CEGS: Configuration Example Generalizing Synthesizer

Jianmin Liu, *Tsinghua University;* Li Chen, *Zhongguancun Laboratory;*
Dan Li, *Tsinghua University;* Yukai Miao, *Zhongguancun Laboratory*

https://www.usenix.org/conference/nsdi25/presentation/liu-jianmin

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

CEGS: Configuration Example Generalizing Synthesizer

Jianmin Liu[1], Li Chen[2], Dan Li[1], Yukai Miao[2]
[1]*Tsinghua University* [2]*Zhongguancun Laboratory*

Abstract

Network configuration synthesis promises to increase the efficiency of network management by reducing human involvement. However, despite significant advances in this field, existing synthesizers still require much human effort in drafting configuration templates or coding in a domain-specific language. We argue that the main reason for this is that a core capability is missing for current synthesizers: identifying and following configuration examples in configuration manuals and generalizing them to arbitrary topologies.

In this work, we fill this capability gap with two recent advancements in artificial intelligence: graph neural networks (GNNs) and large language models (LLMs). We build CEGS, which can automatically identify appropriate configuration examples, follow and generalize them to fit target network scenarios. CEGS features a GNN-based Querier to identify relevant examples from device documentations, a GNN-based Classifier to generalize the example to arbitrary topology, and an efficient LLM-driven synthesis method to quickly and correctly synthesize configurations that comply with the intents. Evaluations of real-world networks and complex intents show that CEGS can automatically synthesize correct configurations for a network of 1094 devices without human involvement. In contrast, the state-of-the-art LLM-based synthesizer are more than 30 times slower than CEGS on average, even when human experts are in the loop.

1 Introduction

Network configuration management plays a central role in network operations (NetOps). Manually managing configurations is costly and susceptible to human errors that can cause severe network downtime [13, 14, 35]. This is especially important for modern large-scale networks with more than tens of thousands of devices. To increase NetOps efficiency and correctness, researchers have made continuous efforts in configuration synthesis [1, 6, 7, 16, 17, 22, 39, 43, 44, 46]. With the descriptions of intents and network topology from network operators as input, configuration synthesis systems are expected to generate correct configurations for network devices to achieve the intents.

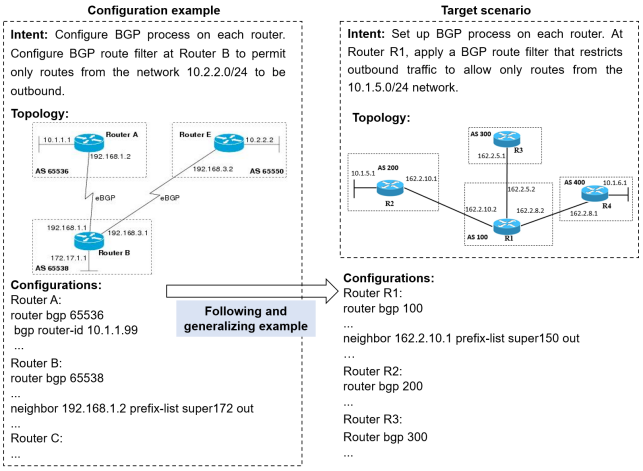


Figure 1: An instance of EFG in configuration synthesis.

Despite considerable advancements in configuration synthesis, existing proposals still require significant human effort. For example, NetComplete [17] requires operators to manually provide configuration templates where some parameters are left symbolic for the synthesizer. Robotron [44] also requires expert effort to construct suitable templates for configuration generation. For Aura [39], operators must learn a new domain-specific language, and code in this language to express their high-level intents.

The main reason for human involvement, we believe, is that a core capability is missing from all existing synthesizers: example following and generalization (EFG). In other words, they cannot identify, follow, and generalize the examples provided by device documentation, and thus must rely on human experts to perform the task. We show an example of EFG in Figure 1, where the configuration example[1] from the Cisco device documentation has a topology with three nodes for an intent, and the target scenario has a topology with four nodes. To make use of an existing synthesizer, such as NetComplete, experts must first look through the device

[1] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/configuration/xe-16/irg-xe-16-book/connecting-to-a-service-provider-using-external-bgp.html

documentation [2] for this example, identify the appropriate configuration snippets, assign the snippets to different devices, and mark symbolic parameters for synthesizers to fill, so that configuration templates can be generated for each device.

EFG is currently a human process, because it requires three sub-capabilities: understanding of topologies in both the example and target scenario, comprehension of natural language description of intents, and code generation of device configuration templates. However, with the recent advances of graph neural networks (GNNs) and large language models (LLMs), we believe that EFG can be automated. This is because 1) GNNs have shown remarkable capability in modeling node characteristics and similarities in many related applications, such as knowledge graph alignment [56,62] and social recommendation [18,55]; and 2) LLMs have demonstrated unprecedented capabilities in comprehending human intents, producing contextually relevant content, and even writing sequential programs in systems such as CoPilot [23] and Jigsaw [28].

Inspired by these achievements, this paper investigates the following research question: *Can we automate EFG for configuration synthesis?* We answer this question affirmatively with the design and implementation of a configuration synthesis system that we call CEGS (Configuration Example Generalizing Synthesizer). This involves solving three main challenges:

C1 Accurately identifying relevant configuration examples from the manual. Given a target network scenario consisting of user intents and target topology, we must find the most relevant configuration examples from the manual. Thus, we need to design a Querier to accurately identify the configuration example that closely aligns with the user intent and target topology from the manual.

C2 Associating the devices in the target topology with those in the example. Establishing accurate associations between devices in the target and example topologies is crucial for EFG. To automate this, we need to design a Classifier to accurately map the nodes in the example to the devices in the target topology, so that the configuration snippets for each node in the example can be assigned to the devices in the target topology.

C3 Accelerating synthesis without compromising correctness. As COSYNTH [36], a pioneering work that applies LLMs to configuration synthesis, indicates, an LLM usually cannot produce the correct configurations for all devices with its first attempt, and it requires multiple iterations (loops) for error correction. Each loop involves call to the LLM inference, and in some scenarios (§4), we show that, without human intervention, COSYNTH is unable to generate the correct results with as many as 300 loops. Since LLM inference is costly, we intend to design an efficient synthesis method that minimizes the number of loops without compromising correctness.

[2] Documentation includes configuration manuals, guides, *etc.*.

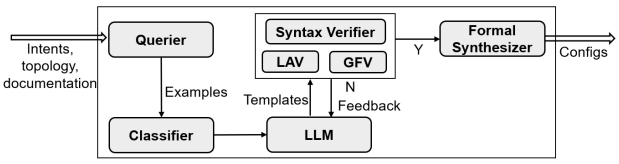


Figure 2: High-level workflow of CEGS.

CEGS comprehensively addresses the above challenges. Taking natural language intents, a target network topology, and the device documentation as input, CEGS attempts to synthesize the correct configuration for network devices in the target topology without human involvement. CEGS utilizes GNN and LLM to achieve EFG automation, and has six core components, namely Querier, Classifier, Syntax Verifier, Local Attribute Verifier (LAV), Global Formal Verifier (GFV), and Formal Synthesizer. The workflow (Figure 2) is as follows: firstly, the Querier identifies the most relevant configuration examples from the device documentation that align closely with the user intents and target topology. Secondly, the Classifier maps the nodes in configuration examples to appropriate devices in the target topology. Thirdly, CEGS leverages an LLM to generate configuration templates for all devices based on configuration examples, leaving the specific parameters of network policies symbolic. Then, CEGS utilizes the Syntax Verifier, LAV, and GFV to verify the templates. Finally, the Formal Synthesizer takes the intents, the topology, and the templates, and attempts to fill the symbolic parameters with values to achieve the intents. In this process, CEGS automatically feeds any errors caught by the Syntax verifier, LAV, GFV, and the synthesizer back to the LLM for error correction.

We enable this workflow with the following contributions:

- We design the Querier to identify relevant examples from the device manual, solving challenge C1. The Querier adopts a two-stage recommendation strategy. The first stage uses an LLM to process the natural language user intent, and then measures the similarity of the user intent to the descriptions of examples. In the second stage, the Querier further incorporates topological similarity information with a GNN, and determines the most appropriate example. In our evaluation with 90 configuration synthesis scenarios, the Querier achieves 100% accuracy in recommending examples.
- We design the Classifier to establish accurate associations between devices in the target and the example topologies, solving challenge C2. Specifically, the Classifier considers both textual and topological similarities to determine the relationships. In our evaluation with 90 configuration synthesis scenarios, CEGS also achieves 100% accuracy in terms of device classification.
- We propose an efficient LLM-driven synthesis method to accelerate synthesis without compromising correctness, solving challenge C3. Compared to prior work, our key innovations are: 1) we guide the LLM configuration generation with appropriate examples from device documen-

tations, instead of relying solely on the "memory" of the LLM; 2) we stipulate the LLM to only generate templates, because they are bad at logical reasoning and analysis [33], thus they struggle to generate specific values to achieve user policy intents; and 3) we instead use Formal Synthesizer to fill in templates, which is significantly faster than calling LLM multiple times with expert in the loop [36].

- We evaluate CEGS using Static, OSPF, and BGP routing intents and a variety of real-world topologies ranging from 20 to 1094 routers. Our results show that CEGS can automatically synthesize correct configurations based on examples, without any human involvement. Compared to state-of-the-art LLM-based synthesis systems COSYNTH, CEGS automatically synthesizes correct configurations for BGP no-transit intents within 2 minutes, while COSYNTH fails to generate correct configurations within one hour. Even with human expert in the loop, COSYNTH is more than 30 times slower than CEGS.

Limitations. This paper introduces the EFG capability to network configuration synthesizers. Enabling EFG does not solve all the challenges in configuration synthesis, and our limitations are as follows. Firstly, we are constrained by the coverage of device documentations: if there is no example of a user's intent, CEGS cannot produce a satisfactory result. When this happens, users of CEGS should ask the vendor for related examples. Secondly, CEGS is also subject to the capability of the Formal Synthesizer. In other words, to guarantee correctness, we can only support intents that are also supported by the Formal Synthesizer we use (NetComplete [17]). We consider the extension of the capabilities of the Formal Synthesizer to be orthogonal to CEGS, and do not claim any contribution in this aspect. Thirdly, the generated configuration is correct only when an LLM can correctly convert a natural language intent to a formal intent. While our evaluation demonstrates 100% accuracy in intent formalization, this result is specific to our evaluation dataset. The generalizability of this accuracy to other datasets or scenarios is not guaranteed and remains an area for further investigation.

2 BACKGROUND AND MOTIVATION

2.1 Current synthesizers require expert effort

Generating and editing configurations of devices based on ever-changing operational intents are daily tasks of NetOps engineers, and configuration synthesizers promise to reduce the load on human operators by generating correct configurations automatically. Recently, many efforts have been made in configuration synthesis [1, 6, 7, 16, 17, 22, 39, 43, 44, 46]. Despite significant advancements, existing synthesizers still require human involvement in the synthesis process.

Existing configuration synthesizers fall into two types: Domain Specific Language (DSL)-based synthesis, and template-based synthesis. Both types require significant expert effort to work. DSL-based synthesizers require experts to express high-level intents using DSLs, such as Datalog (SyNet [16]), RPL (Aura [39]), and regular expressions (Propane [6]). The learning curve and the amount of code needed for large-scale networks are both intimidating for NetOps engineers.

In this paper, we focus on reducing the human efforts for the template-based synthesizers. These systems take a network topology, high-level intents, and configuration templates as input and automatically fill the blanks in the templates. For example, NetComplete [17] autocompletes the configuration templates to correct network-wide configurations. NetComplete uses SMT constraints to model the semantics of network configurations at the level of individual route advertisements and routing policies. Specific configuration values (e.g., BGP policy parameters) are made symbolic, allowing the SMT solver to find a set of values that satisfy the desired policy.

For template-based synthesizers, we observe that the process which requires the most human effort is the process of converting a set of high-level intents into a set of templates for all devices in the topology. For NetOps experts, this process typically involves identifying appropriate configuration examples in the device documentation, assigning the snippets in the examples to devices, and marking symbolic parameters based on intents. We name this process as *example following and generalization* (EFG).

