ChatGPT

Comparative Analysis of AI Coding Agent Benchmarks

SWE-Bench Verified and Variants (Lite, Bash Only, etc.)

A. Source Materials:
- **Original SWE-Bench Paper:** *"Can Language Models Resolve Real-World GitHub Issues?"* (Princeton, 2023) [1].
- **OpenAI SWE-Bench Verified Report:** OpenAI Research Blog (Aug 2024) [2] [3] – announced a human-validated subset.
- **GitHub Repo:** *SWE-bench: Real-world software issues from GitHub* [4].
- **Hugging Face Dataset:** `princeton-nlp/SWE-bench_Verified` (500 validated issues) [5].
- **Leaderboard/Website:** swebench.com – hosts leaderboards for Full, Verified, Lite, Bash-Only, Multilingual, Multimodal splits [6] [7].

B. Benchmark Deep Dive:
- **Goal:** Evaluate LLM-based *coding agents* on **resolving real GitHub issues** by generating code patches that fix failing tests [2] [8]. It aims to move beyond toy problems to genuine software bug fixing in-context.
- **Gap Addressed:** Tackles the *reality gap* in coding evals – prior benchmarks (e.g. HumanEval) use synthetic or single-file prompts, whereas SWE-Bench uses **full repositories and issue threads** to assess autonomous coding in realistic settings [1].
- **Agent Type & Interface:** Agents are given a **repository codebase** plus an **issue description**, and they must autonomously edit or create files to resolve the issue [9] [10]. This tests tool-using agents (with file I/O and code execution) rather than just static code generation.
- **Task Format:** Each task comes from a *real GitHub pull request*. It includes a problem statement (issue text) and is evaluated against the project's unit tests: *FAIL_TO_PASS* tests that initially fail (and should pass after the fix) and *PASS_TO_PASS* regression tests that must remain passing [11] [12]. The agent does **not** see the tests, only the issue and codebase.

C. Dataset Construction Methodology:
- **Sources & Size:** The full SWE-Bench test set contains *~2,294* issues drawn from 12 popular open-source Python repositories [13] [14]. SWE-Bench **Lite** is a 300-issue subset curated to be easier/less costly to evaluate [15]. **Verified** is a 500-issue subset that underwent human screening for solvability [16] [17]. **Bash-Only** uses the same 500 Verified tasks but constrains agents to a minimal Bash-based scaffold (the "mini-SWE-agent" environment) to test capability with only CLI editing [18] [7]. Specialized splits also cover **Multilingual** (issues from multi-language repos) and **Multimodal** (issues involving images or GUI, ~517 tasks) [19] [15]. All issues are real and come with ground-truth patches and test cases.
- **Difficulty & Validation:** The Verified subset was constructed by **professional developer annotators** who filtered out issues deemed infeasible (e.g. vague reports, mis-specified tests) [16] [20]. This addressed the finding that some original tasks were unsolvable or misleading, which caused benchmarks to underestimate agent performance [21] [22]. Verified tasks skew slightly easier than the original (fewer truly "impossible" bugs) but still represent realistic bug-fix complexity [23] [24]. Lite further skews to trivial fixes

(few tasks estimated >1 hour by humans [23]). Bash-Only is not easier per se, but tests a restricted action space.

D. Evaluation Framework:

- **Metrics:** The primary metric is **% Resolved**, i.e. the percentage of issues for which the agent's patch passes *all* tests (both fix-specific and regression tests) [25] [26]. Each task is essentially pass/fail. Leaderboards report this success rate. Secondary metrics include efficiency (cost or steps to solve) on some leaderboards [27], but correctness is paramount.
- **Execution & Verification:** Agents are evaluated in a sandbox where they can iterate on code. When an agent submits a proposed code edit, the evaluation harness runs the project's test suite. Passing all tests signifies success [25]. Failing tests mean the issue is not resolved. Because tests are the oracle, this is an **automated, binary evaluation** – no human judge needed if tests are good. (In Verified, test quality was improved/ensured by the human annotation process [28] [3].)
- **Benchmark Variants:** *Full vs. Lite vs. Verified* splits allow tiered evaluation (with Verified considered the new standard) [29] [22]. *Bash-Only* uses the same tasks but restricts the agent's tool usage to examine how well models can operate under limited scripting capabilities [18]. All variants share the same test-based verification method. The infrastructure is containerized (Docker-based) for reproducibility [30], and an official harness (the SWE-bench CLI and *mini-SWE-agent* scaffold) is provided for consistent agent evaluation across submissions [31] [18].

E. Uniqueness Factor:

- **Realism:** SWE-Bench was the *first benchmark to use full multi-file real-world codebases and issue discussions* as input [32] [9]. This makes it far more realistic than prior single-function coding tests. Agents must handle reading and modifying a *large codebase*, similar to a human contributor addressing a bug.
- **Autonomy Focus:** It specifically evaluates **autonomous coding agents** – i.e. can an agent plan and execute a fix without step-by-step user prompting? The tasks require understanding a bug report, locating the bug, editing code, and verifying the fix, all unsupervised. This inspired research into agent "scaffolds" that loop through these steps.
- **Community & Extensions:** SWE-Bench spawned a *family* of benchmarks broadening the evaluation surface [33]. The **Bash-Only** variant is unique in testing an agent's ability to solve coding tasks purely with shell commands (no editor), reflecting scenarios like DevOps scripting [18]. **Multilingual** and **Multimodal** splits extend the concept to polyglot codebases and issues involving images/UX [34]. This modular extensibility is a unique aspect – it's not a single benchmark but an evolving suite.
- **Human-Validated Subset:** The introduction of **SWE-Bench Verified** was notable for addressing benchmark quality. By removing "impossible" or misleading tasks, Verified provides a *cleaner signal* of model progress [29] [17]. Indeed, models performed much better on Verified (OpenAI reported GPT-4 solving 33.2% vs 20% on original) [35] [22], confirming the original had underestimated true capability. This highlights how careful curation can prevent false pessimism.

F. New Benchmark Inspiration:

- **Future Work (from Papers):** Researchers noted that beyond correctness, evaluating *efficiency, incremental improvement, and collaboration* are next steps [36] [37]. For example, OpenAI's preparedness report uses SWE-Bench to gauge model autonomy risks [36], suggesting future benchmarks might track not only if an agent can fix a bug, but how *autonomously* (cost, time, interventions needed).
- **Identified Gaps:** Even with Verified, SWE-Bench focuses on bug **fixing** in Python. Gaps remain in *security-related tasks*, *long-horizon projects*, and *other languages*. Also, some tasks still have brittle tests or multiple valid solutions not fully captured by tests (could accept slightly different patches). This calls for evaluation

methods beyond binary unit tests – perhaps semantic diff analysis or human review for creative solutions.
- **Transferable Design Patterns:** SWE-Bench's design (real issues + tests) can be **transferred to other domains**. For a smart contract security benchmark, one could similarly mine GitHub for historical Ethereum/Solidity vulnerabilities and their fixes, bundling each with a reproduction test (e.g. failing assertion or exploit script that the agent must prevent). The *FAIL_TO_PASS* pattern of a test that initially fails and should pass after the fix is directly applicable to security patches. Likewise, *PASS_TO_PASS* regression tests mirror the need to ensure no new vulnerability is introduced. This structure ensures an objective, automated check – a design proven effective by SWE-Bench [25].
- **Brainstorming Directions (Smart Contract Security & Education):** Combining SWE-Bench's realism with educational goals suggests a **"Secure Contract Bench"** where agents are given a buggy smart contract and a test suite including both exploit attempts and normal-functionality tests. The agent must produce a patch that stops the exploit while preserving functionality – analogous to SWE-Bench's dual test requirement [25]. Inspired by SWE-Bench Verified, we would carefully curate tasks to avoid unsolvable challenges (e.g. ambiguous CTF puzzles), focusing instead on known vulnerability patterns (reentrancy, integer overflow, etc.) with clear fixes. This benchmark would encourage **autonomous auditing agents**: given a contract and a description of a bug or attack scenario, can an agent pinpoint and fix the vulnerability? It could incorporate the *multi-file context* (contracts often import libraries or have multiple files) and use frameworks like Hardhat/Foundry to run tests, much as SWE-Bench uses PyTest. The test-driven evaluation provides a solid ground truth for educational feedback (students or agents know immediately if their fix works). Additionally, SWE-Bench's **scaffold approach** (providing a fixed agent API for interacting with the codebase) could be borrowed to structure how agents interact with blockchain development tools (for example, an agent could be allowed to run `forge test` or query a local Ethereum node as part of its toolkit). Overall, SWE-Bench demonstrates the value of realistic, automated evals – a principle that would greatly benefit a smart-contract security benchmark.

SWE-Bench Pro

A. Source Materials:
- **Paper:** *"SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?"* (X. Deng *et al.*, arXiv 2509.16941, rev. Nov 2025) [38] [39].
- **GitHub:** `scaleapi/SWE-bench_Pro-os` – Open-source portion of SWE-Bench Pro (Scale AI, 2025) [40].
- **Scale AI Blog:** *"SWE-Bench Pro: Raising the Bar for Agentic Coding"* (Sept 2025) – outlines design and results [41].
- **Leaderboard:** Listed on LLM performance trackers (e.g. llm-stats.com) with top model scores (GPT-5 ~23%) [42].

B. Benchmark Deep Dive:
- **Goal: Reset the baseline** for coding agents by introducing a benchmark of *enterprise-grade, long-horizon software tasks*. SWE-Bench Pro's goal is to measure if AI agents can handle **complex development projects** that require substantially more work than the short GitHub issues in original SWE-Bench [41] [43]. It asks not just "can an agent fix a bug?" but **"Can an agent act as a proficient developer on large-scale tasks?"**.
- **Gaps Addressed:** It targets several gaps: (1) **Task Length & Complexity** – prior benchmarks' tasks could often be solved in minutes; Pro includes tasks taking a human *hours or days*, involving multi-step implementation and design decisions [44]. (2) **Diversity of Domains** – Pro spans *41 repositories* including enterprise software, B2B services, dev tools, and even proprietary code, whereas original SWE-Bench was limited to 12 open-source Python libs [45]. (3) **Data Contamination** – it introduces held-out and confidential

test sets to ensure models haven't memorized solutions [46] [47]. Overall, it addresses the *"graduation"* of coding agents from toy bugs to realistic software engineering tasks at scale.
- **Agent Type:** SWE-Bench Pro assumes an **autonomous coding agent** equipped to handle reading and modifying large codebases over a long session. Agents must synthesize new code, potentially across multiple files, following a high-level issue description. The tasks demand *planning, code search, multi-file edits, testing*, etc. Essentially it benchmarks an **AI software engineer** rather than a code assistant.
- **Task Format:** Each task is derived from a real development scenario (e.g. implement a new feature, refactor for performance, fix a complex bug) drawn from actively maintained projects [43]. The format still provides a repository and a problem description, but the *scope* is larger – patches in Pro are on average ~9x larger than those in original SWE-Bench [41]. Many tasks require changing *multiple components or writing new modules*. The dataset is partitioned into three subsets: **Public** (tasks from 11 open-source repos, fully released), **Held-out** (tasks from 12 repos kept private for evaluation), and **Commercial** (tasks from 18 private company codebases, results reported but tasks not released) [45]. This structure ensures a broad distribution while preventing overfitting on all tasks.

C. Dataset Construction Methodology:
- **Sources:** 1,865 tasks collected from 41 large-scale repositories [45]. Repos were chosen to represent *enterprise software challenges* – including complex web apps, data pipelines, and developer frameworks. Some portion involves proprietary code via partnerships (e.g. early-stage startup code) to inject tasks truly unseen in public corpora [46] [47].
- **Task Selection:** All tasks went through **human verification**: developers confirmed that each task is solvable and prepared with *"sufficient context to ensure resolvability"* [39]. This often meant adding or clarifying instructions so that an external agent has all needed info (which in real work might come from discussion). The tasks are explicitly designed to be **contamination-resistant** – the closed sets ensure that even if models were trained on GitHub, they won't have seen the solutions. Public tasks are also checked to avoid ones likely in model training (e.g. they avoid trivial or famous issues).
- **Languages & Difficulty:** Unlike original SWE-Bench (Python-only), Pro's tasks span multiple languages common in enterprise code (likely Python, Java, JavaScript, etc., though the paper emphasizes diversity). Task difficulty is high: these are "hours to days" projects for humans [44], meaning many will involve understanding thousands of lines of code and making nontrivial changes. The human labor invested in constructing Pro was significant (~5,800+ hours per Scale's reports [48]). This effort included writing or extracting *unit and integration tests* to validate each task's solution, similar in spirit to SWE-Bench's test approach but at greater scale.

D. Evaluation Framework:
- **Metrics:** The primary metric remains **pass rate (% of tasks solved)**, defined as the agent's output passing all tests for that task [39]. However, given task complexity, *partial credit* or tiered evaluation is also considered. The paper analyzes **failure modes** by clustering where agents went wrong [49], indicating evaluation isn't just a number but also an error characterization (e.g. logical errors vs. tool invocation errors). Additionally, because tasks are so lengthy, they measure agent consistency via multiple attempts or cost-bounded performance (though specifics are not in the excerpt, presumably metrics like "success under X dollar API budget" are tracked, as hinted by community discussions).
- **Infrastructure:** Pro uses a robust harness (likely building on the original SWE-Bench framework) that can manage *long-running evaluations*. Tasks may require spinning up containers with substantial dependencies or running integration tests that take minutes. The evaluation likely imposes **time and cost limits** on agents per task, given the potential "hours" scope – e.g. an agent might be given a few thousand GPT tokens or a certain number of steps to attempt a solution. Agents are expected to use an automated

scaffold that orchestrates reading the issue, reading relevant files, proposing code, running tests, etc., similar to how original SWE-Bench agents were structured but scaled up.
- **Verification:** Each task has an authoritative test or verification procedure. For open-source tasks, a test suite or evaluation script is provided. For proprietary tasks, Scale likely ran the agent solutions internally to compute scores, since those aren't public. By keeping tests private for held-out and commercial tasks, they ensure evaluation integrity (participants can't tailor solutions to known tests). This setup mirrors a competition-style eval: participants run agents on public tasks for development, and blind eval on hidden tasks determines true generalization.

