A Self-Correcting Multi-Agent LLM Pipeline for Verified Kubernetes Network Policy

1st Umar Mahmood
Dept. of Electronics Engineering
Jeju National University
Jeju, Korea
umarmahmood637@gmail.com

2nd Wang-Cheol Song
Dept. of Electronics Engineering
Jeju National University
Jeju, Korea
philo@jejunu.ac.kr

***Abstract*—Authoring correct Kubernetes NetworkPolicy objects from natural language security requirements is difficult and error-prone. Large language models (LLMs) can automate this translation, but utility depends heavily on model size: powerful models with eight billion or more parameters achieve near-perfect accuracy yet demand hardware unsuitable for air-gapped or resource-constrained deployments, while smaller one-to-three billion parameter models produce correct policies only 70 to 75 percent of the time without additional support. This paper presents NetPolAgent, a multi-agent pipeline that closes this gap. Four specialised agents (a Generator, Critic, Verifier, and Refiner) run on locally-hosted models via Ollama. The Verifier is a deterministic test harness rather than an LLM: it applies each candidate policy to a live Kubernetes cluster and runs real connectivity tests between probe pods, with structured feedback passed to the Refiner for correction until all tests pass. We evaluate on a dataset of 1,410 policies covering CIDR-scoped egress and namespace-level ingress isolation. Large models achieve close to 100 percent with or without the pipeline; smaller models improve from 70 to 75 percent to 97 to 98 percent, reaching near parity with their larger counterparts. These results demonstrate that architecture, rather than raw model capacity, is the primary driver of correctness in resource-constrained deployments.**

***Index Terms*—Kubernetes, NetworkPolicy, large language models, multi-agent systems, network security, policy synthesis, self-correction**

I. Introduction

Container orchestration with Kubernetes has become the dominant model for deploying networked applications at scale. By default, every pod in a cluster can communicate with every other pod without restriction. NetworkPolicy objects are the Kubernetes-native mechanism for enforcing traffic segmentation: they specify which pods may send or receive traffic, on which ports, and from or to which namespaces or external CIDR ranges (Classless Inter-Domain Routing).

Writing correct NetworkPolicy objects is harder than it appears. The semantics are additive and whitelist-only, meaning every rule opens traffic rather than closing it. A missing `policyTypes` field silently disables enforcement without any API warning, and constructs such as a double-negative `ipBlock` with an `except` clause are easy to get wrong. Critically, the Kubernetes API accepts syntactically valid but semantically incorrect policies without complaint: a policy can pass API validation yet still allow all traffic because `policyTypes` was omitted or `podSelector` was left too broad. The only way to catch such mistakes is to test real connectivity in a live cluster.

LLMs offer a natural solution, translating plain-language security requirements directly into NetworkPolicy YAML [1]. Large instruction-tuned models with eight billion or more parameters achieve close to perfect accuracy [2], but require high-end GPU hardware and are often impractical in air-gapped or edge environments. Smaller one-to-three billion parameter models are far more practical to deploy locally, yet produce correct policies only 70 to 75 percent of the time without additional support, a real barrier to adoption in constrained environments.

We propose NetPolAgent, a multi-agent pipeline that closes this gap without requiring large models. A Critic agent checks each candidate policy for syntax and semantic issues. A Verifier deploys the policy to a live Kubernetes cluster and runs real connectivity tests, catching semantic errors that static analysis misses. A Refiner takes structured feedback from both and produces a corrected policy for the next iteration. The whole system runs on locally-hosted models via Ollama [3] with no external API calls. With this architecture, small-model pass rates rise from 70 to 75 percent to 97 to 98 percent, reaching near parity with much larger models. The main contributions of this paper are:

- A complete multi-agent pipeline (Section III) for translating natural language intents into verified Kubernetes NetworkPolicy YAML, runnable on commodity hardware with any Ollama-compatible model.
- A dataset of 1,410 annotated policy intents with ground-truth verified YAML (Section IV).
- An experimental study (Sections IV and V) showing that the pipeline raises small-model pass rates from 70–75% to 97–98%, while large models remain near 100% with or without the architecture.

Section II reviews related work. Section III describes the system. Section IV covers the experimental setup. Section V presents results. Section VI concludes.