We believe that the automation of EFG can save considerable expert effort. However, automating EFG is challenging, because it requires three sub-capabilities: understanding of topology in both the example and target scenario, comprehension of natural language description of intents, and code generation of device configuration templates. Fortunately, the recent achievements of GNNs and LLMs in AI give us hope in automating these human capabilities. Next, we briefly introduce these two technologies and their prior applications in the area of configuration synthesis.

2.2 LLMs & GNNs for configuration synthesis

GNNs and LLMs have been explored separately in the literature on configuration synthesis, but they have never been combined, to the best of our knowledge.

Graph Neural Networks (GNNs) are tailored to process graph-like data, thus can be useful for topology understanding for EFG. GNNs have been shown to be capable of learning node-level and graph-level representations by aggregating information from neighboring nodes and edges. They are widely applied in various reasoning tasks on graphs [58, 60]. In relation to configuration synthesis, ConfigReco [24] is a configuration recommendation tool that uses GNN. It features a knowledge graph to model network configurations and adopts GNN to recommend configuration templates for manual configuration. However, ConfigReco can only recommend high-likelihood configuration snippets to NetOps engineers, and it cannot generalize the examples to target topologies.

LLMs [3,4,9,47] have demonstrated impressive capabilities in answering questions [41, 42], understanding user intents [34,37,52], comprehending manuals [11,32], generating commands [20, 53, 59], and even composing sequential programs

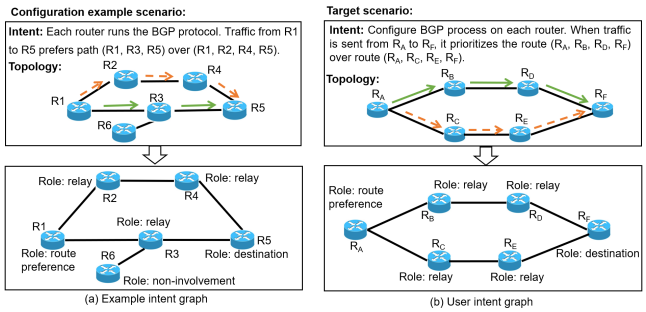


Figure 3: Configuration example and target scenarios, along with the corresponding intent graphs.

in systems like CoPilot [23] and Jigsaw [28]. All of these capabilities show immense potential for automating EFG in configuration synthesis. Using LLMs for configuration synthesis has already been explored in previous work. Rajdeep et al. [36] designed COSYNTH, a configuration synthesis system that uses LLMs to generate network configurations that are aligned with the high-level intents. NETBUDDY [49] [51], a LLM-based synthesis system, leverages LLM to simplify configuration synthesis by splitting configuration generation into multiple fine-grained steps. However, both COSYNTH and NETBUDDY directly employs LLMs to generate concrete configurations based on the specified intents without providing any configuration examples. Thus, COSYNTH and NETBUDDY still require human involvement in the loop to guide the synthesis. For example, as demonstrated in §4.3, without human involvement, COSYNTH and NETBUDDY cannot successfully compile intents into correct network configurations with as many as 300 loops. PreConfig [31] is a pretrained LLM designed for automating network configuration. However, PreConfig only can generate configuration snippets, and it still unable to synthesize network-wide configuration for target network scenarios to achieve specific high-level intents.

2.3 Motivating CEGS's Design Decisions

With the above investigation, we design CEGS which leverages GNN and LLM to automate EFG. To this end, we address the three main challenges. In the following, we list each challenge as **C#** and its related design choice(s) *D#*.

C1. Accurately identifying relevant configuration examples from the device documentation. Given a target network scenario consisting of a network topology and a set of high-level intents, we need to first identify pertinent configuration examples from the device documentation that closely align with the user intent and target topology.

D1. Transforming the recommendation problem into a graph similarity problem. We observe that the most appropriate configuration example should satisfy two criteria simultaneously: 1) the natural language descriptions of the example's intent and the user's intent must be similar; and 2) they share topological similarities. We design a data structure called *intent graph* to take both factors into account. In each intent graph, the nodes represent devices and edges represent links between the devices, thus they correspond directly to network topologies. We embed the high-level intent into the intent graph by adding an attribute called "*Role*" to each node. We use an LLM (GPT-4o) prompted by few-shot examples to process the intent and generate roles for each node (details in §3.1). In this way, we embed the high-level intent into the network topology. Figure 3 illustrates a configuration example scenario and a target scenario, along with the corresponding example intent graph and user intent graph.

With the intent graphs, we transform the problem of finding the most appropriate example to the problem of finding the most similar example intent graph to the user intent graph, which is suitable for GNN algorithms to solve. To accelerate the search, we also propose a two-stage recommendation strategy, and the first stage is to narrow the search space by filtering out irrelevant examples with the help of an LLM (GPT-4o, details in §3.1). The Querier then adopts a GNN algorithm that solves the above graph similarity problem to recommend the most relevant example.

C2. Associating the devices in the target topology with those in the example. To successfully generalize, we must obtain the correspondence between the nodes in the example intent graph and those in the user intent graph. With this correspondence, we can then assign the configuration snippets from the example to the devices in the target topology.

D2. Exact matching of role description of nodes. We observe that, in most cases, nodes can be matched using only the "Role" attribute in the intent graph. Therefore, the Classifier first associates nodes in the user intent graph to nodes in the example intent graph, if their "Role" attribute matches exactly.

D3. GNN-based mapping based on neighborhood similarity. However, in some cases, a node in the user intent graph can be matched with multiple nodes in the example intent graph. To solve this problem, we observe that the configuration of a device is also highly affected by its neighbors. Taking Figure 3 as an example, router $R2$ and $R3$ in the example topology have the same role: acting as a relay, but they have different neighborhood: router $R2$ links a source and a relay, while router $R3$ links a source, a destination, and a router which does not participate in annoucement forwarding. Thus, router $R2$ and $R3$ have different configurations of routing policies affected by their neighborhood topologies. From Figure 3, we observe that router R_B in the target topology has the same role as router $R2$ and $R3$ in the example topology, but router R_B has a neighborhood more similar to that of router R_2 than that of router R_3, so the optimal association of R_B' in the example topology should be R_2.

Therefore, we propose a GNN-based node classification algorithm in the Classifier, because GNNs can learn node-level representations by aggregating the neighborhood information and are well-suited to this problem. If a device is matched to multiple nodes in the example intent graph, the GNN algorithm measures the neighborhood similarity between these nodes and recommends the node in the example intent graph

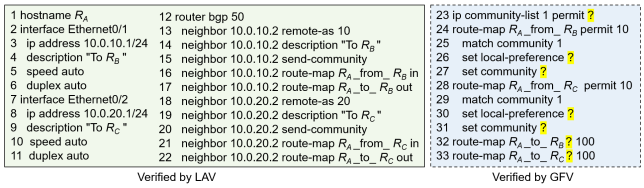


Figure 4: Configuration template for router R_A in Figure 3.

with the highest similarity.

C3. Accelerating synthesis without compromising correctness. Although recent research has demonstrated that LLMs are capable of writing network configurations [36], it is still challenging for LLM to produce concrete network-wide configurations, because it involves logical reasoning, which is not a strong suit for LLMs [36]. As a result, COSYNTH requires many human-in-the-loop iterations to generate the correct configurations. Thus, we seek a new synthesis mechanism to accelerate synthesis and reduce cost (*i.e.*the number of calling LLMs). In the meantime, it is imperative to guarantee the correctness of synthesis results.

D4. Guiding the LLM generation with appropriate examples. LLM can better complete reasoning tasks by introducing a few-shot prompt [12] or combining retrieval-augmented generation (RAG) [21]. Thus, to efficiently guide the LLM in generating the correct configuration, we provide appropriate examples to the LLM as relevant context.

D5. Restricting the LLM to generate only templates. In our configuration generation tasks, determining specific values of network policy parameters for all devices in a network is a highly complex reasoning problem that need to take the entire network into account. To solve this problem, we restrict the LLM to generate only configuration templates, leaving the specific values of network policy parameters as symbols.

D6. Using the Syntax Verifier, LAV, GFV and Formal Synthesizer to guarantee correctness. To guarantee the configuration correctness, we first use the Syntax Verifier to check whether the templates contain invalid syntax. We then utilize the LAV and GFV to jointly verify the semantic correctness of the templates. The LAV performs local verification on individual device templates to check whether the template follows the target topology and user intent descriptions. The GFV conducts global verification across all device templates to verify whether they can collectively achieve the global network policy intent. Any errors checked by the Syntax Verifier, LAV and GFV are fed to LLM for error correction. We finally use the Formal Synthesizer to take the target topology, user intents, and templates for all devices to fill the network policy parameters with concrete values to achieve the user intent.

Take the template (Figure 4) of router R_A in Figure 3 as an example, this template leaves some parameters of BGP import or export policies symbolic, such as *community* value and *preference* value, denoted by a question mark (?). We use the LAV to locally verify the interface configurations and BGP basic configurations from lines 2-22 based on the topology and intent descriptions, and then use GFV to globally verify

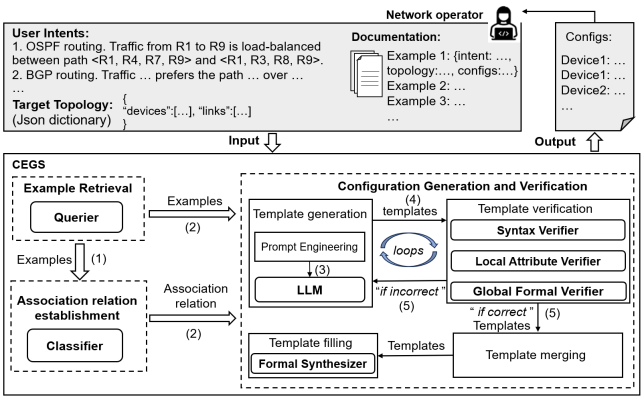


Figure 5: CEGS overview.

the policy configuration templates (lines 23-33) across all devices. Finally, the Formal Synthesizer fills the templates.

3 CEGS DESIGN

Given device documentation and a target network scenario including a target topology T_{tgt} specified in a JSON dictionary and a set of high-level intents $F=\{f^1_{user}, f^2_{user}, \ldots, f^n_{user}\}$ in natural language, CEGS automatically synthesizes network configurations via three phases: retrieval phase, association phase, and generation phase, as shown in Figure 5.

In the retrieval phase, CEGS uses the Querier to retrial relevant configuration examples from device documentation. In the association phase, CEGS utilizes the Classifier to establish accurate associations between devices in the target and example topologies. In the generation phase, given the user intents, target topology, as well as configuration examples and association relations, CEGS aims to generate correct network configuration for all devices in target topology to achieve all user intents. Specifically, we propose an iterative LLM-driven synthesis method to fulfill this task. In each iteration, we first generate configuration templates using an LLM, and validate these templates using the Syntax Veifier, LAV, and GFV. If any errors are checked, they are added to the prompt of the LLM for the next iteration. This phase stops when a set of templates are successfully generated, or when a predefined upper limit (300 in the evaluation) of loops is reached. Finally, we merge templates and fill in them using Formal synthesizer.

3.1 Querier

Given a set of high-level intents F and target topology T_{tgt}, the Querier aims to recommend the most relevant configuration example z_i for each user intent f^i_{user} from the device documentation. To achieve this, we first parse configuration examples from the device documentations. Since the format of device documentations provided by different operators varies, we develop a parsing framework similar to NAssim [11] to convert them into a uniform format. The formats are a uniform container for standardizing the diversity of manual formats. To parse the topology represented in a figure, we utilize GPT-4o to convert the topology figure into the predefined JSON format, followed by manual proofreading of the topology in-

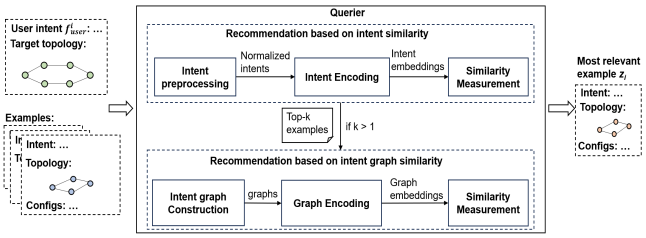


Figure 6: Detailed workflow of Querier.

formation.[3] We use a JSON dictionary to store the parsed configuration examples. Each example consists of a high-level intent in natural language, a network topology specified in JSON dictionary and the configurations of all devices in that topology. We process all the examples using the Querier to obtain the corresponding example intent graphs.

