CARE-Bench: A Capability-Stratified, Cost-Aware Benchmark for Agentic Software-Security Detection

Andi Xia[1,*]

1 Georgia Institute of Technology, Atlanta, GA, USA
[*] Corresponding author. ✉ axia112358@gmail.com
Code available at: https://github.com/ax112358/care-bench

Abstract. Agentic systems for software-security detection have advanced quickly, from passive classifiers to cyber reasoning systems that autonomously find and patch vulnerabilities. Yet progress is hard to measure: systems are reported on incomparable datasets, with metrics that ignore cost, and on corpora whose labels are known to be noisy and prone to train-test contamination. We present CARE-Bench, an evaluation harness and protocol that makes results across heterogeneous agentic detectors directly comparable. CARE-Bench makes three design commitments. First, it is capability-stratified: every system is scored along the four levels of detection, localization, triage, and remediation, so that a single-call classifier and a multi-agent patcher are placed on the same ladder. Second, it treats cost as a first-class metric, reporting tokens, dollars, and wall-clock time, and dollars per true positive, so that accuracy and expense are reported together. Third, it is contamination-resistant, providing temporal-holdout and de-duplication controls that address the documented label-noise and leakage problems of existing corpora. CARE-Bench does not introduce a new vulnerability dataset; instead it unifies existing public corpora under one canonical schema through dataset adapters, and ships as open-source code with a reproducible runner, a metric suite, detector interfaces, and a leaderboard schema that rejects submissions omitting cost or contamination provenance. We describe the design, position it against recent benchmarks, and release the harness to support reproducible, cost-aware evaluation of the next generation of security agents.

Keywords: Benchmark, Vulnerability Detection, LLM Agents, Reproducibility, Cost-Aware Evaluation, Open Source.

1 Introduction

Background. Software vulnerabilities are a recognized national-security risk, and the volume of code now far exceeds what manual review and the limited supply of expert security engineers can sustain. In response, a new class of agentic systems couples large language models (LLMs) with planning, memory, and external tools to autonomously analyze code, reason about behavior, and identify, triage, and help remediate vulnerabilities; the DARPA AI Cyber Challenge (AIxCC) showed such cyber reasoning

systems finding and patching flaws in real open-source software largely autonomously [1, 2].

The measurement problem. This progress is difficult to measure rigorously, for three reasons. (i) Results are reported on incomparable datasets and tasks, so a number from one paper rarely transfers to another. (ii) Metrics usually ignore cost, even though an agent that issues hundreds of tool calls per file has a very different operating profile from a single prompt, and AIxCC made per-task dollar cost a headline figure [2]. (iii) Evaluation corpora are often contaminated: recent work reports that widely used datasets can be inaccurate in a large fraction of samples and that reused CVE corpora may have ground-truth labels little better than chance, while duplicate functions leak across train and test splits [3, 11]. Without a shared, cost-aware, contamination-resistant protocol, apparent improvements may reflect evaluation artifacts rather than real capability.

Our approach. We present CARE-Bench, an evaluation harness and protocol rather than a new corpus. It rests on three commitments. It is capability-stratified, scoring every system along the four levels of the detection taxonomy, namely detect, localize, triage, and remediate, so heterogeneous systems share one ladder. It treats cost as a first-class metric, reporting dollars, tokens, and time, and dollars per true positive. And it is contamination-resistant, providing temporal-holdout and de-duplication controls. CARE-Bench unifies existing public datasets through adapters into a single canonical schema and is released as open-source code with a reproducible runner, a metric suite, detector interfaces, and a leaderboard schema. Figure 1 shows the pipeline.

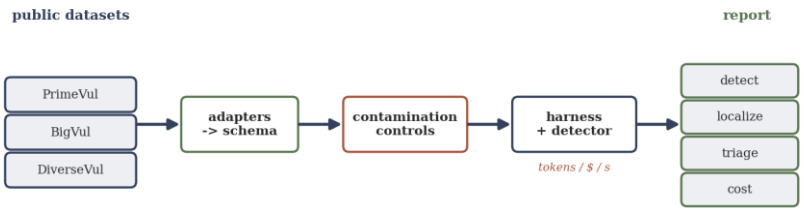

Fig. 1. The CARE-Bench pipeline. Public datasets are normalized by adapters into a canonical schema, passed through contamination controls, and run through a detector under a reproducible harness; the capability-stratified metric suite reports detection, localization, triage, and remediation quality alongside cost.

Contributions. We make three contributions. First, a benchmark design that is simultaneously capability-stratified, cost-aware, and contamination-resistant, with a canonical schema and metric suite that operationalize all three. Second, an open-source harness, with dataset adapters that unify existing public corpora without redistributing them, detector interfaces for single-step, reason-act, and multi-agent systems, and a leaderboard schema that enforces cost and contamination provenance. Third, a

positioning of CARE-Bench against recent benchmarks that shows how stratification and cost-awareness fill a gap left by task-specific evaluations.

The paper is organized as follows. Section 2 reviews related benchmarks and datasets. Section 3 presents the design. Section 4 describes the open-source harness. Section 5 reports experiments that characterize the instrument and the field, namely a meta-analysis of reported results, a harness demonstration, a contamination-control study, a metric-sensitivity analysis, and a cost model. Section 6 constructs a real dataset, CARE-Bench-Live, with the included pipeline and compares five detectors on it. Section 7 discusses limitations and Section 8 concludes.

