
Siri for Network Configuration

Ze Yang

University of Illinois Urbana-Champaign
zeyang2@illinois.edu

Ziheng (Jack) Chen

University of Illinois Urbana-Champaign
zihengc2@illinois.edu

Shengkun Cui

University of Illinois Urbana-Champaign
scui8@illinois.edu

Abstract

Network configuration is crucial to the connectivity, reliability, and security of large-scale cloud systems. Existing network configuration validation tools like Batfish focus on detecting network configuration errors but not correcting them. Consequentially, correcting network configuration errors remains mostly a manual process, making it complicated and error-prone. How to efficiently validate and correct network configurations with less engineering intervention remains a research challenge. The recent advancement of Large Language Models (LLMs)’s code generation capability and the rapid adaptation of the “configuration-as-code” paradigm shed new light on this problem. Specifically, LLM-aided configuration validation and correction solutions have started to emerge. This project characterizes a state-of-the-art LLM (GPT-4 Turbo)’s capability to identify and correct network configuration errors using a total of 100 faulty network configuration files covering 5 commonly encountered fault categories in a production environment. Our result shows that GPT-4 Turbo successfully identifies and corrects 70% of these faults without any context and subsequently improves to 85% by applying in-context learning. Furthermore, this project provides directions for future research by highlighting key side-effects, including hallucinations, observed in LLM-aided configuration validation and correction, as demonstrated through case studies.

1 Introduction

Due to rapid development and advancement in cloud and web services, production systems often require thousands of configuration changes per day Barroso et al. (2018); Tang et al. (2015); Sherman et al. (2005); Oppenheimer et al. (2003), leading to an increasing number of misconfigurations. Network configurations (referred to as configurations hereafter), besides the software configurations, are crucial to the operation of large-scale cloud systems. Network misconfigurations are among the dominating causes of production incidents according to Gunawi et al. (2016); Yin et al. (2011); Oppenheimer et al. (2003), which are the major cost of support at cloud and data center vendors [cdep - 67, 92]. These configurations generally serve three main purposes: (1) Connectivity: configurations are used to describe the interconnection of HW components such as computing, storage, remote memory, etc. It is also used for SW components such as Kubernetes clusters. (2) Reliability: configurations ensure the reachability for key network services like SSH, DNS, etc. (3) Security: configurations control network traffic flow and prevent unwanted accesses via firewall and routing rules. Existing configuration validation tools such as Batfish Fogel et al. (2015) emphasize misconfiguration detection but not correction. Consequentially, current methods for correcting misconfigurations involve manual inspection with extensive engineering efforts, which are expensive and often error-prone. To address these issues, the paper explores the use of Large Language Models (LLMs) like GPT and Codex for automatic configuration validation and correction.

LLMs offer advantages such as automatic scalability, generalization, and reasoning abilities, making them suitable for misconfiguration detection. Recently, LLM-aided network management has started to emerge, which proves that LLM can generate code for network traffic analysis and network topology manipulation Mani et al. (2023). On the other hand, the efficacy of LLMs on configuration correction tasks is not guaranteed since they are not fine-tuned to network configuration tasks, and providing context might be difficult since the context size might not be sufficient to fit a large number of configurations, and LLMs might hallucinate and provide overly generalized or erroneous output, which can lead to increased human diagnosis and remediation effort, becoming counterproductive. In this project, we show how LLMs, trained on extensive internet data, can encode knowledge about common and project-specific configurations. We first test LLM’s ability to identify and correct network configuration errors by injecting 5 predefined fault categories onto a known correct configuration. We then characterize the failure modes of state-of-the-art (SOTA) LLMs on these tasks, summarizing the kind of errors the LLM cannot detect. Our result shows that GPT-4 Turbo cannot detect 95% of the functional errors. We then made the LLM-generated improvements pointing in the direction to improve the answer. We also utilized different fine-tuning methods to improve the foundation model output under each failure mode. Finally, we analyzed the common artifacts encountered during our experiments.

