% and 66.67%), entries (76.85% and 63.89%), values (81.25% and 77.08%), entry dependencies (17.95% and 71.79%), and value dependencies (52.17% and 52.17%). In contrast, *SlsDetector* achieves

Table 9. RQ2: The average number of misconfigured parameters correctly identified as misconfigured across different categories for the compared LLM-based methods and our approach *SlsDetector*.

Methods	Resource types (90)	Entries (108)	Values (48)	Entry dependencies (39)	Value dependencies (23)
Basic LLM method	62 (68.89%)	83 (76.85%)	39 (81.25%)	7 (17.95%)	12 (52.17%)
Separated LLM method	84 (93.33%)	88 (81.48%)	42 (87.50%)	34 (87.18%)	21 (91.30%)
Few-shot LLM method	60 (66.67%)	69 (63.89%)	37 (77.08%)	28 (71.79%)	12 (52.17%)
SlsDetector	84 (93.33%)	93 (86.11%)	43 (89.58%)	38 (97.44%)	19 (82.61%)

higher detection effectiveness, identifying 93.33% of misconfigured resource types, 86.11% of entries, 89.58% of values, 97.44% of entry dependencies, and 82.61% of value dependencies. These results confirm that each constraint incorporated into *SlsDetector* contributes positively to detecting corresponding misconfigurations.

For the basic LLM method, the most significant gap between basic LLM and *SlsDetector* is observed in entry dependency detection. Further analysis reveals that the basic LLM method struggles to identify cross-entry dependencies, particularly those involving cloud service resources that must co-occur with event sources in serverless functions. For instance, the configuration entry `RestApiId` under an event source of type "Api" should be associated with configuration entries of the "AWS::Serverless::Api" resource type. The basic LLM method fails to capture such relationships, leading to a high number of undetected misconfigurations. The few-shot LLM method performs worse than both *SlsDetector* and the basic LLM method in detecting misconfigured configuration entries. This is because configuration entries are widely distributed across diverse resource types, making it challenging to generalize simple examples of the few-shot LLM method across all configurations. The few-shot prompting in the few-shot LLM method lacks sufficient coverage of constraint detection and contextual awareness, limiting its ability to guide accurate detection across a wide variety of configuration entries. In addition, the separated LLM method achieves comparable *recall* to *SlsDetector*, indicating that it can accurately detect misconfigurations across different categories, as shown in Table 9. However, it suffers from a high false positive (FP) rate, as correctly configured parameters are frequently misclassified as misconfigured. This indicates that splitting multi-dimensional constraints into separate prompts and integrating their results is insufficient for effective misconfiguration detection. A more holistic approach, incorporating multiple interdependent constraints, is essential to enhance effectiveness.

Overall, these results highlight the limitations of relying on the raw capabilities of LLMs (basic LLM method), splitting multi-dimensional constraints into separate prompts (separated LLM method), and employing few-shot prompting with a small sample set (few-shot method). These baselines rely on isolated prompts or limited examples. Another possible reason for their weak performance is that they receive less attention during prompt engineering. The effectiveness of *SlsDetector* stems from its ability to integrate multi-dimensional constraints holistically, ensuring that LLM-based inference is guided by contextual dependencies. These constraints are systematically designed across various configuration dimensions, allowing *SlsDetector* to capture intricate relationships to obtain more effective misconfiguration detection.

Ans. to RQ2: *SlsDetector* outperforms the LLM-based baseline methods across all metrics using the default ChatGPT-4o. Results also suggest that integrating multi-dimension constraints is beneficial for handling misconfiguration detection in serverless applications.

Table 10. RQ3: Evaluation metrics results of *SlsDetector* across five repetitions.

