Foundation AI
By CISCO

Antares: Foundation Models for Agentic Vulnerability Localization

Supriti Vijay[*,1], Aman Priyanshu[*,1], Didier Chapoteau[1], Arthur Goldblatt[1], Jianliang He[1,2,†], Kimia Majd[1], Fraser Burch[1], Baturay Saglam[1,2,†], Takahiro Matsumoto[1], Zhuoran Yang[1,2,†], Amin Karbasi[1]

Abstract

Vulnerability localization is a fundamental step in software security, requiring models to reason over large codebases and iteratively identify vulnerable implementations. We present **Antares**, a family of compact language models (350M, 1B, and 3B parameters) for agentic vulnerability localization. Based on IBM Granite base models, Antares is trained through a two-stage pipeline that combines supervised fine-tuning on cybersecurity reasoning and repository exploration data with reinforcement learning from verifiable rewards over vulnerable repositories. Across extensive evaluations, Antares-3B approaches GPT-5.5 while outperforming open-weight models over 200× larger in size. The Antares family further enables fast, low-cost local inference, completing a full 500-task evaluation sweep in approximately 15 minutes on a single H100 GPU, corresponding to an amortized evaluation time of under 2 seconds and less than $0.002 per task. We publicly release Antares-350M and Antares-1B at https://huggingface.co/collections/fdtn-ai/antares.

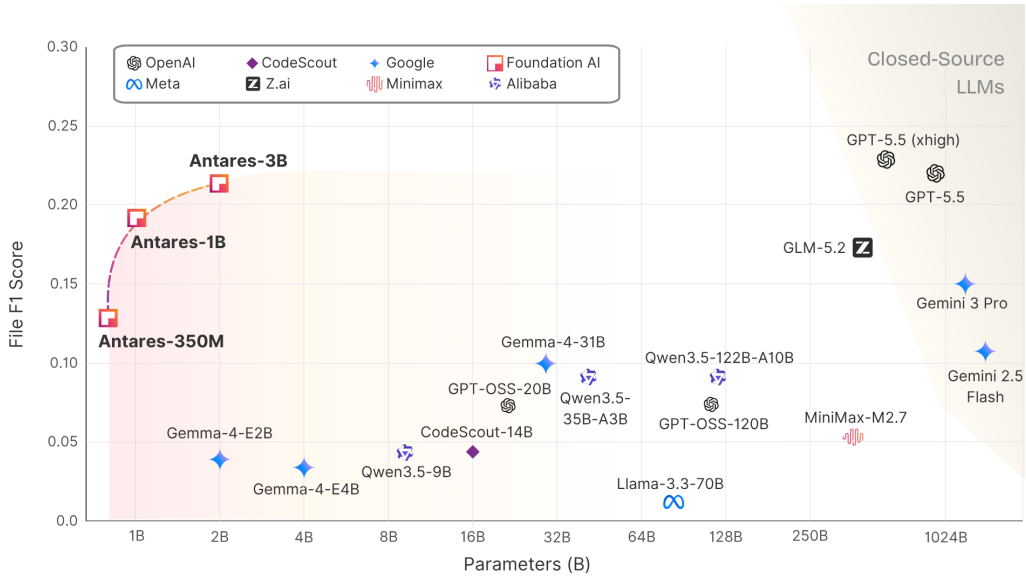


Figure 1 | F1 score versus model size on Vulnerability Localization Benchmark (VLoc Bench) [7], a repository-scale benchmark comprising 500 tasks across 290 unique real-world repositories, where models receive only a CWE category description and must identify vulnerable implementation files in real codebases. Antares models form the Pareto frontier among evaluated models, achieving the strongest localization quality at small parameter scales. Antares-3B reaches near-frontier closed-source performance while remaining orders of magnitude smaller than GPT-5.5 and Gemini-family baselines.

[1]Foundation AI–Cisco Systems Inc. [2]Yale University [†]Work done while at Foundation AI

[*]Equal Contribution. Corresponding authors: {suprivij,ampriyan}@cisco.com

Contents

1. Introduction

Modern software repositories are too large, modular, and dependency-rich for vulnerability remediation to begin with manual inspection alone [36]. Once a vulnerability is disclosed, the first operational question is not whether the weakness exists in the abstract, but where the vulnerable implementation lives. Accurately localizing that code is the step that enables patching, triage, regression testing, and downstream security review [36]. Yet repository-scale vulnerability localization remains difficult because the relevant evidence is rarely contained in a single function or file. It is distributed across imports, call paths, framework conventions, configuration boundaries, and implementation-specific idioms [14, 55].

Human security researchers solve this problem interactively. They do not read an entire repository from top to bottom. They form hypotheses from the vulnerability class, search for likely entry points, inspect candidate files, follow call chains, compare naming conventions, and revise their search as new evidence appears. Vulnerability localization is therefore not simply a static code understanding task; it is an agentic reasoning problem over a live software environment.

Existing approaches only partially address this setting. Static analysis tools such as CodeQL [12], SonarQube [44], and Semgrep [38] provide scalable rule-based detection, but their effectiveness is limited by predefined patterns and the coverage of their analysis front ends. Recent project-level and agentic vulnerability detection systems move beyond isolated function classification but often fall into one of two regimes: either they rely on static-analysis front ends to surface candidate locations, inheriting the recall limits of those tools, or they wrap frozen frontier models in search scaffolds, incurring high cost without owning the underlying localization policy [6, 27, 29, 52, 58, 60]. In both cases, the system does not learn end-to-end how to search a repository from a bare vulnerability description.

We introduce **Antares**, a family of compact language models trained specifically for agentic vulnerability localization. Given only a CWE category description and read-only terminal access to a repository, Antares autonomously searches the codebase, inspects files, gathers evidence, and submits the vulnerable implementation paths. Unlike systems that depend on pre-extracted context, SAST-generated candidates, crash traces, or external frontier APIs, Antares performs localization end-to-end from the repository itself.

Antares consists of 350M, 1B, and 3B parameter models initialized from IBM Granite checkpoints and post-trained through a two-stage pipeline. Supervised fine-tuning teaches cybersecurity reasoning, repository exploration, and terminal interaction. Reinforcement learning then optimizes complete multi-turn trajectories using verifiable file-level localization rewards. This training setup directly rewards the behavior required at deployment time: searching strategically, verifying candidates, and submitting vulnerable files under a fixed terminal budget.

We evaluate Antares on Vulnerability Localization Benchmark (VLoc Bench) [7], a repository-scale benchmark comprising 500 tasks drawn from 290 unique real-world vulnerable repositories. All models are evaluated under the same constrained agent protocol: read-only Docker sandbox, no network access, a fixed terminal-command budget, and only the CWE category description as input. This setting tests whether a model can act like a security localization agent rather than merely classify a preselected code snippet. To test whether this policy transfers beyond security, we additionally evaluate general issue-driven code localization in Appendix B. There, Antares-3B remains competitive with dedicated CodeScout models trained specifically for SWE-Bench localization, approaching the 4B CodeScout baseline despite being trained for vulnerability localization rather than issue-resolution file localization.

Our results, summarized in Figure 1, show that targeted post-training can matter more than raw model scale. Antares-3B reaches 0.223 File F1, approaching GPT-5.5 while outperforming substantially larger open-weight models, including GLM-5.2. Antares-1B achieves the highest recall among all evalu-

ated systems, and even Antares-350M outperforms several larger general-purpose baselines. Behavioral analysis further shows that reinforcement learning induces a search–verify–refine strategy rather than generic repository browsing, while reducing run-to-run variance across model scales.

Finally, Antares is designed for deployment constraints that matter in security workflows. We expose Antares through a local CLI for file-level vulnerability localization. The deployment preserves the benchmark's CWE-conditioned task, default inspection budget, and ranked file-submission protocol while adding repository isolation and machine-readable reporting. When inference is hosted within the user's trust boundary, proprietary source code need not be sent to third-party APIs. In our evaluation setup, the full 500-task VLoc Bench evaluation completed in approximately 15 minutes on a single H100 GPU, corresponding to under two seconds and less than $0.002 per task. The CLI provides human-readable and structured outputs, including SARIF for code-scanning integration, and supports CI/CD, security triage, and closed-network or air-gapped deployment when the model weights and inference endpoint are hosted locally.

2. Related Work

2.1. Static Analysis and ML-Based Vulnerability Detection

Traditional vulnerability detection relies on static analysis tools such as CodeQL [12], SonarQube [44], and Semgrep [38] that identify security flaws through predefined patterns and dataflow rules. While these tools have become standard in development workflows, comprehensive evaluations consistently expose fundamental limitations including susceptibility to evasion techniques, inability to reason about novel vulnerability classes, and poor adaptation to unfamiliar codebases [2]. Machine learning approaches, from graph neural networks to large language models fine-tuned for vulnerability detection, improve recall on benchmark datasets but exhibit significant precision instability across projects and remain constrained to single-pass analysis over fixed code snippets [51]. Recent project-level and agentic vulnerability detection systems move beyond isolated functions, but often rely on static-analysis front ends, pre-extracted contexts, crash traces, or frozen frontier models rather than end-to-end learned localization from a bare repository [6, 27, 29, 52, 58, 60]. These limitations are particularly acute in repository-scale settings where vulnerability context spans multiple files and successful localization requires iterative navigation rather than one-shot classification.

2.2. Security-Specialized Language Models

Rather than improving detection tools directly, an alternative line of work trains language models to internalize security knowledge. Lily-Cybersecurity-7B [37], DeepHat-V1-7B [10], Foundation-Sec-8B [23, 59], and Primus [64] demonstrate that continued pretraining or fine-tuning on security corpora covering vulnerability assessment, threat intelligence, and penetration testing can effectively transfer domain knowledge to models ranging from 7B to 13B parameters. Foundation-Sec-8B-Reasoning [62] extends this line through GRPO with verifiable rewards, producing the first open-source reasoning model for cybersecurity. These models excel at classification and structured reasoning but cannot act on their knowledge within real software environments, lacking the ability to navigate repositories, execute terminal commands, or iteratively explore codebases to locate vulnerable implementations.

2.3. Agentic Information Retrieval and Code Localization

Classical dense retrieval systems [22, 54] optimize a single query and assume the right evidence is surfaceable in one retrieval pass. Reasoning-aware retrievers [9, 40] and reinforced query rewriting systems [35] improve intent alignment by conditioning on chain-of-thought traces or learning refor-

mulations against retrieval feedback, but remain single-turn in their search strategy. Interactive and agentic retrieval methods [19, 21, 25, 66] address this by training LLMs to issue tool calls, read results, and iteratively refine queries. Think Before You Retrieve [53] further demonstrated that compact models (350M to 1.2B) can learn dynamic multi-turn retrieval strategies through turn-level GRPO rewards, outperforming larger specialized systems despite being 200 to 400 times smaller.

The proliferation of LLM-powered coding agents [3, 4, 31] has driven demand for dedicated code localization and context-building capabilities within these systems. SWE-grep [33] and Composer [8] demonstrate that reinforcement learning produces emergent efficiency behaviors including parallel tool calls and search-heavy exploration strategies that match frontier retrieval accuracy at substantially higher throughput. CodeScout [46] formalizes this direction by training code search agents purely with GRPO on SWE-Bench [20] using file-level F1 as the reward signal across scales from 1.7B to 14B parameters, while FastContext [65] trains exploration subagents that return compact file-and-line citations to a main solving agent. Rather than targeting general software engineering, we extend this paradigm to terminal-based repository exploration specialized for vulnerability localization.

2.4. Reinforcement Learning for Agentic Models

Several concurrent works advance reinforcement learning for agentic language models at frontier scale. DeepSeek-R1 [13] pioneered GRPO [41] for reasoning, demonstrating that group-relative advantages with binary rewards can train explicit reasoning traces without learned reward models. The Qwen3 series [61] established a multi-stage recipe of SFT cold start followed by GRPO and strong-to-weak distillation. Large-scale agentic RL has since been applied in Kimi K2 [49], GLM-4.5 [47], and MiniMax-M2 [28], each training models with hundreds of billions of parameters on agentic trajectories from real and synthetic environments. We adopt the same GRPO formulation and two-stage approach but apply it to multi-turn agent trajectories in the security domain at compact model scales rather than frontier-scale general reasoning.

2.5. Agentic Cybersecurity Benchmarks

Existing agentic benchmarks evaluate either general software engineering or offensive security capabilities but not repository-scale vulnerability localization. SWE-Bench [20], SWE-Bench Verified [30], and SWE-Bench Pro [11] test issue resolution across Python repositories without requiring security-specific reasoning. τ-bench [63] and τ^2-bench [5] extend agentic evaluation to tool-agent-user interaction and dual-control environments, respectively, testing policy adherence without security-specific reasoning. NYU CTF Bench [39], CyberGym [57], and ExploitGym [56] evaluate offensive capabilities assuming the vulnerability location is already known, while CTIBench [1] tests security knowledge through classification tasks without any agentic interaction. This gap motivates VLoc Bench [7], a 500-task benchmark requiring models to simultaneously navigate unfamiliar codebases efficiently and recognize vulnerability patterns associated with specific CWE categories, a combination none of the above benchmarks evaluate.

3. The Antares Family

Antares is a family of compact language models designed for repository-scale vulnerability localization. Unlike conventional code generation models, Antares is trained to operate as an interactive software security agent capable of reasoning over complete repositories. Given a CWE-ID and its generic category description alongside read-only access to a software repository through a constrained terminal interface, the model autonomously explores the repository, gathers evidence, and identifies the source files containing the vulnerability.

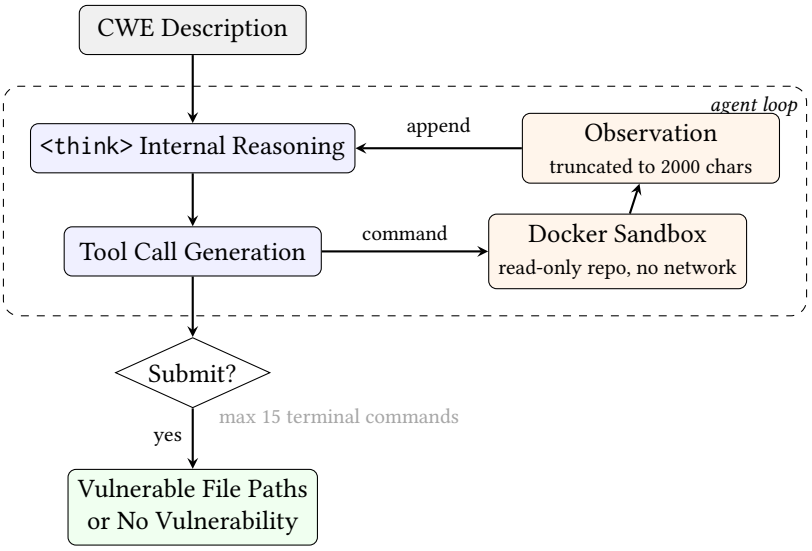


Figure 2 | Antares inference loop for a single evaluation task. Given only a CWE category description, the model iteratively reasons, issues read-only terminal commands against the repository sandbox, and incorporates observations until it identifies vulnerable files or exhausts its turn budget.

The Antares agentic system consists of two tightly integrated components: a cybersecurity-specialized language model and an execution environment that exposes a restricted set of terminal operations. During inference, the model interacts with the repository through standard command-line utilities, including repository search, file inspection, and directory navigation. This interaction enables Antares to incrementally construct an understanding of unfamiliar codebases instead of relying solely on a fixed context window. To support deployment across diverse computational environments, Antares is developed at three model scales (350M, 1B, and 3B parameters) that share a common architecture and post-training pipeline.

Rather than specializing solely for code generation or cybersecurity question answering, Antares is optimized for the complete vulnerability localization workflow. This includes interpreting CWE category descriptions, planning repository exploration strategies, identifying relevant implementation components, synthesizing evidence across multiple files, and producing the final localization prediction. Throughout this report, we evaluate how these capabilities emerge through supervised fine-tuning and reinforcement learning over interactive terminal trajectories.

3.1. Model Family

Antares consists of three decoder-only transformer models containing 350M, 1B, and 3B parameters. All models are initialized from IBM Granite 4.0 checkpoints [15, 16, 17, 45] and share the same tokenizer and architectural blueprint: grouped-query attention, SwiGLU MLP activations, RMSNorm, RoPE positional embeddings, and shared input/output embedding matrices. The three variants differ only in layer count, hidden dimension, and attention head configuration. Scaling the model family allows us to evaluate how repository exploration and vulnerability localization capabilities evolve with model capacity while maintaining a fixed training pipeline.

3.2. Agentic Execution Environment

Antares is not evaluated as a standalone sequence model. It operates inside a constrained agent loop that exposes a small set of read-only tools for repository exploration and final submission. Given a

Table 1 | Antares model variants with base checkpoint, maximum context length, and intended deployment tier. All three models target local, low-resource inference without requiring datacenter hardware.

Model	Parameters	Base Model	Context	Intended Deployment
Antares-350M	350M	Granite 4.0 350M	32K	Mobile / IoT
Antares-1B	1B	Granite 4.0 1B	128K	Laptop / Workstation
Antares-3B	3B	Granite 4.0 Micro	128K	Laptop / Any Single GPU

CWE category description, the model iteratively reasons, issues terminal commands, observes truncated command outputs, and either submits vulnerable file paths or declares that no vulnerability is present. The environment restricts interaction to read-only repository navigation and inspection, disables network access, and enforces a fixed terminal-command budget. These constraints make localization a controlled agentic task: the model must decide what to search, which files to inspect, when to refine its hypothesis, and when to submit.