2 Related Work

We organize prior work into six categories: function-level vulnerability datasets, repository-level datasets, secure-code-generation benchmarks, agentic-security task benchmarks, autonomous cyber reasoning systems, and evaluation-methodology studies. For each we state how CARE-Bench relates to it, and Table 2 then situates the most directly comparable benchmarks on the CARE-Bench axes.

Function-level vulnerability datasets. The dominant framing labels an isolated function vulnerable or benign. Devign learned program semantics with graph neural networks [14], BigVul paired C/C++ functions with CVE summaries and fixes [15], and DiverseVul broadened project and weakness coverage [16]. CVEfixes automated the collection of vulnerable-and-fixed code from the National Vulnerability Database, yielding 11,873 CVEs across 272 CWE types [29]. Repeated audits, however, found these corpora carry duplication and label noise that inflate reported accuracy; PrimeVul responds with de-duplicated, accurately-labeled functions [11], and SecVulEval pushes to statement-level labels over thousands of real CVEs, where even strong models reach only about 24% F1 [12]. CARE-Bench consumes these corpora through adapters rather than replacing them, and adds the cross-dataset schema and contamination controls that make results over them comparable.

Repository-level datasets. Real vulnerabilities often span functions and files, so a second line moves beyond the single function. ReposVul assembles repository-level, interprocedural vulnerability data [17], VulEval evaluates inter- and intra-procedural detection together over 4,196 CVEs, 232,239 functions, and 4,699 repositories, and reports that adding dependency context raises detection [30], and eyeballvul draws from a weekly-updated stream of disclosures to resist contamination at repository scale [13]. CARE-Bench's schema records repository and commit provenance and disclosure dates, so repository-level corpora and their temporal structure are first-class inputs to its controls.

Secure-code-generation benchmarks. A distinct family measures whether models generate insecure code rather than whether they detect it. CyberSecEval probes the security behavior and risks of frontier models [6], LLMSecEval provides natural-language prompts for security evaluation [33], and CodeLMSec elicits insecure completions to build its test set [32]. Outcome-driven efforts such as CWEval score functionality and security jointly across 119 tasks, 31 CWEs, and five languages [31], while

SecRepoBench moves secure-coding evaluation to the repository level [34]. These benchmarks target generation, the dual of detection; CARE-Bench focuses on detection but borrows their insistence on outcome-based, specification-clear scoring.

Agentic-security task benchmarks. Closest to our setting are benchmarks that evaluate agents on security tasks. JITVul links functions to their vulnerability-introducing and fixing commits across 879 CVEs and 91 weakness types, and finds ReAct agents outperform plain LLMs at distinguishing vulnerable from benign code [18]. SEC-bench builds reproducible repositories with harnesses and gold patches and reports that state-of-the-art code agents reach at most 18% on proof-of-concept generation and 34% on patching, at a construction cost under one dollar per instance [19]. SecureAgentBench evaluates code agents on 105 secure-coding tasks [20], and PatchEval assesses automated repair across languages and CWEs [21]. Each targets a single task; CARE-Bench is complementary, providing the capability ladder and the cost-aware, contamination-resistant protocol under which such task-specific results can be placed and compared.

Autonomous cyber reasoning systems. At the high-autonomy end, the DARPA AI Cyber Challenge fielded cyber reasoning systems that found and patched vulnerabilities in real open-source software largely autonomously, reporting per-task dollar cost as a headline figure [1, 2], and Project Naptime and its Big Sleep successor demonstrated agentic discovery of real flaws [5, 7]. Named tool-augmented detectors such as GPTScan [8], LLMDFA [9], and the multi-agent LLM-SmartAudit [10] occupy the middle of the ladder. CARE-Bench is the measurement substrate these systems currently lack: a common protocol on which their very different capabilities and costs can be reported in one place.

Evaluation methodology and contamination. A final thread studies evaluation itself. Audits report that widely used datasets can be inaccurate in a large fraction of samples and that reused CVE corpora may have ground-truth labels near chance [3, 19], while general agent benchmarks such as SWE-bench and the SWE-agent interface established the reproducible-runner discipline we adopt [27, 28]. CARE-Bench operationalizes this thread directly, building temporal-holdout and de-duplication controls into the protocol rather than leaving them to each evaluator.

Synthesis and gap. Read together, function- and repository-level datasets supply ground truth, secure-generation benchmarks measure the dual problem, agentic-task benchmarks supply task-specific evaluations, cyber reasoning systems supply the high-autonomy frontier, and methodology studies supply reproducible runners and contamination warnings. Yet no existing artifact scores heterogeneous detectors on one capability ladder while making cost first-class and contamination explicit. Table 2 makes the gap concrete: across the most comparable benchmarks, capability coverage is partial, cost is rarely reported, contamination control is uneven, and almost none unify multiple datasets under one schema. A single-call classifier and a multi-agent cyber reasoning system remain reported in incommensurable terms. CARE-Bench is designed to close this gap, and the rest of the paper develops and releases it.

Table 2. Existing benchmarks and datasets situated on the CARE-Bench design axes. A check denotes the property is a stated design goal of that work; partial denotes limited support. The pattern motivates a unifying, capability-stratified, cost-aware, contamination-resistant protocol.