LLM Factor	Metrics	Repetition 1	Repetition 2	Repetition 3	Repetition 4	Repetition 5	Mean
temperature=0	*precision*	71.83%	70.78%	70.35%	75.28%	76.14%	72.88%
	recall	91.88%	91.23%	84.74%	86.04%	87.01%	88.18%
	F1-score	80.63%	79.72%	76.88%	80.30%	81.21%	79.75%
temperature=0.2	*precision*	67.82%	74.30%	71.71%	72.18%	73.86%	71.97%
	recall	82.79%	86.36%	83.12%	85.06%	84.42%	84.35%
	F1-score	74.56%	79.88%	76.99%	78.09%	78.79%	77.66%
temperature=0.5	*precision*	77.35%	71.05%	73.70%	71.24%	71.84%	73.04%
	recall	85.39%	86.04%	82.79%	87.66%	81.17%	84.61%
	F1-score	81.17%	77.83%	77.98%	78.60%	76.22%	78.36%

5.3 RQ3: Impact of Non-determinism on *SlsDetector*

We explore how the non-determinism of LLMs impacts our evaluation results, with a temperature of 0 by default. As detailed in Section 4.5, each experiment is repeated five times. We analyze their results shown in Table 10. Results show that while the non-determinism of LLMs influences evaluation results, its effect is relatively minor, with a variance of about 5 percentage points. When the temperature of LLMs is 0, *SlsDetector* consistently achieves high effectiveness across different trials. *precision* ranges from 70.35% to 76.14%, *recall* varies between 84.74% and 91.88%, and *F1-score* falls between 76.88% and 81.21%. Even the lowest values, i.e., *precision* at 70.35%, *recall* at 84.74%, *F1-score* at 76.88%, are still higher than *precision* (19.06%), *recall* (70.78%), and *F1-score* (30.03%) of the data-driven approach. Furthermore, the lowest metric values for *SlsDetector* remain approximately 20 percentage points higher than the average results (i.e., *precision* at 51.65%, *recall* at 65.00%, *F1-score* at 57.55%) of the basic LLM-based method. This suggests that our conclusions regarding *SlsDetector* are not affected by the non-determinism of LLMs.

To further assess the robustness of *SlsDetector* in real-world scenarios, we analyze the experimental results at temperatures 0.2 and 0.5. As shown in Table 10, there is a relatively minor impact, with a variance of approximately 5 percentage points on the same evaluation metric. Despite these variations, the effectiveness remains consistently high across all evaluation metrics, comparable to the default setting (temperature = 0). Specifically, when temperature = 0.2, *precision* ranges from 67.82% to 74.30%, *recall* varies between 82.79% and 86.36%, and *F1-score* falls between 74.56% and 79.88%, with respective differences of 6.48, 3.57, and 5.32 percentage points. When temperature = 0.5, *precision* ranges from 71.05% to 77.35%, *recall* varies between 81.17% and 87.66%, and *F1-score* falls between 76.22% and 81.17%, with respective differences of 6.31, 6.49, and 4.95 percentage points. Overall, even the lowest observed values surpass those of the data-driven approach. Additionally, the lowest metrics values remain approximately 20 percentage points higher than the average results of the compared basic LLM method. These results confirm that *SlsDetector* maintains strong robustness in real-world settings.

> **Ans. to RQ3:** Our conclusions are not impacted by the non-determinism of LLMs.

5.4 RQ4: Generalization Capability of *SlsDetector*

To explore the generalization capability of *SlsDetector*, we use three additional LLMs: Llama 3.1 (405B) Instruct Turbo model, Gemini 1.5 Pro model, and DeepSeek V3 model. *SlsDetector* consistently achieves high effectiveness across all metrics, with *precision*, *recall*, and *F1-score* values exceeding 70%, regardless of the LLM utilized. Table 11 shows their results. Specifically, with the Llama 3.1 (405B) Instruct Turbo, *SlsDetector* achieves a *precision* of 70.27%, *recall* of 78.38%, and an *F1-score* of 74.05%. With the Gemini 1.5 Pro model, *SlsDetector* yields a *precision* of 71.72%, *recall* of 74.35%,

Table 11. RQ4: Results about *SlsDetector* and basic LLM method using various LLMs.