This execution environment serves two roles. During training, it provides the multi-turn trajectories on which GRPO rewards are computed. During evaluation, it ensures that all models are compared under the same repository access, tool budget, and submission protocol. The Antares CLI described in Section 6 packages the Antares agent protocol for practical deployment. It preserves the benchmark's CWE-conditioned task, default repository-tool budget, and file-level submission semantics while adding production-specific repository isolation and integration features.

3.3. Why Small Models?

Cybersecurity workflows place different constraints on models than general coding benchmarks. In many real deployment settings, security tools need to run close to the codebase, integrate into existing developer and security workflows, and operate under strict privacy, latency, and cost constraints. This is especially important for vulnerability localization, where the model may need to inspect proprietary repositories and interact repeatedly with the environment before producing a result. The security domain further demands that inference remain in-house: sending source code to external APIs introduces supply-chain risk and is often prohibited by enterprise security policies.

Rather than relying exclusively on frontier-scale models, effective cybersecurity agents benefit from high tokens-per-second throughput at low cost. Multi-turn agentic loops amplify latency and cost linearly with conversation depth, making compact models that sustain high generation speed on commodity hardware particularly attractive for this setting. Antares-3B generates over 1,500 tokens per second on a single GPU, completing a full 500-task evaluation sweep spanning 290 repositories in ~15 minutes. At average cloud rental rates ($2–$4 per H100-hour across commodity GPU providers), this translates to under one dollar per complete evaluation run, enabling repeated repository-scale evaluation at low marginal cost. Comparable frontier API usage exceeds one hundred dollars for the same workload.

We therefore design Antares around this assumption: a compact, specialized model trained directly for terminal-based vulnerability localization can deliver competitive accuracy while remaining practical for integration into CI/CD pipelines, code review systems, and security triage workflows that require fully local, closed-network operation.

4. Training Data

Training Antares requires data that spans two complementary competencies, namely understanding vulnerability patterns and navigating unfamiliar codebases through terminal interaction. We construct separate datasets for supervised fine-tuning and reinforcement learning, reflecting the distinct objectives of each training stage. The SFT corpus establishes broad security knowledge, deep research capability, and terminal fluency, while the RL dataset supplies verifiable localization tasks constructed through proprietary curation pipelines.

Table 2 | Training data composition across SFT and RL stages.

Component	Fraction	Purpose
SFT Corpus		
Cybersecurity Reasoning	71.5%	Teaches vulnerability concepts, CWE/CVE reasoning, advisory interpretation, and security analysis.
Deep Research & General	13.1%	Preserves broad multi-step reasoning, evidence aggregation, and general instruction-following behavior.
Code Search Trajectories	15.4%	Teaches terminal-based repository exploration, file inspection, and iterative code search.
RL Corpus		
Repository Localization Tasks	—	Provides verifiable end-to-end vulnerability localization tasks over repository snapshots curated through proprietary pipelines.

4.1. Security and Deep Research Corpus

The supervised fine-tuning dataset is organized into three categories that together cover the skills required for vulnerability localization. Following the methodology established in Foundation-Sec-8B-Reasoning [62], SFT builds a broad foundation before reinforcement learning specializes the policy.

Cybersecurity reasoning and deep research data together use GPT-OSS-120B [32] as the unified teacher model. This single-teacher design is deliberate: cross-domain teacher mixing has been shown to introduce distribution shifts and higher output entropy compared to single-teacher baselines [48], so we source all reasoning traces from one model to maintain consistent reasoning style across the entire non-terminal corpus. The security portion covers CWE taxonomy, CVE-to-CWE mappings, threat modeling, and advisory interpretation. The deep research component follows the long-horizon trajectory synthesis methodology of OpenResearcher [26], comprising multi-turn web search and evidence aggregation workflows that maintain broad multi-step reasoning capabilities.

4.2. Terminal Trajectories

Code search trajectories constitute the remaining 15% of the SFT corpus. Each trace captures a complete tool-use conversation in which a model navigates a repository to locate specific files given natural-language descriptions. Traces follow the full interaction protocol consisting of system prompt, reasoning, terminal command, observation, and answer. Trajectories span Java, JavaScript, Go, Swift, Python, and

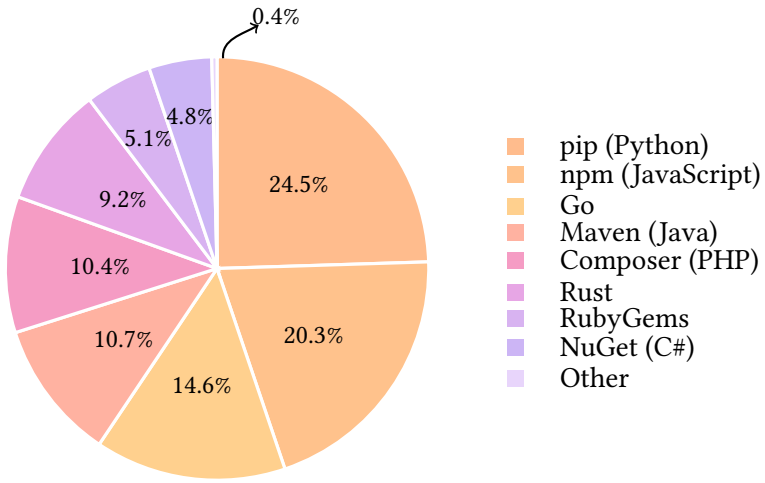


Figure 3 | Ecosystem distribution of the RL training corpus. The dataset spans nine package ecosystems, with Python and JavaScript constituting the largest shares.

additional ecosystems.

This component is critical because it teaches the model how to explore repositories using only a terminal. Rather than relying on static retrieval, the model learns to issue appropriate commands, interpret directory structures, follow import chains, and refine searches based on intermediate observations. GRPO later specializes this general search capability toward vulnerability localization specifically.

All SFT data uses a unified message format with explicit reasoning traces wrapped in thinking tags. The cybersecurity reasoning component provides what to look for and the code search component teaches how to look, but these two capabilities remain disconnected until the reinforcement learning stage bridges them by optimizing for finding security vulnerabilities through terminal-based repository exploration.

4.3. Reinforcement Learning Dataset

The reinforcement learning dataset consists of vulnerable repository snapshots curated through proprietary data-generation and filtering pipelines. Each task provides a complete codebase containing a known vulnerability alongside ground-truth file labels identifying the vulnerable source files.

The dataset covers 9 software ecosystems (pip, npm, Go, Maven, Composer, Rust, RubyGems, NuGet, and Others) spanning 255 unique CWE categories. During GRPO rollouts, each repository snapshot is extracted at runtime into an isolated Docker sandbox, providing the model with a realistic exploration environment identical in structure to production codebases.

Training and evaluation datasets are strictly disjoint. Throughout training, we verify that the GRPO training corpus has no overlap with VLoc Bench, ensuring that performance reflects genuine generalization rather than memorization of specific repositories or vulnerability patterns.

4.4. Data Filtering

Ground-truth labels identify the implementation files containing each vulnerability. Test files, documentation, and configuration are excluded from the label set, ensuring that the reward signal during GRPO training reflects localization of vulnerable code rather than identification of ancillary changes.

5. Training Pipeline

Antares is trained using a two-stage post-training pipeline designed to progressively develop cybersecurity knowledge, terminal interaction skills, and repository exploration strategies. Rather than optimizing all capabilities simultaneously, each stage targets a distinct aspect of the vulnerability localization problem. The first stage (supervised fine-tuning) establishes format compliance, security domain knowledge, and basic terminal fluency. The second stage (reinforcement learning) optimizes the policy for end-to-end vulnerability localization over multi-turn agent trajectories.

5.1. Supervised Fine-Tuning

5.1.1. Objective

The SFT stage transforms each Granite 4.0 base model from a general-purpose language model into a terminal-capable security reasoning agent. The base models achieve near-zero File F1, on our evaluation benchmark prior to fine-tuning. While they possess tool-calling capabilities, they have no notion of structured terminal interaction and produce degenerate outputs when placed in an agentic loop. SFT addresses this by teaching three capabilities simultaneously: cybersecurity domain knowledge, structured deep-research behavior, and terminal interaction protocols.

5.1.2. Training Procedure

We fine-tune each model on the full SFT corpus for one epoch using AdamW (β_1=0.9, β_2=0.999), a learning rate of 5×10^{-5} with cosine decay, and a global batch size matched to 8×H100 throughput. Training completes on a single 8×H100 node. All three model sizes (350M, 1B, 3B) use the same data mix and hyperparameters with no per-size tuning applied at this stage.

5.1.3. Auxiliary Objectives

Standard SFT trains the model to imitate assistant actions, including reasoning traces and tool calls, but it provides no direct supervision on how the model should represent environment observations. In our setting, these observations are central to the task: the model must interpret directory listings, search results, and file contents before deciding which command to issue next. Without an auxiliary signal, this grounding is learned only indirectly from the relationship between observations and subsequent actions, which is particularly challenging for smaller models.

We therefore incorporate auxiliary supervision during SFT to improve the model's representation of terminal feedback. The goal is to make environment observations useful for downstream repository navigation, while avoiding objectives that overfit to repository-specific surface forms.

A natural approach is token-level observation prediction. Objectives such as ECHO [43] add auxiliary cross-entropy loss on raw environment observation tokens, giving the model a direct learning signal on terminal outputs. This objective is well motivated: predicting observations can help the model learn how commands affect the environment and how terminal feedback should inform future actions. However, terminal outputs in repository exploration are highly instance-specific. Directory listings, file paths, and grep results vary substantially across codebases, so exact token prediction can emphasize surface-level reconstruction rather than transferable understanding of terminal feedback. We include this objective as an ablation in Section 8.

Antares instead uses semantic conditioning as its SFT auxiliary objective. Rather than predicting the exact tokens in an observation, semantic conditioning operates at the representation level. It encourages the model to learn similar internal representations for observations with similar functional roles, such

as directory listings, search outputs, and source-code snippets, even when their surface tokens differ across repositories. This follows recent work on latent-space objectives for language models [18, 50], which suggests that representation-level supervision can provide a more transferable learning signal when surface forms are highly variable.

As shown in Table 7, semantic conditioning provides the strongest SFT initialization across all Antares model sizes, outperforming both standard SFT and token-level observation prediction. This stage establishes format compliance, security knowledge, terminal interaction, and observation grounding. Reinforcement learning is then used to optimize complete trajectories, converting this grounded terminal capability into a focused search–verify–refine policy for vulnerability localization.

5.2. Reinforcement Learning

5.2.1. Objective

The RL stage applies Group Relative Policy Optimization (GRPO) [41] to optimize the SFT policy for end-to-end vulnerability localization. Rather than using a learned reward model, we employ verifiable multi-component rewards computed programmatically from each trajectory. The objective is to transform unfocused terminal exploration into targeted, strategic vulnerability search.

5.2.2. Environment

During each GRPO rollout, a repository archive is extracted into an isolated, Docker-backed workspace. The model interacts with the repository through the same constrained interface used at evaluation time, with access limited to read-only file-system navigation and inspection. Network access and package installation are disabled, requiring the model to rely entirely on the repository contents available within the workspace. Terminal command outputs are truncated to 2,000 characters before being appended to the trajectory, bounding context growth and preventing unusually long outputs from dominating subsequent interactions.

5.2.3. Agent Loop

Each rollout follows a fixed interaction protocol: the model receives a system prompt containing a CWE category description, then iteratively generates reasoning (wrapped in thinking tags), issues tool calls, receives observations, and continues until it submits a localization prediction or exhausts its turn budget. Three tools are available: `terminal` (read-only repository navigation and inspection commands), `submit_vulnerable_files` (ranked file paths), and `submit_no_vulnerability_found` (declare clean). The training budget allows up to 15 assistant turns and 15 observation turns per rollout, followed by one final submission action. Rollout generation uses temperature 0.7, a maximum response length of 4,096 tokens, and a maximum model context of 16,384 tokens.

5.2.4. GRPO Algorithm and Reward Function

For each prompt, we sample multiple complete multi-turn trajectories, compute relative advantages from verifiable reward components, and update the policy using a clipped GRPO objective over assistant action tokens only.

We use a multi-component verifiable reward combining localization quality, valid submission behavior, tool-use compliance, exploration behavior, and penalties for malformed file predictions. All reward components are computed programmatically from trajectory text, with no learned reward model.

These components provide denser feedback than a binary success signal while preserving verifiability.

Table 3 | GRPO reward components. The reward combines verifiable signals for localization accuracy, valid task completion, tool-use behavior, and malformed-output avoidance. All components are computed programmatically from trajectory text with no learned reward model.

Component	Purpose
Localization quality	Measures agreement between submitted file paths and ground-truth vulnerable files.
Submission behavior	Encourages the model to complete the task through the appropriate submission tools rather than failing to submit or prematurely declaring no vulnerability.
Tool-use compliance	Rewards valid interaction with the agent loop and discourages malformed tool calls.
Exploration behavior	Encourages the model to gather evidence from the repository before making a final localization prediction.
Malformed-output penalty	Penalizes invalid or hallucinated file predictions that cannot be resolved cleanly through the structured submission interface.

Early in training, submission and tool-use signals help stabilize the agent loop and encourage consistent task completion. As the policy begins to submit valid predictions more reliably, localization quality becomes the primary signal distinguishing higher- and lower-quality trajectories within each group. Unlike Foundation-Sec-8B-Reasoning [62], which required an explicit format penalty to prevent reward hacking, our agent loop naturally constrains output format because the model must produce valid tool calls to receive observations.

5.2.5. Infrastructure and Training Configuration

We use veRL [42] as the training framework with vLLM as the inference backend. veRL was selected because its multi-turn rollout pipeline supports custom tool-call parser registration, enabling integration with Granite's tool-calling format for mid-trajectory generation and conversation management. Custom patches were required for Granite tool-call parser registration, domain-specific agent loop integration, and reward component logging.

Training uses a single 8× NVIDIA H100 80GB node. The actor model is distributed across GPUs with FSDP, while rollout inference uses vLLM with fixed rollout budgets across model scales. We use low-learning-rate GRPO with KL regularization against the SFT reference policy, small rollout groups, and optimizer states offloaded to CPU.

5.3. Discussion

The most visible effect of GRPO is reduced performance variance across rollouts. The SFT policy exhibits high stochasticity with inconsistent trajectories across different repositories. After GRPO, the policy produces stable and repeatable search strategies, transitioning from unfocused exploration to targeted vulnerability search given a particular CWE category and repository structure. Quantitative results across model scales are presented in Section 8.

6. Deployment: Antares CLI

The Antares CLI packages the agent protocol described in Section 5.2.3 as a deployment interface for file-level vulnerability localization. It preserves the core evaluation semantics: the model receives a CWE-conditioned prompt, explores the repository under a bounded inspection budget, and either submits file paths or declares that no matching vulnerability was found. The default 15-call inspection budget matches the evaluation configuration but can be adjusted at deployment. The production harness additionally operates over an immutable repository snapshot and provides a dedicated file-reading tool.

The CLI supports both targeted analyses over explicit CWE identifiers and repository-wide sweeps over user-specified or automatically selected CWE sets. For automatic selection, the repository is profiled against the bundled MITRE CWE taxonomy before independent investigations are launched in parallel. A local planning mode allows users to preview the selected categories and supporting evidence without invoking the model. The CLI produces both human-readable and structured reports, including SARIF 2.1.0 output for GitHub Code Scanning. An optional failure-on-findings policy supports CI gating without treating candidate detections as fatal by default. Outputs are file-level candidates intended for human validation; line-level localization and remediation remain outside the current system's scope.

A non-interactive JSON interface supports integration with coding assistants and orchestration frameworks by accepting structured requests and returning findings, summary statistics, metadata, and per-CWE results. This interface allows external agents to invoke Antares as a tool without requiring PTY or signal management. The CLI delegates model execution to a user-configured, streaming, OpenAI-compatible endpoint rather than hosting inference directly. The released Antares-350M and Antares-1B models can be served within the user's environment, enabling closed-network or air-gapped operation when both the model weights and inference endpoint are hosted locally.

7. Experimental Setting

7.1. Benchmark

We evaluate all models on VLoc Bench, a vulnerability-localization benchmark comprising 500 tasks drawn from 290 unique real-world repositories, spanning 6 package ecosystems and 147 unique CWE categories, with 78% of entries carrying assigned CVE identifiers. Some repositories contribute multiple tasks corresponding to distinct vulnerabilities, advisories, or pull requests. Each task pairs a repository snapshot containing a known vulnerability with ground-truth implementation file labels. Evaluation proceeds in two phases: Phase A (localization) requires identifying vulnerable files given a CWE description, while Phase B (verification) presents patched code and expects the model to declare no vulnerability present. The experiments in this report focus on Phase A localization; Phase B is included in the benchmark specification but is not evaluated here.

7.2. Metrics

We evaluate whether models identify the correct vulnerable files, rather than simply classifying a repository as vulnerable. For each task, we compare the submitted files with the ground-truth set and compute file-level precision, recall, and F1. We macro-average each metric across all 500 tasks.

- **File F1:** For each task, the harmonic mean of file-level precision and recall. This is the primary metric used for model comparisons.
- **Precision:** For each task, the fraction of submitted file paths that appear in the ground-truth set.
- **Recall:** For each task, the fraction of ground-truth vulnerable files included in the submitted file paths.

- **Abstain Rate:** The fraction of tasks for which the model submits no vulnerable file paths, either by explicitly declaring that no matching vulnerability was found or by failing to produce a valid localization submission.

Because every Phase A task contains a known vulnerability, abstaining receives zero precision, recall, and File F1 for that task.

7.3. Models Evaluated

We compare Antares against frontier closed-source models, large open-weight models, and small open-weight models, all evaluated using the same harness, tools, task inputs, interaction budget, and generation settings of temperature 0.3 and top-p 1.0. We run each model three times and report the average across the three runs.

