ConfigSwarm: A Multi-Agent Framework on Versioned Kernel Configuration Graphs

Yang Han[1,2,3], Kaichun Yao[1], Libo Zhang[1]
[1]Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
[3]University of Chinese Academy of Sciences, Beijing, China
Email: hanyang23@mails.ucas.ac.cn, yaokaichun@outlook.com, libo@iscas.ac.cn

***Abstract*—Linux kernel configuration is critical for workload-specific performance, security, and resource footprint, but it remains difficult to automate. Any automated approach must generate configuration edits that satisfy executable `Kconfig` constraints and remain robust under rapid kernel evolution, where options and constraint logic drift across releases. Existing LLM-based approaches often produce non-executable edits, treat each release as an isolated snapshot, and fail to retain execution feedback as reusable knowledge. We propose ConfigSwarm, a multi-agent framework for kernel configuration QA and tuning under version drift. ConfigSwarm grounds agent reasoning in the Versioned Kernel Configuration Knowledge Graph (VKConfig-KG), which combines executable `Kconfig` constraints with a lightweight semantic taxonomy for intent grounding. To enable efficient cross-version migration, ConfigSwarm adopts a Backbone-and-Diff factorization that reuses an anchor backbone and localizes maintenance to sparse deltas. Four specialized agents (*Interaction*, *Parsing*, *Alignment*, and *Validation*) coordinate via typed artifacts and a validation-in-the-loop policy with staged checks from static feasibility to compile/boot and workload measurements, while writing structured outcomes back to VKConfig-KG for iterative refinement. Experiments show that ConfigSwarm improves QA reliability and closed-loop tuning validity, reduces cross-version adaptation cost by over 86%, and raises build success rate to 73.3%.**

***Index Terms*—Linux kernel configuration, LLMs, Multi-agent systems, Kernel version drift, Constraint reasoning.**

I. Introduction

Customizing the Linux kernel configuration is critical for workload-specific objectives in performance, security, and resource footprint. It requires selecting and tuning thousands of build-time options that are organized by hierarchical menus and constrained by dependency logic, selections, and mutual exclusions [1]. Modern releases expose more than 18,000 options [2], [3]. Moreover, the practical difficulty is compounded by *kernel evolution*: across adjacent versions, options may be added or removed, constraint expressions can change, and subsystem behaviors may shift, making previously valid configurations and tuning heuristics invalid or suboptimal. As a result, kernel configuration in real deployments is a moving target, and recommendations must remain executable under hard constraints and transferable under version drift rather than assuming a static option set or a single fixed environment.

Corresponding author: libo@iscas.ac.cn

Although recent work has explored LLM-based assistance for kernel configuration and tuning (e.g., AutoOS [4]), robustness under kernel evolution remains limited. First, natural-language-driven suggestions often fail to respect executable `Kconfig` logic (e.g., `depends on`, `select`, mutual exclusions, and `choice` rules), yielding configurations that cannot be built or booted in practice [5]. Second, version migration is often handled in a snapshot-based manner: each release triggers a full re-parse of `Kconfig` sources and re-computation over the entire option space, even when changes are sparse. This incurs high wall-clock overhead (compile/boot/benchmark cycles) and unnecessary token cost. Third, execution signals (constraint violations, build/boot failures, workload regressions) are rarely persisted as structured state, so the same failure modes are repeatedly rediscovered across iterations and versions.

To address these challenges, we propose **ConfigSwarm**, a multi-agent framework for kernel configuration QA and tuning under version drift. ConfigSwarm grounds LLM reasoning in a shared, version-scoped substrate, the versioned kernel configuration knowledge graph (**VKConfig-KG**), which stores executable `Kconfig` constraints for feasibility checking and a lightweight semantic taxonomy for intent grounding. To avoid full rebuilds across releases, we introduce a **Backbone-and-Diff** factorization that reuses an anchor release and applies sparse deltas to materialize version-specific views on demand. Building on this substrate, specialized agents translate user intent into version-scoped edits and validate candidates through staged checks, from static constraint checking to compile/boot tests and, when needed, workload measurements; validation outcomes are written back as structured updates to guide subsequent iterations and future versions. Experiments show that ConfigSwarm improves the reliability of graph-grounded diagnostic QA and closed-loop tuning under a fixed verification budget, while reducing cross-version adaptation cost by over 86% and raising build success rate to 73.3% (vs. 26.7% for LLM-only).

