CAP: Detecting Network Device Misconfigurations with Context-Aware Prompting of LLMs

XI JIANG, University of Chicago, USA
AARON GEMBER-JACOBSON, Colgate University, USA
NICK FEAMSTER, University of Chicago, USA

Verifying network configurations is critical for ensuring operational stability, but traditional tools force a trade-off between the high manual effort of model checkers and the semantic shallowness of consistency checkers. More importantly, neither class of tools offers interpretable, human-readable explanations to support diagnosis. Large Language Models (LLMs) present a promising alternative by enabling both detection and explanation. However, naive approaches like full-file, partition-based, or chain-of-thought prompting fail to handle the large, interconnected nature of configuration files. We introduce the Context-Aware Prompting (CAP) framework, which enables an LLM to reason about network configurations like a human expert. CAP first identifies all potentially relevant neighboring, similar, and referenced configuration segments. It then engages the LLM in a structured dialogue, where the model requests the specific context it needs before performing a focused analysis. This on-demand approach provides the necessary context for deep reasoning while avoiding overload. Our evaluation on production network configurations demonstrates that CAP reliably detects a wide range of known and previously unknown configuration errors. CAP substantially outperforms existing LLM-based baselines, while achieving performance comparable to state-of-the-art non-LLM tools. CAP is most effective on misconfigurations that require reasoning over references and intricate dependencies.

CCS Concepts: • **Networks** → **Network management**; • **Computing methodologies** → **Machine learning**; • **Security and privacy** → **Network security**.

Additional Key Words and Phrases: Network anomaly detection, large language model, context retrieval, prompt engineering

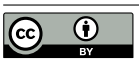
ACM Reference Format:
Xi Jiang, Aaron Gember-Jacobson, and Nick Feamster. 2026. CAP: Detecting Network Device Misconfigurations with Context-Aware Prompting of LLMs. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 2, Article 33 (June 2026), 28 pages. https://doi.org/10.1145/3805631

1 Introduction

Ensuring the stability, security, and performance of networks requires carefully configuring routers, switches, firewalls, etc. A misconfiguration on even a single network device can lead to black holes, unintended network access, inefficient routes, and a range of other issues [25, 63].

Although some configuration tasks are automated—especially in large networks [24, 47, 53, 54, 60]—many (changes to) configurations are manually written by human experts [5]. Network engineers rely on extensive knowledge gained from years of experience with an organization's network(s), vast vendor documentation [2, 3], industry best practices [4], etc. To avoid problems, engineers must consider the interplay between (a change to) a configuration and related aspects of

Authors' Contact Information: Xi Jiang, xijiang9@uchicago.edu, University of Chicago, Chicago, Illinois, USA; Aaron Gember-Jacobson, agemberjacobson@colgate.edu, Colgate University, Hamilton, New York, USA; Nick Feamster, feamster@uchicago.edu, University of Chicago, Chicago, Illinois, USA.



ACM 2476-1249/2026/6-ART33
https://doi.org/10.1145/3805631

devices' configurations. This is challenging for humans, because even a single device's configuration is often hundreds or thousands of lines long with many dependencies between segments [12].

Researchers and practitioners have developed numerous network configuration verifiers to address this challenge. *Model checkers* [1, 6, 9, 11, 23, 28, 51, 55, 57, 71, 72] analyze network-wide forwarding behaviors under various scenarios (*e.g.*, link failures) to detect violations of network objectives (*e.g.*, reachability). These tools rely on algorithms or formalisms that codify protocol- and vendor-specific configuration semantics as well as network-specific forwarding policies. These require significant manual effort to define—only some algorithms and policies can be automatically inferred [15, 31, 56]—and are difficult to define correctly and completely [14, 71]; hence, model checkers have a high overhead to adoption. *Consistency checkers* [1, 22, 29, 30, 34–36, 59], on the other hand, compare configurations across devices or against best-practices and flag deviations as potential issues. While these statistics- or rule-driven approaches require minimal manual effort, they lack semantic depth and the ability to reason about interactions between configuration components—*e.g.*, a router filter's impact on the BGP peers to which it is applied—or intentional deviations—*e.g.*, using a different route filter for a particular peer due to unique forwarding requirements. We also note that model and consistency checkers provide terse explanations for their conclusions—*e.g.*, a single packet that would be forwarded incorrectly or the percentage of configurations that differ—which can be difficult for engineers to interpret and use for further diagnosis or remediation.

Large language models (LLMs) [7, 19] offer a promising avenue to avoid the classic trade-off between semantic depth and ease-of-use in network verification as well as provide human-interpretable explanations of errors. Pretrained on a vast, public corpus of data that includes protocol specifications and vendor documentation, LLMs develop a deep understanding of general network configuration principles. When presented with a specific, previously unseen device configuration at inference time, the LLM applies this broad knowledge to effectively translate the configuration into a reasoned judgment. This process hinges on the model's self-attention mechanism, which mathematically weighs the importance of every token in the configuration relative to every other token. By using attention, the model builds a rich, contextual map—*e.g.*, identifying that a specific router filter defined on line 50 directly impacts a BGP peer defined on line 450. After creating this complex, attention-weighted internal representation, the model's final layers distill it into a simple, probabilistic choice over a small set of output tokens, such as 'correct' or 'incorrect'. This ability to use attention to interpret complex dependencies and then collapse that understanding into a discrete assessment makes LLMs analogous to human experts applying broad domain knowledge to diagnose a specific problem. Furthermore, LLMs have the potential to articulate their reasoning in natural language to help human engineers understand and remediate the errors.

However, recent efforts to leverage LLMs for configuration analysis [16, 44, 45, 65, 69] have missed an important insight: human engineers do not examine entire configurations in a rigid, linear fashion; instead, engineers identify relevant configuration lines and dynamically "jump" to related parts of the configuration. Similarly, LLMs are more effective when provided with *concise, relevant context* that mirrors this human approach. Feeding an LLM the entire configuration at once risks exceeding token limits, and splitting configurations into arbitrary partitions [44] can obscure dependencies between different parts. Furthermore, overloading the model with irrelevant or excessive information can dilute its reasoning capabilities [41, 43, 52], much like a human struggling to recall long-past details. We discuss these issues in more depth in Section 2.3.

The central challenge, therefore, is not only to harness the reasoning capabilities of LLMs, but to do so in a way that produces accurate, scalable, and *human-interpretable* diagnoses over complex, interdependent network configurations. An effective framework must enable an LLM to analyze configurations in a way that is both semantically deep and scalable, mirroring a human expert's ability to dynamically seek out relevant context without being overwhelmed by the sheer volume

of a full configuration file. In this paper, we propose the *Context-Aware Prompting* (CAP) framework, a holistic solution that addresses this challenge through two integrated components:

1. Automatic and Accurate Context Mining: To ground the LLM's reasoning, CAP first performs automated context extraction. We model the configuration's nested structure as a hierarchical tree, where configuration stanzas and parameters (*e.g.*, interface GE1/1/1) are intermediate nodes and terminal commands (*e.g.*, mtu 9000) are leaf nodes. This structure enables the systematic identification of key context categories—such as neighboring, similar, and referenced configuration lines—ensuring a concise yet sufficient corpus is available for analysis. Prior work has explored representing network configurations using structured abstractions such as syntax trees to support analysis and synthesis (*e.g.*, AED [5]). These approaches primarily use structure to encode configurations for static reasoning, often in the form of symbolic constraints over configuration elements. In contrast, CAP uses structure not as an analysis substrate, but as a mechanism for dynamic context retrieval. Specifically, CAP leverages structural relationships to identify and retrieve relevant configuration segments on demand, enabling the LLM to incrementally construct the context needed for reasoning rather than operating over a fixed, pre-encoded representation.

2. Guided Interactive Prompting for Context Management: To prevent context overload and focus the LLM's attention, CAP engages the LLM in a guided, interactive dialogue. While modern LLMs have large context windows, providing excessive or unfocused information is known to degrade reasoning quality [39]. Our framework mitigates this by having the LLM explicitly request only the specific context it needs from the categories mined in the first component. This allows the model to incrementally build its understanding and enables more precise detection. This interaction naturally produces a structured reasoning trace, exposing the intermediate assumptions and contextual evidence used by the model, and making each detection decision transparent and inspectable by human operators.

At a high level, CAP reframes configuration analysis as a context acquisition problem. Rather than requiring the model to reason over an entire configuration or a fixed subset, CAP enables the model to iteratively request and construct the specific context needed for accurate analysis. This combination of structured context mining and interactive prompting allows CAP to support fine-grained reasoning over large, interdependent configurations without incurring the performance degradation associated with context overload. Unlike prior work on configuration synthesis such as AED, which focuses on generating new configurations that satisfy constraints, CAP focuses on diagnosing and explaining misconfigurations by guiding how context is retrieved and presented to an LLM during analysis.

We evaluate CAP on configurations from three diverse networks (Internet2 and two university campuses). Our evaluation demonstrates that (1) CAP accurately detects a wide range of known synthetic and real-world misconfigurations; (2) it can uncover novel, previously unknown issues in production networks, while also highlighting the challenge of false positives with some LLM backends; (3) the choice of LLM significantly impacts reasoning strategies and outcomes; and (4) CAP substantially outperforms existing LLM-based baselines and performs on par with, or better than, state-of-the-art consistency-based tools on a controlled set of complex errors—particularly for misconfigurations that require reasoning over references and cross-configuration dependencies. CAP is designed primarily for pre-deployment configuration verification and change review workflows, rather than live troubleshooting. Accordingly, we focus on misconfigurations that can be identified from device configurations alone, without incorporating live data-plane or control-plane state (*e.g.*, BGP dynamics or link utilization). This design choice avoids reasoning over a large space of operational scenarios and enables CAP to surface broadly applicable configuration errors.

Nevertheless, the interactive nature of CAP allows it to be extended to near-real-time checks through incremental triggering and context caching, which we leave to future work.

2 Motivation

Ensuring network configurations are correct is a notoriously difficult task due to their structural and semantic complexity. Researchers and practitioners have developed numerous network configuration verifiers to help address this issue, but these tools tend to focus on either structure or semantics and do not consider both in tandem. Moreover, these tools provide limited or no human-interpretable explanations of detected errors. Large Language Models (LLMs) offer a promising way to bridge this gap: their ability to capture semantics emerges naturally from learning statistical patterns over large corpora of structured text, enabling them to reason about configuration intent while also producing interpretable explanations. However, effectively using LLMs to identify misconfigurations requires addressing the challenges posed by the large, interconnected nature of network configurations.

In this section, we discuss the structural and semantic complexity of network configurations, the advantages of LLMs compared to existing verifiers, and the challenges that must be addressed to effectively use LLMs to analyze configurations.

```
 1 interfaces {
 2     ge-0/0/1 {
 3         mtu 9000;
 4         unit 0
 5             family ethernet-switching {
 6                 interface-mode trunk;
 7                 vlan members [ 10 20 ];
 8                 filter { input DEMO-FILTER; }
 9 ...
10     ge-0/0/2 {
11         mtu 1500;
12         unit 0 {
13             family ethernet-switching {
14                 interface-mode access;
15                 vlan-id 10;
16                 filter { input DEMO-FILTER; }
17 ...
18 protocols {
19     bgp {
20         group EBGP-PEER {
21             type external;
22             peer-as 65002;
23             neighbor 10.0.2.2;
24 ...
25 firewall {
26     family ethernet-switching {
27         filter DEMO-FILTER {
28             term ALLOW-INTERNAL {
29                 from { source-address 10.0.0.0/16; }
30                 then accept;
31             }
32 ...
```

Fig. 1. Example device configuration in Junos OS

2.1 Network configuration complexity

Structural complexity. Configurations possess a formal, nested structure—as shown in Figure 1—where settings are logically nested within specific blocks—*e.g.*, an MTU is specified (line 3) within an interface block (lines 2-8). However, configurations are also woven together by a web of critical, non-hierarchical relationships where part of a device's configuration refers to another part of the configuration—*e.g.*, an interface block references a packet filter (line 8) defined in a separate block (lines 27-31)—or another device's configuration—*e.g.*, a BGP neighbor statement (line 23) references the address configured in another device's interface (not shown) [12, 48]. Furthermore, configurations are often immense—typically hundreds to thousands of lines per device [22, 32]—and differ syntactically between vendors [12, 59, 71]. Engineers often strive to keep configurations relatively uniform—both within and across devices—to help cope with this complexity [12, 20, 29, 35], but deviations are inevitable.