1.1 Related Work

1.1.1 Configuration Validation

The early identification of configuration errors is crucial in preventing failures. As the configurations responsible for failure handling and fault tolerance are often not required for system initialization, many systems do not rigorously check their settings. Thus, such latent errors are likely to be deployed at scale in production. Consequently, these errors remain latent until their effects result in significant damage, such as compromising failure handling mechanisms.

Previous research indicates that many widely used software systems suffer from latent configuration errors (referred to as LC errors) in critical configurations affecting the system’s reliability, availability, and serviceability. A contributing factor is that a significant percentage (14.0%–93.2%) of these configurations lack specialized code to verify the correctness of their settings during the system’s initialization Xu et al. (2016). To address this issue and enable software systems to detect LC errors early, we aim to leverage LLM for assistance in this process.

To identify potential misconfigurations before implementation, modern configuration management systems typically embrace the “configuration-as-code” approach, where the configuration file is examined for both syntax and logical errors. Such configurations must undergo continuous validation through various stages, including static validation, configuration testing, and manual review and approval. The process begins with the validation of the configuration using predefined correctness rules implemented in the validation tools (referred to as validators) Baset et al. (2017); Leuschner et al. (2017); Huang et al. (2015); Yin et al. (2011). Engineers often utilize these validators as building blocks and write customized test cases when introducing new configuration parameters. Once the configuration passes validation, changes are tested alongside the user application to ensure the expected program behavior under the altered configuration Sun et al. (2020). Finally, similar to the review process for source code, configuration changes undergo scrutiny, usually in the form of a “diff” file, before being approved for production deployment. As a result, such complex validation often frees the configuration from trivial faults, e.g., typos. However, the current state of validation methods is still susceptible to complex errors that have incorrect logic due to the limitations.

1.1.2 LLM Characterization

While LLMs are shown to be increasingly effective on programming code generation tasks Chen et al. (2021); Kojima et al. (2023); Touvron et al. (2023); Poldrack et al. (2023), the discussions on their abilities to localize and correct configuration-related faults are limited. Although our scope is limited to network configurations, we are the first to characterize SOTA LLMs’ misconfiguration localization and correction efficacies.

1.2 Our Contribution

We claim the following contributions.

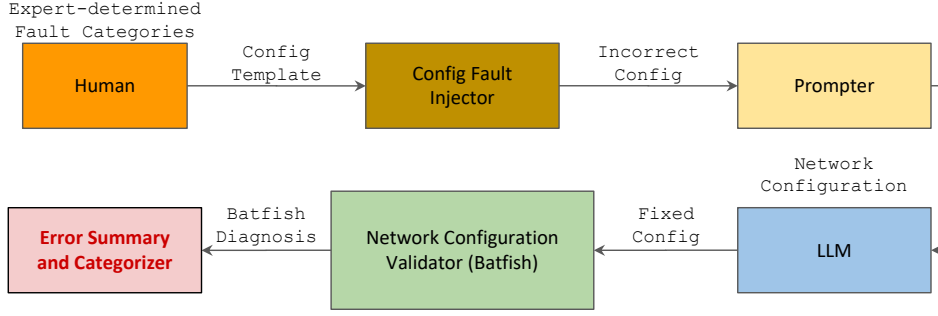


Figure 1: (1) Human sends a correct net configuration to a fault injector, specifying the fault category to be injected as well as the number of faults. (2) The fault injector automatically generates net configurations with altered content and sends it to the prompter. (3) Prompter sets the context for the LLM by providing system and user prompts and asking the LLM to find the error(s). (4) LLM completes the config and passes it to the validator (Batfish) that provides correctness checks. (5) Batfish outputs binary validation results that are summarized.