- **Frontier (closed):** GPT-5.5 (reasoning_effort = `default`, `xhigh`), GPT-5, GPT-5 Mini, GPT-5 Nano, Gemini 3 Pro, Gemini 2.5 Flash, Gemini 3.1 Flash Lite.
- **Open-weight large ($\geqslant$20B):** GLM-5.2, MiniMax-M2.7, Qwen3.5-122B-A10B, GPT-OSS-120B, Llama-3.3-70B, Qwen3.5-35B-A3B, Gemma-4-31B, Qwen3.5-27B, GPT-OSS-20B.
- **Open-weight small (<20B):** CodeScout-14B, Qwen3.5-9B, Gemma-4-E4B, Gemma-4-E2B.
- **Antares family:** 350M (SFT, GRPO), 1B (SFT, GRPO), 3B (SFT, GRPO).
- **Baselines:** Granite 4.0 base models (350M, 1B, 3B) without any post-training.

7.4. Evaluation Protocol

All models are evaluated on the benchmark using an identical agent protocol. Each task runs in a fresh Docker container (Ubuntu 24.04) with 2 CPU cores, 4GB RAM, network disabled, and a 10-second per-command timeout. The container is destroyed after each task.

The agent protocol provides a budget of 15 terminal commands per task, followed by one final submission action, with up to 3 retries on unsuccessful commands. The submission action does not count toward the terminal-call budget. Three tools are available: `terminal` (read-only repository navigation and inspection commands), `submit_vulnerable_files` (ranked file paths), and `submit_no_vulnerability_found` (declare clean). The model receives only the CWE category description as input, with no advisory text, file hints, or severity details.

Antares models are served via vLLM on a single GPU (bfloat16, max-model-len 32768) with temperature 0.3 and top-p 1.0. External models use their respective API endpoints. Evaluation runs 16 parallel workers processing entries concurrently.

8. Results

We evaluate Antares across model scale, vulnerability structure, repository complexity, agent behavior, and training stage. Across these analyses, a consistent picture emerges: repository-scale vulnerability localization is not primarily a general-purpose code understanding benchmark. It rewards models that learn how to search, verify, and submit under a constrained interaction budget.

8.1. Task-Specific Training Dominates Parameter Scale

Tables 4 and 5 evaluate whether vulnerability localization improves smoothly with model scale. If general-purpose scale were sufficient, we would expect larger open-weight and frontier models to

Table 4 | Frontier model comparison on VLoc Bench (Phase A). Performance exhibits a capability cliff: models either achieve the 0.186–0.229 tier or fall below 0.152 regardless of general-purpose scale.

Model	File F1	Precision	Recall
GPT-5.5 (xhigh)	0.229	0.310	0.221
Antares-3B	**0.223**	**0.303**	**0.221**
GPT-5.5 (default)	0.221	0.305	0.211
Antares-1B	**0.209**	**0.262**	**0.224**
GLM-5.2	0.186	0.226	0.186
Gemini 3 Pro	0.152	0.190	0.153
Antares-350M	**0.135**	**0.136**	**0.178**
Gemini 2.5 Flash	0.102	0.132	0.098
GPT-5 Mini	0.098	0.115	0.096
Gemini 3.1 Flash Lite	0.095	0.131	0.090
GPT-5	0.048	0.062	0.048
GPT-5 Nano	0.024	0.038	0.021

dominate smaller specialized models. Instead, the results show a capability cliff. GPT-5.5, Antares-3B, and GLM-5.2 form the only high-performing tier, while many larger general-purpose models fall far below this range.

Antares-3B reaches 0.223 File F1, approaching GPT-5.5 while outperforming substantially larger open-weight models, including GLM-5.2. The comparison against static analysis tools shows the same pattern from the opposite direction: rule-based scanners recover some vulnerable files, but remain below Antares models because they lack the ability to adaptively inspect repository context. Parameter count alone is therefore a poor predictor of localization performance.

The precision–recall decomposition further suggests that different model sizes learn different operating regimes. Antares-3B behaves conservatively, matching GPT-5.5's recall while maintaining high precision. Antares-1B achieves the highest recall of all evaluated systems, suggesting a search-heavy strategy that finds more candidate vulnerable files. Antares-350M shifts further toward recall at lower precision, consistent with a smaller model that can search effectively but has weaker verification capacity.

Observation 1

Repository-scale vulnerability localization exhibits a capability cliff rather than smooth scaling. Compact models trained specifically for agentic vulnerability localization can outperform open-weight models hundreds of times larger, indicating that task-specific interaction training matters more than parameter count alone.

8.2. Localization Difficulty Follows Structure, Not Severity

We next ask what makes a vulnerability localization instance difficult. A natural hypothesis is that more severe vulnerabilities should be easier to find because they may correspond to more obvious or security-critical code. The results do not support this hypothesis. Instead, difficulty is dominated by repository and vulnerability structure.

Table 5 | Comparison of open-weight models and static analysis tools on VLoc Bench. Antares achieves the strongest localization performance, suggesting that the task requires capabilities beyond general-purpose scale and rule-based analysis.

Model	Params	File F1	Precision	Recall
Open-Weight Models				
Antares-3B	**3B**	**0.223**	**0.303**	**0.221**
Antares-1B	**1B**	**0.209**	**0.262**	**0.224**
GLM-5.2	753B	0.186	0.226	0.186
Antares-350M	**350M**	**0.135**	**0.136**	**0.178**
Gemma-4-31B	31B	0.101	0.131	0.097
Qwen3.5-27B	27B	0.091	0.116	0.088
Qwen3.5-122B-A10B	125B	0.091	0.124	0.083
Qwen3.5-35B-A3B	36B	0.085	0.115	0.081
GPT-OSS-20B	20B	0.070	0.095	0.065
GPT-OSS-120B	120B	0.069	0.095	0.062
MiniMax-M2.7	229B	0.054	0.078	0.050
CodeScout-14B	14B	0.044	0.065	0.039
Qwen3.5-9B	9B	0.043	0.058	0.039
Gemma-4-E2B	2B	0.039	0.045	0.042
Gemma-4-E4B	4B	0.034	0.039	0.034
Llama-3.3-70B	70B	0.012	0.016	0.014
Static Analysis Tools				
Semgrep	N/A	0.086	0.091	0.155
Semgrep-CWE	N/A	0.052	0.057	0.071
CodeQL	N/A	0.023	0.025	0.030
Horusec	N/A	0.020	0.021	0.038

Figure 4 shows that ecosystem structure strongly affects localization performance. Flat, convention-heavy ecosystems such as pip and npm produce the highest scores across models, while Maven remains difficult for every system. This suggests that localization is easier when vulnerable logic is concentrated in shallow, predictable paths, and harder when evidence is distributed across verbose build hierarchies, framework conventions, and multi-class implementations.

Figure 5 shows the opposite pattern for CVSS severity. Performance varies little across Critical, High, Medium, and Low bins, indicating that vulnerability impact is not the main driver of localization difficulty. The CWE-level results reinforce this interpretation: structurally distinctive vulnerabilities such as code injection and deserialization are easier for Antares, while diffuse data-flow categories such as information exposure remain difficult for every model.

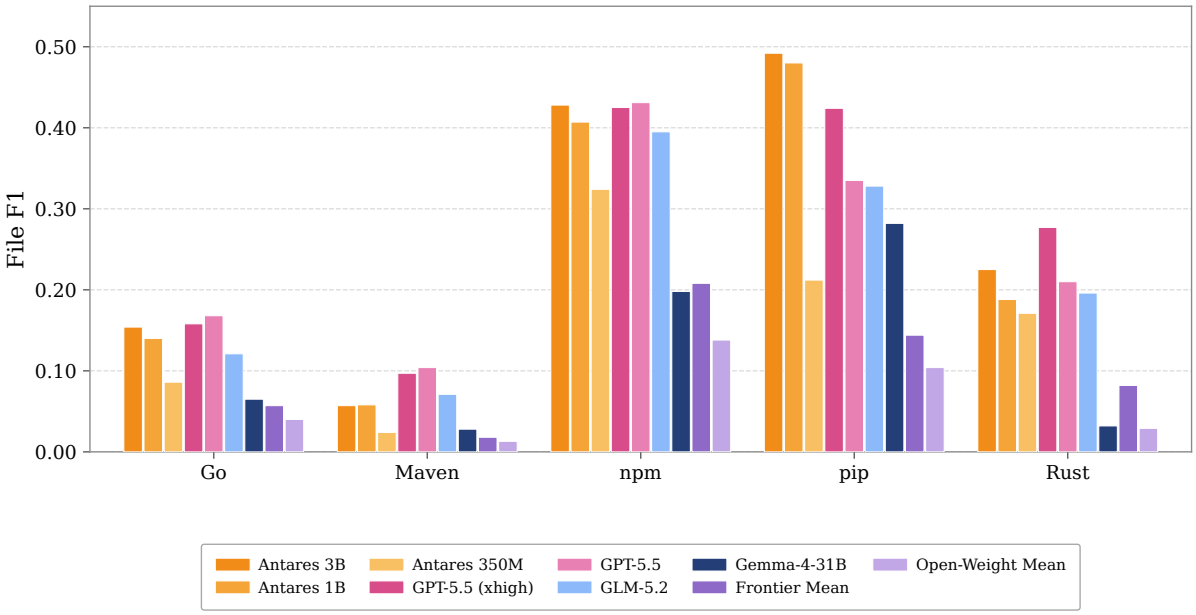


Figure 4 | File F1 disaggregated by package ecosystem (top 5 by frequency). Task counts: Go (n=215), Maven (n=104), npm (n=88), pip (n=52), Rust (n=40). Ecosystem structure determines difficulty uniformly across all models, with pip and npm yielding 7–14× higher scores than Maven regardless of model scale.

Observation 2

Localization difficulty is governed by structural locality rather than vulnerability severity. Ecosystem conventions and CWE-specific implementation patterns determine whether an agent can efficiently narrow the search space.

8.3. Repository Scale and Multi-File Vulnerabilities Remain the Core Bottleneck

We then examine whether repository complexity changes which model strategy is most effective. Our hypothesis is that RL-trained search policies should excel when the repository is small enough for terminal exploration to cover most relevant files, while larger repositories should favor models with stronger long-horizon reasoning.

Figure 6 supports this hypothesis. On repositories under 100 KB, Antares models achieve the strongest performance of any evaluated system, with Antares-1B reaching 0.843 File F1 and Antares-3B reaching 0.828. In this regime, grep-based elimination and targeted file inspection are sufficient to approach complete coverage within the turn budget. As repositories grow, however, the advantage narrows. At the 10+ MB tier, GPT-5.5 variants overtake Antares, suggesting that larger repositories require architectural reasoning beyond efficient search.

The number of ground-truth files provides a second complexity axis. Single-file vulnerabilities are substantially easier for all models, while entries with five or more vulnerable files cause performance to collapse. This shows that current agents are much better at identifying a vulnerable foothold than recovering every file involved in a distributed vulnerability.

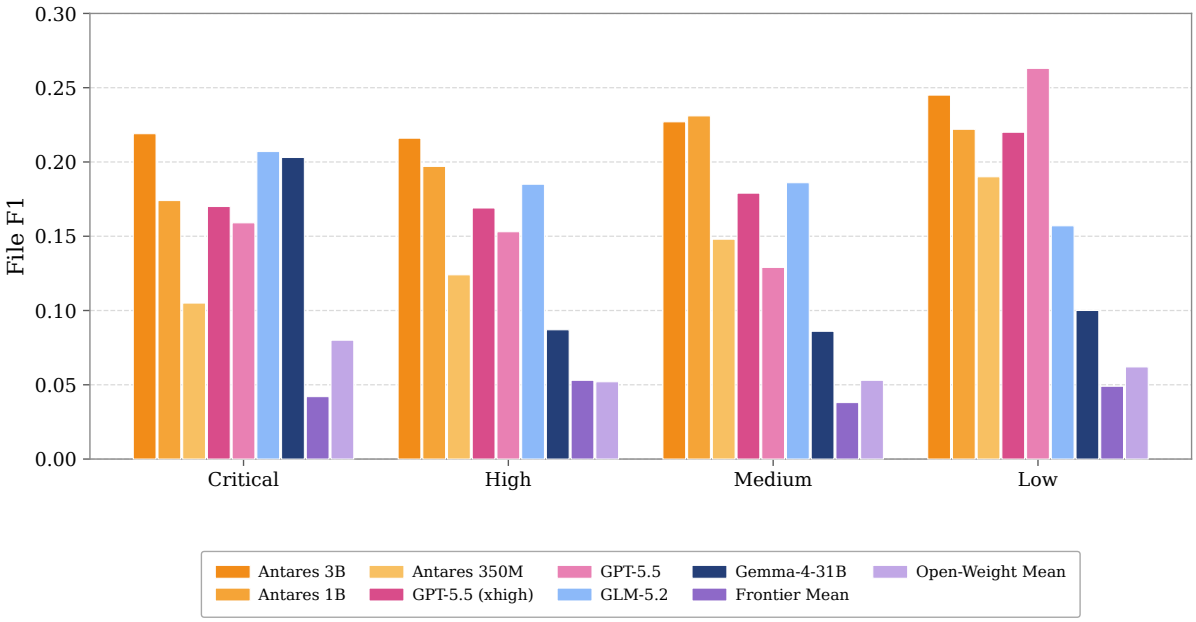


Figure 5 | File F1 disaggregated by CVSS severity. Task counts: Critical (n=57), High (n=219), Medium (n=194), Low (n=30). Performance varies less across severity bins than across ecosystem and repository-structure categories, confirming that localization difficulty is determined by repository structure rather than vulnerability impact.

Observation 3

RL-trained search policies are most effective when repository evidence can be covered within the interaction budget. Performance degrades sharply as vulnerabilities become distributed across larger codebases and multiple ground-truth files.

8.4. GRPO Induces a Search–Verify–Refine Policy

The aggregate scores show that Antares is competitive, but they do not explain how it behaves. We therefore ask whether Antares solves localization by imitating frontier-style repository comprehension, or whether GRPO induces a distinct search policy.

Figure 7 shows that Antares-3B uses a narrower command repertoire than frontier models while maintaining non-trivial transition complexity. It relies heavily on search commands, uses fewer structural exploration commands, and issues fewer total commands per task than GPT-5.5. This suggests that Antares does not try to build a complete mental model of the repository. Instead, it treats localization as elimination: search broadly for vulnerability-relevant terms, read candidate files, and refine the search based on evidence.

The entropy analysis clarifies that this behavior is not simply rigid repetition. Antares has lower category entropy than frontier models, but its bigram entropy remains close to GPT-5.5 and GPT-OSS. In other words, Antares uses fewer action types, but it adapts how it transitions between them. Its behavioral complexity is concentrated in the search–verify–refine loop, which is directly aligned with the file-localization objective.

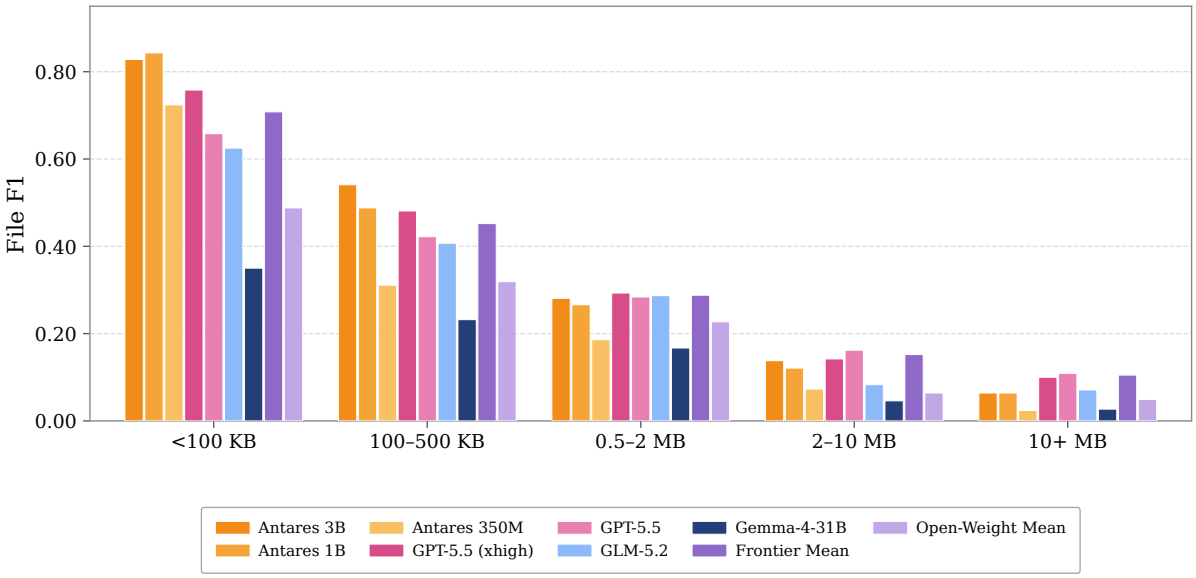


Figure 6 | File F1 by repository size (total codebase). Task counts: <100 KB (n=20), 100–500 KB (n=82), 0.5–2 MB (n=84), 2–10 MB (n=91), 10+ MB (n=223). All models decline sharply as repository size increases, but Antares maintains competitive or superior performance at every scale.

