# Tackling Ambiguity in User Intent for LLM-based Network Configuration Synthesis

Rajdeep Mondal
University of California, Los Angeles
Los Angeles, CA, USA

Nikolaj Bjørner
Microsoft Research
Redmond, WA, USA

Todd Millstein
University of California, Los Angeles
Los Angeles, CA, USA

Alan Tang
Microsoft
Redmond, WA, USA

George Varghese
University of California, Los Angeles
Los Angeles, CA, USA

## Abstract

Beyond hallucinations, another problem in program synthesis using LLMs is ambiguity in user intent. We illustrate the ambiguity problem in a networking context for LLM-based incremental configuration synthesis of route maps and ACLs. Configuration stanzas frequently overlap in header space, making the relative priority of actions impossible for the LLM to infer without user interaction. Measurements in a large cloud identify complex ACLs with 100s of overlaps, showing ambiguity is a real problem. We propose a prototype system, Clarify, augmenting an LLM with a new module called a *Disambiguator* that helps elicit user intent. On a small synthetic workload, Clarify incrementally synthesizes routing policies and interactively disambiguates user intent to ensure correctness.

## CCS Concepts

• **Networks → Network manageability**; • **Human-centered computing → Systems and tools for interaction design**.

## Keywords

Large language models (LLMs), network verification, network configuration synthesis, specification ambiguity, incremental program synthesis

**ACM Reference Format:**

Rajdeep Mondal, Nikolaj Bjørner, Todd Millstein, Alan Tang, and George Varghese. 2025. Tackling Ambiguity in User Intent for LLM-based Network Configuration Synthesis. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3772356.3772402


*HotNets '25, College Park, MD, USA*

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772402

## 1 Introduction

*"Life is ambiguous; there are many right answers - all depending on what you are looking for."* - Roger van Oech

While LLM technology for program synthesis is dramatically improving and hallucinations may disappear, LLMs will never be able to read a user's mind for their intent. Techniques like RAG [2, 6, 11, 15, 26, 35], chain of thought [36] and Agentic AI [13] greatly reduce hallucinations and incorrect output. However, one key bottleneck remains even if an LLM can perform program synthesis perfectly: the need for the user to fully and unambiguously specify their intent. This is difficult to do even for relatively simple settings and is infeasible to expect users to do correctly for realistic tasks.

A recent study [17] on disentangling possible meanings from ambiguous English sentences found that only 32% of the LLM-proposed resolutions were considered correct in crowd-sourced evaluations. Another study [24] showed that LLMs are inconsistent in applying factual knowledge when prompted with ambiguous entities, with performance deteriorating to 75% with under-specified commands. To address this problem, Amazon Bedrock recently introduced their Automated Reasoning framework that enables users to validate logical models extracted from policy documents and identify ambiguities in cases where model outputs can have multiple interpretations [25].

We present an approach that addresses this problem in the context of synthesis for program updates, where an existing piece of code is extended to support new functionality or fix bugs. For concreteness, we focus on updates to network configurations, specifically updates to routing policy (route-maps) and access control (ACLs). Such updates happen frequently and need to be correct, and while LLMs are a natural fit for synthesizing updates, the lack of unambiguous specifications remains a limiting factor in practice [27].

We observe that often the intent of an update is itself relatively simple and unambiguous. However, if care is not taken then this update can easily cause regressions and unexpected behavior through interactions and interference with existing parts of the configuration, based on where the update is inserted. The basic idea of our approach is to leverage this observation by asking the LLM to synthesize a config snippet *in isolation* given the intent of a change, and then to use a new component that we call a *disambiguator* to determine where to place the snippet to satisfy the full user intent.

Figure 1 shows the flow diagram of our proposed approach. Like prior work on LLM-based network configuration synthesis [20], we iterate synthesis with verification. However, our approach performs synthesis *incrementally*: each synthesis call produces a single new stanza to add to an existing configuration policy (route-map or ACL). This stanza is specified and synthesized *in isolation*, which dramatically simplifies the jobs of both the user (in specifying behavior) and the LLM (in synthesizing a correct stanza). We then introduce a new component called a *disambiguator*, which asks targeted behavioral questions to determine where to place the new stanza within the existing configuration, effectively eliciting the full specification from the user in an incremental fashion.