Semantic complexity. Configurations typically contain directives for a multitude of protocols—*e.g.*, IEEE 802.1Q, STP, OSPF, IS-IS, MPLS, BGP—that operate in a distributed fashion and each have their own features and parameters [48]. Furthermore, there are complex dependencies and interactions between these protocols [6, 37, 51].

2.2 Why LLMs?

Recent work has shown Large Language Models (LLMs) are a promising avenue for generating and analyzing configurations [16, 40, 44, 45, 50, 65, 66, 69]. The suitability of LLMs for configuration tasks

is analogous to their well-established success in code analysis [38, 46, 61] and generation [21, 33, 67]. In particular, LLMs' suitability for these tasks stems from: (1) LLMs' broad "understanding" of network configuration developed through extensive pretraining on vast datasets, and (2) LLMs' context-aware reasoning that combines user prompts (*e.g.*, configuration snippets and questions) with pretrained knowledge to generate responses (*e.g.*, assessments of configuration correctness).

Broad Knowledge Developed Through Pretraining: LLMs develop a deep, contextual understanding of network configurations during pretraining on extensive datasets that include protocol specifications, vendor documentation, and Q&A forum posts [7, 62]. This process allows them to learn configurations' rigid syntax, hierarchical organization, and critical long-range dependencies. The Transformer architecture [64] is the primary reason for this effectiveness, as its core mechanisms are designed to model exactly these properties. Specifically, positional encodings allow the model to learn the strict sequential order and formal hierarchy, enabling it to understand, for instance, that an '`mtu`' setting belongs to a specific '`interface`' block. In parallel, the multi-head self-attention mechanism is crucial for capturing the non-hierarchical, long-range dependencies, as it allows every token to directly attend to every other token, regardless of their distance. This enables the model to connect an '`input DEMO-FILTER`' statement to the '`filter DEMO-FILTER`' block defined tens or hundreds of lines away, creating a probabilistic repository of knowledge akin to a human engineer's conceptual model of a configuration.

Integrating prompts and pretrained knowledge for context-aware reasoning: When a user provides configuration snippets and instructions in a prompt (*e.g.*, "evaluate this filter for conflicts"), an LLM combines its pre-trained knowledge with the contextual information in the prompt. The self-attention mechanism enables the model to cross-reference tokens in the prompt with its stored knowledge, *i.e.*, probabilistic relationships learned during training. By estimating the likelihood of each token or sequence ($p(\text{token}_i \mid \text{token}_1, \ldots, \text{token}_{i-1})$), the model highlights deviations from high-probability patterns, flagging potential misconfigurations such as overlapping address ranges or misplaced `deny` directives. This integration mirrors how a human engineer draws on their knowledge and experience to evaluate configurations and apply abstract principles to specific cases.

Advantages over non-LLM-based network verifiers: The features described above make LLM-based approaches easier to use and more broadly applicable than traditional network verifiers based on model checking or consistency checking, particularly when diagnostic insight is required.

Model checkers [1, 6, 9, 11, 23, 28, 51, 55, 57, 71, 72]—which create and analyze logical models of network behavior to verify compliance with network requirements—heavily rely on the manual specification of algorithms and policies. These are hard to define correctly and completely [14, 71], even with the assistance of other tools [15, 31, 56]. Moreover, when violations are detected, model checkers typically provide limited, low-level counterexamples that require substantial expert effort to interpret and diagnose. In contrast, LLMs require no human-defined rules for specific tasks, leveraging their ability for in-context learning [17] to adapt to the provided configuration. This makes LLMs better suited for analyzing configurations involving vendor-specific nuances or un(der)specified network requirements. For example, an LLM trained on public documentation may correctly interpret a Juniper-specific 'apply-groups' directive—a non-standard inheritance mechanism—that a generic, protocol-based model checker would miss, especially when the framework provides it with the concise, relevant context of the group being applied. One caveat is that LLMs' reliance on probabilistic reasoning prevents them from computing precise forwarding behaviors in some scenarios, so networks with well-specified or stringent requirements may choose to use model checkers in tandem with LLMs.

Consistency checkers [1, 22, 29, 30, 34–36, 59]—which analyze patterns in the configurations themselves—identify potential errors by flagging deviations from pre-defined or inferred norms.

This approach scales well and is effective for surfacing structural anomalies, but it treats all deviations uniformly and does not reason about whether a deviation is intentional or semantically justified. As a result, consistency checkers offer limited diagnostic guidance beyond highlighting differences. In contrast, LLMs integrate their pre-learned understanding of common practices with the specific configuration context provided in the prompt. By understanding the configuration as an interconnected system, an LLM can determine if a deviation in one part is a valid customization that is justified by a dependency in another. For example, a consistency checker might flag an interface with an MTU of 1500 as an anomaly if all other interfaces use an MTU of 9000. An LLM, however, when provided with context showing this interface connects to an external network, can use its broad knowledge to reason that 1500 is not a misconfiguration, but a necessary and correct setting for ensuring compatibility with the external peer. Furthermore, LLMs can produce context-grounded, human-readable explanations that are valuable for further diagnosis and remediation.

2.3 Challenges of using LLMs for network verification

Effectively identifying misconfigurations using LLM's broad knowledge and context-aware reasoning requires addressing the challenges posed by the complexity of network configurations. Simply providing the entire file as input is often infeasible due to token limits and, more importantly, the performance degradation that occurs from context overload [39]. This has led to several intuitive but ultimately flawed alternatives.

One might first consider advanced prompting techniques like Chain-of-Thought (CoT) to improve reasoning [18, 68, 70]. However, CoT is a reasoning strategy, not a context retrieval mechanism. It can only reason about the information already present in the prompt. If a critical policy definition is thousands of lines away and thus cannot be included in the context window, CoT is powerless to access it; its chain of thought will be based on incomplete information, leading to flawed analysis.

A second approach, partition-based prompting [44], directly addresses token limits and context overload by breaking the configuration into smaller chunks. While this allows the model to process the entire file, it guarantees context fragmentation. For example, an import policy defined in one segment may silently discard routes that an export policy defined elsewhere is intended to advertise; a model that analyzes each partition in isolation would miss this cross-sectional conflict entirely. This method is also fundamentally incompatible with CoT, as it physically severs the long-distance contextual links that a step-by-step reasoning process requires.

A third but similarly flawed approach is to analyze only configuration changes (diffs). While seemingly efficient, this method suffers from the same context fragmentation problem as partitioning. A change that adds a single line—such as applying a new packet filter to an interface—is meaningless without the context of both the filter's definition and the other properties of that interface, some of which may be located far away from the change itself.

Ultimately, these naive approaches fail because they cannot maintain a global, interconnected view of the configuration, which is precisely what a human expert relies on. An effective LLM-based tool must therefore emulate this expert behavior by intelligently gathering all context relevant to a specific configuration statement under review. This core need to manage and provide crucial, local or distant context–whether to resolve dependencies, learn design patterns, or understand local interactions—directly motivates our framework. More fundamentally, this highlights that the central challenge is not only performing reasoning over configurations, but determining what information is necessary to support that reasoning. This aspect—selectively acquiring the right context—is not addressed by prior structure-based or constraint-based approaches. CAP directly addresses this gap by modeling configuration analysis as a context acquisition problem.

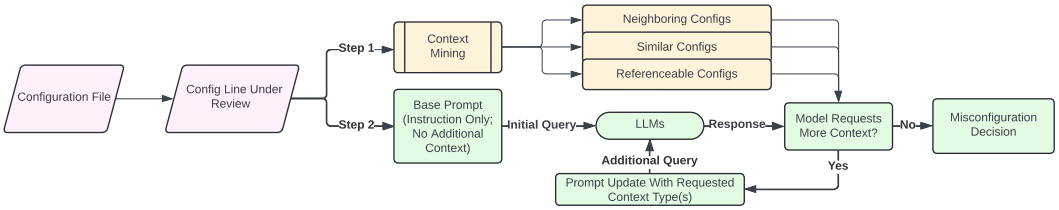


Fig. 2. CAP system overview.

3 Overview

We introduce a tool that tackles the aforementioned challenges via a *context-aware prompting* (CAP) mechanism. This approach strategically manages and incorporates only the relevant parts of configurations for each line under review, ensuring that LLMs obtain the required context without succumbing to overload. At the same time, this line-by-line methodology provides maximum granularity and emulates the workflow of a human expert, who typically focuses on a single statement to trace its specific dependencies and impacts. To operationalize this approach, CAP is composed of two main components, as illustrated in Figure 2:

1. Context Mining: In this component, CAP mines relevant context for a given configuration line. The core challenge lies in the complex nature of "relevance" itself, which is not defined by mere proximity in a file. To emulate an expert's intuition, our definition of relevance is comprehensive, designed to capture a configuration's local interactions through physically adjacent (neighboring) lines, its adherence to network-wide patterns via functionally similar lines, and its distant dependencies through syntactically linked (referenced) policies. This mining process is guided by the three primary characteristics of network configurations: their hierarchical structure, relative uniformity [29, 35], and abundance of references [12].

2. Guided Prompting: Once the relevant context has been mined, the online component engages the LLM through an interactive prompting process. Instead of overwhelming the model with all the extracted context at once, CAP interacts with the LLM in a guided manner.

We discuss these two components in detail in Sections 4 and 5, respectively.

4 Context Mining

This section details the design of our context mining engine. We first formalize the structure of configuration files and introduce the concept of a *fully-qualified configuration statement* (FQCS). Then, based on the characteristics of configurations discussed in Section 2.1, we define the three principal types of context our engine extracts: (1) neighboring configuration, (2) similar configuration, and (3) referenced configuration. Figure 3 presents examples of the types of context we consider in this work, each providing unique insights that contribute to the evaluation of a FQCS.

While prior systems such as AED also model configurations as syntax trees, their use of structure is fundamentally different. In AED, syntax trees serve as a foundation for constructing symbolic encodings and reasoning about configuration updates via constraint solving. In contrast, CAP uses the hierarchical structure to define context mining primitives (neighboring, similar, referenced), which are then used to dynamically supply only the information required for LLM-based reasoning. This distinction reflects a shift from structure as a representation for static analysis to structure as a mechanism for guiding context retrieval during interactive reasoning.

4.1 Representing configurations

Since configurations have a structured, hierarchical format (Section 2.1), we represent each device's configuration as a tree (T). Each node (V) corresponds to a specific category (*e.g.*, '`interfaces`',

Configuration Statement Under Review *(FQCS for line 8)*

```
interfaces -> ge-0/0/1 -> unit 0 -> family ethernet-switching -> filter -> input = DEMO-FILTER
```

Extracted Neighboring Configurations *(Other settings for the same interface unit)*

```
interfaces -> ge-0/0/1 -> unit 0 -> family ethernet-switching -> interface-mode = trunk
interfaces -> ge-0/0/1 -> unit 0 -> family ethernet-switching -> vlan members = [10, 20]
```

Extracted Similar Configurations *(The same parameter used on a different interface)*

```
interfaces -> ge-0/0/2 -> unit 0 -> family inet -> filter -> input = DEMO-FILTER
```

Extracted Intra-Device Referenced Configurations *(Finding the definition of the referenced 'DEMO-FILTER')*

```
firewall -> family inet -> filter DEMO-FILTER -> term ALLOW-INTERNAL -> from -> source-address = 10.0.0.0/16
firewall -> family inet -> filter DEMO-FILTER -> term ALLOW-INTERNAL -> then = accept
```

Configuration Statement Under Review *(FQCS for line 23)*

```
protocols -> bgp -> group EBGP-PEER -> neighbor = 10.0.2.2
```

Extracted Cross-Device Referenced Configurations *(from Router R2)*

```
protocols -> bgp -> group EBGP-PEER -> neighbor = 10.0.2.1
```

Fig. 3. Examples of the context mined by CAP for the sample configuration in Figure 1.