1. Characterizes SOTA LLM’s efficacy and failure modes (GPT-4 Turbo) on network configuration validation and correction.
2. Improves LLM efficacy via in-context learning and provides practical examples in network configuration validation and correction.
3. Presents failure case studies and highlights the current limitations, side-effects, and future research directions of SOTA LLM (GPT-4 Turbo) in LLM-aided network configuration validation and correction.
4. Proposes a validation-in-the-loop approach using the Batfish network configuration tool for LLM post-correction validation.

2 Details of the Approach

2.1 System Design Overview

The process begins with the human sending a correct network configuration to the fault injector, specifying the desired fault category and the number of injected faults in a single file. The fault injector programmatically introduces faults by changing a small segment in the original configuration, e.g., an interface name, based on the specified fault category and count, which then generates a modified network configuration. These altered configurations are then passed to the prompter, which establishes the context for the Language Model (LLM) by providing system and user prompts, and optionally knowledge on network faults. The LLM is tasked with identifying and rectifying the top injected faults, ranked by importance, within the network configurations. The completed configurations are then forwarded to the Batfish validator, which performs correctness checks. Batfish outputs binary validation results, indicating the success or failure of the configurations. The overall result is summarized, providing insights into the LLM’s ability to identify and correct faults in network configurations under various fault categories. The process is repeated multiple times to test for a certain number of faults.

2.2 Error Category and Generation

We used a Python script to generate diverse and intentionally flawed network configuration files for testing network configuration analysis tools. The script employs a variety of injection methods to introduce invalid elements into a given base configuration file, into the lines that are not commented. These methods include injecting random invalid symbols, altering IP addresses and subnets with invalid values (e.g., out of range), manipulating interface configurations (e.g., changing to invalid interface names), modifying path configurations, and randomly swapping control words (e.g., "permit",

Table 1: Error categories

Category	Sub-Category	Explanation	Example	Number
Syntax	Invalid Subnets	Randomly inserting ['@', '%', '^', '&', '(', ')'] in subnet expressions	service timestamps log datetime msec → service timestamps log datetime msec^	10
Syntax	Invalid Interfaces	Randomly inserting ['@', '%', '^', '&', '(', ')'] in interface expressions	2.128.0.0/9 → 2.128.0.0/9	20
Range	IP out-of-range	Randomly changing a part of the IP out of range (only 1 IP address is changed)	neighbor 2.1.2.322 activate	5
Range	IP out-of-range	Randomly changing a part of the IP out of range (multiple IP addresses are changed)	neighbor 2.1.2.322 activate, neighbor 2.322.2.200 activate	20
Functional	Incorrect Permissions	Randomly swapping a permission keyword ['permit', 'deny', 'remark'] with another	ip community-list expanded as1_community permit _1: → ip community-list expanded as1_community remark _1:	20

"deny", "remark"). Each injection method produces multiple modified configurations of the same category, creating a set of test cases for evaluating the effectiveness of network configuration analysis tools in handling diverse and erroneous inputs.

2.3 Model Selection

GPT-4 Turbo (simplified as GPT-4T) is asserted to be the most sophisticated and extensively utilized large language model (LLM) currently available. In comparison to GPT-3.5-turbo and GPT-4, GPT-4T achieves better overall performance. It underwent training on data preceding April 2023, thus having more foundational knowledge. In addition, it comes with a token limit of 128K per query, which makes it possible to provide it with an extensive amount of domain knowledge to test its in-context learning ability while still being economically friendly.

2.4 Model Parameter

It is important to recognize that GPT-4T, like many other LLMs, exhibits inherent randomness in their inference processes. To address this issue, we set its temperature to 0, ensuring a consistent output for identical input Xu et al. (2024). This approach effectively mitigates the unpredictability associated with the inference process.

2.5 Batfish Network Configuration Validation Tool

We choose Batfish Fogel et al. (2015) for network configuration validation due to its comprehensive capabilities. Batfish is a widely adopted open-sourced tool in network configuration analysis. It simulates and scrutinizes network configurations. This enables the detection of issues, misconfigurations, and security violations before live network deployment.