Basic LLM Method (BL)	*precision*	*recall*	*F1-score*	***SlsDetector***	*precision*	*recall*	*F1-score*
GPT-4o	51.65%	65.00%	57.55%	GPT-4o (vs BL)	72.88% (↑ 21.23%)	88.18% (↑ 23.18%)	79.75% (↑ 22.20%)
Llama	48.88%	58.38%	53.09%	Llama (vs BL)	70.27% (↑ 21.39%)	78.38% (↑ 20.00%)	74.05% (↑ 20.96%)
Gemini	44.41%	22.86%	30.11%	Gemini (vs BL)	71.72% (↑ 27.31%)	74.35% (↑ 51.49%)	72.93% (↑ 42.82%)
DeepSeek	56.59%	57.66%	57.11%	DeepSeek (vs BL)	70.71% (↑ 14.12%)	77.86% (↑ 20.19%)	74.10% (↑ 16.99%)
Gemini with Type	45.91%	40.58%	43.06%	Llama 3.1 8B	60.42%	75.06%	66.85%

and an *F1-score* of 72.93%. With the DeepSeek V3 model, *SlsDetector* yields a *precision* of 70.71%, *recall* of 77.86%, and an *F1-score* of 74.10%. Among these, *SlsDetector* with ChatGPT-4o offers the highest effectiveness, while *SlsDetector* with the Gemini 1.5 Pro model shows comparatively lower metrics but still achieves a high *F1-score* of 72.93%.

We also evaluate the basic LLM method with different LLMs, shown in Table 11. We observe considerable variability. While the basic LLM method achieves *precision*, *recall*, and *F1-score* values approaching or exceeding 50% when using ChatGPT-4o, Llama 3.1 (405B) Instruct Turbo, and DeepSeek V3, its effectiveness drops substantially with the Gemini 1.5 Pro model, where *precision* is 44.41%, *recall* is 22.86%, and *F1-score* is 30.11%. This indicates a key limitation of the basic LLM method: its effectiveness is dependent on the specific LLM used. In contrast, *SlsDetector* provides the ability to maintain consistent effectiveness across different models, showing its generalization.

We compare the effectiveness differences between *SlsDetector* and the basic LLM method when using the same LLM. From Table 11, *SlsDetector* outperforms the basic LLM method by over 20 percentage points across all evaluation metrics, regardless of ChatGPT-4o and Llama 3.1 (405B) Instruct Turbo models. For the Deepseek V3 model, *SlsDetector* also improves on all evaluation metrics by almost 20 percentage points, which are respectively 14.12, 20.19, and 16.99. With the Gemini 1.5 Pro model, *SlsDetector* outperforms the basic LLM method with even greater gains, achieving 27.31 percentage points higher in *precision*, 51.49 percentage points higher in *recall*, and 42.82 percentage points higher in *F1-score*. The effectiveness gap is especially pronounced with Gemini 1.5 Pro, showing an effectiveness difference of around 50% in *recall* and *F1-score*, underscoring the effectiveness of our approach. The weak performance shown by the basic LLM method when using the Gemini 1.5 Pro model is likely related to the type specification of the configuration file given to the LLM, making it difficult for the LLM to infer the intended task. In this situation, we try to explicitly specify the configuration file type when using the Gemini 1.5 Pro model in the basic LLM method. Specifically, we indicate that the configuration file is the type of AWS SAM before performing misconfiguration detection. The obtained values for *precision*, *recall*, and *F1-score* are 45.91%, 40.58%, and 43.06%, respectively. These results suggest that explicitly specifying the file type leads to a modest improvement in performance across all metrics, with a particular gain in *recall*, and *F1-score*. While more misconfigurations are identified under this setting, the performance gap, especially in *recall*, and *F1-score*, remains significant, at approximately 40%, when compared to our proposed approach. Therefore, the relatively weak performance of the Gemini 1.5 Pro model appears to stem not primarily from the lack of type specification in the prompt, but from intrinsic limitations of the Gemini model itself.

As consumer hardware becomes increasingly capable of running smaller-scale LLMs, such as Llama 3.1 8B, evaluating the effectiveness of *SlsDetector* in smaller LLMs is crucial for real-world

applicability. To investigate this, we apply the Llama 3.1 8B model to our approach and assess its effectiveness. The results indicate that the obtained *precision*, *recall*, and *F*1-*score* are 60.42%, 75.06%, and 66.85%, respectively. Although these values are relatively lower than those achieved with larger LLMs (ChatGPT-4o, Gemini 1.5 Pro, Llama 3.1 405B, and DeepSeek V3), they remain superior to the effectiveness of the basic LLM method using large-scale models, demonstrating the effectiveness of our approach even with smaller LLMs. This underscores the adaptability and effectiveness of *SlsDetector* even when applied to smaller, more accessible models. Thus, *SlsDetector* shows promise for real-world applications, particularly in contexts where computational resources may be constrained.