To improve query efficiency and accuracy, we adopt a two-stage recommendation strategy. Figure 6 shows the detailed workflow. For each user intent f^i_{user}, the Querier first adopts text encoding and similarity measurement methods to recommend top-k configuration examples that have the same type of intent as the user intent f^i_{user}, thus filtering out irrelevant examples. The Querier then adopts a GNN algorithm to recommend top-1 example from the top-k examples based on intent graph similarity. The output of the Querier is a set of configuration examples $Z=\{z_1,z_2,..,z_n\}$, where each z_i is the example for each user intent f^i_{user}.

3.1.1 *Recommendation based on intent similarity*

Problem Formulation. The intent is specified in natural language, so recommending configuration examples with the same type of intent can be regarded as a semantic alignment problem. Given a user intent f^i_{user} and example intent f^j_{exp}, this problem is solved by evaluating whether the semantics of intent f^i_{user} and intent f^j_{exp} are consistent.

Intent preprocessing. Due to the flexibility of natural language expression, the same intent can be expressed in various ways. For example, a path-preference intent may be described as "*For traffic from R1 to R4, the route (R1, R3, R4) is preferred over route (R1, R2, R4).*" or "*Traffic from R1 to R4 prioritizing taking (R1, R3, R4) over (R1, R2, R4).*". In addition, there may be some proper nouns in the intent description, such as specific device names and IP address, which do not impact the semantic meaning of the intent. To improve the accuracy of example recommendation, we propose an intent preprocessing approach that normalizes original intents into specific unified expression formats without specific proper nouns. Specifically, we first define a unified expression format for each intent type. We then leverage GPT-4o to convert the original intent into a unified expression format using a few-shot prompt (described in detail in Appendix A.1). For example, for the example intent and user intent described in Figure 3, we use GPT-4o to both convert them into the unified expression format "BGP routing. Traffic from *source* to *destination* prefers *path1* over *path2*". In this unified format, generic terms "*source*" and "*destination*" replace specific device names, and "*path1*" and "*path2*" replace detailed path information. We denote a normalized intent with respect to an original intent f as $norm(f)$.

Intent Encoding. To evaluate the semantic similarity between two intents, Sentence-BERT (SBERT) [40] is the most popular NLP model suitable for this task, as it can efficiently map each sentence to a vector space where semantically similar sentences are close to each other. For each pair of normalized intents of user intent $norm(f^i_{user})$ and example intent $norm(f^j_{exp})$, denoted as $(norm(f^i_{user}),norm(f^j_{exp}))$, we first use the encoder $e(\cdot)$ of SBERT to encode them into a pair of embedding vectors $(e(norm(f^i_{user})),e(norm(f^j_{exp})))$. We then use cosine similarity to measure the similarity between these two embedding vectors, which is calculated by

$$Sim(e(norm(f^i_{user})),e(norm(f^j_{exp})))= \quad (1)$$

$$\frac{e(norm(f^i_{user}))\cdot e(norm(f^j_{exp}))}{\|e(norm(f^i_{user}))\|\times\|e(norm(f^j_{exp}))\|} \quad (2)$$

The cosine similarity calculates the cosine of the angle between two vectors, which is often used to measure the semantic similarity, typically involving text data. If this similarity exceeds a threshold τ (0.9 in the evaluation), we consider intent f^i_{user} and f^j_{exp} to be semantically aligned, i.e., they belong to the same type of intent. Therefore, the configuration example corresponding to intent f^j_{exp} is regarded as a potential reference example c^i_j. We define a set $C^i=\{c^i_1,c^i_2,...,c^i_k\}$ to denote these potential reference examples, where k is the number of these examples. If k=1, we directly consider this example as the most relevant example for the user intent f^i_{user}.

3.1.2 *Recommendation based on intent graph similarity*

The Querier recommends the most relevant example from the top-k examples based on the intent graph similarity.

Problem Formulation. The most relevant example should have the similar intent to the user intent and simultaneously share topological similarity. Thus, we construct an intent graph for a pair of the intent and topology to consider both the two factors at the same. In this way, we transform the problem of recommending the most relevant example into a graph similarity problem.

Intent Graph Construction. Given an intent and a network topology, we first formulate the topology as an undirected graph, where each node v and each edge denote a device and a physical link in this topology, respectively. We then embed the intent into the graph by adding a specific attribute called "Role" to each node based on the intent. We refer to the topology graph embedding the intent as the intent graph g. We use GPT-4o to achieve this more effectively, due to their superior natural language processing capability. We first define a role set for each type intent. The role set should cover all possible roles that a device may play in fulfilling that type of intent. For example, for the path preference intent, as shown

[3] Note: Manual proofreading is required only once for accurate parsing. The parsed examples can be applied to numerous relevant target scenarios.

in Figure 3, we define the role set as *{route preference, relay, non-involvement, destination}*, where *route preference* refers to a device that prefers a neighbor for traffic transmission, such as $R1$ and R_A in Figure 3. *non-involvement* denotes a device that does not participate in traffic transmission, such as $R6$ in Figure 3. We then utilize GPT-4o to extract "Role" r_v for each node v from the intent using few-shot prompt. We describe attribute extraction with GPT-4o in detail in Appendix A.2. Under complex network scenarios, we can use a feature combination to define a role.

Graph Encoding. Give an intent graph g, we first leverage FastText [8], a NLP model, to encode the attribute values of nodes into vectors. FastText is efficient to learn word embedding through learning n-grams at the character level. We construct a corpus including the attribute values of all intent types, and adopt unsupervised learning to train a FastText model on this corpus. For each node $v \in V$, we utilize the trained FastText model to encode its textual attribute r_v into a vector $x_v = e(r_v)$, where $e(\cdot)$ is the encoder.

We then adopt GraphSAGE [25], the most popular GNN algorithm, to calculate the embedding of each node v in the graph g. Before describing how a GraphSAGE model is trained in this work, we show how GraphSAGE is used to perform node embedding. GraphSAGE uses L aggregators (denoted $AGGREGATE_l, \forall l \in \{1,2,...,L\}$) and a set of weight matrices $W^l, \forall l \in \{1,...,L\}$ to generate node embedding through l loops. At each loop l, node v's embedding is calculated by

$$h_v^l \leftarrow \sigma(W^l \cdot CONCAT(h_v^{l-1}, h_{\mathcal{N}(v)}^l)) \quad (3)$$

where, σ is a nonlinear activation function, CONCAT is a concatenation operator. $\mathcal{N}(v)$ is the set of neighbor nodes of the node v. The initial value of node embedding is its feature vector, i.e., $h_v^0 = x_v$. $h_{\mathcal{N}(v)}^l$ is the neighborhood representation of node v at the l-th loop, which is calculated by

$$h_{\mathcal{N}(v)}^l \leftarrow AGGREGATE_l(\{h_u^{l-1}, \forall u \in N(v)\}) \quad (4)$$

where h_u^{l-1} is neighbor node u's embedding at the $l-1$-th loop.

We use a global pooling method to aggregate the embeddings of all nodes in a graph g into a single vector as the representation of the graph through a MeanPool operator. The embedding of a graph g is represented as:

$$E_g = MeanPool(\{h_v, \forall v \in V\}) \quad (5)$$

where h_v denotes the embedding of node v, which is this node embedding obtained from the last loop L.

For each pair of user intent graph g_{user}^i and example intent graph g_{exp}^i, we use euclidean distance to measure the similarity between their graph embedding vectors $E_{g_{user}^i}$ and $E_{g_{exp}^i}$. A similarity score is calculated by

$$Score(E_{g_{user}^i}, E_{g_{exp}^i}) = \frac{1}{d(E_{g_{user}^i} - E_{g_{exp}^i})} \quad (6)$$

where $d(\cdot)$ is the L1 distance, which is often used in graph or node evaluation to capture the absolute differences between corresponding features. For a user intent graph g_{user}^i associated with the user intent f_{user}^i and target topology T_{tgt}, we use the above graph similarity measurement to find the top-1 similar example intent graph, and consider the configuration example associated with the top-1 example intent graph as the most relevant example z_i for the user intent f_{user}^i.

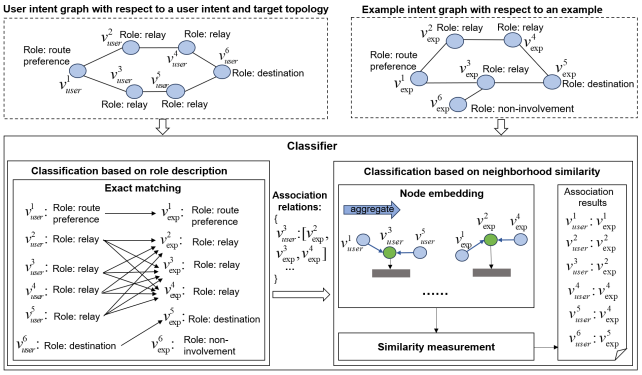


Figure 7: An instance of establishing the association between device in target and example topologies.

3.2 Classifier

For each user intent f_{user}^i and its corresponding configuration example z_i, the Classifier aims to establish the association between devices in target topology T_{tgt} and those in topology T_{exp}^i of the example z_i. The Classifier adopts a two-stage classification strategy to establish accurate association relationships on intent graphs constructed in §3.1.2. The Classifier first establishes the association relation based on the device's roles. If a device in target topology T_{tgt} is associated with multiple devices in example topology T_{exp}^i that have the same roles, the Classifier then adopts GraphSAGE to map this device to a specific one among these multiple devices in example topology T_{exp}^i based on neighborhood similarity. Figure 7 shows an instance of the Classifier establishing the association between devices in the target and example topologies.

Problem Formulation. We formalize this problem as a node classification problem on an intent graph pair of g_{user}^i and g_{exp}^i, where g_{user}^i is the user intent graph with respect to user intent f_{user}^i and target topology T_{tgt}, as well as g_{exp}^i is the example intent graph corresponding to the example z_i of the user intent f_{user}^i. In an intent graph g_{user}^i (or g_{exp}^i), each node v_{user} (or v_{exp}) is characterized by its role $r_{v_{user}}$ (or $r_{v_{exp}}$).

3.2.1 *Classification based on role description*

Given a pair of user intent graph g_{exp}^i and example intent graph g_{user}^i, the Classifier maps the nodes in the example intent graph g_{user}^i to these nodes in the intent graph g_{user}^i based on their roles. For each node v_{exp} in the graph g_{exp}^i, the Classifier sequentially evaluates whether the role $r_{v_{user}}$ of each node v_{user} in the user graph g_{user}^i exactly match those $r_{v_{exp}}$ of node v_{exp}, and if so, establishes an association between node v_{exp} and node v_{user}. In other words, node v_{user} is associated with v_{exp}, if and only if $r_{v_{user}} = r_{v_{exp}}$.