Observation 4

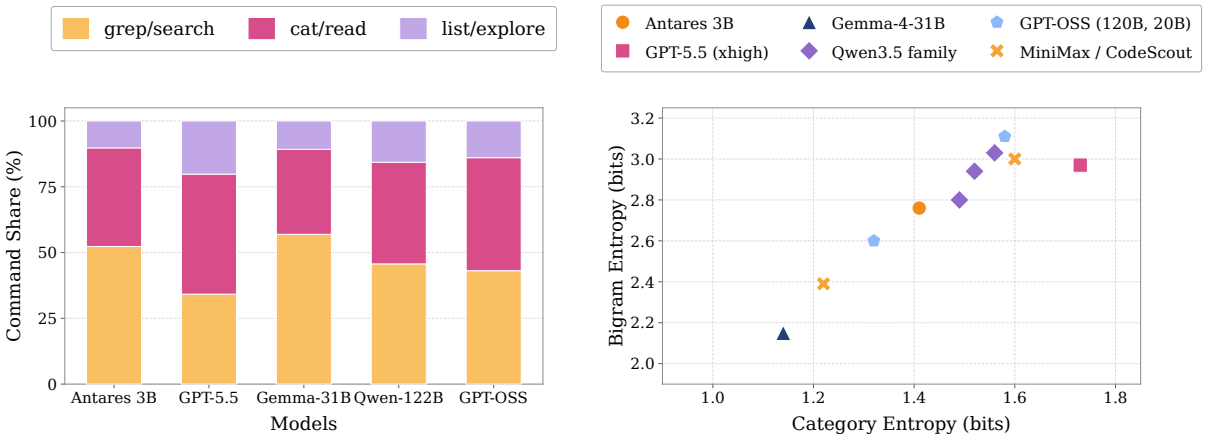
GRPO induces a specialized search–verify–refine policy. Antares-3B is less behaviorally broad than frontier models, but its transitions remain adaptive and concentrated on the actions most useful for file-level localization.

8.5. Training Progression and Emergent Specialization

We next ask how each stage of the Antares training pipeline contributes to the final agent. The central hypothesis is that SFT and GRPO play different roles: SFT should teach the model how to operate in the terminal environment, while GRPO should teach the model which interaction strategies are useful for vulnerability localization. We test this by comparing base, SFT, and GRPO checkpoints across all three model scales, then analyzing how behavior changes after reinforcement learning.

Performance Progression Figure 8 quantifies the contribution of each training stage. The base Granite models achieve near-zero File F1, indicating that general tool-calling ability alone is insufficient for repository-scale vulnerability localization. SFT with semantic conditioning lifts all three model scales into functional localization agents, with the 1B and 3B models clustering together at 0.188 and 0.198 File F1, while the 350M model remains lower at 0.108. This suggests that terminal-based vulnerability localization requires a minimum capacity threshold, but that SFT alone can already teach the basic interaction protocol.

GRPO then improves all three scales, but the gains are not uniform. The 350M model receives the largest relative improvement, increasing by 25%, while the 1B and 3B models improve by 11–13%. This pattern suggests that GRPO is most valuable when the SFT policy has learned the environment format but has not yet discovered reliable search behavior. At larger scales, the SFT initialization already captures more of the useful strategy space, leaving less room for policy optimization to improve mean performance.



(a) Command category distribution.

(b) Behavioral entropy per model.

Figure 7 | Behavioral strategy analysis across 500 tasks from one representative run. (a) Command allocation shows Antares-3B maintains the highest search ratio among models with balanced read profiles. (b) Entropy scatter reveals Antares-3B achieves lower category entropy than all frontier models while sustaining moderate transition complexity through its search-verify loop.

Table 6 | **Operational metrics from SFT to GRPO across model scales.** Values are averaged across three runs. σ denotes the standard deviation of File F1 across runs, and Files Sub. denotes the mean number of file paths submitted per task.

Scale	Stage	File F1	σ	Prec.	Rec.	Abstain	Files Sub.
350M	SFT	0.108	0.0083	0.149	0.101	5.8%	1.30
350M	**GRPO**	**0.135**	**0.0031**	**0.136**	**0.178**	**1.4%**	**4.23**
1B	SFT	0.188	0.0052	0.263	0.179	3.0%	1.55
1B	**GRPO**	**0.209**	**0.0030**	**0.262**	**0.224**	**0.6%**	**2.95**
3B	SFT	0.198	0.0062	0.240	0.228	7.5%	2.68
3B	**GRPO**	**0.223**	**0.0022**	**0.303**	**0.221**	**4.0%**	**1.84**

The error bars in Figure 8 are as important as the mean improvements. GRPO reduces run-to-run standard deviation by 42–65% across all scales (Table 6), indicating that RL collapses the policy toward a smaller set of high-return trajectories rather than merely increasing average performance. This variance reduction is operationally important: a single GRPO evaluation run provides a more reliable estimate of model behavior than a single SFT run.

Observation 5

SFT teaches Antares to operate as a terminal agent with security knowledge, while GRPO makes that behavior more reliable and task-directed. The primary effect of GRPO is not only higher File F1, but lower variance and more stable localization behavior.

Emergent Scale-Dependent Specialization We then ask whether GRPO induces the same strategy across model scales. Before RL, all three models follow a similar read-dominant imitation policy, with 27–32% search commands and 54–60% file-reading commands. This is expected: SFT trains all scales on

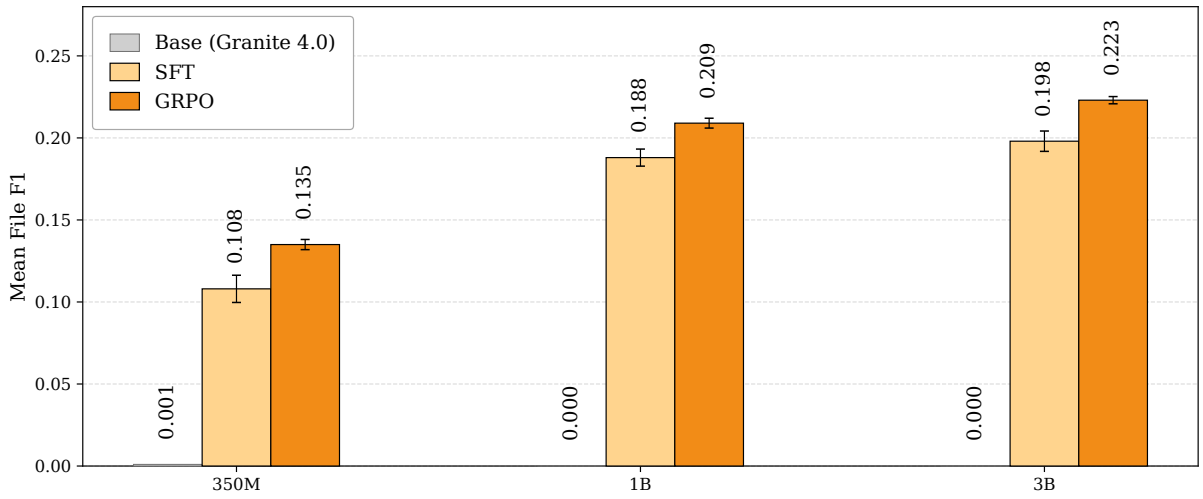


Figure 8 | Mean File F1 across the three-stage training pipeline (Base → SFT → GRPO) for each Antares model scale, averaged over 3 independent evaluation runs on VLoc Bench.

the same behavioral distribution, so the models initially imitate the same style of repository exploration.

After GRPO, this shared behavior disappears. Because the reward does not prescribe a specific distribution over search, read, and exploration command categories, each model scale discovers its own operating point. The 350M and 1B models shift sharply toward search-heavy behavior, using 87–89% search commands and submitting more files. This high-recall strategy compensates for limited verification capacity by maximizing coverage. In contrast, the 3B model maintains a more balanced search/read policy, using 52% search and 37% read commands, while submitting fewer files at higher precision. This divergence suggests that GRPO does not teach a single universal localization algorithm. Instead, it exposes a capacity-dependent tradeoff between search coverage and verification quality. Smaller models benefit from broad search and higher submission volume, while the 3B model has enough capacity to verify candidates more selectively.

Observation 6

GRPO induces scale-dependent specialization. Smaller Antares models learn high-recall search-heavy policies, while Antares-3B learns a more selective search-and-verify policy with higher precision.

Table 7 | SFT File F1 by auxiliary objective. Semantic conditioning produces the strongest initialization for GRPO at all scales, with the largest margin at 350M where explicit terminal grounding is most critical.

Auxiliary Objective	350M	1B	3B
No auxiliary loss	0.021	0.151	0.164
ECHO [43]	0.076	0.174	0.177
Semantic conditioning (ours)	**0.108**	**0.188**	**0.198**

Auxiliary Objective Ablation Finally, we test whether the SFT auxiliary objective affects downstream agent quality. Table 7 shows that semantic conditioning is the strongest initialization at every model

scale. The effect is largest at 350M, where semantic conditioning improves File F1 by 5.1× over no auxiliary loss. At 3B, the gain narrows to 20.7%, suggesting that larger models can partially infer terminal dynamics from ordinary next-token supervision, while compact models require more explicit grounding.

This result supports the view that auxiliary objectives shape the behavioral distribution available to GRPO. A weak SFT initialization gives RL fewer useful trajectories to reinforce; a stronger initialization exposes more viable search, read, and submit behaviors. Semantic conditioning therefore acts less like a small additive improvement and more like a multiplier on the effectiveness of the entire post-training pipeline.

Observation 7

Semantic conditioning provides the strongest SFT initialization for terminal-based vulnerability localization. Its effect is largest for compact models, where explicit grounding of terminal observations is most important.

8.6. Does Vulnerability-Localization Training Transfer Beyond VLoc Bench?

The evaluations above focus on repository-scale vulnerability localization, the capability directly optimized during Antares training. To test whether the learned policy transfers beyond this setting, we additionally evaluate Antares on issue-driven code localization and structured multi-turn tool use. Full results are reported in Appendix B.

On SWE-Bench, Antares remains competitive with substantially larger models trained specifically for code localization, despite having no exposure to SWE-Bench repositories, issue descriptions, or issue-resolution data. On BFCL-v3, Antares shows its largest gains over the corresponding Granite base models in multi-turn orchestration, while aggregate function-calling performance remains broadly comparable. Together, these results indicate that Antares acquires transferable repository-navigation and sequential tool-use capabilities rather than a policy narrowly specialized to VLoc Bench.

9. Discussion

Where Current Agents Still Fail The hardest entries are not random failures; they concentrate in repositories with large search spaces, many files, and diffuse vulnerability evidence. In the highest-complexity quartile, Antares-3B falls below 0.04 File F1, compared to 0.55 on the easiest quartile. This degradation is shared across models: even GPT-5.5 fails on a subset of entries where other models recover signal, and entries where no evaluated model achieves non-zero F1 are dominated by structurally complex Go and Maven repositories.

The underlying failure mode is **signal dilution**. In large repositories, vulnerability-relevant patterns such as unsafe calls, missing validation, or attacker-controlled data flow may appear in many benign contexts. The agent must therefore identify not only a suspicious pattern, but the specific instance that participates in the vulnerable implementation. This requires reading and integrating surrounding context across a scale that exceeds the effective working memory of current terminal agents. As a result, models often find plausible candidate files but fail to distinguish the security-critical implementation from syntactically similar benign code.

Distributed vulnerabilities create a related failure mode. When the ground truth spans multiple files, models frequently identify an initial vulnerable foothold but miss supporting files along the call path, configuration boundary, or validation chain. This explains why performance drops sharply on entries with many ground-truth files: the challenge is not merely finding one relevant file, but recovering the complete implementation slice that constitutes the vulnerability.

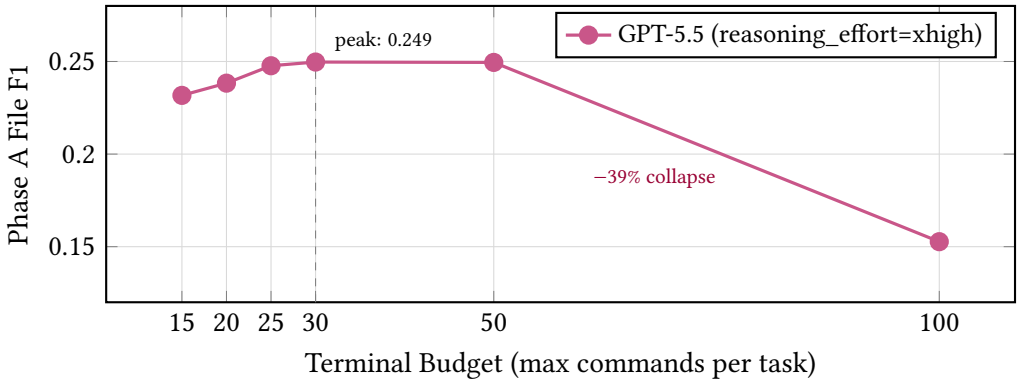


Figure 9 | GPT-5.5 File F1 as a function of terminal command budget. Performance saturates around 30 commands and degrades at 100 commands, suggesting that localization is limited by evidence prioritization rather than command count alone.

Does More Tool Use Help? A natural hypothesis is that these failures arise because agents simply need more terminal commands. Turn-budget experiments do not support this explanation. As shown in Figure 9, increasing the budget from 15 to 30 commands yields only a modest improvement for GPT-5.5, after which performance saturates. At a budget of 100 commands, performance drops sharply as the model over-explores and loses confidence in its candidate set.

This suggests that the limiting factor is not the number of available commands, but the model's ability to decide **where** to look and when to stop. More interaction can even be harmful when the agent lacks a stable evidence-ranking strategy: additional grep hits, directory listings, and file reads expand the candidate set faster than the model can resolve it. Future progress will therefore require better evidence memory, candidate ranking, and multi-file reasoning rather than simply longer tool budgets.

Operational Efficiency Although these limitations remain, Antares completely changes the operational cost profile of repository-scale localization. Antares-3B completes the full 500-task evaluation sweep in approximately 15 minutes on a single H100 GPU with 16 parallel workers, corresponding to an estimated inference cost of less than $1 per evaluation sweep using commodity H100 rental rates (Figure 10). In comparison, the strongest open-weight baseline, GLM-5.2, requires approximately 50 minutes and $12.50 through OpenRouter, while GPT-5.5 requires approximately 5 hours and $141 through the OpenAI API.

This gap matters because vulnerability localization is not usually a one-off query. Practical deployment requires repeated scans across repositories, branches, dependency updates, and CI/CD events. When deployed with a local inference endpoint, Antares enables repeated repository-scale evaluation without sending proprietary source code to third-party APIs, making the system suitable for closed-network security workflows where cost, latency, and source-code privacy are deployment constraints rather than secondary considerations.

10. Safety and Responsible Disclosure

Model release We release the Antares-350M and Antares-1B models on Hugging Face; Antares-3B, our most competitive variant, is retained for internal use and is not part of this release. Because Antares is a dual-use artifact—trained specifically to localize exploitable vulnerabilities in arbitrary repositories—we release the models with explicit acceptable-use restrictions. The models are intended for defensive security research, vulnerability assessment, remediation, and evaluation. We prohibit use for offensive

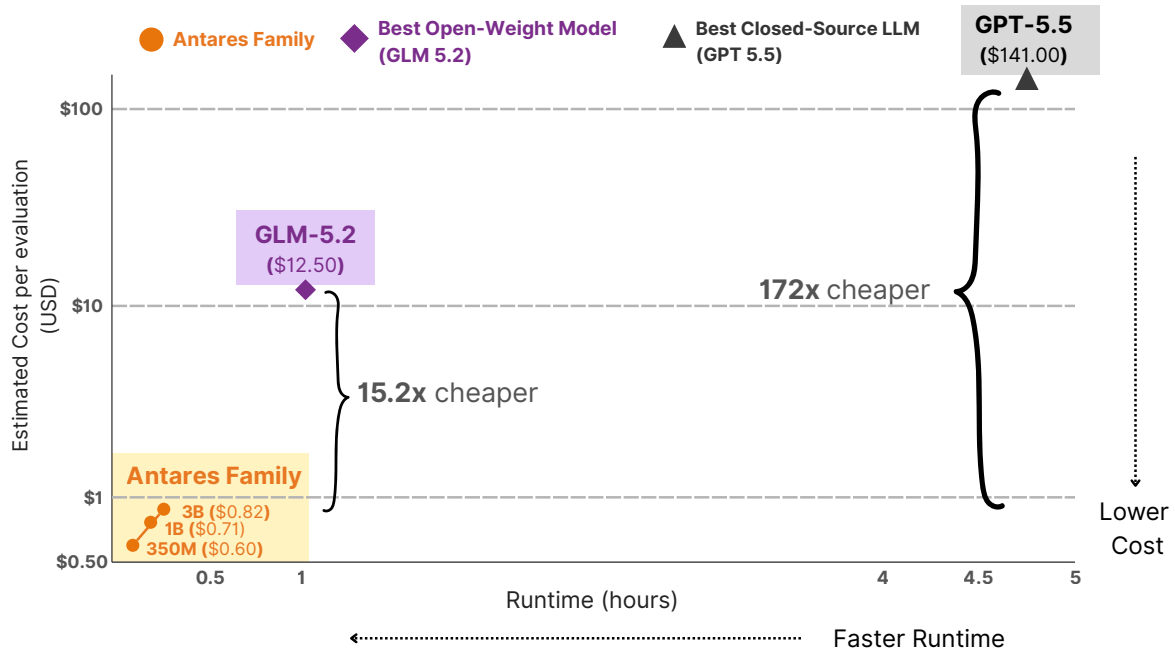


Figure 10 | Runtime and estimated inference cost for evaluating the VLoc Bench. All evaluations were performed using 16 parallel workers. Antares models were evaluated on a single H100 GPU, with costs estimated from publicly available H100 hourly rental prices. GLM-5.2 was evaluated through OpenRouter, while GPT-5.5 was evaluated through the OpenAI API. Reported runtime and cost reflect our evaluation setup and may vary depending on API pricing, deployment configuration, provider infrastructure, and rate limits.

cyber operations, unauthorized vulnerability discovery or exploitation, attacker enablement, credential theft, malware development, or any activity intended to compromise systems without authorization.

