METAVUL: AGENTIC HARNESS DISCOVERY FOR VULNERABILITY DETECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM) agents are increasingly used to automate vulnerability detection, but reliable detection remains a challenging problem because agents must distinguish subtle security-relevant differences from benign code variation. While underlying LLM capability has improved significantly, agent accuracy also depends heavily on the *harness*: the code that defines the scaffolding around the LLM. Harnesses are typically fixed and largely hand-written, limiting adaptation to evaluator feedback. Additionally, harness improvements for one dataset or model may fail to transfer to others. To address these challenges, we introduce MetaVul, a self-improving framework for harness discovery in vulnerability detection. Starting from a seed structured reasoning scaffold, MetaVul uses a workflow-based optimization loop to iteratively refine the harness using feedback from paired vulnerable and patched functions, all while preserving the structured reasoning and verdict format. At each iteration, it searches over the full history of prior candidate harnesses, with access to their trajectories, logs, and metrics, and it returns a set of Pareto-optimal harnesses rather than a single best candidate. We apply MetaVul to discover harnesses for architectures including Codex CLI and Claude Code. A validation set (202 pairs) from TitanVul and CWE-Eval is used for optimization, while the unseen PrimeVul dataset (430 pairs) is kept as the held-out set. We later test whether harnesses learned from one model transfer to others not seen during optimization. Results on both frontier and small LLMs show that the optimized harnesses improve pairwise vulnerability detection up to 23.3 percentage points on the validation set (Opus 4.6) and up to 25.8 points on the held-out test set (DeepSeek V4 Flash). Harnesses learned with one model also transfer to others. We show that harnesses optimized with Opus 4.6 transfer to Opus 5, improving held-out pairwise accuracy by 14.7 points without the need for further optimization. Code is available at `https://github.com/BTP24VFnNd/submission`.

1 INTRODUCTION

Artificial intelligence (AI) agents utilize a large language model (LLM) as their core reasoning component for taking actions toward a given goal. "Vibe coding," or using AI tools, particularly by way of coding agents, is becoming increasingly prevalent across the software industry. Consequently, the growing volume and pace of AI-assisted code generation increase the importance of effective vulnerability detection. LLM agents rely not only on the capabilities of their underlying model, but also on the harnesses that structure how those agents reason, interact with tools, and arrive at a decision. Designing these harnesses, known as *harness engineering*, has become a practice of its own (Lopopolo, 2026; Rajasekaran, 2026). For vulnerability detection, these harnesses are typically hand-designed and are fixed during deployment, even though their structure and context can substantially alter an agent's results. In practice, vulnerability detection is substantially affected by how the agent inspects the code, and how the harness will shape that workflow (Yildiz et al., 2025). This understanding directly motivates a harness that can learn its design systematically through evaluation feedback.

Recent work studies the evaluation, evolution, and adaptation of agent harnesses using graded scores, verifiers, execution traces, or prior experience (Lee et al., 2026; Ursekar et al., 2026; Huang et al., 2026a; Zhang et al., 2026; Huang et al., 2026b; Esakkiraja et al., 2026; Nie et al., 2026), while OneDayAgent develops a fixed long-horizon harness (Zheng et al., 2026); however, none investigates

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

harness optimization for vulnerability detection or incorporates security-specific objectives such as CWE identification, vulnerability localization, and vulnerable–patched discrimination. Harness engineering for vulnerability detection, however, poses challenges. Firstly, **false alarms are costly.** In practical security settings, analysts receive far more alerts than they can effectively triage. Several industry studies identify alert fatigue and analyst burnout as major problems (Tariq et al., 2025). This problem is well documented among traditional vulnerability-detection tools (Charoenwet et al., 2024; Sajadi et al., 2026) and persists with recent LLM-based detectors (Li et al., 2026a). As recently as April 2026, an independent security company reported poor discriminative performance on PrimeVul (Ding et al., 2024)[1]; even Claude Opus 4.6, a flagship model shown by Anthropic to discover previously unknown vulnerabilities Carlini et al. (2026), struggled to reliably distinguish vulnerable functions from their patched counterparts (Walker, 2026). These findings independently corroborate our motivation and demonstrate that vulnerability discrimination remains an open and practically relevant problem. Importantly, they show that stronger underlying models do not eliminate the need to design the surrounding agentic workflow effectively. Rather, harnesses and their associated workflow choices matter, as different reasoning and verification configurations can yield vastly different behavior. Secondly, **harness optimization is non-trivial.** A vulnerability-detection harness must control how an agent inspects code, structures its analysis, and reasons about security. It is also multi-objective: a strong harness must accurately classify and localize vulnerabilities, identify their CWE, ground its conclusions in code evidence, and distinguish vulnerable functions from their patched counterparts. Ding et al. (2024) show why traditional classification metrics, such as accuracy and F1, are insufficient on their own: models that perform well on conventional benchmarks approach random guessing on PrimeVul's paired evaluation. Optimizing for a single metric may therefore reward superficial patterns rather than robust vulnerability reasoning. HarnessDev (Wu et al., 2026) shows that a harness can overfit to the model that wrote it, memorize development examples, or improve one capability while silently regressing another. It can also overfit its search tasks: Wang et al. (2026) find that gains observed when harness evolution and evaluation share the same tasks largely disappear on held-out tasks, suggesting that optimization can learn task-specific shortcuts rather than generalizable harness improvements.

We address these challenges with **MetaVul**, which iteratively improves vulnerability-detection harnesses using validation feedback and returns a Pareto frontier over discrimination and localization. A sandboxed detector evaluates each harness, while held-out datasets and cross-model evaluation test generalization. We ask whether feedback-driven search improves vulnerability-detection harnesses and whether the gains transfer to unseen data and models (**RQ1**), how MetaVul compares with other LLM-based or agentic methods (**RQ2**), and whether MetaVul's gains rely on data leakage, embedded answers, or trivial verdict bias (**RQ3**). MetaVul improves pairwise accuracy by 11.4 to 23.3 points ($1.6\times$–$4.3\times$) on validation and by up to 25.8 points ($2.9\times$) on 430 unseen PrimeVul pairs. Harnesses also transfer to models that took no part in optimization, with gains up to 14.7 points ($1.7\times$). MetaVul outperforms prior LLM-based detectors without evidence of shortcut exploitation.

2 Related Work

Previous work has explored LLM-based vulnerability detection, harness engineering and optimization, and structured reasoning strategies for code. **LLM-based vulnerability detection.** Ding et al. (2024) show that LLMs struggle to distinguish vulnerable functions from their patched versions under paired evaluation; chain-of-thought (CoT) prompting (Wei et al., 2022) improves pairwise performance over two-shot prompting but stays below random guessing. GPTLens (Hu et al., 2023) pairs an auditor that proposes candidate vulnerabilities with a critic that filters false positives, VulTrial (Widyasari et al., 2026) makes use of courtroom-style agents that debate a verdict, and ConfColl (Tsai et al., 2025) escalates from a single agent to retrieval-augmented and multi-agent reasoning when needed. All four methods use GPT-4o as the backbone (OpenAI, 2024; Tang et al., 2026), and we compare against them in Sec. 6. At project scale, Li et al. (2026a) report low recall, high false-discovery rates and high computational cost. **Harness engineering and optimization.** Recent work treats the agent harness (prompts, tools, memory, context management and control flow) as an optimization target independent of model weights (Lee et al., 2026; Ursekar et al., 2026; Lopopolo, 2026; Rajasekaran,

[1] Following PrimeVul, we adopt *pairwise accuracy* as a core metric alongside accuracy and F1. High pairwise accuracy indicates that the detector can distinguish vulnerable code from its fixed version, reducing false alerts and, consequently, alert fatigue. We explain this further in Sec. 4.2.

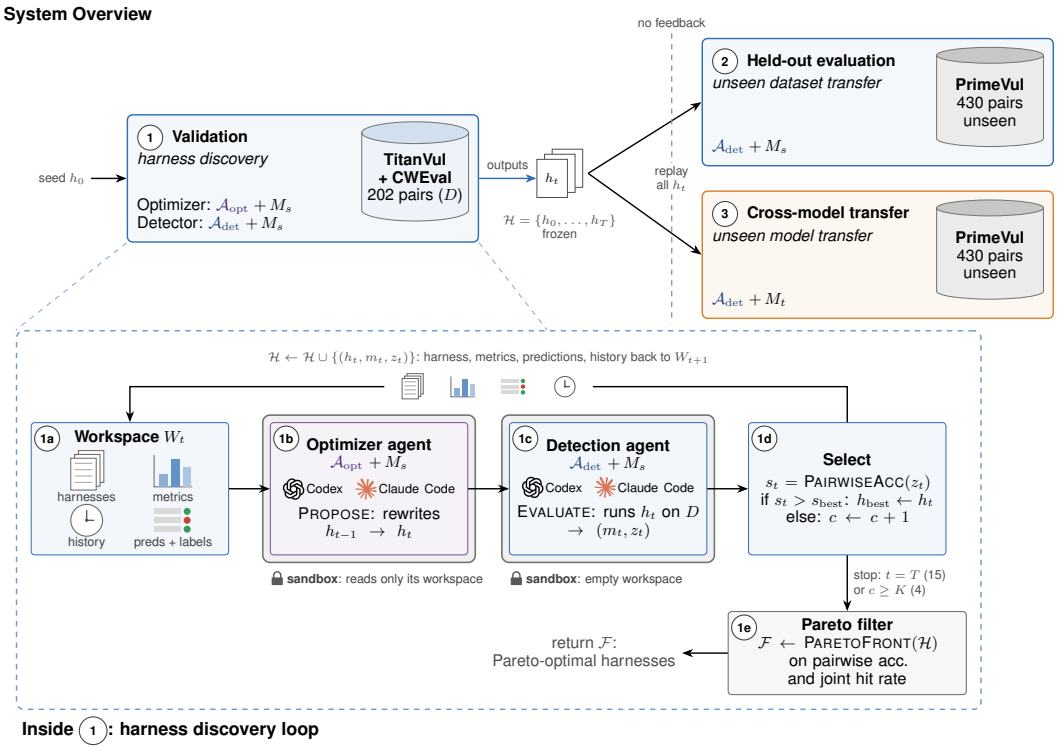


Figure 1: **MetaVul pipeline.** (1) Starting from seed h_0, harnesses are discovered on the validation set. In each iteration, (1a) the workspace stores previous harnesses and results, (1b) $\mathcal{A}_{\text{opt}}$ proposes a new harness h_t, (1c) $\mathcal{A}_{\text{det}}$ evaluates it, and (1d) the best result is selected. The Pareto filter (1e) retains harnesses based on pairwise accuracy and joint hit rate. Each harness is then frozen and tested on unseen PrimeVul using (2) the search model M_s and (3) a different transfer model M_t. PrimeVul provides no optimization feedback.

2026), in contrast to weight-level methods such as ORPO (Hong et al., 2024). GEPA (Agrawal et al., 2026) optimizes prompts from execution trajectories and model feedback, and SecureForge (Liu et al., 2026) optimizes system prompts for secure code generation using automated vulnerability auditing. Meta-Harness, MemoHarness, DarwinX, StarHarness, and TTHE explore complementary forms of harness adaptation using execution experience, search, held-out selection, and test-time adaptation (Lee et al., 2026; Huang et al., 2026b; Zhang et al., 2026; Esakkiraja et al., 2026; Nie et al., 2026), and OneDayAgent supports long-horizon execution with task decomposition, persistent memory and output verification (Zheng et al., 2026). HarnessOpt-Bench, Evo-Bench and HarnessDev evaluate whether models can build or improve harnesses that generalize beyond their optimization tasks (Ursekar et al., 2026; Huang et al., 2026a; Wu et al., 2026), and Wang et al. (2026) find that apparent gains may vanish on held-out tasks or reflect extra test-time search, underscoring the need for held-out evaluation. **Reasoning strategies for code.** Structured chain-of-thought (SCoT) organizes reasoning around program structures (Li et al., 2023), Modularization-of-Thought (MoT) decomposes it into hierarchical modules (Pan & Zhang, 2025), and Tree of Thoughts (ToT) explores multiple reasoning paths (Yao et al., 2023). Despite this, to our knowledge, none of these methods have explored harness optimization for vulnerability detection, nor tested whether such harnesses transfer to unseen datasets and models. MetaVul automatically optimizes harnesses for this task, selects them on security-specific objectives (telling vulnerable functions apart from their fixes, i.e., pairwise accuracy, and joint CWE and line localization), and evaluates them on an unseen dataset and on models that took no part in optimization.

