EXPLORING LARGE LANGUAGE MODELS AS CONFIGURATION VALIDATORS: TECHNIQUES, CHALLENGES, AND OPPORTUNITIES

BY

XINYU LIAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Assistant Professor Tianyin Xu
Professor Darko Marinov

ABSTRACT

Misconfigurations are major causes of software failures. Existing practices rely on developer-written rules or test cases to validate configurations, which are expensive to implement and maintain, and are hard to be comprehensive. Machine learning (ML) for configuration validation is considered a promising direction, but has been facing challenges such as the need of large-scale field data and system-specific models, which are hard to generalize. Recent advances in Large Language Models (LLMs) show promise in addressing some of the long-lasting limitations of ML-based configuration validation.

This thesis presents a first analysis on the feasibility and effectiveness of using LLMs for configuration validation. We empirically evaluate LLMs as configuration validators by developing a generic LLM-based configuration validation framework, named Ciri. Ciri employs effective prompt engineering with few-shot learning based on both valid configuration and misconfiguration data. Ciri checks outputs from LLMs when producing results, addressing hallucination and nondeterminism of LLMs. We evaluate Ciri's validation effectiveness on eight popular LLMs using configuration data of ten widely deployed open-source systems.

Our analysis (1) confirms the potential of using LLMs for configuration validation, e.g., Ciri with Claude-3-Opus detects 45 out of 51 real-world misconfigurations, outperforming recent configuration validation techniques. (2) explores design space of LLM-based validators like Ciri, especially in terms of prompt engineering with few-shot learning and voting. We find that using configuration data as shots can enhance validation effectiveness. (3) reveals open challenges: Ciri struggles with certain types of misconfigurations such as dependency violations and version-specific misconfigurations. It is also biased to the popularity of configuration parameters, causing both false positives and false negatives.

We discuss the promising directions to address these challenges and further improve Ciri. Chain-of-Thoughts (CoT) can mimic the reasoning process of a human expert, which makes the validation more transparent and potentially more accurate. Additionally, LLMs can generate environment-specific scripts to run in the target environment, that can help identify issues like misconfigured paths, unreachable addresses, missing packages, and invalid permissions. We also plan to explore extending Ciri into a multi-agent framework, where Ciri can interact with additional tools such as Ctest and Cdep through agent frameworks.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deep gratitude to my advisor, Assistant Professor Tianyin Xu. My journey with Tianyin began during my undergraduate summer research program. At that time, I was quite new to the research and had no idea about what is the research. Despite my limited background, Tianyin consistently demonstrated patience, guiding me with great care, helping me to understand the essence of research, and instilling in me the confidence to pursue it. Over the course of three years, from 2021 to 2024, Tianyin excelled in his role as a professor, which by his saying, on helping every student to be successful. On research, Tianyin always gave me the freedom to choose the research topics I wanted to work on and pushed me to think critically about the problems at hand. One of the most valuable practices I've adopted, that learnt from Tianyin, is the habit of reflecting and jotting down my thoughts daily. I feel incredibly fortunate and grateful to have had Tianyin as my advisor, and I will always cherish the experience of working with him.

I would also like to extend my heartfelt gratitude to my co-advisor, Professor Darko Marinov. He is undoubtedly the most knowledgeable individual I've had the pleasure of encountering. I enjoy discussing with him a lot, as he always provides a deep and distinct perspective on any topics. During moments of decision-making, he has consistently been a pillar of support, offering invaluable suggestions that opened new perspectives for me. He cares students a lot and always tries to help them in any way he can, not limited to research. Moreover, it's rare to find a senior professor like Darko who commits such personal time to students, evident in the hours he spent with me debugging code. And one treasure he gave me is how to write bash scripts. I am grateful for his guidance and support throughout my graduate studies. The memories of sharing homemade cake and cookies, and enjoying his jokes, are cherished memories that I will always hold dear.

Futhermore, I would like to express my appreciation to Assistant Professor Minjia Zhang, with whom I had the pleasure of collaborating on several research projects. His invaluable help and insights greatly contributed to the success of my research.

I want to express my deepest gratitude to my parents, they always stand by me, offer invaluable advice on life and support every decision I make.

Finally, I would like to thank my fellow students, including Shuai Wang, Yinfang Chen, Sam Cheng, Parth Thakkar and so on. Exploring life at Urbana-Champaign with them and discussing research topics in the lab was a wonderful experience. I hope that everyone has a bright future and that we will have the opportunity to meet again in the future.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Modern software systems undertake hundreds to thousands of configuration changes on a daily basis [1, 2, 3, 4, 5, 6, 7]. For example, at Meta/Facebook, thousands of configuration file "diffs" are committed daily, outpacing the frequency of code changes [1, 2]. Other systems such as at Google and Microsoft also frequently deploy configuration changes [3, 5, 6]. Such velocity of configuration changes inevitably leads to misconfigurations. Today, misconfigurations are among the dominating causes of production incidents [2, 3, 7, 8, 9, 10, 11, 12, 13, 14]. For example, misconfiguration is the second largest root-cause category of service disruptions at a main Google production service [3], while Facebook/Meta reported that 16% of production incidents are triggered by configuration changes [1].

To detect misconfigurations, today's configuration management systems employ the "configuration-as-code" paradigm and enforce continuous configuration validation, ranging from static validation, to configuration testing, and to manual review and approval [1]. The configuration is first checked by validation code (aka *validators*) based on predefined correctness rules [1, 15, 16, 17, 18, 19, 20, 21]; in practice, validators are written by engineers [1, 15, 16]. After passing validators, configuration changes are then tested with code to check program behavior [22, 23]. Lastly, configuration changes are reviewed like source-code changes, where the change, commonly in the form of a configuration file "diff", is reviewed before production deployment.

The aforementioned pipeline either relies on manual inspection to spot misconfigurations in the configuration file diffs, or requires significant engineering efforts to implement and maintain validators or test cases. However, these efforts are known to be costly and incomprehensive. For example, despite that mature projects all include extensive configuration validators, recent work [24, 25, 26, 27, 28, 29, 30] repeatedly shows that existing validators are insufficient. The reasons are twofold. First, with large-scale systems exposing hundreds to thousands of configuration parameters [31], implementing validators for every parameter becomes a significant overhead. Recent studies [1, 24] report that many parameters are uncovered by existing validators, even in mature software projects. Second, it is non-trivial to validate a parameter, which could have many different correctness properties, such as type, range, semantic meaning, dependencies with other parameters, etc.; encoding all of them into validators is laborious and error-prone, not to mention the maintenance cost [32, 33].

Using machine learning (ML) and natural language processing (NLP) to detect misconfigurations has been considered a promising approach to addressing the above challenges. Compared to manually written static validators, ML or NLP-based approaches are automatic, easy to scale to a large number of parameters, and applicable to different projects. Several ML/NLP-based misconfiguration detection techniques were proposed [34, 35, 36, 37, 38, 39, 40, 41]. The key idea is to first learn correctness rules from field configuration data [34, 35, 36, 37, 39, 41, 42, 43, 44] or from documents [38, 40], and then use the learned rules to detect misconfigurations in new configuration files. ML/NLP-based approaches have achieved good success. For example, Microsoft adopted PeerPressure [36, 45] as a part of Microsoft Product Support Services (PSS) toolkits. It collects configuration data in Windows Registry from a large number of Windows users to learn statistical golden states of system configurations.

However, ML/NLP-based misconfiguration detection is also significantly limited. First, the need for large volumes of system-specific configuration data makes it hard to apply those techniques outside corporations that collect user configurations (e.g., Windows Registry [41]) or maintain a knowledge base [40]. For example, in cloud systems where the *same* set of configurations is maintained by a small DevOps team [1, 4], there is often no enough information for learning [25]. Moreover, prior ML/NLP-based detection techniques all target specific projects, and rely on predefined features [39], templates [35], or models [40]. As a result, it is hard to generalize them to different projects.

Recent advances on Large Language Models (LLMs), such as GPT [46] and Codex [47], show promises to address some of the long-lasting limitations of traditional ML/NLP-based misconfiguration detection techniques. Specifically, LLMs are trained on massive amounts of public data, including configuration data—configuration files in software repositories, configuration documents, knowledge-based articles, Q&A websites for resolving configuration issues, etc. Hence, LLMs encode extensive knowledge of both *common* and *project-specific* configuration. Such knowledge can be utilized for configuration validation without the need for manual rule engineering. Furthermore, LLMs show the capability of *generalization* and *reasoning* [48, 49] and can potentially "understand" configuration semantics. For example, they can not only understand that values of a port must be in the range of [0, 65535], but also reason that a specific configuration value represents a port (e.g., based on the name and description) and thus has to be within the range.

In fact, LLMs can be further improved with training data. Even in the case that fine-tuning is expensive, LLMs can be enhanced with few-shot learning [50, 51, 52]. As a result, one can use production configuration data to improve the effectiveness of LLM-based validation. Today's LLMs commonly expose free-form text interface. With effective prompts, LLMs can be instructed to output structured results that are easy to parse. One can input configuration snippets in any format (INI/XML/YAML) and get back structured validation results (e.g., a JSON object), facilitating easier parsing and application.

Certainly, LLMs have limitations. They are known for hallucination and non-determinism [53, 54]. Additionally, LLMs have limited input context, which can pose challenges when encoding extensive contexts like configuration file and related code. Moreover, they are reported to be biased to popular content in the training dataset. Fortunately, active efforts [55, 56, 57, 58, 59] are made to address these limitations.

1.2 CONTRIBUTION

In this thesis, we present a first analysis on the feasibility and effectiveness of using LLMs such as GPT and Claude for configuration validation. Our goal is to empirically evaluate the promises of using LLMs as effective configuration validators and to understand the challenges. As a first step, we empirically evaluate LLMs in the role of configuration validators, without additional fine-tuning or code generation. We focus on basic misconfigurations (those violating explicit correctness constraints) which are common misconfigurations encountered in the field [9]. We do not target environment-specific misconfigurations or bugs triggered by configuration. We discuss how to further build on this work to detect those in §6.

To do so, we develop Ciri, an LLM-empowered configuration validation framework. Ciri takes a configuration file or a file diff as the input; it outputs detected misconfigurations along with the reasons that explain them. Ciri integrates different LLMs such as GPT-4, Claude-3, and CodeLlama. Ciri devises effective prompt engineering with few-shot learning based on existing configuration data. Ciri also validates the outputs of LLMs to generate validation results, coping with the hallucination of LLMs. A key design principle of Ciri is separation of policy and mechanism. Ciri can serve as an open framework for experimenting with different models, prompt engineering, training datasets, and validation methods.

We study Ciri's validation effectiveness using eight popular LLMs including remote models (GPT-4, GPT-3.5, Claude-3-Opus, and Claude-3-Sonnet), and locally housed models (CodeLlama-7B/13B/34B and DeepSeek). We evaluate ten widely deployed open-source systems with diverse types. Our study confirms the potential of using LLMs for configuration validation, e.g., Ciri with Claude-3-Opus detects 45 out of 51 real-world misconfigurations, outperforming recent configuration validation techniques. Our study also helps understand the design space of LLM-based validators like Ciri, especially in terms of prompt engineering with few-shot learning and voting. We find that using configuration data as shots can enhance validation effectiveness. Specifically, few-shot learning using both valid configuration and misconfiguration data achieves the highest effectiveness. Our results also reveal open challenges: Ciri struggles with certain types of misconfigurations such as dependency violations and version-specific misconfigurations. It is also biased to the popularity of configuration parameters, causing both false positives and false negatives.

In summary, this thesis makes the following contributions:

- A new direction of configuration validation using pre-trained large language models (LLMs);
- Ciri, an LLM-empowered configuration validation framework and an open platform for configuration research;
- An empirical analysis on the effectiveness of LLM-based configuration validation, and its design space;
- We have released Ciri and other research artifacts at `https://github.com/ciri4conf/ciri`.