3.2.2 *Classification based on neighborhood similarity*

If a node v_{user} in user intent graph g_{user}^i is associated with multiple nodes in example intent graph g_{exp}^i, the Classifier further

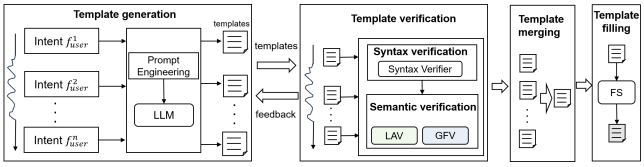


Figure 8: The workflow of LLM-driven synthesis method.

maps this node to a specific one among these multiple nodes. The Classifier first uses FastText to encode the node's textual attribute $r_{v_{user}}$ into a vector $x_{v_{user}}$, as described in §3.1. The Classifier then uses GraphSAGE to calculate the embedding $h_{v_{user}}$ of node v_{user} by aggregating its neighborhood embeddings in the graph g^i_{user}, as described in Equation 3 and 4. Simultaneously, for each node v_{exp} in graph g^i_{exp} that has the same roles with v_{user}, the Classifier uses FastText and GraphSAGE to calculate its embedding $h_{v_{exp}}$. Finally, the Classifier uses the Euclidean distance to measure the similarity between two node embedding vectors. For each node embedding pair of $h_{v_{user}}$ and $h_{v_{exp}}$, a similarity score is calculated by

$$Score(h_{v_{user}}, h_{v_{exp}}) = \frac{1}{d(h_{v_{user}} - h_{v_{exp}})} \tag{7}$$

where, $d(\cdot)$ is the L1 paradigm. The Classifier uses the above similarity measurement to map the node v_{user} to the specific node v_{exp} that has the highest similarity score.

3.2.3 *Training GraphSAGE model*

Given some intent graph pairs and node association relations in these graphs, we can generate a training corpus for training a GraphSAGE model. We construct positive pairs from nodes with associated relationships and negative pairs from nodes that are not associated. We train GraphSAGE model by minimizing the following margin-based ranking loss functions:

$$L = \sum_{(u,v)\in S} \sum_{(u,v')\in S'} [d(h_u, h_v) + \gamma - d(h_u, h_{v'})]_+ \tag{8}$$

where S is the set of positive pairs, S' denotes the set of negative pairs of the node u. h_u, h_v and $h_{v'}$ are the embedding of node u, v and v', which are calculated by GraphSAGE using Equation 3 and 4. γ (set to 3 in the evaluation) is a margin hyperparameter that separates positive and negative pairs.

3.3 An efficient LLM-driven synthesis method

To accelerate synthesis without compromising correctness, we propose an efficient LLM-driven synthesis method. Figure 8 shows the detailed workflow, including four stages. In the template generation, CEGS aims to generate templates with respect to each user intent. To guarantee the configuration correctness, CEGS first verifies template correctness. Once LLM correctly generates templates for all intents, CEGS merges these templates and then employs the Formal synthesizer to fill in the templates so that all intents are achieved.

3.3.1 *Template generation*

In this stage, CEGS leverages an LLM to generate templates for all devices in target topology T_{tgt} with respect to each user intent f^i_{user}. Due to the limited context window of the LLM, it cannot generate templates for all devices under large-scale

```
1 interface Ethernet1/0/0                    9 router ospf 50
2  ip address 47.86.50.1 255.255.255.0       10  router-id 10.10.1.1
3  description "To R2"                       11  network 47.86.50.0 0.0.0.255 area 0
4  ip ospf cost ?                            12  network 99.24.63.0 0.0.0.255 area 0
5 interface Ethernet1/0/1                    13  network 127.2.10.0  0.0.0.255 area 0
6  ip address 99.24.63.1 255.255.255.0
7  description "To R3"
8  ip ospf cost ?
```

Verified by LAV

Figure 9: Configuration template for router R1 with respect to an OSPF routing intent described in Figure 5.

typologies at once. To address this problem, we divide the devices in the topology into multiple subsets and generate the templates of devices in each subset in batches. We verify the templates for all devices collectively, so generating configurations for each batch separately do not compromise the correctness of the final configuration [4]. To speed up synthesis, we parallelize the template generation for devices in each subset, rather than processing them sequentially.

For each user intent f^i_{user}, CEGS first uses LangChain [10] to construct a prompt for each batch. We show an instance of the prompt construction in Figure 14 in Appendix A.4. We then call the LLM multiple times in parallel, feeding each batch's prompt to generate templates for all devices simultaneously. Each template is the configuration where some of network policy parameters (e.g., OSPF cost, route-map import or export parameters) are left symbolic, as illustrated in Figure 4 and Figure 9.

3.3.2 *Template verification*

In this stage, CEGS verifies configuration templates with respect to each user intent. For the templates corresponding to the intent f^i_{user}, CEGS first uses the Syntax Verifier to check the template syntax correctness, and then utilizes the LAV and GFV to jointly verify their semantic correctness.

Syntax verification. To verify template syntax correctness, we develop a Syntax Verifier by extending the Batfish [19] parser to treat symbols defined in the templates as valid syntax. If Syntax Verifier checks any invalid syntax, it reports relevant lines with invalid syntax to LLM for error correction.

Semantic verification. To speed up synthesis, it is crucial to provide localization feedback for LLMs to rapidly correct errors. To achieve this, we develop the LAV and GFV to jointly verify template semantic correctness by performing local verification and global verification. If LAV or GFV check any errors, it provides localized feedback in natural language.

- **LAV** The LLM may produce incorrect templates that do not follow the target topology and user intent descriptions, such as misconfiguring interfaces or incorrectly declaring or missing networks or neighbors. For this, we develop the LAV (that we wrote in Python) to perform local verification on each device template, checking whether it complies with the specified descriptions. Specifically, for each device template, the LAV first parses device attributes such as interface attributes (e.g., interface names, IP addresses), protocol attributes (e.g., AS number, pro-

[4] In our experiments, the size of subset is 40.

cess id) from the topology and intent descriptions. It then checks if the template correctly configures each attribute following these descriptions. If the LAV checks any errors, it provides localized feedback which indicates which configuration lines on which devices may be incorrect.

- **GFV** The LLM may produce templates that do not follow the intended global network policy. For this, we develop the GFV based on the NetComplete to perform global verification on the templates across all devices. The GFV ensures that configurations with interdependencies and cross-device impacts are correct. It reduces this task to a constraint satisfaction problem that it solves with SMT solvers [15]. The GFV uses SMT to encode the templates and uses variables to represent the symbolic in templates. To encode the user intents, the GFV first employs GPT-4o to convert the user intent from natural language to formal specification defined by the NetComplete using a few-shot prompt. We provide an instance how to use GPT-4o to convert natural language intents into formal specifications in Appendix A.3 and discuss the accuracy in section.4.2. The GFV then encodes the formal specification as SMT constraints. The GFV attempts to find a set of values to fill in the policy templates so that the formal specifications are achieved. If the GFV cannot find an appropriate set of values, it provides localized feedback indicating which devices have incorrect network policy configuration templates. Even when the GFV finds feasible values, we do not immediately fill in the templates. Instead, we retain the templates and fill in them collectively during the template filling phase based on all the intents together. [5]

Taking the configuration verification for an BGP routing intent as an example, we illustrate how our LAV and GFV work. Figure 4 shows the configuration template of a device with respect to a BGP routing intent (the user intent shown in Figure 3). We use the LAV to locally verify each device's interface configuration (lines 2-11 in Figure 4) and BGP basic configuration (lines 12-22 in Figure 4) including AS number and neighbor settings. The LAV first extracts the interface description, AS number, and IP addresses of all neighbors from the target topology description. It then checks whether the interface configuration and BGP basic configuration follow these descriptions. Since the routing policy configurations of individual devices affect each other and work together to achieve the intended global routing policy, we use the GFV to globally perform a formal verification on policy configuration templates (lines 23-33 in Figure 4) for all devices.

In some cases, we can complete the template verification using only the LAV, such as for an OSPF routing intent. Figure 9 illustrates a configuration template for a device with respect to an OSPF routing intent. Under the OSPF protocol, achieving a specific global routing policy involves setting an appropriate OSPF cost for each link. The templates represents the OSPF costs as symbols, which are comprehensively filled using the Formal Synthesizer in template filling phase. Thus, we can use the LAV to locally verify each device's template by checking whether the interface configuration (lines 1-8) correctly sets interface attributes following the target topology description and contains the OSPF cost settings (line 4 and line 8). At the same, we use the LAV to locally check whether OSPF basic settings and network statement settings (lines 9-13) comply with the topology description.

[5] While it is theoretically possible for both the formal specification and the template to be incorrectly generated yet still satisfy the SMT constraints, such cases are rare in practice. We plan to address this in future work.

3.3.3 Template merging

Once the LLM correctly generates configuration templates with respect to all intents. We merge the templates of each device with respect to all intents into a whole. In general, the configuration consists of multiple independent configuration segments, each of which implements a specific function. Thus, we merge the templates based on configuration segments. For each device, we merges the configuration segments associated with each specific function across all intents, and then merges the segments corresponding to different functions.

3.3.4 Template filling

Since NetComplete [17] can support configuration synthesis under large-scale realistic networks and cover more type of intents, we select NetComplete as our Formal Synthesizer (FS). We first employ GPT-4o to transform each user intent in natural language into the formal specification defined by NetComplete using a few-shot prompt. We then feed the merged templates for all devices, along with the formal specifications associated with all intents and the target topology to NetComplete to generate concrete values for policy parameters, thereby filling in the templates. The configurations generated by the Formal Synthesizer are correct and can satisfy the specified intents, because the Formal Synthesizer uses SMT-based formal methods to fill templates according to intent constraints, which has been validated in [17].

4 EVALUATION

We implemented CEGS in around 8K lines of Python code. We evaluate CEGS from three aspects: 1) How does CEGS compare to the state-of-the-art LLM-based configuration synthesis system COSYNTH [36] and NETBUDDY [49] [51]? 2) How does CEGS perform in achieving various set of intents across different network scales? 3) How do the core components of CEGS perform?

4.1 Experiment setup

We run all our experiments on a server with 80 core 2.1 GHz processors and 256GB RAM. We evaluate CEGS, COSYNTH, and NETBUDDY in generating Cisco router configurations that comply with the specific routing intents.

Topologies. We sampled 20 real-world ISP PoP-level network topologies from Topology Zoo dataset [29], with scales ranging from 20 to 754 routers. These topologies include 12 commercial networks and 8 educational networks, covering

various areas such as metro, region, country, and continent. Moreover, we constructed a larger topology with 1094 routers by merging three of sampled topologies from the region area.

Intents. We consider six types of prevailing routing intents, covering Static, OSPF, and BGP protocols. For Static protocol, we examine static route intents. For OSPF protocol, we evaluate ECMP and Any-path intents. For BGP protocol, we investigate Simple, Ordered, and No-transit intents. Each intent is defined between a randomly selected source *src* and destination *dst* pair and feasible paths. For a static route or a BGP Simple intent, we randomly select one feasible path p from *src* to *dst*, specifying that traffic from *src* to *dst* must follow the path p. For ECMP, Any-path, or Ordered intents, we first randomly select two available paths p_1 and p_2 from *src* to *dst*. For an ECMP intent, traffic from *src* to *dst* is load-balanced among paths p_1 and p_2. For a Any-path intent, traffic from *src* to *dst* can be forwarded along either path p_1 or p_2. For an Ordered (i.e., path preference) intent, traffic from *src* to *dst* prefers path p_1 over path p_2. We store the source-destination pair and feasible paths for each intent to facilitate configuration validation using Batfish in Sec.4.2. We utilize GPT-4o to generate intent description using few-show examples provided by NetOps experts. We describe these few examples and prompt construction in detail in Appendix.A.5.

Configuration examples corpus. We parsed 300 configuration examples from Cisco documentation and a cloud service provider, covering 40 types of intents, such as BGP routing, OSPF routing, and more. The examples for each type intent cover various networks with different topological features.

LLM selection. In the following experiments, we utilize GPT-4o to generate configurations, owing to its superior capabilities in configuration generation, compared to other models, as shown in Table 9 in Appendix B.2.

GraphSAGE model. We ask NetOps experts to help construct 100 intent graphs. For each pair of matching graphs, we ask the experts to label the associations between corresponding nodes in the two graphs. We then treat the node and its associated counterpart in the other graph as positive node pairs, while pairing the node with other non-associated nodes as negative pairs. These labeled pairs are used to create a training corpus, which is then used to train a GraphSAGE model using the loss function described in equation 8. This model consists of two hidden layers, each with 128 neurons, and is trained with a learning rate of 0.001.

Metrics. We present the synthesis system performance using three metrics: loops, synthesis time, and SMT solving time. We measure intent diversity using a semantic richness metric, defined as the average similarity between pairs of intents of the same type using SBERT embeddings. In our evaluation, the semantic richness across intent types is 0.52.