*Corresponding Author: Wang-Cheol Song (email: philo@jejunu.ac.kr)
ISBN 978-3-903176-82-9

II. Related Work

Large language models trained on code have demonstrated strong capability at generating structured artefacts from natural language. Codex [4] and CodeGen [5] established that models trained on public code repositories can produce configuration files and shell commands from brief descriptions. More recent models such as DeepSeek-Coder [6] and instruction-tuned Llama 3 [2] extend this to interactive chat-style intent specification. A consistent finding is that generation quality scales with model size: larger models handle edge cases and structural constraints better, while smaller models are cheaper to deploy but require additional support to reach comparable accuracy.

Several projects have applied LLM generation to Infrastructure-as-Code (IaC) tools including Terraform, Ansible, and Helm [7, 8]. These studies find that models produce syntactically valid configurations but frequently miss semantic constraints specific to the target system. Commercial tools such as GitHub Copilot [9, 10] offer inline YAML completion but provide no validation layer. Kubernetes NetworkPolicy is a particularly demanding target because correctness cannot be verified from the manifest alone, actual traffic behaviour depends on the CNI plugin, namespace state, and pod labels at runtime, making empirical validation against a live cluster a necessary step.

Multi-agent decomposition has produced strong results across several domains. AutoGen [11] enables multi-agent conversations where agents adopt roles such as planner, executor, and reviewer. MetaGPT [12] maps software engineering roles to LLM agents passing structured artefacts between them. For network policy verification, VeriFlow [13] checks OpenFlow rules for reachability violations in real time, NP-Guard [14] statically analyses NetworkPolicy manifests for anti-patterns, and NetAssert [15] provides a declarative connectivity test framework. Teaching LLMs to self-correct via execution feedback has been explored in programming [16], temporal logic [17], and reinforcement-based reflection [18]. NetPolAgent combines role-separated agents with a deterministic live-cluster verifier, replacing an LLM reviewer with a test harness that catches semantic errors no static analysis tool can detect.

III. System Architecture

Figure 1 shows the NetPolAgent pipeline. The system takes a natural language network intent and works through a four-agent loop, iterating up to five correction cycles before reporting failure. Source code is available at [19].

A. Agent 1: Generator

The Generator takes a natural language intent and produces an initial candidate NetworkPolicy YAML via a locally-hosted LLM through Ollama [3]. Its system prompt includes the Kubernetes NetworkPolicy schema, whitelist semantics, structural rules, and few-shot examples covering every policy class in the dataset (Listing 1). When a policy fails, the previous candidate