Batfish's operation involves first creating a logical model of the network's control and data planes from configuration files. It translates network configurations into a Datalog variant, enabling a comprehensive analysis of configurations and their interactions. By executing the Datalog program,

Batfish produces logical relations representing the data plane and essential computational concepts, and it uses an automatic constraint solver to verify various correctness properties, proactively detecting errors and assessing their impact on network forwarding.

In summary, while various tools like NAPALM, pyATS, NCC, Netmiko, and YANG-based tools exist, Batfish stands out for its unique features:

1. It parses network device configurations, allowing users to query and analyze the network's state without making any changes to the actual devices.
2. It enables users to check if their network configurations comply with specified policies, ensuring that security and operational requirements are met.
3. It is designed to work with configurations from various network equipment vendors, providing a vendor-agnostic approach to network validation.

2.6 Evaluation Metric

In evaluating the performance of our GPT-4T for predicting network configuration failures, we focus on measuring the success rate (true positive rate or TPR) for both top 1 and top 5 predictions. Our methodology compares the model's predictions against a dataset of known failures and expert assessments to determine accuracy. For the top-1 success rate, the case is considered a success when the LLM ranks the corresponding faults as the top failure. For the top-5 success rate, the case is considered a success if the fault is identified among the top-5 failure cases. This approach quantifies the model's efficacy in identifying the most relevant and likely network failures among possible false positives, providing a robust measure of its predictive capability in handling complex network configurations.

2.7 Prompt Formatting

The prompt used for the project consists of two parts: a system prompt and a user prompt. A system prompt is a way of providing context and instructions to GPT-4T, such as specifying a particular goal or role for GPT-4T before asking it a question or giving it a task. In short, the system prompts state the ways for the model to act and primary instructions to follow, and the user prompts contain the question/query and secondary instructions to follow.

Our Prompt Template is shown below:

```
System Prompt:
You are a network configuration operator.
And you will find error(s) and guarantee the correctness of planned
or current network configurations.
{In-context learning examples}
-----
User Prompt:
The network configuration is below:
{Config Input}
Pls help me find the most possible error(s) in the configuration and help
me correct it.
The answer format should be a JSON like below and pls only return JSON:
Correctness: Yes, if there is any error, and No, if there is no error
RootCause:
The top five error(s) with their line number, content, and explanation of
the error(s).
If there are less than five errors, list as many as you have.
Example: Provided an example for output format control...(omit for space)
```

The system prompt defines GPT-4T as the network configuration operator to ensure configuration correctness. In the initial experiments, we do not provide GPT-4T with any context to learn from. In subsequent experiments, we add additional domain-specific context to the system prompts to improve the answer accuracy, which is referred to as *in-context learning* (Section 3.3). The user prompt

provides the network configuration and output format instructions. Output examples will be added and used to improve the stability and accuracy of the output format.

2.8 Sample Output

Below is a sample response of the GPT-4T on correcting an injected fault that belongs to the “Invalid Symbol” subcategory.

```
{
  "Correctness": "Yes",
  "RootCause": [
    "1. Line 4 with 'service timestamps log datetime msec^' has an invalid character '^' at the end of the line, which is not a valid syntax in the configuration.",
    "2. Line 63 with 'ip access-group INSIDE_TO_AS3 out' references an access list 'INSIDE_TO_AS3' for outbound traffic on the interface GigabitEthernet0/0, but the access list is defined as 'ip access-list extended INSIDE_TO_AS3' which is for inbound traffic. The 'out' keyword is incorrect in this context.",
    "3. Line 97 with 'aggregate-address 2.128.0.0 255.255.0.0 summary-only' has an incorrect subnet mask. The correct subnet mask for a /9 prefix should be '255.128.0.0'.",
    "4. Line 118 with 'access-list 101 permit ip host 1.0.1.0 host 255.255.255.0' has an incorrect wildcard mask. The 'host' keyword should be followed by a single host IP address, not a subnet. The correct line should be 'access-list 101 permit ip host 1.0.1.0 0.0.0.255'.",
    "5. Line 119 with 'access-list 101 permit ip host 1.0.2.0 host 255.255.255.0' has the same issue as the previous line, with an incorrect wildcard mask. The correct line should be 'access-list 101 permit ip host 1.0.2.0 0.0.0.255'."
  ],
  "Corrected config": {...} omit for space.
}
```