CHAPTER 2: EXPLORATORY EXAMPLES

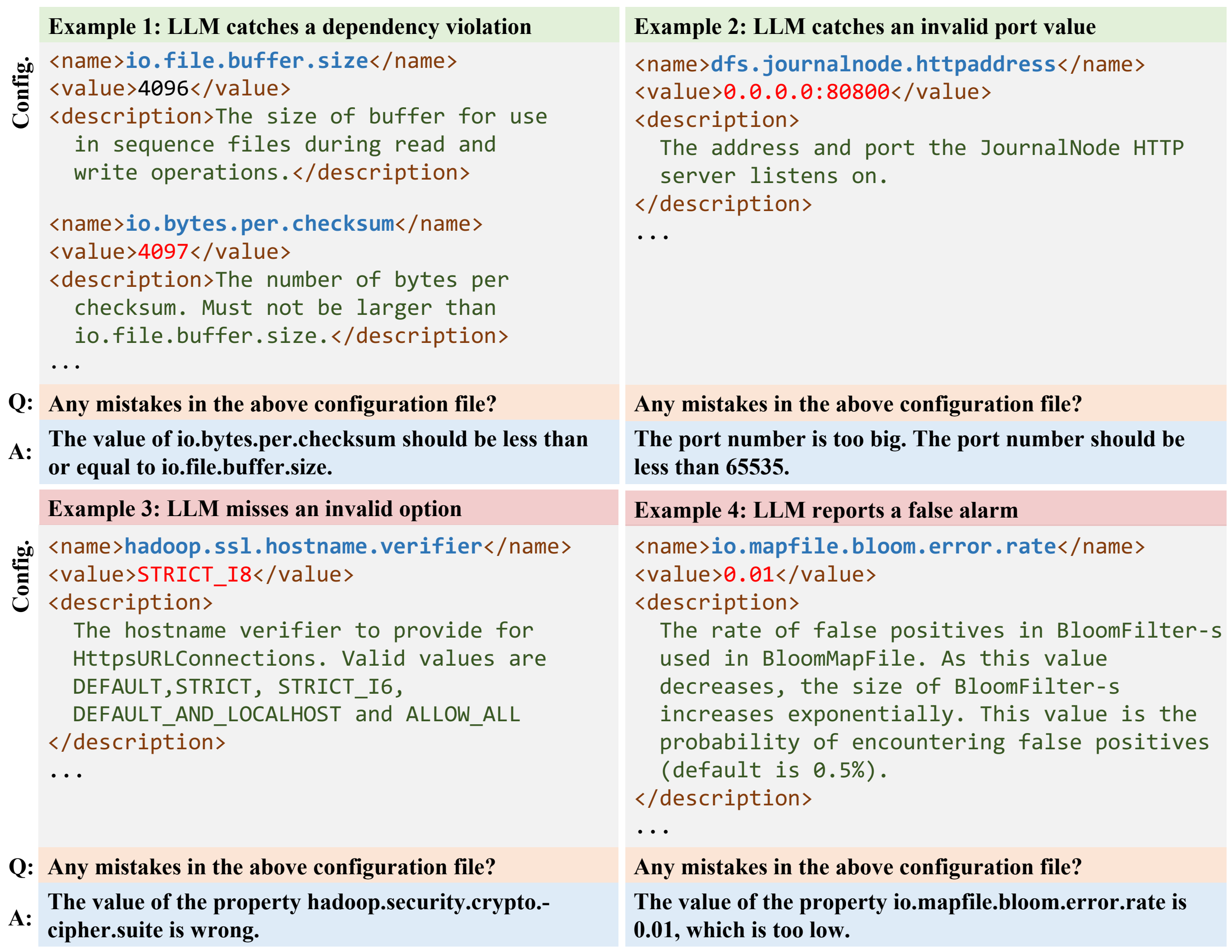
We explore using LLMs to validate configuration out of the box. We show that vanilla LLMs can detect misconfigurations. However, they are prone to both false negatives and false positives that require careful handling. Figure 2.1 presents four examples, two of which the LLM successfully detects misconfigurations, and two of which the LLM misses the misconfiguration or reports a false alarm. These examples were generated using the GPT-3.5-Turbo LLM [60].

Detecting violation of configuration dependency. Validating dependencies between configuration parameters has been a challenging task in highly-configurable systems [10, 61]. LLMs can infer relations between entities from text at the level of human experts [62], which allows LLMs to infer dependencies between parameters in a given configuration file based on their names and descriptions. Figure 2.1 (Example 1) presents a case where values of two dependent parameters were changed (i.e., "*buffer.size*" and "*bytes.per.checksum*)". After understanding the value relationship dependency between these two parameters, the model determines that the change in "*bytes.per.checksum*" has violated the enforced dependency, and provides the correct reason for the misconfiguration.

Detecting violation with domain knowledge. Written configuration validation rules often require significant manual efforts to produce and maintain. They are difficult to scale, due to the diverse types and functionalities of configuration parameters. A state-of-the-art LLM is trained on a massive amount of textual data and possesses basic knowledge across a wide range of professional domains. An LLM thus could be capable of understanding the definition of a configuration parameter and reasoning with its semantics. When the LLM encounters a configuration parameter such as IP address, permissions, and masks, it invokes the domain knowledge specific to the properties of those parameters. Figure 2.1 (Example 2) presents a case where an `HTTP` address has been misconfigured to a semantically invalid value. The model detects the misconfiguration, reasons that its value is out of range, and further suggests a potential fix.

Missed misconfiguration and false alarm. Despite that LLMs have demonstrated impressive performance across many tasks since its recent emergence, at the current stage of development, however, LLMs as configuration validators are not without errors. Examples 3 and 4 in Figure 2.1 show two cases where the LLM makes mistakes.

In Example 3, the configuration file has provided a description of the changed parameter "*hostname.verifier*" and explicitly listed the valid value options of the parameter. However, the model fails to realize that the parameter is misconfigured to an invalid, non-existent option



Config.

Example 1: LLM catches a dependency violation

```
<name>io.file.buffer.size</name>
<value>4096</value>
<description>The size of buffer for use
  in sequence files during read and
  write operations.</description>

<name>io.bytes.per.checksum</name>
<value>4097</value>
<description>The number of bytes per
  checksum. Must not be larger than
  io.file.buffer.size.</description>
...
```

Q: **Any mistakes in the above configuration file?**

A: **The value of io.bytes.per.checksum should be less than or equal to io.file.buffer.size.**

Example 2: LLM catches an invalid port value

```
<name>dfs.journalnode.httpaddress</name>
<value>0.0.0.0:80800</value>
<description>
  The address and port the JournalNode HTTP
  server listens on.
</description>
...
```

Q: **Any mistakes in the above configuration file?**

A: **The port number is too big. The port number should be less than 65535.**

Config.

Example 3: LLM misses an invalid option

```
<name>hadoop.ssl.hostname.verifier</name>
<value>STRICT_I8</value>
<description>
  The hostname verifier to provide for
  HttpsURLConnections. Valid values are
  DEFAULT,STRICT, STRICT_I6,
  DEFAULT_AND_LOCALHOST and ALLOW_ALL
</description>
...
```

Q: **Any mistakes in the above configuration file?**

A: **The value of the property hadoop.security.crypto.-cipher.suite is wrong.**

Example 4: LLM reports a false alarm

```
<name>io.mapfile.bloom.error.rate</name>
<value>0.01</value>
<description>
  The rate of false positives in BloomFilter-s
  used in BloomMapFile. As this value
  decreases, the size of BloomFilter-s
  increases exponentially. This value is the
  probability of encountering false positives
  (default is 0.5%).
</description>
...
```

Q: **Any mistakes in the above configuration file?**

A: **The value of the property io.mapfile.bloom.error.rate is 0.01, which is too low.**

Figure 2.1: Example 1 and 2 show the LLM correctly catches and reasons the misconfigurations. Example 3 and 4 show the LLM misses the misconfiguration or reports a valid configuration as erroneous.

(STRICT_I8). In Example 4, the description suggests that the parameter "*bloom.error.rate*" ranges from 0 to 100 (percentage), whereas the actual scale is 0 to 1 (fraction). This inconsistency supposedly confuses the model making it mark 0.01 (a valid value) as invalid.

Both examples show that directly using off-the-shelf LLMs as configuration validators would result in false negatives and false positives. The incorrect validation results can be attributed to hallucination [63]. A simple explanation is that LLMs are exposed to potentially contradictory data during training, which causes confusion to the model at the inference.

CHAPTER 3: CIRI: A LLM-EMPOWERED CONFIGURATION VALIDATION FRAMEWORK

We develop Ciri, an LLM-empowered configuration validation framework. Ciri takes a configuration file or a file diff as the input, and outputs a list of detected misconfigurations along with the reasons to explain the misconfigurations. Ciri supports different LLMs such as GPT, Claude, CodeLlama, and DeepSeek [63, 64].[1]

Figure 3.1 gives an overview of Ciri. Ciri turns a configuration validation request into a prompt to the LLMs (§3.1). The prompt includes (1) the target configuration file or diff, (2) a few examples (aka *shots*) to demonstrate the task of configuration validation, (3) code snippets automatically extracted from codebase, and (4) directive question and metadata. To generate shots, Ciri uses its database that contains labeled configuration data, including both valid configurations and misconfigurations. Ciri sends the same query to the LLMs multiple times and aggregates responses into the final validation result (§3.2).

Ciri applies to any software project, even if it has no labeled configuration data of that project in its database. regardless their file format or complexity. Ciri exhibits transferability (using data from one project and applying it to others), the ability to transfer configuration-related knowledge across projects when using configurations from different projects as shots (Finding 4). Ciri's configuration validation effectiveness can also be further improved by generating quality shots (Finding 3) and code snippets (Finding 5).

3.1 PROMPT ENGINEERING

3.1.1 Prompt structure

Ciri generates a prompt that includes four elements: a) the content of input configuration file or file diff, b) the shots as valid configurations or misconfigurations with questions and ground truth responses for few-shot learning, c) code snippets automatically extracted from available codebase, and d) a directive question for LLM to respond in formatted output. Figure 3.2 shows an illustrative example of the prompt generated by Ciri. It contains N shots, the content of to-be-validated configuration file, and the code snippet enclosed within ⟨Usage⟩ followed by the directive question.

Ciri phrases the prompting question as *"Are there any mistakes in the above configuration for [PROJECT] version [VERSION]? Respond in a JSON format similar to the following: ..."*. The [PROJECT] and [VERSION] are required inputs of Ciri because the validity of

[1]Adding a new LLM in Ciri takes a few lines to add the query APIs.