Benchmark / dataset	Capability span	Cost	Contam.	Multi-DS
Devign/BigVul [14,15]	detect	no	no	no
PrimeVul [11]	detect	no	dedup	no
SecVulEval [12]	detect-localize	no	partial	no
eyeballvul [13]	detect	no	temporal	no
VulEval [30]	detect (repo)	no	temporal	no
JITVul [18]	detect	no	commit-paired	no
SEC-bench [19]	triage-remediate	yes	reproduced	no
PatchEval [21]	remediate	partial	sandboxed	no
CARE-Bench (ours)	**detect-remediate**	**first-class**	**temporal+dedup**	**yes**

3 Benchmark Design

CARE-Bench rests on a canonical schema and three design commitments. We describe each in turn.

3.1 Canonical Schema

Every dataset adapter normalizes native records into a single Sample schema, so that tasks, metrics, and detectors are written once and run against any source. A sample carries the code under analysis, its language, the ground-truth label, and, when available, the implicated CWE identifiers, the vulnerable line numbers, a reference patch, and a disclosure date used for temporal holdout. Keeping the schema small and explicit is what makes cross-dataset comparison well defined.

3.2 Capability Stratification

What and why. Agentic detectors differ in ambition: some only flag a fragment as vulnerable, others localize the fault, judge its exploitability, or propose a patch. Scoring them with one metric obscures these differences. CARE-Bench therefore stratifies evaluation into four cumulative levels, detect, localize, triage, and remediate, mirroring the detection taxonomy. A system evaluated at a higher level is also scored at the lower ones, so the ladder is monotone and a system's profile, not a single number, is the unit of comparison.

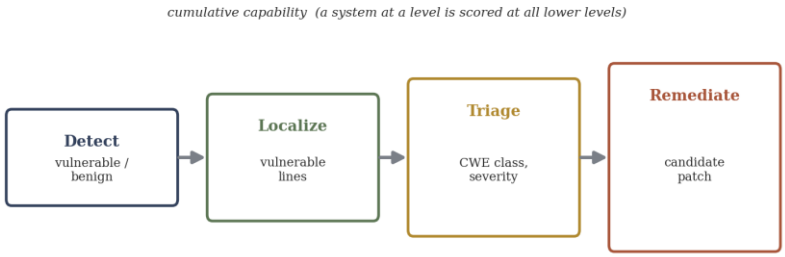

Fig. 2. The capability ladder. The four levels are cumulative: a system evaluated at a higher level is also scored at every lower one, so CARE-Bench compares capability profiles rather than a single number.

Metrics per level. Detection reports precision, recall, F1, accuracy, and balanced accuracy; the last is emphasized because the heavy class imbalance of real corpora lets a trivial always-benign predictor score high plain accuracy. Localization compares predicted and gold line sets via precision, recall, F1, intersection-over-union, and a top-k hit rate. Triage scores CWE prediction as a multi-label problem with exact-match, Jaccard, and any-hit. Remediation is delegated to a pluggable patch verifier, since sound patch validation requires execution. Table 1 summarizes the levels.

Table 1. The four capability levels, the output each requires, and the metrics CARE-Bench reports.

Level	Required output	Metrics
Detect	vulnerable / benign	precision, recall, F1, balanced accuracy
Localize	vulnerable line numbers	line P/R/F1, IoU, top-k hit
Triage	CWE class(es)	exact-match, Jaccard, any-hit
Remediate	candidate patch	pluggable patch verifier (exec-based)

3.3 Cost as a First-Class Metric

Every run records, per sample, the tokens consumed, dollars spent, and wall-clock seconds, which the harness aggregates and from which it derives dollars per true positive. The per-true-positive figure is the quantity an operator cares about, and it is defined to be infinite when a run finds nothing, so a zero-recall system is not flattered by a small denominator. Reporting cost beside accuracy makes the accuracy-versus-expense trade-off explicit rather than hidden, and reflects the operational framing of competitions that report per-task dollar cost [2, 19].

3.4 Contamination Resistance

Because static corpora are prone to label noise and train-test leakage [3, 11], CARE-Bench provides two controls. Temporal holdout keeps only samples disclosed on or after a cutoff date, approximating a future-proof split so a model cannot have memorized a vulnerability disclosed after its training cut, in the spirit of stream-sourced benchmarks [13]. De-duplication removes whitespace-normalized duplicate functions that otherwise straddle the split and inflate scores. Both are one-line transformations over the canonical schema, so any adapter inherits them.

4 The Open-Source Harness

CARE-Bench is released as a small, dependency-free Python package so that runs are reproducible and easy to audit. Its structure follows the design: a schema module, dataset adapters with a registry, a metric suite organized by capability level, detector interfaces with trivial baselines, a synchronous runner, an evaluator, and contamination utilities. Listing 1 shows the end-to-end flow.

Listing 1. Running a detector over a dataset and scoring it (Python).

```
from carebench.datasets import get_adapter
from carebench.agents import DeterministicHash
from carebench import harness, evaluate, contamination

samples = get_adapter('primevul', '/path/to/primevul').load()
samples = contamination.temporal_holdout(samples, '2024-01-01')
samples = contamination.dedup_by_code(samples)

result = harness.run(DeterministicHash(), samples, level='triage')
report = evaluate.evaluate(result, samples)
print(report['detection'], report['cost'])
```

Datasets without redistribution. CARE-Bench ships no third-party data. Each adapter documents the official source of its corpus and normalizes the public release into the canonical schema, leaving licensing with the original authors. Adapters are provided for PrimeVul, BigVul, and DiverseVul, and the registry makes adding a new corpus a matter of implementing one conversion method. A small synthetic toy set is bundled solely so the harness and tests run with no downloads; it is explicitly not a benchmark.