'`filter`'), instance (*e.g.*, '`ge-0/0/1`', '`DEMO-FILTER`'), or parameter (*e.g.*, '`mtu`', '`input`') within the configuration. A complete path $P = \{\top \rightarrow V_1 \rightarrow V_2 \rightarrow \cdots \rightarrow V_k = v_k\}$ from the root ($\top$) to a leaf (V_k) represents a single, *fully-qualified configuration statement* (FQCS). For example, the path listed at the top of Figure 3 corresponds to line 7 in Figure 1, where the leaf node ($V_k = v_k$) represents the configuration parameter ('`input`') and its associated value ('`DEMO-FILTER`'), and the preceding nodes ($V_1, V_2, \ldots, V_{k-1}$) represent the configuration segments the parameter is nested within ('`interfaces`', '`ge-0/0/1`', ..., '`filter`').

4.2 Neighboring configurations ($N_m(P)$)

Neighboring configurations are statements that are structurally close to the FQCS under review, typically within the same functional block. They provide the immediate, local context necessary to understand the holistic state and intended behavior of a specific entity, such as an interface or routing protocol. For instance, when analyzing the statement '`vlan members [10 20]`', its neighboring context is crucial. A neighboring statement like '`interface-mode trunk`' confirms that the VLAN configuration is active and relevant, while its absence would indicate a misconfiguration, as it is invalid to define multiple VLAN members on a non-trunking port.

Formally, we define the neighboring configurations of path P as the set of paths P' that share a common ancestry with P up to depth m. (V_i and V'_i are the i-th node in P and P', respectively.)

$$N_m(P) = \{P' \in T \mid V'_i = V_i \text{ for } 1 \leq i \leq m, \text{ and } P' \neq P\}$$

The depth parameter m controls the scope of the context. A larger m results in a tighter, more relevant neighborhood. A simple and effective choice is to set $m = k - 2$, meaning we extract all sibling configurations that share the same direct parent and grandparent. This captures the most immediate context without including potentially irrelevant data from higher up in the hierarchy. While we use this default for our evaluation, we acknowledge that a full study on the impact of different values of m is a valuable area for future work.

4.3 Similar configurations ($S_m(P)$)

Similar configurations involve the same type of parameter but applied in slightly different contexts (*e.g.*, the input filter settings on all other interfaces across the network). Comparing these helps detect inconsistencies—which, as discussed in Section 2.2, may or may not be intentional.

Formally, we define the similar configurations of path P as the set of paths P' that share a common ancestry with P up to depth m and terminate with the same parameter node V_k, but differ

in the path between m and k.

$$S_m(P) = \{P' \in T \mid V'_i = V_i \text{ for } 1 \le i \le m, V'_k = V_k, \text{ and } P' \neq P\}$$

The choice of the depth parameter m allows for a crucial trade-off between the breadth and relevance of the comparison. Our default choice is to set $m = 1$, which typically represents a major functional block (like '`interfaces`' or '`protocols`'). This provides a meaningful balance, which can be illustrated by considering the alternatives:

- A choice of $m = 0$ would be too broad, because it encompasses a device's entire configuration. For example, searching for a parameter like '`then`' would incorrectly group together packet filter actions and route filter actions, which are semantically different.
- A choice of m that is too large (*e.g.*, $m = k-2$, matching on a specific interface like '`ge-0/0/1`') would be too restrictive. A search for similar filter settings would find no configurations on other interfaces, defeating the purpose of the comparison.

The flexibility of m also helps manage comparisons within large functional blocks. For instance, in a Juniper configuration, '`protocols`' is a V_1 node containing specific protocols like '`bgp`' and '`ospf`' as V_2 nodes. Our default of $m = 1$ would compare settings across BGP and OSPF. This may seem odd, but in practice, the final parameter names (V_k) are typically distinct between protocols, so irrelevant comparisons are rare. However, for a more targeted analysis, a user could set $m = 2$ to find similar settings only within BGP or within OSPF. While our default provides a reasonable starting point, to suit different analysis cases, CAP supports the flexibility to choose optimal default values for m for different parameter types.

4.4 Referenced Configurations ($R_{intra}(P)$ & $R_{cross}(P)$)

Referenced configurations are the definitions for user-defined identifiers—custom labels created by an administrator to name a specific object, such as a packet filter, route filter, or BGP peer group. Mining this context is crucial for understanding dependencies. The nature of these references can be either internal to a single device's configuration or span across a pair of devices on the network (Section 2.1). CAP is designed to handle both.

Intra-Device References. This is the most common form, where a value in one part of a device's configuration refers to an object defined elsewhere in the device's configuration. For example, the configuration statement '`input DEMO-FILTER`' (line 8 in Figure 1) references the block defining the rules for the packet filter `DEMO-FILTER` (lines 27-31). This equates to finding all paths where the reference value is an intermediate node instead of a leaf value—*e.g.*, all paths of the form '`firewall → family ethernet-switching → filter DEMO-FILTER → ...`'.

Formally, we define the intra-device referenced configurations of path P, which ends with value v_k, as the set of paths P' that contain v_k as an intermediate node. (V_j is the leaf node in P'.)

$$R_{intra}(P) = \{P' \in T \mid P' = \top \to \cdots \to v_k \to \cdots \to V_j = v_j\}$$

A primary challenge in extracting intra-device referenced configurations is the risk of incorporating irrelevant context. This risk stems from the difficulty of distinguishing between *primitive values* and *reference values*. Primitive values are self-contained values, such as integers (*e.g.*, an MTU of '`1500`'), enumerations (*e.g.*, interface mode '`trunk`'), and IP addresses. In contrast, reference values are custom labels (or *object identifiers*) created by an administrator for specific, user-defined entities, such as the packet filter name '`DEMO-FILTER`'.

This distinction is critical because it dictates our context-mining strategy: a reference value acts as a pointer, implying that a full definition exists elsewhere in the configuration and *must* be retrieved to understand its meaning. Misidentifying a reference value as a primitive value would cause the system to fail to search for its definition, likely leading it to miss dependency-related

errors. Conversely, mistakenly treating a primitive value as a reference value would trigger a pointless and computationally expensive search for a non-existent or mismatched definition.

To address this challenge, we initially adopted a heuristic approach based on existence checks within the configuration's hierarchical tree structure. We assumed that any parameter value appearing as an intermediate node elsewhere in the configuration must be user-defined. However, this strategy proved fragile; it could be misled by coincidental naming and lacked a true understanding of the configuration's grammar and semantics. We then explored a majority-voting mechanism designed to enforce consistency by assuming that a given parameter (*e.g.*, `timeout`) should consistently be assigned either a primitive value or a reference value. But this method also suffered from a critical limitation: it could not distinguish between identical values used in different semantic contexts. For example, when encountering '`vlan-id 1000`', a naive search for the definition of the reference value '`1000`' within the configuration may erroneously retrieve '`filter 1000`'—a packet filter unrelated to VLAN definitions—leading to incorrect or incoherent interpretations.

To overcome these limitations, CAP replaces brittle heuristics with an LLM-powered solution that simultaneously addresses both classification and resolution. Specifically, CAP constructs a single prompt for each configuration statement, instructing the LLM to perform two coupled tasks:

(1) **Classify the Value:** Determine whether the parameter value is a primitive value or a reference value that points to another configuration object.
(2) **Generate a Disambiguation Key:** If the value is classified as a reference, produce a "unique contextual keyword" drawn from the expected definition path. This key serves to disambiguate overlapping or overloaded identifiers.

This design leverages the LLM's deep understanding of vendor-specific syntax to resolve ambiguity in a single, efficient step. For instance, when analyzing the line '`... -> input = 1000`', the LLM might return a JSON object such as: `{"isUserDefined": true, "disambiguationKeyword": 'filter'}`. Equipped with this output, the framework's mining engine gains both intent and context: it not only recognizes that a reference lookup is required, but also knows precisely how to constrain the search. By filtering for configuration paths that include both the target value ('`1000`') and the disambiguation keyword ('`filter`'), CAP accurately retrieves the intended definition while ignoring unrelated objects (*e.g.*, VLANs). This unified, multi-part query enables robust handling of ambiguous identifiers—without relying on brittle, hand-crafted rules.

Cross-Device References. A more complex challenge arises when a configuration line refers to another device on the network. This is common in distributed protocols such as BGP, OSPF, and VPNs. For example, consider the BGP configuration statement '`neighbor 10.0.2.2`' (line 23 in Figure 1). The value '`10.0.2.2`' references a peer router (*R2*). To fully verify this configuration, CAP must examine not just the configuration for *R1*, but also the configuration of the referenced peer, *R2*. This introduces the need for cross-device reference resolution.

CAP handles such cases through a static, rule-based three-step process:

(1) **Identify Cross-Device Reference:** Using a manually maintained rule set, CAP identifies that certain patterns—such as '`neighbor <ip>`' in BGP—imply cross-device linkage. In this case, CAP detects that '`10.0.2.2`' refers to another device. Although this is currently hardcoded, the set of rules is small and stable across common routing and tunneling protocols.
(2) **Locate Target Device:** CAP then determines which configuration file corresponds to the device at '`10.0.2.2`'. To do this, each configuration tree T is statically annotated with an *identifier set* $ID(T)$, which includes all known identifiers belonging to that device. These identifiers are extracted from commands such as '`host-name R2`' and '`address 10.0.2.2/24`'. For example, the configuration for device *R2* might include:

```
system {
    host-name R2;
```

```
...
    family inet {
        address 10.0.2.2/24;
```

From this, CAP populates $ID(T_{R2}) = \{R2, 10.0.2.2\}$ and concludes '`10.0.2.2`' refers to *R2*.

(3) **Find Reciprocal Configuration:** To ensure symmetric configuration, CAP searches within T_{R2} for lines that refer back to *R1*. In particular, CAP searches T_{R2} for any path that contains (1) a value matching an identifier in $ID(T_{R1})$, and (2) a matching *peer keyword*, such as '`neighbor`', which governs the semantics of the peer relationship. For example, a valid reciprocal line in T_{R2} might be '`neighbor 10.0.2.1`'. A match is only valid if both the identifier and the peer keyword match the original line on *R1*. The *peer keyword* of a path P, denoted peerkw(P), is the command keyword that directly governs the cross-device reference.

Formal Definition. Let T_A be the configuration tree of the current device and T_B that of the referenced device. Let $ID(T_A)$ and $ID(T_B)$ be their respective identifier sets, containing all statically extracted IPs and hostnames. Let $P \in T_A$ be a path containing a leaf value v_k that matches some $id \in ID(T_B)$. Then the set of cross-device references is defined as:

$$R_{cross}(P) = \{P' \in T_B \mid \exists\, id \in ID(T_A)\ s.t.\ id \in P' \text{ and } peerkw(P') = peerkw(P)\}$$

This ensures that a path P' in the target device T_B refers back to the original device T_A both by identifier and within the same peer construct. The total reference set $R(P)$ includes both intra-device references (within T_A) and these cross-device reciprocal references. This resolution process is essential for detecting asymmetric misconfigurations—such as a BGP session configured on one side but missing on the other—and ensures consistency across distributed protocol deployments.