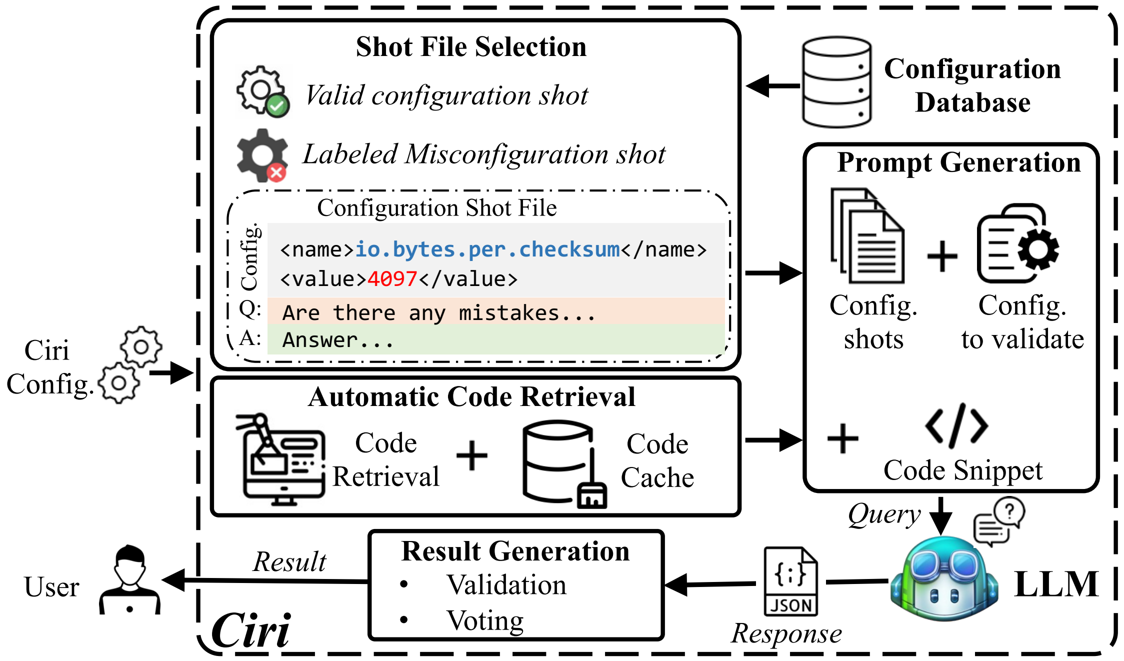
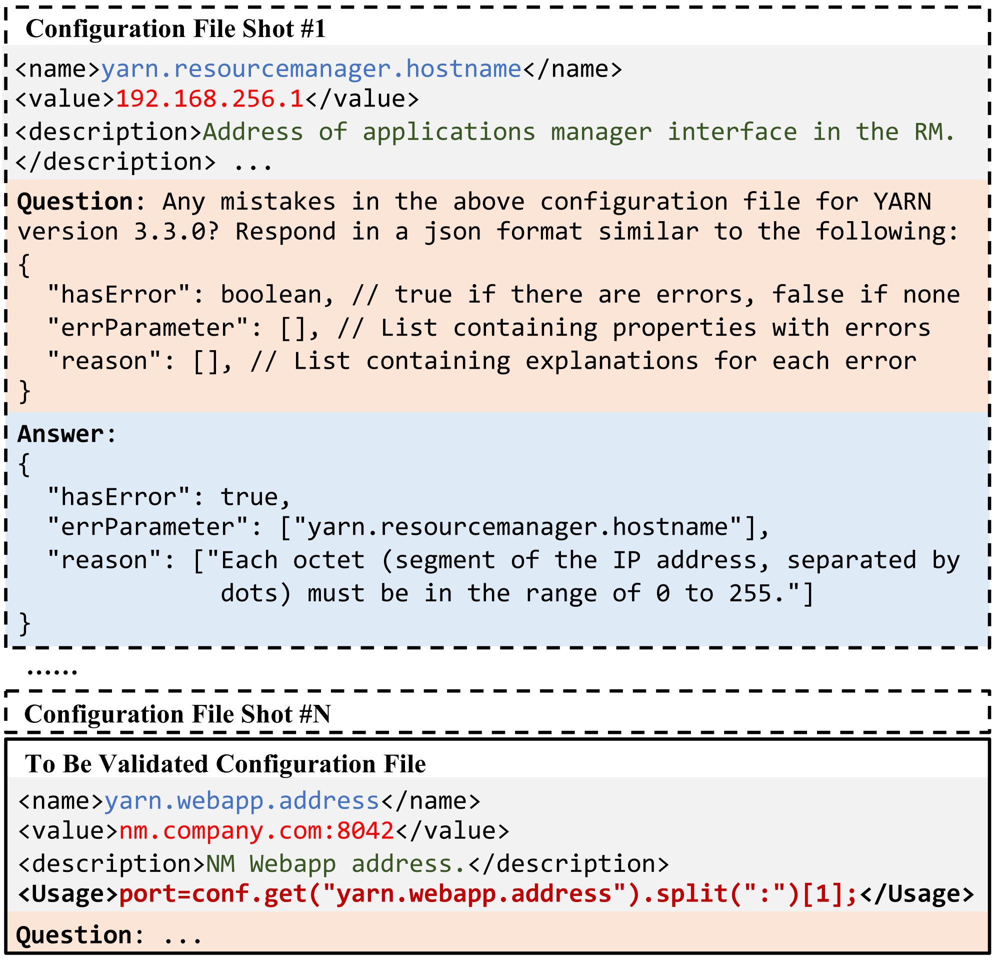


Figure 3.1: System overview of Ciri.

configuration can change by project and project version [32, 33]. A parameter's validity may change from one version to another due to shifts in usage, and parameters could be added or removed as system evolves. For instance, the number of parameters in Hadoop MapRedure module increases from 17 to 173 over seven years [31]. This prompt format enforces the LLM to respond in a unified JSON format for result aggregation (§3.2). However, responses from LLMs sometimes may still deviate from the anticipated format [53, 54]. In such cases, Ciri retries a new query to the LLM.

3.1.2 Few-shot learning

Ciri leverages the LLM's ability to learn from examples at inference time (aka few-shot learning) to improve configuration validation effectiveness. To do so, Ciri simply inserts shots at the beginning of each prompt. Each shot contains a configuration snippet, the prompting question, and its corresponding ground truth. Figure 3.2 shows an example, where there are N shots. "Configuration File Shot #1" is the first shot, in which the parameter "yarn.resourcemanager.hostname" is misconfigured. This shot also contains the prompting question (orange box) and the ground truth (blue box).


Configuration File Shot #1

```
<name>yarn.resourcemanager.hostname</name>
<value>192.168.256.1</value>
<description>Address of applications manager interface in the RM.
</description> ...
```

```
Question: Any mistakes in the above configuration file for YARN
version 3.3.0? Respond in a json format similar to the following:
{
  "hasError": boolean, // true if there are errors, false if none
  "errParameter": [], // List containing properties with errors
  "reason": [], // List containing explanations for each error
}
```

```
Answer:
{
  "hasError": true,
  "errParameter": ["yarn.resourcemanager.hostname"],
  "reason": ["Each octet (segment of the IP address, separated by
              dots) must be in the range of 0 to 255."]
}
```

......

Configuration File Shot #N

To Be Validated Configuration File

```
<name>yarn.webapp.address</name>
<value>nm.company.com:8042</value>
<description>NM Webapp address.</description>
<Usage>port=conf.get("yarn.webapp.address").split(":")[1];</Usage>
Question: ...
```

Figure 3.2: An example prompt generated by Ciri.

3.1.3 Shot generation

Ciri maintains a database of labeled valid configurations and misconfigurations for generating valid configuration shots (*ValidConfig*) and misconfiguration shots (*Misconfig*). A ValidConfig shot specifies a set of configuration parameters and their respective valid values. A valid value of a parameter can be its default value, or other valid values used in practice. On the other hand, a Misconfig shot specifies a set of parameters and their values, where only one of the parameter values is invalid. We provide more details on how to generate valid/invalid configuration values in §4.

For a given configuration of a specific project, Ciri by default generates shots using configuration data of the same project. If Ciri's database does not contain configuration data for the target project, Ciri will use data from other projects to generate shots. As shown in Finding 4, LLMs possess transferrable knowledge in configuration across different projects.

Ciri supports multiple methods for selecting data to generate shots, including randomized selection, category-based selection, and similarity-based selection (selecting data from configuration with the highest cosine similarity). We did not observe major differences when using different selection methods. So, Ciri uses randomized selection by default.

3.1.4 Augmenting with code

Recent work shows that retrieval-augmented generation (RAG) can enhance LLMs by incorporating additional information [65, 66]. In the context of configuration, each configuration parameter has corresponding program context, such as data type, semantics, and usage. In Figure 3.2, the code snippet enclosed within ⟨Usage⟩ is automatically extracted from the source code, which uncovers semantics that the parameter value is expected to include a ":" symbol, with the split segment representing a port.

Ciri uses a simple but effective code retrieval strategy: To retrieve the most effective code snippet, Ciri employs the following strategy: a) searching codebase with parameter names and retrieving relevant snippets; b) prioritizing code over comments and documents; c) selecting the longest snippet if multiple options are available, as longer snippets tend to be more comprehensive with details, and d) deduplicating retrieved snippets. The retrieved code snippet will be stored in a cache for efficiency. The code retrieval strategy is effective in improving the effectiveness of configuration validation (Finding 5).

3.1.5 Addressing token limits

LLMs limit input size per query by the number of input tokens. For example, the token limits for GPT-3.5-Turbo are 16,385. To navigate these constraints, Ciri compresses the prompt if its size exceeds the limit. Ciri first tries to put the target configuration and the directive question in the prompt, then maximizes three Misconfig shots with one ValidConfig shot (Finding 3) to fit into the remaining space. If the configuration cannot fit into the token limit, Ciri transforms it into a more compact format, e.g., transforming an XML file into INI format. If the compressed input still cannot fit, Ciri aborts and returns errors. In practice, configuration files and diffs are small [1, 31] and can easily fit existing limits. For example, prior study inspects configuration files collected from Docker, where each file contains 1 to 18 parameters, with eight on average [67]. For very large configurations, Ciri can split them into multiple snippets and validate them separately.

3.2 RESULT GENERATION

The JSON response from LLMs contains three primary fields: Each field has specific syntactic and semantic requirements as outlined below:

- a) "*hasError*": a boolean value indicating whether misconfigurations are detected,
- b) "*errParameter*": a list of misconfigured parameters,
- c) "*reason*": a list of explanations of the detected misconfiguration, corresponding to "*errParameter*".

3.2.1 Validation against hallucination

We employ a few rules to address the hallucination of LLMs. For example, if *hasError* is false, both *errParameter* and *reason* must be empty. Similarly, if *hasError* returns true, *errParameter* and *reason* must be non-empty with the same size. The answer to "*errParameter*" must not contain repeated values.

Given the potential for LLMs to deviate from the expected response format, we've designed a checker mechanism as a safeguard. This checker validates the results returned by the LLMs, flagging any responses that fail to meet the stipulated requirements. If the results do not pass the checker, we will discard them and query the LLM again.

3.2.2 Voting against inconsistency

LLMs can produce inconsistent outputs in conversation [68], explanation [69], and knowledge extraction [70]. To mitigate inconsistency, Ciri uses a multi-query strategy—querying the LLM multiple times using the same prompt and aggregating responses towards a result that is both representative of the model's understanding and more consistent than a single query. Ciri uses a frequency-based voting strategy: the output that recurs most often among the responses is selected as the final output [57].

Note that the "*reason*" field is not considered during voting due to the diverse nature of the response. After voting, Ciri collects reasons from all responses associated with the selected *errParameter*. The reason field is important as it provides users with insights into the misconfiguration, which is different from the traditional ML approaches that only provide a binary answer with a confidence score. Ciri clusters the reasons based on TF-IDF similarity [71], and picks a reason from the dominant cluster. We find that this mechanism is robust to hallucination— hallucinated reasons were often filtered out as they tended to be very different from each other.

3.3 CIRI CONFIGURATION

Ciri is customizable, with a key principle of separating policy and mechanism. Users can customize Ciri via its own configurations. Table 3.1 shows several important Ciri configurations and default values. The default values are chosen by pilot studies using a subset of our dataset (§4).

Table 3.1: configuration of Ciri and its default values.

Parameter	**Description**	**Default Value**
Model	Backend LLM. Also allows users to add other LLMs.	GPT-4
Temperature	Tradeoff between creativity and determinism.	0.2
# Shots	The number of shots included in a prompt.	Dynamic
# Queries	The number of queries with the same prompt.	3

CHAPTER 4: BENCHMARKS AND METRICS

Our study evaluates ten mature and widely deployed open-source projects: Alluxio, Django, Etcd, HBase, Hadoop Common, HDFS, PostgreSQL, Redis, YARN, ZooKeeper, which are implemented in a variety of programming languages (Java, Python, Go, and C). They also use different configuration formats (XML and INI) with a large number of configuration parameters. Table 4.2 lists the version (SHA) and the number of parameters at that version.

We evaluate Ciri on the aforementioned projects with eight LLMs: GPT-4-Turbo, GPT-3.5-Turbo, Claude-3-Opus, Claude-3-Sonnet, CodeLlama-7B/13B/34B, and DeepSeek-6.7B, which differ in model sizes and capabilities. All of these models have also been trained with a large amount of code data, where prior work has demonstrated their promising capability in handling a number of software engineering tasks [52, 72, 73].

4.1 CONFIGURATION DATASET

Our study uses two types of datasets: real-world misconfiguration datasets and synthesized misconfiguration datasets.

4.1.1 Real-world misconfiguration

To our knowledge, the Ctest dataset [67] is the only public dataset of real-world misconfigurations; it is used by prior configuration research [22, 34, 74, 75]. The dataset contains 64 real-world configuration-induced failures of five open-source projects, among which 51 are misconfigurations, and 13 are bugs. We discuss the results of Ciri on real-world misconfigurations in Finding 2.

4.1.2 Synthesized misconfiguration

Since real-world configuration dataset (§4.1.1) is too small, to systematically evaluate configuration validation effectiveness, we create new synthetic datasets for each evaluated project. First, we collect default configuration values from the default configuration file of each project, and real-world configuration files from the Ctest dataset (collected from Docker images [67, 76]) for those projects included in Ctest. We then generate misconfigurations of different types. The generation rules are from prior studies on misconfigurations [24, 26, 27, 28], which violates the constraints of configuration parameters (Table 4.1). Notably, prior studies show that the generation rules can cover 96.5% of 1,582 parameters across four projects [26].

Table 4.1: Misconfiguration generation (we use generation rules from prior work [22, 26, 27, 28], which reflects real-world misconfigurations). “Subcategory” lists rules to generate different misconfigurations for the same configuration parameter.

