GRAPHGUARD: GRAPH-CONSTRAINED NEURO-SYMBOLIC SYNTHESIS OF NETWORK DEFENSE CONFIGURATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Translating high-level intents into executable configurations is essential for modern network defense. Existing Large Language Model (LLM)- and Retrieval-Augmented Generation-based methods still rely heavily on unconstrained LLM reasoning to assemble configuration fields and instantiate runtime parameters, making reliable configuration synthesis difficult. This difficulty stems from fragmented configuration knowledge across heterogeneous devices, interdependent configuration choices tied to dynamic network contexts, and generated configurations that may violate deterministic constraints or deployment requirements. To address these issues, we propose GraphGuard, a graph-constrained neuro-symbolic framework that translates high-level defense intents into verified configurations for heterogeneous devices. GraphGuard organizes configuration knowledge into a unified typed five-slot space, uses graph-constrained joint synthesis to compose compatible configuration components, and deterministically instantiates runtime-dependent parameters under explicit operational constraints. A structured verifier then guides bounded repair through graph-defined legal edits, while a backend-independent Intermediate Representation (IR) enables device-specific compilation. Experiments in heterogeneous DDoS mitigation environments show that GraphGuard achieves 93.45% Joint Exact Match and 75.41% Safe Synthesis Success, with a 1.50% unsafe acceptance rate on infeasible instances, while maintaining practical end-to-end synthesis latency.

1 INTRODUCTION

Modern network defense increasingly relies on high-level intents to state security goals, allowing operators to express what a network should achieve rather than how to configure it (Beckett et al., 2016; El-Hassany et al., 2018; Ramanathan et al., 2023). While existing automation tools can efficiently handle simple and static intents (e.g., blocking a specific malicious IP), synthesizing configurations for complex and dynamically changing defense intents remains challenging (Beurer-Kellner et al., 2022; Liu et al., 2025). For instance, translating a natural-language strategy such as "limit high-frequency sessions with abnormal Transport Layer Security (TLS) fingerprints" into executable commands requires precisely defining match conditions, dynamic numerical thresholds, and specific execution scopes, making this translation process complex, context-dependent, and tightly coupled to the configuration schemas and device capabilities of heterogeneous defense equipment.

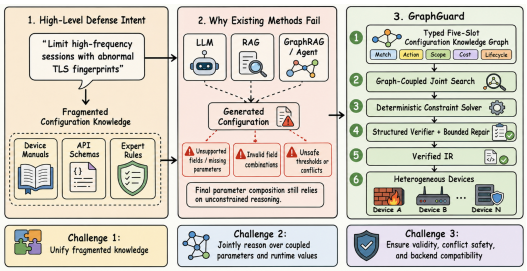


Figure 1: GraphGuard translates high-level defense intents into verified configurations.

Large Language Models (LLMs) have achieved widespread success in natural language processing tasks and shown promise in automating network configuration generation (Wang et al., 2024; Mondal et al., 2023). However, when applied to strict configuration synthesis, they may halluci-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

nate unsupported fields, omit required parameters, or produce syntactically valid but operationally unsafe configurations (Wang et al., 2024; Mondal et al., 2023). To mitigate these issues, Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) augments generation with retrieved external documents, which can include device manuals in configuration synthesis. Standard RAG, however, mainly retrieves unstructured text chunks and does not explicitly represent dependencies among configuration parameters, making it difficult to assemble independently retrieved fields into a complete and valid configuration. Although GraphRAG methods (He et al., 2024; Gutiérrez et al., 2024; Edge et al., 2024; Guo et al., 2025; Luo et al., 2025b) can expose structural relations among retrieved facts, existing pipelines generally use graph-retrieved knowledge primarily as context and still rely on the LLM to produce the final output. Without deterministic constraints, LLMs cannot reliably determine cross-field compatibility (e.g., whether a TLS-fingerprint match can be combined with a specific rate-limiting action and execution scope) or accurately infer runtime numerical values (e.g., a rate-limit threshold under current traffic and available CPU) (Poesia et al., 2022; Hè et al., 2026). Agentic frameworks (Yao et al., 2023; Luo et al., 2025a) enable multi-turn reasoning, but they cannot enforce execution-time constraints without deterministic external feedback. Therefore, existing methods fail to achieve accurate configuration generation for heterogeneous network equipment.

Synthesizing safe defense configurations from natural language raises three key challenges. **First, configuration knowledge for heterogeneous defense devices is difficult to unify into an executable representation.** Because heterogeneous devices use different syntax and parameter definitions, and relevant knowledge is scattered across device manuals, API schemas, and expert rules, converting this fragmented information into a consistent, machine-readable format remains difficult (Wang et al., 2024; Liu et al., 2025). **Second, configuration parameters are strongly interdependent and depend on dynamic network contexts.** A selected match condition restricts the permissible defense actions and execution scopes, while numerical parameters, such as rate-limit thresholds, must be dynamically computed from real-time traffic and available resources (e.g., CPU load or session capacity) rather than inferred through unconstrained language-model reasoning (Ramanathan et al., 2023; Liu et al., 2025). **Third, semantic flexibility in generation often introduces hidden execution risks.** Even if a synthesized configuration appears structurally sound, it may contain invalid parameters that fail to compile on target devices (Wang et al., 2024; Mondal et al., 2023), or inadvertently block critical business traffic (e.g., overlapping with VIP whitelists) when deployed, leading to severe operational disruptions (Al-Shaer & Hamed, 2004; Beckett et al., 2017; Prabhu et al., 2020).

To address these challenges, we propose **GraphGuard**, a graph-constrained neuro-symbolic framework for translating natural-language defense intents into verified device configurations. GraphGuard organizes heterogeneous configuration knowledge into a unified five-slot representation consisting of *Match, Action, Scope, Cost, and Lifecycle*, providing a common configuration space across different devices. Given a defense intent and the current network context, an LLM interprets the request and identifies relevant configuration candidates, while the configuration graph guides their combination into a coherent configuration. Runtime-dependent parameters (e.g., rate-limit thresholds) are then determined under explicit resource and policy constraints rather than directly generated by the LLM. Before release, the resulting configuration is verified against configuration and deployment constraints; invalid configurations are repaired only through graph-defined legal edits and are subsequently re-verified. Finally, the verified configuration is represented using a backend-independent Intermediate Representation (IR) and compiled into device-specific configurations. We evaluate GraphGuard in heterogeneous distributed denial-of-service (DDoS) mitigation environments, covering end-to-end synthesis quality, component contributions, and related generalization and deployment settings. The results show that GraphGuard consistently improves configuration correctness and safe synthesis while retaining practical end-to-end latency.

Our main contributions are threefold:

- We introduce a unified typed configuration representation that organizes heterogeneous configuration primitives, parameter domains, dependencies, conflicts, and backend capabilities into an executable five-slot configuration space, bridging natural-language defense intents and device-specific configurations.

- We develop a graph-constrained synthesis framework that combines LLM intent understanding with joint configuration composition and deterministic parameter instantiation,

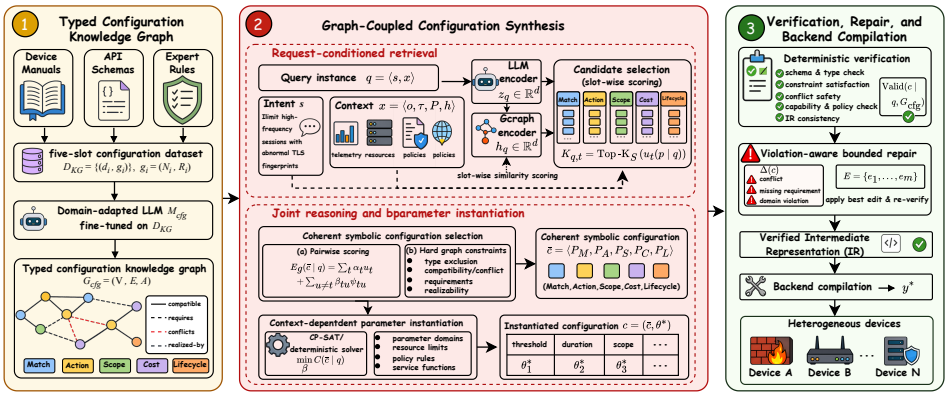


Figure 2: Overview of GraphGuard for graph-guided configuration synthesis, verification, repair, and backend compilation.

enabling interdependent configuration choices to adapt to runtime traffic, resources, policies, and device capabilities.

- We introduce verifier-guided bounded repair and backend-independent IR compilation, restricting corrective generation to graph-defined legal edits and preserving verified configuration semantics across heterogeneous execution backends.

2 Related Work

Intent-Based Network Configuration Synthesis. Declarative network languages provide abstractions and composition rules for policies (Foster et al., 2011; Monsanto et al., 2013; Anderson et al., 2014). Propane, NetComplete, and Aura translate routing objectives, partial configurations, or high-level policies into device configurations (Beckett et al., 2016; El-Hassany et al., 2018; Ramanathan et al., 2023). Example-driven synthesis extends this approach across topologies and configuration instances (Beurer-Kellner et al., 2022; Liu et al., 2025). Their guarantees rely on formal policies, examples, or structured sketches. GraphGuard instead addresses natural-language defense intents whose structural choices and numerical parameters depend jointly on runtime state. DMKG (Niu et al., 2026) connects traffic evidence with attack-defense knowledge. Its strategy-level reasoning is complementary to the executable configuration synthesis studied here.

Knowledge- and Constraint-Guided Generation. LLMs translate natural-language requirements into network configurations but can produce invalid outputs (Mondal et al., 2023; Wang et al., 2024). RAG supplies documentary evidence (Lewis et al., 2020), while graph-based retrieval exposes relational and higher-order knowledge (Edge et al., 2024; Luo et al., 2024; 2025b). Agentic retrieval further supports multi-turn interaction (Luo et al., 2025a), but retrieved context alone does not enforce configuration validity. Synchromesh constrains generation through semantic checks (Poesia et al., 2022), and ConstraintLLM combines constraint-aware retrieval with solver-guided correction (Shi et al., 2025). GraphGuard builds on this separation of interpretation and feasibility. Its focus is a typed configuration space that couples cross-slot selection, runtime parameter solving, and verified backend compilation, rather than the combination of an LLM and a solver alone.

3 Preliminaries

This section introduces the basic concepts and formulates the defense configuration synthesis task, including defense intents and runtime context, the typed configuration knowledge graph, and the five-slot configuration representation.

Defense Intent and Runtime Context. A defense intent specifies the desired network defense behavior in natural language. Let s denote such an abstract intent. This intent s must be realized under a specific runtime context $\mathbf{x} = \langle o, \mathbf{r}, \mathcal{P}, \mathbf{h} \rangle$, where o contains network observations and attack

evidence, $\mathbf{r}$ records the current load and available resources, $\mathcal{P}$ contains the applicable business, security, and topology policies, and $\mathbf{h}$ records the capabilities, versions, and health status of the available execution backends. To jointly capture the requested behavior and its operational environment, we encapsulate them into a single configuration synthesis instance $q = \langle s, \mathbf{x} \rangle$.