To summarize, our contributions are:

1) We propose **ConfigSwarm**, a multi-agent framework for kernel configuration QA and workload-driven tuning under version drift, built around typed coordination artifacts and a shared, versioned state (**VKConfig-KG**).
2) We introduce a **Backbone-and-Diff** factorization for ver-

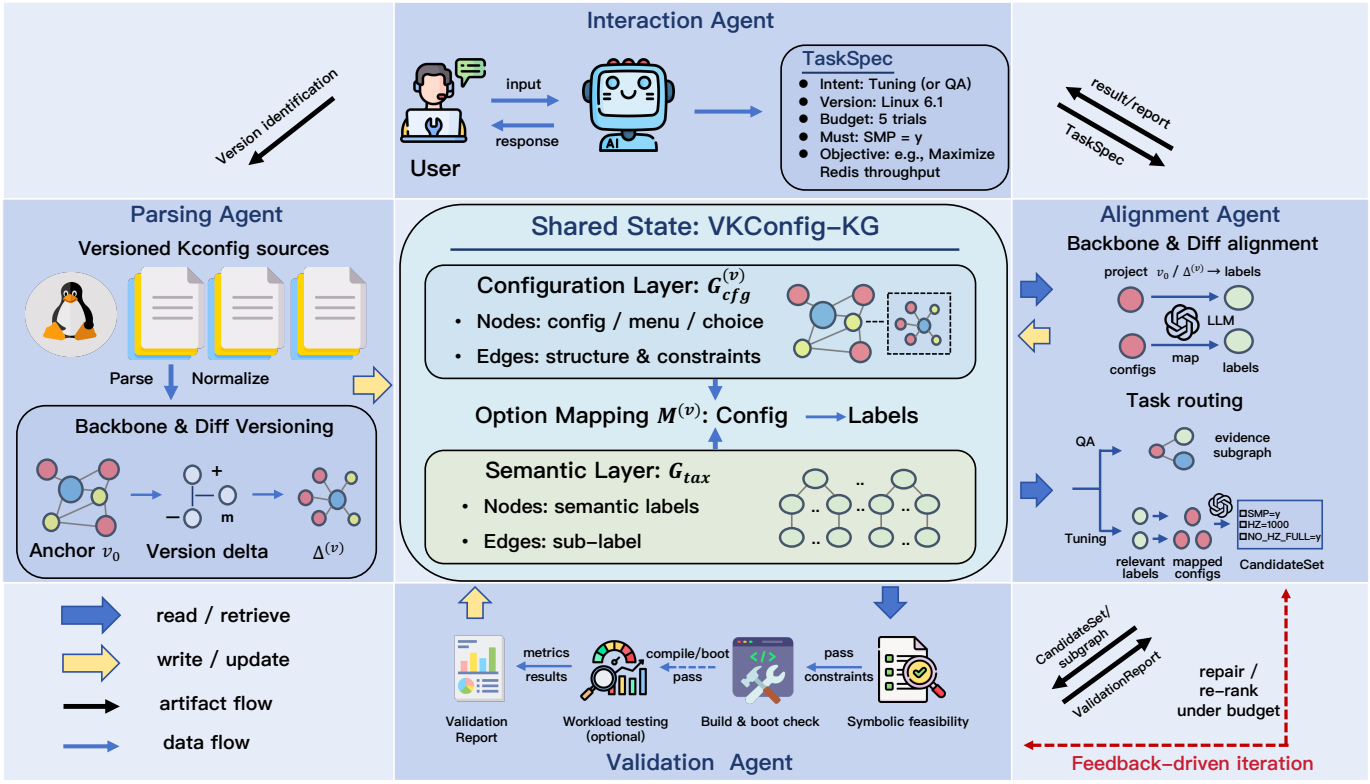

Fig. 1: ConfigSwarm framework. The Interaction agent converts requests into a TaskSpec (intent, version, constraints, objective, budget). Parsing and Alignment retrieve constraint-aware evidence for QA or propose candidate edits for tuning, and the Validation agent performs staged checks (feasibility, compile/boot, optional workload) and writes back structured outcomes to VKConfig-KG for feedback-driven iteration.

sioned configuration reasoning that localizes maintenance and alignment to sparse release changes, avoiding full re-processing per version.