Detectors and baselines. A detector implements a single predict method returning a structured prediction with cost fields. The package provides interface skeletons for single-step LLM, reason-act, and multi-agent detectors, matching the architecture axis of the taxonomy, plus two trivial deterministic baselines that run anywhere and make the harness self-contained. The baselines are intentionally weak sanity-check floors, not methods.

Leaderboard schema. A submission is a JSON record carrying the metric report and required provenance: the model, the agent architecture, whether tools were used, the contamination control applied, and the date. The validator rejects any submission that omits the cost or contamination fields, encoding the benchmark's commitments as a hard requirement rather than an option.

5 Experiments and Analysis

Because CARE-Bench is an instrument rather than a detector, our experiments characterize the instrument itself and the field it measures, not a new system. We (i) aggregate the reported state of the field, (ii) demonstrate the harness end to end, (iii) show the contamination controls remove exactly the structure they target, (iv) analyze the sensitivity of the metrics to class imbalance, and (v) model the cost metric. Every number below is either a real figure reported by a cited source or a genuine output of the released code on controlled synthetic data or closed-form formulas; none is a claim about an unmeasured real detector.

5.1 The Reported State of the Field

Figure 3 aggregates headline numbers reported by recent systems and benchmarks. The bars are taken verbatim from the cited sources and are measured on different data under different protocols, so they are deliberately not comparable; the figure visualizes precisely the fragmentation CARE-Bench is built to remedy. Two regularities are visible. Numbers vary by a factor of four across studies, from around 18% on proof-of-concept generation [19] to about 87% precision for a tool-augmented dataflow detector on its own benchmark [9], and tasks that demand higher capability, such as patch generation, report markedly lower success than function-level detection.

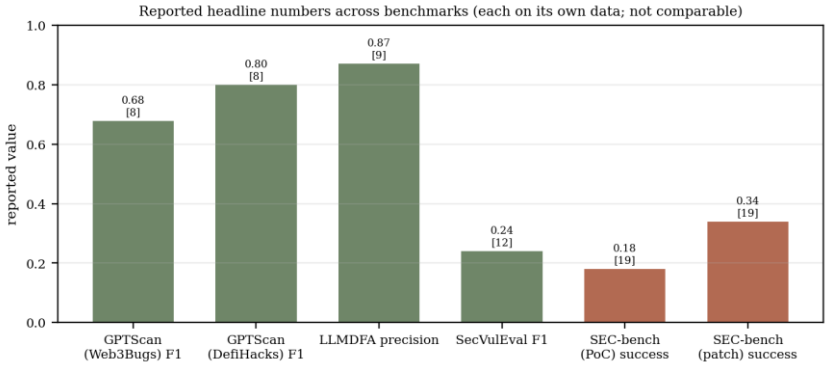

Fig. 3. Headline numbers reported by recent systems and benchmarks, each on its own data and therefore not directly comparable (citations on bars). Sage bars are accuracy-family metrics; clay bars are task success rates. The spread motivates a unifying protocol.

Table 3. The same reported results situated on the CARE-Bench axes, with each figure cited to its source. Figures are measured on different data and are not directly comparable across rows.

Benchmark	Top level	Cost reported?	Reported headline (from source)
JITVul [18]	detect	no	ReAct agents > plain LLMs on vuln/benign
SEC-bench [19]	remediate	yes ($/inst.)	<=18% PoC gen, <=34% patching
SecVulEval [12]	detect	no	best model ~24% F1 (statement-level)
GPTScan [8]	detect	yes ($/kLoC)	67.8-80% F1 on its datasets
CARE-Bench (ours)	remediate	yes (first-class)	protocol; unifies the above

5.2 Harness Demonstration

To show the pipeline runs end to end, we evaluate the two bundled trivial baselines on the synthetic toy set. We stress that Table 4 is a harness demonstration on synthetic data, not a result about any real system: the toy set exists only to exercise the code paths. The always-vulnerable baseline attains perfect recall and a balanced accuracy of 0.50, the value expected of a degenerate predictor, while the deterministic-hash baseline behaves like a coin flip. The point is that one command produces a complete, well-formed report with detection, localization, triage, and cost sections.

Table 4. Harness demonstration on the bundled synthetic toy set (six samples). Trivial baselines on illustrative data, included only to show the pipeline runs; not a result about any real detector.

Baseline (toy set)	Recall	Bal. acc.	USD / TP
always-vulnerable	1.00	0.50	0.00
deterministic-hash	0.67	0.33	0.00

5.3 Contamination-Control Study

We next verify that the contamination controls remove exactly the structure they target. We construct a synthetic corpus of 1,000 samples in which 30% are whitespace-variant duplicates of earlier samples and disclosure dates are spread uniformly over 2019-2025, then apply the two controls. The results, shown in Figure 4, match the injected structure to within sampling noise: de-duplication removes the 300 planted duplicates, leaving the 700 unique functions, and a temporal holdout at January 2024 retains 287 samples, an observed 28.7% against an expected two-of-seven years, or 28.6%. This is a test of

the instrument, run on the released code; it confirms the controls behave as specified, which is what lets a user trust them on real corpora whose contamination is unknown.