Typed Configuration Representation. The typed graph $\mathcal{G}_{\text{cfg}} = (\mathcal{V}, \mathcal{E}, \tau, \rho, \mathcal{A})$ contains reusable prototypes, typed relations, and attributes. The node-type mapping τ assigns prototypes to $\mathcal{T} = \{M, A, S, C, L\}$: Match, Action, Scope, Cost, and Lifecycle. These represent traffic predicates, defense operations, enforcement targets, resource models, and lifecycle controls. The edge-type mapping $\rho : \mathcal{E} \to \mathcal{R}_{\text{edge}}$ assigns each edge a relation type, including compatibility, dependency, conflict, realization, and alternatives. The attribute assignment $\mathcal{A}$ supplies parameter domains, units, applicability conditions, provenance, and backend mappings. A symbolic configuration $\bar{c} = \langle p_M, p_A, p_S, p_C, p_L \rangle$, with $p_t \in \mathcal{V}_t = \{p : \tau(p) = t\}$, fixes the structure. A complete configuration $c = (\bar{c}, \boldsymbol{\theta})$ adds values in the joint declared domain $\mathcal{D}(\bar{c})$.

Problem Definition. Over the candidate space $\mathcal{C}(\mathcal{G}_{\text{cfg}})$, synthesis maximizes intent alignment, cross-slot coherence, and operational utility:

$$c^\star \in \arg \max_{c \in \mathcal{C}(\mathcal{G}_{\text{cfg}})} J(c \mid q) \quad \text{s.t.} \quad \text{Valid}(c \mid q, \mathcal{G}_{\text{cfg}}) = 1, \quad \mathcal{B}_{\text{exec}}(c, q) \neq \varnothing. \tag{1}$$

Validity requires structural, parameter, resource, policy, topology, and lifecycle consistency. The set $\mathcal{B}_{\text{exec}}(c, q)$ contains available backends that can realize c. The task returns an executable realization on a selected backend, or ABSTAIN if no valid configuration exists. Full candidate-space and output definitions are provided in Appendix A.1. This optimization defines the task; bounded search need not recover its global optimum.

4 METHODOLOGY

As shown in Figure 2, GraphGuard constructs a typed configuration knowledge graph offline and performs graph-coupled synthesis, verification, bounded repair, and backend compilation online.

4.1 TYPED CONFIGURATION KNOWLEDGE GRAPH CONSTRUCTION

Prototype Extraction and Graph Admission. GraphGuard constructs the global typed configuration knowledge graph $\mathcal{G}_{\text{cfg}}$ from a collection of unstructured security documents $\mathcal{D}_{\text{raw}}$, which includes device manuals, API schemas, and reviewed expert rules. To train a domain-adapted extractor, we first annotate source segments to build a labeled five-slot configuration dataset $\mathbb{D}_{\text{KG}} = \{(d_i, g_i)\}_{i=1}^{N_{\text{KG}}}$, where $g_i = (\mathcal{N}_i, \mathcal{R}_i)$ contains prototypes and typed relations extracted from segment d_i. A base LLM is fine-tuned on these records, producing the domain-adapted extractor M_{cfg}. Each extracted prototype records a description, parameter schema, applicability conditions, backend mappings, and provenance. Parameter binding distinguishes OBSERVED, SELECTED, OPTIMIZED, and DERIVED values.

Canonicalization normalizes identifiers and units. Schema checks reject malformed records before merging prototypes and inserting relations between canonical endpoints. These checks establish well-formedness, not factual completeness. Extraction records and binding modes are specified in Appendix A.3.

4.2 GRAPH-COUPLED CONFIGURATION SYNTHESIS

Given a configuration synthesis instance $q = \langle s, \mathbf{x} \rangle$, GraphGuard retrieves relevant prototypes from the global knowledge graph $\mathcal{G}_{\text{cfg}}$, selects a coherent symbolic configuration $\bar{c}$, and solves its context-dependent parameters $\boldsymbol{\theta}$ to obtain the complete configuration $c = (\bar{c}, \boldsymbol{\theta})$.

Request-Conditioned Retrieval and Structural Selection. An accessible language encoder maps q to $\mathbf{z}_q$, and a relation-aware GNN maps each prototype to $\mathbf{h}_p$. A unary scorer $u_t(p \mid q)$ combines these representations. Each slot retains the K_s highest-scoring prototypes satisfying applicability predicate $\Gamma_p(\mathbf{x})$, forming $\mathcal{K}_{q,t}$ and the induced subgraph $\mathcal{G}_q$. Although the induced subgraph $\mathcal{G}_q$ filters out irrelevant knowledge, simply selecting the top-scoring prototype for each slot

may yield structurally invalid combinations. GraphGuard therefore combines the unary matching score with typed cross-slot relations to evaluate overall structural coherence. The pairwise potential $\psi_{tu}(p_t, p_u \mid q)$ combines request, prototype, and relation embeddings:

$$E_\Theta(\bar{c} \mid q) = \sum_{t \in \mathcal{T}} \alpha_t u_t(p_t \mid q) + \sum_{t<u} \beta_{tu} \psi_{tu}(p_t, p_u \mid q), \tag{2}$$

where Θ contains the graph-encoder and scorer parameters, and α_t, β_{tu} weight their contributions. The hard predicate Ω_q requires declared dependencies and rejects conflicts. For example, a session-limit *Action* will be rejected if the selected *Scope* lacks stateful connection-tracking capabilities. The admissible space is $\mathcal{F}_q = \{\bar{c} \in \prod_t \mathcal{K}_{q,t} : \Omega_q(\bar{c}) = 1\}$. Typed beam search expands slots in order $M \to A \to S \to C \to L$, checks relations once their endpoints are selected, and retains at most B complete candidates in $\overline{\mathcal{C}}_q \subseteq \mathcal{F}_q$. Pruning is approximate: beam exhaustion does not prove task-level infeasibility. The complete algorithm is provided in Appendix B. Slot supervision and conflicting substitutions train the graph encoder and scorers, not the base LLM, search, or solver. The scorer interfaces, including the treatment of prompt-only candidates, are specified in Appendix D.1.

Context-Dependent Parameter Solving. To instantiate the configuration, GraphGuard anchors the abstract structure to the current network state based on the predefined binding modes. For each symbolic configuration $\bar{c} \in \overline{\mathcal{C}}_q$, it directly binds observed values from the network observations o (e.g., extracting the anomalous TLS JA3 hash), selects graph-declared alternatives (e.g., choosing the `drop` action type), and evaluates registered deterministic functions to derive dependent values. The remaining optimized parameters $\boldsymbol{\theta}$ (e.g., dynamic rate-limit thresholds or block durations) are solved jointly over their declared domain $\mathcal{D}(\bar{c})$:

$$\begin{aligned} \boldsymbol{\theta}^\star(\bar{c}, q) = \arg \min_{\boldsymbol{\theta} \in \mathcal{D}(\bar{c})} \; & C_q(\bar{c}, \boldsymbol{\theta}) \\ \text{s.t.} \quad & U_k(\mathbf{r}) + D_k(\bar{c}, \boldsymbol{\theta}; q) \le \mathrm{Cap}_k(\mathbf{r}), \; \forall k, \\ & \mathrm{Eff}(\bar{c}, \boldsymbol{\theta}; o) \ge \eta(q), \\ & g_j(\bar{c}, \boldsymbol{\theta}; q) \le 0, \; \forall j. \end{aligned} \tag{3}$$

Here, C_q is operational cost, and U_k, D_k, Cap_k are current use, additional demand, and capacity of resource k. The encoded effectiveness requirement is $\eta(q)$, while g_j covers parameter, policy, topology, and lifecycle rules. CP-SAT solves supported finite-domain encodings in declared integer units. Effectiveness is an encoded requirement, not measured traffic-cleaning capacity. `INFEASIBLE` discards the current motif, whereas `UNKNOWN` does not establish infeasibility. A feasible assignment produces $c_{\bar{c}}$, ranked by $J(c_{\bar{c}} \mid q) = E_\Theta(\bar{c} \mid q) - \lambda_C \widetilde{C}_q(c_{\bar{c}})$, where $\widetilde{C}_q$ is normalized cost. Optimality is claimed only when certified by the solver.

4.3 Deterministic Verification, Bounded Repair, and Backend Compilation

Instantiated configurations are processed in descending order of their final objective score $J(c \mid q)$. A fixed verifier set $\mathcal{J}_{\mathrm{ver}}$ comprehensively checks slot types, graph relations, parameter domains, units, resources, policies, topology, and lifecycle constraints. The detected violation set $\Delta(c; q)$ is defined as $\Delta(c; q) = \{v_j(c, q) \mid v_j \in \mathcal{J}_{\mathrm{ver}}, \; v_j(c, q) \neq \varnothing\}$. Each violation identifies the affected slot or parameter, the observed value, the required condition, and its graph or policy reference. For example, a calculated rate-limit duration that exceeds the maximum permitted window triggers an out-of-range parameter violation. A candidate with $\Delta(c; q) = \varnothing$ proceeds directly to backend compilation.

Bounded Graph-Defined Repair. For an invalid instantiated candidate c_k, its violation set $\Delta_k = \Delta(c_k; q)$ and the active subgraph $\mathcal{G}_q$ define a finite legal edit set $\mathcal{E}_{\mathrm{rep}}(c_k, \Delta_k)$. Legal edits may replace a prototype with a graph-declared alternative, assign a value inside a declared domain, or resolve affected parameters. For example, if a deep-inspection *Scope* violates memory constraints, a legal edit may switch it to a less resource-intensive basic-tracking *Scope*. These edits strictly cannot introduce undeclared prototypes, fields, or arbitrary values. A base LLM selects one edit e_k from the legal set, after which all constraints are re-evaluated:

$$\begin{aligned} \mathrm{id}(e_k) &= M_0(q, c_k, \Delta_k, \{\mathrm{id}(e) : e \in \mathcal{E}_{\mathrm{rep}}(c_k, \Delta_k)\}), \\ c_{k+1} &= \mathrm{ReSolve}(\mathrm{Apply}(c_k, e_k), q, \mathcal{G}_q). \end{aligned} \tag{4}$$

Here, M_0 is the frozen base LLM. A response outside the enumerated edit identifiers is rejected, and an LLM-selected edit is not assumed to be globally optimal. An edit is retained only when it decreases the weighted violation severity $V(c;q) = \sum_{v_j \in \mathcal{J}_{\text{ver}}} w_j \mathbb{I}[v_j(c,q) \neq \varnothing]$. Repair terminates when the candidate becomes valid, no legal edit remains, or the maximum round $T_{\max}$ is reached. An unrepaired candidate is discarded.

Backend Compilation and Release. A verified configuration is serialized into a versioned IR and compiled for a supported backend. Release requires schema conformance, interface acceptance, and round-trip consistency of values, units, scope, and lifecycle. GraphGuard returns the first passing realization in pre-repair candidate order and policy-ranked backend order, or ABSTAIN. The release gate and its conditional properties are formalized in Appendices B.2 and B.4. Its guarantee is conditional on encoded constraints and the checked interface, not missing knowledge or unmodeled device behavior.