> **Ans. to RQ4:** *SlsDetector* exhibits generalization capability, consistently achieving highly effective results across various LLMs. In contrast, the effectiveness of the basic LLM method varies significantly depending on the chosen LLM. When using the Gemini 1.5 Pro model, *SlsDetector* outperforms the basic LLM method by approximately 50 percentage points in both *recall* and *F*1-*score*.

6 THREATS TO VALIDITY

Data Leakage Concerns. One potential risk when using LLMs is data leakage, as these models are trained on vast datasets. Specifically, open-source configuration data may have been exposed to the LLMs utilized in *SlsDetector*, raising concerns about memorization of our evaluated error-free configurations available on platforms such as GitHub. However, during our evaluation, we found that the model did not recognize outdated configuration values as correct, suggesting that the error-free configurations we evaluated were not fully present in the LLM's training data. Note that outdated configuration values were manually corrected before our experimental evaluation.

Our evaluation data also includes both real-world and manually injected misconfigurations. The ground truths for real-world errors are established through an analysis of developer discussions on GitHub to identify root causes. Injected misconfigurations are deliberately introduced into correct configurations through misconfiguration generation rules. These misconfigurations were not exposed to the LLM during training. In addition, the number of configuration files evaluated with errors (58 real-world + 26 injected = 84) significantly exceeded those without errors (26). Thus, the likelihood of our effectiveness results being significantly affected by data leakage is negligible.

We also compare the effectiveness of the basic LLM method without our multi-dimensional constraints. The basic LLM method yields a *precision* of 51.65%, a *recall* of 65.00%, and an *F*1-*score* of 57.55%, indicating low effectiveness. If our evaluation dataset had been exposed to the LLMs, we would expect the basic LLM method to achieve significantly higher results. However, the results do not reflect this, suggesting that our evaluation dataset was not exposed to the LLMs. *SlsDetector* incorporates multi-dimensional constraints to detect the same evaluation dataset, resulting in improved metrics: a *precision* of 72.88%, a *recall* of 88.18%, and an *F*1-*score* of 79.75%. These enhancements indicate that our design effectively boosts detection results, rather than being influenced by potential data leakage.

Impact of LLM Knowledge Limitations. Since *SlsDetector* relies on the internal knowledge of the LLMs, it may struggle with misconfigurations involving rarely used dependencies, a challenge faced by traditional data-driven methods. To assess this, we analyze whether our approach encounters similar limitations, particularly when handling seldom-used configuration dependencies. As shown in Table 6, the data-driven method correctly identifies only 30.77% of misconfigured entry dependencies and 43.48% of misconfigured value dependencies, whereas our approach achieves 97.44% and 82.61%, respectively. These results suggest that LLMs possess broader internal knowledge,

allowing them to detect a wider range of misconfigurations than data-driven methods. However, 100% accuracy is not achieved, indicating that LLMs have inherent knowledge limitations. Some configurations may not be fully covered in the LLM's pre-trained knowledge base, leading to occasional misclassifications. Despite these challenges, our approach achieves significant improvements over traditional data-driven methods in identifying misconfigured dependencies, highlighting the advantages of leveraging LLMs for misconfiguration detection while also acknowledging their inherent limitations.

Randomness Concerns. Randomness in evaluation results primarily stems from the inherent nondeterminism of LLMs. To address this, we set a crucial parameter, specifically initializing temperature as 0, to ensure that the model produces consistent outputs for the same input. While this can reduce randomness, minor fluctuations may still arise due to underlying probabilistic mechanisms within the model. To further minimize randomness, we conduct five independent experiments for each experimental setup and use the mean results as the final outcome. By combining temperature control with multiple experiment repetitions, we ensure that the impact of randomness on our evaluation results is effectively minimized.