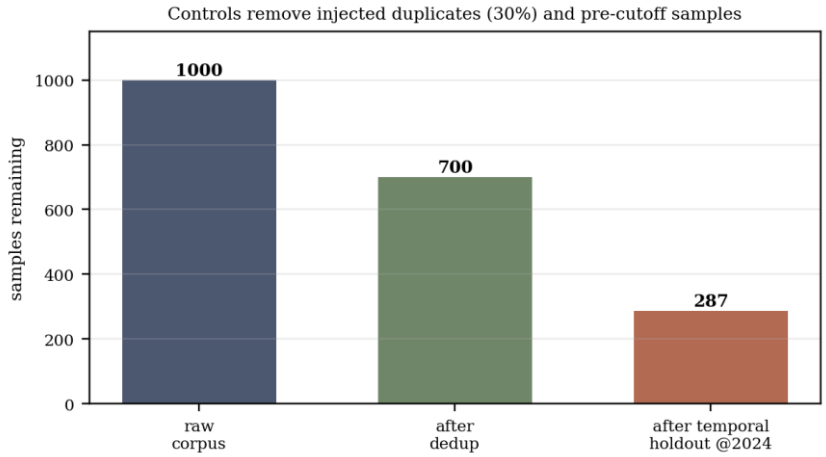

Fig. 4. Contamination controls on a synthetic corpus with known structure (1,000 samples, 30% injected duplicates, disclosure dates over 2019-2025). De-duplication recovers the 700 unique functions; temporal holdout at 2024 retains 28.7%, matching the expected 28.6%.

5.4 Metric-Sensitivity Analysis

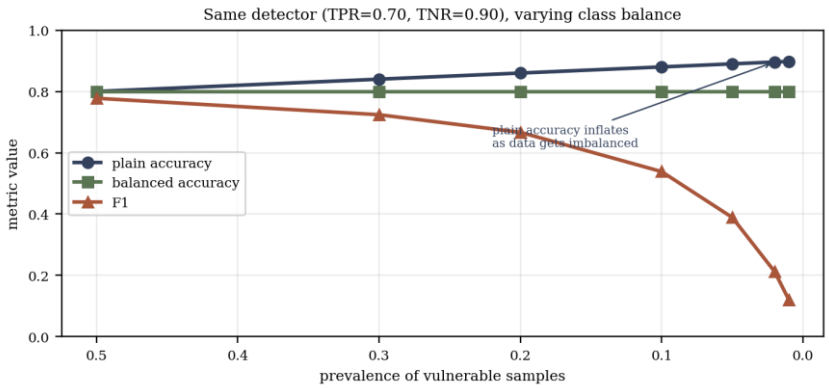

Fig. 5. Metric sensitivity to class imbalance for a fixed detector (TPR=0.70, TNR=0.90), computed with the released code. Balanced accuracy is invariant; plain accuracy inflates and F1 collapses as vulnerable samples become rare, motivating CARE-Bench's emphasis on balanced accuracy.

A central design choice is to emphasize balanced accuracy. Figure 5 shows why. Holding a detector fixed at a true-positive rate of 0.70 and a true-negative rate of 0.90, we sweep the prevalence of vulnerable samples from 50% down to 1%, the regime of real

corpora, and recompute the metrics with the released code. Balanced accuracy stays constant at 0.80, correctly reflecting the unchanged detector, whereas plain accuracy inflates from 0.80 to 0.90 purely because benign samples dominate, and F1 collapses from 0.78 to 0.12 as precision craters. A benchmark that ranked systems by plain accuracy on an imbalanced corpus would therefore reward the data distribution, not the detector, which is the failure CARE-Bench avoids by foregrounding balanced accuracy and reporting the full confusion matrix.

5.5 Cost-Model Analysis

Finally we make the cost metric concrete. For a detector with recall r on a corpus of prevalence p and a fixed per-sample cost c, the dollars per true positive are c divided by the product of p and r, since only a p-fraction of samples are vulnerable and only an r-fraction of those are found. Figure 6 plots this relationship at p=0.10 for three per-sample costs spanning a cheap single call, a mid-range pipeline, and an expensive agent. The curves are closed-form and illustrative, included to show how CARE-Bench's cost metric exposes trade-offs rather than to estimate any specific system: a detector can be an order of magnitude more expensive per call yet competitive per finding if its recall is high, and conversely a cheap detector with low recall can be costly per true positive. Reporting cost per finding beside accuracy is what surfaces this trade-off.

6 CARE-Bench-Live: A Constructed Dataset and Live Comparison

To show that CARE-Bench supports the defining activity of a benchmark, comparing different methods on a freshly constructed dataset, we built one with the included pipeline and ran several detectors on it. Unlike the synthetic demonstrations above, every number in this section is a real measurement on real CVE-derived code; we report it as observed, including where strong methods do poorly.