5 EXPERIMENTS

This section presents the experimental setup, results, and analysis. We address seven complementary research questions: **RQ1:** How effective is end-to-end configuration synthesis? **RQ2:** Which components contribute to structural quality, numerical validity, and safe release? **RQ3:** Does synthesis generalize to unseen compositions and linguistic variations? **RQ4:** How does synthesis adapt to constraint and runtime-state changes? **RQ5:** How do candidate, search, and repair budgets affect performance? **RQ6:** How well do backend interfaces preserve configuration semantics? **RQ7:** How dependent are the gains on the language-model backbone?

5.1 EXPERIMENTAL SETUP

Benchmark and Protocol. GraphGuard-Bench contains 3,030 instances spanning HTTPS, TCP SYN, and UDP floods: 900 IID, 600 compositional-OOD, 600 linguistic-OOD, and 930 robustness cases. Each pairs an intent with observations, resources, policies, and backend state. Acceptable motifs and parameter intervals remain outside method inputs. The ablation suite adds 300 stress cases to IID. The reference setting uses three candidates per slot, beam width eight, and at most three repair rounds; the main and ablation tables pool seeds 11, 23, and 37. Device A and Device B use distinct interface contracts representing Huawei 1908 and NSFOCUS ADS management paths. Backend evaluation checks interface acceptance, scope, and IR–payload consistency, not physical mitigation performance. The split boundaries are provided in Appendix A.

Baselines and Metrics. The LLM evaluation protocol uses Qwen3-Max unless a backbone is explicitly varied. We compare direct generation (Direct LLM and LLM+Full Manuals), retrieval (TextRAG, GraphRAG, PathRAG, and HyperGraphRAG), iterative synthesis (ReAct+Validator), and deterministic Symbolic-Only (Lewis et al., 2020; Edge et al., 2024; Chen et al., 2025; Luo et al., 2025b; Yao et al., 2023). Their information-access controls are specified in Appendix E.1. Slot-F1 and Config-F1 evaluate synthesized candidates before the final release decision, while JEM requires a correct final output. Range Accuracy and NPE evaluate parameter instantiation conditional on emitting a configuration. Conflict-F1 evaluates forbidden-combination detection. SSS requires semantic, constraint, and backend correctness jointly; UAR measures unsafe acceptance on infeasible cases, and ORR measures refusal on feasible cases. Exec. reports interface executability on emitted feasible configurations. End-to-end latency and local processing time are reported separately. Metric definitions and denominator identities are provided in Appendix C; the supplied aggregates alone do not establish confidence intervals or statistical significance.

5.2 OVERALL SYNTHESIS EFFECTIVENESS (RQ1)

Table 1 separates structural accuracy from safe end-to-end synthesis. PathRAG obtains 82.64 JEM but 35.48 SSS, indicating that retrieving coherent relational evidence is not sufficient for valid numerical and operational decisions. GraphGuard's SSS exceeds ReAct+Validator by 7.85 percentage points, alongside a 7.95-point JEM increase. Their conditional NPE values are approximately 0.0016 and 0.0085, respectively. These scores evaluate emitted configurations, while ORR sepa-

	Configuration quality					Safety and execution					
Method	Slot	Cfg	JEM	Range	NPE↓	Conf.	SSS	UAR↓	ORR↓	Exec.	P95↓
Direct LLM	69.16	66.88	14.91	26.97	0.4058	54.21	0.59	72.09	7.87	61.80	4.213
Full Manuals	71.44	70.25	18.63	35.61	0.2918	59.63	1.78	72.87	8.74	67.51	4.620
TextRAG	72.34	71.86	19.86	47.60	0.1716	65.82	3.37	72.87	9.31	72.96	4.166
GraphRAG	93.29	91.79	79.16	61.61	0.0834	72.91	23.30	73.26	9.78	80.26	4.833
PathRAG	94.24	93.09	82.64	74.86	0.0404	74.25	35.48	73.26	9.92	81.88	4.595
HyperGraphRAG	93.16	91.28	70.80	87.92	0.0147	75.64	44.19	73.26	10.03	82.64	4.163
ReAct+Validator	94.54	94.35	85.50	92.26	0.0085	87.31	67.56	6.80	13.12	96.89	7.107
Symbolic-Only	66.37	80.37	78.26	94.59	0.0015	84.56	63.56	3.64	20.40	93.50	0.475
GraphGuard	**97.11**	**97.03**	**93.45**	**94.59**	**0.0016**	**96.09**	**75.41**	**1.50**	**4.80**	**99.58**	**4.167**

Table 1: Overall synthesis results on the pooled IID test set. Slot, Cfg, and Conf. denote Slot-F1, Config-F1, and Conflict-F1. Full Manuals denotes LLM+Full Manuals. All values are percentages except NPE and P95. The P95 column reports end-to-end latency in seconds, including external-model inference. Range, NPE, and Exec. use emitted feasible configurations; JEM and ORR use all feasible cases. Bold identifies the complete system, not every column optimum.

Variant	Slot	JEM	NPE↓	Conf.	SSS	UAR↓	ORR↓	Exec.	Local P95↓
GraphGuard	**97.42**	**93.37**	**0.0053**	**80.58**	**73.86**	**1.50**	**4.80**	**97.20**	**2.268**
w/o Typed Graph	77.97	45.81	0.0454	33.49	37.06	8.90	12.80	85.00	2.642
Independent Slots	55.77	24.51	0.4752	33.49	19.72	13.70	35.40	65.20	0.854
w/o Query Activation	96.99	92.87	0.0024	80.42	73.44	5.80	2.10	95.40	19.251
w/o Pairwise Scoring	79.88	47.06	0.0065	76.44	37.89	3.50	4.30	94.80	3.081
w/o Hard Constraints	97.43	83.50	0.0059	80.58	68.54	12.30	15.20	83.50	4.798
Greedy Search	97.07	89.50	0.0184	80.58	73.67	5.20	9.10	93.10	3.233
w/o CP-SAT	91.19	81.78	0.1447	76.87	41.81	9.80	12.28	92.40	4.743
w/o Verification	91.13	88.75	0.0015	0.00	70.33	45.20	2.10	85.98	3.631
w/o Bounded Repair	91.14	80.50	0.0645	80.58	69.83	3.20	17.50	95.00	3.861
w/o IR Check	97.42	93.37	0.0053	80.58	64.47	6.80	6.50	82.40	3.407
Symbolic-Only	64.36	75.87	0.0646	69.21	60.25	3.60	20.40	93.50	0.478

Table 2: Component ablation on 3,600 pooled outcomes (1,200 cases and three seeds). Slot and Conf. denote Slot-F1 and Conflict-F1. Percentages are reported except for NPE and local P95. Local P95 is reported in milliseconds and excludes external-model inference; it covers local search, solving, verification, repair, and compilation. NPE and Exec. use emitted feasible configurations.

rately measures refusal on feasible requests. The reported P95 is 4.167 s versus 7.107 s for ReAct+Validator, but GraphGuard is not uniformly faster than retrieval baselines or Symbolic-Only. These are aggregate comparisons, not a claim of zero unsafe acceptance or latency dominance.

5.3 Component and Mechanism Analysis (RQ2)

Table 2 distinguishes three module roles. First, removing typed knowledge or pairwise scoring lowers SSS to 37.06 and 37.89, while Independent Slots reduces JEM to 24.51. Removing hard graph constraints and replacing beam search with greedy selection reduce JEM by 9.87 and 3.87 points, respectively. Second, removing CP-SAT lowers SSS to 41.81, consistent with a numerical-feasibility bottleneck that structural selection alone does not resolve. Third, removing bounded repair raises ORR by 12.70 points and lowers JEM by 12.87 points, consistent with losing recoverable candidates at the release gate. Removing IR checking preserves the reported structural scores but lowers SSS by 9.39 points. Query activation has a small SSS effect but reduces local P95 from 19.251 to 2.268 ms. The matched-verifier controls and their limitations are described in Appendix E.1; their UAR differences cannot establish the effect of pruning alone.

5.4 Compositional and Linguistic Generalization (RQ3)

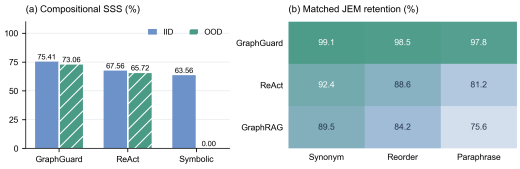


Figure 3: Generalization: (a) IID and compositional-OOD SSS; (b) matched-pair JEM retention under meaning-preserving transformations. Retention is conditional on correct original predictions, not absolute transformed-set JEM.

Figure 3a shows that GraphGuard retains 73.06 SSS on unseen compositions, 7.34 points above ReAct+Validator. Its IID-to-OOD decrease is 2.35 points, compared with 1.84 for ReAct+Validator: the advantage is higher absolute OOD performance, not the smallest degradation. Symbolic-Only falls to zero under its exact-motif lookup implementation; this does not characterize all symbolic synthesis. In Figure 3b, GraphGuard's matched-pair JEM retention ranges from 97.8% to 99.1%, whereas ReAct+Validator falls to 81.2% under paraphrasing. These paired scores test stability of initially correct interpretations and cannot replace unconditional accuracy. The pairing and denominator are defined in Appendix E.2.

5.5 Constraint and Runtime-State Robustness (RQ4)

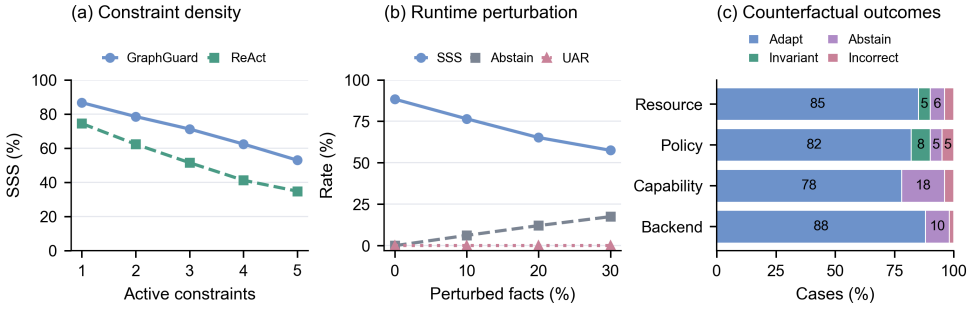


Figure 4: Robustness: (a) SSS under increasing constraint density; (b) SSS, abstention, and UAR under runtime perturbation; (c) mutually exclusive counterfactual outcomes. Correct adaptation, justified invariance, and correct abstention are separated.

As shown in Figure 4a, increasing constraint density reduces SSS for both methods; GraphGuard retains 52.96% at five constraints versus 34.81% for ReAct+Validator. Under 30% runtime perturbation, SSS decreases to 57.50%, abstention rises to 17.50%, and reported UAR remains zero (Figure 4b). This separates conservative refusal from successful synthesis without attributing the entire SSS decline to infeasibility. Counterfactual outcomes provide a finer diagnostic (Figure 4c): capability changes induce the largest correct-abstention share, 18%, while resource and policy changes permit some unchanged configurations. Counting adaptation, justified invariance, and correct abstention yields 95–98% correct responses across perturbation types. These categories evaluate reactions to changed facts, not traffic-cleaning effectiveness.