Deployment Evaluation Limitation. In our experimental evaluation, we do not conduct real-world deployment to assess the effectiveness of our misconfiguration detection approach. Such deployment introduces numerous uncontrolled variables, such as hardware differences, network conditions, and external dependencies, making systematic evaluation highly challenging. This is also why prior misconfiguration detection research [50, 55, 58] has commonly relied on curated datasets of configuration files with known ground-truth errors rather than real-world deployment. Our evaluation follows this established practice.

7 DISCUSSION

Generalization of *SlsDetector*. Although this paper primarily focuses on serverless computing configurations, the design principles underlying *SlsDetector* are broadly applicable to various software configuration contexts. The core methodology of *SlsDetector* is based on multi-dimensional structural constraints, which define generalized detection rules by capturing fundamental configuration components of serverless applications rather than domain-specific parameters. These constraints operate at an abstract level, making them adaptable across different configurable systems without requiring extensive customization. This characteristic aligns with broader principles in software configuration research [49, 54]. While serverless platforms like Google Cloud Functions and Microsoft Azure Functions currently lack a formal configuration schema, our approach remains applicable if standardized configuration objects are available. The core of *SlsDetector* relies on analyzing configuration constraints rather than being tied to a specific cloud provider.

On the other hand, *SlsDetector* leverages LLMs for misconfiguration detection, allowing it to support diverse configuration formats, including JSON, XML, infrastructure-as-code templates, and program code. Since the detection process is structured around high-level constraints, adapting *SlsDetector* to new environments primarily requires adjusting key constraint definitions. Furthermore, our evaluation in RQ4 demonstrates that *SlsDetector* maintains consistent effectiveness across different LLMs, e.g., ChatGPT-4o, Gemini 1.5 Pro, Llama 3.1 (405B), and DeepSeek V3, suggesting the generalization capability of *SlsDetector*. As fine-tuned LLMs continue to advance, the efficiency of *SlsDetector* may further improve, making it a practical and scalable solution for a wide range of software configuration systems.

Scope of Error Detection. The scope of this paper is misconfiguration detection within configuration files of serverless applications. Therefore, if the application logic is implemented within the configuration file, *SlsDetector* can detect related errors. We consider two cases: incorrect function definitions and misconfigured function triggers. (1) For function definitions explicitly specified

in the "Handler" entry of the configuration file, *SlsDetector* applies its configuration entry and value checks to detect issues such as missing or improperly defined handlers. However, if function definitions reside outside the configuration file (e.g., in the application's source code), they are beyond the scope of this work. *SlsDetector* is specifically designed to validate AWS SAM configurations, not to perform general-purpose source code analysis. (2) Since function triggers are also defined within the configuration file, *SlsDetector* can effectively detect their misconfigurations. It leverages entry- and value-level dependency analyses to examine the relationships between event sources and their corresponding functions, enabling the identification of incorrect event-trigger bindings directly within the configuration. Ensuring completely error-free deployments requires a combination of approaches, including misconfiguration detection, program analysis, and runtime error detection and repair, similar to practices in traditional software development. This paper focuses on misconfiguration detection as a critical step toward reliable deployment.

Design Choice. In the context of LLM-based code generation, many approaches rely on iterative refinement guided by runtime feedback. However, such mechanisms typically require actual deployment, which introduces significant variability and is not standard practice in misconfiguration detection, as discussed in Section 6. Therefore, *SlsDetector* is designed to perform static misconfiguration detection, rather than iterative refinement based on runtime feedback. Its primary goal is to reduce early-stage deployment failures caused by configuration file errors, rather than to serve as a comprehensive deployment verification.

8 RELATED WORK

8.1 Serverless Computing

The increasing adoption of serverless computing has attracted widespread interest from the research community, particularly within SE. A broad range of topics has been explored, including literature reviews [60], evolution and current state [56], and analyses of serverless application characteristics [26, 27]. Additional research has delved into the challenges faced by developers [61], the development of stateful serverless applications [20], and methods for testing and debugging [38]. For example, a comprehensive literature review [60] was conducted to explore the breadth and depth of serverless computing research. Eismann *et al.* [26] analyzed 89 serverless applications to assess them from multiple dimensions. Wen *et al.* [61] identified 36 challenges developers face when developing serverless applications, highlighting configuration issues as a prominent concern. Despite these efforts, to the best of our knowledge, no prior work has addressed misconfiguration detection in serverless computing. This paper fills this gap by introducing *SlsDetector*.