CLI release The Antares CLI is bundled with the Antares-1B Hugging Face repository rather than released independently. Because it operates over potentially untrusted repositories and may connect to a user-configured inference endpoint, the deployment is designed around repository containment and explicit data boundaries. Before inference, eligible files are copied into a temporary read-only snapshot. Symlinks that resolve outside the repository are discarded, and the agent is restricted to parsed, read-only inspection commands. It cannot modify the repository, access the network, traverse beyond the snapshot, or inspect sensitive credential locations. Repository profiling used for automatic CWE selection is performed locally. Repository contents may themselves contain instructions intended to manipulate the model. To reduce this risk, content returned through model-requested inspection is scanned before being added to the transcript, and detected prompt-injection patterns are quarantined. Query and sweep operations may send the task instructions, repository paths, initial file inventory, and source content selected during inspection to the configured inference endpoint. Private local traces may also retain prompts, model responses, source excerpts, commands, paths, and Git metadata until deletion.

11. Conclusion

We introduced Antares, a family of compact language models for agentic vulnerability localization. By combining cybersecurity reasoning, terminal exploration trajectories, and reinforcement learning from verifiable file-level rewards, Antares learns to localize vulnerable implementations directly from a repository and a CWE description, without relying on static-analysis candidates or frozen frontier-model scaffolds.

Our results show that targeted post-training can compensate for substantial differences in model scale. On VLoc Bench, Antares-3B approaches GPT-5.5 while outperforming substantially larger open-weight models, and GRPO induces more stable, scale-dependent search strategies across the Antares family. At the same time, performance remains limited on large repositories, distributed vulnerabilities, and vulnerability classes defined by diffuse data flow, highlighting the need for stronger long-horizon repository reasoning.

Overall, Antares demonstrates that compact, locally deployable models can perform meaningful agentic security analysis when trained directly on the interaction pattern required by the task. To support continued research and responsible evaluation, we publicly release Antares-350M and Antares-1B through Hugging Face and bundle the inference CLI with the Antares-1B model repository.

Acknowledgements

We thank Xuhong He, Karen Kui, Abhinav Chinta, Hadas Birin, Howard Lin, Huaibo Zhao, and Yasukazu Hirata for their support, guidance, and feedback throughout this work.

We also thank the S&TO team, including Theo Morales, Aaron Carter, Thomas Bartlett, Omar Santos, and Anthony Grieco, for internally testing the model and providing feedback that helped inform the release process.

We are grateful to Jen Yokoyama, Marc Jones, and Elena Garcia from the legal team for their careful review and guidance.

We also thank Elizabeth Adkison and Emile Antone from the marketing team, as well as Susan O'Brien, Blake Thompson Heuer, Carro Halpin, and Nicole Greggs from the PR team, for their support with the release process and broader launch coordination.

References

[1] Md Tanvirul Alam, Dipkamal Bhusal, Le Nguyen, and Nidhi Rastogi. Ctibench: A benchmark for evaluating llms in cyber threat intelligence, 2024. URL https://arxiv.org/abs/2406.07599.

[2] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. "False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing . In 2024 IEEE Symposium on Security and Privacy (SP), pages 3979–3997, Los Alamitos, CA, USA, May 2024. IEEE Computer Society. doi: 10.1109/SP54263.2024.00019. URL https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00019.

[3] Anthropic. Claude code. https://www.anthropic.com/product/claude-code, 2026. Anthropic product page. Accessed: 2026-06-29.

[4] Anysphere. Cursor: The ai code editor. https://cursor.com/, 2026. Cursor product page. Accessed: 2026-06-29.

[5] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment, 2025. URL https://arxiv.org/abs/2506.07982.

[6] Wachiraphan Charoenwet, Kla Tantithamthavorn, Patanamon Thongtanunam, Hong Yi Lin, Minwoo Jeong, and Ming Wu. Agenticscr: An autonomous agentic secure code review for immature vulnerabilities detection, 2026. URL https://arxiv.org/abs/2601.19138.

[7] Cisco Foundation AI. Vulnerability localization benchmark. Manuscript, 2026. Available at https://github.com/cisco-foundation-ai/vulnerability-localization-benchmark.

[8] Cursor Team. Composer: Building a fast frontier model with rl. https://cursor.com/blog/composer, 2025. Cursor research blog. Accessed: 2026-06-29.

[9] Debrup Das, Sam O' Nuallain, and Razieh Rahimi. Rader: Reasoning-aware dense retrieval models, 2025. URL https://arxiv.org/abs/2505.18405.

[10] DeepHat. Deephat-v1-7b. https://huggingface.co/DeepHat/DeepHat-V1-7B, 2025. Hugging Face model repository. Finetuned from Qwen2.5-Coder-7B. Accessed: 2026-06-29.

[11] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

[12] GitHub. Codeql: Semantic code analysis. https://codeql.github.com, 2025. GitHub documentation. Accessed: 2026-06-29.

[13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli

Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. Nature, 645(8081):633–638, 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

[14] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing, 2025. URL https://arxiv.org/abs/2501.18160.

[15] IBM Granite Team. Granite-4.0-1b. https://huggingface.co/ibm-granite/granite-4.0-1b, 2025. Hugging Face model repository. Accessed: 2026-06-29.

[16] IBM Granite Team. Granite-4.0-350m. https://huggingface.co/ibm-granite/granite-4.0-350m, 2025. Hugging Face model repository. Accessed: 2026-06-29.

[17] IBM Granite Team. Granite-4.0-micro. https://huggingface.co/ibm-granite/granite-4.0-micro, 2025. Hugging Face model repository. Accessed: 2026-06-29.

[18] Samy Jelassi, Mujin Kwun, Rosie Zhao, Yuanzhi Li, Nicolo Fusi, Yilun Du, Sham M. Kakade, and Carles Domingo-Enrich. Matching features, not tokens: Energy-based fine-tuning of language models, 2026. URL https://arxiv.org/abs/2603.12248.

[19] Pengcheng Jiang, Jiacheng Lin, Lang Cao, Runchu Tian, SeongKu Kang, Zifeng Wang, Jimeng Sun, and Jiawei Han. Deepretrieval: Hacking real search engines and retrievers with large language models via reinforcement learning, 2025. URL https://arxiv.org/abs/2503.00223.

[20] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

[21] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.09516.

[22] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020. URL https://arxiv.org/abs/2004.04906.

[23] Paul Kassianik, Baturay Saglam, Alexander Chen, Blaine Nelson, Anu Vellore, Massimo Aufiero, Fraser Burch, Dhruv Kedia, Avi Zohary, Sajana Weerawardhena, Aman Priyanshu, Adam Swanda, Amy Chang, Hyrum Anderson, Kojin Oshiba, Omar Santos, Yaron Singer, and Amin Karbasi. Llama-3.1-foundationai-securityllm-base-8b technical report, 2025. URL https://arxiv.org/abs/2504.21039.

[24] Paul Kassianik, Baturay Saglam, Huaibo Zhao, Blaine Nelson, Supriti Vijay, Aman Priyanshu, and Amin Karbasi. Fapo: Fully automated prompt optimization of multi-step llm pipelines, 2026. URL https://arxiv.org/abs/2606.19605.

[25] Xiaoxi Li, Jiajie Jin, Guanting Dong, Hongjin Qian, Yongkang Wu, Ji-Rong Wen, Yutao Zhu, and Zhicheng Dou. Webthinker: Empowering large reasoning models with deep research capability, 2025. URL https://arxiv.org/abs/2504.21776.

[26] Zhuofeng Li, Dongfu Jiang, Xueguang Ma, Haoxiang Zhang, Ping Nie, Yuyu Zhang, Kai Zou, Jianwen Xie, Yu Zhang, and Wenhu Chen. Openresearcher: A fully open pipeline for long-horizon deep research trajectory synthesis, 2026. URL https://arxiv.org/abs/2603.20278.

[27] Hanzhi Liu, Chaofan Shou, Xiaonan Liu, Hongbo Wen, Yanju Chen, Ryan Jingyang Fang, and Yu Feng. Synthesizing multi-agent harnesses for vulnerability discovery, 2026. URL https://arxiv.org/abs/2604.20801.

[28] MiniMax, :, Aili Chen, Aonian Li, Baichuan Zhou, Bangwei Gong, Binyang Jiang, Boji Dan, Changqing Yu, Chao Wang, Cheng Ma, Cheng Zhong, Cheng Zhu, Chengjun Xiao, Chengyi Yang, Chengyu Du, Chenyang Zhang, Chi Zhang, Chuangyi Huang, Chunhao Zhang, Chunhui Du, Chunyu Zhao, Congchao Guo, Da Chen, Deming Ding, Dianjun Sun, Dongyu Zhang, Enhui Yang, Fei Yu, Guang Zheng, Guodong Zheng, Guohong Li, Haichao Zhu, Haigang Zhou, Haimo Zhang, Han Ding, Hao Zhang, Haohai Sun, Haolin Lyu, Haonan Lu, Haoyu Wang, Huajie Shi, Huiyang Li, Jiacheng Chen, Jian Zhang, Jiaqi Zhuang, Jiaren Cai, Jiaxin Pan, Jiayao Li, Jiayuan Song, Jichuan Zhang, Jie Wang, Jihao Gu, Jin Zhu, Jingwei Dong, Jingyang Li, Jingyu Zhang, Jingze Zhuang, Jinhao Tian, Jinli Liu, Jinyi Hu, Jun Tao, Jun Zhang, Junbin Ruan, Junhao Xu, Junjie Yan, Junteng Liu, Junxian He, Kang Xu, Ke Ji, Ke Yang, Kecheng Xiao, Keyu Duan, Keyu Li, Le Han, Letian Ruan, Li Yuan, Lianfei Yu, Liheng Feng, Lijie Mo, Lin Li, Lingye Bao, Lingyu Yang, Lingyuan Zhou, Loki, Lu Chen, Lunbin Ceng, Ming Li, Ming Zhong, Mingliang Tao, Mingyuan Chi, Mujie Lin, Nan Hu, Ningxin Chen, Peiyin Zhu, Peng Gao, Pengcheng Gao, Pengfei Li, Penglin Li, Pengyu Zhao, Qibin Ren, Qidi Xu, Qihan Ren, Qile Li, Qin Wang, Quanliang Chen, Qunhong Ceng, Rong Tian, Rui Dong, Ruitao Leng, Ruize Zhang, Shanqi Liu, Shaoyu Chen, Sheng Jia, Shun Yao, Shuoran Zhao, Shuqi Yu, Sichen Li, Sicheng Pan, Songquan Zhu, Tengfei Li, Tian Xie, Tiancheng Qin, Tianrun Liang, Wei Liu, Weiqi Xu, Weitao Li, Weixiang Chen, Weiyu Cheng, Weiyu Zhang, Wenhu Chen, Wenqian Zhao, Xiancai Chen, Xiangjun Song, Xiangyuan Wang, Xiao Luo, Xiao Su, Xiaobo Li, Xiaodong Han, Xiaojie Wu, Xihao Song, Xingyi Han, Xinyu Guan, Xuan Lu, Xun Zou, Xunhao Lai, Xutong Li, Yan Gong, Yang Wang, Yang Xu, Yangsen Wang, Ye Tang, Yicheng Chen, Yinran Qiu, Yiqi Shi, Yiting Guo, Yiwen Huang, Yixuan Wang, Yongyi Hu, Yu Gao, Yu Zhang, Yuanxiang Ying, Yuanzhen Zhang, Yubo Wang, Yuchen Song, Yufeng Yang, Yuhang Meng, Yuhang Miao, Yuhao Li, Yujie Liu, Yulin Hu, Yunan Huang, Yunji Li, Yunyi Huang, Yusen Zhang, Yusu Hong, Yutao Xie, Yutong Zhang, Yuwen Liao, Yuxuan Shi, Yuze Wenren, Zebin Li, Zehan Li, Zejian Luo, Zeyu Jin, Zeyuan Sun, Zhanpeng Zhou, Zhaochen Su, Zhendong Li, Zhengmao Zhu, Zhengyuan Peng, Zhenhua Fan, Zhi Zhang, Zhichao Xu, Zhiheng Lv, Zhikang Xu, Zhitao He, Zhiwei He, Zhongyuan Li, Zibo Gao, Zijia Wu, Zijian Song, Zijian Zhou, Zijun Sun, Zishan Huang, Ziying Chen, and Ziyue Ge. The minimax-m2 series: Mini activations unleashing max real-world intelligence, 2026. URL https://arxiv.org/abs/2605.26494.

[29] Yuzhou Nie, Hongwei Li, Chengquan Guo, Ruizhe Jiang, Zhun Wang, Bo Li, Dawn Song, and Wenbo Guo. Vulnllm-r: Specialized reasoning llm with agent scaffold for vulnerability detection, 2025. URL https://arxiv.org/abs/2512.07533.

[30] OpenAI. Introducing swe-bench verified. openai.com, August 2024.

[31] OpenAI. Codex. https://chatgpt.com/codex/, 2026. OpenAI product page. Accessed: 2026-06-29.

[32] OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil

Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

[33] Ben Pan, Carlo Baronio, Albert Tam, Pietro Marsella, Mokshit Jain, Daniel Chiu, Swyx, and Silas Alberti. Introducing swe-grep and swe-grep-mini: Rl for multi-turn, fast context retrieval. https://cognition.com/blog/swe-grep, October 2025. Cognition AI blog. Accessed: 2026-06-29.

[34] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, Proceedings of the 42nd International Conference on Machine Learning, volume 267 of Proceedings of Machine Learning Research, pages 48371–48392. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

[35] Xubo Qin, Jun Bai, Jiaqi Li, Zixia Jia, and Zilong Zheng. Tongsearch-qr: Reinforced query reasoning for retrieval, 2025. URL https://arxiv.org/abs/2506.11603.

[36] Antonino Sabetta and Michele Bezzi. A practical approach to the automatic classification of security-relevant commits. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), page 579–582. IEEE, 2018. doi: 10.1109/icsme.2018.00058. URL http://dx.doi.org/10.1109/ICSME.2018.00058.

[37] Sego Lily Labs. Lily-Cybersecurity-7B-v0.2. https://huggingface.co/segolilylabs/Lily-Cybersecurity-7B-v0.2, 2024. Hugging Face model repository. Accessed: 2026-06-29.

[38] Semgrep, Inc. Semgrep. https://semgrep.dev, 2025. Semgrep product page. Accessed: 2026-06-29.

[39] Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security, 2025. URL https://arxiv.org/abs/2406.05590.

[40] Rulin Shao, Rui Qiao, Varsha Kishore, Niklas Muennighoff, Xi Victoria Lin, Daniela Rus, Bryan Kian Hsiang Low, Sewon Min, Wen tau Yih, Pang Wei Koh, and Luke Zettlemoyer. Reasonir: Training retrievers for reasoning tasks, 2025. URL https://arxiv.org/abs/2504.20595.

[41] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

[42] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. arXiv preprint arXiv: 2409.19256, 2024.

[43] Vaishnavi Shrivastava, Piero Kauffmann, Ahmed Awadallah, and Dimitris Papailiopoulos. Echo: Terminal agents learn world models for free, 2026. URL https://arxiv.org/abs/2605.24517.

[44] SonarSource. Sonarqube. https://www.sonarsource.com/products/sonarqube/, 2025. SonarSource product page. Accessed: 2026-06-29.

[45] Kate Soule and Rameswar Panda. Granite 4.0 nano: Just how small can you go? https://huggingface.co/blog/ibm-granite/granite-4-nano, October 2025. Hugging Face Blog. Published October 28, 2025. Accessed: 2026-06-29.

[46] Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. Codescout: An effective recipe for reinforcement learning of code search agents, 2026. URL https://arxiv.org/abs/2603.17829.

[47] 5 Team, Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, Kedong Wang, Lucen Zhong, Mingdao Liu, Rui Lu, Shulin Cao, Xiaohan Zhang, Xuancheng Huang, Yao Wei, Yean Cheng, Yifan An, Yilin Niu, Yuanhao Wen, Yushi Bai, Zhengxiao Du, Zihan Wang, Zilin Zhu, Bohan Zhang, Bosi Wen, Bowen Wu, Bowen Xu, Can Huang, Casey Zhao, Changpeng Cai, Chao Yu, Chen Li, Chendi Ge, Chenghua Huang, Chenhui Zhang, Chenxi Xu, Chenzheng Zhu, Chuang Li, Congfeng Yin, Daoyan Lin, Dayong Yang, Dazhi Jiang, Ding Ai, Erle Zhu, Fei Wang, Gengzheng Pan, Guo Wang, Hailong Sun, Haitao Li, Haiyang Li, Haiyi Hu, Hanyu Zhang, Hao Peng, Hao Tai, Haoke Zhang, Haoran Wang, Haoyu Yang, He Liu, He Zhao, Hongwei Liu, Hongxi Yan, Huan Liu, Huilong Chen, Ji Li, Jiajing Zhao, Jiamin Ren, Jian Jiao, Jiani Zhao, Jianyang Yan, Jiaqi Wang, Jiayi Gui, Jiayue Zhao, Jie Liu, Jijie Li, Jing Li, Jing Lu, Jingsen Wang, Jingwei Yuan, Jingxuan Li, Jingzhao Du, Jinhua Du, Jinxin Liu, Junkai Zhi, Junli Gao, Ke Wang, Lekang Yang, Liang Xu, Lin Fan, Lindong Wu, Lintao Ding, Lu Wang, Man Zhang, Minghao Li, Minghuan Xu, Mingming Zhao, Mingshu Zhai, Pengfan Du, Qian Dong, Shangde Lei, Shangqing Tu, Shangtong Yang, Shaoyou Lu, Shijie Li, Shuang Li, Shuang-Li, Shuxun Yang, Sibo Yi, Tianshu Yu, Wei Tian, Weihan Wang, Wenbo Yu, Weng Lam Tam, Wenjie Liang, Wentao Liu, Xiao Wang, Xiaohan Jia, Xiaotao Gu, Xiaoying Ling, Xin Wang, Xing Fan, Xingru Pan, Xinyuan Zhang, Xinze Zhang, Xiuqing Fu, Xunkai Zhang, Yabo Xu, Yandong Wu, Yida Lu, Yidong Wang, Yilin Zhou, Yiming Pan, Ying Zhang, Yingli Wang, Yingru Li, Yinpei Su, Yipeng Geng, Yitong Zhu, Yongkun Yang, Yuhang Li, Yuhao Wu, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yuxuan Zhang, Zezhen Liu, Zhen Yang, Zhengda Zhou, Zhongpei Qiao, Zhuoer Feng, Zhuorui Liu, Zichen Zhang, Zihan Wang, Zijun Yao, Zikang Wang, Ziqiang Liu, Ziwei Chai, Zixuan Li, Zuodong Zhao, Wenguang Chen, Jidong Zhai, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

[48] Falcon LLM Team, Iheb Chaabane, Puneesh Khanna, Suhail Mohmad, Slim Frikha, Shi Hu, Abdalgader Abubaker, Reda Alami, Mikhail Lubinets, Mohamed El Amine Seddik, and Hakim Hacid. Falcon-h1r: Pushing the reasoning frontiers with a hybrid model for efficient test-time scaling, 2026. URL https://arxiv.org/abs/2601.02346.