5.6 Candidate, Search, and Repair Budgets (RQ5)

Figure 5a shows diminishing SSS gains around the reference budget. At $K_s = 1$, SSS remains 45.2% for all beam widths because one candidate per slot leaves one structural path. At $K_s = 3$, increasing beam width from four to eight raises SSS from 71.5% to 75.4%, while doubling it again adds only 0.4 points; increasing K_s to eight at $B = 8$ adds 1.1 points. Repair is evaluated on the initially invalid 10.33% of proposals (Figure 5b): at three attempts, LLM-guided selection recovers 7.7% of this subset, versus 7.1% for severity-greedy and 4.8% for random selection. The implied overall recovery is approximately 0.80 percentage points, and the local profile excludes external inference.

5.7 Backend Portability (RQ6)

Figure 6 separates target-state adaptation from compiler consistency. Transferred SSS is 82.15% for A→B and 73.85% for B→A, below the respective target-native scores by 5.07 and 1.84 points. The

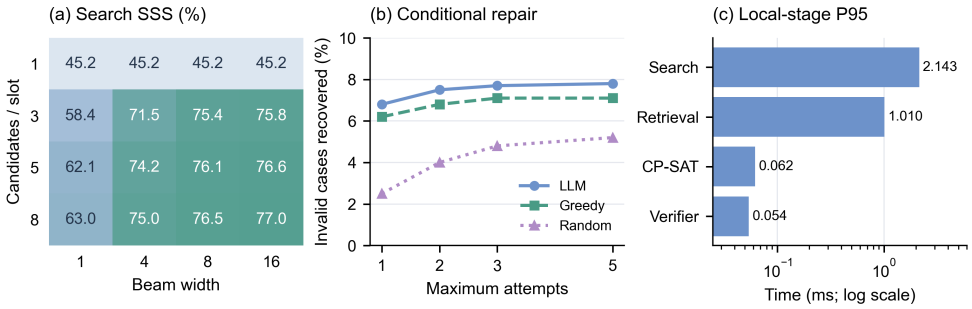


Figure 5: Budget analysis: (a) SSS for candidate and beam budgets, including the single-candidate boundary; (b) conditional recovery of initially invalid proposals; (c) local-stage P95 in milliseconds. Stage quantiles are not additive.

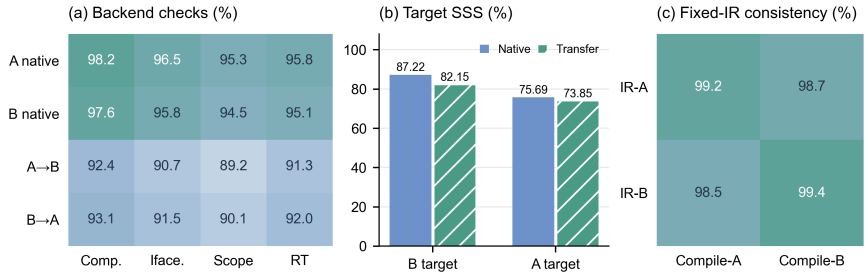


Figure 6: Backend portability: (a) compilation, interface acceptance, scope, and round-trip checks; (b) native versus transferred target SSS; (c) round-trip consistency for fixed IR compiled by each backend. All scores are percentages.

fixed-IR test preserves prototypes, values, scope, and lifecycle and obtains 98.5–99.4% round-trip consistency. Unlike RQ3's composition and wording shifts, RQ6 tests target-specific realization; the semantic projection is defined in Appendix F.

5.8 Backbone Dependence (RQ7)

Figure 7 reports SSS across three backbones. GraphGuard obtains 63.50%, 71.20%, and 75.41% with Qwen2.5-7B, Qwen2.5-32B, and Qwen3-Max, respectively, while direct generation remains below 1%.

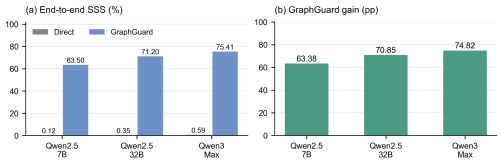


Figure 7: Backbone comparison: SSS for direct generation and GraphGuard, with the corresponding absolute gains.

6 Conclusion

In this paper, we introduce GraphGuard, a graph-constrained neuro-symbolic framework for automated defense-configuration synthesis. By coupling query-conditioned structural inference over a typed configuration graph with deterministic numerical solving and bounded repair, GraphGuard checks structural and numerical consistency before backend-specific release. The evaluation separates end-to-end synthesis quality from component effects, distribution shifts, and backend translation, with safety claims restricted to the constraints and interfaces being checked.

AI USE STATEMENT

Generative AI tools were used only to aid or polish the writing. Details are described in the paper. The authors are responsible for the final text and scientific claims.

REFERENCES

Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE INFOCOM*, volume 4, pp. 2605–2616, 2004.

Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 113–126, 2014.

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*, pp. 328–341, 2016. doi: 10.1145/2934872.2934909.

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the 2017 ACM SIGCOMM Conference*, pp. 155–168, 2017. doi: 10.1145/3098822.3098834.

Luca Beurer-Kellner, Martin Vechev, Laurent Vanbever, and Petar Veličković. Learning to configure computer networks with neural algorithmic reasoning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/04cc90ec6868b97b7423dc38ced1e35c-Abstract-Conference.html.

Boyu Chen, Zirui Guo, Zidan Yang, Yuluo Chen, Junze Chen, Zhenghao Liu, Chuan Shi, and Cheng Yang. PathRAG: Pruning graph-based retrieval augmented generation with relational paths, 2025. URL https://arxiv.org/abs/2502.14902.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph RAG approach to query-focused summarization, 2024. URL https://arxiv.org/abs/2404.16130.

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)*, pp. 579–594. USENIX Association, 2018. URL https://www.usenix.org/conference/nsdi18/presentation/el-hassany.

Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pp. 279–291, 2011.

Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. LightRAG: Simple and fast retrieval-augmented generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 10746–10761. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.568. URL https://aclanthology.org/2025.findings-emnlp.568/.

Bernal Jiménez Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. HippoRAG: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1902.

Hongyu Hè, Minhao Jin, and Maria Apostolaki. Making logic a First-Class citizen in generative ML for networking. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 801–824. USENIX Association, 2026. URL https://www.usenix.org/conference/nsdi26/presentation/he.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-Retriever: Retrieval-augmented generation for textual graph understanding and question answering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/efaf1c9726648c8ba363a5c927440529-Abstract-Conference.html.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474, 2020.

Jianmin Liu, Li Chen, Dan Li, and Yukai Miao. CEGS: Configuration example generalizing synthesizer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 1327–1347. USENIX Association, 2025. URL https://www.usenix.org/conference/nsdi25/presentation/liu-jianmin.

Haoran Luo, Haihong E, Yuhao Yang, Tianyu Yao, Yikai Guo, Zichen Tang, Wentai Zhang, Shiyao Peng, Kaiyang Wan, Meina Song, Wei Lin, Yifan Zhu, and Anh Tuan Luu. Text2NKG: Fine-grained n-ary relation extraction for n-ary relational knowledge graph construction. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-0861.

Haoran Luo, Haihong E, Guanting Chen, Qika Lin, Yikai Guo, Fangzhi Xu, Zemin Kuang, Meina Song, Xiaobao Wu, Yifan Zhu, and Anh Tuan Luu. Graph-R1: Towards agentic GraphRAG framework via end-to-end reinforcement learning, 2025a. URL https://arxiv.org/abs/2507.21892.

Haoran Luo, Haihong E, Guanting Chen, Yandan Zheng, Xiaobao Wu, Yikai Guo, Qika Lin, Yu Feng, Zemin Kuang, Meina Song, Yifan Zhu, and Anh Tuan Luu. HyperGraphRAG: Retrieval-augmented generation via hypergraph-structured knowledge representation. In *Advances in Neural Information Processing Systems*, volume 38, 2025b. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/df55ee6e59f8ac4a625219e11fe9ddba-Abstract-Conference.html.

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*, pp. 189–195, 2023. doi: 10.1145/3626111.3628194.

Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 1–13. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto.

Jialin Niu, Mingzhe Xing, Lei Zhang, Da An, and Yong Cui. DMKG: DDoS defense enhanced by multimodal knowledge graph data. In *2026 IEEE/ACM International Symposium on Quality of Service (IWQoS)*, pp. 1–6. IEEE, 2026. doi: 10.1109/IWQoS70441.2026.11661125.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 953–967. USENIX Association, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/prabhu.

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)*, pp. 629–644. USENIX Association, 2023. URL https://www.usenix.org/conference/nsdi23/presentation/ramanathan.

Weichun Shi, Minghao Liu, Wanting Zhang, Langchen Shi, Fuqi Jia, Feifei Ma, and Jian Zhang. ConstraintLLM: A neuro-symbolic framework for industrial-level constraint programming. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 15999–16019. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.809.

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), 2024. doi: 10.1145/3656296.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

A CONFIGURATION KNOWLEDGE AND BENCHMARK

This appendix specifies the representation, information boundary, and split policy used to evaluate configuration synthesis. Knowledge extraction and downstream synthesis are separate tasks. The evaluation studies synthesis with a frozen configuration graph and does not report an extraction-accuracy experiment.

A.1 FORMAL TASK DEFINITIONS

Section 3 defines the runtime context, typed prototypes, and complete configuration. The induced candidate space is

$$\mathcal{C}(\mathcal{G}_{\text{cfg}}) = \left\{ (\bar{c}, \boldsymbol{\theta}) \;\middle|\; \bar{c} \in \prod_{t \in \mathcal{T}} \mathcal{V}_t,\ \boldsymbol{\theta} \in \mathcal{D}(\bar{c}) \right\}. \tag{5}$$

Problem Definition. Given a synthesis instance q and the shared typed configuration knowledge graph $\mathcal{G}_{\text{cfg}}$, defense configuration synthesis first selects a highest-scoring valid configuration $c^\star$ and then chooses an available backend $b^\star$ capable of executing it. Let $\mathcal{B}(\mathbf{h})$ denote the execution backends available under the current backend state $\mathbf{h}$, and let $\mathcal{Y}_b$ denote the space of executable configurations accepted by backend $b \in \mathcal{B}(\mathbf{h})$. For a candidate configuration c, the Boolean predicate $\text{Valid}(c \mid q, \mathcal{G}_{\text{cfg}})$ indicates whether c satisfies the backend-independent structural, parameter, resource, policy, topology, and lifecycle constraints specified by $\mathcal{G}_{\text{cfg}}$ and q. The predicate $\text{Executable}(c, b \mid q)$ indicates whether backend b supports and can realize c under the current backend state $\mathbf{h}$. The available backends capable of realizing c are collected in the set:

$$\mathcal{B}_{\text{exec}}(c, q) = \{ b \in \mathcal{B}(\mathbf{h}) \mid \text{Executable}(c, b \mid q) = 1 \}. \tag{6}$$

Equation 1 defines the task objective. Once a valid configuration $c^\star$ is selected, choose $b^\star \in \mathcal{B}_{\text{exec}}(c^\star, q)$ and a realizing payload $y^\star \in \mathcal{Y}_{b^\star}$. The task mapping returns $y^\star$ when a valid executable configuration exists and ABSTAIN otherwise. Bounded candidate search approximates this mapping and may fail to find an existing solution.