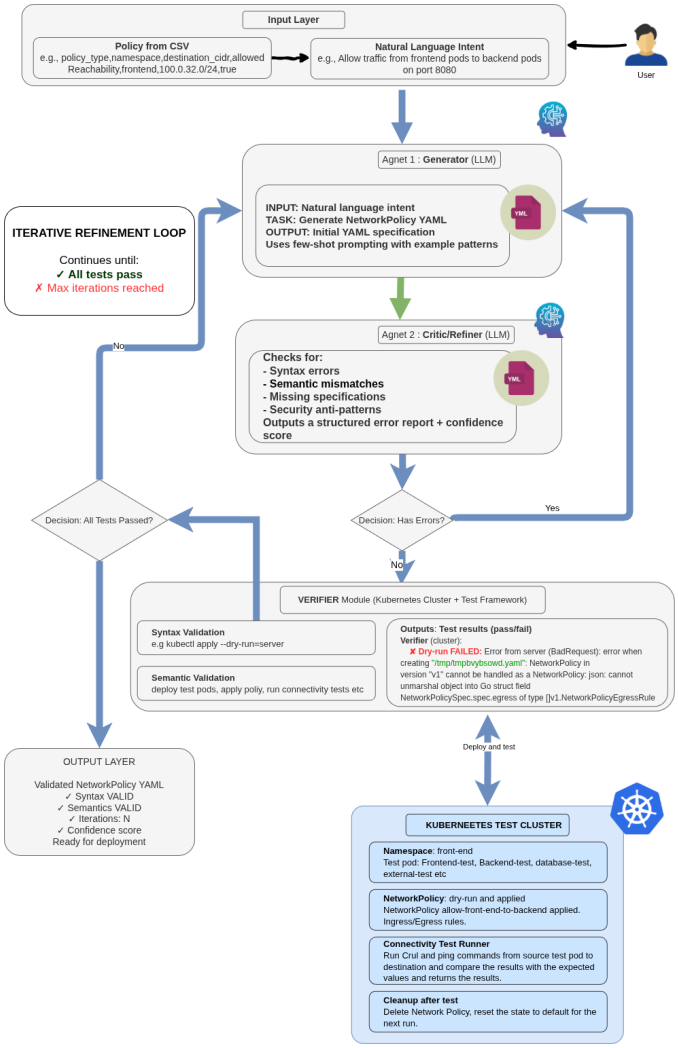

Fig. 1. Multi-Agent Self-Correcting Architecture for Kubernetes NetworkPolicy Generation and Validation. Four agents (Generator, Critic, Verifier, and Refiner) operate in an iterative refinement loop with empirical validation on a live Kubernetes test cluster.

YAML is passed back so the model builds on its prior attempt rather than starting over.

Output is constrained to valid YAML only, with every response required to include `apiVersion`, `kind`, `metadata.name`, `metadata.namespace`, `spec.podSelector`, and `spec.policyTypes`, since the Verifier consumes output directly with no post-processing. Without the correction loop, small models fail predictably: omitting `policyTypes`, misapplying whitelist semantics, mixing up ingress and egress rule structure, or using invalid protocol values such as ICMP. Figure 2 shows typical examples.

```
You are a Kubernetes NetworkPolicy YAML
    generator.
Your ONLY output is valid YAML -- no prose, no
    markdown
fences, no explanation.
Rules:
- Always include: apiVersion, kind, metadata.
```

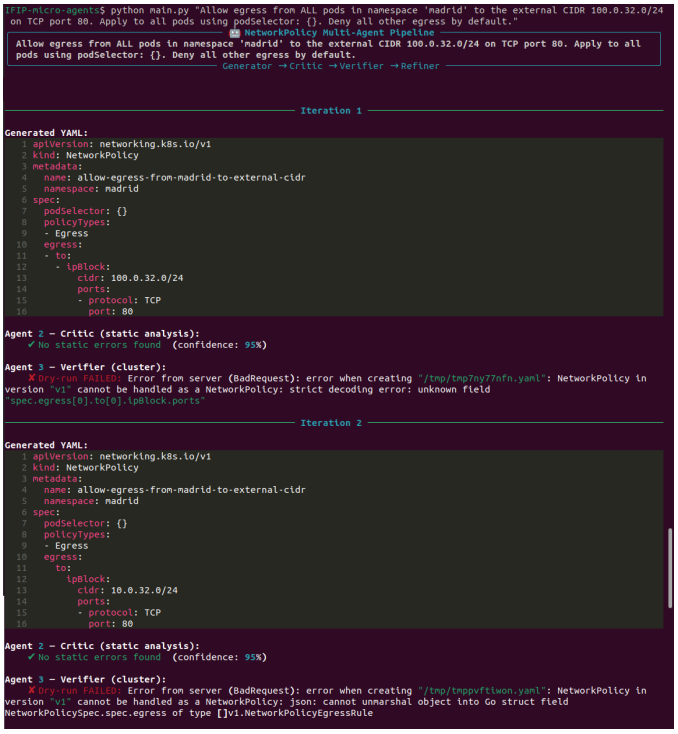

Fig. 2. Examples of Failure of Small Models

```
    name,
  metadata.namespace, spec.podSelector, spec.
      policyTypes
- Always specify policyTypes explicitly (Ingress
    , Egress,
  or both)
- For policies that apply to ALL pods, use
    podSelector: {}
- NEVER invent pod labels unless the intent
    names a
  specific app
- Protocols: TCP/UDP/SCTP only(Stream Control
  Transmission Protocol)) -- ICMP is NOT valid
Few-shot examples:
# Example 1: deny-all ingress (isolation)
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
...
```

Listing 1. Generator system prompt.