3) We design a **validation-in-the-loop** tuning policy with staged checks and structured write-back, enabling reliable optimization and reusable failure attribution across iterations and kernel versions.

II. The ConfigSwarm Framework

ConfigSwarm is a multi-agent framework for kernel configuration QA and tuning under kernel evolution, as illustrated in Figure 1. It orchestrates specialized agents (parsing, alignment, validation, and interaction) via structured intermediate artifacts and a shared, version-aware state, referred to as the Versioned Kernel Configuration Graph (VKConfig-KG), which exposes executable configuration constraints and a lightweight semantic taxonomy for intent grounding. This section introduces the shared-state schema with backbone–diff versioning, followed by agent roles and task workflows.

A. Shared State: VKConfig-KG with Backbone-and-Diff Versioning

a) State schema: VKConfig-KG serves as the shared state of ConfigSwarm, providing a version-consistent substrate for constraint reasoning, semantic grounding, and validation write-back. Formally, we represent it as

$$\mathcal{S} = \Big(\{\mathcal{G}_{\text{cfg}}^{(v)}\}_{v \in \mathcal{V}_{\text{ver}}}, \mathcal{G}_{\text{tax}}, \{\mathcal{M}^{(v)}\}_{v \in \mathcal{V}_{\text{ver}}} \Big), \tag{1}$$

where $\mathcal{G}_{\text{cfg}}^{(v)}$ denotes the version-scoped configuration graph, $\mathcal{G}_{\text{tax}}$ is the shared semantic taxonomy, and $\mathcal{M}^{(v)}$ links configuration entities to semantic labels. In implementation, the shared state also carries structured validation outcomes and write-back records produced during task execution.

VKConfig-KG consists of two coupled layers: (i) a *configuration layer* that stores per-version configuration graphs $\{\mathcal{G}_{\text{cfg}}^{(v)}\}_v$, whose nodes represent configuration entities (e.g., `config`, `menu`, `choice`) and whose edges encode hierarchical structure and hard constraints (e.g., dependencies, selections, mutual exclusions, and choice rules); and (ii) a shared *semantic layer* $\mathcal{G}_{\text{tax}}$ that organizes semantic labels, capturing kernel capabilities and subsystems, as a lightweight taxonomy.

The configuration layer is instantiated by parsing version-specific `Kconfig` sources, whereas $\mathcal{G}_{\text{tax}}$ is curated from the kernel's subsystem decomposition and capability hierarchy, providing a stable semantic coordinate system across versions. Version-scoped mappings $\mathcal{M}^{(v)}$ are maintained by the agent workflows; for the anchor release, $\mathcal{M}^{(v_0)}$ is bootstrapped from

option descriptors (`help`/`prompt`) and menu paths. Together, $\mathcal{G}_{\text{tax}}$ and $\mathcal{M}^{(v)}$ support intent grounding by routing task intents and constraints through the label space to relevant options, while drift-aware maintenance is localized to options touched by $\Delta^{(v)}$.

b) Backbone-and-diff evolution: To handle kernel drift efficiently, we select an anchor release v_0 (e.g., Linux 6.1) and materialize its configuration graph $\mathcal{G}_{\text{cfg}}^{(v_0)}$ as a backbone snapshot. On this anchor, we also bootstrap the option–label mappings $\mathcal{M}^{(v_0)}$ into the shared taxonomy $\mathcal{G}_{\text{tax}}$, establishing both a structural and semantic anchor for subsequent versions.

For each subsequent version v, rather than rebuilding the full configuration graph from scratch, we record only the incremental changes relative to the backbone as a version diff

$$\Delta^{(v)} = \left(\Delta_{\text{V}}^{(v)}, \Delta_{\text{E}}^{(v)}, \Delta_{\text{attr}}^{(v)}\right), \tag{2}$$

where the three terms denote changed configuration entities, changed structural or constraint edges, and updated attributes such as `help` text, default values, or modified constraint expressions.