3 METHODOLOGY

MetaVul separates harness discovery from evaluation on unseen data and models. Starting from a seed harness h_0, the optimizer $\mathcal{A}_{\text{opt}}$ uses validation feedback to propose new harnesses, while the

Algorithm 1 METAVUL harness discovery loop

Require: Seed harness h_0, validation pairs D, maximum iterations T, patience K, optimizer agent $\mathcal{A}_{\text{opt}}$, detection agent $\mathcal{A}_{\text{det}}$

$(m_0, z_0) \leftarrow \text{EVALUATE}(\mathcal{A}_{\text{det}}, h_0, D)$ ▷ m: metrics; z: predictions
$\mathcal{H} \leftarrow \{(h_0, m_0, z_0)\}$; $h_{\text{best}} \leftarrow h_0$
$s_{\text{best}} \leftarrow \text{PAIRWISEACC}(z_0)$; $c \leftarrow 0$
for $t \leftarrow 1$ to T **do**
  $W_t \leftarrow \text{BUILDWORKSPACE}(\mathcal{H}, h_{t-1})$
  $h_t \leftarrow \text{PROPOSE}(\mathcal{A}_{\text{opt}}, W_t)$ ▷ keeps SCoT and verdict schema
  if h_t is invalid or the optimizer fails **then**
    $c \leftarrow c + 1$; **continue**
  end if
  $(m_t, z_t) \leftarrow \text{EVALUATE}(\mathcal{A}_{\text{det}}, h_t, D)$ ▷ fresh call per function
  $s_t \leftarrow \text{PAIRWISEACC}(z_t)$
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{(h_t, m_t, z_t)\}$
  if $s_t > s_{\text{best}}$ **then**
    $h_{\text{best}} \leftarrow h_t$; $s_{\text{best}} \leftarrow s_t$; $c \leftarrow 0$
  else
    $c \leftarrow c + 1$
  end if
  if $c \geq K$ **then break**
  end if
end for
$\mathcal{F} \leftarrow \text{PARETOFRONT}(\mathcal{H};\ \text{PAIRWISEACC}, \text{JOINT})$ ▷ non-dominated
return $\mathcal{F}$

Table 1: Datasets used by MetaVul. Circled numbers correspond to the pipeline stages in Fig. 1.

Validation set D	
Dataset	TitanVul + CWEval
Pairs	202
Languages	161 C, 41 C++
CWE pairs	123
CWEs	79
Used for	① Validation

Held-out set	
Dataset	PrimeVul
Pairs	430
Languages	C/C++
CWE pairs	430
CWEs	61
Used for	② Held-out evaluation ③ Cross-model transfer

isolated detector $\mathcal{A}_{\text{det}}$ evaluates each candidate. After discovery, MetaVul retains the Pareto-optimal harnesses and evaluates them on held-out PrimeVul using the search and transfer models. Fig. 1 presents the full pipeline, and Algorithm 1 details the discovery loop.

Harness discovery and selection. MetaVul iteratively generates and evaluates candidate harnesses before selecting those that best balance its evaluation objectives. ***Discovery loop.*** MetaVul first evaluates the seed harness h_0. At each iteration t, the optimizer uses previous harnesses and results to propose h_t, which the detector evaluates on D. The results are added to the search history. Discovery ends after T iterations or K iterations without improving pairwise accuracy. ***Agent roles.*** The optimizer $\mathcal{A}_{\text{opt}}$ revises the harness using validation feedback. The detector $\mathcal{A}_{\text{det}}$ runs the proposed harness and returns predictions for scoring. The agents may use the same model, but they run separately and never share state. ***Pareto selection.*** After the loop, MetaVul returns the Pareto frontier of all evaluated harnesses over two validation objectives, pairwise accuracy and joint hit rate (Sec. 4.2), both maximized (PARETOFRONT in Algorithm 1). A harness dominates another if it is at least as good on both objectives and strictly better on at least one; the frontier $\mathcal{F}$ contains every harness that no other harness dominates, and harnesses with identical scores are kept once.

Sandboxing. It is critical that the held-out set is never exposed during discovery, and that neither discovery nor evaluation is influenced by system files, default instruction files such as `CLAUDE.md`, or user settings. We therefore sandbox every agent. Each agent runs as an isolated CLI process that cannot request or receive additional permissions. The agents are separated by the data they can reach. As shown in Fig. 1, the agent and its model serve two roles: an optimizer and a detector (which we also call the evaluator). ***Optimizer.*** The optimizer can only access its workspace, which holds the validation history: earlier harnesses, their metrics, and their predictions with labels. The results of each iteration are added to this workspace for the next one. For Claude Code, the working directory is set to the workspace, the read tool is restricted to it, write and edit tools are denied, and shell access is limited to read-only commands. No `CLAUDE.md` files, settings, or memory from the user or project are loaded. For Codex, an OS-level sandbox gives read-only access to the workspace, with no writes and no network. ***Detector.*** The detector is launched fresh for every function, with no workspace, no labels, no trajectory, and no memory between calls. Its only input is the rendered harness and a single function. Claude Code runs it from `/tmp` with no tools and an empty system prompt. Codex runs it in an empty temporary directory under the same read-only, offline sandbox. For held-out evaluation on the unseen PrimeVul dataset and for cross-transfer, predictions are never placed in any workspace. The only thing that changes between iterations is the harness under test. Detailed agent restrictions are given in App. G.

4 EXPERIMENTAL DESIGN

We outline our experimental setup: the datasets, seed harness, agents, models, and configurations (Sec. 4.1), and the evaluation metrics (Sec. 4.2).

4.1 DATASETS, MODEL, AND AGENT SELECTION

Datasets. We use separate datasets for optimization and testing, so that we can measure how harness improvements carry over to an unseen dataset and to models that took no part in optimization. Every sample is a vulnerable function paired with its patched version. Table 1 summarizes both splits.

- **Validation set D.** We handcraft a dataset D from TitanVul and CWEval. TitanVul (Li et al., 2026b) merges seven public vulnerability datasets, deduplicates them, and uses multi-agent LLM validation to keep only genuine vulnerability fixes understandable at the function level, reducing noisy labels and unrelated changes. We sample 150 C/C++ pairs, stratified by CWE. CWEval (Peng et al., 2025) evaluates both the functional correctness and the security of LLM-generated code; each of its security-critical tasks has functional and security tests and a secure and an insecure reference implementation. We use all 52 of its C and C++ tasks, pairing each insecure implementation with its secure one. Together these form D: 202 pairs (404 functions). Because 79 of the 150 TitanVul pairs have no CWE label, CWE accuracy on D is computed over the remaining 123 pairs.
- **Held-out set.** PrimeVul (Ding et al., 2024) collects vulnerable functions and their fixes from real open-source C/C++ projects with stricter labeling than earlier datasets, and we use its paired test set, a hard benchmark on which LLMs reach low pairwise accuracy, as our held-out set. Because TitanVul aggregates public vulnerability datasets, we checked it for overlap with D: 5 PrimeVul pairs contain the same functions and fixes as TitanVul pairs in D. We remove them, leaving 430 pairs (860 functions) from 141 projects. Every vulnerable function has a CWE label, and no PrimeVul data is shown to the optimizer.

Seed harness. A harness is a standalone Python file that prepares a function for the detection agent and parses its verdict. Every run starts from the same 151-line SCoT seed harness (Li et al., 2023), provided in App. B.[2]

Agent selection. We focus on two CLI coding agents, Codex CLI and Claude Code. In each discovery session, the optimizer $\mathcal{A}_{\text{opt}}$ and the detector $\mathcal{A}_{\text{det}}$ are the same CLI agent running the same model. They differ only in what they can access: the optimizer reads a workspace of earlier harnesses, metrics and labeled predictions, while the detector sees only the harness and a single function (Sec. 3). In the unseen-dataset and cross-model transfer phases, only $\mathcal{A}_{\text{det}}$ runs, evaluating the frozen harnesses on PrimeVul.

Model selection. We optimized harnesses with Inkling-Small (Thinking Machines Lab, 2026), DeepSeek V4 Flash (OpenRouter, 2026a), Opus 4.6 and GLM-5.3 (OpenRouter, 2026b). We tested cross-model transfer on Opus 5, Nemotron 3.5 Lightning (Thinking Machines Lab, 2026; OpenRouter, 2026f), GPT-4o (OpenRouter, 2026d), GLM-4.7 Flash (OpenRouter, 2026c) and GPT-OSS-120B (OpenRouter, 2026e). These models cover smaller and frontier models, including open-weight and closed models. Opus 4.6 ran through our Claude subscription, whose usage limits every optimizer and detector call draws on, so we chose the remaining models from per-token providers (OpenRouter and Tinker; App. J.1).

Configurations. Opus 4.6 and Opus 5 run natively in Claude Code through our subscription; all other models reach Claude Code or Codex CLI through a local LiteLLM proxy to OpenRouter or Tinker. Run lengths follow inference budgets: Inkling-Small and DeepSeek V4 Flash ran 15 refinement iterations, GLM-5.3 ran up to 15 with a patience of 4 and stopped after iteration 8, and Opus 4.6 ran four, limited by subscription usage. Serving, timeout and decoding details are in App. H.

[2] In preliminary experiments, SCoT achieved higher accuracy and precision than a plain instruction, MoT, and ToT. Its structure also supports reasoning about sequences, branches, loops, and patch effects.

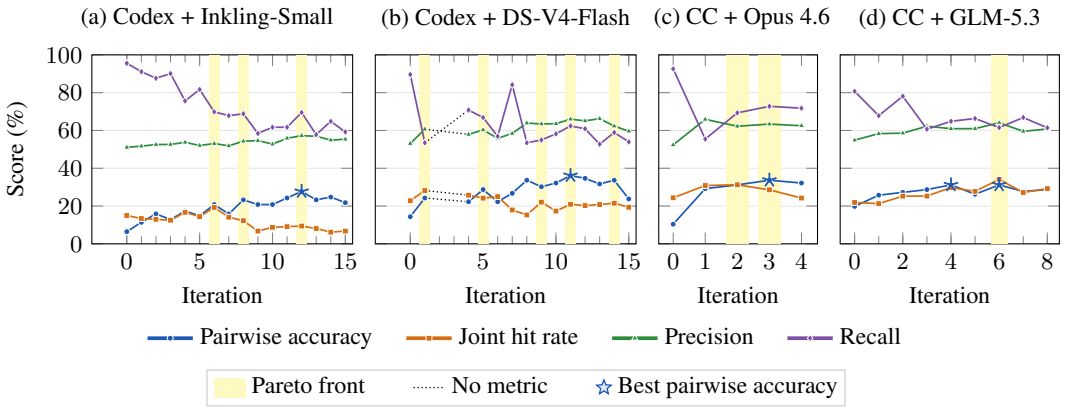


Figure 2: **Validation metrics across optimization.** Each panel shows pairwise accuracy, joint hit rate, precision, and recall on TitanVul+CWEval for one agent-model configuration (CC = Claude Code; DS = DeepSeek).[7] Pairwise accuracy generally rises with precision while recall declines, indicating fewer false alarms but more missed vulnerabilities. Joint hit rate varies by model. marks the Pareto front and ☆ the best pairwise accuracy.

4.2 METRICS

We evaluate each harness on its detection, discrimination, and localization ability. ***Traditional detection metrics.*** We treat each function as a binary decision with vulnerable as the positive class, and report accuracy, precision, recall and F1 over all functions. ***Pairwise accuracy.*** Let pair i contain a vulnerable function x_i^v and its patched version x_i^p. For a binary detector f, pairwise accuracy over N pairs is $\mathrm{PairAcc}(f) = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[f(x_i^v) = 1 \wedge f(x_i^p) = 0]$. A pair counts only when both decisions are correct, which makes the metric stricter than ordinary accuracy.[3] Pairwise accuracy is the target metric used in Algorithm 1. ***CWE and localization metrics.*** These score only true positives, the vulnerable functions i that the harness flags, and only those with the relevant ground truth: a set of CWEs C_i (usually one, sometimes several) and the gold lines G_i. None of the three datasets marks which lines are vulnerable, so we derive G_i from each pair: it contains the lines of the vulnerable function that are modified or deleted by the fix, or the lines around the insertion point if the fix only adds lines.[4] For CWEval, the fix is the secure reference implementation. For each such function, the harness returns an ordered list of CWE guesses, whose first guess is $\hat{c}_i^{(1)}$, and a set of evidence lines $\hat{G}_i$. *CWE accuracy* is the percentage whose CWE list contains a CWE in C_i. *Loc F1* is the harmonic mean of line-level precision $|\hat{G}_i \cap G_i|/|\hat{G}_i|$ and recall $|\hat{G}_i \cap G_i|/|G_i|$, each averaged over functions, and *loc hit rate* is the percentage with at least one correct line ($\hat{G}_i \cap G_i \neq \emptyset$). ***Joint hit rate.*** Joint (JOINT in Algorithm 1) asks whether a correctly flagged function is also explained correctly: the right vulnerability class in the right place. Let T be the set of flagged vulnerable functions that have both C_i and G_i. Function i counts as correct only when its first CWE guess is in C_i and it has at least one correct line: $\mathrm{Joint} = \frac{1}{|T|}\sum_{i \in T} \mathbf{1}[\hat{c}_i^{(1)} \in C_i] \cdot \mathbf{1}[\hat{G}_i \cap G_i \neq \emptyset]$. Only the first CWE guess is scored, so a harness cannot earn credit by listing several candidate CWEs. Regarding parse failures, a function without a parseable verdict is counted as incorrect and stays in the denominator for all metrics, so every PrimeVul result covers all pairs.[5]

5 DISCOVERY AND TRANSFER

Motivation & Approach. We ask whether a feedback-driven design can improve vulnerability-detection harnesses over a starter seed. We consider metrics such as pairwise accuracy and localization because we want both discriminatory and localization ability. However, a harness must serve multiple

[3] A detector that labels both the vulnerable function and its patch as vulnerable gets 50% accuracy and full recall, but $\mathrm{pAcc} = 0$. Thus, pAcc tests whether the detector recognizes the fix. [4] Each dataset pairs vulnerable code with a confirmed secure version. We use the lines changed by the fix as the best available vulnerability locations. [5] Parse failures are rare: 0.36% overall and at most 0.63% in any run. [7] We omit accuracy, F1, and CWE accuracy because accuracy follows precision, F1 summarizes precision and recall, and CWE accuracy overlaps with Joint. Table 6 reports all metrics.