4.5 Composable Context Primitives

It is important to note that the three principal context types—neighboring, similar, and referenced—serve as atomic building blocks, or *primitives*. A key strength of our framework is the ability to compose these primitives to form more complex and powerful context queries. For example, we can compose $R(N(P))$, which translates to finding the *definitions* of values that appear in configuration lines *neighboring* the line under review. This compositional nature gives CAP the flexibility to emulate the multi-step, intuitive reasoning of a human expert.

5 Guided Interactive Prompting

While extracting accurate and relevant context is essential, it is equally important to ensure that the LLM can effectively process it. Rather than presenting all available information at once, CAP adopts a guided, interactive prompting framework that allows the model to request context incrementally. Supplying only the most pertinent information at each stage helps maintain focus, reduce unnecessary distractions, and improve overall accuracy.

This staged approach enhances performance in three key ways. First, it enables the model to stay aligned with the specific issue under review. Second, by gradually incorporating related information, the model builds a coherent understanding of the configuration [42, 58], which is critical for identifying subtle issues such as dependency conflicts or policy misapplications. Third, limiting prompt size minimizes token overhead, resulting in faster and more stable outputs. For example, when analyzing a multi-layer firewall policy, CAP may first present the rule under review, then selectively add nearby rules or referenced definitions as needed. Unlike conventional prompt-chaining, which provides fixed context in a sequence of static prompts, CAP dynamically expands the context based on the model's evolving needs—enabling more efficient and targeted analysis.

1. Initial Prompting: The process begins not by asking for a misconfiguration analysis, but by first asking the LLM what information it needs to perform one. The initial prompt provides the LLM with only the specific configuration line under review and a menu of the available, pre-mined context categories (Neighboring, Similar, and Referenced). The LLM is instructed to respond with a JSON object specifying which, if any, of these context types it requires to make an informed decision. This first step also delegates the task of differentiating between pre-defined and user-defined values to the LLM. Appendix B Figure 5 shows an example of this initial query for an MTU configuration line. The structured JSON response is a key design element. The inclusion of a "reason" field in both the context request and the final analysis is crucial for transparency and explainability. By requiring the LLM to articulate why it needs more context or why it has flagged an error, the framework provides a clear audit trail of the model's "thought process." This allows a human expert to quickly understand and validate the system's conclusions, which is essential for building trust.

2. Contextual Options: In the initial prompting, CAP also offers the model the option to request additional context based on its understanding of the configuration line and the misconfiguration detection request. These options are based on our categorized context types, including: neighboring ($N(P)$), similar ($S(P)$) , and referenceable contexts ($R(P)$), as well as referenceable contexts on neighboring configurations ($R(N(P))$). By allowing the model to choose which context to receive, CAP ensures that the LLM is only given information that is most relevant to the detection task at hand, reducing the risk of context overload. Appendix B Figure 6 illustrates an example request response where the LLM requests neighboring context and referenceable contexts on neighboring configurations after the initial prompt to aid in analyzing the MTU setting.

3. Context Provision: After the model specifies its context requirements in the initial step, the framework retrieves that exact information from the mined corpus. CAP then constructs the second and final prompt, which includes the specific context the LLM asked for. Appendix B Figure 7 provides an example of this prompt, where the requested neighboring and similar configurations are provided to the model.

4. Final Analysis: With this tailored, context-rich prompt, the LLM then performs its final analysis. The model's output is a structured decision indicating whether a misconfiguration was found, and if so, providing a reason based on the provided information. Appendix B Figure 8 shows an example of the final misconfiguration decision regarding an incorrect MTU value, delivered after the model received the context it requested.

6 Evaluation

To evaluate the effectiveness of CAP, we assess its performance across a range of real-world configurations. Our evaluation is structured around four key research questions:

(1) Can CAP accurately detect known misconfigurations in controlled and real-world settings?
(2) Can CAP identify novel, previously unknown issues in large-scale networks?
(3) What types of contextual information do different LLMs request when diagnosing issues, and how does this affect their performance?
(4) How does CAP's detection capability compare to existing state-of-the-art verification tools?

6.1 Evaluation Setup

Datasets: Our evaluation uses three distinct sets of network configurations drawn from a nation-wide research and education network (Internet2), a medium-scale campus network (Campus A), and a large-scale campus network (Campus B). For each environment, we analyze a representative subset of routers rather than exhaustively evaluating every device. Exhaustive analysis across

Dataset	Vendor(s)	No. Devices Sampled	Total No. of Devices	Device Roles	Avg. Config Length (lines)
Internet2	Juniper	17	40+	Core & Aggregation Routers	18,944
Campus A	Aruba	8	190+	Access Switches	5,563
Campus B	Cisco, Juniper	5	1,600+	Distribution & Aggregation Routers	6,261

Table 1. Overview of Network Configurations Used in Evaluation.

Model	Context Window	Max Output	Training Cutoff
GPT-4o [7]	128,000	4,096	Oct. 2023
Claude Sonnet 4.5 [10]	200,000	64,000	Jul. 2025
DeepSeek-V3 [19]	128,000	8,000	Jul. 2024
Gemini 2.5 Pro [26]	1,048,576	65,535	Jan. 2025
LLaMA 4 Maverick [49]	1,000,000	8,192	Aug. 2024

Table 2. Large language models used in our evaluation, including their context window sizes, maximum output lengths, and training cutoffs.

all routers is computationally expensive and unnecessary, as configurations within a single administrative domain typically follow a small number of recurring roles and structural patterns, causing misconfiguration behaviors to repeat across devices [13]. Accordingly, our goal is to assess CAP's ability to reason about complex, interdependent configuration semantics rather than to enumerate every repeated instance of a pattern, and sampling is used solely to make evaluation tractable. In production, CAP is designed to run incrementally on configuration changes and can be applied independently to all devices without modification. Importantly, although misconfiguration detection is evaluated on a sampled subset of routers, context extraction is not similarly restricted: cross-device context ($R_{\text{cross}}(P)$) is extracted from the entire network, enabling CAP to reason over global network context even when analyzing individual configuration lines.

We emphasize that campus networks present a particularly challenging setting for configuration analysis. Unlike large-scale data center or ISP networks—which are often highly engineered for uniformity and rely heavily on centralized automation [8, 27]—campus networks evolve organically over long periods, incorporate heterogeneous vendors and protocols, and contain numerous localized exceptions driven by administrative, security, and research requirements [13]. These characteristics make campus configurations structurally diverse and semantically complex, providing a demanding testbed for context-aware misconfiguration detection.

Querying LLMs:

We evaluate CAP using multiple state-of-the-art LLMs: GPT-4o [7], Claude Sonnet 4.5 (Claude) [10], DeepSeek-V3 (DS) [19], Gemini 2.5 Pro (Gemini) [26], and LLaMA 4 Maverick (LLaMa) [49]. Table 2 summarizes the key characteristics of each model, including context window size, maximum output length, and training cutoff. We use different subsets of these models across experiments, depending on the task and analysis objectives. We use the official chat completions API provided by each LLM, which is inherently stateless and thus ensures reproducibility, as each request is executed independently. To support multi-step diagnosis, our framework emulates a stateful interaction by explicitly reconstructing the full, ordered conversation history for every API call, including prior prompts, model context requests, retrieved context, and intermediate reasoning steps.

6.2 Can CAP Detect Known Misconfigurations?

We first evaluate CAP's effectiveness in detecting misconfigurations with a known ground truth. This is a crucial baseline for establishing accuracy. We conduct this analysis in two distinct settings: first, using synthetically generated but realistic errors injected into production configurations from Internet2, and second, using real, known misconfigurations from the Campus A network. For

the synthetic evaluation, we include configurations containing both single and multiple injected misconfigurations to ensure coverage of a range of realistic error scenarios.

Detecting Synthetic Misconfigurations in Internet2: To evaluate CAP in a controlled yet operationally realistic setting, we obtained snapshot configurations from the Internet2 backbone network, consisting of Juniper devices, and injected a diverse set of synthetic configuration updates. In total, we introduced 58 synthetic misconfigurations spanning a wide range of error types and structural characteristics, including **missing configuration lines**, **incorrect or undefined references**, and **replicated configuration stanza(s)**. Specifically, missing-line misconfigurations arise when required configuration statements are omitted, resulting in incomplete or implicitly incorrect behavior despite otherwise valid syntax. Reference-based misconfigurations involve references to configuration objects (*e.g.*, groups, policies, or interfaces) that are incorrect, undefined, or inconsistent with their intended definitions, often requiring resolution across multiple configuration sections. Replicated stanza(s) misconfigurations occur when configuration stanzas are duplicated across devices/configurations but contain subtle inconsistencies or errors, leading to divergent behavior despite apparent structural similarity. These misconfigurations span multiple functional domains such as packet filters, route filters, interfaces, routing protocols, and group-based configurations, and are derived from issues reported in prior studies [23, 35] as well as operator experience. A detailed list of injected misconfigurations is provided in Appendix Table 9.

To assess not only detection capability but also robustness against spurious alerts, for each injected misconfiguration we additionally constructed a corresponding correct configuration update that preserves intended semantics and does not introduce errors. This results in a total of 116 synthetic configuration updates, evenly split between misconfigurations and correct changes.

Many injected misconfigurations require non-local reasoning across multiple configuration blocks or inherited configuration structures (*e.g.*, Juniper's `apply-groups` mechanism), and therefore cannot be reliably identified by inspecting individual configuration lines in isolation. This setting allows us to assess whether CAP can effectively retrieve and reason over the relevant contextual information needed to resolve such dependencies, while also avoiding incorrect diagnoses when similar structural patterns arise in correct configurations.

For this case study, we evaluate CAP using two widely deployed LLM backends, **Claude** and **GPT-4o**, which represent state-of-the-art models with complementary strengths in structured, code-oriented analysis and general-purpose reasoning. Table 3 summarizes detection performance stratified by misconfiguration characteristics. For each characteristic, we separately report results for cases where the characteristic is present ("Yes") and absent ("No"). A misconfiguration is marked as detected, *i.e.*, true positive, *only if CAP's decision and reasoning align with the true cause of the error*. Metrics are reported as rate-based quantities computed within each subset, including true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), and false negative rate (FNR). Table 3 presents detection performance stratified by key misconfiguration characteristics, reporting rate-based metrics separately for cases in which each characteristic is present or absent. Overall, the results highlight substantial variation in CAP's effectiveness depending on the structural signals available in the configuration.