B. Agent 2: Critic

The Critic checks each candidate policy against the original intent, returning a structured JSON report (Listing 2). A model that just wrote a policy is not well placed to spot its own mistakes, a phenomenon known as self-confirmation bias (the tendency of a model to overlook errors in its own output); giving the same model a different role, that of a security auditor with a JSON output contract rather than a YAML generator, shifts its objective from synthesis to diagnosis. No additional model weights are needed, and the only extra cost is one inference call per iteration.

```
You are a Kubernetes NetworkPolicy security
    auditor.
Analyze the given YAML against the stated intent
    .
Return ONLY a JSON object -- no prose, no
    markdown fences.
Schema:
{
  "has_errors": <bool>,
  "errors": [
    {"kind": "<syntax|semantic|security>",
     "message": "<description>"}
  ],
  "confidence": <float 0.0-1.0>
}
Check for:
SYNTAX:
- Missing required fields: apiVersion, kind,
  metadata.name, metadata.namespace, spec.
      podSelector,
  spec.policyTypes
- Invalid protocol values (ICMP is not valid)
- Malformed YAML structure
SEMANTIC:
- Does the policy match the stated intent?
- Are selectors targeting the right pods?
- Are ports correct and complete?
```

Listing 2. Critic system prompt.

C. Agent 3: Verifier

The Verifier is a deterministic Python module, not an LLM. It talks to a live Kind Kubernetes test cluster running Kindnet CNI [3] and works in two phases.

1) Phase 1: API Dry-Run Validation: The candidate YAML is submitted to the Kubernetes API server using `kubectl apply --dry-run=server` before any pod is scheduled, catching schema violations, unsupported field names, invalid enum values, and malformed resource names. If this step fails, the Verifier returns an error report immediately and skips Phase 2, saving the cost of pod scheduling.

2) Phase 2: Live Connectivity Validation: If Phase 1 passes, the Verifier deploys the policy with lightweight `busybox:1.36` probe pods. For ingress isolation policies, one authorised and one unauthorised client pod are deployed alongside a server pod in the policy namespace, with connectivity tested using `nc -zw 5 <pod-ip> <port>`. For egress reachability policies, the client sits in the policy namespace while the server goes into a separate `netpol-client` namespace, simulating an external endpoint without ingress interference.

Expected outcomes come directly from the YAML. A policy permitting egress only to a specific external CIDR should block connections to the test server, whose pod IP (from `10.244.x.x`) falls outside that range — a failure that passes API validation silently and is one of the most common small-model errors in single-shot mode. All probe pods and NetworkPolicy objects are cleaned up after each run before the next iteration starts.

D. Agent 4: Refiner

The Refiner receives a consolidated error report from the Critic, the API dry-run, and the Verifier's connectivity tests (Listings 3 and 4), and produces a corrected YAML. The key

design choice is minimal editing: the Refiner only touches fields flagged in the error report and leaves everything else in place, since full regeneration proved counterproductive in practice, often reintroducing mistakes that had already been fixed.

```
You are a Kubernetes NetworkPolicy YAML repair
    agent.
Fix the provided YAML based on the listed errors
    .
Output ONLY corrected YAML -- no prose, no
    markdown fences.
Rules:
- Make the MINIMUM changes necessary to fix the
    errors
- Preserve all correct structure and intent
- Protocols: only TCP, UDP, SCTP (never ICMP)
- Always include policyTypes
- Match the original intent exactly
- Namespace in metadata must match the intent's
    namespace
```

Listing 3. Refiner system prompt.

```
Intent: <natural language description>
Current YAML:
<previous candidate yaml>
Errors to fix:
=== STATIC ANALYSIS ERRORS ===
[SYNTAX] Missing spec.policyTypes field
[SEMANTIC] Selector targets wrong pods
=== API DRY-RUN ERROR ===
spec.policyTypes: Required value
=== FAILED CONNECTIVITY TESTS ===
  egress-client -> external-server:80:
    expected BLOCKED, got ALLOWED
Return corrected YAML only:
```

Listing 4. Refiner user prompt showing error consolidation.

E. Pipeline Modes and Implementation

The pipeline runs in three modes. `no_refine` runs one pass through the Generator, Critic, and Verifier with no correction step, which lets us isolate how much the Refiner actually contributes. `full` runs the complete loop for up to five iterations and is the standard configuration. `batch` takes a CSV of policies, runs each one through `full`, and writes pass/fail results, error types, and iteration counts to an output file. All large-scale experiments used this mode.