The version-scoped configuration view is then recovered by applying the diff to the backbone:

$$\mathcal{G}_{\text{cfg}}^{(v)} = \textsc{Apply}\left(\mathcal{G}_{\text{cfg}}^{(v_0)}, \Delta^{(v)}\right). \tag{3}$$

This differential view supports incremental maintenance and localizes mapping updates, since only options touched by $\Delta^{(v)}$ need to be reconsidered for $\mathcal{M}^{(v)}$.

In our implementation, the backbone snapshot and per-version diffs are pre-computed offline, while version-scoped views are materialized on demand when a request targets specific release(s). Together, VKConfig-KG and backbone–diff versioning define the shared state for ConfigSwarm: agents query version-scoped constraints, provenance, and taxonomy anchors, and write back mapping and validation updates during task execution.

B. Agents and Structured Artifacts

ConfigSwarm decomposes end-to-end kernel configuration QA and tuning under kernel evolution into four specialized agents: Interaction, Parsing, Alignment, and Validation. Agents communicate via typed intermediate artifacts and interact through VKConfig-KG as a shared state, enabling modular execution and incremental state maintenance under version drift.

Typed artifacts. Agents coordinate through three typed artifacts: *TaskSpec*, which specifies intent (QA vs. tuning), target version(s), objectives, hard constraints (must/forbid), and a validation budget; *CandidateSet*, which contains a ranked list of version-scoped configuration edits (patches) with brief rationales; and *ValidationReport*, which records feasibility and execution outcomes (constraint checks, compile/boot, and optional workload metrics) together with failure attribution. Selected fields are written back to VKConfig-KG as confidence updates and version-specific exceptions.

Orchestration policy. ConfigSwarm selects an execution path based on the *TaskSpec*. QA requests typically invoke parsing and alignment to retrieve supporting constraints and semantic anchors, while tuning requests additionally invoke validation for executable feedback. Under limited budgets (e.g., a bounded number of compile/boot/benchmark trials), validation is staged from lightweight constraint checks to compile/boot tests and workload measurements.

a) Interaction agent: The interaction agent is the user-facing controller. It converts a natural-language request into a structured *TaskSpec* (cf. typed artifacts), which serves as the execution contract for the downstream agents and determines the workflow (QA-only vs. validation-in-the-loop). Finally, it aggregates downstream artifacts into a user-facing response: QA requests return evidence-grounded answers with traceable constraint/provenance references, while tuning requests return the validated configuration patch(es) from the *CandidateSet* together with feasibility outcomes and measurements summarized in the *ValidationReport*.

b) Parsing agent: Given a target version v, the parsing agent provides a version-scoped physical view $\mathcal{G}_{\text{cfg}}^{(v)}$ by applying the pre-computed diff $\Delta^{(v)}$ to the backbone snapshot. It extracts configuration entities and hierarchies, and normalizes executable constraint logic from `Kconfig` (e.g., `DEPENDS`, `SELECTS`, and choice rules) into a queryable form used by downstream agents. The agent also attaches provenance metadata (file path, menu context, and `help` text) to support traceable diagnostics and failure attribution.

c) Alignment agent: The alignment agent maintains version-scoped option–label mappings $\mathcal{M}^{(v)}$ and uses them to route user intents to executable configuration edits. For newly added or modified options in $\Delta^{(v)}$, it performs *diff-to-taxonomy projection*: it gathers the option's textual descriptors (`prompt`/`help`) together with its local structural context (menu path and dependency/selection neighborhood), and assigns one or more label nodes in $\mathcal{G}_{\text{tax}}$ with supporting evidence. Building on these mappings, the agent supports two retrieval modes: for QA, it extracts a compact *evidence subgraph* around the queried option(s) or the user's stated intent, including prerequisite chains, conflicting constraints, and provenance; for tuning, it resolves the objective into a small set of relevant taxonomy labels, retrieves candidate options via $\mathcal{M}^{(v)}$, and composes a ranked *CandidateSet* of executable edit patches. Before invoking executable validation, it applies lightweight local screening based on dependency, mutual-exclusion, and choice constraints to remove obviously infeasible edits.