3 Experiments and Results

3.1 Handling Single Fault per File

Without providing domain-specific context, we observed that GPT-4T exhibited strong performance in the invalid interface, range, and subnet error categories, with top-1 success rates ranging from 70% to 96%. This result suggests that GPT-4T can effectively identify and correct issues related to invalid interfaces, out-of-range IPs, and invalid subnets. However, it is crucial to note that the "Invalid Symbol" category presents challenges with the random insertion of symbols as GPT-4T only achieves a top-1 success rate of 52%. Moreover, the "Functionality" category, specifically "Incorrect Permissions," posed a significant challenge, with GPT-4T struggling to detect errors in this domain (top-1 success rate is 0%), highlighting the complexity of capturing nuanced functional requirements.

3.2 Handling Multiple Faults per File

This project also characterizes GPT-4T's ability to identify multiple faults within a single configuration file. As the number of faults increases, GPT-4T's success rate in identifying all faults decreases, as shown in Table 3. Nevertheless, GPT-4T demonstrated an ability to accurately pinpoint a higher

Table 2: Overall performance of GPT-4T, without in-context learning.

Category	Sub-Category	Total #	Top-1 Error		Top-5 Error	
			Count	(%)	Count	(%)
Syntax	Invalid Symbol	25	13	(52%)	16	(64%)
Syntax	Invalid Subnets	10	7	(70%)	9	(90%)
Syntax	Invalid Interfaces	20	19	(95%)	19	(95%)
Range	IP out-of-range	25	24	(96%)	25	(100%)
Functional	Incorrect Permissions	20	0	(0%)	1	(5%)
Total		100	63	(63%)	70	(70%)

Table 3: Performance of GPT-4T on finding multiple errors in a file

# of faults per file	Average success rate per file
1	100%
2	100%
3	87%
4	95%
5	88%

number of faults within a single configuration file without sacrificing overall accuracy. This suggests the potential for GPT-4T to excel in scenarios where multiple errors coexist within a configuration, reinforcing their robustness in handling complex network configurations.

3.3 Improve Success Rate with In-Context Learning

A notable feature of high parameter count LLMs such as GPT-4T is the ability to perform in-context-learning, i.e., to improve the quality of the answers by providing context-specific examples in the prompt, without parameter fine tuning Dong et al. (2023). To understand how domain-specific context can help improve GPT-4T’s performance, we re-run the evaluation on “Invalid Symbol” and “Incorrect Permissions” subcategories with network configuration-specific context. An example context included in the prompt for “Invalid Symbol” is shown below:

Additional System Prompt for In-context Learning:
 You may encounter errors like invalid range for IP, invalid syntax, invalid character insertion, functional misconfig and so on.
 For invalid character insertion, some characters like @, %, ^, &, (,), / may be mistyped into the config.
 e.g., "%" is mistyped into the config "%version 15.2", and "/" is mistyped into "ip prefix-list inbound_route_filter seq 5 deny 2.0.0.0/8 le 32".