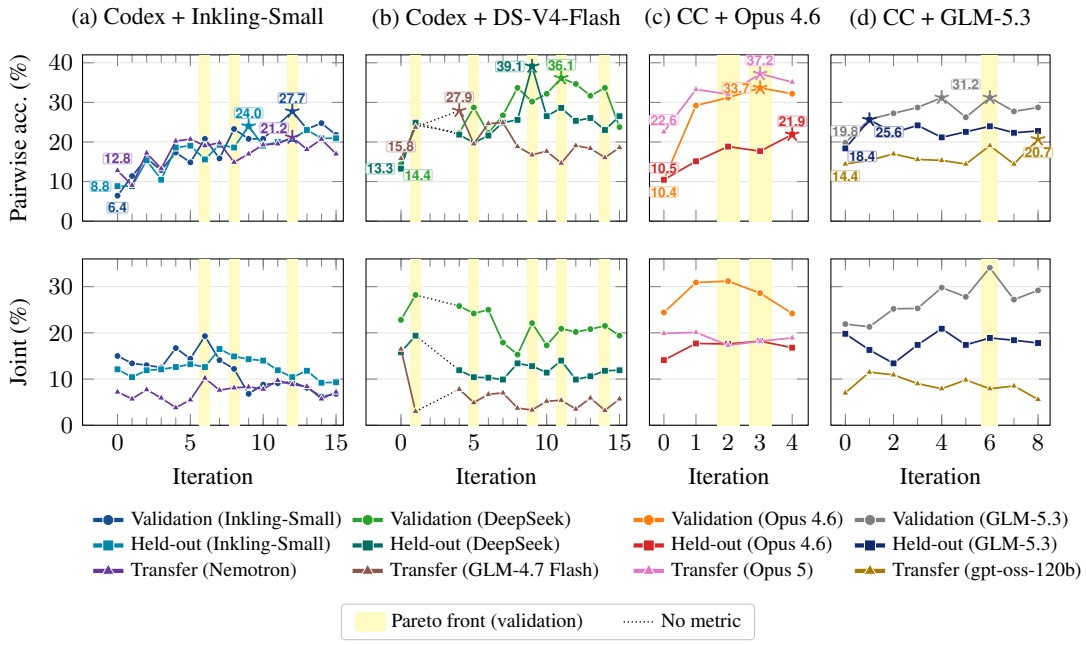


Figure 3: **Pairwise accuracy and joint hit rate across discovery iterations.** Pairwise accuracy (top) and joint hit rate (bottom) across optimizer iterations for four optimization sessions. CC = Claude Code; DS = DeepSeek. Validation uses TitanVul+CWEval; held-out and cross-model transfer use PrimeVul. ☆ marks each series' best pairwise accuracy, marks the validation Pareto front, and labels show seed and best values. Harnesses are selected using validation only; PrimeVul is never used for selection. No metric marks iterations where the optimizer failed or returned an invalid harness.

objectives at once, and a gain on one metric may cause degradation on another, affecting how the best harness is selected. We therefore explore whether our approach improves pairwise accuracy and how these improvements affect other metrics, such as localization. Gains on the data used for optimization could still reflect overfitting: harness search can learn task-specific shortcuts whose gains disappear on held-out tasks, and a harness can overfit to the model that wrote it (Wang et al., 2026; Wu et al., 2026). This motivates **RQ1:** ***Does feedback-driven search improve vulnerability-detection harnesses, and do the gains transfer to unseen data and models?*** We conduct four optimization runs using Codex CLI and Claude Code with Inkling-Small, DeepSeek V4 Flash, Opus 4.6, and GLM-5.3. Each run starts from the same SCoT seed harness and optimizes it on validation set D using Algorithm 1. At every iteration, we measure pairwise accuracy, precision, recall, CWE identification, and localization (Fig. 2), and we retain the validation Pareto frontier over pairwise accuracy and joint hit rate. Every harness is then frozen and run on the held-out PrimeVul set, once with the model that optimized it (unseen-dataset transfer) and once with a model that took no part in optimization: Nemotron 3.5 Lightning, GLM-4.7 Flash, Opus 5 and GPT-OSS-120B (cross-model transfer; Fig. 3). For each evaluation we report the best harness on the validation frontier. App. D provides additional per-metric comparisons, correlations, and counts of harnesses that improve over the seed on the validation set.

Results. Feedback-driven optimization improves validation pairwise accuracy in all four runs, with best-harness gains over the seed of +21.3 (Inkling-Small), +21.8 (DeepSeek V4 Flash), +23.3 (Opus 4.6) and +11.4 points (GLM-5.3) (Fig. 2), and all 40 discovered harnesses beat the seed. The gains largely carry over: a frontier harness beats the seed in all 8 held-out and cross-model evaluations (Fig. 3). On unseen PrimeVul data with the same model, the gains are +11.9, +25.8, +8.4 and +5.6 points in the same order; on models that took no part in optimization, they are +8.4 on Nemotron 3.5 Lightning, +7.9 on GLM-4.7 Flash, +14.7 on Opus 5 and +4.7 on GPT-OSS-120B. Pairwise accuracy generally rises with precision while recall declines, and its relationship with joint hit rate varies by model, from negative for Inkling-Small and DeepSeek V4 Flash to positive for GLM-5.3 (Table 7, App. D). Only 10 of the 40 harnesses improve localization (Table 8, App. D). These trade-offs motivate selecting a Pareto frontier rather than a single harness based only on pairwise accuracy.

Table 2: **GPT-4o-based methods on PrimeVul.** P-C is the number of pairs for which both the vulnerable and patched functions are classified correctly; Error is the number of pairs in which at least one of the two functions is misclassified; and FPR is the false-positive rate. ↑ means higher is better, ↓ means lower is better, and bold marks the best result in each primary metric.

Method	P-C ↑	Error ↓	Precision ↑	Recall ↑	FPR ↓	Accuracy ↑	F1 score ↑
Ding et al.'s CoT (Ding et al., 2024)	40	395	0.5355	0.1908	0.1655	0.5126	0.2814
GPTLens (Hu et al., 2023)	44	391	0.5144	0.6552	0.6184	0.5184	0.5763
VulTrial (Widyasari et al., 2026)	81	354	0.5317	0.5977	0.5264	0.5356	0.5628
ConfColl (Tsai et al., 2025)	22	413	0.5078	0.8253	0.8000	0.5126	**0.6287**
Our approach (seed)	74	361	0.5185	0.5793	0.5379	0.5207	0.5472
Our approach (iter 12 harness)	**103**	**332**	0.5636	0.4276	0.3310	**0.5483**	0.4863
Our approach on 430 pairs, validation overlap removed[†]							
Our approach (seed)	71	359	0.5157	0.5744	0.5395	0.5174	0.5435
Our approach (iter 12 harness)	100	330	0.5631	0.4256	0.3302	0.5477	0.4848

Notes. Bold marks the best result among methods evaluated on all 435 pairs. [†]Excludes 5 pairs overlapping validation.

RQ1 Findings. MetaVul improves pairwise accuracy by 11.4 to 23.3 points ($1.6\times$ to $4.3\times$) over the seed, and the gains hold on unseen data (5.6 to 25.8 points, $1.3\times$ to $2.9\times$) and unseen models (4.7 to 14.7 points, $1.3\times$ to $1.7\times$). We also identify a key open challenge for the field: improving discrimination, localization, and CWE identification together, since gains in one do not reliably carry over to the others.

6 Comparison to Other Methods

Motivation & Approach. Related work evaluates four LLM-based and agentic vulnerability detectors on PrimeVul using GPT-4o, providing a common backbone for comparison (Sec. 2). This motivates **RQ2:** ***How does MetaVul compare with other LLM-based or agentic methods?*** Because GPT-4o was costly to evaluate through OpenRouter, our budget did not permit transferring every discovered harness. We randomly selected the DeepSeek V4 Flash discovery run and evaluated the six harnesses with the highest validation pairwise accuracy to test whether its strongest candidates transferred to GPT-4o.[8] We also evaluated iteration 7; iteration 13 did not complete. App. F reports all completed transfers, while Table 2 compares the seed with iteration 12, the transferred harness with the highest accuracy.

Results. MetaVul correctly discriminates 103 of 435 pairs, compared with 74 for the seed and 81 for the next-best method, whose results are reported from prior work (Table 2). After removing 5 pairs overlapping validation, MetaVul still correctly discriminates 100 of 430 pairs. It also achieves the highest accuracy and the lowest false-positive rate among methods with non-trivial recall. Because our GPT-4o evaluation covered only a budget-limited subset of discovered harnesses, this result should not be viewed as an upper bound; broader exploration may identify an even stronger harness.

RQ2 Findings. On the same GPT-4o backbone, MetaVul achieves stronger pairwise discrimination than existing vulnerability-detection methods, even after removing validation overlap. This shows that the learned harness improves detection without changing the underlying model.

7 Optimization Integrity

Motivation & Approach. Because the optimizer receives labeled validation feedback, its gains could come from shortcuts rather than better harness design. We therefore ask **RQ3:** ***Do MetaVul's gains rely on data leakage, embedded answers, or trivial verdict bias?*** We test two possible shortcut pathways. First, we verify that the optimizer cannot access PrimeVul during discovery. Second, GPT-OSS-120B serves as a judge and examines all 44 harnesses using only their text. It assigns separate 0–4 ratings for (1) preserving SCoT, (2) avoiding prohibited content such as examples, labels, dataset hints, or verdict-bias instructions, and (3) preserving the output format. Table 3 reports these checks for the Pareto-frontier harnesses, with full judge results in App. A.

[8] At the time of evaluation, GPT-4o cost $10 per million output tokens on OpenRouter.

Table 3: **Shortcut checks for Pareto-frontier harnesses.** Ratings show how many frontier harnesses receive $4/4$ for preserving SCoT, avoiding prohibited content, and preserving the output format. The final columns show reductions relative to each run's seed. App. A reports every frontier harness.

Agent + Model	Frontier iterations	SCoT	No prohibited	Output	FP reduction	Flagged reduction
Codex + Inkling-Small	6, 8, 12	3/3	3/3	3/3	33–44%	31–36%
Codex + DeepSeek V4 Flash	1, 5, 9, 11, 14	5/5	5/5	4/5	44–60%	34–49%
Claude Code + Opus 4.6	2, 3	2/2	2/2	2/2	50%	35–37%
Claude Code + GLM-5.3	6	1/1	1/1	0/1	49%	35%
Total	**11 harnesses**	**11/11**	**11/11**	**9/11**	**33–60%**	**31–49%**

Results. We find no evidence that the Pareto-frontier harnesses exploit the tested shortcuts. All 11 receive $4/4$ for preserving SCoT and avoiding prohibited content. Two receive lower output-format ratings because of conflicting instructions, a quality issue rather than a shortcut. Every frontier harness reduces both false positives and the total number of flagged functions, with false positives falling by 33% to 60%.

RQ3 Findings. We find no evidence that MetaVul's Pareto-frontier harnesses rely on data leakage, embedded answers, or increased flagging. All reduce false positives while preserving the required reasoning scaffold.