## 2 Disambiguation workflow

In the cyclic workflow shown in Figure 1, the user starts with a simple natural language intent to add a stanza to an existing route-map or access-list. The intended behavior is then clarified through differential examples shown to the user over multiple interactions.

### 2.1 LLM Query and Verification

Consider the following routing policy named ISP_OUT written in Cisco IOS syntax [5]. Cisco *route-maps* (Cisco's name for routing policies) can use various ancillary lists for matching against specific BGP attributes. For example, in this configuration, the as-path list D0 checks whether a BGP route originates from ASN 32, while the prefix-list D1 looks for specific prefixes in the route advertisement. The route-map ISP_OUT contains 3 stanzas, which are evaluated in order, and a BGP route is compared against each stanza until a match occurs. Stanza 10 denies all routes matching the as-path list D0, stanza 20 denies any prefix specified in D1, and stanza 30 permits all routes with a local-preference value of 300. Any route that fails to match one of these stanzas is implicitly denied due to a default termination rule.

```
ip as-path access-list D0 permit _32$
ip prefix-list D1 seq 10 permit
   10.0.0.0/8 le 24
ip prefix-list D1 seq 20 permit
   20.0.0.0/16 le 32
ip prefix-list D1 seq 30 permit
   1.0.0.0/20 ge 24

route-map ISP_OUT deny 10
 match as-path D0
route-map ISP_OUT deny 20
 match ip address prefix-list D1
route-map  ISP_OUT permit 30
 match local-preference 300
```

We want to add a new stanza that permits routes with community 300:3 and prefix 100.0.0.0/16 with mask length ≤ 23, setting their metric to 55. So we write the following prompt in simple English language (note that we refrain from including any ISP_OUT specific information in the prompt):

```
Write a route-map stanza that permits
   routes containing the prefix
   100.0.0.0/16 with mask length less
   than or equal to 23 and tagged with
   the community 300:3. Their MED value
    should be set to 55.
```

To prevent LLM errors, we augment the system prompt with a task description (e.g., generate only one route-map stanza in Cisco IOS syntax) and few-shot examples containing similar prompts and their translations.

When the user submits a query, we first invoke an intermediate LLM call ((1) in Figure 1) to classify it as either an ACL or route-map synthesis query. Based on this classification, we retrieve the corresponding system prompts and examples from a database ((2)) and select the appropriate synthesis pipeline. For the given example, the classifier identifies a route-map synthesis query. GPT-4 then generates the following output ((3)):

```
ip community-list expanded COM_LIST
   permit _300:3_
ip prefix-list PREFIX_100 permit
   100.0.0.0/16 le 23

route-map SET_METRIC permit 10
 match community COM_LIST
 match ip address prefix-list PREFIX_100
 set metric 55
```

To verify the generated snippet's correctness, we use GPT-4 again to to produce a JSON specification from the modified user prompt ((3) in Figure 1) as:

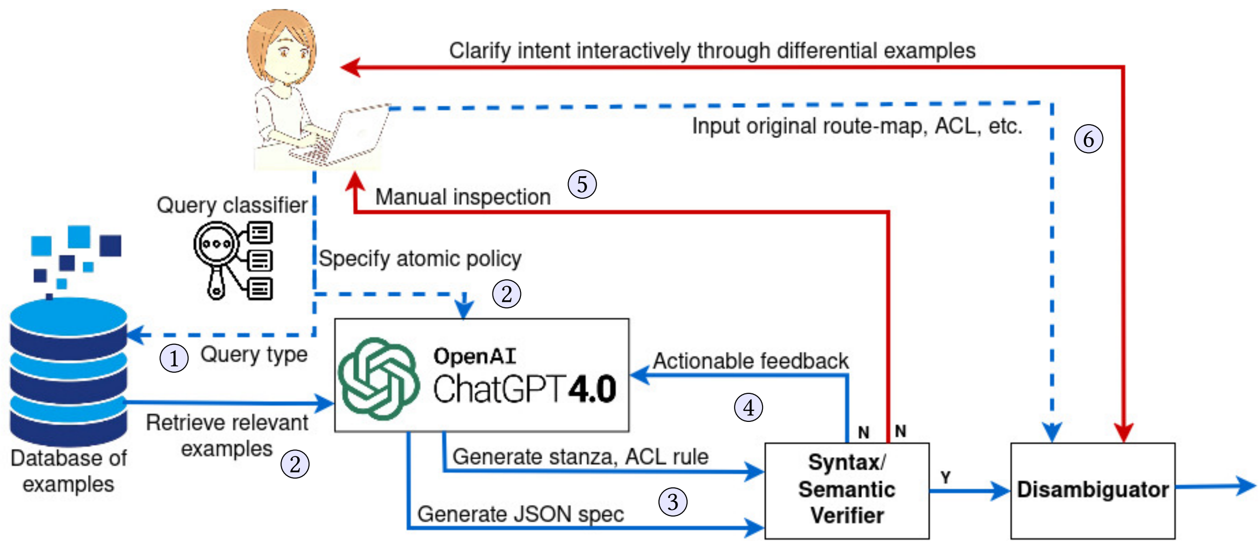


**Figure 1: Incremental synthesis with verification: The user first provides an intent for a single new config rule or stanza, which is used to identify the query type and retrieve context. The LLM generates the corresponding configuration snippet, which is verified for correctness. The disambiguator takes this and the original route-map or ACL as inputs. It searches for the correct insertion point, generating differential examples to highlight behavioral differences across candidate locations and resolve ambiguities through user feedback.**

```
{
    "permit": true,
    "prefix": ["100.0.0.0/16:16-23"],
    "community": "/_300:3_/",
    "set": {"metric": 55}
}
```

The user checks the specification manually to ensure that it has semantics equivalent to the original intent, which for one stanza is easy to cross-check. We then verify that the synthesized stanza meets the synthesized specification using existing Batfish analysis methods for proving behavioral properties of ACLs and route maps (`searchFilters` and `searchRoutePolicies`). The automated verification and feedback cycle continues until the LLM finally produces the correct output or we reach a threshold and punt to the user (⑤ in Figure 1) who starts over or provides more information.

## 2.2 Disambiguation

After verifying syntactic and semantic correctness for the LLM-generated stanza in isolation, the next step is to insert it into the correct location in the original route-map (⑥ in Figure 1). Figure 2 shows 4 possible insertion scenarios. To help the user determine intent, we use a tool called a disambiguator that compares different scenarios (using the `compareRoutePolicies` analysis in Batfish) and helps the user to clarify their preferences through differential examples. Currently, our disambiguator prototype only supports stanza insertions at the top or bottom of the initial route-map (a and b in Figure 2). For example, it finds the following input route (among others) that leads to differential behavior with respect to the route-maps of Figures 2(a) and 2(b):

```
Network: 100.0.0.0/16
AS Path: [{ "asns": [32],
            "confederation": false }]
Communities: ["300:3"]
Local Preference: 100
Metric: 0
Next Hop IP: 0.0.0.1
Tag: 0
Weight: 0
```

There are two possible behaviors, depending on where the new stanza is inserted.