A.2 PROTOTYPE AND RELATION RECORDS

Each configuration prototype has a canonical identifier, one of the five slot types, a description, a parameter schema, applicability conditions, provenance, and backend mappings. A parameter record declares its type, unit, domain, binding mode, and required runtime inputs. An observed parameter is copied from a named observation after type checking. A selected parameter takes a declared categorical value. An optimized parameter is assigned by the solver. A derived parameter is computed by a registered deterministic function whose inputs and output unit are fixed.

Relations have distinct operational roles. Compatibility permits a combination, dependency requires another prototype or capability, and conflict prohibits a combination under its activation condition. Realization links a prototype to a backend mapping. Alternative-choice relations enumerate replacements available to repair. Absence of a conflict edge alone does not establish compatibility: required capabilities and dependencies must also be checked. A conditional relation is evaluated against the same runtime snapshot used for numerical solving.

A.3 EXTRACTION RECORDS AND BINDING MODES

Each extracted prototype $p \in \mathcal{V}$ is represented as $p = \langle t_p, d_p, \mathbf{\Sigma}_p, \Gamma_p, \omega_p, \boldsymbol{\pi}_p \rangle$, where $t_p \in \mathcal{T}$ is its slot type (e.g., *Action*), d_p is a normalized description (e.g., rate limit), Γ_p specifies applicability conditions (e.g., requiring deep packet inspection capabilities), ω_p records backend mappings (e.g., mapping to Nginx `limit_req` directive), and $\boldsymbol{\pi}_p$ stores provenance (e.g., the URL of the source manual). Its parameter schema is defined as $\mathbf{\Sigma}_p = \{(a, \mathcal{X}_{p,a}, u_{p,a}, g_{p,a}) \mid a \in \text{Par}(p)\}$, where $\mathcal{X}_{p,a}$ and $u_{p,a}$ are the parameter domain and unit for a given parameter a. The binding mode $g_{p,a} \in \{\text{OBSERVED}, \text{SELECTED}, \text{OPTIMIZED}, \text{DERIVED}\}$ indicates how the parameter value is acquired. Specifically, values can be obtained from runtime observations (e.g., extracting an anomalous source IP), selected from a declared set (e.g., `drop` or `reject`), solved numerically (e.g., calculating

Table 3: Five-slot configuration schema and the conditions checked before release.

Slot	Semantic content	Required checks
Match	Traffic predicate and observed identifiers	Predicate types, observation bindings, supported inspection capabilities
Action	Requested operation and exposed parameters	Supported action, compatibility, parameter domain and units
Scope	Enforcement target, path, or placement	Target identity, topology, capabilities, protected-service exclusions
Cost	Resource-demand model	Current utilization, capacity, additional demand
Lifecycle	Duration, activation, and removal conditions	Time units, expiry semantics, supported lifecycle operations

bandwidth limits), or computed by a deterministic function (e.g., deriving a subnet mask from an IP address).

The extractor M_{cfg} also produces typed relations $e = (p_i, r, p_j)$, where $r \in \mathcal{R}_{\mathrm{edge}}$. These relations encode dependencies, compatibility, conflicts, etc. For example, an *Action* prototype for SSL decryption is *compatible* only with *Scope* prototypes that support certificate management. Before graph admission, canonicalization normalizes the extracted identifiers and units. Deterministic schema checks are then applied to filter out malformed entries (e.g., invalid types). Accepted prototypes are merged by their canonical identifiers, and accepted relations are inserted between their canonical endpoints, ultimately forming the global configuration knowledge graph $\mathcal{G}_{\mathrm{cfg}}$.

A.4 Public Inputs and Evaluation Records

GraphGuard-Bench represents configuration knowledge from device manuals, API schemas, and expert rules. The benchmark contains 3,030 configuration synthesis instances $q = \langle s, \mathbf{x} \rangle$, where each defense intent s is paired with runtime observations within the runtime context, resource states, applicable policies, and backend states. Each instance is generated together with acceptable five-slot symbolic configurations, valid parameter intervals, forbidden prototype combinations, permissible enforcement scopes, and executable-backend sets. These programmatic oracle records are kept strictly outside the method inputs and are used exclusively by the evaluator.

The public synthesis input contains the intent, observations, resource snapshot, applicable policies, and backend capabilities. Methods may access the released manuals and the knowledge representation permitted by their evaluation condition. The evaluator alone holds acceptable complete motifs, parameter intervals, and the independent semantic and constraint judgments.

Hidden oracle records must not supply the solver with a target answer or an undeclared per-case bound. Constraints that are necessary to solve the public task must be inferable from the public input and permitted knowledge. The evaluator may implement those constraints independently and test held-out combinations. It must not penalize a method for an arbitrary secret requirement that cannot be inferred from any public evidence.

Each benchmark record carries a case identifier, a source-family identifier, a canonical motif identifier, a split assignment, and a runtime-snapshot identifier. Paraphrases and perturbed versions retain their parent identifier. Grouping by this identifier prevents closely related instances from appearing on both sides of a split or being counted as independent observations in significance tests.

A.5 Split Boundaries

The dataset splits evaluate different generalization properties: the IID split varies observations and numerical parameters within known motifs; Compositional OOD holds out complete motifs while exposing their constituent prototypes during training; Linguistic OOD alters terminology and syntax while preserving the intent; Constraint-density shifts increase simultaneous predicates; and Runtime-state shifts hold the defense intent fixed while perturbing resource states, policies, or execution device availability.

IID evaluation varies observations and numerical values within known composition families. Compositional OOD excludes complete five-slot motifs from training and demonstration selection while retaining their constituent prototypes and relation types. An acceptable alternative motif must also be checked against this exclusion rule. Linguistic OOD changes wording, clause order, terminology, or abstraction granularity without changing the acceptable configuration set. A language transformation that changes a whitelist, target, or action is a semantic counterfactual, treated separately in Appendix E.2.

Runtime perturbations change one or more public facts while preserving the intent. The evaluator recomputes feasibility and acceptable outputs under the perturbed snapshot. Such cases are not automatically examples of successful abstention: abstention is correct only if the independent task oracle establishes infeasibility.

Example material, pending replacement with the final implementation and experimental records.

The following manifest specifies the reporting fields for the final benchmark release. Unfilled entries denote unavailable provenance rather than zero-valued measurements.

Table 4: Benchmark manifest fields. The case counts retain the existing benchmark partition; provenance fields require the final release records.

Partition	Cases	Grouping unit	Required release fields
IID	900	Source family	Manifest hash, feasible/infeasible counts
Compositional OOD	600	Complete motif	Training/test motif lists, alternative-motif audit
Linguistic OOD	600	Parent intent	Transformation labels, semantic-equivalence checks
Robustness	930	Parent snapshot	Perturbation rules, recomputed feasibility
Total	3,030		Frozen graph version and evaluator version

The knowledge-source manifest additionally records document title, version, acquisition date, permitted use, and the source span supporting each admitted record. Annotation documentation identifies the labeling instructions, annotator roles, adjudication procedure, and separation between extractor training examples and evaluation instances. These fields document provenance without adding a graph-construction accuracy claim.

B Algorithm and Conditional Properties

B.1 Search, Solving, and Release

Algorithm 1 makes the separation between candidate exploration and release explicit. The beam score includes the unary scores of selected prototypes and pairwise scores whose endpoints have both been selected. Unresolved dependencies are carried forward rather than declared satisfied. A partial candidate is removed only when a violated constraint is already decidable or no completion can satisfy a required dependency.

The solver receives the parameters and constraints of one symbolic configuration. Its `INFEASIBLE` status concerns that configuration's encoded model, not the full candidate space. An `UNKNOWN` status, an empty retrieved slot, and exhaustion of the beam have separate diagnostic codes. All may lead to abstention under the request budget, but only a complete independent feasibility check supports a task-level infeasibility label.

Legal Repair Update. Equation 4 gives the edit-selection and re-solving update. The verifier evaluates every proposed successor against the same runtime snapshot. Invalid identifiers, unchanged candidates, and non-decreasing violation severity do not extend the repair trajectory. An edit that repairs one rule but violates another is accepted only if it passes the declared strict severity-decrease test, and release still requires all checks to pass.

Algorithm 1 GraphGuard inference with bounded search and checked release

Require: Instance q, graph $\mathcal{G}_{\mathrm{cfg}}$, candidate budget K_s, beam width B, repair limit $T_{\max}$
Ensure: Backend payload or ABSTAIN, with a terminal reason

```
Retrieve typed candidate sets K_{q,t} and construct G_q
Initialize the beam with the empty partial configuration
for slot t in M, A, S, C, L do
    Expand each beam entry with candidates in K_{q,t}
    Reject decidable hard violations and retain the top B scored prefixes
end for
Solve each complete prefix and retain feasible assignments with solver status
Rank instantiated configurations by J(c | q)
for instantiated candidate c do
    Evaluate the full violation set Δ(c; q)
    for k = 1, ..., T_max, while Δ(c; q) ≠ ∅ do
        Enumerate finite legal edits and request one edit identifier from M_0
        if the identifier denotes a legal edit then
            Apply the edit, re-solve affected parameters, and recheck all constraints
            Keep the update only if weighted violation severity strictly decreases
        end if
    end for
    if Δ(c; q) = ∅ then
        for available backend b, in declared policy order do
            Compile c, check schemas, interface acceptance, scope, and round trip
            if all release checks pass under the same snapshot then
                return payload and verification record
            end if
        end for
    end if
end for
return ABSTAIN and the recorded terminal reason
```

B.2 BACKEND RELEASE PREDICATE

Even if a synthesized configuration appears structurally sound, it may contain invalid parameters that fail to compile on target devices or inadvertently block critical business traffic (e.g., overlapping with VIP whitelists) when deployed. To address this, instantiated configurations are unified and verified in a device-independent manner before backend mapping.