Category	**Subcategory**	**Specification** **Generation Rules**
Syntax	Data type	Value set = {Integer, Float, Long...} Generate a value that does not belong to the value set Numbers with units Generate an invalid unit (e.g., ”nounit”)
	Path	`^(\/[^\/ ]*)+\/?$` Generate a value that violates the pattern (e.g., `/hello//world`)
	URL	`[a-z]+://.*` Generate a value that violates the pattern (e.g., `file///`)
	IP address	`[\d]{1,3}(.[\d]{1,3}){3}` Generate a value that violates the pattern (e.g., `127.x0.0.1`)
	Port	Data type, value set = {Octet} Generate a value that does not belong to the value set
	Permission	Data type, value set = {Octet} Generate a value that does not belong to the value set
Range	Basic numeric	Valid Range constrainted by data type Generate values outside the valid range (e.g., Integer.MAX_VALUE+1)
	Bool	Options, value set = {true, false} Generate a value that does not belong to the value set
	Enum	Options, value set = {“enum1”, “enum2”, ...} Generate a value that doesn’t belong to set
	IP address	Range for each octet = [0, 255] Generate a value outside the valid range (e.g., `256.123.45.6`)
	Port	Range = [0, 65535] Generate a value outside the valid range
	Permission	Range = [000, 777] Generate a value outside the valid range
Dependency	Control	$(P_1, V, \Diamond) \mapsto P_2$, $\Diamond \in \{>, \geq, =, \neq, <, \leq\}$ Generate invalid control condition $(P_1, V, \neg\Diamond)$
	Value Relationship	$(P_1, P_2, \Diamond)$, $\Diamond \in \{>, \geq, =, \neq, <, \leq\}$ Generate invalid value relationship $(P_1, P_2, \neg\Diamond)$
Version	Parameter change	$(V_1, Pset_1) \mapsto (V_2, Pset_2)$, $Pset_1 \neq Pset_2$ Generate a removed parameter in V_2 or use an added paraemter in V_1

For each project, we build two distinct configuration sets. First, we build a configuration dataset with no misconfiguration (denoted as ValidConfig) to measure true negatives and false positives (Table 4.3). We also build a configuration dataset (denoted as Misconfig) in which each configuration file has one misconfiguration, to measure true positives and false negatives (Table 4.3). Note that a misconfiguration can be a dependency violation between multiple parameter values.

Table 4.2: Evaluated projects and the configuration datasets (ValidConfig and Misconfig) for shot pool and evaluation.

Project	**Version (SHA)**	**# Params**	**ValidConfig**		**Misconfig**	
			# Shot	**# Eval**	**# Shot**	**# Eval**
Alluxio	76569bc	494	13	54	13	54
Django	67d0c46	140	6	18	6	18
Etcd	946a5a6	41	8	32	8	32
HBase	0fc18a9	221	12	50	12	50
HCommon	aa96f18	395	16	64	16	64
HDFS	aa96f18	566	16	64	16	64
PostgreSQL	29be998	315	8	31	8	31
Redis	d375595	94	12	44	12	44
YARN	aa96f18	525	10	40	10	40
ZooKeeper	e3704b3	32	8	32	8	32

To create the Misconfig data for each project, we first check if its configuration parameters fit any subcategory in Table 4.1, and, if so, we apply rules from all matched subcategories to generate misconfigurations for that parameter. For example, an IP-address parameter fits both "Syntax: IP Address" and "Range: IP Address". We do so for all parameters in the project. Then, we randomly sample at most five parameters in each subcategory that has matched parameters, and generate invalid value(s) per sampled parameter. For each subcategory, we further randomly select one parameter from the five sampled ones. We use the selected parameter to create a faulty configuration as a Misconfig shot (§**??**) for that subcategory and add it to the project's shot pool. For the other four parameters, we use them to create four faulty configurations for that subcategory, and use them for evaluation. If a subcategory does not have enough parameters for sampling, we use all the parameters for evaluation. We separate the evaluation set and shot pool to follow the practice that the training set does not overlap with the testing set [62]. We create the ValidConfig dataset for each project using the aforementioned methodology for the Misconfig dataset, except that we generate valid values.

Table 4.2 shows the size for both the ValidConfig and Misconfig datasets for each project. Note that our datasets cover 72%–100% of the entire parameter set of each project.

4.2 METRICS

We evaluate Ciri's effectiveness at both configuration *file* and *parameter* levels: (1) at the file level, we check if Ciri can determine if a configuration file contains misconfigurations; (2) at the parameter level, we check if Ciri can determine if each parameter in the configuration file is valid or not. Table 4.3 describes our confusion matrix. We compute the precision *(TP/(TP+FP))*, recall *(TP/(TP+FN))*, and F1-score at both file and parameter levels. If not specified, we default to macro averaging since each project is regarded equally. We prioritize parameter-level effectiveness for fine-grained measurements and discuss parameter-level metrics by default in the evaluation.

Table 4.3: Definitions for confusion matrix.

Level	**Metric**	**Definition**
File	TP	A misconfigured file correctly identified
	FP	A correct file wrongly flagged as misconfigured
	TN	A correct file rightly identified as valid
	FN	A misconfigured file overlooked or deemed correct
Param.	TP	A misconfigured parameter correctly identified
	FP	A correct parameter wrongly flagged as misconfigured
	TN	A correct parameter rightly identified as valid
	FN	A misconfigured parameter overlooked or deemed correct

CHAPTER 5: EVALUATION AND FINDINGS

We present empirical results on the effectiveness of LLMs as configuration validators with Ciri (§5.1). We analyze how validation effectiveness changes with regard to design choices of Ciri (§5.2). We also present our understanding of when Ciri produces wrongful results (§5.3) and biases (§5.4).

5.1 EFFECTIVENESS OF CONFIGURATION VALIDATION

Finding 1. *Ciri shows effectiveness of using state-of-the-art LLMs as configuration validators. It achieves file- and parameter-level F1-scores up to 0.79 and 0.65, respectively.*

Ciri exhibits remarkable capability in configuration validation. Table 5.1 shows the F1-score, precision, and recall for each project using LLMs with three Misconfig and one ValidConfig shots (Finding 3). The results show that Ciri not only can effectively identify configuration files with misconfiguration (with an average F1-score of 0.72 across 8 LLMs), but also pinpoint misconfigured parameters with explanations (with an average F1-score of 0.56 across 8 LLMs). To better understand the results, we also compare with a basic "idle" model that employs a stochastic method for classifying files as either correct or incorrect, and randomly selects one parameter in each incorrect file as misconfigured. It achieves an average F1-score of 0.02 at the parameter level, which highlights the effectiveness of Ciri. Certainly, the parameter-level F1-scores are about 15% lower than file-level F1-scores, i.e., pinpointing fine-grained misconfigured parameters is a more challenging task for LLMs compared to classifying the entire file as a whole.

Finding 2. *Ciri detects 45 out of 51 real-world misconfigurations, outperforming recent configuration validation techniques, including learning-based [34] and configuration testing [22].*

We conduct experiments to evaluate how Ciri compares with existing validation techniques on a real-world dataset. For this evaluation, we choose the top five LLMs ranked by F1-score at the parameter-level based on the results from Table 5.1. The real-world dataset [67] contains 51 misconfigurations in total (§4), among which Ciri can detect 33-45 misconfigurations, as shown in Table 5.2. Ciri successfully detected 45 using Claude-3-Opus. The six undetected misconfigurations include three due to parameter dependency violations (discussed further in §5.3), and the other three are environment-related issues that are beyond Ciri's current capability. Notably, Ciri seldom reports incorrect detection.

Table 5.1: F1-score, precision, and recall of Ciri evaluated on ten projects with eight LLMs as configuration validators.

Models	**F1-score**											**Precision**	**Recall**
	AL.	DJ.	ET.	HB.	HC.	HD.	PO.	RD.	YA.	ZK.	**Avg**	**Avg**	**Avg**
File-Level													
GPT-4-Turbo	0.69	0.86	0.67	0.73	0.75	0.70	0.72	0.75	0.74	0.70	0.73	0.62	0.89
GPT-3.5-Turbo	0.68	0.71	0.73	0.77	0.78	0.68	0.65	0.71	0.72	0.74	0.72	0.62	0.89
Claude-3-Opus	0.71	0.81	0.70	0.70	0.79	0.74	0.75	0.82	0.77	0.78	0.76	0.65	0.91
Claude-3-Sonnet	0.74	0.77	0.79	0.80	0.81	0.76	0.82	0.84	0.79	0.78	0.79	0.75	0.85
CodeLlama-34B	0.69	0.87	0.80	0.64	0.70	0.67	0.79	0.78	0.66	0.83	0.74	0.65	0.89
CodeLlama-13B	0.71	0.67	0.67	0.71	0.66	0.70	0.71	0.76	0.69	0.77	0.70	0.61	0.85
CodeLlama-7B	0.67	0.80	0.74	0.67	0.67	0.67	0.63	0.73	0.67	0.76	0.70	0.56	0.96
DeepSeek-6.7B	0.72	0.72	0.81	0.52	0.47	0.46	0.70	0.67	0.59	0.84	0.65	0.76	0.66
Parameter-Level													
GPT-4-Turbo	0.52	0.82	0.59	0.49	0.51	0.53	0.43	0.57	0.62	0.53	0.56	0.43	0.81
GPT-3.5-Turbo	0.48	0.55	0.60	0.55	0.58	0.55	0.36	0.54	0.61	0.66	0.55	0.45	0.77
Claude-3-Opus	0.51	0.62	0.53	0.54	0.65	0.60	0.53	0.62	0.60	0.60	0.58	0.57	0.83
Claude-3-Sonnet	0.53	0.65	0.69	0.73	0.69	0.73	0.53	0.64	0.71	0.59	0.65	0.57	0.79
CodeLlama-34B	0.61	0.85	0.76	0.35	0.45	0.35	0.59	0.65	0.46	0.79	0.59	0.51	0.70
CodeLlama-13B	0.54	0.59	0.61	0.48	0.37	0.37	0.50	0.69	0.51	0.73	0.54	0.45	0.68
CodeLlama-7B	0.53	0.67	0.66	0.27	0.28	0.23	0.51	0.68	0.43	0.72	0.50	0.40	0.67
DeepSeek-6.7B	0.58	0.56	0.75	0.48	0.40	0.37	0.44	0.44	0.55	0.73	0.53	0.60	0.55

Since configuration files are typically considered confidential operation data in corporations, we didn't find any released models or dataset from prior papers that use traditional NLP/ML for configuration validation except one work (ConfMiner) that use file contents and histories of commits to learn patterns in configuration to detect misconfigurations [34]. We compare Ciri's results with ConfMiner [34], which was evaluated on the same dataset. ConfMiner utilizes the file content and commit history to identify patterns in configuration to detect misconfigurations. ConfMiner can detect 27 out of 51 misconfigurations, which is 40% less than Ciri. Unlike LLMs that are trained on extensive text data and can comprehend the context of configurations, ConfMiner relies on regular expressions to identify patterns. This approach limits its ability in complex scenarios, such as identifying valid values for enumeration parameters and understanding the relationships between different parameters.

We also compare Ciri with a recent configuration testing technique, namely Ctest [22, 74, 75]. Ctest detected 41 of the real-world misconfigurations without rewriting test code; Ciri outperforms Ctest by 8.9%. The reasons are twofold. First, testing relies on adequacy of the test cases. We find that existing test suites do not always have a high coverage of

Table 5.2: A comparison of Ciri, ConfMiner, and Ctest in detecting real-world misconfigurations. N.R.: "not reported."

Technique	**# Correct Detection**	**# Incorrect Detection**	**# Missed**	**Runtime**
Ciri (Claude-3-Opus)	45 (88.2%)	1	5	20-60 sec
Ciri (GPT-4-Turbo)	41 (80.4%)	2	8	15-40 sec
Ciri (CodeLlama-34B)	39 (76.5%)	1	11	30-70 sec
Ciri (GPT-3-Turbo)	37 (72.5%)	3	11	10-25 sec
Ciri (Claude-3-Sonnet)	33 (67.7%)	1	17	10-30 sec
ConfMiner	27 (52.3%)	N.R.	N.R.	N.R.
Ctest	41 (80.4%)	N.R.	N.R.	20-230 min

configuration parameters. On the other hand, LLMs can validate any parameter. Second, LLMs detected "silent misconfigurations" [9] that are not manifested via crashes or captured by assertions (e.g., several injected misconfigurations silently fell back to default values and passed the test; LLMs detected them likely because they violated documented specifications).