8 Limitations

Language and granularity. All experiments use C/C++ functions, so the results may not carry over to memory-safe or dynamically typed languages. Detection is also performed on single functions, without the calling context or the rest of the repository. **Same model for both roles.** During discovery, the optimizer and detector use the same model, which may favor model-specific harnesses, though cross-model transfer helps evaluate and mitigate this risk. **Seed harness.** Although we selected a strong seed based on a pilot comparison of reasoning strategies, the space of possible seeds is large, so we did not test whether other seeds lead to different or better harnesses. **Workspace components.** We do not establish which optimizer inputs account for the gains. A preliminary ablation of prior harnesses, metrics, score history and labeled predictions was inconclusive (App. J). **Cost and access.** Subscription usage limits and API costs constrained the number of iterations, repeated runs and model pairings. **Training-data contamination.** We cannot fully rule out contamination, since most evaluated models have training cutoffs after the benchmarks were released, but three observations make it unlikely. GPT-4o has a training cutoff (October 2023) that precedes the release of PrimeVul, TitanVul and CWEval, and harnesses learned with DeepSeek V4 Flash still improve on it (79–101 vs. 71 correct pairs; App. F). The seed harness also scores near the floor on pairwise accuracy (10.5%), which memorized fixes would not produce, and grouping 386 PrimeVul pairs by CVE year (2008–2022) shows no monotonic trend.[9]

9 Conclusion

MetaVul automatically discovers vulnerability-detection harnesses that outperform a hand-designed seed. Across four runs, validation pAcc improves by 11.4–23.3 points ($1.6\times$–$4.3\times$). These gains persist across all eight held-out and cross-model evaluations, reaching 25.8 points on unseen PrimeVul data and 14.7 points on an unseen model. On GPT-4o, a transferred harness correctly discriminates 103 of 435 pairs, outperforming existing methods and achieving the highest accuracy. We find no evidence of shortcuts: every Pareto-frontier harness preserves SCoT, avoids prohibited content, and reduces false positives by 33%–60%. These findings expose a broader challenge for future harness optimization: jointly improving discrimination, localization, and CWE identification without trading one capability for another. ***Future work.*** Future work should study this trade-off directly, since techniques that improve discrimination and localization metrics together remain underexplored. It could also swap or pair different optimizer and detector agents, start discovery from other seed harnesses, explore a spec-driven harness design, and add an automatic output-format check to the Pareto filter.

[9] Every PrimeVul pair is a real bug reported in a particular year. We group pairs by year (2008–2022) and measure performance in each group. If the model memorized fixes during training, older bugs that have been public longer should be easier.

AI USE STATEMENT

We used AI to improve transitions and wording in our writing to make it clearer for our audience. We did not use AI to conceive any research ideas. All ideas came from reading the literature and related work, discussions among the authors, and insights from past projects, and we first wrote our own notes and human-authored drafts. We also used Claude Code and Codex for code maintenance and debugging. All AI-assisted text and code were reviewed by the authors.

REPRODUCIBILITY STATEMENT

Our code is available at `https://github.com/BTP24VFnNd/submission`.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Nicholas Carlini, Keane Lucas, Evyatar Ben Asher, Newton Cheng, Hasnain Lakhani, David Forsythe, and Kyla Guru. Evaluating and mitigating the growing risk of LLM-discovered 0-days. Anthropic, February 2026. URL https://www.anthropic.com/research/zero-days.

Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review, 2024. URL https://arxiv.org/abs/2407.12241.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we?, 2024. URL https://arxiv.org/abs/2403.18624.

Esakkivel Esakkiraja, Denis Akhiyarov, Vikas Yadav, Sai Rajeswar, Patrice Bechard, Sridhar Nemala, and Sagar Davasam. Starharness: Evolving harnesses with stratified search for enterprise environments, 2026. URL https://arxiv.org/abs/2608.24804.

Jiwoo Hong, Noah Lee, and James Thorne. Orpo: Monolithic preference optimization without reference model, 2024. URL https://arxiv.org/abs/2403.07691.

Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pp. 297–306, 2023. doi: 10.1109/TPS-ISA58951.2023.00044.

Lisheng Huang, Chen Yang, Hao Zhou, Huatong Song, Zongchao Chen, Ran Le, Yang Song, Wayne Xin Zhao, and Tao Zhang. Evo-bench: Can language models improve agent harness?, 2026a. URL https://arxiv.org/abs/2608.09096.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. Memoharness: Agent harnesses that learn from experience, 2026b. URL https://arxiv.org/abs/2607.14159.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026. URL https://arxiv.org/abs/2603.28052.

Fengjie Li, Jiajun Jiang, Dongchi Chen, and Yingfei Xiong. Llm-based vulnerability detection at project scale: An empirical study, 2026a. URL https://arxiv.org/abs/2601.19239.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation, 2023. URL https://arxiv.org/abs/2305.06599.

Yikun Li, Ngoc Tan Bui, Ting Zhang, Chengran Yang, Xin Zhou, Martin Weyssow, Jinfeng Jiang, Junkai Chen, Huihui Huang, Huu Hung Nguyen, et al. Out of distribution, out of luck: How well can llms trained on vulnerability datasets detect top 25 cwe weaknesses? In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 2096–2108, 2026b.

Houjun Liu, Lisa Einstein, John Yang, Joachim Baumann, Duncan Eddy, Christopher D. Manning, Mykel Kochenderfer, and Diyi Yang. Secureforge: Finding and preventing vulnerabilities in llm-generated code via prompt optimization, 2026. URL https://arxiv.org/abs/2605.08382.

Ryan Lopopolo. Harness engineering: Leveraging Codex in an agent-first world. OpenAI, https://openai.com/index/harness-engineering/, February 2026.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution, 2026. URL https://arxiv.org/abs/2607.08124.

OpenAI. GPT-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

OpenRouter. DeepSeek: DeepSeek V4 Flash model pricing. https://openrouter.ai/deepseek/deepseek-v4-flash-0731, 2026a.

OpenRouter. Z.ai: GLM-5.3 model pricing. https://openrouter.ai/z-ai/glm-5.3, 2026b.

OpenRouter. Z.ai: GLM 4.7 Flash model pricing. https://openrouter.ai/z-ai/glm-4.7-flash, 2026c.

OpenRouter. OpenAI: GPT-4o model pricing. https://openrouter.ai/openai/gpt-4o, 2026d.

OpenRouter. OpenAI: gpt-oss-120b model pricing. https://openrouter.ai/openai/gpt-oss-120b, 2026e.

OpenRouter. NVIDIA: Nemotron 3.5 Lightning model pricing. https://openrouter.ai/nvidia/nemotron-3.5-lightning, 2026f.

Ruwei Pan and Hongyu Zhang. MoT: Modularization-of-thought prompting for effective code generation, 2025. URL https://arxiv.org/abs/2503.12483.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 33–40. IEEE, 2025.

Prithvi Rajasekaran. Harness design for long-running application development. Anthropic, https://www.anthropic.com/engineering/harness-design-long-running-apps, March 2026.

Amirali Sajadi, Saikat Dutta, and Preetha Chatterjee. An empirical analysis of CodeQL false positives and query refinements for Java vulnerabilities, 2026. URL https://arxiv.org/abs/2609.04535.

Wenxin Tang, Xiang Zhang, Junliang Liu, Jingyu Xiao, Xi Xiao, Jinlong Yang, Yuehe Ma, Zhenyu Liu, Zhengheng Li, Zicheng Wang, Wang Luo, Qing Li, Lei Wang, and Peng Xiangli. VulTriage: Triple-path context augmentation for LLM-based vulnerability detection, 2026. URL https://arxiv.org/abs/2605.09461.

Shahroz Tariq, Mohan Baruwal Chhetri, Surya Nepal, and Cecile Paris. Alert fatigue in security operations centres: Research challenges and opportunities. *ACM Comput. Surv.*, 57(9), April 2025. ISSN 0360-0300. doi: 10.1145/3723158. URL https://doi.org/10.1145/3723158.

Thinking Machines Lab. Models & pricing. Tinker documentation, https://tinker-docs.thinkingmachines.ai/tinker/models/, 2026.

Chung-Nan Tsai, Xin Wang, Cheng-Hsiung Lee, and Ching-Sheng Lin. A sequential multi-stage approach for code vulnerability detection via confidence- and collaboration-based decision making. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 21151–21157, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1071. URL https://aclanthology.org/2025.emnlp-main.1071/.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S. Kalmath, Veronica Chatrath, and Yuan Xue. Harnessopt-bench: Evaluating llms at harness optimization, 2026. URL https://arxiv.org/abs/2608.06301.

John Walker. Benchmarking Opus 4.6 for vuln detection: Flashes of brilliance but lots of noise. ZeroPath, https://zeropath.com/blog/benchmarking-opus-4-6-vuln-detection, April 2026.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents, 2026. URL https://arxiv.org/abs/2607.12227.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. Let the trial begin: A mock-court approach to vulnerability detection using llm-based agents. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, ICSE '26, pp. 1030–1042, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400720253. doi: 10.1145/3744916.3773256. URL https://doi.org/10.1145/3744916.3773256.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, Zexuan Wang, Chen He, Chen Huang, Maojia Song, Zhiyuan Zeng, Shaowen Wang, Jinkai Liu, Yunfeng Shi, Jiaheng Liu, Shen Yan, Wenhao Huang, Ge Zhang, and Wenxuan Zhang. Harnessdev: Can llms create and evolve their own agent harness?, 2026. URL https://arxiv.org/abs/2609.01437.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822, 2023. doi: 10.52202/075280-0517. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

Alperen Yildiz, Sin G. Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil M. Divakaran. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories, 2025. URL https://arxiv.org/abs/2503.03586.

Yifan Zhang, Yutong Dai, Juntao Tan, Luyu Yang, Rishi Mullur, Thai Hoang, Zhiyuan Hu, James Zhu, Phil Mui, Silvio Savarese, Ran Xu, and Zeyuan Chen. Darwinx: Evolving agent harnesses through natural selection, 2026. URL https://arxiv.org/abs/2608.07545.

Jingsheng Zheng, Xinyuan Fang, Jintian Zhang, Zhengke Gui, Huajun Chen, and Ningyu Zhang. Onedayagent: Towards a long-horizon harness for autonomous agents, 2026. URL https://arxiv.org/abs/2608.05013.

A Harness Quality and Trajectory Auditing

We audit individual harnesses and their evolution across completed optimization iterations. The framework records both deterministic static analysis and LLM-based semantic judging separately, then tracks each result longitudinally.

Intrinsic harness evaluation. Static evaluation ingests the candidate harness and, only for token-efficiency comparison, a fixed seed harness. Semantic judging compares the candidate harness using a fixed rubric. Neither component reads code samples, labels, predictions, or downstream performance. The trajectory auditor separately reads the completed-iteration history and saved metrics for reporting. Thus, intrinsic evaluation determines whether a harness is structurally and semantically suitable for vulnerability analysis, rather than its efficacy in vulnerability identification. The semantic judge is prohibited from inferring accuracy, precision, recall, or vulnerability-detection performance.

For each harness h_i, the evaluator records four distinct values:

$$(Q_{\text{static}}, Q_{\text{semantic}}, E_{\text{static}}, E_{\text{semantic}})$$

Q_{static} and Q_{semantic} are quality scores, while E_{static} and E_{semantic} are eligibility decisions. The framework does not combine them into a composite score.

Quantitative static quality. Q_{static} ranges from 0 to 100 and comprises four components:

- Template integrity: 40 points
- Schema mechanics: 30 points
- Format hygiene: 15 points
- Seed-relative token efficiency: 15 points

Template integrity checks that the harness is valid text, contains exactly one code placeholder, and has no malformed or unknown template variables. Schema mechanics validates that the vulnerability-report structure is present and complete. Format hygiene checks duplicate instructions and headings, whitespace and line-ending consistency, and Markdown-fence balance. Token efficiency compares the candidate harness with the fixed seed harness.

Qualitative semantic quality. A modular LLM-as-judge evaluates the candidate harness against nine rubric dimensions: role alignment; task identity and primary task; input-object and untrusted-code boundary; SCoT Sequence, Branch, and Loop scaffold; scope and grounding; prohibited behavior; output-contract semantics; internal consistency and instruction precedence; and organization and maintainability. Each dimension is rated from 0–4. Ratings from valid judge attempts are averaged by dimension and weighted to produce Q_{semantic} on a 0–100 scale.

Eligibility criteria. Eligibility acts as a minimum validity gate distinct from quality scoring. E_{static} requires the essential template and schema conditions. E_{semantic} requires no critical semantic failure and average ratings of at least 2 out of 4 for the primary task, SCoT scaffold, scope/grounding, and output-contract dimensions. A harness can therefore receive a relatively high quality score yet remain ineligible.

Harness trajectory auditing. For every completed optimization iteration, the audit records static and semantic quality, eligibility, duplicates identified from normalized harness text, changed-line counts, optimizer change summaries, and saved downstream metrics. The latter include pairwise accuracy, accuracy, precision, recall, F1, parse failures, and sample count when available. The audit does not impose monotonic improvement or construct a combined trajectory score; each measurement remains separate so that changes across iterations can be examined directly.

Table 4 reports every validation Pareto-frontier harness (pairwise accuracy vs. joint hit rate). Ratings are the GPT-OSS-120B judge's 0–4 scores averaged over its valid runs; FP and Flagged compare the seed with each harness on the 404 validation functions.