8.2 Traditional Misconfiguration Detection

Existing misconfiguration detection methods can be categorized into two types: white-box and black-box approaches. White-box approaches [47, 55, 57, 58, 63, 65] generally focus on source code or program analysis to identify misconfigurations within the codebase, relying on defined domain-specific rules. For example, Rex [47] detected dependency violations between source code and configurations that must be updated together. Ctest [55] identified configuration-induced failures in code affected by configuration changes. SPEX [65] employed static program analysis to infer configuration constraints, designing predefined rules from variables in the source code to uncover misconfiguration vulnerabilities. However, these methods are not well-suited for detecting misconfigurations in serverless applications, which rely on YAML-based configuration files rather than source code structures. Serverless-specific misconfigurations, embedded in configuration files, require new approaches that extend beyond traditional white-box techniques.

Black-box approaches [50, 51, 59, 72, 73] are generally data-driven and rely on learning configuration patterns from a dataset of example configurations. For instance, EnCore [72] used numerous configurations to learn and customize rule templates, inferring correlations and detecting misconfigurations in server applications. ConfigC [51] analyzed a dataset of correct configurations to build a language model that could detect errors in new configurations. DRIVE [73] created a Dockerfiles dataset and applied sequential pattern mining to extract frequent patterns, identifying rule violations through heuristic-based reduction and human intervention. However, these data-driven methods have inherent limitations: (i) They require a well-curated dataset, but ensuring the completeness and correctness of such datasets is challenging. As a result, configurations not represented in the training data may be missed, while normal configurations might be incorrectly flagged as anomalies due to dataset gaps. (ii) To compensate for dataset issues, these methods incorporate domain-specific knowledge (e.g., customized rule templates), requiring significant manual effort and continuous checking. These limitations hinder the practical application of data-driven approaches. Our results on RQ1 show that such approaches are less effective in our scenario.

8.3 LLM-based Misconfiguration Detection

LLM-based approaches offer a promising alternative. LogConfigLocalizer [52] addressed configuration-related error localization on Hadoop by leveraging log analysis and LLM. It worked by parsing runtime log messages and comparing them with fault-free logs stored in a database. LogConfigLocalizer then used predefined rules and LLM to localize the suspected root-cause configuration properties. However, LogConfigLocalizer primarily relies on runtime logs, which limits its ability to detect misconfigurations. In contrast, our approach focuses on identifying misconfigurations prior to deployment, offering an earlier intervention for preventing potential runtime errors. A recent paper presented Ciri [43], an LLM-based configuration validator. It demonstrated the potential of LLMs for detecting misconfigurations in systems such as Alluxio, Django, Etcd, and HDFS. However, Ciri depends on an external database containing valid configurations, misconfigurations, related questions, and ground-truth responses. Constructing this database is costly and challenging for various scenarios. In contrast, *SlsDetector* employs zero-shot prompting that does not require external datasets, eliminating the need for predefined data. On the other hand, Ciri used a prompt without any constraint, limiting its ability to detect dependencies [43]. Serverless applications have complex configuration structures and stronger interdependencies, making simple prompt-based methods less effective. Our results on RQ2 and RQ4 show that such a method (i.e., basic LLM method) is less effective in our scenario. Instead, *SlsDetector* incorporates carefully designed multi-dimensional constraints without predefined data, providing a more effective detection for serverless application configurations.