All LLM agents talk to models served locally by Ollama, with no external network calls. The Verifier runs `kubectl` commands via Python subprocess against a single-node Kind cluster on Kubernetes v1.31.2 with Kindnet CNI. Any Ollama-compatible model can be swapped in using a single `--model` flag. The source code and dataset are available at [19]. Listing 5 shows a representative verified output.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-madrid
  namespace: madrid
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 100.0.32.0/24
    ports:
    - protocol: TCP
      port: 80
```

Listing 5. Verified output for: "Allow egress from all pods in namespace madrid to CIDR 100.0.32.0/24 on TCP port 80. Deny all other egress."

TABLE I
LLM MODEL CONFIGURATIONS USED IN EXPERIMENTS.

Configuration	Model (Generator / Critic / Refiner)
Large (DeepSeek)	deepseek-coder-v2
Large (LLaMA3-70B)	llama3:70b-instruct-q4_K_M
Small (Qwen)	qwen2.5-coder:3b
Small (LLaMA3.2)	llama3.2:3b
Tiny (TinyLlama)	tinyllama:latest

IV. EXPERIMENTAL SETUP

All connectivity tests are executed on a single-node Kind (Kubernetes-in-Docker) cluster running Kubernetes v1.31.2 with the default Kindnet CNI plugin. Since NetworkPolicy correctness is defined at the API and CNI enforcement layer, isolation and traffic filtering rules can be verified independently of cluster scale or multi-node topology. Each verification run deploys pods into isolated test namespaces and removes them after completion to ensure no state persists across experiments.

Table I lists the five model configurations evaluated, all served locally via Ollama [3] on the same host as the test cluster with no external API calls. The two large configurations (`deepseek-coder-v2` and `llama3:70b-instruct`) represent the upper end of what is practical on a single high-end workstation. The two small configurations use `qwen2.5-coder:3b` and `llama3.2:3b`. TinyLlama (1.1B) is included as a lower-bound stress test.

Every configuration is evaluated on the complete 1,410-row dataset in `full` pipeline mode (up to five Generator–Critic–Verifier–Refiner iterations per policy), measuring ceiling accuracy when the entire correction loop is active. All models are also evaluated in `no_refine` mode on the same dataset, producing a direct before/after comparison that isolates the Refiner's contribution. TinyLlama is additionally run in `full` mode to verify the pipeline correctly reports failures rather than fabricating pass results when the underlying model cannot produce valid YAML.

We report four metrics: *pass rate* (percentage of policies passing all Verifier connectivity tests); *error type* (syntax errors rejected by the API dry-run vs. semantic errors accepted by the API but failing connectivity tests); *average iterations* (mean correction cycles across all passing policies in `full` mode); and *first-shot rate* (percentage of policies passing on the first Generator attempt without invoking the Refiner).

V. RESULTS

Figure 3 and Table II summarise pass rates across all five model configurations. Both large models (DeepSeek-R1

TABLE II
PASS RATES BY MODEL AND PIPELINE MODE ($N = 1{,}410$).

Model	Mode	Pass / Total	Rate
DeepSeek-R1	full	1410/1410	100.0%
LLaMA3-70B Instruct	full	1410/1410	100.0%
Qwen2.5-Coder (3B)	no_refine	1099/1410	77.8%
Qwen2.5-Coder (3B)	full	1370/1410	97.2%
LLaMA3.2 (3B)	no_refine	1043/1410	73.8%
LLaMA3.2 (3B)	full	1362/1410	96.6%
TinyLlama (1.1B)	full	0/1410	0.0%

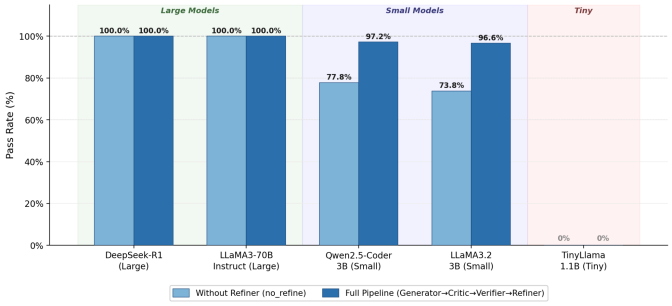