Table 4: Every Pareto-frontier harness: validation metrics, judge ratings, and length.

Agent + Model	Iter.	Pairwise	Joint	SCoT	Prohib.	Output	Judge	Length	FP	Flagged
Codex + Inkling-Small	6	20.8	19.3	4	4	4	99.4	3.8×	185 → 123	378 → 262
	8	23.3	12.2	4	4	4	98.8	3.7×	185 → 117	378 → 256
	12	27.7	9.4	4	4	4	100.0	4.8×	185 → 104	378 → 243
Codex + DeepSeek V4 Flash	1	24.3	28.2	4	4	4	100.0	4.1×	160 → 70	341 → 178
	5	28.7	24.2	4	4	3	90.4	12.0×	160 → 89	341 → 224
	9	30.2	22.1	4	4	4	91.2	19.3×	160 → 64	341 → 175
	11	36.1	20.9	4	4	4	100.0	13.9×	160 → 65	341 → 191
	14	33.7	21.5	4	4	4	99.4	21.5×	160 → 72	341 → 191
Claude Code + Opus 4.6	2	31.2	31.3	4	4	4	100.0	7.8×	170 → 85	357 → 225
	3	33.7	28.6	4	4	4	100.0	7.7×	170 → 85	357 → 232
Claude Code + GLM-5.3	6	31.2	34.1	4	4	3.5	98.0	16.2×	134 → 69	297 → 192

Table 5 lists every harness that the judge rated below the top score on the prohibited-content or output-format dimension.

Table 5: Harnesses below the top rubric rating (GPT-OSS-120B judge, two runs per harness; a rating averages the valid runs). ‡ marks harnesses on the validation Pareto frontier; the only prohibited-content case is not on the frontier, and the two frontier cases concern output format only.

Harness	Dimension	Rating	Finding
Inkling-Small, iter 14	Prohibited content	3.0	Prefers "not vulnerable" when source or sink is unquoted
Inkling-Small, iter 13	Output format	3.5	SCoT reasoning required, then "only" a JSON object
DeepSeek V4 Flash, iter 5‡	Output format	3.0	SCoT reasoning required, then "only" a JSON object
GLM-5.3, iter 2	Output format	3.5	The example output schema leaves line numbers as `null`, but the instructions require integer line numbers
GLM-5.3, iter 3	Output format	3.0	The example output schema leaves line numbers as `null`, but the instructions require integer line numbers
GLM-5.3, iter 6‡	Output format	3.5	The example output schema leaves line numbers as `null`, but the instructions require integer line numbers

B SEED HARNESS

Each discovery run starts from the full SCoT seed harness below (iteration 0). The 151-line Python file prepares each function for analysis and validates the detector's JSON response. The optimizer modifies this file while preserving its overall structure.

```python
#!/usr/bin/env python3
"""
Standalone vulnerability-detection harness.

Plug this into any agent or model backend:
    from harness import build_prompt, parse_verdict
    prompt = build_prompt(code)          # send this to your model
    verdict = parse_verdict(raw_output)  # feed the model's raw text back in
"""
import json
import re
import sys
from typing import Any, Dict, Optional

PROMPT_TEMPLATE = 'You are an expert software engineer and security analyst.\n\nAnalyze the following code for security vulnerabilities.\n\nUse Structured Chain-of-Thought (SCoT):\n- reason using explicit program structures:\n- Sequence (ordered steps)\n- Branch (if/else)\n- Loop (for/while)\n\nRules:\n- Base reasoning only on the given code.\n- Follow actual control flow (paths, checks, state changes).\n- Focus on security-relevant behavior (memory safety, bounds, lifetime, input validation, integer safety, authz/authn, unsafe APIs).\n- Do NOT rewrite or modify code.\n- After reasoning, output ONLY the JSON object described
```

```
↪ below.\n\nOutput format requirements:\n- First: write your SCoT reasoning in plain text
↪ with numbered steps.\n- Then: output ONLY one JSON object (no markdown, no extra text).\n\
↪ nRequired JSON schema:\n{\n  "verdict": {\n    "vulnerable": true/false,\n    "confidence":
↪  0.0-1.0,\n    "cwe": ["CWE-..."],\n    "severity": "low|medium|high|critical|unknown",\n
↪   "summary": "..."\n  },\n  "evidence": [\n    {\n      "file": null,\n      "function":
↪ null,\n      "location": {"start_line": null, "end_line": null},\n      "snippet": "...",\n
↪       "reason": "..."\n    }\n  ],\n  "notes": {\n    "assumptions": [],\n    "limitations":
↪  []\n  }\n}\n\nCode:\n{{CODE}}\n'

REQUIRED_VERDICT_KEYS = {"vulnerable", "confidence", "cwe", "severity", "summary"}
REQUIRED_TOP_KEYS = {"verdict", "evidence", "notes"}


def build_prompt(code: str) -> str:
    """Render the harness prompt for a single code sample."""
    return PROMPT_TEMPLATE.replace("{{CODE}}", code)


def _strip_eos_tokens(raw: str) -> str:
    for tok in ("<|endoftext|>", "<|end|>", "<|im_end|>", "<|im_start|>"):
        raw = raw.replace(tok, "")
    return raw


def _sanitize_control_chars(raw: str) -> str:
    def _replace(m):
        c = m.group(0)
        return {'\t': '\\t', '\b': '\\b', '\f': '\\f'}.get(c, f'\\u{ord(c):04x}')
    return re.sub(r'[\x00-\x08\x0b\x0c\x0e-\x1f]', _replace, raw)


def _find_verdict_object(text: str, strict: bool = True) -> Optional[Dict[str, Any]]:
    decoder = json.JSONDecoder(strict=strict)
    for m in re.finditer(r'\{', text):
        start = m.start()
        try:
            obj, _ = decoder.raw_decode(text, start)
        except json.JSONDecodeError:
            continue
        if not isinstance(obj, dict) or "verdict" not in obj:
            continue
        v = obj["verdict"]
        if isinstance(v, str) and "|" in v:
            continue
        if isinstance(v, dict) and isinstance(v.get("vulnerable"), bool):
            return obj
    return None


def _extract_verdict_direct(text: str) -> Optional[Dict[str, Any]]:
    for m in re.finditer(r'"verdict"\s*:\s*\{', text):
        vstart = m.end() - 1
        depth = 0
        for i in range(vstart, min(vstart + 5000, len(text))):
            if text[i] == '{':
                depth += 1
            elif text[i] == '}':
                depth -= 1
            if depth == 0:
                verdict_str = text[vstart:i + 1]
                for strict in (True, False):
                    try:
                        obj = json.loads(verdict_str, strict=strict)
                        if isinstance(obj, dict) and isinstance(obj.get("vulnerable"), bool):
                            return {"verdict": obj}
                    except json.JSONDecodeError:
                        continue
                break
    return None


def extract_json(raw: str) -> Optional[Dict[str, Any]]:
    """Find the first valid JSON verdict object in raw model output."""
    cleaned = _strip_eos_tokens(raw)

    result = _find_verdict_object(cleaned, strict=True)
    if result:
        return result

    result = _find_verdict_object(cleaned, strict=False)
    if result:
```

```
        return result

    sanitized = _sanitize_control_chars(cleaned)
    result = _find_verdict_object(sanitized, strict=True)
    if result:
        return result

    stripped = re.sub(r'```(?:json)?\s*', '', cleaned)
    result = _find_verdict_object(stripped, strict=False)
    if result:
        return result

    return _extract_verdict_direct(cleaned)


def validate_schema(obj: Dict[str, Any]) -> list:
    errors = []
    for k in REQUIRED_TOP_KEYS:
        if k not in obj:
            errors.append(f"missing top-level key: {k}")
    verdict = obj.get("verdict", {})
    if isinstance(verdict, dict):
        for k in REQUIRED_VERDICT_KEYS:
            if k not in verdict:
                errors.append(f"missing verdict key: {k}")
        if "vulnerable" in verdict and not isinstance(verdict["vulnerable"], bool):
            errors.append(f"verdict.vulnerable should be bool, got {type(verdict['vulnerable']).__name__}")
    else:
        errors.append(f"verdict should be dict, got {type(verdict).__name__}")
    return errors


def parse_verdict(raw_output: str) -> Dict[str, Any]:
    """Parse a model's raw text response into a structured verdict.

    Returns dict with: parsed, errors, vulnerable, confidence.
    """
    obj = extract_json(raw_output)
    if obj is None:
        return {
            "parsed": None,
            "errors": ["no valid JSON verdict found in output"],
            "vulnerable": None,
            "confidence": None,
        }
    schema_errors = validate_schema(obj)
    verdict = obj.get("verdict", {})
    return {
        "parsed": obj,
        "errors": schema_errors,
        "vulnerable": verdict.get("vulnerable") if isinstance(verdict, dict) else None,
        "confidence": verdict.get("confidence") if isinstance(verdict, dict) else None,
    }


def main() -> None:
    """CLI: render the prompt for a code file (or stdin) and print it."""
    code = open(sys.argv[1]).read() if len(sys.argv) > 1 else sys.stdin.read()
    print(build_prompt(code))


if __name__ == "__main__":
    main()
```

C Pairwise Accuracy Example

Fig. 4 shows why standard metrics can hide poor vulnerability discrimination. Detector A makes the correct number of predictions but labels the wrong functions as vulnerable, giving it zero pairwise accuracy. Detector B has the same accuracy and F1 as A but correctly distinguishes one vulnerable function from its fixed version. Detector C labels every function as vulnerable, producing the highest F1 but zero pairwise accuracy because it never distinguishes vulnerable code from its fix. A pair is correct only when the vulnerable function is labeled vulnerable and its fixed version is labeled safe.

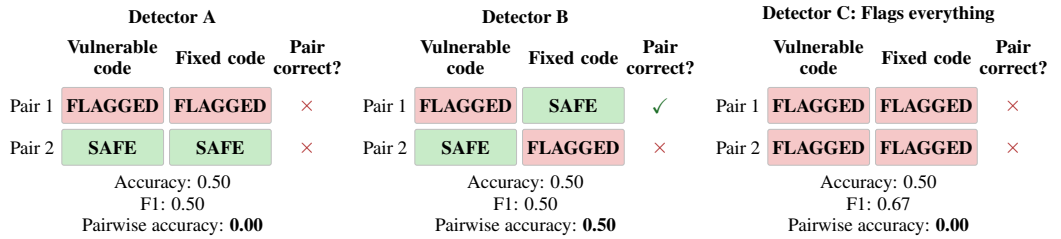


Figure 4: A pair is correct only when vulnerable code is flagged and fixed code is not.

D RQ1: Full Metric Breakdown

This section provides supporting analyses for RQ1. Table 6 compares the seed and best-pairwise harness across all metrics. Table 7 reports how pairwise accuracy correlates with other metrics, and Table 8 counts how many harnesses improve over the seed. Overall, pairwise accuracy aligns with accuracy and precision, but not consistently with recall, CWE identification, localization, or Joint.

Table 6: Seed and best-pairwise harness performance on TitanVul+CWEval (202 pairs). GLM-5.3 ties at iterations 4 and 6.

Agent + Model	Harness	Pairwise	Accuracy	Precision	Recall	F1	CWE acc	Loc F1	Loc hit	Joint
Codex + Inkling-Small	Seed	6.4	0.520	0.511	0.955	0.665	30.8	33.0	61.7	15.0
	Best pairwise (iter 12)	**27.7**	0.589	0.572	0.695	0.627	18.8	28.5	59.7	9.4
Codex + DeepSeek V4 Flash	Seed	14.4	0.552	0.531	0.896	0.667	42.1	32.0	69.1	22.8
	Best pairwise (iter 11)	**36.1**	0.651	0.660	0.624	0.641	36.0	28.6	51.6	20.9
Claude Code + Opus 4.6	Seed	10.4	0.542	0.524	0.926	0.669	46.2	36.4	68.5	24.4
	Best pairwise (iter 3)	**33.7**	0.653	0.634	0.728	0.677	39.8	39.8	67.3	28.6
Claude Code + GLM-5.3	Seed	19.8	0.572	0.549	0.807	0.653	40.0	32.3	73.0	21.9
	Best pairwise (iter 4)	**31.2**	0.616	0.609	0.648	0.628	40.4	32.6	72.5	29.8
	Best pairwise (iter 6)	**31.2**	0.635	0.641	0.615	0.628	47.7	37.4	71.5	34.1

Table 7: Correlation between pairwise accuracy and other metrics. Positive values mean the metrics improve together; negative values indicate a trade-off.