Backend Compilation and Release. A verified configuration c is serialized into an intermediate representation $\mathrm{IR}(c) = \langle c, \mathbf{\Pi}, \boldsymbol{\zeta}_{\mathrm{ver}} \rangle$, where $\mathbf{\Pi}$ stores prototype provenance and $\boldsymbol{\zeta}_{\mathrm{ver}}$ records the graph, policy, resource, and verifier versions. For each available backend b, the compiler maps the unified IR to the specific device context, producing the executable payload $y_b(c) = \mathrm{Compile}_b(\mathrm{IR}(c))$. The payload passes backend validation when the predicate $\mathrm{BackendOK}$ is satisfied:

$$\begin{aligned}\mathrm{BackendOK}(c, b; q) = \mathbb{I}\Big[&\mathrm{Schema}_{\mathrm{IR}}(\mathrm{IR}(c)) \\ &\wedge \mathrm{Schema}_b(y_b(c)) \\ &\wedge \mathrm{InterfaceAccept}_b(y_b(c)) \\ &\wedge \mathrm{Consistent}_b\big(y_b(c), \mathrm{IR}(c)\big)\Big],\end{aligned} \tag{7}$$

where the predicates check IR validity, backend-schema conformance, target-interface acceptance, and the consistency of compiled fields, values, units, scope, and lifecycle settings. Accordingly, we instantiate the theoretical predicates as $\mathrm{Valid}(c \mid q, \mathcal{G}_{\mathrm{cfg}}) = \mathbb{I}[\Delta(c; q) = \varnothing]$ and $\mathrm{Executable}(c, b \mid q) = \mathrm{BackendOK}(c, b; q)$. GraphGuard selects the first verified configuration $c^\star$ with at least one executable backend in the pre-repair candidate order, chooses a policy-ranked backend $b^\star = \mathrm{SelectBackend}(\mathcal{B}_{\mathrm{exec}}(c^\star, q); \mathcal{P}, \mathbf{h})$, and returns $y^\star = y_{b^\star}(c^\star)$. It returns ABSTAIN when no candidate passes deterministic verification and backend validation. The released configuration is selected from the explored candidates, not certified as the global optimum of the task formulation. Verification establishes consistency with encoded constraints and the checked backend

contract, not correctness of missing knowledge or unmodeled device behavior. Appendix B.4 states the corresponding conditional properties.

B.3 Integer Encoding and Numerical Semantics

CP-SAT operates on finite integer domains. For a numerical parameter θ_j, let $\delta_j > 0$ be its declared resolution in a canonical physical unit, and let z_j be its integer representation, with $\theta_j = \delta_j z_j$. A permitted interval $[\ell_j, u_j]$ becomes the integer domain $\lceil \ell_j/\delta_j \rceil \le z_j \le \lfloor u_j/\delta_j \rfloor$. If this domain is empty, the interval has no representable value at the selected resolution.

Resource functions, effectiveness requirements, and costs require explicit linear, Boolean, table, or supported finite-domain encodings. Arbitrary real-valued functions in the abstract formulation cannot be passed directly to CP-SAT. Rational coefficients can be cleared by a common denominator when exact integer scaling is possible. Approximate encodings require a documented error bound and conservative treatment of safety limits. The verifier checks the decoded assignment in canonical units after solving.

The objective value and best bound are recorded separately. A feasible incumbent satisfies the encoded constraints but need not minimize cost. An optimal assignment for a retained motif does not establish an optimal configuration over all motifs, because retrieval and beam pruning restrict the explored space.

B.4 Conditional Release and Termination

Property 1: checked release. Assume that the runtime snapshot remains unchanged during synthesis and release, all required encoded predicates are evaluated correctly, and the backend checker faithfully implements its declared contract. Every released payload then satisfies the encoded verifier predicates and the checked backend contract.

Proof. The release branch is reachable only after the full violation set is empty. Compilation is followed by the backend checks, and a payload is returned only when all of them pass. The stated property follows from these two gates. The argument does not establish semantic correctness of the source knowledge, completeness of the predicates, or behavior outside the interface contract.

Property 2: bounded repair. Assume fixed nonnegative violation weights, a finite set of verifier predicates, finite edit enumeration, and bounded execution time for each LLM, solver, and interface call. Repair terminates after at most $T_{\max}$ attempted updates for a candidate. A retained update strictly decreases the weighted severity and therefore cannot revisit an earlier retained violation profile with the same weight.

Proof. The attempt counter increases on every iteration, including malformed responses and rejected edits, and is bounded by $T_{\max}$. For retained updates, strict decrease prevents a cycle back to an earlier severity value. The attempt limit also covers unsuccessful or repeated LLM proposals. These properties imply termination, not completeness or recovery of every feasible candidate.

Search cost. Let S be the number of slots, K_s the maximum candidates per slot, and B the beam width. There are at most SBK_s prefix expansions. With pairwise contributions computed against at most S selected prototypes, scoring costs $O(S^2BK_s)$, excluding encoder evaluation and predicate evaluation. Top-B selection adds $O(SBK_s \log B)$ with bounded heaps. Numerical solving has instance-dependent combinatorial cost and is not covered by this polynomial search bound. Repair adds at most $BT_{\max}$ attempts when at most B instantiated candidates are processed.

C Evaluation Definitions and Statistical Protocol

C.1 Metric Overview

An evaluator independent of GraphGuard's verifier combines expert acceptable sets, hidden constraints, and execution-device interface responses. We evaluate four dimensions: (1) Structural selection: *Slot-F1* measures individual component accuracy, while *Joint Exact Match (JEM)* requires the complete symbolic configuration to match an acceptable motif. (2) Parameter instantiation:

Normalized Parameter Error (NPE) is zero inside a valid interval and otherwise measures the normalized distance to its nearest boundary. (3) Safety verification: *Unsafe Acceptance Rate (UAR)* is the proportion of infeasible instances for which an unsafe configuration is accepted. *Over-Refusal Rate (ORR)* is the proportion of feasible instances for which the method returns ABSTAIN. (4) Execution and Efficiency: *Executability (Exec.)* measures successful compilation and acceptance by the target interface. Efficiency is reported via P95 latency. *Safe Synthesis Success (SSS)* serves as the primary holistic metric, requiring semantic correctness, hidden-constraint satisfaction, and successful backend execution:

$$\mathrm{SSS} = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}\left[\mathrm{Sem}_n \wedge \mathrm{Con}_n \wedge \mathrm{Exec}_n\right]. \tag{8}$$

SSS is averaged over all cases, and JEM uses all feasible cases. Numerical accuracy and executability use emitted feasible configurations. UAR and ORR use infeasible and feasible cases, respectively. The following subsections give the full definitions, failure handling, and consistency checks.

C.2 Evaluation Units and Failure Handling

Let $\mathcal{Q} = \{q_n\}_{n=1}^{N}$ be the evaluation set, and let $\mathcal{Q}_F$ and $\mathcal{Q}_I$ be its feasible and infeasible subsets under the independent oracle. Their index sets are $\mathcal{I}_F = \{n : q_n \in \mathcal{Q}_F\}$ and $\mathcal{I}_I = \{n : q_n \in \mathcal{Q}_I\}$. Their sizes are N_F and N_I, with $N = N_F + N_I$. A method returns either one final configuration or an abstention with a reason. Malformed output, API failure, timeout, solver uncertainty, and deliberate refusal remain distinct terminal states in the raw record.

Let $\mathcal{I}_E \subseteq \mathcal{I}_F$ denote feasible cases where the method emits a configuration, and let $N_E = |\mathcal{I}_E|$. Emission is recorded before backend acceptance, so an emitted configuration that fails compilation or interface validation remains in this subset.

To isolate the evaluation of numerical synthesis capability and backend compatibility from abstention behavior, metrics including Range Accuracy, NPE, and Executability are computed conditionally on the subset of feasible instances where the method emits a configuration. The abstention rate is evaluated exclusively via ORR.

Slot-F1 and Config-F1 evaluate the retained synthesis candidate before the final release decision. A candidate may therefore receive structural credit even if the method subsequently refuses to release it. Missing candidates receive zero structural credit. JEM evaluates the final output on all feasible cases and assigns zero to abstention, missing output, or an incorrect complete motif. Missing required parameters within an emitted configuration receive unit numerical error and zero Range credit. An empty evaluation subset yields an undefined metric, reported as a dash.

C.3 Structural and Numerical Metrics

For feasible instance q_n, let $\mathcal{A}_n$ be its set of acceptable complete motifs and $\widetilde{c}_n$ the retained candidate before the release decision. For slot t, let $\mathcal{L}_t$ be the annotated reference-label set. Slot-F1 macro-averages class-wise F1 within each slot and then over the five slots:

$$\mathrm{SlotF1} = \frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} \frac{1}{|\mathcal{L}_t|}\sum_{\ell\in\mathcal{L}_t} \frac{2TP_{t\ell}}{2TP_{t\ell} + FP_{t\ell} + FN_{t\ell}}. \tag{9}$$

The confusion counts compare candidate labels with annotated reference labels over all feasible cases. Missing candidate labels are false negatives for their reference class, not a new semantic class. This class-macro metric need not have the same ordering as instance-level JEM.

Let $S(c)$ contain the canonical prototypes and required typed relations of configuration c, and let $f(A,B) = 2|A \cap B|/(|A| + |B|)$. Config-F1 averages $\max_{a\in\mathcal{A}_n} f(S(\widetilde{c}_n), S(a))$ over feasible cases, assigning zero to missing candidates. JEM instead assigns one only when an emitted final symbolic tuple belongs to $\mathcal{A}_n$; all refused feasible cases receive zero.

For required parameter j of case n, let $[\ell_{nj}, u_{nj}]$ be its acceptable interval in canonical units, and let $\widehat{\theta}_{nj}$ be the predicted value. Define the interval width $w_{nj} = u_{nj} - \ell_{nj}$ and the positive normalization

floor ϵ_j, which has the same unit as the parameter. The per-parameter error is

$$e_{nj} = \min\left(1, \frac{\max\{\ell_{nj} - \widehat{\theta}_{nj},\, 0,\, \widehat{\theta}_{nj} - u_{nj}\}}{\max\{w_{nj}, \epsilon_j\}}\right). \tag{10}$$

A missing, nonnumeric, nonfinite, or unit-incompatible value has $e_{nj} = 1$. Range Accuracy averages the interval-membership indicator over required parameters within each emitted feasible case and then over $\mathcal{I}_E$. NPE uses the same two-level averaging on e_{nj} and the same subset. The normalization floor is fixed per parameter schema, not fitted to the method's outputs. The legacy evaluator uses 10^{-12} in canonical numerical units. A changed floor or reference-matching policy requires recomputation rather than relabeling old values.

NPE is dimensionless and is written with a leading zero. A value of 0.0015 means low average clipped interval error under this normalization. It is not automatically a percentage or a measure of physical mitigation quality. Parameter selection errors and numerical errors remain separate: a valid number attached to the wrong action can have low NPE but fail JEM and SSS.

C.4 Safety, Abstention, and Execution

Let a_n indicate that the final unsafe output is accepted on an infeasible instance, and let r_n indicate a final abstention on a feasible instance. Then UAR is $N_I^{-1}\sum_{n\in\mathcal{I}_I} a_n$, and ORR is $N_F^{-1}\sum_{n\in\mathcal{I}_F} r_n$. SSS uses all N cases and requires a correct complete motif, valid parameter assignments, satisfaction of the independent constraints, and successful backend checks. Correct refusal of an infeasible case is reported separately and does not count as successful synthesis.

To isolate the evaluation of numerical synthesis capability and backend compatibility from abstention behavior, metrics including Range Accuracy, NPE, and Executability are computed conditionally on the subset of feasible instances where the method emits a configuration. The abstention rate is evaluated exclusively via ORR.

Executability is $N_E^{-1}\sum_{n\in\mathcal{I}_E} x_n$, where x_n indicates successful compilation and interface acceptance of the emitted configuration. Abstained cases are excluded from this denominator and remain counted by ORR. IR–payload consistency checks whether decoding the compiled payload reproduces the canonical fields, units, scope, and lifecycle semantics of the verified IR.