Certainly, Ctest can detect a broader range of misconfigurations such as the environment-related issues that Ciri cannot. We do not intend to replace configuration testing with LLMs. Instead, our work shows that LLMs can provide much quicker feedback for common types of misconfigurations, so tools like Ciri can be used in an early phase (e.g., configuration authoring) before running expensive configuration testing. As shown in Table 5.2, for a configuration file in the real-world dataset, Ctest takes 20 to 230 minutes to finish [22], while Ciri only takes 10 to 70 seconds.

In summary, our results show that LLMs like GPT, Claude-3, and CodeLlama-34B can effectively validate configurations and detect misconfigurations with a sensibly designed framework like Ciri. Ciri can provide prompt feedback, complementing other techniques like configuration testing.

5.2 IMPACTS OF DESIGN CHOICES

Ciri plays a critical role in LLMs' effectiveness of configuration validation. We explore its design choices and impacts.

Finding 3. *Using configuration data as shots can effectively improve LLMs' effectiveness of configuration validation. Shots including both valid configuration and misconfiguration achieve the highest effectiveness.*

Using validation examples as shots can effectively improve the effectiveness of LLMs. Table 5.3 shows the results of LLMs when the validation query does not include shots. In particular, comparing Table 5.3 to Table 5.1, as indicated by the numbered arrows in Table 5.3, the average F1-score of the LLMs has decreased by 0.03–0.54 at the file level, and decreased by 0.21–0.47 at the parameter level.

We also study Ciri's effectiveness with different shot combinations. We evaluate six N-shot learning settings, where N ranges from 0 to 5. For example, to evaluate Ciri with a two-shot setting, three experiments will be performed: (1) two ValidConfig shots; (2) one ValidConfig shot plus one Misconfig shot; (3) two Misconfig shots. In total, we experiment with 21 shot combinations. Due to cost, we only run experiments on GPT-3.5-Turbo to HCommon. We find that only using ValidConfig shots leads to a decrease in precision, while only using Misconfig shots reduces recall. Clearly, text distribution in the query affects LLMs [77]. LLMs can be biased: if the shots are all misconfigurations, LLMs will be overly sensitive to the specific patterns in the shots, known as overfitting, which causes LLMs miss other types of misconfigurations; if the shots are all ValidConfig, LLMs face challenges in accurately identifying incorrect parameters within the file, leading to false alarms. As shown in Figure 5.1, using both Misconfig and ValidConfig in few-shot learning mitigates the biases and achieves the highest effectiveness, and including three Misconfig shots and one ValidConfig shot in the prompt achieves the highest F1-score at both the file and parameter levels.

Table 5.3: Effectiveness of LLMs without using shots.

Models	**F1-score**		**Precision**		**Recall**	
	F.L.	**P.L.**	**F.L.**	**P.L.**	**F.L.**	**P.L.**
GPT-4-Turbo	0.70 (0.03↓)	0.34 (0.22↓)	0.57	0.23	0.93	0.82
GPT-3.5-Turbo	0.67 (0.05↓)	0.20 (0.35↓)	0.50	0.12	0.99	0.77
Claude-3-Opus	0.69 (0.07↓)	0.37 (0.21↓)	0.64	0.28	0.82	0.69
Claude-3-Sonnet	0.67 (0.12↓)	0.28 (0.37↓)	0.55	0.20	0.89	0.66
CodeLlama-34B	0.66 (0.08↓)	0.12 (0.47↓)	0.50	0.07	0.96	0.52
CodeLlama-13B	0.59 (0.11↓)	0.12 (0.42↓)	0.53	0.07	0.76	0.52
CodeLlama-7B	0.65 (0.05↓)	0.11 (0.39↓)	0.51	0.08	0.91	0.23
DeepSeek-6.7B	0.11 (0.54↓)	0.06 (0.47↓)	0.99	0.50	0.06	0.04

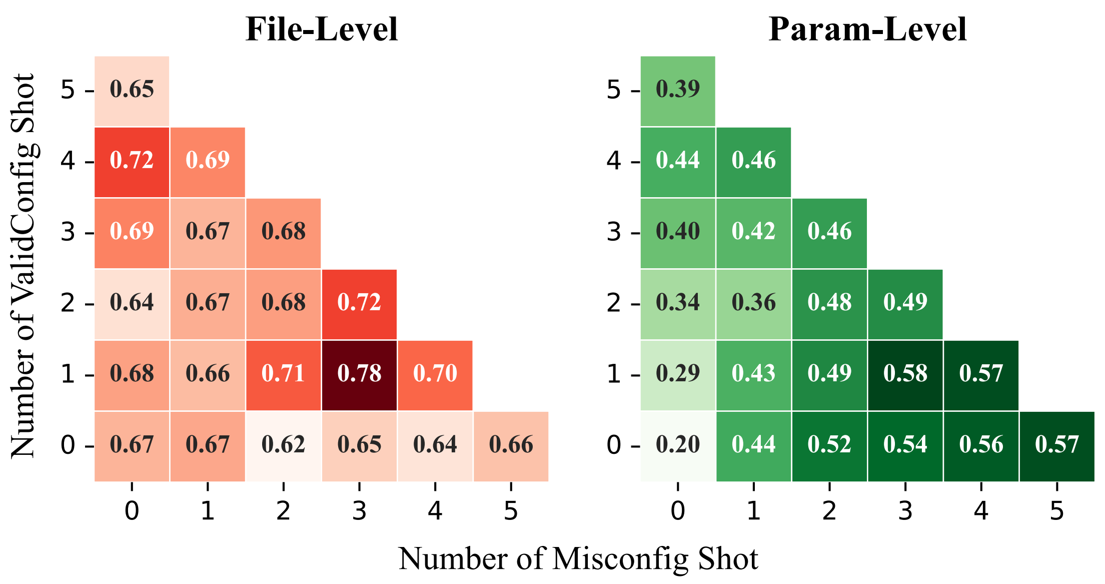


Figure 5.1: F1 scores under different shot combinations.

Finding 4. *Using configuration data from the same project as shots often leads to high validation F1 score. However, even without access to configuration data from the target project, using configuration data from a different project can lead to a improved validation score than zero-shot.*

The configuration file can often include sensitive or proprietary data, like IP addresses, passwords, or various security details [78, 79], typically necessitating its confidentiality and restricted access. Furthermore, user manuals or software documentation might be outdated or imprecise [31, 80]. In situations where configuration data is unavailable, we evaluate whether using configuration data from other systems as shots can improve configuration validation effectiveness on the target system.

Table 5.4 shows the results of using data from other projects as shots for configuration validation on HCommon. By comparing 4-shot HCommon with other columns in Table 5.4, we see that using shots from other projects is not as effective as using shots from the target system. However, the average F1-score is still higher than zero-shot, indicating that using shots from other projects can improve the effectiveness over zero-shot. Our observations highlight that Ciri with LLMs can transfer configuration-related knowledge across different projects for effective configuration validation compared to traditional approaches.

Finding 5. *Ciri's code augmentation approach can help LLMs to better understand the context of the configuration and improve the validation effectiveness reflected in increased F1-scores at both the file- and parameter levels by 0.03.*

We compare the results of GPT-3.5-Turbo with and without code augmentation with four shots. The results show an improvement in F1 scores by 0.03 at both the file and parameter levels. Figure 5.2 exemplifies code snippets retrieved from the codebase by Ciri, which could improve LLMs' comprehension of the configuration context: (1) examples 1 and 2 delineate the parameter types as Integer and Boolean, respectively. (2) example 3 highlights that the parameter should include a ":" symbol, with the latter segment representing a port. (3) example 4 shows that "kerberos" is one valid value for the parameter.

```
Ex1: conf.setInt("tfile.fs.input.buffer.size", fsInputBufferSize);
Ex2: conf.setBoolean("fs.automatic.close", false);
Ex3: port=conf.get("yarn.nodemanager.webapp.address").split(":")[1];
Ex4: "kerberos".equals(conf.get("hbase.security.authentication"));
```

Figure 5.2: Code snippets retrieved by Ciri to aid LLMs.

Finding 6. *Code-specialized LLMs, e.g., CodeLlama, exhibit much higher validation scores than generic LLMs, e.g., Llama-2. Moreover, further scaling up the code-specialized LLMs leads to a continuous increase in validation scores.*

In Table 5.1, we observe a notable trend within the CodeLlama model family: the 13B model demonstrates an improvement in F1-score at the parameter level, by 0.04 over the 7B model; this trend continues with the 34B model, which exhibits a further 0.05 enhancement in F1-score over the 13B model. The observed performance gains can be primarily attributed to the increased capacity for learning and representing complex semantics of configuration values as model size scales. This involves deep comprehension beyond syntax and range violations which are more common in practice [9].

We further evaluated the effectiveness of the Llama-2 model, which is identical to CodeLlama in structure but lacks code-specific training. The Llama-2-13B is not effective, with an average

Table 5.4: F-1 score on HCommon (HC.) using shots from different systems, e.g., HB. refers to using HBase shots. 4-S and 0-S means using four shots and no shots respectively.

Models	**File-Level (F.L.)**						**Parameter-Level (P.L.)**					
	HC.		Dj.	ET.	HB.	**Avg**	HC.		Dj.	ET.	HB.	**Avg**
	4-S	0-S					4-S	0-S				
GPT-3.5-Turbo	0.78	0.67	0.78	0.68	0.74	0.74	0.58	0.20	0.44	0.42	0.51	0.46
Claude-3-Sonnet	0.81	0.67	0.74	0.74	0.78	0.75	0.69	0.28	0.54	0.59	0.67	0.60

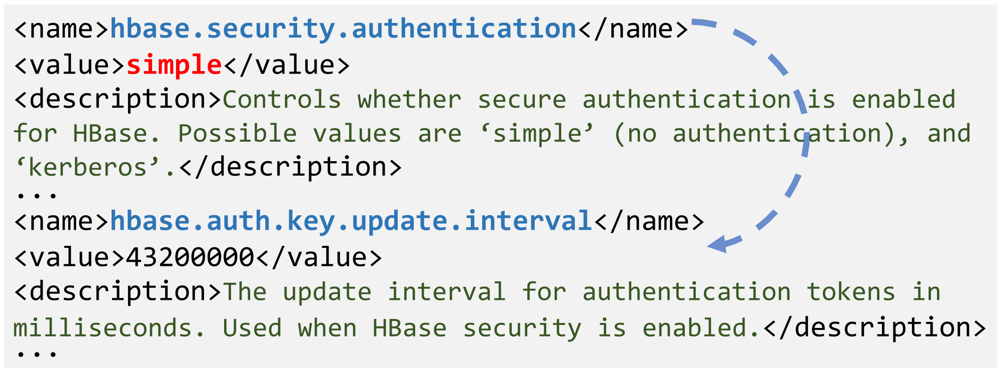
F1-score of 0.05 at the parameter level. This result underscores the role of code-specific training, which enhances LLM's comprehension of configuration in the context of code.

5.3 LIMITATIONS AND CHALLENGES

Finding 7. *With Ciri, LLMs excel at detecting misconfigurations of syntax and range violations with an average F1-score of 0.8 across subcategories. However, LLMs are limited in detecting misconfigurations of dependency and version violations (i.e., 3 out of all 15 sub-categories), with an average F1-score of 0.3 across subcategories.*

Table 5.5 shows Ciri's validation effectiveness per misconfiguration type. The F1-score on detecting misconfigurations of syntax and range violations is consistently above 0.5 across projects, and often reaches 0.8. However, F1-score rarely exceeds 0.5 on misconfigurations of dependency and version violations. Under these two categories, LLMs achieve F1-scores of 0.44–0.54 for misconfigurations that violate value relationship, which is higher than the other two subcategories (control and parameter change); however, it is still much lower than others.