d) Validation agent: The validation agent acts as the system's grounding critic and closes the loop by turning executable outcomes into structured evidence. It evaluates each candidate in an isolated harness using a staged verification policy: (i) symbolic feasibility checks (e.g., SAT-based) against `DEPENDS`/`SELECTS`, mutual exclusions, and choice rules; (ii) engineering viability tests (e.g., compilation and boot sanity checks) to filter build and boot failures; and (iii) workload measurements when quantitative tuning is requested. It outputs a *ValidationReport* with outcomes, key metrics, and failure attribution, and writes back reusable updates to VKConfig-

KG (e.g., version-specific exceptions and validated traces) to steer subsequent iterations.

C. Coordination Workflows for QA and Tuning

Given a user request, the interaction agent produces a *TaskSpec* that specifies target kernel version(s), hard constraints, objective metric(s), and a validation budget. ConfigSwarm then routes the request through one of two coordination workflows: (A) a graph-grounded QA workflow, where answers are generated conditioned on graph-retrieved constraints and provenance; and (B) a feedback-driven tuning workflow, where configuration edits are iteratively refined under executable validation.

a) Workflow A: Graph-grounded QA: For QA requests, the parsing agent materializes the version-scoped view $\mathcal{G}_{\text{cfg}}^{(v)}$ (when needed) and exposes executable constraints with provenance. The alignment agent performs graph-based retrieval to extract a compact evidence subgraph around the queried option(s) or semantic label(s), including key dependency chains, conflicts, and taxonomy anchors. Conditioned on this evidence, the interaction agent synthesizes an answer and returns traceable references (option identifiers, version tags, and constraint paths). When verification is requested (or feasibility is uncertain), the validation agent is invoked in a lightweight mode to confirm constraint consistency.

b) Workflow B: Feedback-driven tuning: For tuning requests, the alignment agent proposes an initial *CandidateSet* of version-scoped configuration edits. The validation agent evaluates candidates with staged executable checks (symbolic feasibility, compile/boot, and optional workload measurement) and returns a *ValidationReport* as feedback. Using this feedback, the alignment agent repairs and re-ranks candidates over iterations until the budget is exhausted, returning the best validated configuration; key outcomes are recorded in VKConfig-KG for reuse across iterations and versions.

Both workflows operate on the same versioned substrate. QA mainly performs graph-grounded retrieval to produce traceable answers and invokes constraint-consistency checks only when verification is needed; tuning, in contrast, uses staged executable validation as the optimization signal and records outcomes for subsequent iterations. We evaluate both workflows under kernel evolution in Section III.

III. Experiments

We evaluate ConfigSwarm under kernel evolution from three perspectives. First, we run a lightweight *graph-grounded QA* suite to assess whether VKConfig-KG provides reliable evidence for dependency reasoning and version drift diagnosis. Second, we measure *feedback-driven tuning* performance under a strict validation budget across representative workloads. Third, we quantify *adaptation efficiency* under backbone–diff versioning and conduct ablations to attribute gains to executable constraints and the feedback loop.

A. Experimental Setup

Compute. Experiments run on an Intel Xeon Platinum 8153 (2.0 GHz) server with 256 GB RAM and 16 TB SSD, running Ubuntu 22.04.4 LTS (x86_64).

Storage. VKConfig-KG is stored in Neo4j [6] and accessed via Cypher queries during agent execution.

Kernel scope. We use Linux 6.1 as the anchor release (backbone), where we parse $\mathcal{G}_{\text{cfg}}^{(6.1)}$ and initialize config–label mappings to the curated taxonomy $\mathcal{G}_{\text{tax}}$. We evaluate version drift adaptation to Linux 6.2–6.5 via backbone–diff updates. Tuning experiments are conducted on the Linux 6.1 backbone to isolate optimization effects from version drift.

Workloads and benchmarks. We tune for Redis (6.0.16) throughput and MySQL (8.0.44) P95 latency. Measurements use redis-benchmark (6.0.16) [7] and sysbench (1.0.20) [8]. We also report UnixBench (5.1.3) [9] as a general-purpose performance reference.

LLM. We use *gpt-5-mini* [10] for all components that require LLM inference. We keep decoding parameters fixed (e.g., temperature and max tokens), and use task-specific prompt templates for QA vs. tuning.