Fig. 3. Pass rate (%) per model under no_refine and full pipeline modes. Large models achieve 100% in both modes. The 3B models gain 19–23% from the correction loop. TinyLlama achieves 0% in both modes.

TABLE III
FAILURE TYPE BREAKDOWN ACROSS PIPELINE MODES.

Model	Mode	Syntax	Semantic
DeepSeek-R1 / LLaMA3-70B	full	0	0
Qwen2.5-Coder (3B)	no_refine	220	93
	full	28	12
LLaMA3.2 (3B)	no_refine	272	97
	full	33	15
TinyLlama (1.1B)	full	1410	0

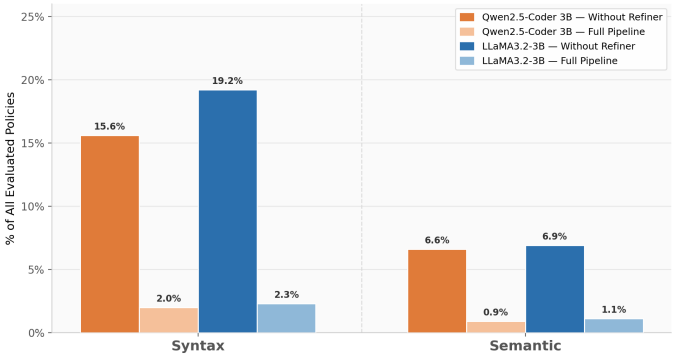

Fig. 4. Syntax and semantic failure counts for the two 3B models before and after the correction loop.

and LLaMA3-70B) hit 100% in both modes, confirming that models at this scale do not need the correction loop. Without the refiner, Qwen2.5-Coder:3b reaches 77.8% and LLaMA3.2-3B reaches 73.8%; enabling the full pipeline brings them to 97.2% and 96.6% respectively, a gain of 19 to 23 percentage points. TinyLlama (1.1B) scores 0% in both modes.

Figure 4 and Table III break failures down into syntax errors (rejected by the API dry-run) and semantic errors (accepted by the API but failing live connectivity tests). For both 3B models, syntax errors dominate in no_refine mode: 220 syntax and 93 semantic for Qwen, and 272 syntax and 97 semantic for LLaMA3.2. The full pipeline cuts both categories sharply, leaving 28 syntax and 12 semantic for Qwen, and 33 syntax and 15 semantic for LLaMA3.2. Semantic errors are the more dangerous of the two: the Kubernetes API accepts these policies silently, so only the live Verifier catches them. TinyLlama produces only syntax errors across all 1,410 policies, confirming it cannot generate well-formed YAML at any point in the pipeline.

Figure 5 shows how many correction cycles policies actually need. Large models average 1.011 iterations per passing policy, meaning 99.3% pass on the first attempt. For the 3B models, first-shot rates are 92.2% for Qwen and 95.1% for LLaMA3.2, with means of 1.143 and 1.082 iterations respectively, keeping overhead well below one extra LLM call per policy on average. Figure 6 profiles wall-clock time per iteration, the Verifier accounts for 7.86–7.98 s (≈74–81% of total per-iteration time) due to pod scheduling and live connectivity tests, while the Generator (1.05–1.07 s), Critic (0.63–0.73 s), and Refiner (1.12 s) together contribute under 3 s, keeping the full pipeline under 11 s per policy on average.

VI. CONCLUSION

NetPolAgent shows that architecture can substitute for model size in resource-constrained environments. Small 3B models produce correct Kubernetes NetworkPolicy objects only 70 to 75 percent of the time on their own, but wrapping them in the Generator, Critic, Verifier, and Refiner loop brings that to 97 to 98 percent, matching much larger models. The Verifier is the most important piece: it is the only component that catches semantic errors where the generated YAML passes API validation but implements the wrong traffic behaviour. These silent mismatches are the most dangerous failure mode in production, and no static analysis tool can find them. The whole pipeline runs locally via Ollama with no external API calls, which makes it practical for air-gapped and edge deployments.

A. Limitations and Future Work

The dataset covers three policy patterns, which is a narrow slice of what real clusters need. Policies with `namespaceSelector`, combined ingress and egress rules, and multi-port specifications are common in production but absent here. All experiments also use Kindnet as the CNI plugin; testing against Calico and Cilium would confirm consistent behaviour across implementations. On the model side, all agent roles currently use the same underlying model; heterogeneous assignments, for example pairing a 3B generator with a larger refiner, could push pass rates closer to