The re-evaluated results with in-context-learning for “Invalid Symbol” and “Incorrect Permissions” are shown in Table 4. Significant increases in both top-1 and top-5 success rates are observed for the “Invalid Symbol” subcategory, with the top-1 success rate increasing from 52% to 84%, and the top-5 success rate increasing from 64% to 100%. We observe limited improvement for the “Incorrect Permissions” category with top-1 and top-5 success rates increasing from 0% to 5% and from 5% to 35%, respectively. We hypothesize that the less improvement for the “Incorrect Permission” subcategory compared to the “Invalid Symbol” subcategory is because each functional fault relies on the specific context. For example, in one case, the correct context is to *permit* DNS packets, while in another case, it is to *deny* DNS packets. The specific contexts depend on the specific functional needs

of a configuration. Since there is no overarching context (e.g., the range of the IP octet is 0-255) for “functional faults”, in-context learning provides limited improvement here. This also highlights the current LLMs deficiency in handling domain-specific queries, as we will highlight in the discussion.

As we move forward, it becomes apparent that while LLMs show promise in addressing certain network configuration challenges, further research and development are needed to enhance their capabilities, especially in understanding and rectifying functional aspects. Our findings underscore the importance of context-awareness in handling symbol-related errors and highlight the need for continuous improvement in GPT-4T to meet the diverse and intricate requirements of network configuration in large-scale cloud systems.

4 Discussion and Conclusions

4.1 Limitation of SOTA LLMs on Network Configuration Validation and Correction

Below, we highlight the limitation of SOTA LLM (GPT-4T) on network misconfiguration localization and correction.

4.1.1 Hallucination

Extrinsic hallucination of LLMs is evident in our evaluation. First, if we do not specify the number of faults in the prompt, GPT-4T identifies multiple configuration faults within a file until the allowable output token length is exhausted. Those faults identified are mostly false positives. To address this problem, we limit the number of faults that GPT-4T needs to return by providing a template in the prompt.

Second, although GPT-4T correctly localizes the fault(s), it does not ask the user for their intent when returning the corrected configurations, that is, the returned configurations are syntactically correct but not necessarily functionally correct. For example, GPT-4T successfully localizes the invalid IP in the command

```
Neighbor 2.296.2.1 update-source Loopback0
```

as the value of the second octet is 296, which exceeds the maximum value of 255 for an octet. However, when GPT-4T returns the correct configuration, it directly changes 296 to 255, which can be functionally incorrect without asking the user’s input.

4.1.2 Degraded Performance on Correcting Functional Faults

Without providing context, GPT-4T performs well in handling simple network configuration faults, such as IP range errors, invalid subnets, and invalid interfaces. With in-context learning, GPT-4T can also handle vendor-specific syntax errors, e.g., invalid symbols, to some extent. However, GPT-4T performs poorly when dealing with syntactically correct but functionally incorrect configurations, despite the usage of in-context learning. It is also more difficult to provide a static, overarching context for functional faults since each functional fault requires a specific context to suit the functional needs of the user. A potential solution is to use dynamic contexts and examples extracted from system architecture and user intentions for in-context learning, with additional costs and expert interventions. The cost of the proposed improvement is the latency and instability associated with the multiple rounds of GPT-4T calls involved and the need for real-time filtering and updating of the context data due to system and application updates. Validation tools such as Batfish extract contexts and rebuild the control and data plane according to the configuration, which could be useful in providing in-context learning examples for LLMs. Such validators can also be used to validate the post-correction configuration generated by LLMs to further limit the impact of hallucination. Recent work Schick et al. (2023) by Meta proves that LLMs can interact with external tools via simple APIs.

4.1.3 Summary

In conclusion, while GPT-4T demonstrates the capability to spot error(s) and generate correct configurations, it relies on certain assumptions and defaults without seeking user input, potentially leading to configurations that may not align with specific user preferences.

Table 4: Overall performance of GPT-4T, with in-context learning using examples of “Invalid Symbol” and “Incorrect Permissions” sub-categories.