4.2 Correctness evaluation

Intent Formalization Accuracy. CEGS uses formal methods to verify and fill in templates created by LLM, which depends on accurately converting natural language intents into formal specifications. CEGS handles this with GPT-4o using few-shot prompts. To evaluate GPT-4o's accuracy, we wrote a Python script that converts each intent from our evaluations into the expected formal specification. We then asked GPT-4o to translate these intents and compared its output with the expected results. Our results show that GPT-4o achieved 100% accuracy in converting 1,350 intents from our evaluations.

Table 1: Comparison results between CEGS and COSYNTH. U stands for unsuccessful generation. ✓ (or ×) indicates that the synthesized configurations satisfy (or do not satisfy) the intents as verified by Batfish.

# No-transit intents	3		21		55	
Synthesis system	CEGS	COSYNTH	CEGS	COSYNTH	CEGS	COSYNTH
Automatic loops	**2**	300(U)	**4**	300(U)	**5**	300(U)
Synthesis time	**24s**	1h(U)	**58s**	1h(U)	**1m32s**	1h(U)
(SMT time)	**(5s)**	(0)	**(13.2s)**	0	**(21s)**	(0)
Verified	✓	×	✓	×	✓	×
Manual loops	**0**	2	**0**	5	**0**	22

Table 2: Comparisons between CEGS and NETBUDDY.

# ECMP intents	2		6		10	
Synthesis system	CEGS	NETBUDDY	CEGS	NETBUDDY	CEGS	NETBUDDY
Automatic loops	**1**	300(U)	**3**	300(U)	**4**	300(U)
Synthesis time	**42s**	3h(U)	**2m2s**	3h(U)	**3m26s**	3h(U)
(SMT time)	**(8s)**	(0)	**(20s)**	(0)	**(32s)**	(0)
Verified	✓	×	✓	×	✓	×
Manual loops	**0**	3	**0**	8	**0**	12

Validation. We validate the configurations synthesized by a synthesis system using Batfish. We wrote a Python script to automate this verification process based on the information collected in Section 4.1 about the source-destination pair and feasible paths corresponding to the intent. Specifically, we utilize Batfish "traceroute" to analyze the packet transmission path from a source to a destination specified by the intent. For a Static route intent or Simple intent, we check whether the path matches the expected path. For an Any-path intent, we first check whether the path is either path $p1$ or $p2$ as specified by the intent. We then break a random link on the path $p1$ and a random link on $p2$, and use Batfish "traceroute" to verify whether there is no path for packet transmission. For an Ordered intent, we first check whether the path matches the preferred path specified by the intent. We then randomly break a link on the preferred path, and use Batfish "traceroute" to verify whether the transmission path is changed to the secondary path specified by the intent. For an ECMP intent, we use Batfish "multipathConsistency" to verify whether there are equal paths as specified by the intent for traffic transmission from its source to destination. For a No-transit intent, we first use Batfish's "traceroute" to verify that there is no transmission path between two routers that should not communicate, as specified in the intent. We then use Batfish "traceroute" to verify whether a feasible path exists between two routers that are expected to communicate.

4.3 Comparison to COSYNTH and NETBUDDY

Since COSYNTH and NETBUDDY limit their scopes to different configuration scenarios, we compare them individually. COSYNTH does not have publicly available code, so we reproduced it based on the provided method descriptions in its

paper. We have verified that our reproduction aligns with the experimental results reported in [36]. For NETBUDDY, we use its released source code [50] for evaluation.

COSYNTH focuses on star networks and BGP no-transit intents, so we compare it in this context. In a star network, a central router connects to a CUSTOMER IP, while other routers connect to various ISPs. A no-transit intent specifies that two routers from different ISPs cannot communicate with each other, but can communicate with the CUSTOMER. We evaluate CEGS and COSYNTH on 3 star topologies from a service provider, each with 4, 8, and 12 routers, and containing 3, 21, and 55 no-transit routing intents, respectively. Table 1 presents the performance of CEGS and COSYNTH. We observe that CEGS can synthesize correct configurations that satisfy different set of no-transit intents within 2 minutes without human involvement. However, COSYNTH fails to synthesize correct configuration within 300 loops and requires extra manual loops to correct errors, taking over 3 hours. Despite extra SMT time due to the Formal Synthesizer, CEGS's total synthesis time is much shorter than COSYNTH's. The main reason is that COSYNTH does not fully leverage the EFG capabilities of LLMs. COSYNTH asks LLM to complete configuration synthesis which involves complex logical reasoning based solely on user intents, without providing any examples or leveraging formal methods for logical reasoning. Thus, it requires human experts for correction. As a result, COSYNTH is more than 30 times slower than CEGS, even with expert involvement.

We compare CEGS with NETBUDDY in achieving ECMP (i.e., load-balancing) intents under a topology with 36 routers. To simplify configuration synthesis, NETBUDDY divides configuration synthesis into three subtasks: translating natural language intents into formal specifications, generating high-level configurations (e.g., routing information), and generating network low-level configurations. However, NETBUDDY still fails to produce correct configurations within 3 hours and needs manual intervention, as shown in Table 2. The main reason is that it lacks appropriate configuration examples for LLM reasoning. In contrast, CEGS utilizes the Querier to provide relevant configuration examples, along with formal methods for logical reasoning. CEGS can quickly synthesize correct configurations within 4 minutes without human involvement, over 45 times faster than NETBUDDY.

4.4 Performance under different set of intents and different network scales

In this subsection, we further evaluate the performance of our CEGS for achieving various set of intents under different network scales. First, we evaluate CEGS performance in synthesizing configurations for varying numbers of intents across different network sizes. Table 3 and Table 4 show the results of our CEGS for synthesizing configurations for OSPF intents and BGP intents, respectively. We observe that the synthesis time and SMT time are proportional to the number of intents and the topology size. This is because more times of LLM inference are required for generating configuration templates and more symbolic policy parameters need be calculated by the Formal Synthesizer. The configurations synthesized by CEGS satisfy the corresponding intents, as verified by Batfish. On average, configurations for each device in OSPF intent scenarios consist of about 20 lines, and those in BGP intent scenarios consist of about 35 lines.

Table 3: CEGS performance in synthesizing configurations for various numbers of OSPF intents.

Network size	Intent type	#Intents	Loops	Synthesis time (SMT time)	Verified
20~50	ECMP	2	2	42.4s (4.2s)	✓
		4	2	1m18s (8.1s)	✓
		6	3	2m10s (18.9s)	✓
	Any-path	2	2	43.5 (4.5s)	✓
		4	2	1m20s (9.2s)	✓
		6	3	2m15s (20.5s)	✓
150~200	ECMP	2	4	1m30s (8.2s)	✓
		4	5	2m5s (17.3s)	✓
		6	5	3m10s (21.4s)	✓
	Any-path	2	4	1m40s (15s)	✓
		4	5	2m15s (24s)	✓
		6	5	3m20s (29s)	✓
1094	ECMP	2	9	5m35s (32s)	✓
		4	9	6m30s (51s)	✓
		6	10	7m50s (1m3s)	✓
	Any-path	2	9	6m44s (2m10s)	✓
		4	10	10m33s (4m23s)	✓
		6	10	14m30s (7m35s)	✓

Table 4: CEGS performance in synthesizing configurations for various numbers of BGP intents.

Network size	Intent type	#Intents	Loops	Synthesis time (SMT time)	Verified
20~50	Simple	2	3	1m20s (10s)	✓
		4	3	2m10s (12.5s)	✓
		6	4	3m5s (19s)	✓
	Ordered	2	4	1m40s (13s)	✓
		4	6	2m28s (16s)	✓
		6	6	3m25s (24s)	✓
150~200	Simple	2	6	3m30s (45s)	✓
		4	7	4m35s (58s)	✓
		6	8	6m8s (1m2s)	✓
	Ordered	2	8	4m15s (40s)	✓
		4	11	5m40s (1m5s)	✓
		6	13	6m55s (1m15s)	✓
1094	Simple	2	18	11m35s (2m50s)	✓
		4	19	13m15s (3m50s)	✓
		6	22	14m40s (4m40s)	✓
	Ordered	2	30	18m40s (8m45s)	✓
		4	34	20m35s (9m50s)	✓
		6	37	24m50s (10m30s)	✓

We then evaluate the performance of CEGS under more complex configuration scenarios, where multiple types of intents must be achieved simultaneously. We construct three scenarios, each involving different combinations of intents across the same or different protocols within a network of 197 routers. In Scenario Scen1, 5 BGP No-Transit and 5 Ordered intents need to be satisfied. Scenarios Scen2 and Scen3 involve different types of intents across different protocols. As shown in Table 5, CEGS can synthesizes configurations that satisfy multiple types of intents simultaneously. Even in the third scenario, including 15 intents across Static, OSPF, and BGP protocols, CEGS can synthesize correct configurations for 197 routers within 11 minutes. Configurations of each

Table 5: CEGS performance in synthesizing configurations for various combinations of multiple types of intents.

Scenarios	Protocols	Intent type	#intents	Loops	Synthesis time (SMT time)	Verified
Scen1	BGP	No-transit Ordered	5 5	13	7m30s (45s)	✓
Scen2	Static OSPF	static route ECMP	5 5	3	4m36s (30s)	✓
Scen3	Static OSPF BGP	static route ECMP Ordered	5 5 5	11	10m30s (1m42s)	✓

Table 6: Ablation study results on six core components.

Querier	Classifier	Syntax Verifier	LAV	GFV	Formal Synthesizer	Loops
×	×	✓	✓	✓	✓	300(U)
✓	×	✓	✓	✓	✓	35
✓	✓	×	✓	✓	✓	300(U)
✓	✓	✓	×	✓	✓	300(U)
✓	✓	✓	✓	×	✓	300(U)
✓	✓	✓	✓	✓	×	50
✓	✓	✓	✓	✓	✓	10

device in scenario Scen1, Scen2, and Scen3 contain about an average of 40 lines, 23 lines, and 52 lines, respectively.

4.5 Ablation Study

To evaluate the contribution of six core components in CEGS, we conduct ablation experiments on the sampled 20 realistic topologies, each with one BGP Ordered intent. Since the Classifier depends on the Querier, when evaluating the Querier, we both remove Querier and Classifier, and instruct GPT-4o to generate configuration templates without examples. For evaluating other components, we disable the specific component's functionality in CEGS. From Table 6, we observe that CEGS achieves the best performance when all components are employed simultaneously. Without the Querier and Classifier, CEGS fails to synthesize correct configurations within 300 loops, indicating the importance of configuration examples for automating EFG. Compared to not using the Classifier, CEGS can more rapidly synthesizes correct configurations by matching examples to appropriate devices using Classifier. Without a Formal Synthesizer, CEGS requires more loops to synthesize correct configurations, as GPT-4o struggles to correctly specify policy parameters. Without the Syntax Verifier, LAV or GFV, CEGS fails to synthesize correct configurations within 300 loops due to the lack of error feedback. We discuss the performance of Querier's recommendation and Classifier's classification strategies in detail in Appendix B. *We discuss how to generalize CEGS in Appendix D.*

5 RELATED WORK

Intent-based Networking. Intent-based networking [26, 30] is a networking paradigm that automates network optimization to achieve desired goals by defining high-level intents, which has been well-adapted in Software-Defined Networks (SDN) [38, 57, 61]. For example, LUMI [27] allows operators to express intents in natural language, converts them into formal intents and automatically generates configurations. LUMI takes operator feedback to verify the intents are correct.

Configuration Verification. Configuration verification tools [2, 5, 19, 45, 48, 54] aim to check whether the configurations comply with the network policies in control plane using formal methods. For example, Minesweeper [5], FSR [48] and Bagpipe [54] uses SMT solvers for verification.