E. Uniqueness Factor:
- **Long-Horizon Focus:** SWE-Bench Pro is the *first benchmark to require long-horizon, multi-file software development*. It moves beyond "single-issue" resolution to more **project-level tasks**, which is unique. Solving a Pro task is akin to a junior developer handling a full Jira ticket that might involve designing a solution, touching front-end and back-end, updating docs, etc. This tests an agent's *planning, decomposition*, and endurance (staying coherent over extended dialogues or tool uses).
- **Enterprise Realism:** Its inclusion of **proprietary code** and a mix of domains makes it more representative of what enterprise devs face (including potentially messy, undocumented code). It is **contamination-resistant** – a pointed response to the concern that models might "remember" training solutions. By having tasks that models definitely have not seen, Pro ensures we truly test novel problem-solving [47].
- **Evaluation of Reliability:** Pro doesn't just report a single accuracy – it also looks at *consistency and failure patterns*. For instance, the introduction of the **pass^k metric** (in line with other contemporary benchmarks) could be applied to see if an agent can solve the same task reliably across multiple independent runs [50]. The benchmark's authors indeed highlight inconsistency as an issue for SOTA models [51]. By examining and clustering failure modes, SWE-Bench Pro uniquely provides insight into *why* agents fail on big tasks (e.g. do they get stuck on understanding code, or on coordinating edits?) – a diagnostic dimension absent in simpler benchmarks.
- **Scale & Community Impact:** With 1,865 problems, it's one of the *largest coding agent benchmarks* to date. It effectively raised the bar: whereas top models were hitting ~80% on original SWE-Bench Lite by mid-2025 [52], on Pro the top scores dropped to ~23% [42], reintroducing headroom for research. This jumpstarted efforts in more advanced agent architectures and prompted the community to consider agent **productivity, not just correctness** (e.g. how to make an agent that can work for 8 hours on a task without drifting). SWE-Bench Pro is thus a watershed in emphasizing *agent autonomy at scale*.

F. New Benchmark Inspiration:
- **Future Work & Gaps:** The Pro paper suggests investigating **hierarchical agents** or *decomposition strategies* as current models struggled with long contexts [49]. Future benchmarks might include *even longer projects or cross-repository changes*. One gap to fill is *collaborative tasks*: Pro tasks are single-agent, but real enterprise work often involves multiple agents or human hand-off. A future direction could be benchmarking an agent's ability to take over a partially completed project or to cooperate with another AI agent.
- **Design Patterns to Borrow:** SWE-Bench Pro demonstrates the value of **partitioning benchmark levels** (public vs. private vs. commercial) to balance open research with realistic evaluation [46]. A smart contract security benchmark could emulate this by having a public set of known vulnerabilities, and a private set of novel or real undisclosed vulnerabilities for evaluation – ensuring models don't just memorize solutions. Also, Pro's tasks require *maintaining context over long interactions*; for an educational security agent, one could similarly require an agent to perform a sequence of actions (e.g. deploy contract, run exploit script, patch code, re-run tests) that mirrors a realistic security audit workflow.

- **Inspiration for Smart Contracts:** Smart contract exploits often involve **long-horizon scenarios** (e.g. a series of transactions to drain funds). Inspired by SWE-Bench Pro, a benchmark could present an agent with a complex DeFi project and an objective ("detect and fix any vulnerability that allows stealing funds"). This would test multi-step reasoning: identify vulnerability, write an exploit to confirm it, then patch the code – a process possibly spanning dozens of files and requiring days of analysis in real life. Pro shows it's feasible to evaluate such complex behavior by breaking down verification into *final-state comparisons* (e.g. did the agent's patch prevent the exploit, without breaking other functionality?). Additionally, Pro's emphasis on *hours-long tasks* suggests using time-extended simulations: e.g. run the agent in a persistent Ganache/Hardhat testnet environment and see if it eventually secures the contract. The key inspiration is to **not shy away from complexity** – Pro indicates the community is ready to tackle benchmarks where tasks are as hard as real jobs. For an educational context like YudaiV4, this means challenging students or AI agents with holistic projects, not just toy examples, and using a tiered evaluation (simple bugs vs. advanced exploits) to measure progress from novice to expert level.

Terminal-Bench 1.0

A. Source Materials:
- **Terminal-Bench Paper/Team:** *"Terminal-Bench: A Benchmark for AI Agents in Terminal Environments"* – The Terminal-Bench Team (Apr 2025) [53] [54]. (Cited as a misc technical report by the Terminal-Bench Team).
- **Website:** tbench.ai – official site with tasks, documentation, and leaderboards [55] [56].
- **GitHub:** `laude-institute/terminal-bench` – open-source tasks and harness (Stanford & Laude Institute collaboration) [57] [58].
- **Leaderboards:** *terminal-bench@1.0* results hosted on tbench.ai (snapshot May–Oct 2025) [59].
- **Notable Blog:** OpenBlock Labs blog on achieving #1 (59% success) with their OB-1 agent (Sep 2025) [60] [61].

B. Benchmark Deep Dive:
- **Goal:** Evaluate AI agents' ability to **master real-world command-line (CLI) tasks** end-to-end [62] [63]. Terminal-Bench tests if an agent can autonomously perform software engineering workflows in a Linux terminal – such as building software, managing servers, running scripts – simulating how a DevOps or SRE engineer might use a shell. The goal is to push beyond static code gen: can agents *execute* and *recover from errors* in an interactive environment? [64]
- **Gaps Addressed:** Terminal-Bench filled a gap for evaluating **interactive, tool-using agents**. Prior coding benchmarks didn't require the model to actually run code or use system tools. Terminal-Bench introduces tasks that require *compilation, environment configuration, network calls, file operations*, etc., thus testing an agent's integration of reasoning with action. It also addresses reliability – measuring not just if an agent can output correct code, but if it can handle a multi-step process with failure points (e.g. debug a compilation error and try again) [64] [65]. Essentially it tests *procedural intelligence and resilience* in a realistic dev environment.
- **Agent Type:** The benchmark targets **autonomous terminal agents**. These agents typically follow a loop of reading the task description, executing shell commands, reading command outputs, editing files (via an editor or command-line text edits), and so on, until completion. Unlike one-shot LLM answers, the agent here is an *interactive problem-solver* that must persist state (maintain a working directory and memory of past outputs) [65]. Many participants used specialized agent frameworks (e.g. Terminus or MiniChain) that connect an LLM to a Linux shell in a Docker container. Agents might have limited tools: common CLI programs, a text editor like `vim` or `nano`, compilers, etc. The human analogy is an engineer SSHing into

a fresh machine to accomplish a task with only terminal access.
- **Task Format:** Each Terminal-Bench task is a **natural language scenario** plus a prepared container environment. For example: *"Build the Linux kernel 6.9 from source, then prove you added a custom printk by running QEMU"* [66] , or *"Configure a Git webserver so that pushing to a repo updates a website on port 8080"* [67] . The tasks span categories like *system administration, DevOps, security, data processing, web development* [61] . Difficulty is labeled (easy/medium/hard) per task on the site [66] [68] . Crucially, tasks are **open-ended**: the agent must decide on the exact sequence of commands or code to fulfill the request. The **success criteria** are specified (often implicitly) by the task description – e.g. "if I curl the server I should see 'hello world'" or "the tests in the repo should pass." Each task has hidden verification logic to check success (more below).

C. Dataset Construction Methodology:
- **Sources:** Tasks were created by a mix of domain experts and community contributors (100+ contributors as of late 2025) [69] . Many tasks are inspired by real scenarios (some explicitly credit creators: e.g. Nicholas Carlini or Jan-Lucas Uslu for certain tasks) [70] [71] . The benchmark started with **80 tasks** (v0.1.1 referenced in a related RL paper [72]) and expanded to **100 tasks** by version 1.0 [61] . Each task includes a Docker container image or script setting up the initial state (with necessary files, dummy data, and installed tools) and a short text prompt describing the goal.
- **Task Design:** Organizers ensured tasks are **verifiable and self-contained**. For each task, they wrote a hidden *check script* or test. For example, a task requiring a file to be created will have a script that checks for that file and its contents. Tasks like "training a model to 0.62 accuracy" include an offline test dataset to evaluate the model file the agent produces [73] [74] . Many tasks piggyback on widely understood outcomes (like "if it works, all unit tests will pass" or "the output file should contain X"). During construction, tasks were validated by humans and refined – the news notes they fixed issues like dynamic external dependencies (YouTube's anti-bot changes broke a download task) by either removing or updating those tasks in later versions [75] .
- **Difficulty & Distribution:** Tasks were chosen to cover a wide range of domains and tools: from low-level (compiling kernels, cracking hashes) to high-level (data science scripts, web servers). They also vary in length; easy tasks might involve a few commands, while hard tasks might require writing and debugging multi-file code. The v1.0 selection deliberately avoided tasks requiring internet access or cloud-only resources (agents work offline in provided containers to keep evaluation deterministic). **Security** tasks were included (e.g. cracking an archive password, generating TLS certificates) [76] , which is notable for bridging into our interest of security. The tasks often have *multiple steps* (e.g. generate key, then cert, then verification file) [77] . By version 1.0, the mix of tasks was such that no single tool or simple strategy solves all – requiring generalist competence.

D. Evaluation Framework:
- **Success Metric:** Terminal-Bench uses a straightforward metric: **Success Rate (% of tasks completed)** [78] . If the agent eventually achieves the task's goal state (as determined by the hidden check/test), that task is marked as solved. On leaderboards, agents are ranked by this percentage (with confidence intervals, since some runs have variance) [79] . For example, OB-1 achieved 59.0% on v1.0 (59/100 tasks) [80] .
- **Automation:** Each task's verification is automated. The Terminal-Bench harness runs an agent inside the task's Docker container and monitors for completion. Agents typically have a time or step limit (e.g. up to N minutes or M commands). The harness periodically runs the task's check – or triggers it when the agent claims to be done. If the check passes, the task is marked resolved; if time expires or the agent gives up, it's a failure. Agents can theoretically attempt multiple tries, but the official eval likely gives one continuous attempt with potential internal retries (some agents implement their own retry on failure, as OB-1 did [65]).
- **Infrastructure:** Terminal-Bench provided a standardized **Docker-based sandbox** for each task. This

sandbox isolates the agent's actions and ensures reproducibility. The evaluation harness (Terminus) can scale to running many containers in parallel for large-scale evals. However, v1.0 users noted that running many tasks sequentially is slow (spinning up 100 containers), which led to later development of Harbor for horizontal scaling in v2.0 [81]. The **agent API** is defined: an agent must implement a certain interface (e.g. receiving the task prompt and returning commands). Baseline agent implementations (like *Terminus* default agent with ReAct prompts) were provided for reference [82].
- **Reliability Measures:** While the main metric is binary success, Terminal-Bench also tracks things like average number of commands, retries, and maybe cost if API-based models are used (the website allows plotting "resolved vs. cost" etc. [83]). This gives insight into efficiency. Additionally, tasks can be filtered or grouped by category on the leaderboard, so one can see an agent's specialty (e.g. maybe it solves all web tasks but fails on low-level OS tasks). There wasn't an explicit *multi-run consistency metric* in v1.0, but top agents often report their stable success (OB-1's blog mentions variance and how they mitigate it [84]). Terminal-Bench's evaluation is deterministic given an agent, but agents using randomness or multiple LLMs can yield different outcomes run-to-run, which the framework allows to average out by multiple trials if desired.

E. Uniqueness Factor:
- **End-to-End Autonomy:** Terminal-Bench was a first-of-its-kind for **truly end-to-end evaluation**: the agent must *read instructions, write code or commands, execute them, observe results, debug if needed, and verify the outcome* – all in one loop [64]. It doesn't abstract away execution; the model directly experiences the terminal. This revealed unique failure modes (e.g. agents getting stuck in error loops, misreading command outputs, etc.) that static benchmarks never expose.
- **Realistic DevOps Challenges:** The benchmark's tasks simulate what real engineers do when setting up systems or troubleshooting. For example, configuring a server or training a model involves many interconnected steps. Terminal-Bench demands **persistent state tracking** and **multi-step reasoning** over a long context (shell session), something few other benchmarks require. Agents have to manage memory of what happened earlier in the session (solved in practice by storing a *trace memory* as OB-1 did [65]). This tests long-term coherence and error recovery in a way akin to an actual coding session, which is unique.
- **Broad Tool Use:** It explicitly evaluates how well agents use **external tools** (compilers, package managers, etc.). Many AI agent benchmarks focus on pure "thinking" or code writing; Terminal-Bench checks if the agent can *do things* in a computing environment. It thus helped drive research into tool-formatted outputs (e.g. chain-of-thought with tool calls) and robust function calling in LLMs, because to succeed, agents needed to precisely invoke commands and parse output.
- **Community and Open-Source Approach:** Terminal-Bench quickly became an **industry standard** for agent evaluation [69]. Its open, collaborative approach (with a Discord of 1k+ members, tasks contributed by many, and frequent updates) set it apart from static academic benchmarks. It evolved rapidly (v1.0 in May 2025, many patches by contributors, then v2.0 by Nov 2025). This agility and community buy-in was unique and signaled the emergence of "living" benchmarks that keep raising difficulty as models improve [85] [86].