Misconfigurations that involve explicit references are the most reliably detected by both models. When references are present, Claude achieves a true positive rate of 84.6% and GPT-4o achieves 69.2%, with false positive rates below 8% in both cases. These results indicate that references provide strong relational cues that enable CAP to reason across configuration elements and correctly associate errors with their underlying causes. Nevertheless, reference-based reasoning is not sufficient in all cases: both models still exhibit non-trivial false negative rates, particularly for GPT-4o, reflecting residual ambiguity even when cross-links are available. In contrast, reference-free misconfigurations

Model	Metric	Missing line(s)		Involves reference		Stanza replicated	
		Yes	No	Yes	No	Yes	No
Claude	TP (TPR)	7 (31.8%)	26 (72.2%)	11 (84.6%)	22 (48.9%)	28 (70.0%)	5 (27.8%)
	FP (FPR)	2 (9.1%)	2 (5.6%)	1 (7.7%)	3 (6.7%)	2 (5.0%)	2 (11.1%)
	TN (TNR)	20 (90.9%)	34 (94.4%)	12 (92.3%)	42 (93.3%)	38 (95.0%)	16 (88.9%)
	FN (FNR)	15 (68.2%)	10 (27.8%)	2 (15.4%)	23 (51.1%)	12 (30.0%)	13 (72.2%)
GPT-4o	TP (TPR)	8 (36.4%)	21 (58.3%)	9 (69.2%)	20 (44.4%)	24 (60.0%)	5 (27.8%)
	FP (FPR)	5 (22.7%)	1 (2.8%)	1 (7.7%)	5 (11.1%)	2 (5.0%)	4 (22.2%)
	TN (TNR)	17 (77.3%)	35 (97.2%)	12 (92.3%)	40 (88.9%)	38 (95.0%)	14 (77.8%)
	FN (FNR)	14 (63.6%)	15 (41.7%)	4 (30.8%)	25 (55.6%)	16 (40.0%)	13 (72.2%)
	Total	44	72	26	90	80	36

Table 3. Detection performance by misconfiguration characteristics for Claude and GPT-4o. For each category, results are reported separately for cases where the characteristic is present (Yes) and absent (No). Percentages correspond to rate-based metrics computed within each subset: TPR = $\frac{TP}{TP+FN}$, FPR = $\frac{FP}{FP+TN}$, TNR = $\frac{TN}{TN+FP}$, and FNR = $\frac{FN}{FN+TP}$.

are substantially harder to detect. When references are absent, true positive rates drop below 50% for both models, and false negatives become the dominant error mode. While false positive rates remain relatively low, the high miss rates suggest that isolated configuration statements often lack enough contextual evidence for the models to confidently determine correctness.

Misconfigurations involving missing configuration lines present a distinct challenge. When lines are missing, both models exhibit low recall, with true positive rates of 31.8% for Claude and 36.4% for GPT-4o, alongside elevated false negative rates. At the same time, false positives increase compared to categories with stronger structural cues. This behavior reflects the inherent ambiguity of omission-based errors: the absence of a configuration line may represent either an actual misconfiguration or an intentional design choice, and without additional context, the models frequently struggle to disambiguate these cases.

Replicated configuration stanzas offer partial guidance. When replication is present, detection performance improves, with Claude and GPT-4o achieving true positive rates of 70.0% and 60.0%, respectively, and low false positive rates. However, when replication is absent, both models miss a large fraction of misconfigurations, and false negatives dominate. This pattern suggests that cross-stanza comparison provides useful signals for identifying inconsistencies, but that replication alone is insufficient to fully resolve correctness in the absence of other contextual cues.

Taken together, these results show that CAP performs best when configuration changes expose explicit structural relationships—such as references or repeated patterns—that support relational reasoning. In contrast, omissions and isolated configuration changes remain difficult to assess using static configuration snapshots alone. These limitations stem from the absence of explicit structural signals: in omission-based cases, the model must infer that a configuration element should exist, requiring prior knowledge of expected templates or design patterns, while reference-free configurations lack the relational cues that CAP relies on to guide context retrieval. As a result, the model must depend more heavily on implicit priors, leading to uncertainty or missed detections and creating a persistent trade-off between recall and precision. In contrast, systems such as AED address correctness through explicit symbolic constraints over configuration semantics, enabling them to capture certain classes of errors that are difficult to infer from context alone. This highlights a complementary trade-off: CAP provides flexible, interpretable reasoning without requiring formal models, while constraint-based approaches offer stronger guarantees but require substantial manual specification and modeling effort. Addressing these limitations will likely require augmenting CAP with additional sources of context, such as configuration templates, learned structural patterns,

Misconfigs	Count	Reason	True Positive Rate (TPR)	
			TPR (Any)	TPR (All)
Invalid subnet mask	2	Subnet mask violating binary boundary rules (*e.g.*, 255.255.253.0).	2/2 (100%)	2/2 (100%)
Inconsistent policy naming	14	Misnamed 'smnpread' for 'snmp_communities', but with consistent usage.	14/14 (100%)	14/14 (100%)
Insecure SSH KEX algorithm	1	Insecure 'diffie-hellman-group14-sha1' included in algorithm choices.	1/1 (100%)	0/1 (0%)
Ambiguous rate limit values	2	'Interval' and 'burst' set to 0, leading to nondeterministic behaviors.	2/2 (100%)	0/2 (0%)

Table 4. CAP-detected misconfigurations in comprehensive analysis of Campus A configurations. TPR (Any) indicates detection by at least one model, while TPR (All) indicates detection by all models.

historical configuration evolution, operator intent, or runtime validation signals, which can provide expectations about missing or implicit configuration elements beyond the configuration file itself.

Detecting Known Misconfigurations in Campus A: To complement the synthetic analysis, we test CAP's ability to find known, real-world misconfigurations in the Campus A network. The network operators for this campus were specifically interested in detecting incorrect 'VLAN assignments'—a type of misconfiguration that often requires a network-wide view, as deviations from uniform configurations across devices can indicate potential issues. We ran the analysis with all 5 five LLM backends to assess the robustness and model-independence of CAP when applied to real operational data. In this targeted analysis, CAP demonstrated perfect accuracy, correctly identifying all 6 known cases of incorrect VLAN assignments (100% TPR). All LLM backends yielded identical results. Among the six detected cases, two were previously unnoticed yet genuine misconfigurations that had been causing misrouting and network access issues, representing high-severity problems for network operations. The remaining four cases were intentional, benign misconfigurations applied to test and experimental devices.

6.3 Can CAP Accurately Identify New Issues?

Beyond detecting known error types, a key goal for CAP is to discover latent or novel misconfigurations in large, complex networks. To evaluate this capability, we performed a comprehensive verification of configuration snapshots from Campus A and B. In particular, each line of configuration was fed to CAP to determine if it contained an error. For each misconfiguration detected by CAP, we verified the correctness of the inference with domain experts from the respective campus networks. This open-ended approach allows us to not only assess CAP's scalability and discovery potential but also to evaluate whether different LLM backends—**GPT-4o**, **Claude**, **DS**, **Gemini**, and **LLaMa**—surface distinct issues or exhibit different reasoning patterns.

Campus A: Discovering Genuine Misconfigurations: In the analysis of Campus A configurations, CAP successfully detected several previously unknown types of misconfigurations, as shown in Table 4. We categorize the detected misconfigurations by their potential impact on network operations, as determined by the LLM and verified by experts. Specifically, CAP successfully identified 19 misconfigurations in the Campus A data: two cases of invalid subnet masks, 14 instances of inconsistent policy naming, one insecure SSH KEX algorithm, and two ambiguous rate limit values. All 19 cases were confirmed to be either valid misconfigurations or justifiable concerns, resulting in a true positive rate (TPR) of 100% for this dataset.

The detection results were consistent across the GPT-4o, Claude, and DS backends, all of which identified all 19 cases. LLaMA failed to detect the insecure SSH KEX configuration, and Gemini missed the two ambiguous rate-limit cases; these misses correspond to a small subset of lower-impact misconfigurations and do not affect the overall conclusions.

Campus B: The Challenge of False Positives: In contrast to the genuine misconfigurations identified in Campus A, the analysis of the Campus B snapshots highlights the challenge of false

Misconfiguration Type (False Positives)	Count	GPT-4o	DeepSeek	Claude	Gemini	LLaMA
Duplicate / overlapping DHCP helpers	8	8 / 8	4 / 8	4 / 8	7 / 8	1 / 8
Missing VLANs in trunk links	9	9 / 9	0 / 9	0 / 9	7 / 9	5 / 9
Uncoordinated channel-group assignment	2	2 / 2	0 / 2	1 / 2	2 / 2	2 / 2
Unexpected PIM passive usage	2	2 / 2	0 / 2	0 / 2	0 / 2	0 / 2

Table 5. Campus B non-targeted snapshot. Each cell shows the number of cases flagged as misconfigurations out of the total (**false positives**). All listed configurations were later validated as correct.

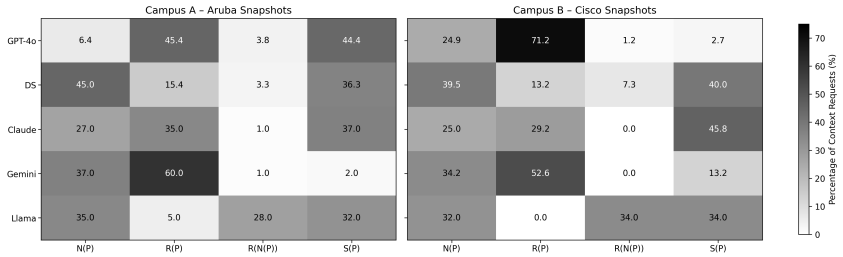


Fig. 4. Distribution of context categories requested by LLM backends.

positives in open-ended discovery. As detailed in Table 5, all configurations flagged as potential issues in Campus B were subsequently validated as correct and intentional by network operators. Representative cases of these false positives include:

- *Duplicate or overlapping DHCP helpers*, which were flagged by the models but were intentionally deployed for redundancy and failover purposes.
- *Missing VLANs in trunk links* and *Uncoordinated channel-group assignments*, which appeared inconsistent when viewed in isolation but were confirmed to be benign due to compensatory downstream configurations or manual overrides.
- *Unexpected PIM passive usage*, which aligned with localized multicast policies specific to that campus, albeit being undocumented in standard configuration templates.

Interestingly, the LLM backends exhibited substantial divergence on this dataset. GPT flagged all 21 cases as misconfigurations, whereas DS flagged only 4 cases (all from the DHCP-helper category). Claude was similarly conservative (5 total flags), while Gemini and LLaMA were substantially more aggressive (16 and 8 flags, respectively), primarily driven by the missing-VLAN and DHCP-helper categories. This difference in behavior underscores the importance of model choice and is analyzed further in the next section.

We acknowledge the limitation of not having a definitive true negative or false negative rate for this non-targeted analysis. This is an inherent challenge in evaluating any network verifier on production networks where the complete ground truth of all latent errors is unknown. The analysis of synthetic misconfigurations presented in the previous section, currently serves as the best available measure in this regard.

6.4 What Context Do LLMs Request?

To understand the reasoning behind the models' detection performance—including the causes of the false positives detailed in the previous section—we now analyze the contextual information that each LLM requests within the CAP framework. In CAP, the model itself explicitly selects the context it wants to see from a predefined library. Once requested, the framework faithfully provides the exact FQCSes without transformation or interpretation.

This design is crucial because it decouples context retrieval from context reasoning. It ensures that downstream decisions—correct or incorrect—are entirely the result of the model's own reasoning over its self-curated input. Crucially, because the model explicitly selects what context it needs,

misclassifications are not caused by faulty information from the framework; rather, they stem from the model's own interpretation of the context it chose to request, or from its failure to request sufficient or relevant context. This allows us to use the context request patterns as a diagnostic signal into each model's internal strategy.

Context Request Patterns Across Models: The differences in false-positive behavior observed in prior experiments align closely with how models select contextual information. Figure 4 summarizes the relative frequency with which each model requests different types of context, revealing clear differences in reasoning style.

Some models strongly favor directly related definitions of the configuration line under review. GPT-4o, for example, focuses heavily on retrieving the immediate definitions associated with a statement, with limited exploration of surrounding or comparable configuration lines. Gemini follows a similar approach, though it consults nearby lines more frequently. These models tend to make confident judgments based on limited but tightly scoped context, which helps them catch certain errors quickly but also leads to more aggressive flagging when intent is ambiguous, particularly in cases involving optional or intentionally omitted configuration elements such as missing VLANs and channel-group assignments. Other models distribute their attention more broadly across nearby and comparable configuration lines. Claude and DS, for instance, consistently examine surrounding context and search for similar configuration patterns elsewhere in the file. This broader view helps them determine whether a configuration reflects a recurring design choice rather than an isolated anomaly, resulting in fewer false positives. LLaMA exhibits a distinct strategy that relies less on directly related definitions and more on indirect context inferred from nearby lines and their associated references. This approach leads to selective detection of inconsistencies that emerge from broader context, while missing issues that depend on narrowly scoped details.