Configuration Synthesis. Configuration synthesis tools [1, 6, 7, 16, 17, 22, 39, 43, 44, 46] aim at synthesizing network configurations out of high-level intents. These tools can be categorized into two types: DSL-based synthesizers and template-based synthesizers. DSL-based synthesizers require operators to express intents using the customized DSL, after which they compile these intents into correct configurations. Template-based synthesizers require network operators to provide suitable configuration templates, which then automatically fill in the templates. However, these synthesizers still require human involvement (using DSL to express intents or designing appropriate configuration templates) in the synthesis process, because they lack a core capability of EFG.

LLMs and GNNs for configuration synthesis. ConfigReco [24] is a GNN-based configuration recommendation tool, which recommends high likelihood configuration snippets to operators. Researchers have explored the ability of LLM to write network configuration. NETBUDDY [49] [51] uses a multi-stage pipeline to leverage LLMs for configuration synthesis. COSYNTH [36] adopts a verified prompt programming strategy that combines the LLM with verifiers to translate intents into router configurations. However, NETBUDDY and COSYNTH still require expert involvement in the synthesis process, as they do not fully utilize the potential EFG capability of LLM. PreConfig [31], a pretrained model, is designed to automate network configuration. Although it has the potential ability to generate configuration snippets, it cannot generate complete network-wide configurations.

6 CONCLUSIONS

We presented CEGS that leverages LLMs to automate EFG in configuration synthesis. CEGS takes a set of high-level intents in natural language, a network topology and the device documentation as input, and automatically synthesize correct network configurations. Our results show that CEGS can synthesize correct configurations for complex intents on various realistic networks. While this paper focuses on configuration synthesis, we believe that the automation of EFG can be applied to other configuration management tasks.
This work raises no ethical concerns.

7 Acknowledgments

We thank our shepherd Sanjay Rao and the anonymous NSDI reviewers for their constructive comments. Dan Li and Yukai Miao are the corresponding authors. This work was supported by National Key R&D Program of China (2022YFB3105000), the Beijing Outstanding Young Scientist Program (No. JWZQ20240101008), the National Natural Science Foundation of China under Grant U23B2001, and was supported by Zhongguancun Laboratory.

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: Incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, 2020.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[6] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 437–451, 2017.

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Harrison Chase. Langchain.(2022). *https://github.com/langchain-ai/langchain*, 2022.

[11] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 281–297, 2022.

[12] Daixuan Cheng, Shaohan Huang, Junyu Bi, Yuefeng Zhan, Jianfeng Liu, Yujing Wang, Hao Sun, Furu Wei, Denvy Deng, and Qi Zhang. Uprise: Universal prompt retrieval for improving zero-shot evaluation. *arXiv preprint arXiv:2303.08518*, 2023.

[13] Richard Chirgwin. Google routing blunder sent japan's internet dark on friday. *The Register*, 2017.

[14] G. Corfield. British airways' latest total inability to support upwardness of planes* caused by amadeus system outage, July 2018.

[15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[16] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pages 261–281. Springer, 2017.

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[18] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.

[19] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, 2015.

[20] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR, 2023.

[21] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.

[22] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 359–373, 2017.

[23] GitHub. Github copilot: Your ai pair programmer, 2023.

[24] Zhenbei Guo, Fuliang Li, Jiaxing Shen, Tangzheng Xie, Shan Jiang, and Xingwei Wang. Configreco: Network configuration recommendation with graph neural networks. *IEEE Network*, 2023.

[25] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[26] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 86–97, 2018.

[27] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, Walter Willinger, and Sanjay G Rao. Hey, lumi! using natural language for {intent-based} network management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 625–639, 2021.

[28] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[29] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[30] Aris Leivadeas and Matthias Falkner. A survey on intent-based networking. *IEEE Communications Surveys & Tutorials*, 25(1):625–655, 2022.

[31] Fuliang Li, Haozhi Lang, Jiajie Zhang, Jiaxing Shen, and Xingwei Wang. Preconfig: A pretrained model for automating network configuration. *arXiv preprint arXiv:2403.09369*, 2024.

[32] Yuan Li, Yixuan Zhang, and Lichao Sun. Metaagents: Simulating interactions of human behaviors for llm-based task-oriented coordination via collaborative generative agents. *arXiv preprint arXiv:2310.06500*, 2023.

[33] Zhiming Li, Yushi Cao, Xiufeng Xu, Junzhe Jiang, Xu Liu, Yon Shin Teo, Shang-wei Lin, and Yang Liu. Llms for relational reasoning: How far are we? *arXiv preprint arXiv:2401.09042*, 2024.

[34] Dimitrios Michael Manias, Ali Chouman, and Abdallah Shami. Towards intent-based network management: Large language models for intent extraction in 5g core networks. *arXiv preprint arXiv:2403.02238*, 2024.

[35] Meta. Update about the 4 october outage, October 2021.

[36] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. What do llms need to synthesize correct router configurations? In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 189–195, 2023.

[37] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*, 2023.

[38] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[39] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. Practical intent-driven routing configuration synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 629–644, 2023.

[40] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[41] Jaromir Savelka, Arav Agarwal, Christopher Bogart, and Majd Sakr. Large language models (gpt) struggle to answer multiple-choice questions about code. *arXiv preprint arXiv:2303.08033*, 2023.

[42] Zhenwei Shao, Zhou Yu, Meng Wang, and Jun Yu. Prompting large language models with answer heuristics for knowledge-based visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14974–14983, 2023.

[43] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of fault-tolerant distributed router configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–26, 2018.

[44] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.

[45] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 94–107, 2023.

[46] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[48] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.

[49] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval: Can llms facilitate network configuration? *Proceedings of the ACM on Networking*, 2(CoNEXT2):1–25, 2024.

[50] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval github repository. https://github.com/NetConfEval/NetConfEval, 2024. Accessed: 2024-04-23.

[51] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. Making network configuration human friendly. *arXiv preprint arXiv:2309.06342*, 2023.

[52] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. Boosting static resource leak detection via llm-based resource-oriented intention inference. *arXiv preprint arXiv:2311.04448*, 2023.

[53] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[54] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications*, pages 765–780, 2016.

[55] Le Wu, Peijie Sun, Richang Hong, Yanjie Fu, Xiting Wang, and Meng Wang. Socialgcn: An efficient graph convolutional network based model for social recommendation. *arXiv preprint arXiv:1811.02815*, 2018.

[56] Kun Xu, Liwei Wang, Mo Yu, Yansong Feng, Yan Song, Zhiguo Wang, and Dong Yu. Cross-lingual knowledge graph alignment via graph matching neural network. *arXiv preprint arXiv:1905.11605*, 2019.

[57] Ze Yang and Kwan L Yeung. Sdn candidate selection in hybrid ip/sdn networks for single link failure protection. *IEEE/ACM Transactions on Networking*, 28(1):312–321, 2020.

[58] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. Qa-gnn: Reasoning with language models and knowledge graphs for question answering. *arXiv preprint arXiv:2104.06378*, 2021.

[59] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

[60] Yongqi Zhang and Quanming Yao. Knowledge graph reasoning with relational digraph. In *Proceedings of the ACM web conference 2022*, pages 912–924, 2022.

[61] Ziyao Zhang, Liang Ma, Kin K Leung, and Franck Le. More is not always better: An analytical study of controller synchronizations in distributed sdn. *IEEE/ACM Transactions on Networking*, 29(4):1580–1590, 2021.

[62] Qi Zhu, Hao Wei, Bunyamin Sisman, Da Zheng, Christos Faloutsos, Xin Luna Dong, and Jiawei Han. Collective multi-type entity alignment between knowledge graphs. In *Proceedings of The Web Conference 2020*, pages 2241–2252, 2020.

Given an intent, please convert it into one of the expression format defined as follows:
Format1: <protocol> routing. Traffic from source to destination prefers path1 over path2.
Format2: <protocol> routing. Traffic from source to destination must follow the path.
Format3: <protocol> routing. Traffic from source to destination is load-balance between path1 and path2.
…

For example:
==========
Intent: Each router runs the BGP protocol. Traffic from from R1 to R4 prioritizing taking (R1, R3, R4) over (R1, R2, R4).
Normalization: BGP routing. Traffic from source to destination prefers path1 over path2.

…
==========

Intent: All routers are configurated with BGP protocol. When traffic is sent from R_A to R_F, it prioritizes the route (R_A, R_B, R_D, R_F) over route (R_A, R_C, R_E, R_F).

Figure 10: Example prompt for normalizing intents.

Given a configuration intent and a network topology, please extract a role attribute for each node in the topology from the intent.

For example:
==========
Intent: Each router runs the BGP protocol . Traffic from R1 to R5 prefers the path (R1, R3, R5) over (R1, R2, R4, R5).
Topology: {"routers": [R1, R2, R3, R4, R5], "links": [{"node1": {"name": "R1"}, "node2": {"name": "R2"}},...]}
Role attributes: {"R1":route preference, "R2": relay, "R3": relay, "R4": relay, "R5": destination, "R6": non-involvement}

Intent: ...
Topology: ...
Role attributes: ...

Intent: ...
Topology: ...
Role attributes: ...
…
==========

Intent: ...
Topology: ...
Role attributes:

Figure 11: Example prompt for extracting attributes for each device in the topology from the intent.

Given a routing requirement type, the source and destination of traffic flows, as well as the possible transmission paths of traffic flows, generate an intent description to express this requirement.

Requirement definition:
Static route or Simple requirement: given a path *p*, this requirement specifies that traffic from a source to a destination must pass the path *p*.
ECMP requirement: given two path *p1* and *p2*, this requirement specifies that traffic from a source to a destination is load-balanced between path *p1* and *p2* .
….

Examples:
==========
The source and destination of traffic flows: R1, R9
Possible transmission paths: (R1, R3, R5, R7, R9), (R1, R4, R8, R9)
Requirement: OSPF ECMP

Intent description: Configure OSPF routing process on each device and implement an OSPF routing policy to load-balance traffic from R1 to R9 across the paths (R1, R3, R5, R7, R9) and (R1, R4, R8, R9).

…
==========

The source and destination of traffic flows: routerA, routerF
Possible transmission paths: (routerA, routerC, routerD, routerF), (routerA, routerB, routerE, routerF)
Requirement: OSPF ECMP

Intent description:

Figure 12: Example prompt for generating intent description.

A Prompt Construction

In this section, we provide detailed examples of prompt construction involved in configuration synthesis with our CEGS.

A.1 Intent preprocessing prompt

Give a user intent, we normalize it using unified expression formats. We utilize GPT-4o to convert it into this format using a few-shot prompt. The prompt includes three parts: the definition of unified expression formats, few examples, and the user intent. Figure 10 illustrates an example of such a prompt.

A.2 Role extraction prompt

Given a pair of an intent and a topology, we leverage GPT-4 to extract a role attribute for each device in the topology from the intent using prompts with few-shot examples to embed the intent to the topology graph. Figure 11 shows a detailed example prompt.

A.3 Intent conversion prompt

CEGS utilizes the formal tool to verify and fill in templates. For this, CEGS first uses GPT-4o to convert the user intent from natural language to formal specification defined by the formal tool using a few-shot prompt. A example prompt is shown in Figure 13.

A.4 Template generation prompt

Figure 14 shows an example of prompt for LLM to generate configuration templates. The prompt includes six parts: background description, configuration example, target scenario, association relation between devices in the target and example topologies, format definition of the configuration template, and instructions. The configuration example consists of a high-level intent, a topology specified in a JSON dictionary and the configurations of all nodes in the topology. The target network scenario includes the user intent f^i_{user} and target topology T_{tgt}. The instruct indicates which devices in the topology the LLM should generate templates for.

A.5 Intent construction prompt

In the evaluation in Sec.4, we generate intent description for each target scenario using GPT-4o. We first ask NetOps experts draft few examples for each type intent. We then utilize GPT-4o to generate intent description using a few-shot prompt, as shown in Figure 12.

B Detailed Experimental Results

In this section, we provide detailed experiment results of ablation study and comparison between different LLMs.

B.1 Ablation study

We conduct detailed ablation studies on the Querier's recommendation strategies and Classifier's classification strategies to further evaluate the performance of Querier and Classifier.