B. Graph-grounded QA Evaluation

TABLE I: **Diagnostic validity under kernel drift.** Prerequisite recovery for constraint reasoning and change-type classification for drift identification. **Gain** reports the relative improvement of ConfigSwarm over the LLM-only baseline.

Diagnostic Task	Metric	LLM-only	ConfigSwarm	Gain (×)
Constraint Reasoning	Exact Match (EM)	0.250	**0.450**	1.8
	Recall (Prereq.)	0.387	**0.736**	1.9
	Macro F1	0.435	**0.768**	1.8
Drift Identification	Accuracy	0.300	**0.850**	2.8
	Macro F1	0.222	**0.849**	3.8

We evaluate a lightweight *graph-grounded* diagnostic QA suite to assess whether VKConfig-KG provides reliable evidence for two capabilities: (i) prerequisite recovery for constraint reasoning and (ii) drift identification via change-type classification under version drift. The suite contains 200 queries spanning Linux 6.1–6.5 and covers both newly introduced options and modified dependency expressions. Unlike an LLM-only baseline, answers are generated conditioned on graph-retrieved constraint evidence and provenance, so the suite directly reflects the reliability of graph grounding. Table I shows that ConfigSwarm improves prerequisite coverage and exact matching by about 1.8–1.9×, and boosts drift identification by 2.8–3.8× over the LLM-only baseline. The recall gain is particularly important because missing prerequisites directly leads to invalid or uncompilable configurations. These diagnostics underpin the closed-loop tuning experiments by enabling traceable failure attribution under drift.

C. Feedback-driven Tuning Evaluation

We evaluate ConfigSwarm on end-to-end workload tuning under a fixed validation budget, and compare against three baselines: (i) the default `defconfig`, (ii) AutoOS as a vanilla

TABLE II: **End-to-end optimization performance.** Results are averaged over 3 independent runs with a strict budget of 5 iterations per run. We report both the absolute values and the relative improvement compared to the Default baseline. **BSR** denotes Build Success Rate. Note that for MySQL Latency, a lower value indicates better performance.

Method	**BSR**	**Redis Throughput (↑)**		**MySQL P95 Latency (↓)**		**UnixBench Index (↑)**	
	(Validity)	**RPS (req/s)**	**Improv.**	**ms**	**Reduction**	**Score**	**Improv.**
Default (`defconfig`)	100%	26,852.9	—	62.47	—	448.6	—
AutoOS	26.7%	27,390.6	+2.0%	60.58	-3.0%	459.8	+2.5%
KG-RAG (Static)	46.7%	28,336.6	+5.5%	56.84	-9.0%	467.9	+4.3%
ConfigSwarm	**73.3%**	**29,685.4**	**+10.5%**	**55.82**	**-10.6%**	**472.7**	**+5.4%**

TABLE III: **Evolutionary adaptation efficiency across kernel versions.** Comparison between the baseline (Full Rebuild) and ConfigSwarm (Backbone+Diff) when migrating from Linux 6.1. Token usage is measured in Millions (M).

Target Ver.	**Drift Complexity ($\Delta^{(v)}$)**				**Adaptation Time (min)**		**Token Cost (M)**		**Cost Saving**
	Added	**Rmvd**	**Mod**	**Total Δ**	**Full Rebuild**	**ConfigSwarm**	**Full Rebuild**	**ConfigSwarm**	
Linux 6.2	165	44	450	659	516.5	**18.9** (27×)	134.6	**4.9** (27×)	**96.3%**
Linux 6.3	352	224	919	1,495	508.3	**38.3** (13×)	134.5	**10.1** (13×)	**92.5%**
Linux 6.4	541	280	1,156	1,977	540.6	**54.0** (10×)	135.8	**13.6** (10×)	**90.0%**
Linux 6.5	705	296	1,600	2,601	579.6	**78.0** (7.4×)	136.8	**18.4** (7.4×)	**86.5%**

LLM-based tuner, and (iii) KG-RAG (Static), which retrieves relevant options from VKConfig-KG to guide configuration edits, but applies no iterative feedback-driven refinement based on validation outcomes. All methods share the same computational budget of 5 compilation attempts per run and results are averaged over 3 independent runs.