The difference can be attributed to the inherent nature of different types of misconfigurations. Misconfigurations of syntax and range violations are more common in practice [9], from which LLMs learned extensive knowledge. In such a case, domain-specific knowledge from LLMs is sufficient to detect these misconfigurations. But, misconfiguration of dependency and version violations is often project-specific, as exemplified in Figure 5.3. They are tied to detailed history and features of the project, and thus hard to be captured or memorized by LLMs if the LLM is not fine-tuned on project-specific data. This discrepancy between misconfiguration types exposes existing LLM's limitation.



```
<name>hbase.security.authentication</name>
<value>simple</value>
<description>Controls whether secure authentication is enabled
for HBase. Possible values are ‘simple’ (no authentication), and
‘kerberos’.</description>
...
<name>hbase.auth.key.update.interval</name>
<value>43200000</value>
<description>The update interval for authentication tokens in
milliseconds. Used when HBase security is enabled.</description>
...
```

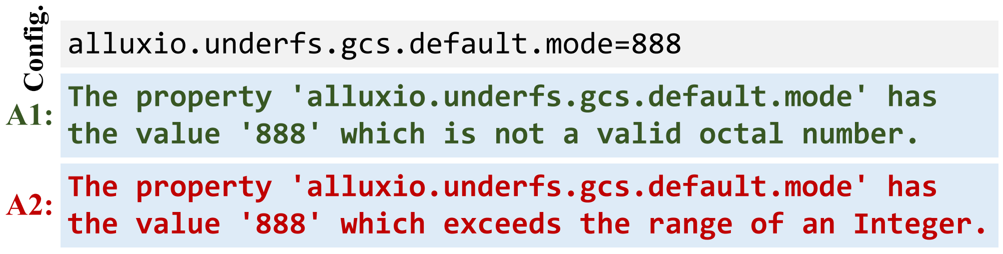
Figure 5.3: Misconfiguration that violates control dependency that LLMs cannot detect. The update interval for authentication is set but the secure authentication is disabled.

Table 5.5: Parameter-level F1-score by misconfiguration types from Table 4.1. N.A. means no evaluation samples. Color Mapping: GPT-4-Turbo, Claude-3-Opus, CodeLlama-34B

Category	Sub-category	AL.	DJ.	ET.	HB.	HC.	HD.	PO.	RD.	YA.	ZK.	Avg
Syntax	Data Type	0.84	1.00	1.00	0.67	0.94	0.89	0.70	0.78	0.89	0.80	**0.85**
		0.67	0.80	1.00	0.80	0.94	1.00	0.80	0.74	1.00	0.80	**0.85**
		0.93	1.00	1.00	0.25	0.67	0.80	1.00	1.00	0.50	1.00	**0.82**
	Path	0.50	1.00	1.00	0.89	0.73	0.73	0.57	0.57	1.00	0.73	**0.77**
		1.00	1.00	0.80	0.89	0.89	1.00	1.00	0.67	0.89	0.73	**0.89**
		1.00	1.00	1.00	0.86	0.50	0.00	1.00	0.57	1.00	1.00	**0.79**
	URL	0.67	0.80	1.00	N.A.	0.80	0.80	N.A.	N.A.	N.A.	N.A.	**0.81**
		1.00	0.67	0.73	N.A.	1.00	0.89	N.A.	N.A.	N.A.	N.A.	**0.86**
		1.00	1.00	0.75	N.A.	1.00	0.60	N.A.	N.A.	N.A.	N.A.	**0.87**
	IP Address	0.70	N.A.	N.A.	0.73	0.94	0.89	N.A.	0.84	0.94	0.76	**0.83**
		0.73	N.A.	N.A.	0.84	0.94	0.84	N.A.	0.80	0.94	0.84	**0.85**
		1.00	N.A.	N.A.	0.82	0.75	0.75	N.A.	1.00	1.00	1.00	**0.90**
	Port	0.74	N.A.	N.A.	0.78	0.94	0.82	N.A.	0.94	N.A.	0.84	**0.84**
		0.70	N.A.	N.A.	0.82	0.82	0.67	N.A.	0.84	N.A.	0.94	**0.80**
		0.94	N.A.	N.A.	0.71	1.00	0.62	N.A.	0.75	N.A.	1.00	**0.84**
	Permission	0.89	N.A.	N.A.	0.80	0.78	1.00	N.A.	N.A.	N.A.	N.A.	**0.87**
		0.89	N.A.	N.A.	0.80	0.82	1.00	N.A.	N.A.	N.A.	N.A.	**0.88**
		0.86	N.A.	N.A.	0.50	0.50	0.40	N.A.	N.A.	N.A.	N.A.	**0.56**
Range	Basic Numeric	0.73	1.00	0.75	0.73	0.60	0.67	1.00	0.89	1.00	0.80	**0.82**
		0.67	0.80	0.89	0.46	0.50	0.67	1.00	0.67	0.89	0.80	**0.73**
		0.75	1.00	0.89	0.36	0.00	0.50	0.67	0.75	0.50	1.00	**0.64**
	Bool	1.00	0.75	1.00	0.80	1.00	1.00	N.A.	0.89	1.00	0.80	**0.92**
		1.00	0.55	0.73	0.67	0.89	0.89	N.A.	0.73	1.00	0.80	**0.80**
		1.00	0.86	1.00	0.00	0.67	0.00	N.A.	0.67	0.50	0.75	**0.60**
	Enum	0.36	N.A.	0.86	0.73	0.67	0.89	0.89	0.75	0.80	N.A.	**0.74**
		0.36	N.A.	0.89	1.00	0.86	0.75	0.89	0.89	0.80	N.A.	**0.80**
		0.86	N.A.	1.00	0.75	0.57	0.60	0.75	1.00	0.75	N.A.	**0.78**
	IP Address	0.70	N.A.	N.A.	0.73	0.94	0.89	N.A.	0.84	0.94	0.76	**0.83**
		0.73	N.A.	N.A.	0.84	0.94	0.84	N.A.	0.80	0.94	0.84	**0.85**
		1.00	N.A.	N.A.	0.82	0.75	0.75	N.A.	1.00	1.00	1.00	**0.90**
	Port	0.74	N.A.	N.A.	0.78	0.94	0.82	N.A.	0.94	N.A.	0.84	**0.84**
		0.70	N.A.	N.A.	0.82	0.82	0.67	N.A.	0.84	N.A.	0.94	**0.80**
		0.94	N.A.	N.A.	0.71	1.00	0.62	N.A.	0.75	N.A.	1.00	**0.84**
	Permission	0.89	N.A.	N.A.	0.80	0.78	1.00	N.A.	N.A.	N.A.	N.A.	**0.87**
		0.89	N.A.	N.A.	0.80	0.82	1.00	N.A.	N.A.	N.A.	N.A.	**0.88**
		0.86	N.A.	N.A.	0.50	0.50	0.40	N.A.	N.A.	N.A.	N.A.	**0.56**
Dependency	Control	0.50	N.A.	0.00	0.00	0.00	0.25	0.00	0.00	0.00	N.A.	**0.09**
		0.00	N.A.	0.29	0.00	0.00	0.00	0.00	0.00	0.00	N.A.	**0.04**
		0.40	N.A.	0.67	0.29	0.00	0.33	0.00	0.40	0.00	N.A.	**0.26**
	Value Relationship	1.00	N.A.	N.A.	0.75	0.36	0.22	0.40	N.A.	0.50	N.A.	**0.54**
		0.80	N.A.	N.A.	0.67	0.29	0.25	0.40	N.A.	0.33	N.A.	**0.46**
		0.00	N.A.	N.A.	0.29	0.44	0.57	0.67	N.A.	0.67	N.A.	**0.44**
Version	Parameter Change	0.00	N.A.	0.29	0.44	0.36	0.00	0.29	0.33	0.00	N.A.	**0.21**
		0.00	N.A.	0.75	0.00	0.75	0.00	0.00	0.80	0.57	N.A.	**0.36**
		0.00	N.A.	0.67	0.00	0.29	0.00	0.00	0.00	0.00	N.A.	**0.12**

Finding 8. *Among the detected misconfigurations, LLMs correctly explained reasons for 93.9% of the misconfigurations; meanwhile, 6.1% of the reasons are misleading.*

When an LLM detects a misconfiguration, Ciri also asks the LLM to explain the reason to aid debugging and fixing the misconfiguration (§3.1). To evaluate these explanations, we randomly select one answer in which the misconfiguration is correctly detected per ⟨subcategory, project, LLM⟩ tuple, and collect a total of 740 answers (resulting from 2,220 queries). Upon careful manual review, we determined that 93.9% of the reasons given by the LLMs are clear and explain the misconfigurations. 3.1% of the answers contain a mix of correct and incorrect reasons across queries. Ciri filters out incorrect reasons using the voting mechanism (§3.2.2) as correct reasons are dominating. Figure 5.4 presents an example of mixed reasons, with the second reason being an instance of hallucination.



Config.
```
alluxio.underfs.gcs.default.mode=888
```
A1:
```
The property 'alluxio.underfs.gcs.default.mode' has
the value '888' which is not a valid octal number.
```
A2:
```
The property 'alluxio.underfs.gcs.default.mode' has
the value '888' which exceeds the range of an Integer.
```

Figure 5.4: Correct and incorrect reasons returned by LLMs.

In short, with frameworks like Ciri, LLMs can effectively validate configurations for syntax or range violations; but are less effective for the configurations that involve dependencies between parameters and software versions, showing the challenges for LLMs to reason about interactions between parameters and between configuration and code [81]. To address those misconfigurations, one can re-train or fine-tune LLMs with data related to dependency and versions. Lastly, while LLMs often provide correct explanations on misconfigurations that can aid debugging, it is crucial for developers to use these explanations with discretion, as they may not be consistently accurate.

5.4 BIASES

Finding 9. *LLMs are biased to popular parameters: Ciri is more effective in detecting misconfigurations of popular parameters, but also reports more false alarms on them.*

To measure the popularity of a configuration parameter, we count the number of exact-match search results returned by Google when searching the parameter name and call it *G-hits*.

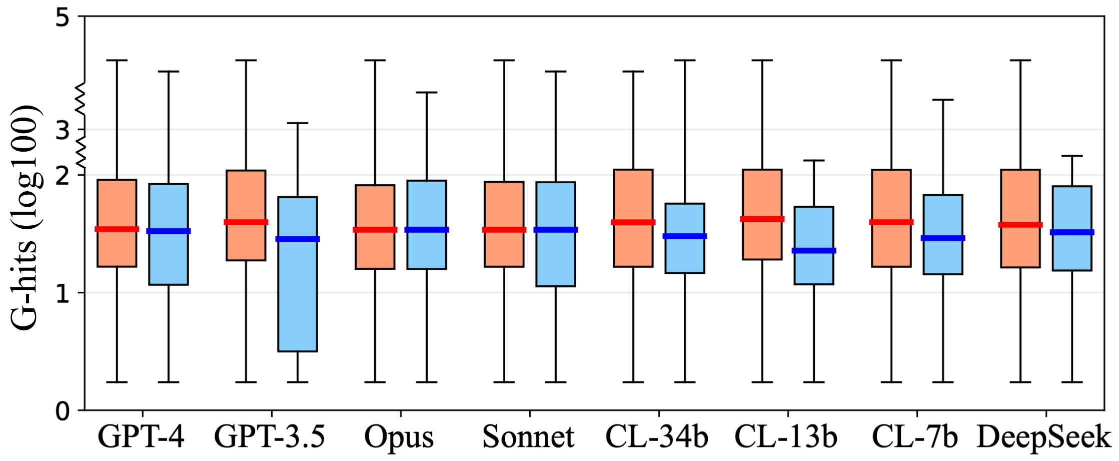


Figure 5.5: The G-hits distribution of the correctly detected misconfigurations (orange), and the G-hits distribution of the missed misconfigurations (blue). The bars in box plots indicate medians. CL refers to CodeLlama.

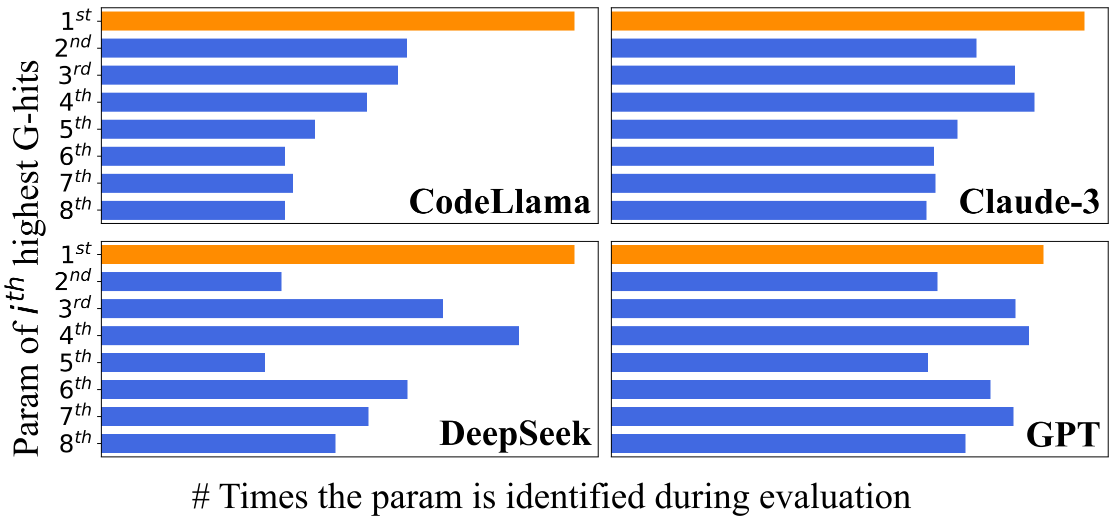


Figure 5.6: Frequency of the identified parameter with i^{th} highest G-hits in a config file.