Conflict-F1 must name its binary target. For forbidden-combination detection, the positive label is a violated relation in a prespecified candidate set, and F1 is $2TP/(2TP+FP+FN)$. Instance-level infeasibility detection is a different task: its positives are infeasible requests, including numerical or backend infeasibility without a forbidden pair. These two F1 scores are not interchangeable.

With all rates expressed as fractions, $p_F = N_F/N$, and $p_E = N_E/N_F$, the reporting definitions imply

$$\begin{aligned} \mathrm{SSS} &\le p_F\,\mathrm{JEM}, \\ \mathrm{SSS} &\le p_F(1-\mathrm{ORR}), \\ \mathrm{SSS} &\le p_F p_E\,\mathrm{Exec}, \\ \mathrm{JEM}+\mathrm{ORR} &\le 1, \\ 0 \le \mathrm{NPE} &\le 1-\mathrm{Range}\,. \end{aligned} \tag{11}$$

The first bound assumes the same exact-motif semantic criterion for SSS and JEM. The NPE bound requires clipped errors and identical parameter weights for Range and NPE. Since NPE and executability are conditional, neither $\mathrm{NPE} \ge \mathrm{ORR}$ nor $\mathrm{Exec}+\mathrm{ORR} \le 1$ applies. Applying the remaining identities requires the corresponding numerators, denominators, and feasible-case share before rounding.

Conditional and Refusal-Penalized Scores. An alternative unconditional parameter metric, $\mathrm{NPE}_{\mathrm{all}}$, assigns unit error to every feasible case without an emitted configuration. If non-emission occurs only through abstention, let $r = \mathrm{ORR}$; then

$$\begin{aligned} \mathrm{NPE}_{\mathrm{all}} &= r + (1-r)\,\mathrm{NPE}, \\ \mathrm{Exec}_{\mathrm{all}} &= (1-r)\,\mathrm{Exec}\,. \end{aligned} \tag{12}$$

Here $\mathrm{Exec}_{\mathrm{all}}$ counts successful execution over all feasible cases, with abstention contributing zero. The corrected all-feasible summaries for GraphGuard are $\mathrm{NPE}_{\mathrm{all}} = 0.0495$, $\mathrm{Exec}_{\mathrm{all}} = 0.9480$, and $r = 0.0480$; for ReAct+Validator they are 0.1384, 0.8420, and 0.1310. Equation 12 yields the approximate conditional table values $0.0016/99.58\%$ and $0.0085/96.89\%$, respectively. These entries are conversions of rounded summaries. If other non-emission outcomes exist, their actual share must replace r in this conversion.

C.5 Pairing, Uncertainty, and Timing

The statistical unit is the base instance, not each paraphrase or repeated seed independently. Paired bootstrap resampling selects base-instance groups and retains their methods, seeds, and transformed variants together. Confidence intervals summarize the paired difference rather than comparing overlapping marginal intervals. Binary comparisons use an exact paired test for one observation per independent case or a cluster-aware test when repeats are retained. Continuous differences use paired resampling or a permutation procedure whose exchangeability assumptions are stated. Multiple confirmatory comparisons are grouped in advance for multiplicity correction.

End-to-end latency starts before the first model request and ends at final acceptance or terminal failure. It includes model waiting time, all repair calls, retrieval, solving, verification, and compilation. Local stage times are reported separately. P95 is computed from per-request elapsed times and cannot be obtained by adding stage P95 values. Successful-request latency is accompanied by completion and timeout rates. Token cost includes initial calls, repair, and retries.

D Implementation Interfaces and Prompts

Example material, pending replacement with the final implementation and experimental records.

This section gives a concrete implementation specification and prompt examples. It does not identify these examples as the prompts or model checkpoints that produced the retained tables.

D.1 Retrieval and Structured Scoring

Request-Conditioned Candidate Retrieval. An accessible request encoder maps the instance q into a dense representation vector $\mathbf{z}_q$. Concurrently, a relation-aware Graph Neural Network (GNN) encoder computes a structural feature embedding $\mathbf{h}_p$ for each prototype $p \in \mathcal{V}$. To measure the semantic alignment between the requested instance q and each prototype's structural embedding $\mathbf{h}_p$, GraphGuard computes a matching score u_t and retains the active candidate set $\mathcal{K}_{q,t}$:

$$\begin{aligned} u_t(p \mid q) &= \mathbf{w}_t^\top \tanh(W_t[\mathbf{z}_q; \mathbf{h}_p] + \mathbf{b}_t)\,, \\ \mathcal{K}_{q,t} &= \mathrm{TopK}_{K_s}\Big(\{p \in \mathcal{V}_t \mid \Gamma_p(\mathbf{x}) = 1\}; u_t(\cdot \mid q)\Big). \end{aligned} \tag{13}$$

Here, the applicability predicate $\Gamma_p(\mathbf{x})$ checks required capabilities, verifying, for instance, whether the current backend supports TLS fingerprint extraction. To prevent candidates from one highly relevant slot from exhausting the retrieval budget, the TopK_{K_s} operator strictly retains the K_s highest-scoring prototypes for each specific slot type. For the intent of limiting high-frequency sessions with abnormal TLS fingerprints, the retained prototypes may include an abnormal-TLS *Match*, a session-limit *Action*, a gateway *Scope*, a state-consumption *Cost* model, and a duration-based *Lifecycle*. Their induced subgraph $\mathcal{G}_q = \mathcal{G}_{\mathrm{cfg}}[\bigcup_{t\in\mathcal{T}} \mathcal{K}_{q,t}]$ compactly contains these retained candidates and their typed relations.

Graph-Coupled Structural Selection. Although the induced subgraph $\mathcal{G}_q$ filters out irrelevant knowledge, simply selecting the top-scoring prototype for each slot may yield structurally invalid combinations. GraphGuard therefore combines the unary matching score with typed cross-slot relations to evaluate overall structural coherence. For candidate prototypes $p_t \in \mathcal{K}_{q,t}$ and $p_u \in \mathcal{K}_{q,u}$, their relation potential is

$$\psi_{tu}(p_t, p_u \mid q) = \mathbf{a}_{tu}^\top \tanh\Big(W_{tu}[\mathbf{z}_q; \mathbf{h}_{p_t}; \mathbf{h}_{p_u}; \mathbf{e}_{tu}] + \mathbf{b}_{tu}\Big), \tag{14}$$

where $\mathbf{e}_{tu}$ encodes their typed relations within $\mathcal{G}_q$.

D.2 Learned Components and Training

The domain-adapted extractor M_{cfg} is an offline component. The inference LLM M_0 interprets requests and selects legal repair identifiers. These models have different inputs, supervision, and execution times. A hosted chat-completion endpoint does not by itself expose a dense hidden-state representation. The request-encoding interface therefore requires either an accessible encoder or a separately declared embedding model applied to the serialized intent and runtime context. A prompt-only Top-K candidate generator is an alternative implementation and must not be described as a trained GNN scorer.

Let the request encoder produce $\mathbf{z}_q \in \mathbb{R}^{d_q}$, the graph encoder produce $\mathbf{h}_p \in \mathbb{R}^{d_g}$, and the relation encoder produce $\mathbf{e}_{tu} \in \mathbb{R}^{d_r}$. For hidden scorer width d_s, the unary matrix has shape $d_s \times (d_q + d_g)$, while the pairwise matrix has shape $d_s \times (d_q + 2d_g + d_r)$. Biases and output vectors have dimension d_s. The coefficients $\alpha_t, \beta_{tu}, \lambda_C, \lambda_J$ are nonnegative and selected using training or validation data.

The graph encoder and structured scorers are trained using the joint loss $\mathcal{L}_{\text{syn}} = \mathcal{L}_{\text{slot}} + \lambda_J \mathcal{L}_{\text{joint}}$. Here, $\mathcal{L}_{\text{slot}}$ is the cross-entropy loss over acceptable prototypes for each slot, while $\mathcal{L}_{\text{joint}}$ contrasts acceptable configurations against hard negatives formed by conflicting substitutions. For instance, pairing a TLS-based *Match* with a network-layer routing *Scope* forms a typical hard negative. The base LLM, beam search, and CP-SAT solver are not updated. Appendix D separates the extractor, request encoder, graph encoder, and scoring heads and specifies their training and inference interfaces.

One multi-reference training specification uses the acceptable slot set $\mathcal{A}_{q,t}$, the set $\mathcal{A}_q$ of acceptable complete motifs, and a negative set $\mathcal{N}_q$ containing no acceptable motif:

$$\begin{aligned} \mathcal{L}_{\text{slot}}(q) &= -\frac{1}{|\mathcal{T}|} \sum_t \log \sum_{p \in \mathcal{A}_{q,t}} P_\Theta(p \mid q, t), \\ \mathcal{L}_{\text{joint}}(q) &= \max\left\{0,\, m + \max_{c^- \in \mathcal{N}_q} E_\Theta(c^- \mid q) - \max_{a \in \mathcal{A}_q} E_\Theta(a \mid q)\right\}, \end{aligned} \tag{15}$$

where P_Θ is the softmax of unary scores within a slot and $m > 0$ is a training margin. The combined loss is the slot loss plus λ_J times the joint loss. Negatives are formed by substitutions checked against the training-time acceptable sets, never against hidden test labels. A complete training record must identify encoder checkpoints, graph layers, hidden dimensions, optimizer, learning rate, batch size, epochs, negative sampling, validation selection, and random seeds.

D.3 Extraction and Candidate Prompts

The extraction prompt makes source grounding explicit. A provenance span accompanies each extracted condition, and unsupported relations are left unasserted rather than completed from general model knowledge.

Listing 1: Example offline extraction prompt.

```
SYSTEM
Extract configuration knowledge only from the supplied source.
Use exactly these slot types: Match, Action, Scope, Cost, Lifecycle.
For each prototype, return its identifier, type, description,
parameter domains, units, binding modes, applicability, and evidence.
For each relation, return source, type, target, condition, and evidence.
Do not invent a default value, capability, or backend mapping.
Return one JSON object with "prototypes" and "relations".

USER INPUT FIELDS
source_id; document_version; source_span; source_text; schema
```

The inference prompt exposes only public inputs and the permitted evidence. The candidate response identifies prototypes rather than inventing device-specific commands. Numerical values delegated to the solver remain unassigned at this stage.

Listing 2: Example candidate-grounding prompt.

```
SYSTEM
Map the defense intent to the supplied typed configuration prototypes.
Use only identifiers present in the candidate catalogue.
For each slot, return up to K ordered candidate identifiers.
Bind observed values only when present in the runtime observations.
Do not guess optimized parameters or hidden evaluation labels.
If a slot has no supported candidate, return an empty list for it.
Return JSON with keys "Match", "Action", "Scope", "Cost", "Lifecycle",
and "observed_bindings". Do not return device commands.

USER INPUT FIELDS
intent; observations; resources; policies; backend_capabilities;
candidate_catalogue; K
```

D.4 Repair and Direct-Generation Prompts

Repair selects an enumerated edit. It does not rewrite the full configuration or modify the constraint set. The selected identifier is checked before applying the associated deterministic operation.