Ablation study of recommendation strategies. Table 7 shows the ablation study results on the recommendation strategies of the Querier. The results convey four important insights. *First*, without adopting intent preprocessing, the Querier achieves 85% accuracy in example recommendations. Owing to inaccurate examples, CEGS is unable to successfully synthesize correct configurations for all networks. This is due to the flexibility of natural language expression, which leads to diverse expressions of the same type intent, posing a challenge for accurate identification. In constract, by adopting an intent standardization preprocessing method, the Querier shows significant improvement in recommending configuration examples with 100% accuracy, enabling our CEGS to successfully synthesize correct configurations for all networks within an average of 28 loops. In addition,

Given that intents in natural language, your task is to transform the intents into the responding formal specifications. There are three types of formal specifications: PathReq(), ECMPPathsReq(), and PathOrderReq().

1. PathReq() is used to represent a specific transmitting path requirement, which is defined as: PathReq(<protocol>, <destination>, <path>, False). It indicates that traffic destined for <destination> transmits following the path <path>.

2. ECMPPathsReq() is used to represent a load-balance requirement, which is defined as: ECMPPathsReq(<protocol>, <destination>, [PathReq(<protocol>, <destination>, <path1>, False), PathReq(<protocol>, <destination>, <path2>, False)], False). It indicates that traffic destined for <destination> is load-balanced along the path <path1> and the path <path2>.

3. PathOrderReq() is used to represent a path-preference requirement, which is defined as: PathOrderReq(<protocol>, <destination>, [PathReq(<protocol>, <destination>, <path1>, False), PathReq(<protocol>, <destination>, <path1>, False)], False). It indicates that traffic destined for <destination> prefers the path <path1> over the path <path2>.

In these formal specifications, <protocol> represents the routing protocol used by the router, with a value range of [Protocols.BGP, Protocols.OSPF, Protocols.Static].

==========

For example:

Intent: BGP routing. The traffic from RouterA to RouterD prefers the path of <RouterA, RouterB, RouterD> over the path of <RouterA, RouterC, RouterD>.

Specification: [PathOrderReq(Protocols.BGP, 'RouterD', [PathReq(Protocols.BGP, 'RouterA', [RouterA, RouterB, RouterD], False), PathReq(Protocols.BGP, 'RouterD', [RouterA, RouterC, RouterD], False)], False)

…

==========

Intent: All routers are configurated with BGP protocol. When traffic is sent from R_A to R_F, it prioritizes the route (R_A, R_B, R_D, R_F) over route (R_A, R_C, R_E, R_F).

Specification:

Figure 13: Example prompt for converting intents into formal specifications.

compared with not using the second-stage recommendation, the Querier uses graph similarity measurement method to recommend the most relevant examples, allowing CEGS to synthesize the correct configuration faster. This is because the graph similarity method can find examples that share similar topological features to the target topology, thus guiding the Classifier better match the example to devices.

Ablation study on classification strategies. Table 8 shows the ablation study results on the classification strategies of the Classifier. We observe that the complete Classifier achieves the best performance. The Classifier achieves 90% accuracy by relaying on exact matching to establish association between devices in the target and example topologies based device's roles. This is because, in some cases, the device configuration is also highly affected by its neighborhood while exact matching cannot measure the neighborhood similarity between devices. By combining exact matching with neighborhood similarity measurement method, the Classifier enhances association establishment accuracy and synthesis speed, allowing CEGS to synthesize correct configurations with an average reduction of 8 loops.

B.2 Comparison of different LLMs in configuration generation

We evaluate the performance of different LLMs for configuration generation, including LLaMA-2-7B, LLaMA-2-13B, Mixtral-8x7B-Instruct, Gemma-1.1-7B-IT, LLama-3-70B-Instruct, and GPT-4o. Table 9 shows the comparison results. We leverage these LLMs to generate configurations under two target scenarios. One scenario consists of a small-scale star network with 5 nodes, and 6 simple no-transit intents. Another scenario consists of a larger network with 22 nodes, and one complex path-preference intent. We observed that the performance of GPT-4o is the best, followed by LLama-3-70B-Instruct. Therefore, in our evaluation, we leverage GPT-4o to generate configuration templates, due to its outstanding configuration generation ability.

Table 7: Ablation study results on the recommendation strategies of the Querier. U stands for unsuccessful generation.

Intent similarity	Intent preprocessing	Intent graph similarity	Accuracy	Loops
✓	×	×	85%	200(U)
✓	✓	×	100%	28
✓	✓	✓	100%	10

Table 8: Ablation study results on the classification strategies of the Classifier.

Exact matching	Neighborhood similarity	Accuracy	Loops
✓	×	90%	18
✓	✓	100%	10

C A case study

We provide a complete case to show our CEGS how to synthesize correct configurations, given a target network scenario. Figure 15 shows a target network scenario consisting two user intents and a target network topology.

C.1 Identifying configuration examples

We utilize the Querier to recommend the most relevant example for each user intent. Take intent f_1 as an example, we elaborate on how we use the Querier to recommend a configuration example. First, we normalize the intent f_1 into an unified expression format of *"BGP routing. Traffic from source to destination prefers path1 over path2"* using GPT-4o

You are an expert in writing network configurations. We provide you with a configuration example. Given a target network scenario including a target topology specified in a JSON dictionary and the target intent, your task is to generate the configuration template for each device in the target topology, following the configuration example, so that the target intent is satisfied.

Configuration example:

Topology T1: {" devices": [" U_A " : {AS: 100}, ...], " links": [{"node1": {"name": "U_A", "interface": "Ethernet2/0/0", "ip address": "10.17.43.1/24"}, "node2": {{"name": "U_B", "interface": "Ethernet2/0/0", "ip address": "10.17.43.2/24}}, ...]}

Intent I1: All routers are configured with the BGP protocol. When traffic originates from router U_A and is destined for U_D, it prioritizes the path (U_A, U_B, U_D) over (U_A, U_C, U_D).

Configurations:

Configuration of U_A : hostname U_A \n interface Ethernet2/0/0\n ip address 10.17.43.1/24 \n …

……

Target network scenario:

Target topology T2: {" devices": [" R_A " : {AS: 10}, ...], " links": [{"node1": {"name": "R_A", "interface": "Ethernet1/0/0", "ip address": "10.25.32.5/24"}, "node2": {{"name": "R_B", "interface": "Ethernet1/0/0", "ip address": "10.25.32.10/24}}, ...]}

Target Intent I2: Each router runs the BGP protocol. Traffic form router R_G to R_D selects the path (R_G, R_F, R_D) as preferred over (R_G, R_E, R_D).

Association relations between routers in the target topology T2 and routers in example topology T1:

$R_A : U_E$, $R_B : U_E$, $R_C : U_F$, $R_D : U_F$, $R_E : U_C$, $R_F : U_B$, $R_G : U_A$

Format requirements for configuration template:

This configuration template considers OSPF cost settings and BGP route-map configuration as templates, while the rest remains as specific configurations. The format of OSPF cost configuration template and BGP route-map configuration template are:

1. OSPF cost configuration template: this template leaves the ospf cost as symbol, represented by [value].

2. BGP route-map policy configuration template:

1) Follow the import route-map name and export route-map name, which are defined as 'RMap_{router}_from_{neighbor}' and 'RMap_{router}_to_{neighbor}', respectively, (where router and neighbor should be replaced with the router's name and its neighbor's name.).

2) The route-map configuration template should leave policy parameters (e.g., local preference value, community value) as symbolic which is represented by [value]. For instance:

```
ip community list 1 [value]
route-map RMap_RouterF_from_RouterM permit 10
 match community 1
 set local-preference [value]
 set community [value] additive
```

3) If the route-map configuration uses the community or prefix, you must first provide the corresponding definition of "ip community list" or "ip prefix-list". The exclamation mark (!) should be used to separate the route-map configuration of different neighbors.

Please directly generate the configurations for routers {R_B, R_F, …} in the target topology T2, referring to the configurations of their association nodes in the example topology T1, so that the target intent I2 is satisfied. Note: You cannot provide any comments in the response, nor provide any comments in the generated configuration.

Figure 14: Example prompt for generating configuration templates.

Table 9: Comparison results of different LLMs for configuration generation. U stands for unsuccessful generation.

Target network scenario	Models					
	LLaMA-2-7B	LLaMA-2-13B	Mixtral-8x7B-Instruct	Gemma-1.1-7B-IT	LLaMA-3-70B-Instruct	GPT-4o
5-node network, 6 no-transit intents	50(U)	50(U)	50(U)	50(U)	5	2
22-node network, 1 path-preference intent	50(U)	50(U)	50(U)	50(U)	30	4

with the few-shot prompt shown in Figure 10. The intent of each configuration example is also normalized into a specific expression format during the configuration example parsing process. For example, the intents of configuration examples shown in 16 and 17, also are normalized as *"BGP routing. Traffic from source to destination prefers path1 over path2"*. These two examples, whose intent semantically match intent f_1 with a similarity score of 1, are considered potential reference examples for intent f_1. We then construct the corresponding user intent graph and two example intent graphs, and using GraphSAGE to calculate the similarity between them. The similarity between the example intent graph of configuration example in Figure 16 and the user intent graph is 0.86, while the similarity between the example intent graph of configuration example in Figure 17 with the user intent graph is 0.75. Thus, we select the configuration example z_1 in Figure 16 as the example associated with intent f_1. Similarly, we recommend the best configuration example for intent f_2. The configuration example in Figure 17 is chosen due to its higher similarity between its example intent graph and the user intent graph corresponding to intent f_2. From Figures 15, 16, and 17, we can observe that when considering intent f_1, the example intent graph corresponding to example z_1 is more similar to the intent graph for f_1, particularly since the nodes R_A, R_B, and R_D in Figure 15 closely match the node attributes and neighborhood relations of nodes V_A, V_B, and V_D in Figure 2. However, only node R_A exhibits high similarity to node U_A in 17. Therefore, the user intent graph for f_1 is closer to the example intent graph of configuration example z_1.

Intents:

Intent f_1: Each router runs the BGP protocol. Traffic form router R_A to R_E selects the path (R_A, R_C, R_E) as preferred over (R_A, R_B, R_D, R_E).

Intent f_2: Each router runs the BGP protocol. Traffic form router R_G to R_D selects the path (R_G, R_F, R_D) as preferred over (R_G, R_E, R_D).

Topology:

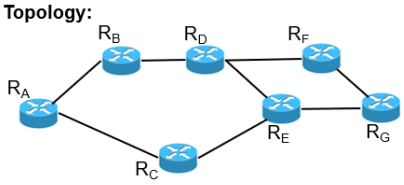


Figure 15: A target network scenario.

Intent: All routers are configured with the BGP protocol. When traffic originates from router V_A and is destined for V_F, it prioritizes the path (V_A, V_B, V_D, V_F) over (V_A, V_C, V_E, V_F).

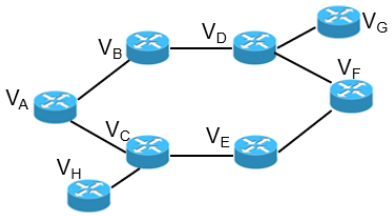


Figure 16: Configuration example z_1 for the user intent f_1.

C.2 Establishing the association between devices in the target and example topologies

For each user intent f_1, we utilize the Classifier to establish the association between the device in the target topology and those in the topology of example z_i corresponding the intent f_i. Figure 18 shows the association results with respect to each example.

C.3 Generating configurations

We first leverages GPT-4o to generate configuration templates for each device in the target topology with respect to each user intent in turn. Figure 14 shows the example prompt for generating configuration templates. Take router R_D in the target topology as an example, Figure 19 and Figure 20 show the configuration template for router R_D with respect to intent f_1 and intent f_2, respectively. We use the Syntax Verifier, LAV, and GFV to verify these templates. We then merge the configuration templates for each device with respect to the two user intents. Figure 21 shows the merged templates for router R_D. Finally, we use the Formal Synthesizer to specify the specific parameters in the templates. Figure 22 shows the full configuration of router R_D.