Metric	Inkling-Small	DeepSeek V4 Flash	Opus 4.6	GLM-5.3
Accuracy	+0.88	+0.86	+1.00	+0.64
Precision	+0.92	+0.89	+0.40	+0.76
Recall	−0.72	−0.21	0.00	−0.77
F1	−0.61	+0.07	+0.40	−0.47
CWE acc (any-match)	−0.63	−0.33	−0.50	+0.33
CWE acc (first CWE)	−0.54	−0.28	−0.10	+0.80
Loc precision	−0.36	+0.19	+0.40	+0.08
Loc recall	+0.02	−0.19	−0.90	−0.04
Loc F1	−0.26	−0.27	+0.10	+0.29
Loc hit rate	+0.20	−0.43	−0.60	+0.02
Joint (any-match CWE)	−0.73	−0.56	−0.80	+0.44
Joint	−0.64	−0.54	−0.10	+0.84

Table 8: Number of harnesses that beat the seed on pairwise accuracy, localization, and Joint. The last two columns show the largest gains over the seed.

	Harnesses beating the seed on				Largest gain over seed	
Agent + Model	Pairwise	+ Loc hit	+ Loc F1	+ Joint	Pairwise	Loc F1
Codex + Inkling-Small	15 / 15	9	2	2	+21.3	+0.9
Codex + DeepSeek V4 Flash	13 / 13	0	1	4	+21.8	+3.7
Claude Code + Opus 4.6	4 / 4	1	4	3	+23.3	+4.4
Claude Code + GLM-5.3	8 / 8	0	6	7	+11.4	+5.1
Total	40 / 40	10	13	16		

E Additional Results

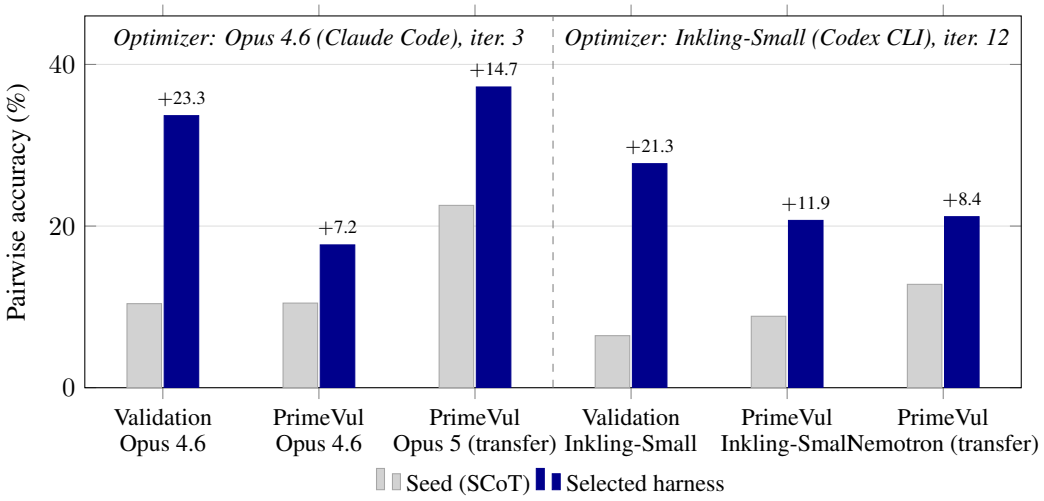


Figure 5: Pairwise accuracy of the seed and the harness selected on validation; labels give the gain in points. Transfer bars use a model that took no part in optimization.

E.1 Expanded per-iteration plots

Figs. 6–9 repeat the columns of Fig. 3 at full width, with the value of every point printed. Light yellow bands mark the Pareto frontier on validation and stars mark the best pairwise accuracy of each series, as in Fig. 3.

Due to constrained hosted-model inference token budgets, we ran four refinement iterations with Opus 4.6 and five with GPT-5.4-mini rather than the 15 iterations used for Inkling-Small.

GPT-5.4-mini improved pairwise validation accuracy from 16.83% for the seed harness to 23.27% at iteration 1, a gain of 6.44 points. The validation-selected harness achieved 16.78% on PrimeVul. Because a matching held-out seed evaluation wasn't recorded, we report the absolute held-out score rather than an improvement.

Open-weight models. We additionally evaluated whether the same harness discovery loop remains effective with smaller, locally hosted open-weight models. We served Qwen3-8B-AWQ through vLLM on an NVIDIA RTX 4080 GPU (16 GB VRAM) and the model served as both optimizer and evaluator (with thinking enabled for both roles). Sampling settings were temperature 0.6, top-p 0.95, top-k 20, minimum-p 0, base seed 42, a 4,096-token thinking budget, and at most 6,144 output tokens. We ran the full 202-pair TitanVul+CWEval validation set for 10 refinement iterations, selected the checkpoint with the highest pairwise validation accuracy, and evaluated the seed and every iteration separately on all 435 held-out PrimeVul pairs. Invalid responses (truncated, schema invalid, missing verdict, or missing reasoning telemetry) were rejected and retried with deterministic seed increments and only complete schema-valid records were scored. vLLM context cap was increased from 16,384 to 32,768 tokens before retrying inputs that exceeded the initial cap, and no rejected responses contribute to reported metrics

We repeat the validation and held-out evaluations with Qwen3-8B-AWQ (Tables 9 and 10). Fig. 10 compares their pairwise-accuracy trajectories and marks the validation-selected checkpoint.

With Qwen3-8B-AWQ, pairwise validation accuracy increased from 8.91% for the seed harness to 15.84% at iteration 8, gain of +6.93 points. On held-out PrimeVul, the iteration-8 harness reached 14.25% versus 13.79% for the seed, a smaller gain of +0.46 points. These results provide supporting evidence that MetaVul's refinement loop can operate consistently under local open-weight inference. The modest held-out gain, weaker improvement than Opus 4.6, and inconsistent CWE and localization metrics suggest that model capability remains an important factor.

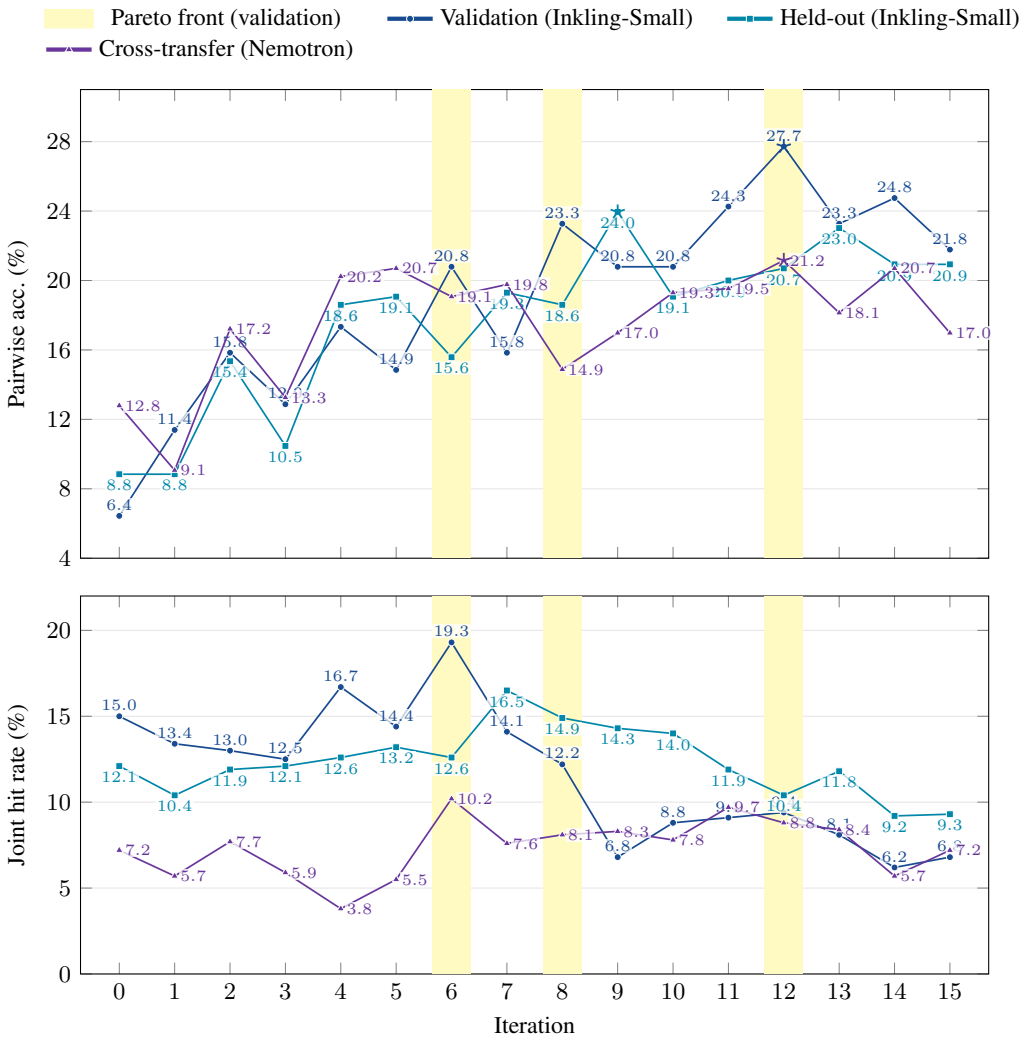


Figure 6: Expanded Fig. 3(a): Codex + Inkling-Small.

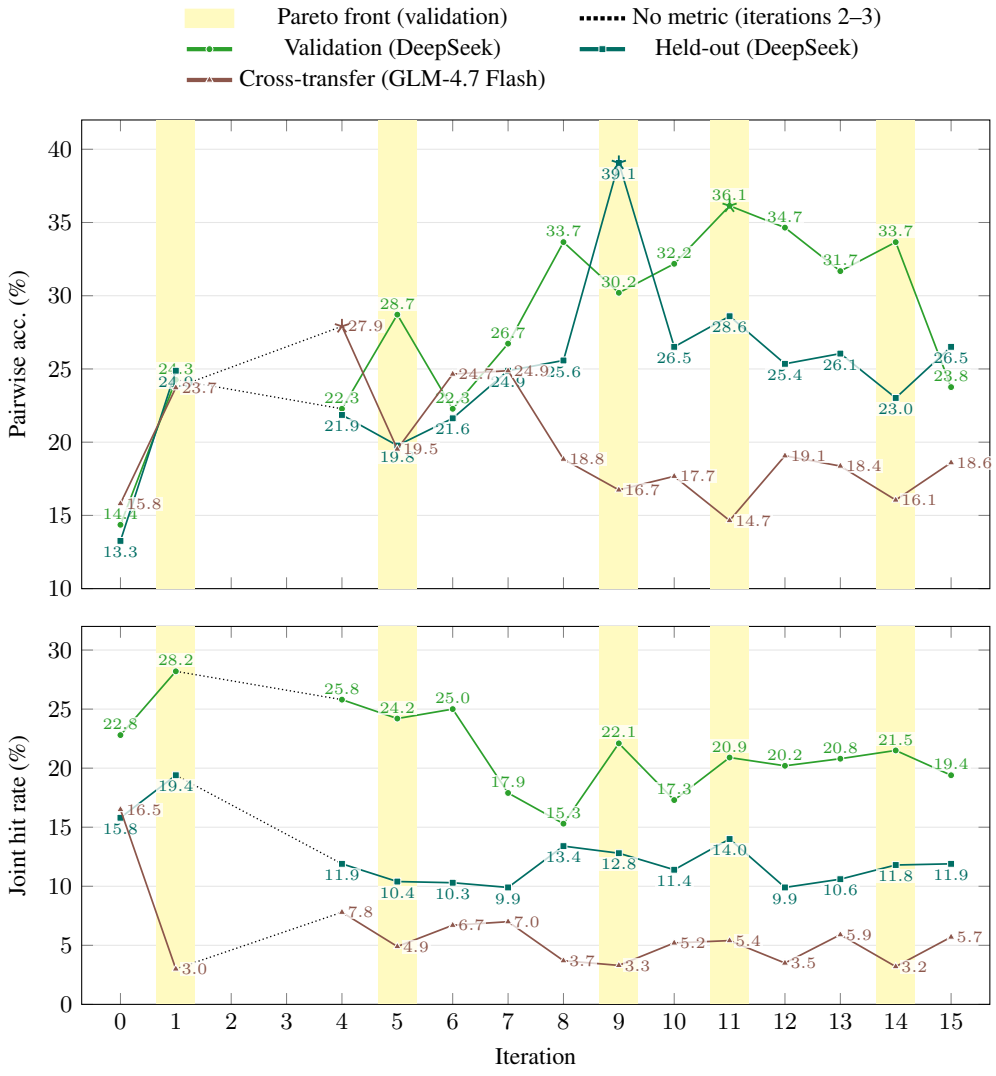


Figure 7: Expanded Fig. 3(b): Codex + DeepSeek V4 Flash.