F. New Benchmark Inspiration:
- **Future Work from Paper:** By late 2025, Terminal-Bench 2.0 was released with harder tasks and better verification [87] [88]. The team also introduced **Harbor**, a framework to scale container-based evals and integrate training (RL/SFT) directly with the eval loop [89] [90]. This hints that future benchmarks will not be just passive evaluations but *training playgrounds*, where agents can learn from the environment. Another idea is incorporating **multi-agent collaboration** (Terminal-Bench so far is single-agent, but one could imagine tasks requiring two agents to coordinate).
- **Identified Gaps:** Terminal-Bench 1.0's limitations included some flaky tasks (external dependencies,

timing issues) – v2.0 addressed many with thorough verification [75]. However, it still focused on *single-machine scenarios*. A gap for future benchmarks is distributed systems or cloud-based tasks (e.g. deploying a microservice cluster, which could be a next frontier). Also, while Terminal-Bench had some security tasks, it did not specifically target *vulnerability exploitation* in the interactive sense (most tasks were constructive). There's room for a benchmark where an agent might play the role of an attacker or a defender in a live environment.
- **Transferable Patterns:** The concept of a **sandboxed evaluation environment** is directly transferable. For smart contract security, one could provide a pre-set blockchain dev environment (e.g. a Hardhat project with a vulnerable contract and test scripts). The agent would get a terminal with tools like `forge` or `ganache-cli` available, and perhaps a prompt: "Find and exploit the vulnerability in Contract X to steal funds (there's a test for whether funds can be stolen). Then propose a fix and verify the fix passes all tests." This is analogous to Terminal-Bench's style: present a realistic task, let the agent interact with tools to solve it. The Terminal-Bench approach of **monitoring an agent's every command and checking final state** would work here too – e.g. use Foundry tests as the success criteria.
- **Brainstorming – Interactive Smart Contract Challenges:** Inspired by Terminal-Bench, a *"Solidity Secure Terminal-Bench"* could involve tasks like: setting up a local Ethereum node, deploying a contract, performing a series of transactions to exploit it, then applying a patch and demonstrating the exploit no longer works. The agent would need to use both coding (to edit the contract) and CLI blockchain tools (to run tests or transactions). This teaches not just identifying a vulnerability but the full exploit lifecycle. Another idea from Terminal-Bench is the **persistent agent memory** – OB-1 had a trace log to avoid repeating mistakes [65]. In an educational security agent, having the agent keep a log of what attacks failed and why (and then trying different strategies) would be invaluable and could be evaluated (did the agent learn within a session?). Finally, Terminal-Bench underscores the value of *authentic tasks over quizzes*. For YudaiV4, rather than multiple-choice questions about security, embedding those concepts in a practical task (e.g. "the contract has a reentrancy bug, exploit it and fix it") will yield deeper insights into an AI or student's true capability. Terminal-Bench's success suggests that if you make the challenge realistic and provide an automated way to check success, the community will rise to it – which is exactly what we'd want for a smart contract security education benchmark.

Terminal-Bench 2.0

A. Source Materials:
- **Announcement:** *"Introducing Terminal-Bench 2.0 and Harbor"* – Official TBench News (Nov 7, 2025) [91] [88].
- **Terminal-Bench 2.0 Leaderboard:** Live at tbench.ai (select version 2.0) [59].
- **Harbor Framework:** Documentation at harborframework.com – introduced alongside TB 2.0 for scalable eval and training [92] [93].

B. Benchmark Deep Dive:
- **Goal: Increase difficulty and reliability** of the benchmark in step with advancing agent capabilities [69] [88]. Terminal-Bench 2.0 aims to stay on the *frontier of complexity* so that state-of-the-art agents are challenged. It also explicitly aimed to fix verification issues and better validate each task's correctness. In short: "harder, better verified" tasks and an improved evaluation harness [91].
- **What Changed:** TB 2.0 introduced **new tasks and refined old ones**. Some simpler 1.0 tasks may have been retired or replaced with more complex variants. For example, tasks that were solved by nearly all agents might be removed in favor of ones that require deeper reasoning or new tools. The goal is that tasks reflect scenarios frontier models still struggle with, forcing innovation (since v1.0's top agents reached ~60%

success, 2.0 presumably brought that down initially).
- **Agent Type & Use Case:** Same type – terminal-based autonomous agents – but now *with Harbor* as the default harness. Agents in 2.0 are expected to integrate with Harbor for cloud-based parallel execution and possibly agent fine-tuning. This suggests a shift: not only evaluating agents but enabling an *ecosystem where agents can be optimized on the benchmark*. The inclusion of Harbor's RL/SFT interface [89] signals an ambition for agents that learn to solve these tasks, not just static evaluation.
- **Task Format:** Remains a collection of CLI challenges, but with improvements. The news post gives an example of a problematic 1.0 task: `download-youtube` was unstable due to external site changes [75]. In 2.0, tasks like that would be reworked or removed. They mention **"substantial manual and LM-assisted verification went into each task"** [88] – meaning for every task they likely double-checked that the described goal is achievable with the provided environment and that the success criteria truly reflect the goal. We can infer tasks might be more robust (no one-day-breaking issues, clearer goals). Possibly, new tasks might involve more *multi-step pipelines* or integration of multiple domains (to be "harder"). Also, given OB-1's mixture-of-models success in v1.0 [94], 2.0 might include tasks to specifically thwart naive ensemble methods (e.g. requiring a consistent strategy rather than try-fail loops).

C. Dataset Construction Methodology:
- **Task Updates:** The TB team reviewed all v1.0 tasks, informed by community feedback, and fixed or removed any with flaws [75]. For instance, dynamic tasks were stabilized, ambiguous instructions were clarified, and trivial solutions were patched. They also likely added *brand new tasks* to cover capabilities that emerged after 1.0. Given the time (Nov 2025), new tasks might involve emerging tech (perhaps container orchestration, advanced AI tool usage, etc.) to reflect what "frontier labs" want agents to do next.
- **Verification Strengthening:** Each 2.0 task underwent **manual QA and sometimes LLM-assisted QA** [88]. This means multiple people tried the tasks (and possibly an LLM agent as well) to ensure that the success check truly correlates with accomplishing the intended goal (no false negatives or unintended shortcuts). They likely added more **assertions in the check scripts** to catch edge cases. The mention that labs commented on the "highest quality environments they have seen" [95] indicates an extreme attention to detail — tasks are self-contained, reliably reproducible, and unambiguous.
- **Size & Domains:** The total number of tasks in 2.0 isn't stated explicitly, but presumably it's in the same order (~100). However, "harder" implies possibly a few very easy tasks were removed, and some extra hard ones added, so the distribution skews toward medium-hard. The domain spread likely remains broad, but they might emphasize categories that align with difficult real-world problems (maybe more *security and networking tasks*, as those are tricky). Given security was a theme in examples (cracking, TLS setup) [76], they might have added tasks for things like *"exploit this intentionally vulnerable app"* to up the ante.

D. Evaluation Framework:
- **Harbor Integration:** The biggest change in framework is the **Harbor platform** [92] [93]. Harbor allows scaling to thousands of container runs (useful for parallel evaluation or for running multiple trials per agent for stability). It also standardizes interfaces so any agent can plug in and be evaluated under identical conditions. For TB 2.0, Harbor likely became the default way to run the benchmark (no more manual Docker orchestration). This improves consistency and speed.
- **Metrics:** Success rate remains the core metric. However, they may have introduced **tiered scoring** or separate leaderboards for certain categories (the TB site frontpage in 2.0 era shows combined chart views, etc. – possibly enabling per-category insight) [96]. Also, since Harbor supports RL training, one could measure not just static performance but improvement over time – though that's outside the scope of the official "score", it's facilitated.
- **Reliability & Regression:** With 2.0's focus on verification, the evaluation is more reliable – fewer false

passes, and tasks remain solvable over time. The mention of better quality suggests the eval harness now includes checks for things like "the agent didn't just hard-code the expected output" (for instance, adding canary values that ensure the agent actually performed the task, not simply echoed a string). In terms of **tiers**, Terminal-Bench might consider a *"100% correctness plus style"* kind of metric in future – but currently, a pass is a pass. That said, the introduction of *Harbor's RL loop* hints at potentially new metrics like *time to solve* or *number of actions*, since improving those would be part of optimizing an agent. They might surface such metrics for analysis if not for ranking.

E. Uniqueness Factor:

- **Harder Tasks = New Capabilities Tested:** Terminal-Bench 2.0's uniqueness lies in **staying ahead of the curve**. It's essentially a *benchmark that evolves*. Unlike static benchmarks that become easier as models improve, 2.0 proactively increased difficulty. This sets a precedent for *versioned benchmarks* in AI: the test gets updated as soon as it's "mastered," analogous to video game levels. This approach is unique in that it acknowledges the rapid progress in AI agents and continuously raises the bar.
- **Verification Rigor:** The use of *LM-assisted verification* is an innovative twist [88]. It implies they used GPT-4 or similar to help test each task for potential loopholes or mis-specifications. Using AI to adversarially test AI benchmarks is a new level of rigor. This likely makes 2.0 tasks more *robust against prompt exploits or unintended solutions*. For example, if an agent tried to simply echo the target phrase instead of actually performing steps, the verification might catch that because they preempted it. This yields a benchmark that truly measures the intended skill (e.g. configuring a server *properly*, not just printing "server configured").
- **Coupling with Improvement Mechanisms:** Terminal-Bench 2.0 did not just come as a test, but with Harbor to help agents get better [89]. This combination is unique: it blurs the line between benchmark and training environment. It acknowledges that complex tasks might require agents to learn from experience. By providing infrastructure to both evaluate and train, Terminal-Bench 2.0 stands out as a *benchmark that fosters progress*, not just measures it. This concept – where a benchmark is accompanied by a platform for iterative improvement – could be highly relevant to educational benchmarks (encouraging a curriculum or practice loop).

F. New Benchmark Inspiration:

- **Future Work & Dynamic Benchmarks:** Terminal-Bench's versioning suggests that a *Smart Contract Security benchmark should plan for evolution*. As solvers (humans or AIs) get better, introduce new vulnerabilities or more complex contracts. Terminal-Bench demonstrates how to manage this via version updates and engaging a community to contribute harder challenges. For YudaiV4, one could have an initial set of Solidity vulnerabilities and plan to expand it (e.g. add new categories like flash loan attacks, cross-chain exploits) as prior ones become well-solved.
- **Quality Assurance:** The meticulous verification in TB 2.0 shows the importance of eliminating false signals. For a security education benchmark, this means thorough testing of each challenge: not only ensuring the exploit and fix are well-defined, but also that the agent can't "cheat" (for instance, an agent shouldn't succeed by calling an API that just flags the vulnerability without understanding it). We can adopt **AI-assisted validation**: use an LLM to simulate potential unintended solutions or to ensure that any solution truly patches the issue. This guarantees the benchmark remains a reliable gauge of skill.
- **Integration of Training:** The Harbor concept could translate to a *"gym" for secure code agents*. Imagine a setup where an agent can iteratively practice on exploit scenarios, with the benchmark framework measuring its improvement. For education, this is fantastic – students (or their AI assistants) could get immediate feedback and try multiple approaches. The benchmark then isn't one-and-done; it's a continuous learning platform. From TB 2.0, we learn that making the evaluation harness **flexible and scalable** (cloud containers, multi-trial runs) can enable such use.

- **Benchmark as Competition and Coach:** Terminal-Bench's community vibe (Discord, leaderboards, contributions) is something to emulate. A smart contract security benchmark could host public leaderboards (who can build an agent that finds and fixes the most bugs?) and also provide guided hints or analysis for participants (like how TB's analysis tools let you see where agents fail). This dual role, as both a contest and a teaching tool, is inspired by Terminal-Bench's journey from 1.0 to 2.0 – where they not only posed harder problems but gave agents better tools (Harbor) to solve them. For YudaiV4, that might mean providing participants with both problems and automated feedback or even AI suggestions if they get stuck (somewhat akin to an RL environment where the goal is to maximize security score). In summary, Terminal-Bench 2.0 teaches us that the next-gen benchmarks will be **dynamic, community-driven, and tied into the training loop** – principles highly relevant for creating a living benchmark in the security domain.

CriticalPoint (CritPt)

A. Source Materials:
- **Paper:** *"Probing the Critical Point (CritPt) of AI Reasoning: A Frontier Physics Research Benchmark"* (D. Guu *et al.*, arXiv 2509.26574, Sep 2025) [97] [98].
- **Inspirehep Entry:** Summarizes CritPt's physics focus [99].
- **Turing Newsletter (Oct 2025):** Brief introduction noting CritPt as first benchmark for frontier physics reasoning [100].

B. Benchmark Deep Dive:
- **Goal:** Evaluate LLMs on **advanced physics problem-solving**, mirroring the reasoning needed in frontier *research-level* physics. CritPt's goal is to find the "critical point" where AI transitions from pattern-matching textbook problems to genuinely *integrated reasoning* on open-ended scientific challenges [101] [102]. Essentially, can AI act like a junior researcher tackling new physics problems?
- **Gaps Addressed:** It addresses a gap in benchmarks: existing academic tests (AP exams, Olympiad problems) are limited to well-structured, known-solution problems, and models were already nearing human-level on those. CritPt introduces **open-format, multi-step problems** beyond the training distribution [103] [104]. It focuses on tasks that are *realistic entry-level research questions*, not trivia or competition puzzles. This tests *deep reasoning, synthesis of concepts, and handling of uncertainty* – areas where models still falter. Also, CritPt emphasizes *trustworthiness of reasoning* in a domain where mistakes can be subtle (physics), which general benchmarks ignore.
- **Agent Type:** The benchmark is for *LLM agents with reasoning trace output*. Typically, an agent would produce a step-by-step solution (derivations, calculations) and a final answer (which could be a numeric array, symbolic expression, or code function). It's not interactive (the user isn't guiding it), but the problems are so complex that any solution from the model will involve multi-turn internal reasoning. The agent is allowed to use tools like Python (some tasks explicitly allow writing a small code to compute an answer) [105]. This is more of a *problem-solving assistant* agent rather than a tool-using agent in environment; the tool use is limited to computational assistance within the reasoning process.
- **Task Format:** CritPt provides **71 "challenges"**, each akin to a mini research problem with a physics narrative [106]. For example, a challenge might describe a novel scenario in quantum computing or astrophysics and ask for analysis or derivation (e.g. "Derive the energy levels for a particle in X potential under Y conditions"). Each challenge is *composite*, meaning it can naturally be broken into sub-tasks. Indeed, CritPt also provides **190 "checkpoints"**, which are intermediate sub-problems derived from the full challenges [106]. These checkpoints correspond to steps a human might break the problem into (like part (a), (b), etc.), representing knowledge recall or simpler reasoning steps on the way to solving the big problem.

The format is free-form: questions are open-ended (not multiple choice) and answers may be numeric with tolerance or symbolic.