Implications for Framework Design and Model Choice: These observations highlight that the false positives observed in CAP are not artifacts of random noise—they are direct reflections of each model's interpretive strategy. By making context selection explicit and observable, CAP enables operators to understand these trade-offs and choose LLM backends that best match their tolerance for false alarms versus missed errors. At the same time, this raises an important question about our framework design: in letting the LLM choose its own context to avoid information overload, have we risked the model not requesting *enough* context?

To quantify this trade-off, we conducted an ablation study comparing our standard *ask-for-context* approach against a *provide-all-context* baseline, which supplies all extracted context types identified by the context miner—including $N(P)$, $S(R)$, $R(P)$, and $R(N(P))$—on the challenging *Campus B* dataset. The results, shown in Table 6, confirm that our design is a justified and effective choice. While providing full context significantly reduced GPT-4o's false positives (from 21 down to 5), it did so at the cost of a threefold increase in prompt size. For the rest of the models, the additional context provided marginal or no accuracy benefit, while largely increasing their computational overhead. This study validates that our iterative approach is a more efficient and effective design.

Ultimately, CAP's explicit, LLM-driven interface provides a transparent foundation for diagnosing such decisions. For a network operator, the choice of model would depend on their goal: GPT-4o's high-recall approach may be preferable for deep, exploratory audits where finding every potential issue is critical, even at the cost of reviewing false alarms. In contrast, DeepSeek's high-precision approach is better suited for continuous, automated checks where minimizing false positives and operator alerts is the primary concern.

Framework	Model									
	GPT-4o		DeepSeek-V3		Claude 4.5		Gemini 2.5		LLaMa 4	
	FP	Ctx	FP	Ctx	FP	Ctx	FP	Ctx	FP	Ctx
CAP (Standard)	21	~180	4	~148	5	~107	16	~198	8	~230
CAP-Full Context	5	~650	4	~650	4	~650	11	~650	4	~650

Table 6. Ablation results on Campus B, comparing the standard iterative CAP framework against a version that provides full context by default. FP denotes false positives; Ctx denotes average context tokens.

Type	Unique Cases	Number of Detected Misconfigurations (TP or TN)					
		CAP (Any)	**CAP (All)**	**Ciri (Any)**	**FFP (Any)**	**Batfish**	**Diffy**
Missing Line(s)	22	9/22 (40.91%)	6/22 (27.27%)	0/22 (0.00%)	0/22 (0.00%)	0/22 (0.00%)	11/22 (50.00%)
Involves Reference	13	11/13 (84.62%)	9/13 (69.23%)	1/13 (7.69%)	0/13 (0.00%)	2/13 (0.00%)	9/13 (69.23%)
Replicated Block	39	28/39 (71.79%)	24/39 (61.54%)	0/39 (0.00%)	0/39 (0.00%)	5/39 (0.00%)	31/39 (79.49%)
Correct Config Updates	58 (True Negative)	56/58 (96.55%)	50/58 (86.20%)	40/58 (68.97%)	58/58 (100.00%)	58/58 (100.00%)	58/58 (100.00%)
Total Accuracy	**116**	**91/116 (78.44%)**	**77/116 (66.37%)**	**46/116 (39.66%)**	**58/116 (50.00%)**	**63/116 (54.31%)**	**92/116 (79.31%)**

Table 7. Comparison of misconfiguration detection on the synthetic dataset. CAP indicates whether *any* model or *all* models correctly detected a given misconfiguration. The "Correct Config Updates" row reflects true negative.

6.5 How Does CAP Compare to Existing Tools?

Finally, we compare CAP against four representative state-of-the-art tools using the same synthetic misconfiguration dataset derived from Internet2 configurations. The dataset contains 58 injected misconfigurations spanning eight structural and semantic categories, along with 58 correct configuration updates. This controlled setting, with explicitly annotated ground truth, enables a direct, feature-by-feature comparison across a wide range of misconfiguration characteristics.

The tools considered include:

- Two alternative LLM-based approaches: a full-file prompting model (*FFP*) and the partition-based Q&A system *Ciri* [44].
- Two consistency and anomaly checkers: *Batfish* [23] (restricted to syntax and structural checks) and *Diffy* [30].

For a fair comparison, CAP, FFP and Ciri all use the same **GPT-4o** and **Claude** backends. We do not compare against forwarding-property model checkers, as no complete forwarding specifications are available for these networks. Table 7 summarizes detection outcomes stratified by misconfiguration type. A detection is considered correct if a tool correctly identifies an injected misconfiguration (true positive) or correctly classifies a valid configuration update as non-erroneous (true negative). For LLM-based methods, a detection outcome is considered a true positive *only if the backend's reasoning correctly identifies and aligns with the actual underlying error.*

Across all misconfigurations and correct updates, CAP achieves a total accuracy of 78.4% when considering detection by any backend model and 66.4% when requiring consensus across all models. While the task remains challenging across all tools, CAP consistently achieves the strongest overall performance among LLM-based approaches and performs competitively with specialized non-LLM tools across most misconfiguration categories.

Comparison Against LLM-based Approaches Both Ciri and FFP perform poorly under this evaluation, achieving overall accuracies of 39.7% and 50.0%, respectively. Neither approach detects any misconfigurations involving missing configuration lines or replicated blocks, and both exhibit extremely limited success on reference-based misconfigurations. These results highlight the limitations of static prompting strategies when misconfigurations require reasoning across multiple configuration blocks or depend on relationships that are not locally visible.

Ciri's partition-based design restricts analysis to isolated configuration fragments, preventing the model from resolving dependencies that span partitions. As a result, Ciri fails entirely on missing-line and replicated-block misconfigurations and detects only a small fraction (1 case) of

reference-based errors, where the referenced definition occasionally falls within the same partition. FFP, despite having access to the entire configuration file, performs no better. Presenting the full configuration at once leads to severe context overload, making it difficult for the model to identify which portions of the configuration are relevant to a potential error. This results in uniformly low true-positive rates across all three categories. Note this does not mean that these methods fail to report errors; rather, the reported detections are later found to all be false positives.

In contrast, CAP's context-aware iterative prompting mechanism enables selective retrieval of configuration fragments in response to model-driven context requests. This allows CAP to substantially outperform both LLM baselines on reference-based misconfigurations (84.6% under CAP (Any)), replicated blocks (71.8%), and missing-line errors (40.9%), where resolving cross-line dependencies and implicit expectations is essential.

Comparison Against Consistency and Anomaly Checkers Consistency- and anomaly-based tools exhibit different strengths across the three misconfiguration categories. Batfish achieves an overall accuracy of 54.3%, but this result is largely an artifact of the evaluation setup rather than evidence of effective misconfiguration detection. In our dataset, approximately half of the injected configuration updates are correct by construction and therefore should not be flagged. Batfish's conservative analysis appropriately produces no warnings for these valid updates, which contributes positively to its overall accuracy. However, for the remaining updates that intentionally introduce misconfigurations, Batfish only detected 5 out of the 58 cases. As a result, its accuracy is driven almost entirely by correctly accepting valid configurations, rather than by identifying erroneous ones.

Diffy represents the state of the art in misconfiguration detection based on cross-configuration comparison and serves as our primary baseline for evaluating whether our LLM-based approach can match or exceed established performance. By comparing configurations across devices or versions to identify deviations from common patterns, Diffy demonstrates relatively strong performance across misconfiguration categories. It detects most replicated-block errors (79.5%) and correctly identifies 9 out of 13 reference-related misconfigurations. Somewhat surprisingly, Diffy is less effective for missing-line errors, detecting only 11 out of 22 such cases. A closer inspection reveals that all missing-line misconfigurations detected by Diffy also involve replicated stanzas—that is, identical configuration blocks appear in other devices or versions, making the absence of a line detectable through direct comparison. In contrast, for missing-line errors that do not coincide with replicated stanzas, Diffy fails to detect any cases, as the absence of a statement does not manifest as a clear anomaly when no duplicated reference configuration exists for comparison.

CAP achieves performance comparable to Diffy overall and exhibits a clear advantage for reference-based misconfigurations. For errors involving references, CAP attains comparable or higher detection performance, with an upper bound of 11/13 (84.62%) when considering detection by any backend model and a lower bound of 9/13 (69.23%) when requiring agreement across all models. While the lower bound matches Diffy's performance, the higher upper bound reflects CAP's ability to explicitly retrieve and reason over referenced configuration context. As a result, CAP can detect reference-related misconfigurations even when they do not produce strong structural anomalies.

For missing-line errors, although CAP does not detect substantially more cases than Diffy overall (9 versus 11), its detections span a broader set of scenarios. Of the 9 missing-line misconfigurations detected by CAP, 4 involve replicated stanzas, while 5 occur in configurations without any replicated counterparts. This result highlights CAP's ability to operate both when direct cross-configuration comparison is possible and when it is not, leveraging semantic reasoning over configuration intent and context rather than relying solely on structural duplication.

Importantly, unlike Batfish and Diffy, CAP augments each detection with an explicit, human-readable reasoning trace that explains how resolved references and retrieved context support the final decision (detailed responses can be found in Appendix Table 8). This reasoning capability is particularly valuable for reference-based and non-replicated missing-line misconfigurations, where understanding dependency chains and implicit assumptions is essential for validating diagnoses and guiding remediation.

7 Discussion and Future Work

A key insight from our study is that the choice of LLM backend is consequential: different models exhibit distinct context-request and reasoning strategies that directly impact detection performance. This observation has important implications for operators, suggesting that the most effective model may depend on the specific verification task rather than a single universally optimal choice.

A second insight is that CAP's interactive, ask-for-context design functions not only as a verifier but also as a diagnostic tool. The context a model requests exposes its reasoning strategy—whether it prioritizes local semantics or global structure—making the interaction both more efficient than providing all context upfront and more transparent. This visibility into the model's reasoning process is critical for building operator trust in its conclusions.

A third insight from our evaluation is that while CAP is effective at detecting misconfigurations that expose explicit structural relationships, it is less reliable for omission-based and reference-free errors. These cases lack the contextual signals needed to guide context retrieval, making them inherently difficult to detect from static configurations alone.

Building on these insights and limitations, several directions for future work emerge. One promising avenue is an ensemble or meta-LLM framework that dynamically selects or combines models based on the configuration region or verification task. Another direction is enriching context mining with additional signals—such as configuration templates, historical evolution, operator intent, or runtime validation data—to better handle cases where structural cues are absent. Finally, CAP's structured outputs naturally support automated remediation, allowing the framework to propose—and, with operator approval, apply—validated fixes. More broadly, this points toward a shift from treating LLMs as passive analyzers to interactive reasoning agents that actively acquire context, and suggests opportunities to integrate such mechanisms directly into model training or adaptation.

Lastly, a natural concern with interactive prompting is added latency from multiple LLM interactions. In practice, CAP's latency scales with the number of context-request iterations, each consisting of lightweight context retrieval from a pre-mined library and a single LLM inference. Because context extraction is performed offline and amortized across analyses, runtime cost is dominated by LLM inference. Moreover, CAP targets configuration verification workflows—such as change reviews, pre-deployment checks, and periodic audits—rather than inline enforcement, making latencies on the order of seconds acceptable in practice.

8 Conclusion

In this paper, we introduced CAP, a framework that enables Large Language Models to analyze network configurations with the nuanced, context-aware reasoning of a human expert. We demonstrated that naive LLM applications fail due to context overload and fragmentation, and we solve this by introducing a two-part mechanism. First, a systematic context mining engine extracts all potentially relevant neighboring, similar, and referenced (both intra— and cross—device) configurations. Second, an interactive prompting engine engages the LLM in a dialogue, allowing the model to request the specific context it needs for a final, focused analysis. Our evaluation on production network configurations shows that CAP outperforms existing LLM-based baselines,

matches state-of-the-art non-LLM tools, and is most effective for misconfigurations involving references and complex dependencies. In addition, CAP produces human-readable reasoning traces that support operator trust and remediation, demonstrating the practicality of context-aware, interactive prompting for network configuration verification.