We study the correlation between a parameter's G-hits and the effectiveness of LLMs in detecting the misconfigurations. For each configuration file in the Misconfig dataset, we track the frequency of LLMs detecting the parameter's misconfigurations with the i^{th} highest G-hits in each file, where $i = 1...8$. We separate cases when the misconfigured parameter is detected versus missed. As shown in Figure 5.5, the median G-hits of misconfigured parameters being detected is higher than the median G-hits of misconfigured parameters being missed.

We also study the frequency of false alarms across different ranking positions of G-hits within the file. Specifically, for each configuration file in ValidConfig dataset across all ten projects we evaluated, we track the frequency of LLMs mistakenly identifying the parameter with the i^{th} highest G-hits in each file, where $i = 1...8$. We group the results by the model family as shown in Figure 5.6. The distributions reveal a clear skewness towards parameters with higher G-hits, indicating that LLMs are more prone to report false alarms on popular parameters.

The biases can be attributed to the training data of LLMs, which are from public domains easily accessible by search engines like Google. Topics or parameters that are popularly discussed are more likely to be memorized by the LLMs, due to more frequent presence in the training data. So, for configuration validation, LLMs can be less effective for parameters that are not commonly referenced online.

CHAPTER 6: DISCUSSION AND FUTURE WORK

6.1 IMPROVING EFFECTIVENESS OF LLMS AS VALIDATORS.

Despite the promising results, using LLMs directly as configuration validators like Ciri is a starting point to harness the ability of LLMs for configuration validation. Specifically, there are circumstances where LLMs show limitations and biases (§5.3, §5.4). One intricate aspect of configuration validation is understanding configuration dependencies. Integrating LLMs with configuration dependency analysis [61] could be beneficial. For example, tools that use static taint analysis [29, 61] or dynamic information flow analysis [82, 83] can be utilized to extract and analyze configuration dependencies, which can then be fed as input to LLMs. This ensures that dependencies are accurately captured for LLMs.

To further enhance the understanding of LLMs, more information related to configurations can be incorporated, such as change logs, and specifications. Such information can provide valuable context, which has been proven by the prior work [73, 84, 85]. Moreover, We plan to investigate advanced prompting techniques, such as Chain-of-Thoughts (CoT) [57, 86, 87]. For configuration validation, CoT prompting can potentially mimic the reasoning process of a human expert. By eliciting LLMs to generate intermediate reasoning steps toward the validation results, it makes the validation more transparent and potentially more accurate.

We also plan to explore LLMs as agent. In fact, we experimented with two existing agent frameworks, TaskWeaver [88] and Mixtral [89], but both significantly under-performed compared with Ciri. TaskWeaver always fails in the middle due to inability of understanding configuration data; while Mixtral detected few misconfigurations. The challenge lies in crafting adept agents that are specialized in configuration-related tasks. We believe it is still a challenge to build agents that are specialized in configuration-related tasks. A promising direction is extending Ciri into a multi-agent framework, where Ciri can interact with additional tools such as Ctest [22] and Cdep [61] through agent frameworks such as LangChain [90] and AutoGen [91].

Lastly, integrating user feedback loops can be valuable. With user feedback on validation results, the iterative procedure can refine LLMs over time, leading to more accurate responses.

6.2 DETECTING ENVIRONMENT-RELATED MISCONFIGURATIONS.

While our study primarily targets misconfigurations that are common in the field [9], the validity of configuration files can vary across environments. For instance, a configuration parameter can specify a file path, so the file's existence, permission, and content decide its validity. To address these, LLMs can generate environment-specific scripts to run in the target environment. For example, given the configuration file as input, the LLM can generate a Python script as follows.

```
try:
    with open("/path/to/file", "r") as f:
        data = json.loads(f.read())
        print("Valid configuration")
except:
    print("Invalid configuration")
```

Such LLM-generated scripts can help identify issues like misconfigured paths, unreachable addresses, missing packages, or invalid permissions. Notably, these scripts offer a lightweight alternative to configuration tests [23, 74]. However, LLMs could also bring forth the possibility of generating erroneous scripts. To mitigate this, the concept of self-debugging for LLMs, as proposed in recent works [92, 93], can be integrated. This equips LLMs with the ability to introspect and correct their own errors, ensuring the reliability of the scripts they produce and make the configuration validation process robust.

For security concern, running those checks generated by LLMs need to be sandboxed. Moreover, the scripts can be reviewed by humans and transformed into lightweight validators. Given the recent success of code generation tools such as GitHub Copilot, we believe this is a promising direction to explore. In fact, our preliminary experiments show that, given appropriate examples, LLMs can generate such scripts quite well. as long as we provide a few examples of the kind of scripts we want.

6.3 DETECTING SOURCE-CODE RELATED MISCONFIGURATIONS.

In addition to the deployment environment, the system's source code can also affect the validity of a configuration. Often, implicit assumptions or software bugs can obscure the true requirements of a configuration. This could be due to implicit assumptions or even bugs in the software, which make it difficult to reason about the configuration's validity without considering the code. Implicit assumptions or latent software bugs can create ambiguities in understanding the true requirements of a configuration. To further illustrate this point, continuing the above example, if the documentation does not mention that the file needs to be in JSON format, but the code expects such a format, neither an LLM nor a human could infer this constraint based solely on the documentation. Based on the documentation alone, no LLM or human will be to guess that there is a constraint on the format of the file. We explored augmenting LLMs with code snippets (Finding 5), which can reveal parameter types and semantics. This approach can be further improved by integrating advanced program analysis to present both configuration and relevant source code to the LLM. Techniques like static or dynamic program slicing [21, 25, 82, 94] can help identify the relevant code. The LLM can then be tasked with distilling this code into a validator script. While this poses a challenge, the code reasoning capability of LLMs [95, 96] suggests that this is promising and worth further exploration.

6.4 FINE-TUNING LLMS FOR CONFIGURATION VALIDATION.

We also plan to explore fine-tuning to tackle system-specific configuration problems, which is hard to address with common-sense knowledge. Specifically, configuration related software evolution is prevalent, which introduces new parameters and changes the semantics and constraints of existing parameters [32, 33]. This is an important problem to tackle to unleash the full potential of LLMs for configuration validation. A promising solution is to fine-tune LLMs on new code/data, and make LLMs evolution-aware, but it is non-trivial, as due to lack of data on the newly introduced parameters. The LLM community has found promising results in using synthetic data [97, 98, 99, 100] for fine-tuning these models, reducing the need for large amounts of real data. We believe that this is a promising direction to explore for configuration validation as well.

6.5 MORE COMPREHENSIVE ANALYSIS.

As the first attempt to explore configuration validation using LLMs, we are planning a series of further analysis to gain a broader understanding of the domain. Our current study is based on evaluation of ten software systems. Incorporating more projects with more types of configuration designs can provide more comprehensive understanding, since each project could have its unique configuration challenges. Besides, our methodology currently employs a fixed-size parameter set (eight parameters), for both shots and the configuration files to be validated. Evaluating dynamic configuration file sizes could yield varied, and possibly more insightful, validation results. Moreover, the landscape of LLMs is vast and continually evolving. Investigating different LLM versions and architectures could offer a more comprehensive understanding of LLMs' potential in configuration validation.

6.6 ENHANCING HUMAN-LLM COLLABORATION.

Building upon the foundation of using LLMs for configuration validation, a critical aspect of future work involves enhancing the collaboration between human experts and LLMs [101]. This can be achieved by developing more intuitive interfaces and workflows that facilitate seamless interaction between humans and LLMs. For example, interfaces that allow experts to easily provide inputs, adjust parameters, or even directly modify LLM-generated validation scripts could be highly beneficial. Furthermore, establishing effective communication protocols where LLMs can ask clarifying questions to the experts, and vice versa, can significantly improve the accuracy and relevance of the validation process.

CHAPTER 7: RELATED WORK

7.1 CONFIGURATION VALIDATION.

Prior studies developed frameworks for developers to implement validators [1, 15, 16, 17] and test cases [22, 23], as well as techniques to extract configuration constraints [19, 20, 21, 24]. However, manually writing validators and tests requires extensive engineering efforts, and is hard to cover various properties of different configurations [24, 25, 26, 27, 28]. ML/NLP-based configuration validation techniques have been investigated. Traditional ML/NLP-based approaches learn correctness rules from configuration data [34, 35, 36, 37, 39, 41, 43, 44] and documents [38, 40] and then use the learned rules for validation. These techniques face data challenges and rely on predefined learning features and models, making them hard to generalize to different projects and deployment scenarios. We explore using LLMs for configuration validation, which can potentially address the limitations of traditional ML/NLP techniques towards automatic, effective validation solutions.

7.2 LLMS FOR SOFTWARE ENGINEERING.

LLMs have achieved high performance across various tasks such as text classification, text summarization, and logical reasoning [62, 102, 103, 104, 105]. Recently, they are actively applied to software engineering tasks, where they have demonstrated effectiveness in generating, summarizing, and translating code [50, 51, 106, 107, 108, 109, 110], failure diagnosis [73, 111], fault localization and program repair [52, 112, 113, 114]. LLMs for code are also increasingly prominent [50, 85, 115, 116, 117, 118], and are used for coding tasks. We take a first step to apply LLMs for software configuration problems, and show that LLMs have the potential to efficiently automate certain validation tasks and even bring advances over developer-written validators. Ciri as a framework is generic to different LLMs, and can be used to benchmark LLMs in terms of their abilities of configuration validation.

CHAPTER 8: CONCLUDING REMARKS

In this thesis, we present a first analysis on the feasibility and effectiveness of using LLMs such as GPT and Claude for configuration validation. Our goal is to empirically evaluate the promises of using LLMs as effective configuration validators and to understand the challenges. To do so, we develop Ciri, an LLM-empowered configuration validation framework. Ciri accepts configuration files or file diffs as input and provides detected misconfigurations along with explanatory reasons. The framework integrates various LLMs including GPT-4, Claude-3, and CodeLlama, employing effective prompt engineering and few-shot learning based on existing configuration data. Ciri also validates the outputs of LLMs to generate validation results, coping with the hallucination of LLMs. A key design principle of Ciri is separation of policy and mechanism. Ciri can serve as an open framework for experimenting with different models, prompt engineering, training datasets, and validation methods.

Our study evaluates the effectiveness of Ciri's validation capabilities using eight prominent LLMs, including both remote models and locally housed models. We evaluate ten widely deployed open-source systems with diverse types. Our study confirms the potential of using LLMs for configuration validation, e.g., Ciri with Claude-3-Opus detects 45 out of 51 real-world misconfigurations, outperforming recent configuration validation techniques. Our study also helps understand the design space of LLM-based validators like Ciri, especially in terms of prompt engineering with few-shot learning and voting. We find that using configuration data as shots can enhance validation effectiveness. Specifically, few-shot learning using both valid configuration and misconfiguration data achieves the highest effectiveness. Despite the encouraging results, our study revealed that directly using LLMs as configuration validators is ineffective in detecting certain types of misconfigurations such as dependency violations and version-related misconfigurations. It is also biased to the popularity of configuration parameters, causing both false positives and false negatives.

Our work shed light on new, exciting future research directions of using LLMs for software configuration research. We propose integrating LLMs with configuration dependency analysis tools and enriching them with additional context like change logs and advanced prompting techniques. Addressing environment-specific and source-code-related misconfigurations, we suggest the use of LLM-generated scripts and program analysis. The importance of fine-tuning LLMs for specific system configurations and conducting broader analyses across diverse software systems is emphasized. Lastly, we highlight the need to improve collaboration between human experts and LLMs through intuitive interfaces and effective communication, leading to more efficient and accurate configuration validation processes.

REFERENCES

[1] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic Configuration Management at Facebook," in *SOSP*, 2015.

[2] B. Maurer, "Fail at Scale: Reliability in the Face of Rapid Change," *Communications of the ACM*, 2015.

[3] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed. Morgan and Claypool Publishers, 2018.

[4] A. Sherman, P. Lisiecki, A. Berkheimer, and J. Wein, "ACMS: Akamai Configuration Management System," in *NSDI*, 2005.