C. Dataset Construction Methodology:

- **Authorship:** The problems were **hand-crafted by physics PhDs and researchers** across virtually all major physics subfields [107]. Over 7 months, AI researchers and physicists collaborated to select topics and design questions that are solvable but non-trivial [107]. Each expert contributed problems inspired by their own research experience, ensuring authenticity and uniqueness (i.e. not something from a textbook or well-known paper) [108].
- **Criteria:** They enforced strict criteria to avoid data leaks and trivialization [109] [110]. All problems are *unpublished* (novel scenarios), and they were constructed to be **"search-proof"** – meaning you can't just copy-paste the question into Google or an LLM and get the answer [111]. Also, final answers are formatted to be guess-resistant: for instance, a correct answer might be a matrix of specific numbers or a complex expression, which is hard to guess randomly [111]. This ensures that solving requires doing the reasoning, not elimination or guessing.
- **Validation:** Each problem underwent **multiple review stages** [108]. In addition to domain experts validating, they had an internal testing with GPT-4 to see how it performs on early drafts (helping calibrate difficulty). Only after refining problems and ensuring they weren't too easy or ambiguous did they finalize them. They also prepared **complete solutions and grading scripts** for each (though only one example solution is publicly released; others are kept private to avoid contaminating future model training) [112]. The checkpoints (190 of them) were extracted and curated such that they each have a definite answer and can be used to pinpoint where a model's reasoning fails.
- **Size & Domains:** The 71 main challenges cover *nearly all physics domains*: e.g. quantum mechanics, relativity, astrophysics, particle physics, condensed matter, etc. [107] They are described as comparable to *warm-up research exercises for first-year grad students* [113] – meaning they're tough but doable with solid training. The modular checkpoints include things like deriving a formula, recalling a law, or doing a sub-calculation. This dual dataset (challenges + checkpoints) is a clever methodology to analyze performance at different granularities.

D. Evaluation Framework:

- **Metrics:** CritPt evaluates both **final answer accuracy** and possibly the quality of reasoning (though mainly the former in a quantifiable way). For each final answer (which could be numeric, symbolic, or even a function), they developed an **auto-grading pipeline** that compares the model's answer to the ground truth within tolerances [114] [115]. Metrics reported include the percentage of challenges solved and checkpoints solved. The headline result was that GPT-5 (high) got ~4.0% of full challenges correct [116], and other models near 0%, while on checkpoints some did slightly better (indicating they can handle pieces but not entire problems). They also likely track *partial credit* implicitly by the checkpoint performance – e.g. a model might solve 2 of 3 subparts of a challenge, which is informative even if final answer is wrong.
- **Autograder Details:** The grading solution is quite sophisticated: model outputs are normalized into a structured format (they have the model output in Markdown with math, which they then parse) [105]. They use tools like SymPy for symbolic comparison, and numerical checks with tolerances for numeric answers [115]. For functions or code outputs, they can execute them on test cases [105]. They also incorporate physics-specific tolerance (like if an answer is a physical quantity, some relative error margin is allowed). This is all done without human in the loop, which is impressive given the complexity – it solves the "free-form answer grading" problem by making answers machine-comparable through normalization.
- **Reliability Measures:** A big concern addressed is *ensuring the model's reasoning is trustworthy*. While they don't directly score "reasoning soundness" with a number, they did analyze whether the chain-of-thought

contained errors. Possibly the paper discusses using LLM judges or consistency checks, but they note LLM judges can be unreliable for such advanced content [114]. So they stick to objective final answer grading. However, the presence of checkpoints allows a finer evaluation: one can see at which checkpoint the model fails. For example, maybe the model got the conceptual part right but flubbed the algebra in the final step. This is useful diagnostic information, akin to unit tests for each reasoning step. It's a novel evaluation framework where *multi-step reasoning is instrumented with intermediate score points*.

E. Uniqueness Factor:

- **Frontier Difficulty:** CritPt is unique in pitching tasks at the **research frontier** (albeit entry-level). It's arguably the most difficult benchmark for LLMs in 2025 in terms of reasoning depth – current models score essentially near zero on full problems [116]. This is intentional: it identifies the "critical point" beyond which models shift from parroting training data to true reasoning. No other benchmark at the time had this combination of difficulty and breadth in a scientific domain.
- **Domain-Specific Reasoning:** Unlike general benchmarks, CritPt is tailored to *physics*, requiring not just generic logical reasoning but domain knowledge and mathematical rigor. It tests things like dimensional analysis, knowledge of physical laws, and the ability to derive conclusions that are not in any textbook. This specialization is unique and necessary – it's a benchmark of **expert-level reasoning** rather than general knowledge or programming skill. It highlights a dimension of AI performance (scientific reasoning) that others didn't.
- **Private Solutions & Leakage Resistance:** CritPt's methodology of keeping solutions private (except one example) [112] and making problems unpublished is a strong stance on avoiding future contamination. They essentially created a test that will remain valid even as models train on new data, by preventing that data from containing the answers. This is unique; many benchmarks eventually get leaked into training sets, but CritPt is pre-emptively guarding against it. It positions CritPt more like an *ongoing competition or exam* (with an answer key kept secret) – that's unusual in an era of open-source benchmarks but might set a precedent for high-stakes evals.
- **Multi-format Answers:** The need to handle complex answer types (vectors, formulas, code) and the custom autograder is also a unique contribution. It shows how to evaluate very *rich outputs* objectively [115]. This is valuable for any future benchmark where answers aren't simple classifications – for instance, a smart contract security benchmark could produce exploit scripts or patched code; a similar approach to normalization and automated checking could be used, taking inspiration from CritPt's grading pipeline.

F. New Benchmark Inspiration:

- **Future Work:** The authors see CritPt as a framework expandable to other domains or deeper problems. Future work might involve *multi-modal reasoning* (e.g. incorporate experimental data, graphs) or *interactive problem solving* (an AI that can ask for hints or clarification like a grad student with an advisor). They also highlight the need for *improving reasoning reliability* – e.g. developing models that can provide self-consistent derivations, not just final answers, since trusting an AI in science will require scrutinizing its reasoning [117] [118]. So a direction is training models explicitly on showing and verifying each step, which a benchmark like CritPt encourages by design (if a model can solve checkpoints one by one, it's basically showing its work).
- **Identified Gaps:** One gap is that CritPt is static and single-shot: the model gets one chance at a complex problem. Human physicists, however, might break a problem down, realize a mistake, backtrack, etc. There's an opening for an *interactive version* where a model can iteratively refine its solution or maybe request minor clarifications. Also, CritPt currently covers physics only; applying this approach to other domains (like biology or economics for research questions) could be valuable, and indeed the framework is adaptable.
- **Design Patterns to Emulate:** For a smart contract security benchmark, the *checkpoint idea* is gold. Just as

CritPt has multi-step physics problems, a complex security audit can be broken into sub-tasks: e.g. (a) identify potential vulnerability locations, (b) hypothesize the exploit, (c) write a proof-of-concept script, (d) suggest a fix. We could implement "checkpoints" in a security benchmark and evaluate an agent at each stage – this provides granular feedback and partial credit. CritPt's approach ensures we don't just know pass/fail, but *where the agent's reasoning failed*. This is excellent for educational purposes, as students/agents can see if they failed at the identification stage vs the exploitation stage, for example.
- **Brainstorming Directions:** CritPt shows that even advanced LLMs struggle with *complex, unseen logic puzzles*. In smart contract security, we expect current models to also struggle with intricate exploits that require chaining concepts. We can take inspiration to create scenarios that are *beyond just known CTFs*, possibly novel ones designed by experts, to find that "critical point" for security reasoning. Also, CritPt's focus on *trustworthy reasoning* suggests in our domain we might require the AI not just to output a fix, but to explain why it fixes the vulnerability – and evaluate the explanation quality. Perhaps use LLMs to judge if the agent's justification covers the root cause (with caution, as CritPt did note LLM judges can be iffy for complex content [119], but maybe combined with static checks). Finally, the **leakage-resistant** approach is key if our benchmark is to be long-lived. We might keep a set of "secret" exploit challenges that are not released publicly so that AI developers can't train on them, using them only for evaluation (similar to how CritPt kept most solutions private) [112]. This ensures the benchmark continues to measure reasoning, not memory – aligning with CritPt's philosophy. In summary, CritPt inspires us to craft a benchmark that truly probes the edge of AI capabilities in our domain, using rigorous design, hidden answers, multi-step grading, and expecting very low initial scores – because that's how we identify what future progress is needed.

τ-Bench (Tau-Bench)

A. Source Materials:
- **Paper:** *"τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains"* (S. Yao *et al.*, arXiv 2406.12045, Jun 2024) [120] [121].
- **Company Blog (Sierra AI):** "τ-Bench: Benchmarking AI agents for the real-world" by K. Narasimhan (June 20, 2024) [122] [123] – high-level overview of motivations and design.
- **GitHub:** `sierra-research/tau-bench` – code and data for τ-Bench (likely includes tasks, simulators) [124].
- **HAL Leaderboard:** Princeton HAL system lists τ-Bench domain-specific evaluations (e.g. Airline) [125].

B. Benchmark Deep Dive:
- **Goal:** Test AI agents in **dynamic, multi-turn conversations where they must use tools and follow strict policies** to help a user achieve a goal [121] [123]. In simpler terms, τ-Bench evaluates an agent's ability to serve as a helpful assistant in realistic scenarios like customer service, where the agent interacts with a user (who may ask clarifying questions or provide info gradually) and calls external APIs (like database queries or booking systems) under certain rules (business or safety policies). It's about *situated conversational AI*: can the agent get things done cooperatively over a dialogue, not just answer one-shot?
- **Gaps Addressed:** It addresses two major gaps in prior agent benchmarks: (1) **Multi-turn human-agent interaction** – most benchmarks gave the model all info at once; τ-Bench simulates a back-and-forth where the agent must query the user for missing info, remember it, and adjust [126] [123]. (2) **Rule following and tool use** – the agent has to navigate domain-specific guidelines (e.g. privacy rules) and utilize domain APIs correctly to accomplish tasks [121]. This combination wasn't present in earlier benchmarks, which either tested tools in isolation or dialogue in isolation. τ-Bench merges them, reflecting real deployed systems (like

a customer support bot that must both chat naturally and fetch data from a database).
- **Agent Type:** The agent is a **domain-specific assistant** with access to one or more tools (APIs). It receives as input a conversation (with user messages arriving one by one) and outputs either a message to the user or a tool API call. The agent also is given a **policy document** at runtime – a set of textual rules it must follow (e.g. "Don't reveal customer's password; If user asks for a refund over $100, escalate to human", etc.). So the agent here is a *dialogue agent with an integrated tool API skill and a constraints module*. LLMs like GPT-4 with function calling are ideal for this. The user in the benchmark is actually a **simulated user** controlled by an LLM following a script or scenario guidelines [123] [127]. So it's effectively agent vs. (model-based) user in a controlled scenario.
- **Task Format:** τ-Bench is organized into **domains** (at least two in initial release: *Retail* and *Airline*) [128]. Each domain has: a **database** (e.g. customer orders or flight bookings), a set of **API endpoints** (like `LookupOrder`, `ChangeFlight`), and a **policy document** for that domain (company policies). Then they have a set of **scenarios/tasks**, each scenario is a specific user goal (e.g. "User wants to change their flight to a different date and airport" in the Airline domain [127], or "User received a damaged item and wants a refund" in Retail). For each scenario, the simulated user is given an initial profile and a script of possible utterances. The agent doesn't see the underlying structure, it just experiences a conversation starting with a user message and must carry it through to resolution. A task ends when the user's goal is either met or the agent cannot continue. The **goal state** is defined in terms of database changes – e.g. "the flight was successfully rebooked" or "a refund was issued in the order system" [129] [130].

C. Dataset Construction Methodology:
- **Scenario Creation:** Domain experts or the research team manually designed the domain schemas (DB tables), APIs, and a set of realistic tasks by drawing from actual customer service experiences [131]. For each domain, they wrote out guidelines (policies) that an agent should abide by. Then they wrote scenario descriptions (what the user wants, what info they have, etc.). Using GPT-4, they likely generated varied user utterances to simulate how a real customer might behave (the blog mentions using LLMs to create diverse user interactions instead of rigid scripts [132]). They also ensured some scenarios test policy adherence – e.g. a user might ask for something that violates a policy, to see if the agent refuses correctly.
- **Annotations & Goals:** Each scenario comes with an **annotated "goal state"** – basically the final database state that indicates success [129]. They probably use a JSON or record diff to represent that (e.g. `booking.status = "changed"` with new flight details). This is used for evaluation. They also define the *dialogue flow* loosely: user might not give all info initially, so agent should ask. The tasks are designed to require multiple turns (the agent must gather info like booking reference, preferred date, etc. step by step).
- **Domains and Scale:** Initially 2 domains (Retail, Airline) with more envisioned (the Tau²-Bench mention hints at adding Telecom, etc., later [133]). Each domain might have, say, 50 tasks scenario variants, possibly yielding a few hundred dialogues. The tasks were intentionally *diverse and creative* within domain constraints to ensure models can't just learn a single conversation pattern [132]. The scale is not massive in terms of number of tasks (it's about quality of scenarios, not big data), but each task is a fairly rich simulation.
- **Policy and Complexity:** The policy documents in each domain were crafted to be **non-trivial** – meaning the agent has to remember them during conversation. E.g. an airline policy might say "Agent must confirm identity before changing a booking" – if the agent forgets to do that step, that's a failure. These policies were drawn from real guidelines or invented to be realistic. The complexity arises as the agent must juggle satisfying the user vs. obeying rules, which sometimes conflict (user might press for something against policy). This methodology ensures tasks test not just technical API calls but judgment calls in line with rules.