**OPTION 1:**

```
ACTION: permit
Network: 100.0.0.0/16
AS Path: [{ "asns": [32],
            "confederation": false }]
Communities: ["300:3"]
Local Preference: 100
Metric: 55
Next Hop IP: 0.0.0.1
```

```
ip community-list expanded D2 permit
   _300:3_
ip prefix-list D3 permit 100.0.0.0/16
    le 23

route-map ISP_OUT permit 10
 match community D2
 match ip address prefix-list D3
 set metric 55
route-map ISP_OUT deny 20
 match as-path D0
route-map ISP_OUT deny 30
 match ip address prefix-list D1
route-map  ISP_OUT permit 40
 match local-preference 300
```

**(a)**

```
ip community-list expanded D2 permit
   _300:3_
ip prefix-list D3 permit 100.0.0.0/16
    le 23

route-map ISP_OUT deny 10
 match as-path D0
route-map ISP_OUT deny 20
 match ip address prefix-list D1
route-map  ISP_OUT permit 30
 match local-preference 300
route-map ISP_OUT permit 40
 match community D2
 match ip address prefix-list D3
 set metric 55
```

**(b)**

```
ip community-list expanded D2 permit
   _300:3_
ip prefix-list D3 permit 100.0.0.0/16
    le 23

route-map ISP_OUT deny 10
 match as-path D0
route-map ISP_OUT permit 20
 match community D2
 match ip address prefix-list D3
 set metric 55
route-map ISP_OUT deny 30
 match ip address prefix-list D1
route-map  ISP_OUT permit 40
 match local-preference 300
```

**(c)**

```
ip community-list expanded D2 permit
   _300:3_
ip prefix-list D3 permit 100.0.0.0/16
    le 23

route-map ISP_OUT deny 10
 match as-path D0
route-map ISP_OUT deny 20
 match ip address prefix-list D1
route-map ISP_OUT permit 30
 match community D2
 match ip address prefix-list D3
 set metric 55
route-map  ISP_OUT permit 40
 match local-preference 300
```

**(d)**

**Figure 2: Possible insertion points for the LLM-synthesized stanza within the route-map `ISP_OUT`. Snippets highlighted in yellow show the new stanza's location. The AS path list `D0` and prefix list `D1` are omitted for brevity. Data structure names are automatically updated by the tool during insertion.**

```
Tag: 0
Weight: 0
```

**OPTION 2:**

```
ACTION: deny
```

The user is given this information and asked to select which behavior they want. For instance, if the user selects the first option, then disambiguation is complete and we get the final route-map as shown in Figure 2(a). In general, the disambiguation process can require multiple such queries to the user in order to uniquely identify where the new stanza should be placed.

## 3 How common are overlaps?

Ambiguity is a real problem only if route maps and ACLs used in practice have considerable overlap. We developed a Batfish extension to analyze the frequency and scope of overlaps in them within the WAN of a major cloud provider and a university campus network. Two ACL rules are said to have a conflicting overlap if they perform different actions on a packet containing a header that is successfully matched by both. For route-maps, we define two stanzas to have an

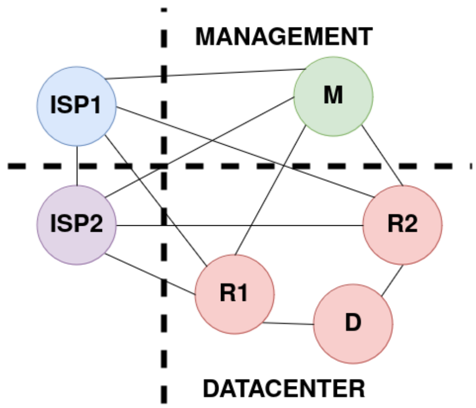


**Figure 3: Network topology used for evaluation**

overlap if there is at least one route advertisement that successfully matches both. We ignore actions for route maps because a route-map stanza may be linked to other route-maps using `goto`, `continue` and `call` statements. Thus the route map overlap calculation is an upper bound.

## 3.1 Overlaps in a Cloud Network

Unfortunately, Batfish did not have support for all vendors used in the cloud network. As a result, we only ran experiments on the configurations that could be parsed. We examined 237 non-identical ACLs, some of which may be created from the same template, and determined that 69 had at least one overlap; 48 of these had an overlap count of more than 20. In one case, an ACL that processes nodes entering the network from an outside network contained dozens of rules permitting and denying combinations of source prefixes, destination prefixes, and protocols. This results in over 100 pairs of overlapping rules. In this case, the ordering of the rules can have a large effect on the behavior of the ACL.