D Discussion on user guide and generalization for CEGS

In this section, we first provide a user guide detailing how network operators can use our CEGS to synthesize configurations in the real network setting. We then provide a comprehensive guide detailing the steps a network operator needs to follow to generalize CEGS with new intents.

Intent: All routers are configured with the BGP protocol. When traffic originates from router U_A and is destined for U_D, it prioritizes the path (U_A, U_B, U_D) over (U_A, U_C, U_D).

Topology:

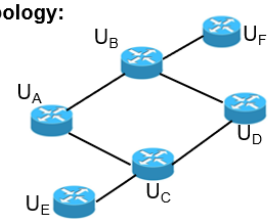


Figure 17: Configuration example z_2 for the user intent f_2.

Association relations between devices in the target topology and those in the topology of the example z_1:

$R_A : V_A$, $R_B : V_B$, $R_C : V_C$, $R_D : V_D$, $R_E : V_F$, $R_F : V_G$, $R_G : V_G$

Association relations between devices in the target topology and those in the topology of the example z_2:

$R_A : U_E$, $R_B : U_E$, $R_C : U_F$, $R_D : U_F$, $R_E : U_C$, $R_F : U_B$, $R_G : U_A$

Figure 18: Association between devices in the target and example topologies.

D.1 A user guide

To utilize CEGS to synthesize configurations for devices in a real network, the network operator needs to prepare the operator intents, network topology in JSON format, and the relevant device documentation. For example, consider a cloud network consisting of three Autonomous Systems (ASes), as shown in Figure 23(b). The operator's network, AS1000, includes four routers: RouterA, RouterB, RouterC, and RouterD. This network is connected to one customer peer AS2000, and two external peers, AS3000 and AS4000. The operator aims to manage traffic flow transmission from the customer peer to two external peers. For this, the operator first needs to draft routing intents in natural language for these traffic flows. Figure 23(a) provides the intent example. The operator then should describe the network topology in JSON format, as illustrated in Figure 23(b). Finally, the operator needs to prepare the device documentation, which should include a configuration guide that contain many configuration examples. At the same time, the operator must ensure that at least one configuration example related to each specified intent is included in the documentation. Figure 23(c) provide a sample documentation example from the cisco[6].

D.2 How to generalize CEGS?

Our CEGS supports static route, OSPF, and BGP configurations across variety of routing intents supported by the Formal Synthesizer (NetComplete). To generalize CEGS to new in-

[6] https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/15-mt/iro-15-mt-book/iro-cfg.html#GUID-EF90EB2B-A283-4F7A-B044-C44BAE9F4E0A

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
ip community-list 1 permit 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RF 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 19: Configuration template of router R_D for intent f_1.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
ip community-list 1 permit 'EMPTY?Value'
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 'EMPTY?Value' additive
 set local-preference 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 21: Configuration template of router R_D for intent f_1 and intent f_2.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 'EMPTY?Value' additive
 set local-preference 'EMPTY?Value'
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 'EMPTY?Value'
 set community 'EMPTY?Value' additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
route-map RMap_RD_to_RE 'EMPTY?Value' 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 20: Configuration template of router R_D for intent f_2.

```
hostname RD
!
interface Ethernet1/0/0
 ip address 10.2.0.4 255.255.255.254
 description "To RB"
 speed auto
 duplex auto
!
interface Ethernet2/0/0
 ip address 10.5.2.1 255.255.255.254
 description "To RE"
 speed auto
 duplex auto
!
interface Ethernet3/0/0
 ip address 10.10.3.1 255.255.255.254
 description "To RF"
 speed auto
 duplex auto
!
interface Ethernet4/0/0
 ip address 10.2.10.12 255.255.255.254
 description "To PeerRD"
 speed auto
 duplex auto
!
ip community-list 1 permit 100:0
!
route-map RMap_External_RD_from_PeerRD permit 10
 set community 100:0 additive
 set local-preference 100
!
route-map RMap_RD_from_RB deny 10
!
route-map RMap_RD_from_RE permit 10
 match community 1
 set local-preference 100
 set community 100:0 additive
!
route-map RMap_RD_from_RF deny 10
!
route-map RMap_RD_to_RB permit 100
!
route-map RMap_RD_to_RE permit 100
!
route-map RMap_RD_to_RE permit 100
!
router bgp 40
 no synchronization
 bgp log-neighbor-changes
 neighbor 10.2.10.12 remote-as 10010
 neighbor 10.2.10.12 description "To PeerRD"
 neighbor 10.2.10.12 advertisement-interval 0
 neighbor 10.2.10.12 soft-reconfiguration inbound
 neighbor 10.2.10.12 send-community
 neighbor 10.2.10.12 route-map RMap_External_RD_from_PeerRD in
 neighbor 10.2.10.12 route-map RMap_RD_to_PeerRD out
 neighbor 10.2.0.5 remote-as 20
 neighbor 10.2.0.5 description "To RB"
 neighbor 10.2.0.5 advertisement-interval 0
 neighbor 10.2.0.5 soft-reconfiguration inbound
 neighbor 10.2.0.5 send-community
 neighbor 10.2.0.5 route-map RMap_RD_from_RB in
 neighbor 10.2.0.5 route-map RMap_RD_to_RB out
 neighbor 10.0.0.9 remote-as 50
 neighbor 10.5.2.2 description "To RE"
 neighbor 10.5.2.2 advertisement-interval 0
 neighbor 10.5.2.2 soft-reconfiguration inbound
 neighbor 10.5.2.2 send-community
 neighbor 10.5.2.2 route-map RMap_RD_from_RE in
 neighbor 10.5.2.2 route-map RMap_RD_to_RE out
 neighbor 10.10.3.2 remote-as 60
 neighbor 10.10.3.2 description "To RF"
 neighbor 10.10.3.2 advertisement-interval 0
 neighbor 10.10.3.2 soft-reconfiguration inbound
 neighbor 10.10.3.2 send-community
 neighbor 10.10.3.2 route-map RMap_RD_from_RF in
 neighbor 10.10.3.2 route-map RMap_RD_to_RF out
!
```

Figure 22: Full configuration of router R_D complying with intent f_1 and intent f_2.

tents for other protocols or network techniques (e.g., VLAN) on specific vendor devices, the user extends CEGS by following six steps:

Configuration example preparation. The user first needs to collect relevant device documentation which contain configuration guide and ensures the configuration guide cover some configuration examples related to the target scenarios.

GraphSAGE training. The Querier and Classifier depend on a GraphSAGE model to recommend examples and build association between devices in the target and example topologies. To achieve this, the user needs to train a GraphSAGE model. First, the user should construct a intent graph dataset that contains numerous matching pairs of intent graphs. These intent graphs should cover the relevant target scenarios, including similar intents and topology structures. Each data in the dataset consists of a pair of intent graphs and the labels indicating the associations between nodes in the two graphs. The user then trains the GraphSAGE model on this dataset using the training method described in Sec.3.2.3. The user may need to tune the GraphSAGE model by adjusting hyperparameters such as the number of layers, number of neurons, and learning rate.

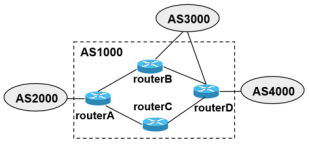
Prompt crafting. The user needs to craft the prompt for the LLM to generate configuration templates. The prompt, as shown in Figure 14, consists of six parts: background description, configuration example, target network scenario, association relation between devices in the target and example topologies, format definition of the configuration template, and instructions. The users should tailor the template format definition according to the configuration templates they need to generate. The user can design customized instructions as needed. In addition, the user also needs to craft other prompts, such as those for intent normalization and role extraction, as described in A.

LAV extension. CEGS develops the LAV to locally verify whether each device template complies with the the target topology and user intent descriptions, such as interface configurations and protocol basic configurations. The user can easily

Intent 1: Configure BGP process on each routers to implement the policy: traffic from AS2000 to AS3000 prefers the path <AS2000, routerA, routerB, AS300> over the path < AS2000, routerA, routerC, routerD, AS3000 >.

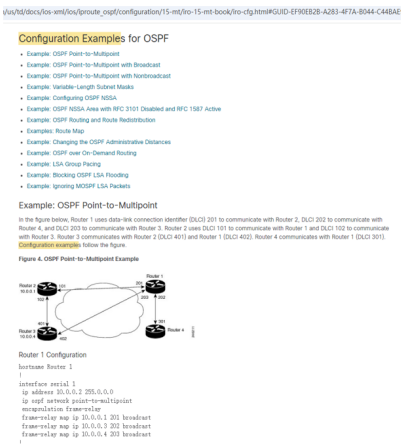
Intent 2: Configure OSPF process on each routers to implement the policy: traffic from AS2000 to AS400 is load-balanced between the path <AS2000, routerA, routerB, routerD, AS4000> and the path <AS2000, routerA, routerC, routerD, AS3000>.

(*a*) Intent description




```
{
    "routers": ["routerA":{"AS": 1000,...}, ...],
    "links": [
      {
          "node1": {
              "name": "routerA",
              "interface": "GigabitEthernet1/0/0",
              "ip address": "10.90.17.1/24"
          },
          "node2": {
              "name": "routerB",
              "interface": "GigabitEthernet1/0/0",
              "ip address": "10.90.17.2/24"
          }
      },
      ...
    ]
}
```

(*b*) Topology description


Configuration Examples for OSPF

- Example: OSPF Point-to-Multipoint
- Example: OSPF Point-to-Multipoint with Broadcast
- Example: OSPF Point-to-Multipoint with Nonbroadcast
- Example: Variable-Length Subnet Masks
- Example: Configuring OSPF NSSA
- Example: OSPF NSSA Area with RFC 3101 Disabled and RFC 1587 Active
- Example: OSPF Routing and Route Redistribution
- Examples: Route Map
- Example: Changing the OSPF Administrative Distances
- Example: OSPF over On-Demand Routing
- Example: LSA Group Pacing
- Example: Blocking OSPF LSA Flooding
- Example: Ignoring MOSPF LSA Packets

Example: OSPF Point-to-Multipoint

In the figure below, Router 1 uses data-link connection identifier (DLCI) 201 to communicate with Router 2, DLCI 202 to communicate with Router 4, and DLCI 203 to communicate with Router 3. Router 2 uses DLCI 101 to communicate with Router 1 and DLCI 102 to communicate with Router 3. Router 3 communicates with Router 2 (DLCI 401) and Router 1 (DLCI 402). Router 4 communicates with Router 1 (DLCI 301). Configuration examples follow the figure.

Figure 4. OSPF Point-to-Multipoint Example

Router 1 Configuration

```
hostname Router 1
!
interface serial 1
 ip address 10.0.0.2 255.0.0.0
 ip ospf network point-to-multipoint
 encapsulation frame-relay
 frame-relay map ip 10.0.0.1 201 broadcast
 frame-relay map ip 10.0.0.3 202 broadcast
 frame-relay map ip 10.0.0.4 203 broadcast
!
```

(*c*) Device documentation

Figure 23: Example input for CEGS.

adapt the LAV for other protocols or network techniques by implementing checks for the protocol-specific basic settings or network technique settings, as well as interface attribute settings when involved.

GFV extension. CEGS develops the GFV to perform global verification on templates across all devices, ensuring that configurations with interdependencies and cross-device impacts are properly validated. For example, BGP route-map policy configurations on different devices influence one another and work together to achieve the intended global routing policy. The user should extend the GFV to globally validate configurations that affect each other across all devices involved in fulfilling the intended network behavior. The GFV provides localized feed about configuration errors for LLM correction. The user can replace the LAV or GFV using other verification tools that can provide localization feedback, indicating which specific configuration on which devices are incorrect.

Formal Synthesizer selection. The user needs to select or develop appropriate Formal Synthesizer that is capable of filling correct parameters in the templates to achieve the intent.

D.3 Why does CEGS need to use formal methods?

In configuration synthesis tasks, it is a significant challenge for LLMs to accurately calculate network policy parameters based on the entire network information. For example, achieving a load-balancing intent under the OSPF protocol requires accurately calculating the cost of each link in the network, which is difficult for LLM reasoning. Therefore, it is essential to use formal method to calculate network policy parameters to guarantee the configuration correctness.