D. Evaluation Framework:

- **Success Criteria:** A scenario is marked successful if at conversation end, the **database state matches the annotated goal state** [129] [130] *and* the agent did not violate any policies during the conversation. They compare the "final DB state" after agent's tool calls to the expected state for that task (for instance, did the agent actually rebook the flight to the requested destination?) [129]. If yes, and the conversation logs show no disallowed action, then it's a success. This **state-based evaluation** is a key feature – it objectively measures outcome rather than subjective dialogue quality [130]. It also allows flexibility in the exact wording of the conversation, focusing on whether the correct outcome was achieved.
- **Pass^k Metric:** Recognizing that one run might not capture reliability, they define **pass^k**: the agent is run k times on slightly varied user behavior for the same task, and pass^k indicates the fraction of runs (or probability) in which it succeeds [134]. For example, pass^8 was reported to be <25% for GPT-4 in retail [51], meaning even if it solved it once, it often failed others – indicating inconsistency. This metric quantifies *agent reliability across multiple trials*, which is important for real-world deployment (an agent needs to be robust, not just lucky once).
- **Policy Violation Checking:** They likely have an automated way to check policy adherence – possibly regex or simple heuristics on the agent's messages (e.g. if policy says "don't reveal password" and the agent's message contains something that looks like a password, that's a fail). They could also use an LLM judge to flag violations or a curated list of "forbidden phrases/actions". The evaluation thus has a binary component (violated policy or not) in addition to the success criterion. If either fails, the conversation is a fail.
- **Tool Use Verification:** The agent's API calls are monitored. The evaluation ensures that *all required tool calls were made correctly*. If, say, the agent outputs a booking confirmation to the user but never actually called the `ChangeFlight` API, that's a failure even if the conversation sounds good – because the DB state wouldn't change. This closes the loophole of an agent faking success in conversation. τ-Bench was noted for not just validating tool syntax but the **effect** of tool use [135]. That is, the benchmark verifies the agent *truly completed the task* via the tools, not just talked about it [135].

E. Uniqueness Factor:

- **Interactive Triad:** τ-Bench pioneered evaluating the *triad of User-Agent-Tools*. It's not just an agent using tools (like WebArena did) and not just an agent talking to a user (like Dialogue benchmarks did), but doing both simultaneously [126] [123]. This makes it uniquely reflective of real-world applications (think customer support bots, personal assistants). The simulated user aspect is novel – using LLMs to generate realistic human behavior in the loop. This allows stress-testing agent adaptability to unpredictable user inputs, which static benchmarks can't do.
- **Rule-Following in Dialog:** The introduction of explicit policy documents and requiring strict compliance is unique. Many benchmarks measure knowledge or goal completion, but τ-Bench also measures *obedience to constraints*. This is crucial for deployments (an AI that solves a user's request but breaks company policy is not a success in practice). By evaluating this, τ-Bench addresses the reliability and safety aspect in a quantifiable way.
- **Objective Goal-State Eval:** In dialogues, evaluation is often fuzzy (BLEU score, human rating). τ-Bench's use of final state comparison as a proxy for success [130] is unique and powerful. It turns a potentially subjective problem (was the customer satisfied?) into an objective one (was the task done in the backend?). This is novel in dialogue benchmarks. It also allows room for creativity in how the conversation is conducted, as long as the end result is correct – encouraging agents that are effective, not just good at mimicking a particular wording.
- **Reliability Emphasis:** The pass^k metric and inconsistency findings [51] underscore something unique: measuring not just *can* an agent do it, but *how often* it can do it reliably. This is a shift from one-shot

performance to a distribution of performance, acknowledging that LLM-based agents can be stochastic or brittle. It pushes for agents that are consistently good, a necessary quality for real-world adoption.

F. New Benchmark Inspiration:

- **Future Work:** τ-Bench authors likely plan to expand domains (covering more industries) and integrate more complex user behaviors (like error correction if the agent makes a mistake mid-dialog). Another avenue is multi-agent interaction – e.g. the agent could escalate to a human or another AI, which could be simulated in future benchmarks. Also, bridging to *multimodal* (imagine a support agent that can also send images or interpret screenshots) could be a next step.
- **Identified Gaps:** One gap is that the user simulator, being an LLM, might not capture all real human quirks. Agents might exploit the predictable patterns of the LLM user. Future benchmarks could involve human-in-the-loop evaluation or more chaotic user behavior. Another gap is evaluation of conversation quality beyond task success – e.g. did the agent maintain politeness, empathy? τ-Bench didn't explicitly score that (as long as policies aren't violated). In customer-facing scenarios, these matter. So, a holistic benchmark might fuse task success with user satisfaction metrics.
- **Design Patterns for Smart Contracts:** The idea of a **Tool-Agent-User loop** can be applied to smart contract security training: consider an interactive scenario where a *User* (or adversary) is interacting with a contract, and the *Agent* is a security monitor or on-the-fly auditor. For example, an agent could be tasked to guide a user through deploying a secure contract. The user might ask to implement a feature that's potentially insecure, and the agent must use tools (linters, scanners) and follow policies (security best practices) to advise or make changes. This would test the agent's ability to handle dynamic requests and enforce security rules (analogous to policies). Or conversely, an *attack simulation*: the agent is a smart contract exploit agent, the user is the contract (responding with transaction outcomes), and the agent must figure out through a dialogue of transactions how to drain funds, while following a "policy" of stealth (not triggering alarms).
- **Brainstorming Directions:** Tau-Bench's success suggests making benchmarks **scenario-based and interactive** for educational purposes. For YudaiV4, instead of static questions about vulnerabilities, we could create a *simulated blockchain environment with a chatbot interface*. A student (or AI agent) has a conversation with a "DeFi user" who reports issues (funds missing, unusual behavior) – the agent must investigate by querying the blockchain (tool calls to etherscan API, etc.), follow security guidelines (like not leaking private info or keys), and finally resolve the issue (maybe by deploying a patch or advising the user). The final state (issue resolved, exploit identified) can be checked automatically. This would train an agent in a realistic workflow akin to being an on-call security engineer. We can measure success by whether the exploit is found and patched (goal state) and whether the agent avoided bad practices (policy: e.g. never expose a private key during debugging). The **pass^k** concept encourages measuring consistency – for instance, run the same exploit scenario with slight variations (different attacker addresses, etc.) and see if the agent catches it every time. The *policy enforcement* angle from τ-Bench is directly relevant: we can encode secure coding guidelines or audit checklists as policies and require the agent to abide (e.g. "always run tests after changes", "never ignore a failing test"). This ensures the educational benchmark not only checks if a vulnerability is fixed, but also if it was fixed using proper procedure. In summary, τ-Bench inspires a *rich, interactive benchmarking style* that could make a smart contract security benchmark far more engaging and realistic, teaching agents (or students) not just to solve problems, but to do so through proper dialogue and process.

SecBench

A. Source Materials:
- **Paper:** *"SecBench: A Comprehensive Multi-Dimensional Benchmarking Dataset for LLMs in Cybersecurity"* (P. Jing *et al.*, arXiv 2412.20787, Jan 2025) [136] [137].
- **GitHub:** `secbench-git/SecBench` – contains dataset and paper, plus a Chinese README [138] [139].
- **Project Website:** secbench.org – likely provides data access and leaderboard info [139].
- **Medium Article:** *"Benchmarking LLMs in Security: Five Datasets Reviewed"* (Dec 2024) – situates SecBench among other security evals (CTI-Bench, SecEval, etc.) [140].

B. Benchmark Deep Dive:
- **Goal:** Provide a **large-scale, expert-focused evaluation of LLMs on cybersecurity knowledge and reasoning** [136]. SecBench aims to assess how well models understand cybersecurity concepts, vulnerabilities, best practices, etc., across a wide range of subdomains and question types. Unlike general QA benchmarks, it zeroes in on security as a specialized area where domain expertise is crucial.
- **Gaps Addressed:** It addresses the lack of dedicated, comprehensive benchmarks in the security domain. Prior efforts were either too small, focused on multiple-choice only, or narrow in scope. SecBench is *multi-dimensional*: it covers multiple question formats (MCQ vs short answer), multiple languages (bilingual), and multiple skill levels (knowledge vs reasoning) [141] [138]. By doing so, it fills gaps such as the need for open-ended security questions (not just MCQ) and non-English security evals. It also specifically challenges models on **logical reasoning in security scenarios**, not just factual recall [141]. This is important because cybersecurity often requires applying knowledge in complex ways (e.g. analyzing an exploit scenario), which general benchmarks don't target.
- **Agent Type:** The benchmark is primarily for evaluating *LLMs as question-answering or explanation agents* in a static setting. The agent would be prompted with a question (either multiple-choice format or free-response question) and must produce an answer (choice label or a short text). So, it's not an interactive agent benchmark but a QA evaluation. However, in the short-answer part, an agent might be expected to generate a few sentences explaining something or giving a specific answer (e.g. "How would you mitigate X vulnerability?" expects a brief explanatory answer). Thus, it tests both *discriminative ability* (MCQ picking) and *generative ability* (formulating an answer).
- **Task Format:** SecBench includes **44,823 multiple-choice questions (MCQs)** and **3,087 short-answer questions (SAQs)** [142]. The MCQs typically have one correct answer (or possibly multiple in some cases, though not sure if multi-select). They are probably styled like "What does this attack do?" with 4 options, etc. The SAQs are open-ended: e.g. "In the context of web security, explain what a CSRF attack is and one method to prevent it." The questions are further classified by **capability level**: Knowledge Retention (straight factual questions or definitions) vs Logical Reasoning (applied, scenario-based questions) [143] [144]. They also span **9 security sub-domains** (like Network Security, Cryptography, Malware, Web Security, etc.) [145]. Many questions likely come from real exam banks, CTF writeups, or were written by experts (they even held a question design contest to source some) [146]. The dataset is two-language: each question is in English or Chinese, or possibly parallel in both for some subset [142] (at least, they mention two mainstream languages, which are those). This tests models' security knowledge across languages.

C. Dataset Construction Methodology:
- **Sources:** They built SecBench by collecting high-quality questions from open sources (textbooks, online quizzes, security forums) and by organizing a **Cybersecurity Question Design Contest** [147]. The contest presumably had security professionals submit challenging questions, ensuring uniqueness and quality. This yields a large pool of expert-validated questions, much larger than previous security QA sets.

- **Multi-Dimensional Design:** The team explicitly structured the dataset to have four "multi's": *Multi-Format* (MCQ & SAQ) [144], *Multi-Level* (knowledge vs reasoning) [143], *Multi-Language* (English & Chinese) [148], *Multi-Domain* (9 subdomains) [145]. This required labeling each question along those axes. For example, every question got tagged as either knowledge retention (KR) or logical reasoning (LR) based on whether it tests recall of known info or requires inference/application [143]. Also, they ensured domain coverage: D1 through D9 categories as listed, covering broad cybersecurity. The dataset overview figure in the README shows the breakdown and confirms these design points.
- **Size and Quality Control:** With ~48k questions total, it's arguably the largest specialized security QA dataset to date [137]. They used LLMs in the pipeline too – specifically, they mention using LLMs to label data (presumably classify questions by topic or difficulty) and to construct a grading agent for short answers [137]. For quality, each question from the contest likely got reviewed by multiple people to ensure correctness of the answer and clarity. Automatic filtering may have removed duplicates or trivial ones. The Chinese portion ensures non-English scenarios (maybe questions about Great Firewall or using Chinese terminology for tech) are represented, making it comprehensive for a global model.

D. Evaluation Framework:
- **MCQ Evaluation:** Straightforward – model's chosen option is compared to the correct answer. They measure accuracy (% correct) across all MCQs, and also broken down by domain, language, and level. This yields metrics like "Knowledge MCQ accuracy vs Reasoning MCQ accuracy" to see how models fare on simple vs complex questions [143]. They likely also computed something like *area under accuracy vs model size* or similar across 16 tested LLMs to show a landscape [149].
- **SAQ Evaluation:** For short answers, they can't just do exact match. SecBench addresses this by using **an LLM-based grading agent** [137]. Essentially, they fine-tuned or prompted an LLM to act as an examiner: given a model's answer and a reference answer, output a score or pass/fail. They mention doing (1) labeling data with LLMs and (2) constructing a grading agent [137]. This suggests they might have used GPT-4 to create reference answers and perhaps grade the SAQs. The grading likely checks if key points are covered. Another possibility is that for SAQs they provided some canonical answers and did semantic similarity or used a rubric. In any case, automatic SAQ grading is notoriously hard, but they attempted it, acknowledging it's not perfect but necessary at this scale. They might have validated the grader on a subset with human judgements.
- **Metrics:** They then report performance of 16 SOTA LLMs (both base and instruction-tuned) on these sets [149]. Likely metrics are overall MCQ accuracy and SAQ score. They possibly combine MCQ and SAQ into an aggregate or treat them separately. The results show that even top models probably don't excel at security-specific reasoning, which is a key insight (the abstract suggests it's the largest and they demonstrate usability by benchmarking models [149]). Also, by dividing into KR vs LR, they can show, e.g., GPT-4 might do okay on knowledge (maybe high accuracy) but much lower on reasoning questions, highlighting a gap.

E. Uniqueness Factor:
- **Security Domain Focus:** SecBench is unique in its **breadth within one domain**. It's arguably the *most comprehensive cybersecurity QA benchmark*, covering everything from compliance standards to exploit techniques, whereas earlier benchmarks were piecemeal (some did only vuln classification, some only policy QA). This specialization means models are tested on material that a security professional is expected to know – a new level of expertise testing for LLMs.
- **Multi-lingual & Multi-format:** The inclusion of Chinese questions is unique among security benchmarks, reflecting the global nature of security (lots of research and practice happens in Chinese too). It ensures models need multilingual knowledge (e.g. Chinese cybersecurity law, terminology). The mix of MCQ and free response is also unique; many benchmarks stick to one format. By doing both, SecBench checks for

both recognition (MCQ) and generation (SAQ) capabilities, which is rare in a single benchmark.
- **Scale and Contest Sourcing:** The sheer scale (nearly 50k questions) curated specifically for security is unmatched. The fact that they held a contest to source questions is a unique approach, engaging the community to contribute and likely yielding creative, authentic problems. This is reminiscent of how some programming contests gather tasks, but new for a static benchmark. It likely improved quality and variety.
- **Automatic Grading Agent:** Using LLMs to grade LLMs in short-answer security questions is a cutting-edge aspect. They basically built an AI TA (teaching assistant) to mark answers [137]. This meta-AI approach is unique and necessary at this scale. It foreshadows a future where evaluation itself can be partially automated even for open-ended answers – a critical innovation for large educational benchmarks.