Turning to route maps, the ones applied to routes from or to external neighbors perform fairly complex logic; thus several contain overlapping stanzas. We examined 800 policies and found 140 contain overlaps. Of these, three were found to have more than 20 overlaps each. In the campus network, each BGP neighbor typically used one route map for importing and one for exporting. In routers we examined in the cloud, it was more common to use a sequence of multiple route maps. Hence, there can be overlaps not just between different stanzas within a single route map, but also between different route maps applied to the same neighbor.

## 3.2 Overlaps in a Campus Network

In the campus network consisting of 1421 device configurations, we analyzed 169 route-maps. We found 2 route-maps with overlapping stanzas. One route-map had three overlapping stanza pairs, of which two were conflicting.

Access-control lists were more widely used in the campus network with 11,088 ACLs. Of these, 37.7% had conflicting rule overlaps. Further, 27% of such ACLs had more than 20 conflicts. This analysis included pairs where one rule's match condition is a proper subset of the other (e.g. `permit tcp host 1.1.1.1 host 2.2.2.` and `deny ip any any`). If we ignore such cases, then the percentage of ACLs with non-trivial overlaps comes out to be approximately 18.6%. Of these, 16.3% show an overlap count of greater than 20.

In summary, overlaps are very common making manual incremental changes perilous whether done by an LLM based system or a human operator. A small error in intent can break existing policies and cause major network downtime [19].

## 4 Disambiguation Algorithm

In this section we formalize the disambiguation problem and our approach. Let *Input* denote the set of all possible input routes or packets to a route map or ACL. We model each of these components abstractly as a sequence of *rules*. Let *Rule* be the set of all possible rules. A rule $S$ matches against an input $r$ based on specified conditions and performs some action on it. If there is a successful match, then the function $matches(r, S)$ returns true, otherwise false.

A route map or ACL is then modeled as a list of rules $\overline{S} = [S_1, S_2, ..., S_n]$, and its semantics is defined by the function $M : Input \rightarrow Rule$ defined as follows:

$$\forall r \in Input,\ M(r) = \text{argmin}_{S \in \overline{S}} \mid matches(r, S)$$

$M$ formalizes which rule each route is handled by — the leftmost rule that matches. Note that route maps and ACLs have an implicit deny statement in the end, which we can model by adding an explicit rule at the end of $\overline{S}$.

We model the disambiguation problem as follows. The user would like to insert a new rule, $\mathcal{S}*$, into $\overline{S}$, and the intent is for the resulting list of rules to satisfy a new semantic function $M' : Input \rightarrow Rule$. Of course, $M'$ cannot be arbitrary — it must have a strong relationship to $M$ in order for it to be possible to be constructed solely by inserting a single new rule. We formalize the required relationship between $M$ and $M'$ in the following three conditions that $M'$ must satisfy:

- $\forall r \in Input,\ M'(r) = M(r)\ \vee M'(r) = S*$
- $\forall r \in Input,\ M'(r) = S* \Rightarrow matches(r, S*)$
- $\forall r, r' \in Input,\ matches(r, S*)\ \wedge\ matches(r', S*)\ \wedge\ M'(r) = M(r)\ \wedge\ M'(r') = S* \Rightarrow M(r) \leq M(r')$

The first condition formalizes the incremental nature of the update: every route is either handled as it was before or is handled by the new rule $S*$. The second condition ensures that any route handled by the new rule is in fact matched by that rule. The final condition is perhaps the most interesting: it ensures that there is at least one location where $S*$ can be inserted into $\overline{S}$ in order to implement $M'$. Specifically, if there are two inputs $r$ and $r'$ that match the new stanza $S*$ but

| Router | #Route-maps | #LLM calls | #Disambiguation |
|---|---|---|---|
| M | 4 | 9 | 5 |
| R1 | 5 | 12 | 6 |
| R2 | 5 | 12 | 6 |

**Figure 4: Statistics for generating and disambiguating the route-maps for Figure 3 incrementally.**

only $r'$ should be handled by the new stanza, then the original stanza handling $r$ must come before the original stanza handling $r'$, so we can place $S*$ somewhere in between.