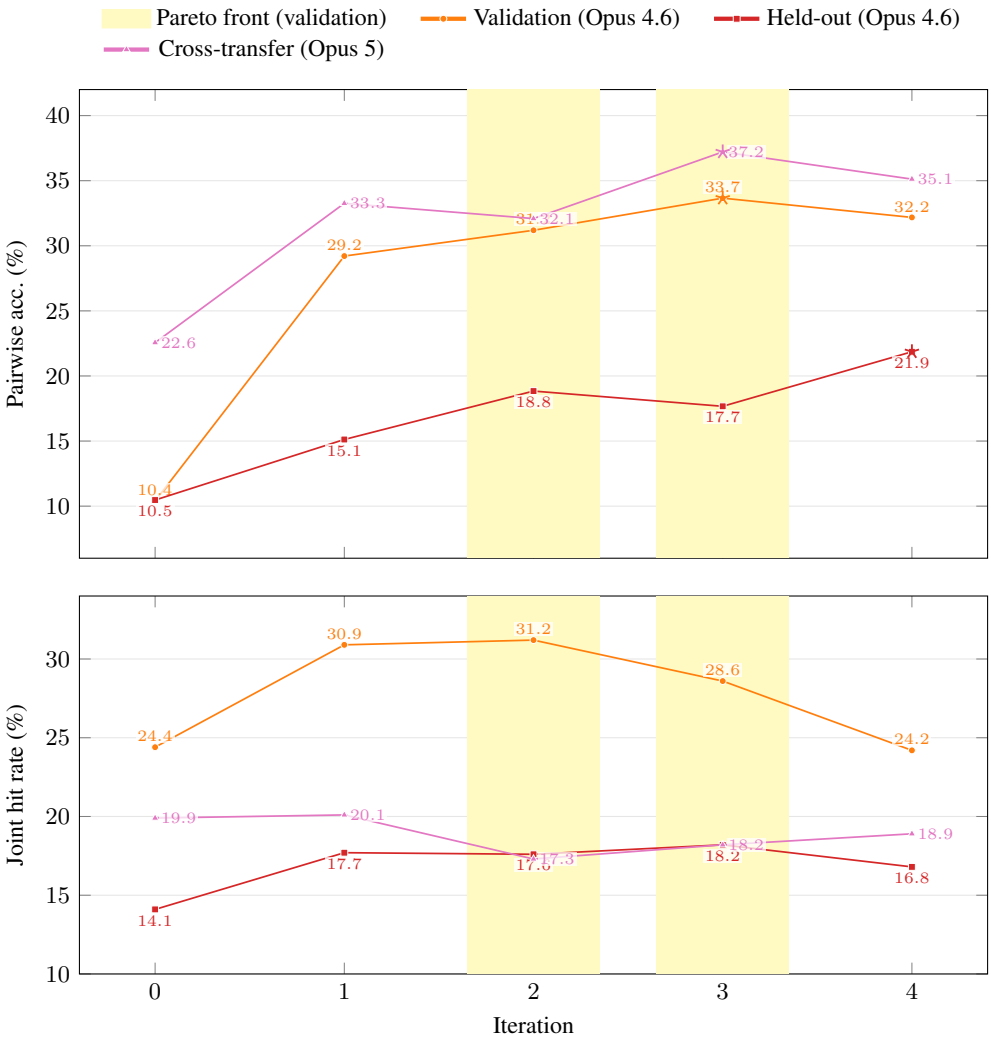


Figure 8: Expanded Fig. 3(c): Claude Code + Opus 4.6.

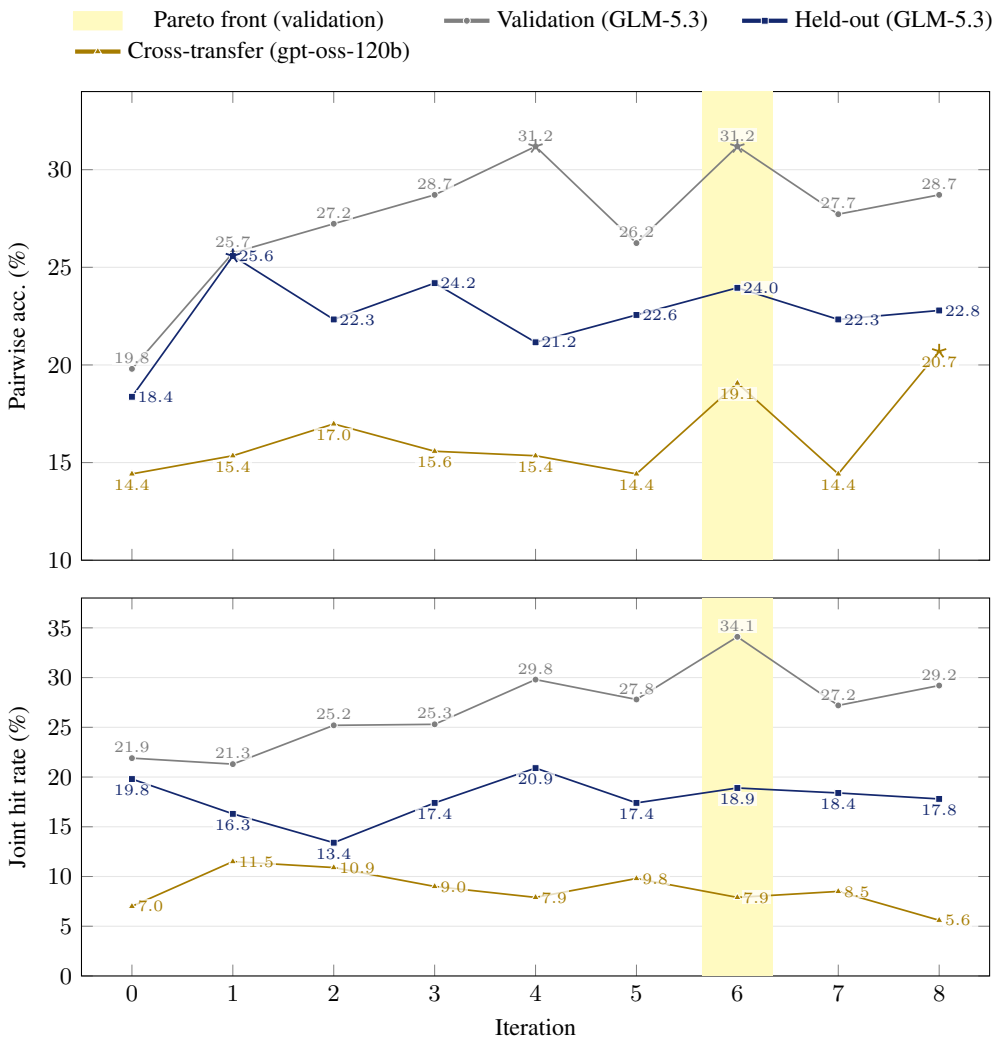


Figure 9: Expanded Fig. 3(d): Claude Code + GLM-5.3.

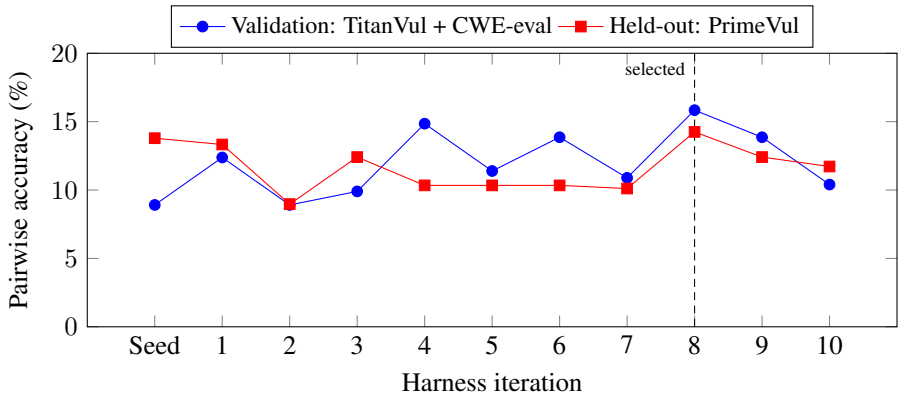


Figure 10: Qwen3-8B-AWQ pairwise accuracy by iteration. Iteration 8 was selected on validation.

Table 9: Qwen3-8B-AWQ validation results on TitanVul + CWEval (202 pairs).

Iter	Pairwise	Acc	F1	Prec	Rec	CWE acc	Loc F1	Loc hit	Parse failures
0 (seed)	8.91	50.2	0.634	0.501	0.861	13.2	23.6	44.8	0
1	12.38	52.5	0.652	0.514	0.891	16.5	24.3	46.1	0
2	8.91	52.2	0.660	0.512	0.926	9.5	24.3	44.4	0
3	9.90	51.5	0.630	0.509	0.827	8.7	25.3	47.9	0
4	14.85	54.5	0.653	0.527	0.856	6.7	20.9	41.0	0
5	11.39	52.5	0.639	0.515	0.842	6.9	23.3	43.5	0
6	13.86	53.0	0.636	0.519	0.822	7.1	22.4	42.8	0
7	10.89	51.2	0.626	0.508	0.817	7.1	22.8	43.6	0
8 (selected)	15.84	54.0	0.626	0.527	0.772	7.5	22.3	44.2	0
9	13.86	52.7	0.635	0.517	0.822	7.8	22.7	43.4	0
10	10.40	50.5	0.615	0.503	0.792	8.5	25.7	47.5	0

Iteration 8 was selected on validation. Parse failures count retained records after retries. F1, precision and recall use 0–1; other performance metrics are percentages.

Table 10: Qwen3-8B-AWQ held-out results on PrimeVul (435 pairs).

Iter	Pairwise	Acc	F1	Prec	Rec	CWE acc	Loc F1	Loc hit	Parse failures
0 (seed)	13.79	52.4	0.636	0.515	0.832	8.8	14.4	26.8	0
1	13.33	52.3	0.641	0.514	0.851	6.8	13.9	25.7	0
2	8.97	51.7	0.656	0.509	0.922	7.0	15.4	29.4	0
3	12.41	51.8	0.635	0.511	0.839	4.1	14.6	26.0	0
4	10.34	51.1	0.626	0.507	0.816	4.5	15.8	27.9	0
5	10.34	50.6	0.611	0.504	0.777	4.4	14.7	27.5	0
6	10.34	51.6	0.629	0.510	0.821	3.9	14.7	24.9	0
7	10.11	51.3	0.619	0.508	0.791	5.2	14.8	27.0	0
8 (selected)	14.25	52.5	0.602	0.518	0.717	5.1	14.2	26.3	0
9	12.41	51.1	0.622	0.507	0.802	4.6	14.2	26.4	0
10	11.72	50.8	0.610	0.505	0.770	4.5	16.8	30.1	0

Iteration 8 was selected on validation. Scores include the five overlapping pairs excluded from the main experiments. Parse failures count retained records after retries. F1, precision and recall use 0–1; other performance metrics are percentages.

Table 11: Reconstructed Inkling-Small token usage per validation sample.

Iter	harness	Completion	Total
0 (seed)	915.9	605.0	1,520.9
1	1,297.9	654.7	1,952.6
2	1,515.9	665.6	2,181.5
3	1,822.4	602.8	2,425.3
4	3,823.3	825.7	4,649.0
5	1,845.7	696.1	2,541.9
6	1,791.1	738.3	2,529.4
7	1,930.1	761.7	2,691.8
8	1,758.1	938.6	2,696.7
9	1,930.1	929.0	2,859.1
10	1,933.1	896.4	2,829.5
11	2,010.1	840.4	2,850.6
12 (selected)	2,105.1	809.2	2,914.4
13	2,387.1	808.0	3,195.1
14	2,295.1	787.5	3,082.6
15	2,786.1	915.2	3,701.3

Means over 404 functions, using the official tokenizer on logged text. Excludes hidden Codex instructions and discarded retries.

E.2 Worked example: selecting the Pareto frontier

We use the Inkling-Small run (Fig. 6) to show how MetaVul selects its Pareto frontier. Each of the 16 evaluated harnesses (the seed and iterations 1–15) is a point with two validation objectives, pairwise accuracy and joint hit rate, both maximized (Table 12). A harness *dominates* another if it is at least as good on both objectives and strictly better on at least one; the frontier is every harness that no other harness dominates.

- Iteration 12 has the highest pairwise accuracy (27.7), so no harness can dominate it.
- Iteration 6 has the highest joint hit rate (19.3), so no harness can dominate it either.
- Iteration 8 (23.3, 12.2) is also undominated: every harness with at least its pairwise accuracy (iterations 11–14) has a lower joint hit rate (6.2–9.4).
- Every other harness is dominated. For example, the seed (6.4, 15.0) is dominated by iteration 6 (20.8, 19.3), and the final iteration 15 (21.8, 6.8) is dominated by iteration 12 (27.7, 9.4).