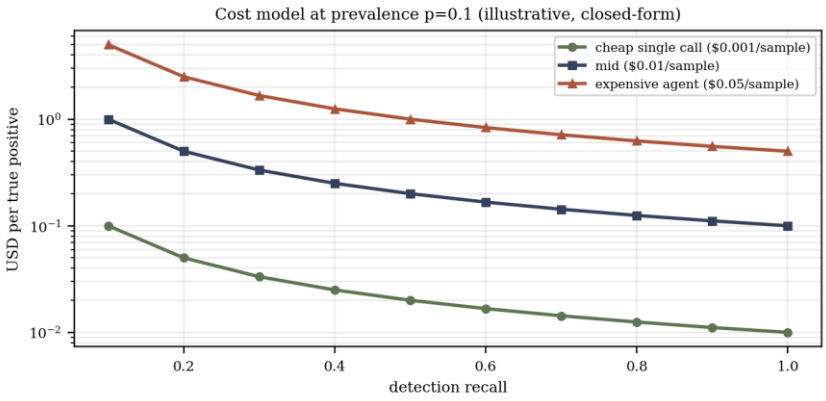

Fig. 6. Cost model: dollars per true positive versus recall at prevalence p=0.10, for three per-sample costs (log scale). Closed-form and illustrative; it demonstrates the cost metric, not any particular detector.

6.1 Construction Pipeline

The dataset, CARE-Bench-Live, is built by an auditable seven-stage pipeline that needs only a GitHub token and an optional NVD key, both read from the environment. The stages are: query the NVD CVE API over a date window and CWE filter; keep only CVEs whose references contain a GitHub commit; fetch each fixing commit's changed files with their before and after contents; extract the enclosing functions around the changed lines (the before version is the vulnerable sample, the after version a paired benign one); construct same-project and cross-project negatives; assign a label-confidence tier; and apply de-duplication and an optional temporal holdout before emitting the canonical schema. Two engineering details proved necessary in practice and are recorded for reproducibility: NVD date parameters must be sent in RFC 3339 form, and the date range must be paginated in roughly 120-day windows, since a single large window times out.

Three findings from real construction. Building from real sources surfaced three facts that shaped the design. First, the canonical examples of severe vulnerabilities, Heartbleed (CVE-2014-0160), Log4Shell (CVE-2021-44228), and Shellshock (CVE-2014-6271), carry no GitHub commit link in their NVD references, so a fixing commit cannot be resolved from NVD alone; the pipeline therefore supports a supplemental patch-commit field. Second, source repositories suffer link rot: the canonical bash repository returns 404 and a mirror must be used, so commit reachability needs a fallback. Third, real fixes routinely span multiple files and functions, which means most extracted samples cannot be cleanly localized to a single function and so receive the lower silver or bronze tier; this is not a defect but direct evidence for the tiered-label design.

6.2 Dataset Statistics

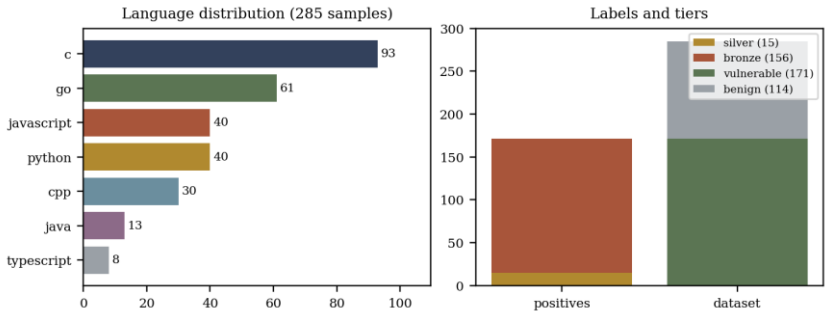

Fig. 7. CARE-Bench-Live composition: language distribution (left) and label/tier breakdown (right). The dominance of the bronze tier is a direct consequence of multi-file real-world fixes.

A single run over five high-frequency weakness types (CWE-79, CWE-89, CWE-787, CWE-125, CWE-416) across a one-year window, after de-duplication, yields 285 samples: 171 vulnerable and 114 benign, a 0.60 prevalence. The corpus spans seven languages, led by C (93), Go (61), JavaScript and Python (40 each), and C++ (30). Confirming the third construction finding, the positive labels are overwhelmingly bronze (156) with only 15 silver, because real fixes are seldom single-function. Figure 7 shows the composition.

6.3 Multi-Method Comparison

Table 5. Detection results on CARE-Bench-Live (n=285, prevalence 0.60). Cost is per true positive; it is zero for methods that call no model. The two LLM methods are served by DeepSeek; dollar costs use published per-token rates (estimates). All numbers are real measurements.

Method	Prec.	Recall	F1	Bal. acc.	USD/TP
Always-vulnerable (trivial)	0.600	1.000	0.750	0.500	0.00
Deterministic-hash (trivial)	0.552	0.462	0.503	0.450	0.00
Heuristic SAST (offline)	0.526	0.058	0.105	0.490	0.00
Single-step LLM (DeepSeek)	0.484	0.269	0.346	0.420	0.0048
ReAct agent (DeepSeek)	0.550	0.292	0.382	0.466	0.0045

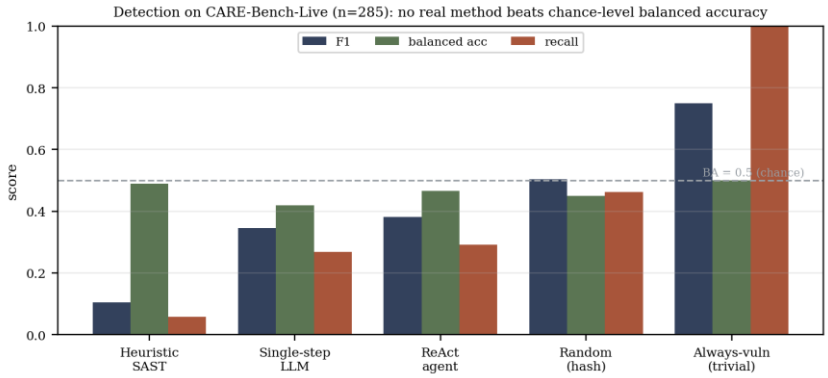

Fig. 8. Detection on CARE-Bench-Live. The dashed line marks chance-level balanced accuracy. No non-trivial method exceeds it, while the always-vulnerable baseline attains the highest F1 purely through perfect recall on a 0.60-prevalence set, illustrating why F1 alone is misleading.