Category	Sub-Category	Total #	Top-1 Error		Top-5 Error	
			Count	(%)	Count	(%)
Syntax	Invalid Symbol	25	21	(84%)	25	(100%)
Syntax	Invalid Subnets	10	7	(70%)	9	(90%)
Syntax	Invalid Interfaces	20	19	(95%)	19	(95%)
Range	IP out-of-range	25	24	(96%)	25	(100%)
Functional	Incorrect Permissions	20	1	(5%)	7	(35%)
Total		100	72	(72%)	85	(85%)

Moreover, the challenge arises when using GPT-4T without specifying the number of injected faults, resulting in an increased likelihood of generating unintended errors. This emphasizes the importance of refining GPT-4T’s usage parameters to enhance its precision and align it more closely with user expectations.

Finally, as shown by our results, providing correct contexts for functional-related faults remains a non-trivial problem as each functional fault requires a specific context that aligns with the specific functional needs of the user. This calls for efficiently extracting dynamic contexts to make in-context learning more efficient for functional faults.

4.2 Limitation of Existing Validation Tools

Below, we discuss the limitations of current validation tools and validation processes.

4.2.1 Expert-dependent Validation

Existing validation tools like Batfish Fogel et al. (2015) provide validation functions, e.g., network reachability analysis, as basic building blocks for expert-dependent configuration testing and validation. Developing comprehensive validation and test cases using these building blocks requires substantial engineering efforts and expert knowledge. In addition, interpreting the validation outputs from these tools requires domain expertise. Notably, Batfish offers a rather generic response such as “not all lines are recognized” when encountering syntax faults, without pinpointing the exact line causing the fault. As for identifying functional faults, Batfish outputs packet trace which consists of a series of packet acceptance statuses between endpoints: a packet is either “ACCEPTED” or “REJECTED” at each endpoint (e.g., BoarderRouter_1 ACCEPTED, LocalRouter_1 ACCEPTED, etc.), while leaving engineer to judge whether the network configuration is functionally correct or not by following the traces. These drawbacks hamper the efficiency of misconfiguration diagnosis, indicating a potential area for improvement in enhancing the specificity of Batfish’s error reporting to reduce the need for expert intervention.

4.2.2 Validation with LLMs In-the-Loop

As demonstrated via our evaluations, SOTA LLM (GPT-4T) is effective in automatically identifying and correcting certain fault categories, with little human intervention. In contrast, current validation tools such as Batfish rely on human expert-designed test cases for identifying network misconfigurations which leaves correction largely a manual process. Nevertheless, we believe that the inclusion of a validation tool like Batfish as a post-correction checker can help catch hallucinations that lead to invalid syntax or functional faults and subsequently reduce them by refining the prompts. However, the drawbacks of the current expert-dependent validation tools as discussed in Section 4.2.1 hinder their ability to be used with LLMs in the validation process.

Mismatched input/output format: LLMs are primarily text and natural language-driven: (1) natural language as prompt input, and (2) natural language as response output. The text-based input and output formats mismatch with the existing validation tools like Batfish, as those tools require a series of API calls to initiate the validation process and output the results as specific data structures (for Batfish, the result is a Pandas Dataframe). This makes direct interfacing between existing validation tools and LLMs rather difficult, as the validation tools by themselves are not able to efficiently generate prompts to query the LLM, while the LLM cannot directly consume the output of the validation tools.

Providing domain-specific context: Our experiments show that providing the correct domain-driven context and utilizing in-context learning can greatly improve the LLM’s success rate in identifying and correcting difficult faults. On the other hand, in-context learning provides little benefit if the context provided is overly generic, which is reflected in the limited improvement observed in the functional faults category. Unfortunately, existing validation tools are not designed to provide context for the LLM’s consumption as there is no built-in mechanism to summarize the domain-specific context as part of the LLM prompt. As a result, this remains a manual process, making it not scalable.