The frontier is therefore iterations 6, 8, and 12, the yellow bands in Figs. 3 and 6. It trades pairwise accuracy against joint hit rate: iteration 12 discriminates best, while iteration 6 names and locates vulnerabilities best.

Table 12: Validation objectives for every Inkling-Small harness (%). For dominated harnesses, one dominating harness is listed.

Iter.	Pairwise	Joint	Status
0	6.4	15.0	dominated by 6
1	11.4	13.4	dominated by 6
2	15.8	13.0	dominated by 6
3	12.9	12.5	dominated by 6
4	17.3	16.7	dominated by 6
5	14.8	14.4	dominated by 6
6	20.8	19.3	**frontier**
7	15.8	14.1	dominated by 6
8	23.3	12.2	**frontier**
9	20.8	6.8	dominated by 6
10	20.8	8.8	dominated by 6
11	24.3	9.1	dominated by 12
12	27.7	9.4	**frontier**
13	23.3	8.1	dominated by 8
14	24.8	6.2	dominated by 12
15	21.8	6.8	dominated by 12

F GPT-4o Transfer Runs

Table 13 lists every harness we evaluated on GPT-4o, the backbone used for the comparison with prior methods in Sec. 6.

Table 13: Cross-transfer to GPT-4o on PrimeVul (860 functions, 430 pairs). Harnesses learned with DeepSeek V4 Flash (Codex CLI) are evaluated on GPT-4o, which took no part in optimization. Pairs gives correctly classified pairs out of 430 (samples without a prediction count as incorrect). (sel.) marks the harness selected on validation. Only the iterations shown were evaluated.

Iter	N	Pairwise	Pairs	Acc	F1	Prec	Rec	CWE acc	Loc F1	Loc hit	Parse
Harness learned with DeepSeek V4 Flash (Codex CLI)											
0 (seed)	860	16.51	71/430	51.7	0.543	0.516	0.574	14.6	15.5	27.9	0
7	860	22.09	95/430	53.5	0.593	0.527	0.677	15.1	11.9	19.6	0
8	859	18.37	79/430	54.0	0.388	0.579	0.291	20.0	16.9	26.4	1
10	860	23.49	101/430	54.2	0.495	0.551	0.449	15.5	10.7	17.1	0
11 (sel.)	860	20.70	89/430	53.0	0.500	0.534	0.470	15.8	15.0	22.3	0
12	860	23.26	100/430	54.8	0.485	0.563	0.426	14.2	18.1	25.7	0
14	860	21.86	94/430	54.4	0.453	0.566	0.377	14.8	13.0	19.8	0

G Agent Sandboxing Details

This section lists the restrictions summarized in Sec. 3.

G.1 Optimizer Agent

Claude Code optimizer.

- **Working directory:** starts in its own per-iteration workspace, `workspaces/opt_<session>_iter<N>_optimizer/`.
- **No file writes or edits:** Write and Edit are denied, since headless mode has no one to approve them.
- **Read tool limited to the workspace:** reading anywhere else needs approval and is denied.
- **`find` and `grep` limited to the workspace:** running them on outside folders is blocked.
- **Shell commands need approval**, except read-only commands and `python3`.
- **Non-interactive:** a single `-p` call with no approval harnesss, so every permission request is denied.

- **Fixed system harness:** `OPTIMIZER_SYSTEM` replaces Claude Code's default system harness.
- **Subscription auth only:** `ANTHROPIC_API_KEY` is removed from its environment.
- **Time limit:** `runner.timeout_s`.
- **Output:** only its final message is used as the new harness.
- **Validation:** the harness must keep the SCoT scaffold and verdict schema, or the iteration counts as failed.
- **Input:** only the current workspace's trajectory (harnesss, metrics, history, labeled predictions). PrimeVul is never copied in.

Codex optimizer.

- **Working directory:** its own per-iteration workspace (`--cd <workspace>`).
- **OS sandbox** with the `benchmark_readonly` permission profile:
  - read-only access to the workspace;
  - read-only access to minimal system runtime paths;
  - no other filesystem access;
  - no writes anywhere.
- **Network disabled** in the permission profile.
- **Web search disabled.**
- **No approvals:** `approval_policy = "never"`, so it cannot escalate permissions.
- **Isolated setup:** a fresh temporary Codex home with only the auth file copied in, so it loads no user config, skills, MCP servers or memories.
- **No instruction files:** `AGENTS.md` and project docs are disabled (`project_doc_max_bytes = 0`).
- **No sub-agents** (`multi_agent = false`).
- **Ephemeral session:** `--ephemeral`, and the Codex rules files are ignored (`--ignore-rules`).
- **Time limit:** `runner.timeout_s`.
- **Output:** only its final message is used as the new harness, with the same SCoT and schema validation.
- **Input:** labeled trajectory only. PrimeVul is never copied in.

G.2 Detection Agent (Evaluator)

Claude Code detection agent.

- **No tools** (`--tools ""`), so it cannot read files or run commands.
- **Empty system harness** (`--system-harness ""`).
- **Working directory `/tmp`:** no workspace and no repo instructions.
- **Fresh process per function:** no memory between calls.
- **Input:** only the rendered harness with one function. No labels, no trajectory, no metrics.
- **Time limit:** `runner.timeout_s`.

Codex detection agent.

- **Empty temporary working directory**, created fresh for each function.
- **The same `benchmark_readonly` OS sandbox:** read-only, no writes, no network.
- **Web search disabled;** no approvals, sub-agents, instruction files, user config, skills, MCP servers or memories.

- **Ephemeral session** and fresh temporary Codex home for each function.
- **Input:** only the rendered harness with one function. No labels, no trajectory, no metrics.
- **Time limit:** `runner.timeout_s`.

G.3 Workspace-Level Restrictions

These apply to all runs.

- **Separate workspaces:** the optimizer's workspace (with labels, suffix `_optimizer`) is separate from any evaluator copy, which has its labels stripped (`_copy_predictions_no_labels`).
- **Validation data only:** PrimeVul data and predictions are never placed in any workspace.
- **Fixed model weights:** only the harness text changes between iterations.

H Run Configurations

Per-run settings. In the Inkling-Small and DeepSeek V4 Flash runs, the detector runs in Codex CLI with reasoning effort set to high, served through Tinker and OpenRouter, respectively. In the Opus 4.6 run, it runs in Claude Code through our subscription, and in the GLM-5.3 run, in Claude Code through OpenRouter. In the DeepSeek V4 Flash, Opus 4.6, and GLM-5.3 runs, the optimizer uses the same agent and model as the detector. The Inkling-Small, DeepSeek V4 Flash, and GLM-5.3 runs allow up to 15 refinement iterations with a patience of 4 (GLM-5.3 stopped after iteration 8); each detector call has a 900-second timeout and up to three retries, with 30 concurrent calls in the Codex runs and 60 in the GLM-5.3 run. Opus 4.6 ran four refinement iterations because of subscription limits. The run configurations specify greedy decoding (temperature 0, top-p 1, at most 4,096 output tokens), but the Claude CLI does not forward these settings to the model. The Opus 4.6 and Opus 5 seed baselines come from standalone runs of the seed harness rather than from the discovery session.

Opus 4.6 ran through our Claude subscription rather than per-token API billing. Subscription limits motivated a maximum of five iterations; the recorded run contains the seed and four refinement iterations. OpenRouter and Tinker runs instead used hosted inference through the local LiteLLM proxy. The OpenRouter configuration pins DeepSeek V4 Flash to FP8 providers and Nemotron to BF16 providers; the Tinker Nemotron run uses the explicit BF16 model identifier. These are separate serving routes, not interchangeable pricing records. We routed Claude Code through a local LiteLLM proxy, which translated its Anthropic Messages API requests into OpenRouter chat completion requests. Each model was registered in the proxy configuration with authentication supplied through an API key (environment variable). Claude Code's auxiliary model aliases were accordingly pinned to the same target model. For GLM-5.3 experimentation, both the optimizer and evaluator used this route. For the GLM-5.3 run, each evaluation attempt had a 900-second timeout and up to three retries. The Claude CLI output cap was 32,000 tokens. Although the saved configuration lists temperature 0, top-p 1 and a 4,096-token limit, the Claude CLI model client does not forward these decoding fields; we therefore do not report them as effective sampling settings. Current provider prices are listed in App. J.1; they are not measurements of total experiment cost. Run lengths depended on inference budgets and stopping criteria. GLM-5.3 used maximum of 15 refinement iterations (T=15) and patience of 4 iterations without improvement (K=4), stopping after iteration 8.

I Context Budget

The optimizer never loads its full history into context: the history outgrows every context window we use, so it reads only the files it opens. Table 14 lists each model's context window and the size of one detection call. The optimizer's history, measured as its workspace at the last optimizer call, grows to 0.8M–3.9M tokens, larger than any context window we use, so the optimizer never loads it: it browses the read-only workspace with tools and reads only the files it opens. Each detection call, by contrast, sends one harness and one function. The largest such harness, about 77K tokens, came from DeepSeek-learned harnesses run on DeepSeek V4 Flash and GLM-4.7 Flash (windows of 1.31M and 200K); on Nemotron, whose window we set to 65.5K, the largest was 22.5K. Every detection harness fit its model's window.

Table 14: Context window and the size of one detection call (the harness plus one PrimeVul function), per model.

Model	Role	Agent	Context window	Per detection call (median / max)
Claude Opus 4.6	optimizer, detector	Claude Code	200K	3.4K / 23.7K
Claude Opus 5	detector (transfer)	Claude Code	200K	3.5K / 23.7K
GLM-5.3	optimizer, detector	Claude Code	200K[a]	5.8K / 27.8K
gpt-oss-120b	detector (transfer)	Claude Code	131K	5.8K / 27.8K
Inkling-Small	optimizer, detector	Codex CLI	256K	2.4K / 44.4K
Nemotron 3.5 Lightning	detector (transfer)	Codex CLI	65.5K[b]	2.5K / 22.5K
DeepSeek V4 Flash	optimizer, detector	Codex CLI	1.31M	5.7K / 76.9K
GLM-4.7 Flash	detector (transfer)	Codex CLI	200K	6.9K / 76.9K
GPT-4o	detector (transfer)	Codex CLI	128K	5.7K / 53.7K

[a] The model accepts 1.31M tokens; Claude Code runs it with the 200K window of the Claude model it is mapped to. [b] Set for Codex; the model accepts up to 1M. Token counts are estimated at 4 characters per token.

J Optimizer-Information Ablation

We ran the discovery loop at five information levels, from the full workspace to the current harness only, with `gpt-5.6-luna` (medium reasoning effort) as both detection and optimizer agent, one run of five iterations per level (Table 15). All levels start from the same scored seed harness and differ only in what the optimizer's workspace contains; the detection agent never sees labels at any level. For `no-labels`, fields whose position reveals the label were also removed, and sample identifiers were replaced by opaque hashes. All levels improved significantly over the seed (from 9.9 to between 19.8 and 23.3 pairwise accuracy, exact McNemar $p \leq 0.005$), and no level differed significantly from another (all $p \geq 0.33$). We consider this ablation inconclusive: it uses a single run per level, a single optimizer configuration (`gpt-5.6-luna`, an efficiency-oriented model, at medium reasoning effort) that differs from those in our main experiments, and validation data only.

Level	Optimizer workspace	It. 1	It. 2	It. 3	It. 4	It. 5	Best	Mean
`full`	harnesses, metrics, history, predictions with labels	19.80	14.36	18.81	13.37	17.33	19.80	16.73
`no-labels`	same, labels and label-revealing fields removed	16.83	19.80	19.80	20.30	17.82	20.30	18.91
`no-predictions`	harnesses, metrics, history	21.78	16.83	19.31	19.31	19.31	21.78	19.31
`no-history`	latest metrics only	16.83	19.31	21.29	23.27	19.80	23.27	20.10
`harness-only`	current harness only	16.34	20.79	19.31	15.84	21.29	21.29	18.71

Table 15: Optimizer-information ablation on the validation set (202 pairs, 404 functions), pairwise accuracy (%). All levels start from the same seed harness (9.9).

J.1 Hosted inference prices

OpenRouter entries are profile headline rates, not necessarily the providers used in our runs. Prices vary by routing, precision, caching and discounts; they do not represent experiment spending. Tinker prices include the listed 50% discount. Model names link to sources. Opus used subscription access.

Model	Service	Input	Output
GLM-5.3	OpenRouter	0.5614	1.764
DeepSeek V4 Flash 0731	OpenRouter	0.022	0.32
GLM-4.7 Flash	OpenRouter	0.06	0.40
GPT-OSS-120B	OpenRouter	0.03	0.17
Nemotron 3.5 Lightning	OpenRouter	0.065	0.18
GPT-4o	OpenRouter	2.50	10.00
Inkling-Small (64K)	Tinker	0.58	1.44
Nemotron 3.5 Lightning BF16 (64K)	Tinker	0.195	0.495

Table 16: Listed inference prices in USD per million tokens.