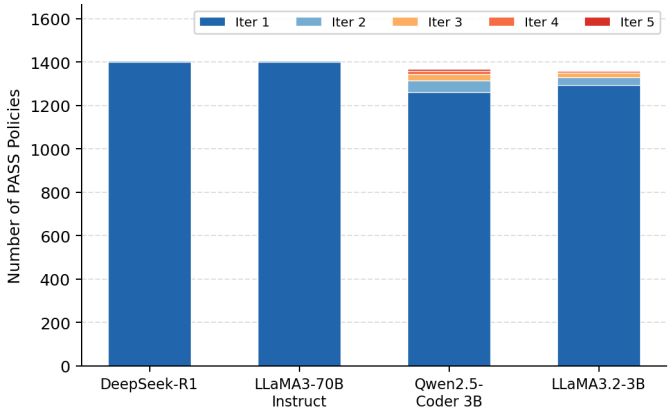

Fig. 5. Stacked distribution of correction cycles for passing policies. Most pass on the first attempt for all models.

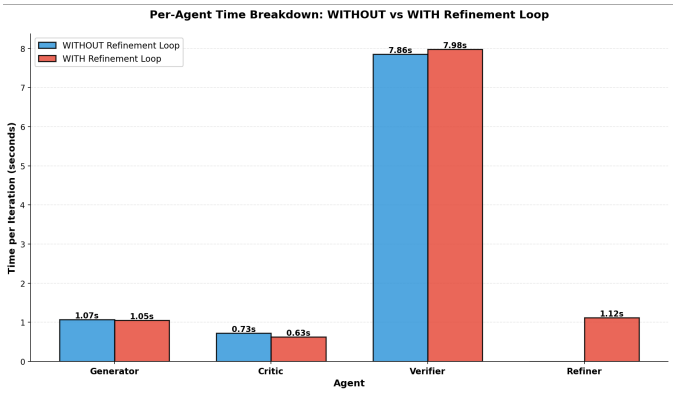

Fig. 6. Per-agent wall-clock time per iteration for the 3B models under `no_refine` and `full` pipeline modes.

100 percent. Finally, supporting ambiguous or underspecified natural language would make the tool much more accessible to non-expert users.

Acknowledgment

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT)(No.RS-2024-00398379, Development of High Available and High Performance 6G Cross Cloud Infrastructure Technology).

References

[1] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa, "Netconfeval: Can llms facilitate network configuration?" *Proceedings of the ACM on Networking*, vol. 2, no. CoNEXT2, pp. 1–25, 2024.

[2] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[3] Ollama Team, "Ollama: Run large language models locally," [Online]. Available: https://ollama.com, 2024.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[5] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[6] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma *et al.*, "Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[7] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, "A survey of using large language models for generating infrastructure as code," in *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*, 2023, pp. 523–533.

[8] Z. Namrud, K. Sarda, M. Litoiu, L. Shwartz, and I. Watts, "Kubeplaybook: A repository of ansible playbooks for kubernetes auto-remediation with llms," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 57–61.

[9] GitHub, Inc., "GitHub Copilot," [Online]. Available: https://copilot.github.com, 2022.

[10] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[11] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *First conference on language modeling*, 2024.

[12] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework," in *The twelfth international conference on learning representations*, 2023.

[13] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the first workshop on Hot topics in software defined networks*, 2012, pp. 49–54.

[14] np-guard, "Network config analyzer," https://github.com/np-guard/network-config-analyzer, 2024, accessed: 2026-02-28. [Online]. Available: https://github.com/np-guard/network-config-analyzer

[15] Control Plane, "netassert: Network security testing for DevSecOps," [Online]. Available: https://github.com/controlplaneio/netassert, 2019.

[16] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[17] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, and C. Trippel, "nl2spec: Interactively translating unstructured natural language to temporal logics with large language models," in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 383–396.

[18] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in neural information processing systems*, vol. 36, pp. 8634–8652, 2023.

[19] "NetPolAgent: A self-correcting multi-agent LLM pipeline for verified Kubernetes network policy," GitHub repository, 2025, accessed: 2026-02-28. [Online]. Available: https://anonymous.4open.science/r/IFIP-mini-agents-for-kuberneetes-Neworkpolicies-36B2