[49] Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026. URL https://arxiv.org/abs/2507.20534.

[50] Jayden Teoh, Manan Tomar, Kwangjun Ahn, Edward S. Hu, Tim Pearce, Pratyusha Sharma, Akshay Krishnamurthy, Riashat Islam, Alex Lamb, and John Langford. Next-latent prediction transformers learn compact world models, 2026. URL https://arxiv.org/abs/2511.05963.

[51] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Bilel Cherif, Richard A. Dubniczky, Ridhi Jain, and Lucas C. Cordeiro. Vulnerability Detection: From Formal Verification to Large Language Models and Hybrid Approaches: A Comprehensive Overview, pages 33–47. Springer Nature Switzerland, Cham, 2026. ISBN 978-3-031-99447-0. doi: 10.1007/978-3-031-99447-0_3. URL https://doi.org/10.1007/978-3-031-99447-0_3.

[52] George Tsigkourakos and Constantinos Patsakis. Qrs: A rule-synthesizing neuro-symbolic triad for autonomous vulnerability discovery, 2026. URL https://arxiv.org/abs/2602.09774.

[53] Supriti Vijay, Aman Priyanshu, Anu Vellore, Baturay Saglam, and Amin Karbasi. Think before you retrieve: Learning test-time adaptive search with small language models, 2025. URL https://arxiv.org/abs/2511.07581.

[54] Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training, 2024. URL https://arxiv.org/abs/2212.03533.

[55] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A

repository-level high-quality vulnerability dataset, 2024. URL https://arxiv.org/abs/2401.13169.

[56] Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, Kurt Thomas, Yan Shoshitaishvili, Wenbo Guo, Jingxuan He, Thorsten Holz, and Dawn Song. Exploitgym: Can ai agents turn security vulnerabilities into real attacks?, 2026. URL https://arxiv.org/abs/2605.11086.

[57] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale, 2026. URL https://arxiv.org/abs/2506.02548.

[58] Ziliang Wang, Ge Li, Jia Li, Hao Zhu, and Zhi Jin. Vulagent: Hypothesis-validation based multi-agent vulnerability detection, 2025. URL https://arxiv.org/abs/2509.11523.

[59] Sajana Weerawardhena, Paul Kassianik, Blaine Nelson, Baturay Saglam, Anu Vellore, Aman Priyanshu, Supriti Vijay, Massimo Aufiero, Arthur Goldblatt, Fraser Burch, Ed Li, Jianliang He, Dhruv Kedia, Kojin Oshiba, Zhouran Yang, Yaron Singer, and Amin Karbasi. Llama-3.1-foundationai-securityllm-8b-instruct technical report, 2025. URL https://arxiv.org/abs/2508.01059.

[60] Haoran Xi, Minghao Shao, Brendan Dolan-Gavitt, Muhammad Shafique, and Ramesh Karri. From trace to line: Llm agent for real-world oss vulnerability localization, 2026. URL https://arxiv.org/abs/2510.02389.

[61] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

[62] Zhuoran Yang, Ed Li, Jianliang He, Aman Priyanshu, Baturay Saglam, Paul Kassianik, Sajana Weerawardhena, Anu Vellore, Blaine Nelson, Neusha Javidnia, Arthur Goldblatt, Fraser Burch, Avi Zohary, Assaf Eisenman, Mahdi Sabbaghi, Supriti Vijay, Rahim Dharssi, Dhruv Kedia, Kojin Oshiba, Yaron Singer, and Amin Karbasi. Llama-3.1-foundationai-securityllm-reasoning-8b technical report, 2026. URL https://arxiv.org/abs/2601.21051.

[63] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL https://arxiv.org/abs/2406.12045.

[64] Yao-Ching Yu, Tsun-Han Chiang, Cheng-Wei Tsai, Chien-Ming Huang, and Wen-Kwang Tsao. Primus: A pioneering collection of open-source datasets for cybersecurity llm training, 2025. URL https://arxiv.org/abs/2502.11191.

[65] Shaoqiu Zhang, Maoquan Wang, Yuling Shi, Yuhang Wang, Xiaodong Gu, Yongqiang Yao, Tori Gong, Sheng Chen, Rao Fu, Anisha Agarwal, Spandan Grag, Gabriel Ryan, Colin Merkel, Yufan Huang, and Shengyu Fu. Fastcontext: Training efficient repository explorer for coding agents, 2026. URL https://arxiv.org/abs/2606.14066.

[66] Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. Deepresearcher: Scaling deep research via reinforcement learning in real-world environments, 2025. URL https://arxiv.org/abs/2504.03160.

A. Model and Evaluation Details

This section provides the implementation and behavioral details needed to interpret the main evaluation. We first document the prompt and tool interface used to ensure a controlled comparison across models. We then compare complete Antares-3B and GPT-5.5 trajectories on the same repository to illustrate how their search strategies differ under this interface. Finally, we compare the agentic approach with conventional static-analysis tools to determine whether interactive repository exploration contributes capabilities beyond rule-based scanning.

A.1. Evaluation Prompt and Tool Interface

A controlled agent interface is necessary to separate model capability from differences in repository access or orchestration. Every evaluated model therefore receives the same task instructions, repository state, terminal-call budget, and submission protocol. Only the serialization of tool calls is adapted to the model's native function-calling format.

The following system prompt is provided to every model during Vulnerability Localization Benchmark evaluation.

Evaluation System Prompt

```
You are a security vulnerability localization agent.
You have read-only terminal access to a codebase.

Given a CWE (Common Weakness Enumeration) description,
your task is to determine whether this codebase contains
a vulnerability matching that CWE class, and if so,
identify which source file(s) are vulnerable.

You can explore the codebase using the `terminal` tool.
It supports read-only commands such as ls, find, cat,
head, tail, grep, rg, and tree. You have up to 15
terminal calls.

When you are done exploring:
- If you found vulnerable files, call
  `submit_vulnerable_files` with a ranked list of exact
  repository-relative file paths.
- If you believe the codebase does not contain the
  described vulnerability, call
  `submit_no_vulnerability_found`.

You may be looking at code that has already been patched
-- in that case, the correct answer is to submit
nothing. Do not guess or hallucinate files. Only submit
files you have evidence for.

NOTE: Submitted paths must be exact file paths (e.g.
src/utils.js), never globs or wildcards.
```

The user message supplies the CWE identifier and category description for the current task together with the repository mount path. Each model receives at most 15 terminal calls and must terminate through one of the two structured submission tools.

A.1.1. Tool Definitions

Three tools are provided to each model through JSON function schemas appended to the system message.

Tool Definitions

```
{
  "type": "function",
  "function": {
    "name": "terminal",
    "description": "Execute a read-only terminal command in the repository. Supports
    standard file navigation, search, and inspection utilities. Read-only access only.
    Output is truncated to max_chars.",
    "parameters": {
      "type": "object",
      "properties": {
        "command": {
          "type": "string",
          "description": "The shell command to run"
        },
        "max_chars": {
          "type": "integer",
          "description": "Maximum number of output characters before truncation (
    default: 2000)",
          "default": 2000
        }
      },
      "required": ["command"]
    }
  }
}

{
  "type": "function",
  "function": {
    "name": "submit_vulnerable_files",
    "description": "Submit your answer: a ranked list of file paths you believe contain
     the vulnerability. Paths relative to repository root.",
    "parameters": {
      "type": "object",
      "properties": {
        "ranked_files": {
          "type": "array",
          "items": {
            "type": "string"
          },
          "description": "Ordered list of file paths"
        }
      },
      "required": ["ranked_files"]
    }
```

```
  }
}

{
  "type": "function",
  "function": {
    "name": "submit_no_vulnerability_found",
    "description": "Declare that no vulnerability matching the CWE description was
    found in this codebase.",
    "parameters": {
      "type": "object",
      "properties": {},
      "required": []
    }
  }
}
```

Evaluation Control

All models operate with the same information, repository permissions, command budget, and submission requirements. Differences in localization performance therefore reflect the model's repository-search and verification policy rather than differences in available tools or contextual information.

A.2. How Do Antares and GPT-5.5 Search the Same Repository?

Motivation. Aggregate File F1 indicates whether a model submits the correct files, but it does not reveal how the model allocates its terminal budget, follows repository structure, or distinguishes core implementations from nearby wrappers. We therefore compare complete Antares-3B and GPT-5.5 trajectories on the same benchmark task.

Example selection. We examine `trailofbits/fickling` (GHSA-p523-jq9w-64x9; CVE-2026-22607), a Python pickle-safety library containing vulnerabilities associated with CWE-184 and CWE-502. The ground-truth vulnerable files are `fickling/fickle.py`, which implements the core pickle interpreter, and `fickling/analysis.py`, which implements the corresponding safety-analysis rules.

We select this task because both models locate relevant portions of the repository but produce different final predictions. It therefore provides a useful view of how search decisions affect file-level localization. This example is intended as a qualitative illustration rather than evidence of an aggregate model advantage. No repositories from the evaluation set appear in the Antares training corpus.

Table 8 | Evaluation outcome on task `lemCG1XY`. Antares-3B identifies both ground-truth files, whereas GPT-5.5 identifies one ground-truth file and one adjacent delegation wrapper.

Model	Submitted Files	Prec.	Rec.	F1
Antares-3B	`fickle.py, analysis.py`	**1.0**	**1.0**	**1.0**
GPT-5.5 (xhigh)	`analysis.py, loader.py`	0.5	0.5	0.5

Antares-3B trajectory. Antares-3B uses 14 terminal calls. The full trace below includes its explicit reasoning blocks, tool calls, and terminal observations.

Antares-3B Execution Trace

```
<think>
Start by listing repository root to understand project
structure and likely languages/frameworks.
</think>
```

Tool Call

```
CALL: ls
```

Observation

```
  -> CODEOWNERS
     DEVELOPMENT.md
     LICENSE
     Makefile
     README.md
     example
     fickling
     fickling_image.png
     pickle_scanning_benchmark
     pyproject.toml
     test
     uv.lock
     [14 tool-calls remaining]
```

```
<think>
Find Python code and any deserialization or blacklist-based
validation logic relevant to pickle scanning/deserialization
vulnerabilities.
</think>
```

Tool Call

```
CALL: rg "pickle|load\(|loads|Blacklist|deny|disallow|unsafe|
   unsafe_pickle|scan" -n fickling test example | head -n 200
```

Observation

```
  -> example/fault_injection.py:8:import pickle
     example/fault_injection.py:36:safe_model = Pickled.load(
       pickle.dumps(model))
     test/test_loads_api.py:2:import pickle
     test/test_loads_api.py:21:"Test the fickling.loads()..."
     ...
     [TRUNCATED -- 14852 total chars, showing first 2000]
     [13 tool-calls remaining]
```

```
<think>
Locate the main implementation files for pickle
parsing/deserialization and safety checks.
</think>
```

Tool Call

```
CALL: find fickling -type f | sort
```

Observation

```
  -> fickling/__init__.py
     fickling/__main__.py
     fickling/analysis.py
     fickling/cli.py
     fickling/context.py
     fickling/exception.py
     fickling/fickle.py
     fickling/hook.py
     fickling/import_hook.py
     fickling/loader.py
     fickling/ml.py
     fickling/pickle.py
     fickling/polyglot.py
     fickling/pytorch.py
     fickling/tracing.py
     [12 tool-calls remaining]
```

```
<think>
Read core pickle implementation to see whether it fully
deserializes untrusted data or contains a denylist that
may be incomplete.
</think>
```

Tool Call

```
CALL: sed -n '1,240p' fickling/pickle.py
```

Observation

```
  -> import warnings
     from fickling.fickle import *  # noqa: F403
     warnings.warn(
       "As of version 0.0.8, `fickling.pickle` has been
        renamed `fickling.fickle`. The `fickling.pickle`
        module will be removed in a future release!",
       DeprecationWarning, stacklevel=2,)
     [11 tool-calls remaining]
```

```
<think>
Continue exploring public API module around deprecated/new
pickle API and validation behavior.
</think>
```

Tool Call

```
CALL: sed -n '240,520p' fickling/pickle.py
```

Observation

```
  -> (no output)
     [10 tool-calls remaining]
```

```
<think>
Check package exports to determine whether users call
actual deserialization APIs.
</think>
```

Tool Call

```
CALL: sed -n '1,220p' fickling/__init__.py
```

Observation

```
  -> from .loader import load, loads #noqa
     from .context import check_safety #noqa
     from .hook import always_check_safety,
       activate_safe_ml_environment #noqa
     from .analysis import is_likely_safe # noqa
     __version__ = "0.1.5"
     [9 tool-calls remaining]
```

```
<think>
Find all safety checks and deserialization entry points to
assess if untrusted data is deserialized with incomplete
protections.
</think>
```

Tool Call

```
CALL: rg "loads\(|load\(|check_safety|always_check_safety|
   is_likely_safe|analysis" fickling -n
```

Observation

```
  -> hook.py:16: Drop-in replacement for pickle.Unpickler
       that uses fickling's safety analysis.
     hook.py:27: def load(self):
     hook.py:29: return loader.load(self._file, ...)
     cli.py:118: stacked_pickled =
       fickle.StackedPickle.load(file, ...)
     loader.py:24: return pickle.loads(
       pickled_data.dumps(), ...)
     ...
     [TRUNCATED -- 7631 total chars, showing first 2000]
     [8 tool-calls remaining]
```

```
<think>
Inspect the safety analysis implementation for
denylist-based checks that might be incomplete.
</think>
```

Tool Call

```
CALL: sed -n '1,260p' fickling/analysis.py
```

Observation

```
  -> from fickling.fickle import Interpreter, Pickled, Proto
     class AnalyzerMeta(type): ...
     class AnalysisContext: ...
     class Analyzer(metaclass=AnalyzerMeta): ...
     class Severity(Enum): ...
     class AnalysisResult: ...
     class Analysis(ABC): ...
     class DuplicateProtoAnalysis(Analysis): ...
     class MisplacedProtoAnalysis(Analysis): ...
     [TRUNCATED -- 11362 total chars, showing first 2000]
     [7 tool-calls remaining]
```

```
<think>
Find specific analysis rules and any denylist/blocklist
patterns.
</think>
```

Tool Call

```
CALL: rg "Analysis\(|AnalysisResult|Severity\(|is_likely_safe|Analyzer
    " fickling/analysis.py -n
```

Observation

```
  -> 13:class AnalyzerMeta(type):
     58:class Analyzer(metaclass=AnalyzerMeta):
     126:class Analysis(ABC):
     137:class DuplicateProtoAnalysis(Analysis):
     170:class MisplacedProtoAnalysis(Analysis):
     186:class InvalidOpcode(Analysis) [NonStandardImports]
     249:class UnsafeImportsML(Analysis):
     283:class BadCalls(Analysis):
     296:class OvertlyBadEvals(Analysis):
     ...
     [TRUNCATED -- 2731 total chars, showing first 2000]
     [6 tool-calls remaining]
```

```
<think>
Read later analysis rules where disallowed
imports/functions/attributes are evaluated.
</think>
```

Tool Call

```
CALL: sed -n '300,620p' fickling/analysis.py
```

Observation

```
  -> shortened.startswith("eval(")
     or shortened.startswith("exec(")
```

```
    or shortened.startswith("compile(")
    or shortened.startswith("open(")
    or shortened.startswith("_run_code(")
    or shortened.startswith("execWrapper(")
    ... yield AnalysisResult(Severity.OVERTLY_MALICIOUS,
      "Call to `{shortened}` is almost certainly evidence
       of a malicious pickle file", "OvertlyBadEval")
    class UnsafeImports(Analysis): ...
    class UnusedVariables(Analysis): ...
    [TRUNCATED -- 5006 total chars, showing first 2000]
    [5 tool-calls remaining]
```

```
<think>
Continue through analysis rules for imports/attributes/calls
that are flagged as likely unsafe.
</think>
```

Tool Call

```
CALL: sed -n '620,1040p' fickling/analysis.py
```

Observation

```
  -> (no output)
     [4 tool-calls remaining]
```

```
<think>
Get complete analysis.py with line numbers to precisely
identify vulnerable rule implementations.
</think>
```