F. New Benchmark Inspiration:
- **Future Work:** SecBench's authors might consider expanding into more interactive formats (like a dialogue-based security troubleshooting benchmark) or adding code-based questions (e.g. give code and ask if it's vulnerable). Also, since they mention "arguably largest", future work may aim to integrate with practical tasks (maybe a SecBench++ that includes coding challenges or log analysis tasks beyond QA). They also might refine the grading agent as LLMs improve, to approach human-level grading reliability on open answers.
- **Gaps:** One gap is that answering questions correctly doesn't guarantee an LLM can *perform* security tasks (like actually exploit or fix code). SecBench is heavy on knowledge and reasoning, lighter on action. Also, MCQs can sometimes be solved via elimination or test-taking tricks rather than true understanding. Future benchmarks might incorporate hands-on tasks (there is mention of SecBench.js for JavaScript vulnerabilities as a separate work [150], which is more coding-focused). So a gap to fill is connecting QA performance with practical skill evaluation.
- **Design Patterns:** From SecBench, we can take the idea of **multi-dimensional coverage**. A smart contract security benchmark could similarly be multi-format: e.g. include multiple-choice questions on concepts (like "What does this Solidity keyword do in terms of security?"), short answers (explain an exploit scenario), and practical coding tasks (fix this contract). Covering multiple levels: basic knowledge (definitions of vulnerabilities) vs applied reasoning (analyzing a complex contract for vulnerabilities). SecBench's structure provides a template for systematically covering all these.
- **Contest and Community:** The idea of a question design contest is great for building a benchmark in a fast-evolving domain like blockchain security. We could invite security experts to contribute scenarios or vulnerable code snippets, ensuring the content is challenging and diverse. This crowdsourcing approach, as SecBench did, yields a robust pool of test items that surpass what a small team might come up with alone.
- **Automated Grading & Tools:** In an educational smart contract setting, we can similarly use tools or LLMs to assist in grading. For example, if asking "Explain why this Solidity function is vulnerable," we could use an LLM fine-tuned on security explanations to judge the answer (similar to SecBench's grader) – or even better, have a set of criteria that we check for (like key points that must appear). SecBench shows this is feasible at scale with careful prompt engineering.
- **Smart Contract Security Benchmark Outline:** Inspired by SecBench, one could envision *SolBench* (just as an example name) with tens of thousands of QA pairs on Ethereum security, across domains like DeFi, NFTs, etc., in English and perhaps another language common in the community (maybe Chinese or Japanese). It would have true/false, MCQs, and short answers. In addition, building on SecBench's limitations, we'd incorporate coding tasks evaluated by testcases (like a hybrid of SecBench and SWE-Bench). The multi-dimensional nature ensures comprehensive evaluation: does the agent know concepts (similar to knowledge retention)? Can it reason about novel scenarios (like reasoning questions)? And can it apply that by actually writing or fixing code (practical tasks)? Combining ideas from SecBench and others would yield a rich benchmark. SecBench specifically reminds us to cover *both knowledge and reasoning*, and to not neglect

the importance of language – perhaps even include *social engineering* or *policy* questions, which are part of security too. Overall, SecBench's success in compiling a broad benchmark encourages us that a similarly broad but context-specific benchmark (smart contracts) is viable and valuable.

SciCode

A. Source Materials:
- **Paper:** *"SciCode: A Research Coding Benchmark Curated by Scientists"* (M. Tian *et al.*, arXiv 2407.13168, Jul 2024) [151] [152]. Published as a NeurIPS 2024 dataset/benchmark paper (25 pages).
- **Website:** scicode-bench.github.io – contains the full benchmark description, problem list, and leaderboard [153] [154].
- **Leaderboard:** Hosted on the site, showing model performance (Claude 3.5-Sonnet ~4.6% in realistic setting) [155].
- **CBORG Portal:** Lawrence Berkeley Lab's portal mentions SciCode as scientist-curated (with 65 main problems, likely referencing an earlier count) [156].

B. Benchmark Deep Dive:
- **Goal:** Evaluate LLMs on **solving complex scientific coding problems**, replicating tasks that scientists encounter in research coding (e.g. simulations, data analysis, numeric methods) [157] [158]. The aim is to push LLM coding capabilities beyond leetcode-style puzzles to problems requiring domain knowledge, reasoning, and coding combined – essentially *AI-as-research-assistant*. SciCode tests if models can act as a scientist writing code to explore or validate scientific hypotheses.
- **Gaps Addressed:** It addresses the gap that most coding benchmarks don't require deep domain context or multi-step reasoning. HumanEval and similar just test implementation of a described function, usually self-contained. SciCode's problems often require understanding scientific concepts and possibly deriving part of the solution analytically before coding [157] [159]. It also emphasizes code *with numerical correctness and scientific validation*, not just any passing output. This measures higher-order skills: mathematical reasoning, using libraries appropriately, verifying results against known scientific facts, etc. No prior benchmarks specifically targeted *scientific programming*, making SciCode a pioneer in this niche.
- **Agent Type:** The agent here is essentially a **code-generation model** that is given a problem description (and optional background info) and must output code (likely Python, given scientific computing context) that solves the problem. The tasks often include writing a function or script and possibly returning certain outputs or making a figure. The agent may also have to produce an answer in a textual form (some problems might ask for a numeric result or confirmation that a simulation matches theory). But primarily, it's about generating correct and efficient code to a complex spec. No interactive tools are used; it's a one-shot or few-shot coding scenario, possibly with an evaluation harness.
- **Task Format:** SciCode consists of **80 main problems**, each decomposed into multiple subproblems (338 subproblems total) [160] [152]. The main problem is a high-level goal (e.g. "Simulate X physical system and verify Y property") which naturally splits into parts: perhaps (a) derive an equation or recall formula, (b) implement the simulation, (c) verify with a test, (d) analyze output. They provide **optional background descriptions** for each problem – these might be a summary of scientific context or hints [152]. Each problem has a **gold-standard solution code** written by a scientist, and a set of **domain-specific test cases** for evaluation [161] [162]. The domains cover 16 subfields across Math, Physics, Chemistry, Biology, Materials Science, etc. [163] [164]. Problems are often inspired by real research (some even tie to Nobel Prize topics as noted) [165], making them authentic and challenging. The tasks require **knowledge recall, reasoning, and**

code synthesis in combination – e.g. one must know a formula (knowledge), decide how to compute it or simulate (reasoning), then code it (synthesis).

C. Dataset Construction Methodology:
- **Curation by Experts:** They gathered problems from 16 diverse science disciplines by directly involving scientists from those fields [166] [163]. Each contributed problems representative of coding tasks they do. These likely came from published research code or well-known computational challenges in that field. They ensured problems were **robust and publication-grade**: many were used in real scientific workflows or to reproduce known results [167] [168]. This means solutions are non-trivial and have been validated scientifically (some even replicating published figures or results).
- **Validation Rounds:** Each problem went through **3 rounds of validation** – by in-domain scientists, out-of-domain scientists, and GPT-4 – to ensure clarity and correctness [168]. In-domain check confirms the solution is correct and the problem is meaningful. Out-of-domain check ensures it's understandable to a general scientist (so LLMs not specialized might still parse it). GPT-4 check gauged if it was already easy for a strong model, which it wasn't (GPt-4 likely failed most) and also helped fix phrasing. This rigorous vetting is akin to peer review for each problem.
- **Test Cases & Difficulty:** For each coding task, they wrote **unit tests and scientific verification tests** [169]. Not just checking output format, but verifying scientifically (like comparing output of simulation to theoretical values or known experimental data). This is beyond typical code benchmarks that just check exact output. Problems were classified by field as listed, but also we can infer difficulty: given the best model got only ~4.6%, these are extremely hard. The dataset includes hints in problem descriptions and clearly states assumptions (to prevent ambiguity). They also purposely included some historically significant problems (like replicating a Nobel-winning calculation) to ensure depth [170]. Size: 338 subparts suggests each main problem averages 4-5 sub-tasks, making them multi-step.

D. Evaluation Framework:
- **Automated Code Execution:** They run the model-generated code against the **scientist-written test suite** for that problem [152] [169]. If all tests pass, the subproblem is considered solved. This is similar to CodeEval/HumanEval style, but tests here are more advanced: they might check, for example, if the numerical error is within tolerance of an analytical solution [169]. They mention domain-specific tests – meaning the evaluation can involve non-trivial comparisons, like validating physical laws (perhaps using simulation results or known invariants).
- **Scoring:** They report metrics like **% subproblems solved** and **% main problems fully solved**. Given they decompose each, they can measure partial success (maybe a model solved 2 of 5 subparts). The headline result: Claude 3.5-Sonnet solved only 4.6% of problems in the *most realistic setting* (which presumably means using only the problem description without extra hints) [171] [155]. They likely differentiate between settings: perhaps one where background info is given vs one where it's not (to see if models use hints effectively). They may also measure success by domain to find which fields models handle better (e.g. maybe easier in basic math, near zero in quantum physics).
- **Realistic vs Simplified Setting:** The paper mentions "most realistic setting" performance [155], implying there might be an easier setting like providing the model with the solution outline or letting it see the optional background description. Possibly they tried an "open-book" scenario where the model gets the background text, and maybe a "closed-book" scenario without it, to mimic how a human might have reference vs not. The realistic one probably means just like a real scenario: model gets the problem and any necessary data, no additional hints beyond what a scientist might have.
- **Human Benchmark:** They might have also measured human performance or at least positioned the difficulty as "requires a graduate student". They note these are at the level a grad student might tackle as a

warm-up research project [113] (CritPt note, but SciCode similarly says only a tiny fraction solved by best model). So the gap between model and human is huge here, highlighting how far we have to go.

E. Uniqueness Factor:

- **Scientific Context Integration:** SciCode stands out by requiring both *domain knowledge* and *coding skill*. It's not enough to know Python; the model must understand scientific concepts (like what is a wavefunction normalization, or how to solve a differential equation). No other coding benchmark covers such domain-specific knowledge integration. It effectively tests *AI's potential in assisting scientific R&D*, a unique angle.
- **Complex Multi-step Problems:** Each SciCode problem is like a mini research problem with multiple steps, which is rare. Typical coding benchmarks have independent tasks, but SciCode's subproblems link together (the output of part (a) might feed part (b), etc.). This means models would ideally break down the problem – engaging capabilities like planning, and persistent state across sub-tasks – something unique in code benchmarks.
- **Extremely High Difficulty Ceiling:** SciCode deliberately chose problems far beyond current LLM abilities (4.6% success for best model) [155]. This gives a huge runway for future improvement. Many benchmarks saturate quickly (e.g. Codex nearly maxing out HumanEval). SciCode ensures we're not near saturation at all, highlighting unique aspects of reasoning that are unsolved.
- **Scientific Validation in Evaluation:** The way SciCode evaluates outputs by comparing against scientific truths (not just exact matches) is also unique [169]. For instance, verifying a simulation by checking it conserves energy or matches a known analytical solution is something no prior code benchmark did. It's an innovation in testing correctness not just programmatically but *semantically*, ensuring the code truly solved the scientific intent.
- **Community Signal:** SciCode's creation involved many institutions and 30 co-authors from diverse fields [166], reflecting a broad consensus on important problems. This multi-university effort is unique for a coding benchmark and indicates it was seen as filling an important need (AI for science evaluation).

F. New Benchmark Inspiration:

- **Future Work:** SciCode authors might expand to more domains (social science code? engineering?) or include multi-language coding (most is Python now). They might also incorporate the notion of interactive problem solving (an agent iteratively coding and checking results). Another angle is considering partial credit for solving some subparts – maybe to train models with reinforcement learning on incremental progress.
- **Identified Gaps:** SciCode's tasks are static and batch-evaluated. In real research, scientists debug and iterate. A gap is testing an agent's ability to debug code when initial attempt fails or refine a model. We could see future benchmarks introducing a feedback loop (like a code eval harness that tells the model which tests failed and see if it can fix, etc.). Also, SciCode is all in code; combining it with natural language (e.g. writing a brief report of findings) could be a next challenge.
- **Design Patterns:** The idea of *curating tasks from real expert workflows* is powerful. For smart contract security, we should similarly pick tasks that experts actually do (e.g. use real exploits from audit reports). SciCode's method of validation (multiple experts + GPT-4) for each problem can ensure high quality in our benchmark too. We want tasks solvable, clearly defined, but not trivial – expert input is key for that.
- **Multi-step structure:** SciCode essentially implements a mini "project" for each problem. A similar approach in a smart contract benchmark could be to present a small project (say 3 contracts interacting) and then have several sub-tasks: (a) identify potential vulnerability, (b) write an exploit script to demonstrate it, (c) patch the code, (d) verify the fix with tests. This echoes SciCode's subproblem chaining, forcing the agent to both find and fix issues and verify them. It's excellent for educational purposes – it mirrors how a security engineer works step by step.

- **Tolerance and semantic checking:** SciCode's strategy to accept solutions that may not be exactly like reference but are scientifically correct suggests using a similar forgiving check in our domain. For instance, if an agent finds a different exploit path than expected, we should still count it as success if it drains funds. Or if it patches differently but effectively, that's success. So, design grading scripts that check outcomes (does the exploit succeed? does the patched contract pass all tests?) rather than exact code match. This outcome-oriented evaluation, as SciCode did, is crucial for a robust benchmark.
- **Combining Knowledge + Coding:** SciCode proves that benchmarks can test pure knowledge (like deriving a formula) alongside coding. In a security benchmark, we might incorporate theoretical questions (what's the complexity of this attack?) with coding tasks. Possibly present a vulnerability scenario and ask both conceptual and coding questions. This blend ensures well-rounded evaluation, akin to SciCode's approach to require both recall and application.
- **Educational Focus:** SciCode's aim of shedding light on AI for scientific discovery parallels what we want for AI in security. The benchmark can highlight where AI fails – e.g. maybe it fails at logic in finding an exploit, analogous to SciCode showing failure at multi-step reasoning. This tells us what to teach or improve. SciCode's structure – optional background, multiple parts – could even be turned into a curriculum, guiding learning. Similarly, our benchmark could double as a learning tool: optional hints for each subtask, etc.
In summary, SciCode encourages us to make benchmarks that are *hard but structured*, that mimic real expert workflows, and that provide a lot of insight via fine-grained tasks. Applying these lessons will make a smart contract security benchmark much more impactful as both a test and a teaching instrument.