Given these conditions on $M'$, we can use binary search to solve the disambiguation problem and determine where to insert $\mathcal{S}*$ into $\overline{S}$. First, we collect all rules $\{\mathcal{S}\}$ (maintaining their relative order) in $\overline{S}$ for which $\exists r \in Input$ such that $matches(r, S) \ \wedge matches(r, S*)$. Then, we extract the middle rule from $\{\mathcal{S}\}$, and ask the user to clarify the desired behavior by showing them differential examples. Based on the user's choice, one half of the subset is discarded and search continues in the other half. Hence, users are queried a logarithmic number of times to fully disambiguate insertion.

Note that we do not assume a single fixed location for inserting a new rule. If the initial assumptions hold, our algorithm will find one valid insertion point, with all such locations being equivalent in terms of the resulting route-map or ACL behavior. However, this can pose a problem when inserting multiple rules sequentially. There can be situations where the order in which they are added, and the choices made by our algorithm on where to insert them, can cause the approach to fail even though there is a solution. For the special case where all the inserted rules should be contiguous, our algorithm would still succeed. Lastly, Clarify does not yet have support for deleting or modifying existing rules in the configuration. That is left as future work.

## 5 Evaluation

To validate Clarify's efficacy in implementing router configurations incrementally from scratch, we created a synthetic topology (Figure 3) inspired by an example from Lightyear [30] and implemented the following global policies on it:

- Reused prefixes within the datacenter and management should be mutually invisible.
- The special prefix 10.1.0.0/16 (which is a service within the datacenter) should be visible to M.
- M should prefer the path through R1 to reach 10.1.0.0/16.
- No bogon prefixes should be advertised.
- ISP1 and ISP2 should be mutually unreachable via our network

These policies are similar to those used in wide-area networks [30]. Following Lightyear, we decomposed these global policies into local policies for each router, and incrementally synthesized the configurations for $R1$, $R2$ and $M$. Figure 4 shows statistics for the number of route-maps per router, number of calls made to the LLM and the number of times the user had to clarify their choice during updates. Some route-maps were reused because similar policies were applied on interfaces, reducing the number of LLM calls.

Using the prompting technique of Section 2, GPT-4 was able to synthesize the correct stanza every time in a single pass and no errors were detected. While promising, this is a simple topology with only a few stanzas in each route-map. Much more experience is required with real operators.

## 6 Related Work

Recent work [27] discusses the critical need for specifications for LLM-based systems and identifies ambiguity as a key challenge. They propose multiple strategies for disambiguation, taking inspiration from regular human interactions. These approaches aim to improve the LLM's ability to handle ambiguities. We instead observe that semantic comparison, common in differential verification [9, 31, 37], can instead be used to interactively resolve ambiguities efficiently and correctly.

There are several prior approaches to network configuration synthesis. Closest to our work are the techniques that also employ generative AI to translate English intents [10, 14, 16, 18, 20, 32, 33]. Some of these approaches tackle the problem of LLM errors, for example by including a verifier in the loop [20, 21], by pretraining on a networking dataset [16], or by leveraging existing configuration templates to reduce the search space [12, 18, 38]. Some of these methodologies have been used in the broader domain of software synthesis [28, 34]. However, none of these techniques handle the issue that we address, namely ambiguity in the user intent.

Other work on network configuration synthesis uses two main approaches: (1) users specify intents in a domain-specific language (DSL), and these intents are then automatically compiled to device configurations[1, 3, 4, 8, 22, 23]; and (2) users specify intents through an existing set of configuration templates [7, 29]. These techniques eliminate ambiguity by using a precise intent language. However, the first approach requires users to be conversant with the DSL, while the second requires mapping arbitrary intents to templates.