Acknowledgments

We thank the anonymous associate editors and reviewers for their constructive feedback and suggestions. This work was supported in part by the National Science Foundation under Awards #2334996, #2319603, and #2324515.

References

[1] [n. d.]. Batfish: An open source network configuration analysis tool. https://batfish.org.
[2] [n. d.]. Cisco Networking Software (IOS & NX-OS) - Support and Downloads. https://cisco.com/c/en/us/support/ios-nx-os-software.
[3] [n. d.]. Juniper Networks - Junos OS Documentation. https://juniper.net/documentation/product/us/en/junos-os.
[4] [n. d.]. Mutually Agreed Norms for Routing Security (MANRS). https://manrs.org.
[5] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. AED: incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*.
[6] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 201–219.
[7] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review* 38, 4 (2008), 63–74.
[9] Ehab Al-Shaer and Mohammed Noraden Alsaleh. 2011. ConfigChecker: A tool for comprehensive security configuration analytics. In *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*. IEEE, 1–2.
[10] Anthropic. 2025. Introducing Claude Sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5. https://www.anthropic.com/news/claude-sonnet-4-5 Accessed 2025.
[11] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 155–168.
[12] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *Symposium on Networked Systems Design and Implementation (NSDI)*.
[13] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.
[14] Rudiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2021. Metha: Network Verifiers Need To Be Correct Too!. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[15] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[16] Mark Bogdanov. 2024. *Leveraging Advanced Large Language Models To Optimize Network Device Configuration*. Ph. D. Dissertation. Purdue University Graduate School.
[17] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[18] Qiguang Chen, Libo Qin, Jinhao Liu, Dengyun Peng, Jiannan Guan, Peng Wang, Mengkang Hu, Yuhang Zhou, Te Gao, and Wanxiang Che. 2025. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models. *arXiv preprint arXiv:2503.09567* (2025).
[19] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437
[20] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[21] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering* (2024).

[22] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 43–56.
[23] 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)*. 469–483.
[24] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management Plane Analytics. In *Proceedings of the 2015 Internet Measurement Conference (IMC)*. 395–408.
[25] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM*.
[26] Google Cloud. 2025. Gemini 2.5 Pro. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-pro Accessed 2025.
[27] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.
[28] Alan Jeffrey and Taghrid Samak. 2009. Model checking firewall policy configurations. In *2009 IEEE international symposium on policies for distributed systems and networks*. IEEE, 60–67.
[29] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding network misconfigurations by automatic template inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 999–1013.
[30] Siva Kesava Reddy Kakarla, Francis Y Yan, and Ryan Beckett. 2024. Diffy: Data-Driven Bug Finding for Configurations. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 199–222.
[31] Ali Kheradmand. 2020. Automatic Inference of High-Level Network Intents by Mining Forwarding Patterns. In *Symposium on SDN Research (SOSR)*.
[32] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. 2011. The evolution of network configuration: a tale of two campuses. In *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference (IMC)*.
[33] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2024. LLM-based and retrieval-augmented control code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 22–29.
[34] Franck Le, Sihyung Lee, Tina Wong, Hyong S Kim, and Darrell Newcomb. 2006. Characterization and problem detection of routing policy configurations. (2006).
[35] Franck Le, Sihyung Lee, Tina Wong, Hyong S Kim, and Darrell Newcomb. 2006. Minerals: using data mining to detect router misconfigurations. In *Proceedings of the 2006 SIGCOMM workshop on Mining network data*. 293–298.
[36] Franck Le, Sihyung Lee, Tina Wong, Hyong S Kim, and Darrell Newcomb. 2008. Detecting network-wide and router-specific misconfigurations through data mining. *IEEE/ACM transactions on networking* 17, 1 (2008), 66–79.
[37] Franck Le, Geoffrey G. Xie, and Hui Zhang. 2007. Understanding Route Redistribution. In *2007 IEEE International Conference on Network Protocols (ICNP)*.
[38] Cheryl Lee, Chunqiu Steven Xia, Longji Yang, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. 2024. A unified debugging approach via llm-based multi-agent synergy. *arXiv preprint arXiv:2404.17153* (2024).
[39] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).
[40] Fuliang Li, Haozhi Lang, Jiajie Zhang, Jiaxing Shen, and Xingwei Wang. 2024. PreConfig: A Pretrained Model for Automating Network Configuration. arXiv:2403.09369 [cs.NI] https://arxiv.org/abs/2403.09369
[41] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. [n. d.]. Can long-context large language models understand long contexts? ([n. d.]).
[42] Lei Li, Yongfeng Zhang, and Li Chen. 2023. Prompt distillation for efficient llm-based recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 1348–1357.
[43] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060* (2024).
[44] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. 2023. Configuration validation with large language models. *arXiv preprint arXiv:2310.09690* (2023).
[45] Chang Liu, Xiaohui Xie, Xinggong Zhang, and Yong Cui. 2024. Large Language Models for Networking: Workflow, Advances and Challenges. *IEEE Network* (2024). doi:10.1109/MNET.2024.3510936
[46] Qianou Ma, Hua Shen, Kenneth Koedinger, and Sherry Tongshuang Wu. 2024. How to teach programming in the ai era? using llms as a teachable agent for debugging. In *International Conference on Artificial Intelligence in Education*. Springer, 265–279.
[47] Ajay Mahimkar, Zihui Ge, Xuan Liu, Yusef Shaqalle, Yu Xiang, Jennifer Yates, Shomik Pathak, and Rick Reichel. 2022. Aurora: conformity-based configuration recommendation to improve LTE/5G service. In *Proceedings of the 22nd ACM*

Internet Measurement Conference (IMC).
[48] David A Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtỳsson, and Albert Greenberg. 2004. Routing design in operational networks: A look from the inside. *ACM SIGCOMM Computer Communication Review* 34, 4 (2004), 27–40.
[49] Meta AI. 2025. The Llama 4 Herd: The Beginning of a New Era of Natively Multimodal AI. https://ai.meta.com/blog/llama-4-multimodal-intelligence/. https://ai.meta.com/blog/llama-4-multimodal-intelligence/ Accessed 2025.
[50] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to Synthesize Correct Router Configurations?. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets)*. 189–195. doi:10.1145/3626111.3628194
[51] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 953–967.
[52] Hongjin Qian, Zheng Liu, Peitian Zhang, Kelong Mao, Yujia Zhou, Xu Chen, and Zhicheng Dou. 2024. Are Long-LLMs A Necessity For Long-Context Tasks? *arXiv preprint arXiv:2405.15318* (2024).
[53] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. 2023. Practical Intent-driven Routing Configuration Synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[54] Amber Rasche. 2023. 5 Questions for Scott Richmond and Jon-Paul Herron on Embracing Network Automation at ESnet. https://internet2.edu/5-questions-embracing-network-automation-at-esnet/.
[55] Ronald W Ritchey and Paul Ammann. 2000. Using model checking to analyze network vulnerabilities. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 156–165.
[56] Prakhar Sharma and Vinod Yegneswaran. 2023. PROSPER: Extracting Protocol Specifications Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets)*.
[57] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2020. Probabilistic Verification of Network Configurations. In *SIGCOMM*.
[58] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–19.
[59] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd D. Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *ACM SIGCOMM 2021 Conference*.
[60] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[61] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, et al. 2024. Debugbench: Evaluating debugging capability of large language models. *arXiv preprint arXiv:2401.04621* (2024).
[62] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[63] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. 2010. California fault lines: understanding the causes and impact of network failures. In *SIGCOMM*.
[64] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
[65] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. 2024. NetConfEval: Can LLMs Facilitate Network Configuration? *Proceedings of the ACM on Networking* 2, CoNEXT2 (2024), 1–25.
[66] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. 2023. Making Network Configuration Human Friendly. arXiv:2309.06342 [cs.NI] https://arxiv.org/abs/2309.06342
[67] Jianxun Wang and Yixiang Chen. 2023. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 284–289.
[68] Yu Wang, Shiwan Zhao, Zhihu Wang, Heyuan Huang, Ming Fan, Yubo Zhang, Zhixing Wang, Haijun Wang, and Ting Liu. 2024. Strategic chain-of-thought: Guiding accurate reasoning in llms through strategy elicitation. *arXiv preprint arXiv:2409.03271* (2024).
[69] Zehao Wang, Dong Jae Kim, and Tse-Hsun Chen. 2024. Identifying Performance-Sensitive Configurations in Software Systems through Code Analysis with LLM Agents. *arXiv preprint arXiv:2406.12806* (2024).
[70] Yige Xu, Xu Guo, Zhiwei Zeng, and Chunyan Miao. 2025. Softcot: Soft chain-of-thought for efficient reasoning with llms. *arXiv preprint arXiv:2502.12134* (2025).

[71] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *SIGCOMM*.
[72] Peng Zhang, Dan Wang, and Aaron Gember-Jacobson. 2022. Symbolic router execution. In *ACM SIGCOMM Conference*, Fernando Kuipers and Ariel Orda (Eds.).

A Appendix

B Additional Design Details

In the section we provide the additional design details including the actual prompts used in CAP.

Step 1: Initial Prompting

```
I have a line of network route configuration for a Junipter router and need to determine if there is any
    ↪ misconfiguration:
Configuration Line Under Review: groups-->INTERFACE-BACKBONE-->interfac
es--><xe-*>-->mtu=True
Given this information, please now proceed to analyze the provided configuration to identify any
    ↪ potential misconfigurations. Do not hallucinate if no misconfiguration is occurring.
Additional contextual information with respect to the configuration line under review can be provided
    ↪ upon request if useful for identifying SYNTAX misconfigurations, include (1) Neighboring
    ↪ Configuration Lines; (2) Configuration Lines with Similar Pattern and Nature; (3) Referenceable
    ↪  Configuration Lines at both intra and cross-device Level; (4) Referenceable Configuration
    ↪ Lines for Neighboring Configuration Line. Additionally, you must classify the terminal value in
    ↪  the line under review ('True') as either a pre-defined value that is part of the configuration
    ↪  language or a custom, user-defined identifier.

Respond in a json format strictly adhering to the following and do not provide any other output:
If no further contextual information is needed:
{
    "Misconfigured": boolean, // true if there is any misconfiguration
    "Parameter": [], // List of misconfigured parameters
    "reason": [] // List of misconfiguration explanations
}
If further contextual information is needed:
{
  "requestedInfo": [], // An array of integers from the menu above. Leave empty if no context is needed.
  "reason": // Explanation of why the requested context is necessary.,
  "isUserDefined": boolean, // 'false' if 'True' is a pre-defined value, 'true' if it is a custom
      ↪ identifier.
  "disambiguationKeyword": '', // A keyword expected to appear alongside the value in its definition
      ↪ path, used to disambiguate references.
  "classificationReason": // A brief explanation for your classification.
}
```

Fig. 5. The initial dialogue step where the framework solicits context requirements from the LLM.

Step 2: Example LLM Response for Context Request

```
{
    "requestedInfo": [1, 2],
    "reason": ["Neighboring configuration lines are necessary to confirm the syntax structure in context
        ↪ .", "Configuration lines with a similar pattern and nature are needed to ensure consistent
        ↪ parameter usage."],
    "isUserDefined": false,
    "disambiguationKeyword": '',
    "classificationReason": "The value 'True' is a standard boolean keyword in many configuration and
        ↪ programming languages, not a custom, user-defined identifier."
}
```

Fig. 6. The LLM's response, specifying the exact context it requires for its analysis and providing determination for whether the parameter value is user-defined.