[5] S. Mehta, R. Bhagwan, R. Kumar, B. Ashok, C. Bansal, C. Maddila, C. Bird, S. Asthana, and A. Kumar, "Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis," in *NSDI*, 2020.

[6] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *Site Reliability Workbook: Practical Ways to Implement SRE.* O'Reilly Media Inc., 2018.

[7] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software Configuration Engineering in Practice: Interviews, Surveys, and Systematic Literature Review," *TSE*, 2018.

[8] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," in *SOCC*, 2016.

[9] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An Empirical Study on Configuration Errors in Commercial and Open Source Systems," in *SOSP*, 2011.

[10] T. Xu and Y. Zhou, "Systems Approaches to Tackling Configuration Errors: A Survey," *ACM Computing Surveys*, vol. 47, no. 4, 2015.

[11] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" in *USITS*, 2003.

[12] S. Kendrick, "What Takes Us Down?" *USENIX ;login:*, 2012.

[13] A. Rabkin and R. Katz, "How Hadoop Clusters Break," *IEEE Software Magazine*, 2013.

[14] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, "Set the configuration for the heart of the os: On the practicality of operating system kernel debloating," 2020.

[15] M. Raab and G. Barany, "Challenges in Validating FLOSS Configuration," in *OSS*, 2017.

[16] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable Declarative Configuration Specification and Validation for Applications, Systems, and Cloud," in *Middleware*, 2017.

[17] P. Huang, W. J. Bolosky, A. Sigh, and Y. Zhou, "ConfValley: A Systematic Configuration Validation Framework for Cloud Services," in *EuroSys*, 2015.

[18] L. Leuschner, M. Küttler, T. Stumpf, C. Baier, H. Härtig, and S. Klüppelholz, "Towards Automated Configuration of Systems with Non-Functional Constraints," in *HotOS-XVI*, 2017.

[19] X. Liao, S. Zhou, S. Li, Z. Jia, X. Liu, and H. He, "Do You Really Know How to Configure Your Software? Configuration Constraints in Source Code May Help," *IEEE Transactions on Reliability*, 2018.

[20] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *TSE*, 2015.

[21] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static Detection of Silent Misconfigurations with Deep Interaction Analysis," in *OOPSLA*, 2021.

[22] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing Configuration Changes in Context to Prevent Production Failures," in *OSDI*, 2020.

[23] T. Xu and O. Legunsen, "Configuration Testing: Testing Configuration Values as Code and with Code," *arXiv:1905.12195*, 2019.

[24] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do Not Blame Users for Misconfigurations," in *SOSP*, 2013.

[25] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early Detection of Configuration Errors to Reduce Failure Damage," in *OSDI*, 2016.

[26] S. Li, W. Li, X. Liao, S. Peng, S. Zhou, Z. Jia, and T. Wang, "ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection," *IEEE Transactions on Reliability*, 2018.

[27] L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A Tool for Assessing Resilience to Human Configuration Errors," in *DSN*, 2008.

[28] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X. Liao, "Challenges and Opportunities: An In-Depth Empirical Study on Configuration Error Injection Testing," in *ISSTA*, 2021.

[29] T. Wang, H. He, X. Liu, S. Li, Z. Jia, Y. Jiang, Q. Liao, and W. Li, "ConfTainter: Static Taint Analysis For Configuration Options," in *ASE*, 2023.

[30] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and Detecting On-the-Fly Configuration Bugs," in *ICSE*, 2023.

[31] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have Given Me Too Many Knobs! Understanding and Dealing with Over-Designed Configuration in System Software," in *ESEC/FSE*, 2015.

[32] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?" in *ICSE*, 2014.

[33] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An Evolutionary Study of Configuration Design and Implementation in Cloud Systems," in *ICSE*, 2021.

[34] R. Bhagwan, S. Mehta, A. Radhakrishna, and S. Garg, "Learning Patterns in Configuration," in *ASE*, 2021.

[35] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection," in *ASPLOS*, 2014.

[36] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic Misconfiguration Troubleshooting with PeerPressure," in *OSDI*, 2004.

[37] F. Le, S. Lee, T. Wong, H. S. Kim, and D. Newcomb, "Minerals: Using Data Mining to Detect Router Misconfigurations," Carnegie Mellon University, Tech. Rep. CMU-CyLab-06-008, 2006.

[38] C. Xiang, H. Huang, A. Yoo, Y. Zhou, and S. Pasupathy, "PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations," in *ATC*, 2020.

[39] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems," in *KDD*, 2006.

[40] R. Potharaju, J. Chan, L. Hu, C. Nita-Rotaru, M. Wang, L. Zhang, and N. Jain, "ConfSeer: Leveraging Customer Support Knowledge Bases for Automated Misconfiguration Detection," in *VLDB*, 2015.

[41] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," in *LISA*, 2003.

[42] E. Kiciman and Y.-M. Wang, "Discovering Correctness Constraints for Self-Management of System Configuration," in *ICAC*, 2004.

[43] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic Automated Language Learning for Configuration Files," in *CAV*, 2016.

[44] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing Configuration File Specifications with Association Rule Learning," in *OOPSLA*, 2017.

[45] Q. Huang, H. J. Wang, and N. Borisov, "Privacy-Preserving Friends Troubleshooting Network," in *NDSS*, 2005.

[46] "ChatGPT," https://openai.com/blog/chatgpt, 2022.

[47] "Codex," https://openai.com/blog/openai-codex, 2022.

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," *arXiv:2201.11903*, 2023.

[49] J. Huang and K. C.-C. Chang, "Towards Reasoning in Large Language Models: A Survey," in *Findings of ACL*, 2023.

[50] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.

[51] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago et al., "Competition-level code generation with alphacode," *Science*, 2022.

[52] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *ICSE*, 2023.

[53] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung et al., "A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity," *arXiv:2302.04023*, 2023.

[54] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen et al., "Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models," *arXiv:2309.01219*, 2023.

[55] Anthropic, "Introducing 100k context windows," https://www.anthropic.com/index/100k-context-windows, 2023.

[56] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman, "Webgpt: Browser-assisted question-answering with human feedback," *arXiv:2112.09332*, 2022.

[57] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-Consistency Improves Chain of Thought Reasoning in Language Models," *arXiv:2203.11171*, 2023.

[58] Y. Liu, Y. Yao, J.-F. Ton, X. Zhang, R. Guo, H. Cheng, Y. Klochkov, M. F. Taufiq, and H. Li, "Trustworthy llms: a survey and guideline for evaluating large language models' alignment," *arXiv:2308.05374*, 2023.

[59] P. Manakul, A. Liusie, and M. J. F. Gales, "Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models," *arXiv:2303.08896*, 2023.

[60] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," *arXiv:1909.08593*, 2019.

[61] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems," in *ESEC/FSE*, 2020.

[62] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al., "Language models are few-shot learners," *arXiv:2005.14165*, 2020.

[63] OpenAI, "Gpt-4 technical report," *arXiv:2303.08774*, 2023.

[64] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[65] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *arXiv:2005.11401*, 2021.

[66] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "Realm: Retrieval-augmented language model pre-training," *arXiv:2002.08909*, 2020.

[67] "Openctest," https://github.com/xlab-uiuc/openctest, 2020.

[68] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu et al., "Towards a human-like open-domain chatbot," *arXiv:2001.09977*, 2020.

[69] O.-M. Camburu, B. Shillingford, P. Minervini, T. Lukasiewicz, and P. Blunsom, "Make up your mind! adversarial generation of inconsistent natural language explanations," *arXiv:1910.03065*, 2019.

[70] Y. Elazar, N. Kassner, S. Ravfogel, A. Ravichander, E. Hovy, H. Schütze, and Y. Goldberg, "Measuring and improving consistency in pretrained language models," *arXiv:2102.01017*, 2021.

[71] W. contributors, "tf–idf," https://en.wikipedia.org/wiki/Tf%E2%80%93idf, 2024.

[72] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A Large Language Model for Verilog Code Generation," *arXiv:2308.00708*, 2023.

[73] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen et al., "Empowering Practical Root Cause Analysis by Large Language Models for Cloud Incidents," *arXiv:2305.15778*, 2023.

[74] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-Case Prioritization for Configuration Testing," in *ISSTA*, 2021.

[75] S. Wang, X. Lian, D. Marinov, and T. Xu, "Test Selection for Unified Regression Testing," in *ICSE*, 2023.

[76] T. Xu and D. Marinov, "Mining Container Image Repositories for Software Configurations and Beyond," in *ICSE-NIER*, 2018.

[77] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" *arXiv:2202.12837*, 2022.

[78] A. Rahman, C. Parnin, and L. Williams, "The seven sins: security smells in infrastructure as code scripts," 2019.

[79] Q. Huang, H. Wang, and N. Borisov, "Privacy-preserving friends troubleshooting network," in *NDSS*, 2005.

[80] W. S. Tan, M. Wagner, and C. Treude, "Wait, wasn't that code here before? detecting outdated software documentation," 2023.

[81] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On Essential Configuration Complexity: Measuring Interations in Highly-Configurable Systems," in *ASE*, 2016.

[82] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *OSDI*, 2010.

[83] Attariyan, Mona and Chow, Michael and Flinn, Jason, "X-ray: automating {Root-Cause} diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[84] Z. Zhang, A. Zhang, M. Li, H. Zhao, G. Karypis, and A. Smola, "Multimodal chain-of-thought reasoning in language models," *arXiv:2302.00923*, 2023.

[85] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *CLOUD*, 2019.

[86] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou et al., "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, 2022.

[87] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," *arXiv:2210.03493*, 2022.

[88] B. Qiao, L. Li, X. Zhang, S. He, Y. Kang, C. Zhang, F. Yang, H. Dong, J. Zhang, L. Wang, M. Ma, P. Zhao, S. Qin, X. Qin, C. Du, Y. Xu, Q. Lin, S. Rajmohan, and D. Zhang, "Taskweaver: A code-first agent framework," 2024.

[89] "Mixtral," https://huggingface.co/blog/mixtral, 2023.

[90] "Langchain," https://github.com/langchain-ai/langchain, 2022.

[91] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023.

[92] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[93] L. Pan, M. Saxon, W. Xu, D. Nathani, X. Wang, and W. Y. Wang, "Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies," *arXiv preprint arXiv:2308.03188*, 2023.

[94] A. Rabkin and R. Katz, "Precomputing Possible Configuration Error Diagnosis," in *ASE*, 2011.

[95] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv:2005.00653*, 2020.

[96] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu et al., "Automating code review activities by large-scale pre-training," in *ESEC/FSE*, 2022.

[97] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," *arXiv:2304.12244*, 2023.

[98] J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han, "Large language models can self-improve," *arXiv:2210:11610*, 2022.

[99] P. Haluptzok, M. Bowers, and A. T. Kalai, "Language models can teach themselves to program better," *arXiv:2207.14502*, 2023.

[100] Y. Wang, Y. Kordi, S. Mishra, A. Liu, N. A. Smith, D. Khashabi, and H. Hajishirzi, "Self-instruct: Aligning language models with self-generated instructions," *arXiv:2212.10560*, 2023.

[101] T. Xu, V. Pandey, and S. Klemmer, "An hci view of configuration problems," 2016.

[102] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler et al., "Emergent abilities of large language models," *arXiv:2206.07682*, 2022.

[103] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, 2022.

[104] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[105] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar et al., "Llama: Open and efficient foundation language models," *arXiv:2302.13971*, 2023.

[106] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv:1808.09588*, 2018.

[107] T. Ahmed and P. Devanbu, "Few-shot training LLMs for project-specific code-summarization," in *ASE*, 2022.

[108] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv:2102.04664*, 2021.

[109] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, 2020.

[110] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," *arXiv:2110.06773*, 2021.

[111] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending Root-Cause and Mitigation Steps for Cloud Incidents Using Large Language Models," in *ICSE*, 2023.

[112] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "Vulchecker: Graph-based vulnerability localization in source code," in *USENIX Security*, 2023.

[113] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *ICSE*, 2023.

[114] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv:2308.04738*, 2023.

[115] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., "Codebert: A pre-trained model for programming and natural languages," *arXiv:2002.08155*, 2020.

[116] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv:2203.13474*, 2022.

[117] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. tau Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," *arXiv:2204.05999*, 2023.

[118] F. F. Xu, U. Alon, G. Neubig, and V. J.Hellendoorn, "A Systematic Evaluation of Large Language Models of Code," *arXiv:2202.13169*, 2022.