Table II reports end-to-end tuning results under a fixed budget. ConfigSwarm attains the highest build success rate (73.3%), markedly higher than the LLM-only and static KG-RAG baselines, indicating that validation-in-the-loop substantially improves deployability.

Among the configurations that pass validation, ConfigSwarm also achieves the best overall gains across workloads: +10.5% Redis throughput, 10.6% reduction in MySQL P95 latency, and +5.4% UnixBench improvement over `defconfig`. These results show that executable feedback can improve both validity and performance within the same iteration budget.

Overall, the results demonstrate that ConfigSwarm's feedback-driven loop improves both *deployability* (higher BSR) and *effectiveness* (larger performance gains) compared to one-shot LLM tuning and static retrieval-based baselines, under the same validation budget.

D. Version Drift Adaptation (Backbone–Diff)

We evaluate how efficiently ConfigSwarm adapts to kernel evolution when migrating from the anchor release (Linux 6.1) to later versions. The baseline (*Full Rebuild*) re-processes each target release end-to-end, including full `Kconfig` parsing, constraint extraction, and semantic alignment to the taxonomy. In contrast, ConfigSwarm reuses the 6.1 backbone and applies the same pipeline only to the drift delta $\Delta^{(v)}$ via backbone–diff updates.

TABLE IV: **Ablation study. w/o Feedback Loop** removes validation-driven iterative write-back and refinement but keeps one-pass validation. **w/o KG Constraints** removes SAT-based dependency/conflict checks in VKConfig-KG, reducing the system to text-based retrieval.

Variant	**Safety Metric**		**Optimization Metric**	
	BSR	**Δ vs Full**	**Redis Gain**	**Δ vs Full**
ConfigSwarm (Full)	**73.3%**	—	**+10.5%**	—
w/o Feedback Loop	53.3%	-20.0%	+6.5%	-4.0%
w/o KG Constraints	40.0%	-33.3%	+3.0%	-7.5%

Table III shows that backbone–diff adaptation is consistently cheaper than full rebuild across Linux 6.2–6.5. Despite increasing drift complexity (from 659 to 2,601 changed items), ConfigSwarm reduces adaptation time from 8–10 hours to 18.9–78.0 minutes, yielding a $7.4\times$–$27\times$ speedup. Token usage exhibits the same trend, dropping from $\sim$135M tokens per version under full rebuild to 4.9–18.4M under ConfigSwarm, corresponding to 86.5%–96.3% cost savings. Overall, these results support the proposed state factorization, $\mathcal{G}^{(v)} = \mathcal{G}_{\text{base}} \oplus \Delta^{(v)}$, which enables incremental maintenance under version drift by concentrating computation on the changed subset rather than re-processing the full kernel space.

E. Ablation Study

Table IV quantifies the contribution of (i) the feedback loop (write-back) and (ii) KG-level constraint checking to both *safety* and *optimization*.

a) w/o Feedback Loop: disables iterative write-back and candidate refinement, while retaining one-pass validation, including symbolic feasibility checks: BSR drops from 73.3% to 53.3% (-20.0%), and the Redis gain decreases from +10.5% to +6.5% (-4.0%). This confirms that validation is not only

a filter but also a learning signal—without feeding failure attribution and outcomes back to VKConfig-KG, the alignment stage cannot reliably avoid previously observed conflicts and tends to waste the limited budget on repeated or unproductive edits.

b) w/o KG Constraints: Removing SAT-based dependency/conflict checks causes the largest safety regression (BSR 73.3% $\rightarrow$ 40.0%, -33.3%) and also weakens optimization (+10.5% $\rightarrow$ +3.0%, -7.5%). This indicates that purely semantic retrieval, even when concept-grounded, is insufficient for kernel configuration editing: executable constraints are essential to prevent plausible-but-invalid changes and to keep the search within the feasible region.

Overall, both components are necessary: constraint checks provide the feasibility boundary, while feedback-driven write-back improves sample efficiency under strict iteration budgets.