Listing 3: Example legal-edit selection prompt.

```
SYSTEM
Choose one edit from the supplied legal edit list to address the
reported violations while preserving the defense intent.
Return exactly {"edit_id": "<one supplied identifier>"}.
If no edit is suitable, return {"edit_id": null}.
Do not add fields, parameter values, prototypes, or new constraints.

USER INPUT FIELDS
intent; current_configuration; violation_records; legal_edits
```

A direct-generation control receives the same public instance and output schema, but emits a complete five-slot configuration and numerical assignments. Its refusal format is fixed as `{"status":"abstain"}`. Retrieval baselines receive evidence selected by their own retrieval mechanisms. The same-information control instead serializes the exact graph records available to GraphGuard, separating representation access from the search algorithm.

D.5 Execution Manifest

Table 5: Required execution-manifest fields. Example settings are not evidence of completed runs.

Component	Recorded configuration
External model	Provider, immutable model version when available, call date, decoding settings, token limit
Request encoding	Encoder checkpoint, pooling rule, input serialization, embedding dimension
Graph encoder	Relation vocabulary, layers, hidden width, checkpoint hash
Search	Slot order, candidate budget, beam width, tie-breaking, pruning rules
Solver	OR-Tools version, integer resolutions, time limit, workers, seed, status and objective bound
Repair	Prompt hash, legal-edit schema, maximum attempts, severity weights
Backend	Schema version, capability snapshot, compiler version, parser version
Runtime	OS, CPU/GPU, memory, concurrency, warm-up and timing boundaries

The existing design uses a per-slot candidate budget of 3, beam width 8, and at most three repair rounds. These values define the reference operating point for sensitivity analysis, not a substitute for a complete execution manifest. Each request record includes the input hash, method identifier,

prompt hash, model response, token counts, elapsed times, selected motif, solver status, violations, repair edits, compiled payload, and final evaluator outcomes. Credentials are supplied through the runtime environment and are excluded from request logs.

E Diagnostic Protocols

The main comparison and component ablations are reported only in Tables 1 and 2. Figures 3–7 contain the complementary diagnostics. This appendix specifies evaluation boundaries rather than repeating result tables.

E.1 Matched-Information and Matched-Constraint Controls

The LLM-based evaluation protocol uses Qwen3-Max as the shared backbone for neural generation and retrieval-augmented methods, including GraphGuard. Direct LLM generates a complete configuration from the instance alone, whereas LLM+Full Manuals additionally receives the complete source corpus. TextRAG retrieves semantic text chunks; GraphRAG retrieves graph-organized evidence; PathRAG retrieves pruned relational paths; and HyperGraphRAG retrieves higher-order evidence. ReAct+Validator iteratively generates configurations and invokes an external validator. Symbolic-Only performs deterministic rule matching without neural semantic scoring. All methods share the identical public inputs, five-slot schema, timeout policy, and evaluator.

For matched comparisons, the ablation protocol holds the graph version, runtime inputs, evaluator, and model budget fixed. A graph-as-text control receives the same prototypes, relations, numerical domains, and provenance as GraphGuard but serializes them as prompt context. A candidate-matched control receives the same per-slot candidate sets and selects slots independently. A solver-matched control receives the same candidate sets and hard constraints but replaces learned pairwise scores with a fixed deterministic ordering. These controls isolate graph access, joint ranking, and numerical solving without denying baselines information available to the full system.

For the hard-constraint ablation, search-stage pruning is removed while the final verifier remains unchanged. The experiment records candidates expanded, downstream violations, solver calls, and final unsafe releases. If final UAR changes, the analysis first checks whether the verifier's predicate set, snapshot, or release path also changed. A downstream verifier that checks the same constraint correctly should still reject its violation.

Table 6: Matched controls and their isolated factors. These are protocol definitions, not measured results.

Control	Held constant	Changed factor
Graph as text	Knowledge records and model budget	Explicit constrained search versus prompt context
Independent slots	Candidate sets and numerical solver	Joint cross-slot selection
Fixed pairwise scores	Graph, candidates, solver, verifier	Learned structural ranking
No search pruning	Full final verifier and release gate	Early hard-constraint pruning
Repair policy comparison	Legal edits, feedback, attempt limit	Random, severity-greedy, or LLM edit selection

E.2 Paired Semantic Evaluation

Meaning-preserving transformations use a matched paired set with the reference configuration, runtime snapshot, and acceptable configuration set held fixed. Let z_i and z_i' indicate exact-motif correctness before and after transformation. Matched-pair JEM retention uses initially correct cases:

$$\text{Retention} = \frac{\sum_i z_i z_i'}{\sum_i z_i}. \tag{16}$$

A zero denominator is undefined. This measure does not count two incorrect but identical predictions as successful retention. It must be distinguished from unconditional transformed-set JEM and

literal output agreement. Synonym replacement, clause reordering, and paraphrasing are evaluated separately.

Counterfactual evaluation changes a resource, policy, capability, or backend fact and recomputes the acceptable set. The four outcome categories in Figure 4c are mutually exclusive: correct adaptation, justified invariance, correct abstention, and incorrect response. An output need not change if it remains acceptable under the new fact. Correct abstention requires independent infeasibility, not merely search exhaustion. Resource reduction therefore does not automatically imply that an unchanged configuration is incorrect.

E.3 Constraint and Runtime-State Controls

Constraint-density conditions vary the number of simultaneously active resource, policy, topology, and lifecycle predicates. Runtime perturbations vary the facts used by those predicates while preserving the intent. Feasibility is recomputed for each changed snapshot; the feasible-case share and abstention rate are distinct quantities. Zero observed UAR does not establish that every refusal was necessary.

These experiments leave knowledge correctness outside the tested perturbation axis. They do not establish robustness to a missing dependency, an incorrect compatibility edge, or a wrongly encoded numerical bound. A knowledge-corruption extension would require a separate clean evaluator rather than corrupting the method and its oracle together.

E.4 Budget, Repair, and Timing Definitions

Candidate budget K_s controls the number of retained prototypes per slot, while beam width B limits surviving prefixes. The displayed grid uses $K_s \in \{1, 3, 5, 8\}$ and $B \in \{1, 4, 8, 16\}$. Under the fixed-candidate algorithm, $K_s = 1$ admits at most one prefix at every expansion depth, so increasing B cannot change the result. The corrected single-candidate condition has 45.2% SSS at every beam width, consistent with this boundary. Candidate sets and random draws are held fixed across beam widths.

Repair compares policies on the same initially invalid pool. LLM-guided repair selects from legal edit identifiers; severity-greedy repair evaluates severity reductions; random repair samples a legal edit. With initially valid share v_0 and conditional recovery r_T, final validity is $v_0 + (1 - v_0)r_T$, provided recovered cases use the same validity criterion. Neither initial validity nor conditional recovery alone measures semantic SSS. All additional model calls and solver evaluations count toward the corresponding policy's cost.

Local stage timings exclude external-model inference. Per-stage P95 values are marginal quantiles, not an additive decomposition of total P95. The backend timing entries supplied as 4.138, 3.951, 4.053, and 3.858 have conflicting second/millisecond labels across versions and are not used for a latency conclusion until their timer boundary and unit are resolved.

F Backend Contracts, Worked Example, and Limitations

F.1 Target-State Adaptation

In the A→B setting, the intent and observations of a Device A instance are retained, while resource state, backend state, and executable mappings are replaced with the Device B snapshot. Acceptable configurations are recomputed under the target capabilities. The inverse transformation defines B→A. Figure 6 reports both directions without duplicating a result table here.

F.2 Fixed-IR Portability

The fixed-IR protocol for RQ6 separates translation from re-synthesis. In this condition, the intent, selected prototypes, numerical values, scope, and lifecycle are frozen. The same canonical IR is compiled independently for backends A and B when both declare support for that configuration. Unsupported configurations are counted in the coverage report and are not silently discarded to improve the pass rate. Figure 6c reports round-trip consistency for both origin–compiler combinations.

Let D_b denote a backend-specific decoder from accepted payloads to canonical semantics, and let $S(c)$ denote the canonical semantic projection of the verified configuration. A round trip passes when $D_b(\text{Compile}_b(\text{IR}(c))) = S(c)$. The projection includes every safety-relevant field, including units, enforcement scope, action behavior, and expiry semantics. Equality of field names or a successful HTTP response is insufficient.

The capability-adjusted condition instead replaces backend resources and mappings, then synthesizes again. This condition measures adaptation to a different target state. A change in SSS can result from a different feasible set as well as compiler behavior. It cannot alone establish lossless translation. Neither condition tests zero-shot support for a vendor whose schema and mapping rules were never supplied.

F.3 Worked Configuration and Payload Example

Example material, pending replacement with the final implementation and experimental records.

Consider an intent to limit UDP traffic to the protected service at `192.0.2.10`, while preserving a management whitelist. The example IR selects a UDP predicate, a rate-limiting action, the protected-service ingress scope, a declared resource model, and a finite expiry. The illustrative rate is 10,000 packets per second and the duration is 60 seconds. These values demonstrate unit conversion only and are not measured device settings or recommended operating thresholds.

Listing 4: Example canonical IR and two illustrative backend encodings. These field names are not vendor API specifications.

```
IR:
  match: {protocol: udp, destination: "192.0.2.10/32"}
  action: {type: rate_limit, value: 10000, unit: packet/s}
  scope: {service: protected_service, direction: ingress}
  cost: {model: declared_rate_limit_profile}
  lifecycle: {duration: 60, unit: second}
  exclusions: {management_whitelist: preserved}

Backend A contract:
  rate_pps: 10000
  ttl_seconds: 60
  target: protected_service
  direction: ingress

Backend B contract:
  rate_kpps: 10
  ttl_milliseconds: 60000
  object: protected_service
  direction: ingress
```

Both payloads must additionally encode the match predicate, action behavior, cost-related capability requirements, and whitelist exclusion. The shortened listing shows only fields relevant to numerical conversion and scope mapping. A decoder that recovers 10 packets per second from Backend B's `rate_kpps` field fails the round-trip check even if its payload passes schema validation. A missing whitelist or reversed direction also fails semantic consistency.

F.4 Failure Analysis and Scope

Failures are attributed to the first incorrect stage using the preserved intermediate records: unsupported intent interpretation, missing acceptable candidates, incompatible symbolic selection, numerical infeasibility or timeout, incomplete verification, unsuccessful repair, or backend mismatch. A method can pass every encoded constraint and still select a semantically wrong motif. Low NPE therefore cannot substitute for structural accuracy or SSS.

The conditional release property depends on the correctness and coverage of the graph, verifier, and backend contract. It does not guarantee that extracted knowledge is complete. The bounded search

can miss feasible configurations, and finite repair can refuse a recoverable candidate. Resource observations can become stale between synthesis and deployment, so a changed snapshot invalidates the earlier release decision and requires rechecking.

The interface evaluation addresses configuration acceptance and semantic preservation, not packet-processing throughput, real-world attack suppression, or vendor-certified behavior. Deployment would also require authorization, transactional updates, rollback, monitoring, and operational review. These requirements are distinct from intent-to-configuration synthesis.