4.2.3 Summary

In summary, combining LLMs and validation tools presents a promising approach, but the gaps above hinder such an adaptation. For one, enhancing the detailed and LLM-recognizable error reporting capabilities of validation tools could contribute significantly to a more effective and user-friendly configuration validation process with LLMs. Second, providing contexts from the validation tools to the LLM can likely improve the LLM’s fault identification and correcting capabilities, especially on functional faults. Overall, great research opportunities exist for LLM-aided configuration validation with validation tools in the loop. Specifically, two promising directions are the integration of LLM and validation tools and LLM-validation tools codesign, i.e., a validator for LLM and LLM-driven validation.

5 Conclusion

This project characterizes a state-of-the-art (SOTA) LLM’s (GPT-4T) capability for identifying and correcting network configuration faults. We also use Batfish as a post-correction configuration checker. Our results show that while GPT-4T performs well on fault categories involving invalid syntax, it performs poorly on functional faults requiring domain-specific contexts. This project also highlights the limitations of current validation tools (e.g., Batfish) tailored for expert-in-the-loop validations, and provides detailed discussions on potential improvements and future research directions.

6 Individual Contributions

All team members contributed equally to project presentations, project proposals, and final reports. The breakdown of individual contributions is as follows:

- Ze Yang: Network configuration prompter pipeline implementation; prompt designing.
- Shengkun Cui: Fault injector pipeline implementation; Batfish verification pipeline; in-context examples.
- Ziheng (Jack) Chen: Fault injector pipeline implementation; fault designing.

References

- Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Morgan & Claypool Publishers. <https://doi.org/10.2200/S00874ED3V01Y201809CAC046>
- Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track, Las Vegas, NV, USA, December 11 - 15, 2017*, Xiaoyun Zhu and Indrajit Roy (Eds.). ACM, 29–35. <https://doi.org/10.1145/3154448.3154453>

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. [arXiv:2107.03374](https://arxiv.org/abs/2107.03374) [cs.LG]
- Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. 2023. A Survey on In-context Learning. [arXiv:2301.00234](https://arxiv.org/abs/2301.00234) [cs.CL]
- Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel>
- Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, Marcos K. Aguilera, Brian Cooper, and Yanlei Diao (Eds.). ACM, 1–16. <https://doi.org/10.1145/2987550.2987583>
- Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. 2015. ConfValley: a systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 19:1–19:16. <https://doi.org/10.1145/2741948.2741963>
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. [arXiv:2205.11916](https://arxiv.org/abs/2205.11916) [cs.CL]
- Linda Leuschner, Martin Küttler, Tobias Stumpf, Christel Baier, Hermann Härtig, and Sascha Klüppelholz. 2017. Towards Automated Configuration of Systems with Non-Functional Constraints. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal (Eds.). ACM, 111–117. <https://doi.org/10.1145/3102980.3102999>
- Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, HotNets 2023, Cambridge, MA, USA, November 28-29, 2023*. ACM, 196–204. <https://doi.org/10.1145/3626111.3628183>
- David L. Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done About It?. In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*, Steven D. Gribble (Ed.). USENIX. <http://www.usenix.org/events/usits03/tech/oppenheimer.html>
- Russell A Poldrack, Thomas Lu, and Gašper Beguš. 2023. AI-assisted coding: Experiments with GPT-4. [arXiv:2304.13187](https://arxiv.org/abs/2304.13187) [cs.AI]
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. [arXiv:2302.04761](https://arxiv.org/abs/2302.04761) [cs.CL]
- Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. 2005. ACMS: The Akamai Configuration Management System. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*, Amin

- Vahdat and David Wetherall (Eds.). USENIX. <http://www.usenix.org/events/nsdi05/tech/sherman.html>
- Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 735–751. <https://www.usenix.org/conference/osdi20/presentation/sun>
- Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 328–343. <https://doi.org/10.1145/2815400.2815401>
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]
- J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 129–140. <https://doi.ieeecomputersociety.org/>
- Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 619–634. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu>
- Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (Cascais, Portugal) (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 159–172. <https://doi.org/10.1145/2043556.2043572>