SWE-fficiency (SWE-ficiency)

A. Source Materials:
- **Paper Submission:** *"SWE-fficiency: Can Language Models Optimize Real-World Repositories on Real Workloads?"* (J. Ma *et al.*, submitted to ICLR 2026) [172] [173] . Contains 498 tasks, from Sept 2025.
- **OpenReview Link:** Submission ID 9644 (as of Dec 2025) [174] [175] .
- **LinkedIn Post:** Vijay Janapa Reddi's introduction "SWE-Efficiency Benchmark" (Nov 2025) [176] [177] .
- **Project Site:** swefficiency.com – mentioned in LinkedIn [178] (presumably hosts data or results).

B. Benchmark Deep Dive:
- **Goal:** Evaluate AI's ability to perform **performance optimization in software** – specifically, can models make code run faster without breaking it? [179] [180] . This shifts focus from functional correctness (the usual code benchmark goal) to *runtime efficiency*. In essence, SWE-fficiency asks: *given a working but slow program and its test/workload, can an AI refactor or rewrite it to achieve a significant speedup and pass all tests?* It targets the skill of performance engineering, which involves understanding algorithms, memory usage, etc., not just producing any solution.
- **Gaps Addressed:** It addresses the gap that coding benchmarks so far ignore runtime performance. Models may generate correct code, but often inefficiently. There was no standard way to measure if AI can optimize code. SWE-fficiency also tackles *long-horizon reasoning* – figuring out where bottlenecks are in a large codebase and how to fix them, which is different from writing code from scratch. It's a form of debugging/tuning knowledge that prior tasks haven't covered. Also, it introduces real-world **workloads** (actual data or usage scenarios to test speed), whereas typical benchmarks just run trivial tests. This ensures models are evaluated on meaningful improvements (e.g. making a data science library function 2x faster on real data).
- **Agent Type:** The envisioned agent is an **autonomous coder with profiling capability**. It gets as input: a code repository, a description of a performance issue or a slow "workload" (like a script that runs slowly),

and the test suite. The agent must analyze (maybe run tests to identify slow parts), then propose code edits. It likely will operate similar to a SWE-Bench agent (with repository access and tests), but focusing on speed. The tasks allow multi-step attempts: an agent could attempt an optimization, run the workload to gauge speed, iterate if not enough improvement. So this is more of an *optimize loop* than one-shot. However, in evaluation likely they give one shot to produce a patch that meets speedup target. Agents need capabilities like static analysis and understanding of algorithmic complexity.

- **Task Format:** There are **498 tasks**, each derived from real performance-improving pull requests across 9 popular repositories (like NumPy, Pandas, SciPy, etc.) [181]. For each, the benchmark provides the *original codebase with a known inefficiency*, a specific workload or input on which it's slow, and the *target* (the speed that an expert solution achieved). The agent must modify the code to match or exceed that expert speedup while still passing all tests [182]. The tasks often involve optimizing a function or method (e.g. vectorizing a loop, using a better algorithm, avoiding redundant computations). They included the relevant tests and also integrated coverage tools to ensure the agent's changes execute on the workload and don't break untested parts [173]. Each task effectively has a known solution by a human that achieved X times speedup; the model's goal is to reach >= X. Difficulty varies: some optimizations are small (2x faster by changing one line), others require substantial refactoring for 10x speed. Importantly, tasks are at **repository level** – meaning the code to optimize might span multiple files or require understanding interplay of components (so context is large, not a single function always).

C. Dataset Construction Methodology:

- **Data Mining:** They scraped GitHub pull requests that were labeled or described as performance improvements (e.g. PRs with titles like "Improve X algorithm performance") [173]. They filtered for those with measurable speed gains. Using commit history and keywords, they found candidate PRs, then likely manually verified that these PRs correspond to optimizing code (not just trivial fixes).
- **Pipeline for Validation:** For each PR, they reconstructed the scenario: take the *pre-optimization code* as the starting point. Identify the *benchmark or test* the PR author used to demonstrate speedup. This might involve reading the PR discussion or associated issue to find the example input that was slow. They then created a *workload script* or use an existing benchmark from the project that highlights the improvement [183]. Next, they confirmed the expert patch indeed speeds it up by running both versions under that workload (ensuring a baseline speed and expert speed). They probably set a threshold like "expert got 5x speedup, we'll require the agent to get at least say 5x or close". They also gathered the project's test suite as PASS/FAIL criteria to ensure correctness.
- **Ensuring Resolvability:** They mention combining static analysis, coverage tools, and execution validation to ensure tasks are properly set up [173]. For example, they ensure the test coverage hits the parts being optimized (so if an agent speeds up code but breaks something untested, ideally additional tests might catch it). Each task has a known solution (the human PR). They might have had to simplify some tasks if they were too hard or not generalizable. But it sounds like they kept them realistic – 498 tasks is large, meaning they could include many easier ones (small optimizations) and some very hard ones (major rewrites).
- **Languages & Domain:** Focus seems on Python data science libraries (NumPy, Pandas, etc.), possibly some HPC or ML frameworks. They likely limited to Python or similar, since it's easier to run and profile. This domain choice targets things like vectorization, algorithmic improvement, C extension usage, etc., which are common in those repos. It's a good fit because these are well-known bottlenecks and often have tests.

D. Evaluation Framework:

- **Metrics:** The key metric is **Speedup relative to baseline**. They measure the runtime of the agent's solution vs. the original code on the provided workload. If it meets or exceeds the "expert" speedup (or a fraction thereof), it's considered a success [180]. They reported that SOTA agents achieved on average <0.15×

the expert speedup [180] – meaning if experts got say 100% faster (2× speed), agents only got ~15% faster on average, far short. That quantifies performance gap. They probably measure both average speedup and fraction of tasks where agent at least matched expert (success rate).
- **Correctness Check:** The agent's patch must pass the project's test suite (ensuring it didn't break functionality) [182]. So evaluation is twofold: tests passing (must) and speed condition. The combination yields a binary success per task (solved or not). Possibly partial credit if an agent got some speedup but not full target, but likely they define success as meeting or exceeding expert's gain.
- **Infrastructure:** They automate running of tasks in isolated environments because performance measurement can be noisy. They might run multiple trials and average. They ensure the environment is consistent (hardware, libraries) for fair compare. Possibly they had to allow a certain runtime budget for each (some tasks might be heavy numeric computations).
- **Tiers of Difficulty:** Not explicitly stated, but tasks can be categorized by required speedup magnitude or type of optimization. The evaluation might consider those categories in analysis (like models might do okay on "low-hanging fruit" optimizations but fail on algorithmic ones). The coverage tooling suggests they also check if the agent's change touches the expected code area – maybe to verify it attempted to optimize where needed (though ultimate measure is speedup achieved, which inherently requires touching the right spot).

E. Uniqueness Factor:
- **Performance Focus:** This is the first benchmark to prioritize *non-functional requirements* of code (performance) rather than just correctness. It introduces a new dimension to coding evals: quality of solution. That's unique and important because in real software, writing correct code is step 1, making it efficient is step 2 – now AI must face step 2.
- **Repository-Scale Understanding:** SWE-fficiency tasks involve reading and modifying potentially large codebases to find bottlenecks, which is a different cognitive task than writing a small algorithm from scratch. This tests a unique capability: *codebase comprehension and targeted improvement*. Agents have to locate the relevant code for a workload, which is akin to needle-in-haystack debugging. No prior benchmark required that at this scale.
- **Real Workloads & Contamination-Resistant:** Using real repositories and actual heavy workloads (as opposed to toy inputs) is unique. It also ensures that improvements aren't trivial or cheat-y (the agent can't just special-case the example input; the tests likely prevent that and the spirit is to improve general performance). Also, by focusing on performance PRs, these tasks are somewhat niche; it's unlikely models have seen them in training because it's a very specific slice of code history (reducing risk of solutions being regurgitated). And even if they did see the final code somewhere, they still have to produce it given the original, which is not trivial. So it's fairly contamination-resistant by nature.
- **Highlighting New Failure Modes:** The results (0.15× expert speed) show models really struggle with optimization reasoning: e.g. they might fail to identify that a certain loop is the bottleneck or they may propose an optimization that doesn't actually help (or breaks something). This benchmark uniquely uncovers that today's LLMs, while decent at writing correct code, are very poor at improving code performance [180]. It shines light on limitations in *strategic reasoning about code execution*, which is a novel insight.
- **Cross-disciplinary Effort:** It intersects software engineering and systems optimization – bridging AI coding with concepts from compilers and algorithm design. It's unique in engaging those aspects; one might need knowledge of complexity theory or library internals to solve tasks, pushing AI to integrate different knowledge (some tasks might require using a better algorithm, or offloading work to a faster library like using NumPy vector ops instead of Python loops).

F. New Benchmark Inspiration:

- **Future Work:** Authors note significant underperformance, implying lots of room to innovate agent strategies (like profiling loops, etc.). Future benchmarks might incorporate other aspects of code quality (memory optimization, parallelization). Also, turning this into a **competition** could spur progress (who can build an agent to beat these tasks?). They also release their pipeline for others to add tasks [179], so maybe a community will extend it to more repos or languages (C++ performance tuning, etc.).
- **Identified Gaps:** The current tasks are mostly algorithmic optimizations; a gap might be more *micro-optimizations* (like bit-level or caching) or *concurrent optimizations* (e.g. making code thread-safe and faster). Also, measuring things like binary size or memory usage could be future expansions – making it a full spectrum software efficiency benchmark. Another gap is that this is still single-bug fix type; in reality, performance tuning can be open-ended (improve as much as possible). But they simulate that by requiring to match expert's improvement.
- **Design Patterns:** The idea of using **before/after snapshots from real code history** can be applied elsewhere. For smart contract security, we could similarly mine GitHub: find commits where a vulnerability was fixed, use the pre-fix as the task and require the agent to produce the post-fix. That's effectively what our manual tasks would be, but data-mining them systematically like SWE-fficiency did for performance is promising. It yields realistic tasks and a built-in ground truth. We must ensure we have tests and possibly an exploit script to demonstrate the vulnerability (like their workload demonstrates slowness). This pattern – leveraging real fixes – ensures tasks reflect actual issues and prevents us from needing to hand-craft everything.
- **Performance + Security:** Interestingly, some optimizations can introduce or remove vulnerabilities (e.g. caching might introduce stale data issues). While SWE-fficiency focuses on speed, a security benchmark can incorporate aspects of performance indirectly (like requiring a gas optimization in a contract fix). However, more directly, SWE-fficiency's approach to evaluating improvement by *quantitative metrics* (speedup) suggests thinking about *quantitative evaluation in security*, e.g. measuring how much gas was saved by an optimization, or how fully an exploit is mitigated (like residual risk quantification). Usually security is binary (vulnerable or not), but perhaps difficulty can be measured (some fixes might partially mitigate). Still, likely stick to binary pass (exploit fails or not).
- **Autonomous Agent Loop:** The concept of letting an agent iteratively improve and test, which is natural for optimization, could inform an **interactive training mode** for security too. For example, an agent could iteratively try exploits and refine a patch until all tests pass – akin to performance agent trying improvements until speed threshold met. This is more in training than evaluation, but a benchmark could allow multiple attempts and measure how quickly an agent converges to a solution. That could be an interesting metric (like how many tries to secure the contract?), analogous to how many tries to reach speed target.
- **Holistic Code Understanding:** SWE-fficiency tasks require understanding large context and non-local effects (optimize here to speed up there). For security, similarly, vulnerabilities can span multiple contracts or require understanding interplay (e.g. an issue in one contract exploited via another). We can craft tasks that require analyzing multiple files (like a protocol of contracts) to locate and fix a vulnerability. This would push beyond single-file toy problems.
- **Encouraging Efficiency Mindset in Security Education:** The idea that code can be functionally correct but suboptimal parallels code that "works" but is insecure. Both require deeper insight to improve. In an educational setting, one could combine them: ask students to not only fix a bug but also make the contract more gas-efficient. Foundry even has gas snapshot tests, so one can measure gas improvement similarly to speedup. Perhaps as an advanced dimension: a challenge where the best solution is one that both patches the bug and saves gas (like using `unchecked` in Solidity to save gas while avoiding overflow because one knows inputs are safe after fix). It's a stretch, but could be a cool multi-objective challenge.

In summary, SWE-fficiency expands what we consider “solved” in code – not just correct, but optimal. For smart contracts, analogously, “solved” could eventually mean not just no vulnerabilities, but also cost-effective and following best practices. This benchmark’s philosophy nudges us to think of quality metrics in our domain and perhaps incorporate them gradually. Initially, security is the focus (no hacks), but eventually, a truly robust benchmark could also encourage improvements on other axes (gas, maintainability). SWE-fficiency shows it's possible to evaluate such aspects objectively by reusing real-world criteria, which is a powerful lesson for designing comprehensive educational benchmarks.

AlgoTune

A. Source Materials:
- **Paper:** *“AlgoTune: Can Language Models Speed Up General-Purpose Numerical Programs?”* (O. Press *et al.*, arXiv 2507.15887, 2025) – NeurIPS 2025 Datasets & Benchmarks [184] [185].
- **Project Site:** algotune.io – contains authors, arXiv link, code, and interactive logs [186] [187].
- **GitHub:** `oripress/AlgoTune` – benchmark tasks and agent (AlgoTuner) implementation [188].
- **Medium Article:** *“How Fast Can AI Actually Code? Inside AlgoTune’s $1 Gauntlet”* (Nov 2025) [189] – likely a popular explanation.
- **HuggingFace Spaces:** Possibly hosts the AlgoTuner trajectories (the site shows logs for each task).