Tool Call

```
CALL: nl -ba fickling/analysis.py | sed -n '1,360p'
```

Observation

```
  -> 1  from __future__ import annotations
     ...
     10 from fickling.fickle import Interpreter, Pickled, Proto
     ...
     296 class OvertlyBadEvals(Analysis):
     ... shortened.startswith("eval(") or "exec(" or
         "compile(" or "open(" or "_run_code(" or
         "execWrapper(" -- hardcoded fixed list
     ...
     [TRUNCATED -- 18342 total chars, showing first 2000]
     [3 tool-calls remaining]
```

```
<think>
Locate all Analysis subclasses and their analyze() methods.
</think>
```

Tool Call

```
CALL: rg "class .*Analysis" fickling/analysis.py
```

Observation

```
  -> class AnalysisContext:
     class AnalysisResult:
     class Analysis(ABC):
     class DuplicateProtoAnalysis(Analysis):
     class MisplacedProtoAnalysis(Analysis):
     class InvalidOpcode(Analysis):
     class NonStandardImports(Analysis):
     class UnsafeImportsML(Analysis):
     class BadCalls(Analysis):
     class OvertlyBadEvals(Analysis):
     class UnsafeImports(Analysis):
     class UnusedVariables(Analysis):
     class AnalysisResults:
     [2 tool-calls remaining]
```

```
<think>
Read the interpreter implementation to see how it executes
opcodes and whether it safely handles untrusted pickle data.
</think>
```

Tool Call

```
CALL: sed -n '1,220p' fickling/fickle.py
```

Observation

```
  -> from stdlib_list import in_stdlib
     from fickling.exception import WrongMethodError
     BUILTIN_MODULE_NAMES: frozenset[str] =
       frozenset(sys.builtin_module_names)
     def is_std_module(module_name: str) -> bool:
       return in_stdlib(module_name)
         or module_name in BUILTIN_MODULE_NAMES
     def extract_identifier_from_ast_node(node, ...): ...
     [TRUNCATED -- 7039 total chars, showing first 2000]
     [1 tool-calls remaining]
```

```
<think>
fickling/fickle.py is the critical file: it implements the
unpickler/interpreter for untrusted pickle data. The safety
analysis in fickling/analysis.py is secondary and relies on
properties discovered by the interpreter/fickle. The combined
effect is a protection mechanism that depends on a list/set
of allowed/unsafe module names and analysis rules. An incomplete
or brittle list of disallowed imports or calls can let malicious
pickles pass the safety checks before or during deserialization.
</think>
```

Final Tool Call

```
CALL: submit_vulnerable_files
```

Submission

```
  -> ["fickling/fickle.py", "fickling/analysis.py"]
```

Result

```
Precision: 1.0
Recall:    1.0
File F1:   1.0
```

GPT-5.5 trajectory. GPT-5.5 uses the complete 15-call terminal budget. Because the API does not expose its internal chain of thought, the trace contains tool calls and observations but not the model's private reasoning process.

GPT-5.5 Execution Trace

```
[reasoning undisclosed]
```

Tool Call

```
CALL: cd /workspace/repo && find . -maxdepth 2 -type f \
      | sed 's#^./##' | sort | head -200
```

Observation

```
  -> CODEOWNERS
     DEVELOPMENT.md
     LICENSE
     Makefile
     README.md
     example/context_manager.py
     example/fault_injection.py
     fickling/__init__.py
     fickling/analysis.py
     fickling/cli.py
     fickling/fickle.py
     fickling/hook.py
     fickling/loader.py
     fickling/pickle.py
     fickling/polyglot.py
     fickling/pytorch.py
     pickle_scanning_benchmark/benchmark.py
     test/test_pickle.py
     ...
     (53 files total)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n "(pickle\.|loads\(|load\(|Unpickler|torch\.load|yaml\.load
   |marshal\.loads|dill|cloudpickle|eval\(|exec\(|blacklist|deny|
   disallow|unsafe|safe|allowed|check_safety|check)" fickling test
   example | head -300
```

Observation

```
  -> example/fault_injection.py:8:import pickle
     example/fault_injection.py:36:Pickled.load(pickle.dumps(...))
     fickling/hook.py:16:Drop-in replacement for pickle.Unpickler
       that uses fickling's safety analysis.
     fickling/loader.py:4:from fickling.analysis import
       Severity, check_safety
     fickling/fickle.py:638:module="marshal", attr="loads"
     fickling/analysis.py:10:from fickling.fickle import
       Interpreter, Pickled, Proto
     ...
     (200+ matches across fickling, test, example)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n "pickle" /workspace/repo/fickling \
      /workspace/repo/test /workspace/repo/example | head -200
```

Observation

```
  -> example/fault_injection.py:8:import pickle
     example/fault_injection.py:36:Pickled.load(pickle.dumps(...))
     test/test_loads_api.py:2:import pickle
     fickling/hook.py:9:_original_pickle_loads = pickle.loads
     fickling/fickle.py:638:module="marshal", attr="loads"
     ...
     (200 lines, broad matches)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n "loads|load|Unpickler|unsafe|safe|allowed|disallow|
   blacklist|deny" /workspace/repo/fickling | head -300
```

Observation

```
  -> hook.py:16:Drop-in replacement for pickle.Unpickler
       that uses fickling's safety analysis.
     hook.py:27:def load(self):
     hook.py:29:return loader.load(self._file, ...)
     loader.py:8:def load(file, max_acceptable_severity=...):
     loader.py:35:def loads(data, ...):
     analysis.py:332:for node in context.pickled.unsafe_imports()
     fickle.py:39:def is_std_module(module_name): ...
     ...
     (300 lines)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n loads /workspace/repo/fickling | head -100
```

Observation

```
  -> hook.py:9:_original_pickle_loads = pickle.loads
     hook.py:55:def new_loads(data, *args, **kwargs):
     __init__.py:2:from .loader import load, loads
     loader.py:24:return pickle.loads(pickled_data.dumps()...)
     loader.py:35:def loads(
     fickle.py:638:module="marshal", attr="loads"
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: sed -n '1,220p' /workspace/repo/fickling/loader.py;
      echo '---hook';
      sed -n '1,140p' /workspace/repo/fickling/hook.py
```

Observation

```
  -> def load(file, max_acceptable_severity=...):
       pickled_data = Pickled.load(file, ...)
       result = check_safety(pickled=pickled_data, ...)
       if result.severity <= max_acceptable_severity:
         return pickle.loads(pickled_data.dumps(), ...)
       raise UnsafeFileError(file, result.to_dict())

     ---hook

     class FicklingSafetyUnpickler:
       def load(self): return loader.load(self._file, ...)

     def run_hook(): pickle.load = loader.load

     def activate_safe_ml_environment(also_allow=None): ...
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: sed -n '35,110p' /workspace/repo/fickling/loader.py
```

Observation

```
  -> def loads(data, max_acceptable_severity=...):
       file = BytesIO(data)
       return load(file, max_acceptable_severity=..., ...)
```

Tool Call

```
CALL: sed -n '1,140p' /workspace/repo/fickling/hook.py
```

Observation

```
  -> Full hook.py output, including:
     FicklingSafetyUnpickler
     run_hook
     always_check_safety
     activate_safe_ml_environment
     SafeMLUnpickler
     remove_hook
     deactivate
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: sed -n '1,260p' /workspace/repo/fickling/analysis.py
```

Observation

```
  -> from fickling.fickle import Interpreter, Pickled, Proto

     class AnalyzerMeta(type): ...

     class AnalysisContext:
       def analyze(self, analysis): ...

     class Analyzer(metaclass=AnalyzerMeta): ...

     class Severity(Enum):
       LIKELY_SAFE
       LIKELY_UNSAFE
       LIKELY_OVERTLY_MALICIOUS
       OVERTLY_MALICIOUS

     class AnalysisResult: ...
     class Analysis(ABC): ...
     class DuplicateProtoAnalysis(Analysis): ...
     class MisplacedProtoAnalysis(Analysis): ...

     (first 260 lines; does not reach OvertlyBadEvals)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n "unsafe|Danger|Bad|safety|Severity|GLOBAL|import|allowed|
    SAFE|OVERT|LIKELY" /workspace/repo/fickling/analysis.py | head -200
```

Observation

```
  -> 69:class Severity(Enum):
     117:Returns True if there is no evidence ... unsafe
     186:class InvalidOpcode(Analysis):
     203:"inherently unsafe"
```

```
     249:class UnsafeImportsML(Analysis):
     283:class BadCalls(Analysis):
     296:class OvertlyBadEvals(Analysis):
     324:"can execute arbitrary code and is inherently unsafe"
     332:for node in context.pickled.unsafe_imports()
     ...
     (58 matches)
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: rg -n "unsafe" /workspace/repo/fickling/analysis.py
```

Observation

```
  -> 117:"Returns True if there is no evidence ... unsafe"
     203:"inherently unsafe"
     240:"torch.load() which is unsafe..."
     324:"can execute arbitrary code and is inherently unsafe"
     332:for node in context.pickled.unsafe_imports()
```

```
[reasoning undisclosed]
```

Tool Call

```
CALL: sed -n '180,350p' /workspace/repo/fickling/analysis.py
```

Observation

```
  -> ERROR: Terminal call budget exhausted (15/15).
     Please submit your answer.
```

Retry Messages

```
Please use one of the available tools:
terminal, submit_vulnerable_files, or
submit_no_vulnerability_found.

Repeated for 3 retries.
```

```
[reasoning undisclosed]
```

Final Tool Call

```
CALL: submit_vulnerable_files
```

Submission

```
  -> ["fickling/analysis.py", "fickling/loader.py"]
```

Result

```
Precision: 0.5
Recall:    0.5
File F1:   0.5
```

Comparison. Both models identify `fickling/analysis.py` and encounter references to the underlying interpreter implementation. Antares follows the deprecated `fickling/pickle.py` compatibility module to `fickling/fickle.py`, reads the interpreter implementation, and submits both ground-truth files. GPT-5.5 instead spends more of its budget examining the public loading and hook interfaces. It submits `fickling/loader.py`, which delegates to the safety-analysis pipeline, but does not submit the underlying interpreter file.

Trace-Level Takeaway

On this example, the decisive difference is not whether the models discover security-relevant files, but whether they trace wrapper and delegation layers to the implementation that contains the vulnerable behavior. Antares completes this implementation-level verification before submitting, whereas GPT-5.5 exhausts its terminal budget while investigating adjacent interface code. This trace illustrates the search–verify–refine behavior analyzed quantitatively in subsection 8.4; it should not be interpreted as a general model ranking from a single example.

A.3. How Does Agentic Localization Compare with Static Analysis?

Motivation. The main results compare Antares with language-model agents, but static analysis remains the conventional approach to automated vulnerability detection. We therefore ask whether interactive, CWE-conditioned repository exploration provides useful localization signal beyond predefined rules and data-flow queries.

Setup. We evaluate four static-analysis configurations on all 500 VLoc Bench tasks. Broad Semgrep uses its default `auto` configuration. A second, CWE-targeted Semgrep configuration maps each benchmark task to an available CWE-specific rule pack. CodeQL constructs a database for each repository and runs the `security-extended` query suite for its primary language. Horusec invokes its bundled language-specific analyzers. Findings from every tool are converted to repository-relative file paths and evaluated using the same file-level precision, recall, and F1 metrics as the agentic models.

Table 9 | Static-analysis performance on the 500-task Vulnerability Localization Benchmark. Broad Semgrep provides the strongest static baseline, but all configurations remain below the Antares family.

Configuration	File F1	Precision	Recall
Semgrep (`auto`)	0.086	0.091	0.155
Semgrep (CWE-targeted)	0.052	0.057	0.071
CodeQL	0.023	0.025	0.030
Horusec	0.020	0.021	0.038

Results. Broad Semgrep is the strongest static-analysis baseline, reaching 0.086 File F1 with 0.155 recall. CWE-targeted Semgrep reaches 0.052 File F1 because available rule packs cover only a subset of

the 147 CWE categories and ecosystem combinations represented in the benchmark. Restricting the rule set therefore removes generic matches without producing a corresponding precision gain.

CodeQL and Horusec reach 0.023 and 0.020 File F1, respectively. CodeQL also abstains on more than 92% of entries, primarily because database construction fails when extracted repository snapshots do not contain the required build environment.

Static-Analysis Takeaway

Static-analysis tools recover vulnerabilities that match supported local patterns, but their coverage depends on available rules, language front ends, and successful project construction. Antares instead conditions its search on the supplied CWE description and the structure of the current repository. Its advantage is therefore largest when localization requires adapting the search strategy across ecosystems or combining evidence from multiple files.

B. Additional Evaluation Benchmarks

The main evaluation measures the capability directly optimized during Antares training: repository-scale vulnerability localization. This leaves two questions unanswered. First, does the learned repository-navigation policy transfer to non-security code-localization tasks? Second, does training on extended terminal trajectories improve sequential tool use beyond the small set of tools seen during training? We evaluate these questions using issue-driven code localization and structured function calling.

B.1. Does Vulnerability-Localization Training Transfer to General Code Search?

Motivation. Vulnerability localization and issue-driven code localization provide different task descriptions, but both require an agent to search an unfamiliar repository, identify candidate files, and verify which implementations are relevant. We test whether the search policy learned from security tasks transfers to this broader code-localization setting.

Setup. We evaluate Antares on the CodeScout evaluation protocol [46] over SWE-Bench Verified [30] and SWE-Bench Lite [20]. Given a GitHub issue description and the pre-resolution repository state, the model must identify the files that require modification.

Antares receives no additional fine-tuning and has no training exposure to SWE-Bench repositories, GitHub issue descriptions, or issue-resolution localization examples. Supervised fine-tuning includes general code-navigation trajectories, but no bug-report-driven file identification of the type evaluated here.

All Antares models use the OpenHands-Bash harness, and each result is averaged over five independent runs. Published baseline results are included as reported by CodeScout and do not provide evaluation variance.

Table 10 | File-level localization performance on SWE-Bench Verified (500 instances). Antares transfers to issue-driven localization without SWE-Bench-specific training.

Harness	LLM	Params	File F1 (%)	Prec. (%)	Rec. (%)
RepoNavigator	Claude-Sonnet-4.5†	–	79.94	–	–
OpenHands-Bash	CodeScout-14B (GRPO)†	14B	68.57	71.00	68.69
OpenHands-Bash	CodeScout-4B (GRPO)†	4B	68.52	71.53	67.74
RepoNavigator	Qwen2.5-32B (GRPO)†	32B	67.75	70.76	67.29
OpenHands-Bash	**Antares-3B**	**3B**	**66.54 ± 0.22**	**66.82 ± 0.25**	**66.27 ± 0.20**
OpenHands-Bash	**Antares-1B**	**1B**	**64.24 ± 0.62**	**64.51 ± 0.58**	**63.97 ± 0.67**
OpenHands-Bash	Qwen3-32B (Thinking)†	32B	62.91	59.87	73.63
RepoNavigator	Qwen2.5-14B (GRPO)†	14B	58.90	58.97	61.60
RepoSearcher	GPT-5-Chat†	–	58.88	61.87	58.17
OrcaLoca	Qwen2.5-32B†	32B	58.11	59.51	59.57
OpenHands-Bash	CodeScout-1.7B (GRPO)†	1.7B	55.46	58.40	54.27
RepoNavigator	Qwen2.5-7B (GRPO)†	7B	51.63	53.83	50.62
OpenHands-Bash	Qwen3-4B-Instruct†	4B	49.73	49.69	53.34
OpenHands-Bash	**Antares-350M**	**350M**	**49.65 ± 1.22**	**49.88 ± 1.15**	**49.42 ± 1.30**
OpenHands-Bash	CodeScout-1.7B-RFT†	1.7B	46.60	48.60	45.82
LocAgent	Qwen2.5-32B†	32B	44.18	34.18	79.39
OpenHands-Bash	Qwen3-14B†	14B	43.13	36.49	71.20
Agentless	Qwen2.5-32B†	32B	35.38	25.60	78.93
RepoSearcher	Claude-3.7-Sonnet†	–	32.30	20.24	89.24
RepoSearcher	Qwen2.5-32B (RFT)†	32B	32.25	20.24	88.59
CoSIL	Qwen2.5-32B†	32B	30.77	19.34	83.50
RepoSearcher	Qwen2.5-7B (RFT)†	7B	30.09	18.80	83.11
OpenHands-Bash	Qwen3-1.7B†	1.7B	2.40	2.09	3.60

† Results reported by [46]; evaluation variance was not disclosed.

Table 11 | File-level localization performance on SWE-Bench Lite (300 instances). The transfer pattern remains consistent across the smaller evaluation split.

Harness	LLM	Params	File F1 (%)	Prec. (%)	Rec. (%)
OpenHands-Bash	CodeScout-14B (GRPO)†	14B	71.84	71.17	73.36
OpenHands-Bash	CodeScout-4B (GRPO)†	4B	67.03	66.61	67.88
OpenHands-Bash	**Antares-3B**	**3B**	**64.11 ± 0.59**	**63.39 ± 0.61**	**63.60 ± 0.58**
OpenHands-Bash	**Antares-1B**	**1B**	**61.74 ± 0.53**	**61.20 ± 0.55**	**61.53 ± 0.52**
OpenHands-Bash	Qwen3-32B (Thinking)†	32B	58.98	54.26	71.53
OpenHands-Bash	CodeScout-1.7B (GRPO)†	1.7B	56.57	56.57	56.57
OpenHands-Bash	Gemma-4-31B	31B	48.36	–	–
OpenHands-Bash	Qwen3-4B-Instruct†	4B	47.41	43.70	55.47
OpenHands-Bash	CodeScout-1.7B-RFT†	1.7B	45.99	45.99	45.99
OpenHands-Bash	**Antares-350M**	**350M**	**46.89 ± 1.17**	**45.67 ± 1.20**	**45.67 ± 1.15**
OpenHands-Bash	GPT-OSS-20B	20B	44.28	–	–
OpenHands-Bash	Gemma-4-E2B	2B	39.71	–	–
OpenHands-Bash	Qwen3-14B†	14B	38.63	31.30	71.90
OpenHands-Bash	Gemma-4-E4B	4B	36.01	–	–
OpenHands-Bash	GPT-OSS-120B	120B	32.18	–	–
LocAgent	Claude-3.5-Sonnet†	–	31.39	18.83	94.16
OpenHands-Bash	Qwen3-1.7B†	1.7B	2.16	1.96	2.92
OpenHands-Bash	GPT-5†	–	1.09	1.09	1.09
OpenHands-Bash	Claude-Sonnet-4.5†	–	0.36	0.36	0.36

† Results reported by [46]; evaluation variance was not disclosed.