IV. Related work

a) Kernel Configuration and Constraint-Aware Analysis: A kernel configuration is executable only if it satisfies the formal semantics of the `Kconfig` language, including dependency conditions (`depends on`), implications introduced by `select`, and `choice` constraints [5]. Accordingly, prior work has studied diverse analyses of the Linux configuration model, including variability anomalies and failure-inducing options [11], [12], and has developed scalable techniques to obtain diverse valid configurations (e.g., pragmatic sampling) for testing and diagnosis [1], [3]. More recently, LLMs have been explored for natural-language-driven configuration assistance and tuning (e.g., AutoOS [4]), but reliability can degrade when suggested edits are not explicitly checked against `Kconfig` constraints or when cross-version migration is handled via repeated full rebuilds.

b) Retrieval-Augmented and Agentic Frameworks: Reliability of LLM-based systems is often improved through evidence-grounded generation and agentic interaction with external tools. RAG augments generation by conditioning on retrieved evidence from an indexed corpus [13]. Agentic approaches further interleave reasoning and acting, enabling models to invoke tools and execute multi-step procedures [14]–[16]. From the perspective of constrained decision making, these methods can be viewed as combining neural inference with external grounding mechanisms (retrievers, tools, or executors), but they are typically instantiated on static snapshots and do not explicitly model system evolution. ConfigSwarm extends this paradigm to kernel configuration by combining a version-aware symbolic substrate with validation-in-the-loop execution, allowing executable outcomes to be recorded and reused under version drift.

V. Conclusion

In this paper, we propose **ConfigSwarm**, a multi-agent framework for kernel configuration QA and tuning under version drift. ConfigSwarm grounds LLM-based intent understanding in a shared, version-scoped substrate (VKConfig-KG) and enforces feasibility through executable validation. By factoring configuration knowledge into an anchor backbone and sparse diffs, the system localizes cross-version maintenance and reasoning to the changed subset rather than re-processing each release from scratch. On top of this substrate, *Interaction*, *Parsing*, *Alignment*, and *Validation agents* coordinate via typed artifacts and staged checks, and write validation outcomes back as structured updates that can be reused across iterations and versions. Experiments show that ConfigSwarm improves the reliability of graph-grounded diagnostic QA, increases tuning validity and workload performance under a fixed verification budget, and substantially reduces cross-version adaptation cost. Future work will expand the taxonomy and mapping supervision, incorporate richer runtime signals (e.g., tracing and energy), and study transfer across architectures and workloads.

Acknowledgment

This work was supported by the Code Semantic Understanding and Intelligent Generation project under Grant No. E0YD310101. AI tools were used only for language refinement. The authors take full responsibility for the content of this paper.

References

[1] M. Acher, "Learning the linux kernel configuration space: Results and challenges," in *ELC Europe 2019-Embedded Linux Conference Europe 2019*, 2019, pp. 1–49.
[2] L. Torvalds and contributors, "Linux kernel source tree," 2026.
[3] D. Fernandez-Amoros, R. Heradio, J. M. Horcas Aguilera, J. A. Galindo, D. Benavides, and L. Fuentes, "Pragmatic random sampling of the linux kernel: Enhancing the randomness and correctness of the conf tool," in *Proceedings of the 28th ACM International Systems and Software Product Line Conference*, 2024, pp. 24–35.
[4] H. Chen, Y. Wen, L. Cheng, S. Kuang, Y. Liu, W. Li, L. Li, R. Zhang, X. Song, W. Li *et al.*, "Autoos: make your os more powerful by exploiting large language models," in *Forty-first International Conference on Machine Learning*, 2024.
[5] *Kconfig Language*, Linux Kernel Documentation, 2026.
[6] *Neo4j Graph Database*, Neo4j, Inc., 2024.
[7] *redis-benchmark*, Redis Ltd., 2021.
[8] A. Kopytov, *sysbench*, 2020.
[9] *UnixBench*, UnixBench Maintainers, 2011.
[10] OpenAI, "Gpt-5-mini model documentation," 2025. [Online]. Available: https://platform.openai.com/docs/models/gpt-5-mini
[11] J. Mortara and P. Collet, "Capturing the diversity of analyses on the linux kernel variability," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*, 2021.
[12] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 111–120.
[13] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.
[14] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.
[15] 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.
[16] 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.