9 CONCLUSION

Our work opens a promising research direction, showing that LLMs can effectively address configuration issues in cloud applications built on emerging serverless computing. Specifically, we introduced *SlsDetector*, the first LLM-based framework specifically designed for detecting misconfigurations in serverless applications. It did not rely on learning from a large number of real examples. *SlsDetector* leveraged advanced prompt engineering and zero-shot prompting to identify configuration issues with minimal input effort. *SlsDetector* included a prompt generation component that integrates the configuration file to be detected, task description, multi-dimensional constraints, and customized responses. Particularly, the multi-dimensional constraints are tailored to the characteristics of serverless applications, offering context-aware guidance using the Chain of Thought technique. The customized responses focused on both content and format demands, ensuring that the LLM outputs deterministic and clearly explained detection results. Our evaluation

on a curated dataset of 110 configuration files demonstrated that *SlsDetector* achieved a precision of 72.88%, recall of 88.18%, and F1-score of 79.75%, surpassing state-of-the-art data-driven methods by 53.82, 17.40, and 49.72 percentage points, respectively. Furthermore, we investigated the generalization capability of *SlsDetector* across various LLMs, finding that it consistently maintains high effectiveness across these models.

A promising direction for future work is the integration of hybrid static-dynamic misconfiguration detection. While static configuration analysis enables early identification of many classes of configuration issues, dynamic behaviors introduced during deployment present complementary challenges. In future work, we plan to augment our static detection results with selectively captured runtime metadata, broadening the coverage of misconfiguration detection.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62425203 and 62032003, Beijing Natural Science Foundation under Grant No. 4244076, and Beijing Postdoctoral Research Foundation under Grant No. 2024-ZZ-20.

REFERENCES

[1] 2024. Advantages of building serverless applications on AWS. https://www.embitel.com/blog/ecommerce-blog/advantages-of-building-serverless-applications-on-aws.
[2] 2024. Amazon S3. https://aws.amazon.com/s3/.
[3] 2024. AWS CloudFormation template. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-lambda-function.html.
[4] 2024. AWS resource and property types reference. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html.
[5] 2024. AWS SAM. https://aws.amazon.com/cn/serverless/sam.
[6] 2024. AWS SAM resources and properties. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-specification-resources-and-properties.html.
[7] 2024. AWS Serverless Application Repository. https://serverlessrepo.aws.amazon.com/applications.
[8] 2024. AWSTemplateFormatVersion. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/format-version-structure.html.
[9] 2024. Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/.
[10] 2024. Chain-of-Thought Prompting. https://learnprompting.org/docs/intermediate/chain_of_thought.
[11] 2024. Chatbot Arena LLM Leaderboard: Community-driven Evaluation for Best LLM and AI chatbots. https://lmarena.ai/.
[12] 2024. DoorDash confirms data breach affected 4.9 million customers, workers and merchants. https://techcrunch.com/2019/09/26/doordash-data-breach/.
[13] 2024. An example of the configuration file. https://github.com/aws/serverless-application-model/issues/214.
[14] 2024. Google Cloud Functions. https://cloud.google.com/functions.
[15] 2024. Serverless architectures with AWS SAM. https://medium.com/@christopheradamson253/serverless-architectures-with-aws-sam-serverless-application-model-2b83298fbcbc.
[16] 2024. Supplemental material. https://github.com/WenJinfeng/SlsDetector_ConfigurationDetection.
[17] 2024. Utah COVID-19 testing service. https://www.comparitech.com/blog/information-security/utah-covid-test-center-leak/.
[18] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.
[19] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.
[20] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful serverless computing with crucial. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–38.
[21] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 362–374.
[22] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the 19th*

European Conference on Computer Systems. 674–688.

[23] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402* (2023).

[24] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[25] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–38.

[26] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina Abad, and Alexandru Iosup. 2021. The state of serverless applications: collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* (2021).

[27] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.

[28] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 1469–1481.

[29] Yaoqi Guo, Zhenpeng Chen, Jie M. Zhang, Yang Liu, and Yun Ma. 2025. Personality-guided code generation using large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, ACL 2025*.

[30] Fatemeh Hadadi, Qinghua Xu, Domenico Bianculli, and Lionel Briand. 2024. Anomaly detection on unstable logs with GPT models. *arXiv preprint arXiv:2406.07467* (2024).

[31] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM SIGMOD Record* 29, 2 (2000), 1–12.

[32] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. 2021. Survey on serverless computing. *Journal of Cloud Computing* 10, 1 (2021), 1–29.