## 7 Conclusion and Future Work

While much research has focused on handling hallucinations in LLM produced code, our paper deals with incorrect code produced by ambiguity in user-specified intent. We address this problem via a form of incremental synthesis. An LLM writes a formal snippet based on natural language intent, and Clarify employs semantic comparisons to interact with the user to determine where the new snippet should be placed.

A natural question is whether the LLM itself could play the role of the disambiguator. While this would be possible to explore, we believe that symbolic reasoning tools are a better fit, since at that point in the process we have structured inputs with well-defined semantics (candidate configuration updates) and a precise goal (identifying behavioral differences). Our approach still leverages the LLM for what it does better than any other technology — turning natural language specifications into configuration stanzas.

Our paper is only a toy demonstration, and there is much work to be done to make it a practical tool for network configuration updates. First, the tool needs support for inserting entries into other data structures that can have conflicts like prefix lists, community-lists and AS-path lists. Second, the disambiguator presently only handles two insertion locations. Third, we have only used one form of LLM augmentation (few-shot examples). Can chain-of-thought, retrieval-augmented generation, graph RAG or agentic AI do better?

Finally, we note that the problem of intent disambiguation is very general, so our solution is potentially applicable to other settings. For example, disambiguation is required for code generation tasks beyond network configurations, and it can be useful for network configuration updates even if they are performed manually rather than via an LLM.

## 8 Acknowledgement

The work of Rajdeep Mondal, Todd Millstein and George Varghese at UCLA was partly supported by the National Science Foundation under award CNS-2402958.

## References

[1] 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* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 482–495. doi:10.1145/3386367.3431304

[2] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. arXiv:2310.11511 [cs.CL] https://arxiv.org/abs/2310.11511

[3] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 328–341. doi:10.1145/2934872.2934909

[4] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. *SIGPLAN Not.* 52, 6 (June 2017), 437–451. doi:10.1145/3140587.3062367

[5] Cisco Systems, Inc. [n. d.]. Cisco IOS Configuration Fundamentals Command Reference. https://www.cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book.html. Accessed 05-10-2025.

[6] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2025. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL] https://arxiv.org/abs/2404.16130

[7] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 579–594. https://www.usenix.org/conference/nsdi18/presentation/el-hassany

[8] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 261–281. doi:10.1007/978-3-319-63390-9_14

[9] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[10] Ahlam Fuad, Azza H. Ahmed, Michael A. Riegler, and Tarik Čičić. 2024. An Intent-based Networks Framework based on Large Language Models. In *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. 7–12. doi:10.1109/NetSoft60951.2024.10588879

[11] Yunfan Gao, Yun Xiong, Meng Wang, and Haofen Wang. 2024. Modular RAG: Transforming RAG Systems into LEGO-like Reconfigurable Frameworks. arXiv:2407.21059 [cs.CL] https://arxiv.org/abs/2407.21059

[12] Zhenbei Guo, Fuliang Li, Jiaxing Shen, Tangzheng Xie, Shan Jiang, and Xingwei Wang. 2024. ConfigReco: Network Configuration Recommendation With Graph Neural Networks. *IEEE Network* 38, 1 (2024), 7–14. doi:10.1109/MNET.2023.3336239

[13] Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha Oz, and Omer Tripp. 2025. QualityFlow: An Agentic Workflow for Program Synthesis Controlled by LLM Quality Checks. doi:10.48550/arXiv.2501.17167

[14] Beni Ifland, Elad Duani, Rubin Krief, Miro Ohana, Aviram Zilberman, Andres Murillo, Ofir Manor, Ortal Lavi, Hikichi Kenji, Asaf Shabtai, Yuval Elovici, and Rami Puzis. 2024. GeNet: A Multimodal LLM-Based Co-Pilot for Network Topology and Configuration. arXiv:2407.08249 [cs.NI] https://arxiv.org/abs/2407.08249