We evaluated five detectors on CARE-Bench-Live: three trivial or offline baselines (always-vulnerable, a deterministic hash, and a rule-based static analyzer) and two real LLM-backed detectors served by DeepSeek, a single-step classifier and a ReAct agent with grep and line-reading tools. Table 5 reports detection-level results; the LLM methods also report real dollar cost. Dollar figures use DeepSeek's published per-token rates and are estimates, since pricing changes over time; token counts are exact.

Reading the results. Three observations stand out, and we state them plainly. First, no learning-based or agentic method exceeds chance-level balanced accuracy: the best real method, the ReAct agent, reaches 0.466, below 0.5, and the single-step LLM 0.420, while the rule-based analyzer sits at 0.490 with very low recall. Second, the trivial always-vulnerable predictor obtains the highest F1, 0.750, purely because recall is 1.0 on a set where 60% of samples are vulnerable, a textbook illustration of why CARE-Bench foregrounds balanced accuracy and reports the full confusion matrix rather than F1 alone. Third, the ReAct agent improves on the single-step LLM in precision (0.550 vs 0.484), F1 (0.382 vs 0.346), and balanced accuracy (0.466 vs 0.420) at essentially equal cost per finding, consistent with prior reports that reason-act tool use helps [18], though the absolute gains are modest.

Capability and cost. At the triage level the picture is similar (ReAct F1 0.379, single-step 0.372, heuristic 0.105), and the cost metric earns its place: the heuristic analyzer is free but finds only 10 true positives, whereas the LLM methods find around 50 at roughly 0.0045 USD each, so the cheaper-per-call method is not the cheaper-per-finding method once recall is taken into account. These results are not a verdict on any system; they are a snapshot on a small, freshly built, bronze-heavy corpus, and they show that CARE-Bench cleanly surfaces where current methods stand and what they cost. The low absolute scores echo the broader literature, where real-world detection and repair remain hard [12, 19], and they argue for the larger, execution-verified datasets the pipeline is designed to grow toward.

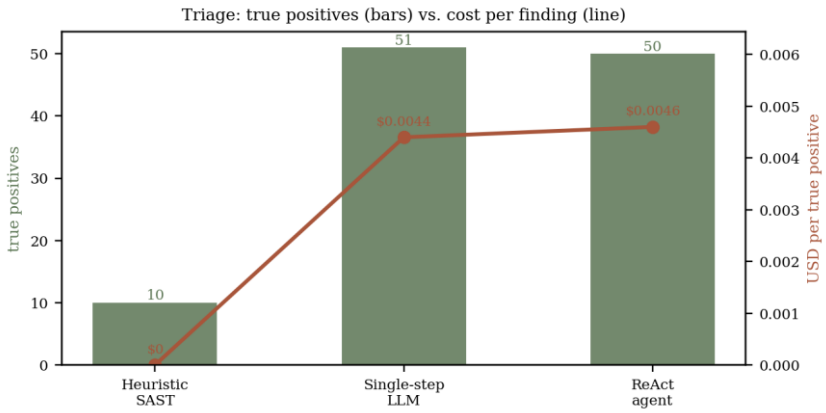

Fig. 9. Triage on CARE-Bench-Live: true positives found (bars) versus cost per true positive (line) for the three runnable methods. The free heuristic finds far fewer true positives; the LLM methods find about five times as many at a small, similar per-finding cost.

7 Limitations and Discussion

Scale and tier of the live snapshot. The CARE-Bench-Live results are a single 285-sample snapshot whose positive labels are overwhelmingly bronze, so the absolute numbers carry real sampling noise and should be read as a demonstration of the protocol, not a definitive ranking of methods. The pipeline is designed to grow the corpus across more weakness types and date windows and to promote samples to higher tiers via execution verification; larger, gold-tier runs are the natural next step.

Inherited label noise. CARE-Bench consumes existing corpora, so it inherits their label errors; its contamination controls mitigate leakage but cannot fix wrong ground truth. Pairing adapters with execution-based verification, as reproducible-artifact benchmarks do [19], is the natural remedy and fits the pluggable-verifier design.

Remediation scoring. Sound patch evaluation requires building and running code, which is language- and project-specific. CARE-Bench specifies the interface and leaves the verifier pluggable rather than prescribing one, a deliberate scope choice that keeps the core dependency-free.

Cost portability. Dollar cost depends on model pricing, which changes over time; CARE-Bench therefore records tokens and seconds alongside dollars, so runs remain interpretable as prices move.

Threats to validity. The bundled toy results are illustrative only, as stated, and must not be read as system comparisons. The cross-benchmark table reports third-party numbers measured on different data; we present it to motivate a common protocol, not to rank systems.