[33] Mia Mohammad Imran, Preetha Chatterjee, and Kostadin Damevski. 2024. Uncovering the causes of emotions in software developer communication using zero-shot llms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[34] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[35] David Kavaler, Sasha Sirovica, Vincent Hellendoorn, Raul Aranovich, and Vladimir Filkov. 2017. Perceived language complexity in GitHub issue discussions and their effect on issue resolution. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 72–83.

[36] Stratos Kourtzanidis, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. 2020. RepoSkillMiner: Identifying software expertise from GitHub repositories using natural language processing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1353–1357.

[37] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

[38] Valentina Lenarduzzi and Annibale Panichella. 2020. Serverless testing: Tool vendors' and experts' points of view. *IEEE Software* 38, 1 (2020), 54–60.

[39] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* (2023).

[40] Shanshan Li, Wang Li, Xiangke Liao, Shaoliang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. 2018. Confvd: System reactions analysis and evaluation through misconfiguration injection. *IEEE Transactions on Reliability* 67, 4 (2018), 1393–1405.

[41] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and opportunities: An in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 478–490.

[42] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. 2022. The serverless computing survey: A technical primer for design architecture. *Comput. Surveys* 54, 10s (2022), 220:1–220:34.

[43] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. 2023. Configuration validation with large language models. *arXiv preprint arXiv:2310.09690* (2023).

[44] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *CoRR* abs/2409.02977 (2024).

[45] Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. LLM-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics, ACL 2025*.

[46] Shangqing Liu, Yanzhou Li, Xiaofei Xie, Wei Ma, Guozhu Meng, and Yang Liu. 2024. Automated commit intelligence by pre-training. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–30.

[47] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, Balasubramanyan Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. 435–448.

[48] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[49] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2021. Learning software configuration spaces: A systematic literature review. *Journal of Systems and Software* 182 (2021), 111044.

[50] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–20.

[51] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic automated language learning for configuration files. In *Proceedings of the Computer Aided Verification: 28th International Conference*. Springer, 80–87.

[52] Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. 2024. Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 13–25.

[53] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 281–295.

[54] Norbert Siegmund, Nicolai Ruckel, and Janet Siegmund. 2020. Dimensions of software configuration: on the configuration context in modern software development. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 338–349.

[55] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 735–751.

[56] Davide Taibi, Josef Spillner, and Konrad Wawruch. 2020. Serverless computing-where are we now, and where are we heading? *IEEE software* 38, 1 (2020), 25–31.

[57] John Toman and Dan Grossman. 2016. Staccato: A bug finder for dynamic configuration updates. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[58] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *Proceedings of the 45th International Conference on Software Engineering*.

[59] Teng Wang, Xiaodong Liu, Shanshan Li, Xiangke Liao, Wang Li, and Qing Liao. 2018. MisconfDoctor: diagnosing misconfiguration via log-based configuration testing. In *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 1–12.

[60] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: a systematic review. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–61.

[61] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 416–428.

[62] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[63] Guangquan Xu, Xinru Ding, Sihan Xu, Yan Jia, Shaoying Liu, Shicheng Feng, and Xi Zheng. 2023. Real-time diagnosis of configuration errors for software of AI server infrastructure. *IEEE Transactions on Dependable and Secure Computing* (2023).

[64] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log parsing with prompt enhanced in-context learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[65] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 244–259.

[66] Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *Comput. Surveys* 47, 4 (2015), 1–41.

[67] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[68] Xin Yin, Chao Ni, and Shaohua Wang. 2024. Multitask-based evaluation of open-source llm on software vulnerability. *IEEE Transactions on Software Engineering* (2024).

[69] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proceedings of the 2021 IEEE 41st International Conference on Distributed Computing Systems*. IEEE, 138–148.

[70] Ding Yuan, Yinglian Xie, Rina Panigrahy, Junfeng Yang, Chad Verbowski, and Arunvijay Kumar. 2011. Context-based online configuration-error detection. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*. 28–28.

[71] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving ChatGPT for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[72] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–700.

[73] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. 2023. DRIVE: Dockerfile rule mining and violation detection. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–23.