[15] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wentau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *CoRR* abs/2005.11401 (2020). arXiv:2005.11401 https://arxiv.org/abs/2005.11401

[16] 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

[17] Alisa Liu, Zhaofeng Wu, Julian Michael, Alane Suhr, Peter West, Alexander Koller, Swabha Swayamdipta, Noah A. Smith, and Yejin Choi. 2023. We're Afraid Language Models Aren't Modeling Ambiguity. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Singapore, 790–807. doi:10.18653/v1/2023.emnlp-main.51

[18] Jianmin Liu, Li Chen, Dan Li, and Yukai Miao. 2025. CEGS: Configuration Example Generalizing Synthesizer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 1327–1347. https://www.usenix.org/conference/nsdi25/presentation/liu-jianmin

[19] Declan McCullagh. [n. d.]. How Pakistan knocked YouTube offline (and how to make sure it never happens again).

[20] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. 2023. What do LLMs need to Synthesize Correct Router Configurations? *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 189–195. doi:10.1145/3626111.3628194

[21] Sean Welleck Pranjal Aggarwal, Bryan Parno. 2024. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. arXiv:2405.19616 [cs.AI] https://arxiv.org/abs/2412.06176

[22] 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 23)*. USENIX Association, Boston, MA, 629–644. https://www.usenix.org/conference/nsdi23/presentation/ramanathan

[23] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 33–49. doi:10.1145/3452296.3472915

[24] Anastasiia Sedova, Robert Litschko, Diego Frassinelli, Benjamin Roth, and Barbara Plank. 2024. To Know or Not To Know? Analyzing Self-Consistency of Large Language Models under Ambiguity. arXiv:2407.17125 [cs.CL] https://arxiv.org/abs/2407.17125

[25] Amazon Web Services. [n. d.]. Validate your Automated Reasoning policy test results. https://docs.aws.amazon.com/bedrock/latest/userguide/validate-automated-reasoning-policy-results.html. Accessed 17-10-2025.

[26] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. 2025. Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG. arXiv:2501.09136 [cs.AI] https://arxiv.org/abs/2501.09136

[27] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. 2024. Specifications: The missing link to making the development of LLM systems an engineering discipline. arXiv:2412.05299 [cs.SE] https://arxiv.org/abs/2412.05299

[28] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *Proceedings of the First International Symposium on AI Verification (SAIV '24)*. Springer-Verlag, 134–155. doi:10.1007/978-3-031-65112-0_7 Montreal, Canada.

[29] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 426–439. doi:10.1145/2934872.2934874

[30] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. 2023. Lightyear: Using Modularity to Scale BGP Control Plane Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. ACM. doi:10.1145/3603269.3604842

[31] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. doi:10.1145/3452296.3472925

[32] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostic, and Marco Chiesa. 2024. NetConfEval: Can LLMs Facilitate Network Configuration? *Proc. ACM Netw.* 2, CoNEXT2 (2024), 7:1–7:25. http://dblp.uni-trier.de/db/journals/pacmnet/pacmnet2.html#WangSFFKC24

[33] 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

[34] Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, and Yunhong Wang. 2024. A Survey on Data Synthesis and Augmentation for Large Language Models. arXiv:2410.12896 [cs.CL] https://arxiv.org/abs/2410.12896

[35] Zilong Wang, Zifeng Wang, Long Le, Steven Zheng, Swaroop Mishra, Vincent Perot, Yuwei Zhang, Anush Mattapalli, Ankur Taly, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. 2025. Speculative RAG: Enhancing Retrieval Augmented Generation through Drafting. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=xgQfWbV6Ey

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[37] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational Network Verification. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 213–227. doi:10.1145/3651890.3672238

[38] Xiaofeng Zhang, Xianming Gao, Peilin Tao, and Tao Feng. 2025. *GraphSynth: Synthesis of Network Configuration Templates Using Large Language Models*. Association for Computing Machinery, New York, NY, USA, 108–114. https://doi.org/10.1145/3728725.3728742