8 Conclusion

Agentic software-security detection is advancing rapidly, but its evaluation is fragmented, cost-blind, and contamination-prone. CARE-Bench addresses these together: it scores heterogeneous detectors on one capability ladder, makes cost a first-class and mandatory metric, and bakes contamination controls into the protocol, all without introducing a new corpus, by unifying existing public datasets through adapters. We release the harness, its metric suite, detector interfaces, and a leaderboard schema that enforces cost and contamination provenance, so that the next generation of security agents can be compared reproducibly and on the terms that matter operationally. We hope CARE-Bench becomes a shared substrate the community extends with new adapters, verifiers, and submissions.

References

1. Defense Advanced Research Projects Agency (DARPA): AI Cyber Challenge (AIxCC). https://aicyberchallenge.com (2023-2025), accessed 2026.
2. Defense Advanced Research Projects Agency (DARPA): AI Cyber Challenge Final Competition results. https://www.darpa.mil/news/2025/aixcc-results (2025), accessed 2026.
3. Authors of the survey: LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. ACM Computing Surveys (2025). Preprint arXiv:2502.07049.
4. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing Reasoning and Acting in Language Models. In: ICLR (2023). arXiv:2210.03629.
5. Glazunov, S., Brand, M.: Project Naptime: Evaluating Offensive Security Capabilities of Large Language Models. Google Project Zero, 20 June 2024.
6. Bhatt, M., et al.: CyberSecEval 2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models. Meta AI (2024). arXiv:2404.13161.
7. Google Project Zero and Google DeepMind: From Naptime to Big Sleep: Using Large Language Models to Catch Vulnerabilities in Real-World Code. Blog post, October 2024.
8. Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In: ICSE (2024). doi:10.1145/3597503.3639117.
9. Wang, C., et al.: LLMDFA: Analyzing Dataflow in Code with Large Language Models. In: NeurIPS (2024). arXiv:2402.10754.
10. Wei, Z., et al.: Advanced Smart Contract Vulnerability Detection via LLM-Powered Multi-Agent Systems. IEEE Transactions on Software Engineering 51(10), 2830-2846 (2025). doi:10.1109/TSE.2025.3597319.
11. Ding, Y., et al.: Vulnerability Detection with Code Language Models: How Far Are We? (PrimeVul). arXiv:2403.18624 (2024).
12. Ahmed, S., et al.: SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection. arXiv:2505.19828 (2025).
13. Chauvin, T.: eyeballvul: A Future-Proof Benchmark for Vulnerability Detection in the Wild. arXiv:2407.08708 (2024).
14. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: NeurIPS (2019).
15. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries (BigVul). In: MSR, pp. 508-512 (2020).

16. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In: RAID, pp. 654-668 (2023).
17. Wang, X., Wen, M., Chen, Y., Liao, Q.: ReposVul: A Repository-Level High-Quality Vulnerability Dataset. In: ICSE Companion (2024). arXiv:2401.13169.
18. Yildiz, A., Teo, S.G., Lou, Y., Feng, Y., Wang, C., Divakaran, D.M.: Benchmarking LLMs and LLM-based Agents in Practical Vulnerability Detection for Code Repositories (JITVul). In: ACL (2025).
19. Lee, H., Zhang, Z., Lu, H., Zhang, L.: SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. arXiv:2506.11791 (2025).
20. Authors of SecureAgentBench: SecureAgentBench: Benchmarking Code Agents on Secure Code Generation. arXiv:2509.22097 (2025).
21. Authors of PatchEval: PatchEval: A New Benchmark for Evaluating LLMs on Patching Real-World Vulnerabilities. arXiv:2511.11019 (2025).
22. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In: NeurIPS (2022). arXiv:2201.11903.
23. Schick, T., et al.: Toolformer: Language Models Can Teach Themselves to Use Tools. In: NeurIPS (2023). arXiv:2302.04761.
24. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language Agents with Verbal Reinforcement Learning. In: NeurIPS (2023). arXiv:2303.11366.
25. Hong, S., et al.: MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. In: ICLR (2024). arXiv:2308.00352.
26. Feng, Z., et al.: CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: Findings of EMNLP, pp. 1536-1547 (2020). arXiv:2002.08155.
27. Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O.: SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In: NeurIPS (2024). arXiv:2405.15793.
28. Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In: ICLR (2024). arXiv:2310.06770.
29. Bhandari, G., Naseer, A., Moonen, L.: CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software. In: PROMISE, pp. 30-39 (2021). doi:10.1145/3475960.3475985.
30. Wen, X., et al.: VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection. arXiv:2404.15596 (2024).
31. Peng, J., Cui, L., Huang, K., Yang, J., Ray, B.: CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. In: IEEE/ACM Intl. Workshop on LLMs for Code (LLM4Code), pp. 33-40 (2025). arXiv:2501.08200.
32. Hajipour, H., et al.: CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In: IEEE SaTML (2024).
33. Tony, C., Mutas, M., Ferreyra, N.E.D., Scandariato, R.: LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. In: MSR, pp. 588-592 (2023).
34. Dilgren, C., Chiniya, P., Griffith, L., Ding, Y., Chen, Y.: SecRepoBench: Benchmarking LLMs for Secure Code Generation in Real-World Repositories. arXiv:2504.21205 (2025).