Step 3: Framework Provides Requested Context

```
Neighboring Configuration Lines:
 - groups-->INTERFACE-BACKBONE-->interfaces--><xe-*>-->unit <*>
    - family inet-->filter-->input-list-->1=backbone-in
    - family inet-->filter-->input-list-->0=uplink-in
    - family inet-->filter-->output=interface-out
    - family iso-->mtu=1497
    - family mpls-->filter-->input=mpls-sample
    - family inet6-->filter-->output=v6filter
    - family inet6-->mtu=9174
    - family inet6-->filter-->input-list-->0=inet6-sample
    - family inet6-->filter-->input-list-->1=v6filter
    - family mpls-->mtu=9174
    - family inet-->mtu=9174

Similar Configuration Lines:
 - groups
    - INTERFACE-BACKBONE-->interfaces--><ae*>-->unit <*>-->family iso--> mtu=1497 (Further definition DO
        ↪  NOT EXIST for the value [1497] assigned to the parameter [mtu] in the rest of the
        ↪ configuration.)
    - INTERFACE-CONNECTOR-->interfaces--><ae*>-->mtu=9192 (Further definition DO NOT EXIST for the value
        ↪  [9192] assigned to the parameter [mtu] in the rest of the configuration.)
```

Fig. 7. The second prompt from the framework, which includes the context the LLM requested.

Step 4: Final LLM Response with Analysis

```
{
    "Misconfigured": true,
    "Parameter": ["mtu"],
    "reason": ["The value 'True' assigned to the parameter 'mtu' is not valid. The 'mtu' parameter
        ↪ typically requires a numeric value representing the maximum transmission unit size."]
}
```

Fig. 8. The LLM's final, context-aware analysis, delivered after receiving the necessary information.

C Additional Experiment Results

In this section, we provide additional quantitative and qualitative results to supplement our main findings.

C.1 Example LLM reasoning for CAIP

Type	Misconfig	errParameter	Reason
Missing Line(s)	firewall-missing-icmp-type	["protocol"]	"The term 'count-unreachable' is missing required match conditions. It has a 'from' clause but lacks the protocol and icmp-type specifications. Based on the pattern of other ICMP counting terms in this filter (count-source-quench, count-parameter-problem, count-echo-reply, count-echo-request), this term should include 'protocol=icmp' and 'icmp-type' specifications to properly match ICMP unreachable messages, but these are absent."
Involves Reference	policy-statement-wrong-comm-add	["add CUST-NOEXP-TRANSIT"]	"The policy-statement EBGP_CUST_FULL_IN is adding the community CUST-NOEXP-TRANSIT, but this community is already being added by the EBGP_CUST_FULL policy which is called within the same term via 'from=policy (EBGP_CUST_FULL && EBGP_CUST_PREF_LOW2HIGH)'. Specifically, the EBGP_CUST_FULL policy's 'discard' term already adds CUST-NOEXP-TRANSIT (along with other CUST-NOEXP-* communities). This creates a redundant community addition that serves no functional purpose and represents a configuration inconsistency."
Stanza Replicated	policy-statement-extra-comm-delete	["delete DISCARD"]	"The configuration line 'ext=term' appears to be malformed or erroneous. This syntax does not match standard Junos policy-statement configuration format and appears in 0/40 other configurations, while all other lines in this policy statement appear in 40/40 configurations. The 'ext=term' statement is not a valid action within a policy-statement term's 'then' clause and should be removed."
Packet Filter	firewall-missing-next	["then=accept"]	"The term count-v4other is missing the 'next term' action in its then clause. All other counting terms in this filter include 'then->next=term' to continue processing to subsequent terms, but count-v4other only has a count action. This causes packets matching this term to be implicitly discarded instead of continuing to the accept term, breaking the intended traffic flow."
Route Filter	policy-statement-connector-missing-term	["then=reject"]	"The policy-statement OESS-L3VPN-3635-EXPORT term direct has a 'then' clause defined but is missing the action statement. Based on similar configurations in the file (OESS-L3VPN-3197-EXPORT and OESS-L3VPN-3603-EXPORT), the term direct should have 'then=accept' specified to complete the policy action, but only 'then' appears without the accept action."
Interface	interface-connector-missing-mtu	["description="I2-S11610 DART | I2-ALBA-ALBA-10GE-11619"]	"The interface unit configuration is missing the MTU parameter. Based on the OESS group pattern, interface units should have an MTU value configured to ensure proper packet handling for L2VPN/L2CCC services."
Routing Protocol	protocols-bgp-connector-missing-import	["import=[SANITY-IN SET-PREF CONNECTOR-IN]"]	"The BGP group INTERNET2 is configured with local-address=162.252.70.255, which is atypical and likely invalid for establishing sessions because .255 is commonly a broadcast address in IPv4 subnets. This line is also unique compared to the other configurations (0/40 occurrences), indicating it deviates from the standard and is a probable misconfiguration that can prevent BGP neighbor sessions from coming up."
Group	policy-conflict	["remove-comms-rt"]	"There is a conflict in the configuration under term 'remove-comms-rt' where the action 'community add I2CLOUD-EXTENDED-TARGET' is followed by 'community delete I2CLOUD-EXTENDED-TARGET' which are contradictory actions."

Table 8. Example LLM Reasoning for CAP.

Table 9. Misconfiguration cases annotated by structural characteristics and configuration scope.

Case	Missing Line(s)	Involves Reference	Replicated Stanza	Packet Filter	Route Filter	Interface Interface	Routing Protocol	Group	Misconfig Detected? Claude	Misconfig Detected? GPT-4o	Context Requested Claude	Context Requested GPT-4o
firewall_missing-icmp-type	✓		✓	✓					TP	TP	$N(P), S(P)$	$N(P), R(P)$
firewall_missing-next	✓		✓	✓					TP	TP	$N(P), S(P)$	$N(P)$
firewall_missing-protocol	✓		✓	✓					FP	FP	$N(P), S(P)$	$N(P), S(P)$
firewall_swap-term			✓	✓					FP/TP	FP/FN	$N(P), S(P)$	$N(P), R(P), R(N(P)), S(P)$
firewall_wrong-icmp-type			✓	✓					TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), S(P)$
firewall_wrong-mask			✓	✓					TP	FN	$N(P), S(P)$	$N(P), S(P)$
firewall_wrong-port			✓	✓					FN	FP	$R(P), S(P)$	$N(P), R(P), S(P)$
firewall_wrong-prefix-list		✓	✓	✓					TP	FP	$R(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_extra-comm-add		✓	✓		✓				FP/TP	FP/TP	$N(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_extra-comm-delete		✓	✓		✓				TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_missing-comm-add	✓	✓	✓		✓				FP	FP	$N(P), S(P)$	$N(P), S(P)$
policy-statement_missing-comm-delete	✓	✓	✓		✓				TP	TP	$N(P), R(P), S(P)$	$N(P), R(P)$
policy-statement_missing-localpref	✓				✓				TP	TP	$N(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_missing-term	✓				✓				FP	FP	$N(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_wrong-action					✓				FP	FN	$N(P), S(P)$	$N(P), S(P)$
policy-statement_wrong-comm-add		✓			✓				TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_wrong-comm-delete		✓			✓				TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_wrong-localpref					✓				FP	FP	$N(P), S(P)$	$N(P), S(P)$
policy-statement_wrong-range					✓				FP	FP	$S(P)$	$N(P), S(P)$
policy-statement_wrong-upto					✓				FP	FP	$N(P), S(P)$	$N(P), S(P)$
prefix-list_missing-address	✓				✓				FP	FN	$N(P), S(P)$	$N(P), S(P)$
prefix-list_missing-subnet	✓				✓				FP	FP	$N(P), S(P)$	$N(P), S(P)$
prefix-list_wrong-mask					✓				FP	FN	$R(P), S(P)$	$N(P), S(P)$
prefix-list_wrong-subnet					✓				FP	FN	$R(P), S(P)$	$N(P), R(P), S(P)$
interface_backbone_mismatch-prefix						✓			FP	FP	$N(P), S(P)$	$N(P), S(P)$
interface_backbone_mismatch-vlan						✓			FP	FP	$N(P), R(P), S(P)$	$N(P), R(P), S(P)$
interface_backbone_missing	✓					✓			FP	FP	$N(P), S(P)$	$N(P), S(P)$
interface_connector_missing	✓					✓			FP	FP/FN	$N(P), S(P)$	$N(P), S(P)$
interface_connector_missing-mtu	✓					✓			FP/TP	FP/TP	$N(P), S(P)$	$N(P), S(P)$
interface_connector_missing-unit	✓					✓			FP	FP	$N(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_connector_mismatch-prefix-list		✓			✓				TP	FP	$N(P), S(P)$	$N(P), R(P), S(P)$
policy-statement_connector_mismatch-terms	✓				✓				FP	FP	$N(P), S(P)$	$N(P), S(P)$
policy-statement_connector_missing-term	✓				✓				TP	TP	$N(P), S(P)$	$N(P), S(P)$
policy-statement_mismatch-term-name					✓				TP	TP	$N(P), S(P)$	$N(P), S(P)$
protocols_bgp_backbone_missing-neighbor	✓						✓		FP	FP	$N(P), S(P)$	$N(P), S(P)$
protocols_bgp_connector_missing-import	✓	✓					✓		TP	FP	$N(P), S(P)$	$N(P), R(P), S(P)$
protocols_bgp_connector_missing-neighbor	✓						✓		FP	FP	$N(P), S(P)$	$N(P), S(P)$
protocols_bgp_connector_wrong-peer-as							✓		FP	FP	$N(P), S(P)$	$N(P), S(P)$
protocols_isis_missing-interface	✓	✓					✓		FP	FP	$N(P), S(P)$	$N(P)$
protocols_mpls_missing-interface	✓	✓					✓		FP	FP	$N(P), S(P)$	$N(P), S(P)$
protocols_mpls_missing-path	✓	✓					✓		FP	FP/TP	$N(P), S(P)$	$N(P), R(P), S(P)$
protocols_rsvp_missing-interface	✓	✓					✓		FP	TP	$N(P)$	$N(P)$
missing closing brace								✓	TP	TP	$N(P), S(P)$	None
invalid keyword for value								✓	TP	TP	$N(P), S(P)$	$N(P), R(P)$
incorrect hierarchical placement								✓	TP	TP	$N(P), S(P)$	$N(P)$
invalid IP address format								✓	TP	TP	$S(P)$	$N(P), R(P)$
MTU exceeds interface max								✓	TP	TP	$R(P)$	$N(P), R(P)$
VLAN ID outside standard range								✓	TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), R(N(P))$
AS number outside valid range								✓	TP	TP	$N(P), S(P)$	$N(P), R(P)$
prefix limit exceeds typical values								✓	TP	TP	$N(P), S(P)$	$N(P), R(P)$
reference to a non-existent group		✓						✓	TP	TP	$N(P), S(P)$	$N(P), R(P), R(N(P))$
contradictory actions in a policy								✓	TP	TP	$N(P), R(P), S(P)$	$N(P), R(P), R(N(P))$
reference to a non-existent filter		✓						✓	TP	TP	$N(P), S(P)$	$N(P), R(P)$
reference to a non-existent policy		✓						✓	TP	TP	$N(P), S(P)$	$R(P)$
applying IPv6 filter to MPLS family		✓						✓	TP	TP	$N(P), R(P), S(P)$	$R(P), S(P)$
sampling disabled for one protocol family	✓							✓	TP	TP	$S(P)$	$R(P), N(P)$
mismatch between VRF target and RD								✓	TP	TP	$N(P), S(P)$	$S(P)$
unusually small MTU for ISO family								✓	TP	TP	$N(P), S(P)$	$R(P), S(P)$

Received January 2026; revised March 2026; accepted April 2026