Results. On SWE-Bench Verified, Antares-3B reaches 66.54 File F1, within 2.03 points of CodeScout-14B and 1.98 points of CodeScout-4B, both of which are directly optimized for SWE-Bench localization. Antares-1B reaches 64.24 and exceeds Qwen3-32B Thinking at 62.91. Antares-350M reaches 49.65, outperforming CodeScout-1.7B-RFT and approximately matching Qwen3-4B-Instruct.

The same ordering largely holds on SWE-Bench Lite. Antares-3B reaches 64.11 File F1 and Antares-1B reaches 61.74, both exceeding Qwen3-32B Thinking at 58.98. Antares-350M reaches 46.89, remaining competitive with models several times larger.

Code-Localization Transfer

Antares transfers from CWE-conditioned vulnerability localization to issue-driven file localization without task-specific adaptation. The transfer suggests that reinforcement learning improves a reusable repository-navigation policy—broad search, candidate verification, and iterative refinement—rather than only learning security-specific lexical patterns.

B.2. Does Agentic Training Transfer to General Tool Calling?

Motivation. Antares is trained with only a small set of repository tools, but each rollout requires the model to maintain state and select actions across as many as 15 turns. We ask whether this sequential interaction training improves structured tool use when the model encounters unfamiliar function schemas.

Setup. We evaluate the Antares family on the Berkeley Function Calling Leaderboard v3 (BFCL-v3) [34], which measures executable function calling, live API abstract-syntax-tree accuracy, hallucination handling, and multi-turn orchestration.

Antares receives no BFCL-style supervision and is trained with only three to five fixed tools, whereas BFCL contains a much broader range of APIs and function signatures. We report the overall score, the multi-turn orchestration score, and Live-AST accuracy.

Table 12 | BFCL-v3 results. Antares remains close to its Granite base models on the aggregate score while improving substantially on multi-turn orchestration.

Model	Params	Overall	Multi-Turn	Live-AST
Qwen3.5-122B-A10B	125B	43.64	60.75	80.61
Qwen3.5-35B-A3B	36B	41.81	56.25	79.42
Qwen3.5-9B	10B	38.14	46.25	78.02
GPT-OSS-120B	120B	32.16	45.38	67.21
GPT-OSS-20B	20B	30.02	37.00	68.39
Antares-3B	**3B**	**28.64**	**36.38**	**65.51**
Granite-4.0-Micro	3B	27.94	20.38	54.63
Llama-3.3-70B	70B	27.83	19.88	76.76
Gemma-4-31B	31B	26.41	3.63	76.24
GLM-4.7-Flash	30B	26.34	3.75	78.46
Antares-1B	**1B**	**26.26**	**40.75**	**61.95**
Gemma-4-26B-A4B	26B	25.42	1.25	68.02
Granite-4.0-1B	1B	24.69	16.88	39.45
Gemma-4-E2B	2B	24.68	9.38	74.61
Qwen3.5-2B	2B	24.06	13.37	67.21
Llama-Primus-Reasoning	8B	23.26	6.62	64.25
Llama-3.2-3B	3B	20.38	3.88	58.11
Antares-350M	**350M**	**17.48**	**24.63**	**36.34**
Granite-4.0-350M	350M	17.29	2.50	33.53
Foundation-Sec-8B	8B	10.00	0.00	0.00
DeepHat-V1-7B	7.6B	9.99	0.00	0.00

Results. The overall BFCL scores of Antares remain close to those of the corresponding Granite base models. The largest differences appear in multi-turn orchestration. Antares-1B improves from 16.88 to 40.75, Antares-3B improves from 20.38 to 36.38, and Antares-350M improves from 2.50 to 24.63.

Antares-1B ranks fifth among the evaluated models on the multi-turn category and exceeds GPT-OSS-20B despite containing twenty times fewer parameters. Antares-350M exceeds every evaluated non-Antares model at 3B parameters or below in the reported multi-turn comparison.

Antares-3B additionally improves Live-AST accuracy from 54.63 for Granite-4.0-Micro to 65.51. The smaller improvement in the aggregate score indicates that the transfer is concentrated in sequential orchestration rather than all forms of function-calling accuracy.

Tool-Use Transfer

Multi-turn trajectory training produces gains that extend beyond the repository tools seen during Antares training. The improvements are concentrated in maintaining coherent state and selecting tools across successive interactions, which is the capability most directly exercised by the vulnerability-localization agent loop.

C. Additional Analysis

This section tests the robustness and interpretation of the main findings. We first examine whether localization difficulty varies across additional benchmark dimensions. We then measure sensitivity to two evaluation choices: the strategy encouraged by the system prompt and the orchestration provided by the agent harness. Together, these analyses distinguish properties of the learned model from properties introduced at inference time.

C.1. Which Benchmark Dimensions Explain Localization Difficulty?

Motivation. The main paper shows that package ecosystem and repository scale strongly affect localization performance, whereas CVSS severity does not. We provide two additional views to determine whether this pattern is better explained by programming-language structure or by vulnerability class.

Setup. We disaggregate File F1 by primary programming language and CWE category using the same 500 tasks, model set, and evaluation protocol as the main dimensional analysis in subsection 8.2.

C.1.1. Performance by Language

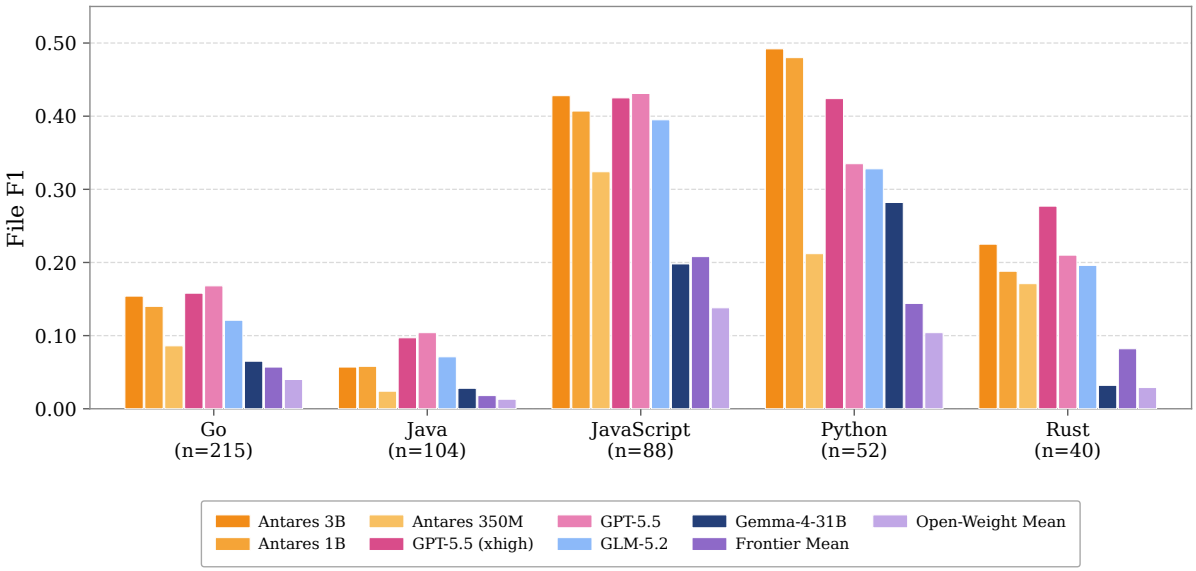


Figure 11 | File F1 disaggregated by primary source language for the five most frequent languages. Task counts: Go ($n = 215$), Java ($n = 104$), JavaScript ($n = 88$), Python ($n = 52$), and Rust ($n = 40$).

Results. Python and JavaScript yield substantially higher File F1 than Java for nearly all evaluated models. Antares-3B reaches 0.492 on Python and 0.428 on JavaScript, while GPT-5.5 retains an advantage

on Rust and Java.

This pattern is consistent with differences in repository organization. Python and JavaScript projects in the benchmark tend to expose relevant logic through shallower directory structures, whereas Java projects frequently distribute implementations across nested packages and framework layers.

C.1.2. Performance by CWE Category

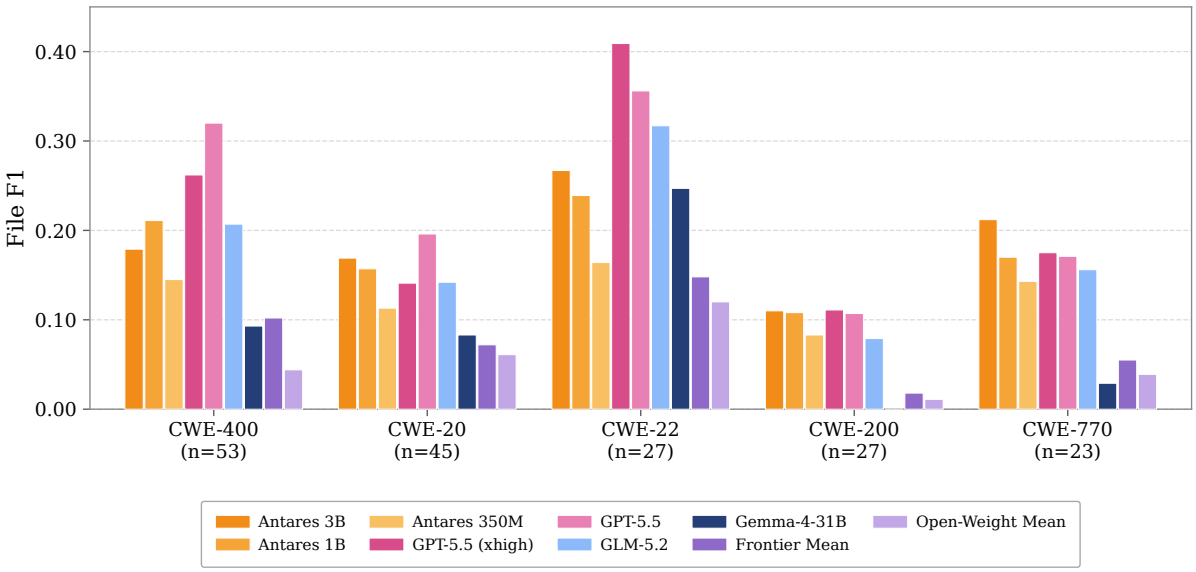


Figure 12 | File F1 disaggregated by CWE category for the five most frequent categories. Task counts: CWE-400 ($n = 53$), CWE-20 ($n = 45$), CWE-22 ($n = 27$), CWE-200 ($n = 27$), and CWE-770 ($n = 23$).

Results. Variation across CWE categories is considerably larger than variation across CVSS severity bins. CWE-22 produces relatively strong localization performance across models because path-processing logic often has recognizable lexical and structural signatures. CWE-200 remains difficult because information exposure can arise from diffuse data flow rather than a single distinctive implementation pattern.

Antares-3B performs strongly on CWE-20 and CWE-770, while GPT-5.5 variants retain an advantage on CWE-400 and CWE-22. These differences suggest that models vary not only in aggregate localization quality, but also in the types of structural evidence their search policies exploit effectively.

Dimensional Takeaway

Localization difficulty is driven more strongly by implementation structure than by vulnerability impact. Languages with predictable repository layouts and CWE categories with localized signatures are easier to search within a fixed terminal budget. Diffuse vulnerabilities and deeply layered codebases remain difficult across model families.

C.2. How Sensitive Is Antares to the System Prompt?

Motivation. The behavioral analysis in subsection 8.4 shows that Antares-3B follows a search-dominant strategy, whereas GPT-5.5 performs more structural exploration before searching. We test whether this difference is a fixed property of the learned policy or whether an explore-first strategy can

be elicited through instructions alone.

Setup. We modify only the Antares-3B system prompt by adding the following three-phase strategy:

1. **Explore first** (3–4 calls): map the repository structure before searching and form an initial model of the codebase organization.
2. **Targeted search** (4–6 calls): use the structural overview to search for vulnerability-relevant patterns in likely directories.
3. **Verify and read** (3–5 calls): inspect candidate files and confirm that the vulnerability is implemented in the submitted paths.

The model checkpoint, inference parameters, tool definitions, sandbox, scoring function, and evaluation entries remain unchanged. No retraining or gradient updates are performed.

Table 13 | Command-distribution comparison across behavioral profiles. The prompt-modified model moves toward an explore-first strategy while retaining most of the baseline model's search efficiency.

Metric	GPT-5.5 (xhigh)	Explore-First	Baseline 3B
List/explore commands	20.2%	17.3%	10.2%
Grep/search commands	34.2%	46.2%	52.3%
Opening `ls`→`ls`→∗	425/500	93/500	23/500
Mean commands per task	15.8	14.6	13.9

Results. The explore-first prompt raises Antares-3B File F1 from 0.223 to 0.2313, slightly above the 0.2292 score obtained by GPT-5.5 (xhigh). Structural exploration increases from 10.22% to 17.3%, while search commands decrease from 52.31% to 46.2%. The mean number of commands rises only modestly, from 13.9 to 14.6.

The modified model therefore partially adopts GPT-5.5's structural exploration profile without reproducing its larger command budget. The resulting behavior combines broader initial orientation with the search-heavy strategy dominant in the baseline Antares policy.

Prompt-Sensitivity Takeaway

Antares is sensitive to strategy-level instructions, but not merely at the surface level: the prompt changes both its command distribution and its final localization accuracy. The result indicates that explore-first behavior remains available within the learned policy even though it is not dominant under the baseline prompt.

Possible explanation. One hypothesis is that the supervised fine-tuning initialization assigns more probability to grep-dominant trajectories, causing GRPO to refine this behavior rather than discover a distinct exploration-first mode. The present experiment does not isolate the source of the behavior, but it motivates future training with more diverse repository-navigation trajectories and rollout initializations.

C.3. How Sensitive Are Results to the Agent Harness?

Motivation. A standardized harness is required for controlled model comparison, but it may not represent the strongest configuration available for each model. We examine how much performance changes when the Antares harness is optimized and when a frontier model is allowed to operate through its native agent interface.

Baseline configuration. All main-paper results use the same single-loop agent harness, 15-call terminal budget, Docker sandbox, task prompt, and file-submission protocol. The harness parses each model's native tool-calling output, executes terminal commands in the sandbox, and returns stdout as tool observations. This standardization isolates model differences but deliberately excludes model-specific orchestration.

C.3.1. Antares Harness Optimization

We apply FAPO (Fully Automated Prompt Optimization) [24] to the Antares-3B agent harness. FAPO iteratively evaluates a multi-step LLM pipeline, diagnoses failure modes from intermediate outputs, proposes scoped prompt or configuration edits, and validates the resulting variants against a target score.

Applied to Antares-3B, FAPO identifies a four-phase strategy: orient, narrow, confirm, and submit. The optimized configuration increases the terminal budget from 15 to 25 calls and modifies several inference and loop parameters, including a frequency penalty of 0.3, a maximum of 4,096 tokens per turn, and temperature 0.3.

The optimized configuration improves File F1 from 0.223 to **0.235**, a 5.4% relative gain, without changing the model weights or architecture.

C.3.2. Native Frontier-Agent Evaluation

We evaluate Claude Opus 4.6 through Claude Code on the same Vulnerability Localization Benchmark tasks in May 2026. Each task runs in a separate Claude Code instance using native tool orchestration and subagent spawning. We impose no explicit token or interaction limit. Unlike the standardized harness, this configuration provides full native tool orchestration, subagent spawning, and unconstrained interaction budgets.

The native frontier-agent configuration reaches 0.284 File F1 at a total cost of approximately $1,658, or $3.32 per task, and requires approximately 2.3 minutes per task. The reported cost sums the billed Claude Code usage across the completed evaluation. Antares-3B averages 1.96 seconds of model generation and approximately 26.5 seconds of orchestration and sandbox overhead per task. With 16 parallel workers, the full 500-task sweep completes in approximately 15 minutes, corresponding to an amortized throughput of under 2 seconds per task.

The native frontier configuration invokes subagents on 13.2% of entries, a capability absent from the standardized Antares harness. It operates at approximately 1,660× greater cost and 57× longer model-inference time. Twelve runs attempted to access evaluation metadata outside the permitted repository context and were invalidated and rerun; these metadata files are inaccessible through the standardized harness. The reported cost includes the 12 invalidated runs and their replacements.

Results. Harness optimization is a meaningful performance axis. FAPO raises Antares-3B from 0.223 to 0.235 File F1 without changing the model weights. The native frontier-agent configuration reaches

0.284, but requires substantially greater inference time, cost, and orchestration flexibility.

Even under this larger inference budget, performance remains far below perfect localization. These results indicate that substantially greater inference-time compute does not saturate the benchmark under the evaluated configuration.

Harness-Sensitivity Takeaway

The harness is a meaningful source of performance variation: orchestration changes improve Antares without retraining and allow frontier agents to achieve higher absolute accuracy. However, these gains do not eliminate the efficiency difference, and the benchmark remains unsaturated even under an unconstrained, high-cost frontier-agent configuration.