B. Benchmark Deep Dive:
- **Goal:** Test whether AI agents can **optimize classical algorithms and math code for runtime performance**, by writing more efficient code than a baseline implementation [187]. If SWE-fficiency deals with real libraries, AlgoTune focuses on well-known algorithms (like sorting, matrix ops, cryptography) and challenges the AI to “tune” them for speed while preserving correctness. The goal is to measure if LLMs can not just solve a problem, but find an optimal or faster solution than the obvious one. It’s a controlled environment to evaluate algorithmic optimization skills.
- **Gaps Addressed:** Prior code benchmarks care only about correctness. AlgoTune specifically checks *performance improvement relative to a reference solution* [190]. It addresses the gap of *algorithmic efficiency*. Also, unlike SWE-fficiency’s big code context, AlgoTune’s tasks are more self-contained functions – so it zeroes in on algorithmic thinking (like choosing the right data structure, applying math tricks). Another gap filled: evaluating iterative refinement. AlgoTune’s agent (AlgoTuner) engages in a loop of code edits and timing tests [191], showcasing how an AI might do performance engineering iteratively, which no static benchmark had done. It’s also a fine-grained measure: each task has a known optimal or at least a target speed, so we can quantify how close the AI gets.
- **Agent Type:** The benchmark expects an agent that can use an **evaluation loop**: propose a solution, run it on some inputs to measure speed, and use the feedback to improve. In practice, they built an agent named **AlgoTuner** that is a framework where an LLM can iteratively edit code, and after each edit, the code is run to get a timing [192] [193]. So the agent is not one-shot; it’s more like an autonomous coder with a test harness. For evaluation, they used AlgoTuner with various models (GPT-5, Claude, etc.) to see how much speedup each can achieve [193]. The tasks themselves allow multiple iterations, but final outcome per model is measured. So it’s an **interactive agent benchmark** albeit constrained to one environment (the tasks and their test-runner).
- **Task Format:** There are **~155 tasks** (the site says “more than one hundred” and snippet mentions 154/155) [188] [194]. Each task is a stand-alone algorithmic problem, typically phrased as “Write a function to do X that runs faster than the reference implementation while producing identical results” [190]. Examples given: gzip compression, AES encryption, SVD (linear algebra) – so covering classic algorithms in sorting, cryptography,

numerical linear algebra, etc. [195] [190]. For each task, they provide: a **reference implementation** (a correct but not optimized solution), a **generator for random inputs** (to test with different sizes) [196] [197], a **checker** to verify correctness of output, and a **reference solver (baseline)** which they likely time against [196] [198]. The agent's goal is to write code that produces the same output as the reference but runs faster on large inputs. So tasks often involve making code more algorithmically efficient (e.g. using better complexity or utilizing libraries like NumPy for vectorization). They mention tasks from math, physics, CS fundamentals, collected from domain experts [188], meaning each is probably a known puzzle or function that has known optimal approaches.

C. Dataset Construction Methodology:

- **Task Sourcing:** They asked domain experts (the author list includes algorithm specialists, numeric computing folks, etc.) to suggest widely used functions that have naive vs optimized implementations [188] [187]. They ended up with 154 tasks covering various domains – presumably focusing on things that can be optimized by code-level changes. Some tasks might be "implement quicksort faster than bubble sort baseline" or "compute prime numbers faster than naive trial division". Others might require using existing libraries (like using efficient matrix operations). They ensured tasks are solvable in code under certain constraints (like using Python, or C if embedded). It appears tasks are in Python, since reference shows Python code example (the PCA solve snippet with sklearn in site) [196] [199].
- **Benchmark Setup:** For each task, they wrote: a `generate_problem` function to produce input data of varying size [196], an `is_solution` validator to check output correctness [200], and a `solve` function as the reference solution (often using a straightforward approach or existing library) [201]. The evaluation environment can then time the agent's proposed `solve` on generated input sizes. They likely set a **score metric** like "speedup factor = reference_time / agent_time" averaged across some input sizes or at a certain large input size. The **AlgoTune score** reported is the harmonic mean of speedups across all tasks for a given model [202] [203] – effectively an overall performance figure.
- **Iterative Agent Implementation:** They developed AlgoTuner to let models improve solutions. This included letting the model see timings and adjust. The site shows that each task's entry has the "complete conversation between model and environment" with edits and timing feedback [204]. This methodology is unique, capturing agent trajectories for each attempt, which gives insight and perhaps training data for future improvement (they publicly share these trajectories logs). The tasks themselves were carefully chosen to have *some headroom for optimization*, not trivial. Also, they ensured correct baseline and likely gave the models some hints (maybe telling them results of profiling or something), though main hint is just speed feedback.

D. Evaluation Framework:

- **Success Metric:** Each task yields a *speedup factor*. They considered a task "solved" if the model achieved >1× (i.e. any speedup) or perhaps above a threshold like >1.1×. But from a benchmark perspective, they mostly look at the numeric speedups and aggregate them. They made a leaderboard where the **AlgoTune score is harmonic mean of speedups on all tasks** [202]. Harmonic mean is appropriate since it penalizes any tasks with low speedup heavily (so you need consistent improvements). The top model (OpenAI's o4-mini presumably GPT-4 variant) got ~1.72× average speedup [205], meaning on average it made code 72% faster than baseline – pretty good, but obviously these are surface-level improvements as they note.
- **Correctness and Validation:** `is_solution` function ensures the optimized code returns correct results on some test inputs [200]. If a model's code fails correctness, it's not counted even if fast. The iterative agent uses correctness check each iteration to ensure it doesn't trade correctness for speed. At final evaluation, only correct solutions count. The agent must achieve speedup *and* maintain correctness to contribute to the score.

- **Single vs Multi-run:** The official metric uses one run of AlgoTuner per model to find best solution for each task. If the agent sometimes fails, presumably they could run multiple times and take best outcome, but likely they did a fixed run and took that outcome to avoid cherry-picking. That's why results are given with ± (some uncertainty) due to agent stochasticity [79].
- **Transparency:** They made all solution trajectories public, which not only helps trust the evaluation but also provides data for further analysis (like seeing patterns in model optimizations or common failures). This is novel for a benchmark – releasing the entire model interactions.

E. Uniqueness Factor:
- **Iterative Optimization Loop:** AlgoTune is unique in its *closed-loop evaluation*, where the model is part of a feedback loop with execution environment. It's more like measuring an agent's capability to self-improve code, rather than just one-shot answer. This dynamic evaluation is a step toward more autonomous AI systems (like AutoGPT style, but concretely applied to coding tasks with an objective metric).
- **Focus on Efficiency Gains:** It explicitly measures how much faster code can get, a dimension ignored by others. This required building not just tasks but the whole instrumentation to measure run-time precisely. That is unique – combining software benchmarking techniques (timing, scaling tests) with AI evaluation. It's an intersection of performance engineering and AI eval not seen before.
- **Domain of Numerical Algorithms:** The benchmark covers a broad swath of algorithms (154 tasks across math/CS/physics problems), which means it's testing models in general algorithmic thinking beyond just typical interview problems. Some tasks might involve continuous math or physics simulation where vectorization is key – these are corners not touched by typical coding tests. It's unique in capturing both *theoretical algorithmic complexity* and *practical code-level micro-optimizations*. For example, tasks like AES encryption might require using bit operations or avoiding Python overhead, testing if the model "knows" low-level performance tricks.
- **Quantitative Score for Agentic Behavior:** Because they got numeric scores per model (GPT-5 had 1.67×, Claude ~1.52×, etc.) [206], it gives a clear ranking of models in an agentic scenario. This is unique as most model benchmarks either measure accuracy or qualitative scores, but here we have something like "model X makes code run Y times faster on average". It's a fresh way to quantify capability.
- **Insights on Limits:** They noted models achieved "surface-level speedups" but no novel algorithms [193] [207]. That uniqueness factor – it revealed that models tend to do things like micro-optimizations or leverage existing libraries, but they didn't invent fundamentally better algorithms. This speaks to the current limit of AI creativity in algorithms and is valuable feedback to research (maybe need better algorithmic reasoning training).

F. New Benchmark Inspiration:
- **Future Work:** AlgoTune could be extended to more complex tasks or multi-language (maybe have C++ versions). Or focus on other resources like memory optimization tasks. Another idea is multi-objective (speed vs memory trade-offs). They provided the trajectories, which might be used to train improved agents (like an RL fine-tune). So future work might create a "AutoCoder" that learns from AlgoTune logs to get better at optimization.
- **Identified Gaps:** While the tasks cover many algorithms, they are still relatively small-scale programs. Real software might need combining multiple algorithmic improvements globally; AlgoTune tasks are isolated functions. Also, the iterative loop is simplistic (the agent sees runtime, but not detailed profiles). A future agent might query a profiler tool. For our interest: bridging the idea to security, as below.
- **Transferable Patterns:** The interactive approach can map to security: imagine an agent that iteratively tries to exploit and refine its approach or tries to patch and tests, akin to how AlgoTune tries code and times it. A *"SecTune"* could be an agent that tries exploit payloads and learns from which one succeeded (though

that might risk training on the eval). But as an eval, we could allow an agent multiple attempts to hack a contract with limited feedback (like gas used or error messages) and measure success in terms of number of attempts or time to breach. That's analogous to runtime feedback. It would measure how efficiently an agent can penetrate security, which is a scary but interesting benchmark. Conversely, an agent fixing a contract might iteratively run tests and refine until all pass – evaluating if it can triage progressively.
- **Quantitative metric for security:** AlgoTune shows you can go beyond pass/fail into "how much better?". For security, if we had metrics like how much fund the attacker stole or how much gas saved in a fix, we could have partial credit metrics. For example, if an exploit agent only steals 50% of funds vs. an optimal 100%, that's a measure. Or if a patch closes some vulnerability but not another, maybe exploit difficulty increases (hard to quantify). Security mostly remains binary (secure/insecure), but perhaps aspects like gas cost can be integrated.
- **Encouraging iterative learning in education:** For a training scenario, one can do like AlgoTune and let students (or AI) iterate on their code with immediate performance or security feedback. Foundry allows printing gas usage or execution traces. One could imagine an educational game: "improve this contract's gas usage" with live feedback. This merges security and efficiency (because in Solidity, gas efficiency is a security factor sometimes – too high gas can break things). The AlgoTune pattern of iterative refinement with an automated judge could make learning interactive.
- **Benchmark integration:** We might incorporate a subset of tasks in our benchmark that look at gas optimization or micro-optimizations in contracts. For instance, one task could be "refactor this Solidity function to be cheaper in gas by at least 2x while preserving behavior." That's directly analogous to AlgoTune tasks but in smart contract context. It would teach about efficient patterns (like using `calldata` vs `memory`, unchecking math, etc.). This could be a "bonus round" in a security benchmark – not core vulnerability finding, but an adjacent skill that's valued.
- **Autonomous Security Agent:** AlgoTune's demonstration of an autonomous agent loop is inspiring for building an autonomous security auditor or exploiter agent. We see that with small tasks and feedback, the AI can gradually improve. A similar framework could allow an AI to test a contract, find something, adjust strategy. This might be too advanced now, but building a benchmark around it could spur development of such agents. Already, frameworks like Agents (OpenAI) or AutoGPT are exploring these loops.
In conclusion, AlgoTune's key inspiration is **the iterative agent evaluation** and focus on improvement, which can be translated to iterative vulnerability discovery or patching. It also underscores measuring beyond binary success, pushing us to think how to quantify partial successes or efficiency in security fixes.

SRE-Bench

A. Source Materials:
- **GitHub Repo:** `agentkube/SRE-bench` – description of SRE-bench tasks (Dec 2025) [208].
- **Rootly AI Labs (Groq OpenBench) Blog:** "Rootly joins Groq OpenBench with an SRE-focused benchmark" (Aug 28, 2025) [209] [210] – describes a similar SRE benchmark integrated in OpenBench.
- **Latent Space Podcast (Oct 2025):** Mentions "SRE-bench: Cambrian explosion of code evals" [211].
- **Awesome SRE Agents List:** likely references SRE-Bench (though the GitHub snippet is more a list) [212].

B. Benchmark Deep Dive:
- **Goal:** To evaluate AI agents on **Site Reliability Engineering (SRE) tasks** – essentially, real-world ops and incident response scenarios, especially in cloud/Kubernetes environments [208]. The goal is to see if an AI agent can troubleshoot outages, make safe infrastructure changes, triage alerts, and follow SRE best practices autonomously. It extends coding agents into the realm of DevOps/SRE, which involves not just

writing code but managing systems.
- **Gaps Addressed:** Prior benchmarks (like Terminal-Bench) cover DevOps tasks in a sandbox, but SRE-Bench explicitly focuses on *reliability and incident workflows*: e.g. diagnosing a failing service, adjusting Kubernetes configs, analyzing logs. It fills a gap by testing *decision-making under incomplete information and high-stakes constraints* (like an incident with pressure). It also emphasizes *policy-aware actions* (SREs have runbooks and must avoid making things worse). Essentially, it moves beyond pure technical tasks to scenario-based evaluations where judgment and adherence to procedures are key – a nuance other agent benchmarks hadn't fully addressed.
- **Agent Type:** The agent is akin to an **automated on-call engineer**. It likely interacts in a CLI environment or possibly via a chat interface with a system. It might be given monitoring data, logs, a system state, and possibly a user prompt ("Site is down with X error"). The agent can execute commands (kubectl, etc.), look at logs, apply fixes (like rolling back a deployment). This is an interactive agent that needs to gather info and take multi-step actions, similar to Terminal-Bench but more focused on large-scale system tools (Kubernetes, cloud APIs) rather than a single machine's terminal. Policies (like incident playbooks) might be provided and must be followed. The description mentions tasks across Kubernetes environments, which implies the agent likely works within a simulated or actual K8s cluster environment with certain resources representing a production system [208].
- **Task Format:** Examples of tasks: "incident response – e.g. database CPU spiking, agent must identify cause and mitigate", "infrastructure change – e.g. increase replicas of a service safely", "observability triage – interpret dashboards/logs to pinpoint an error", "reliability improvement – e.g. add an alert or failover config to prevent future incident" [208]. Each task probably comes as a scenario description (problem statement), initial system state (like current cluster state, logs, metrics), and a target end state (incident resolved, or improvement implemented without breaking things). The tasks might be open-ended (there might be multiple correct approaches as long as it's fixed with minimal side-effects). Because these are high-level tasks, evaluation likely involves checklists or LLM judging to confirm if the agent's actions achieved resolution in line with best practices (like no policy violated, correct root cause addressed). Rootly's SRE benchmark, integrated into OpenBench, had four tests with ~1200 samples each focusing on things like incident triage, likely similar tasks [213]. It suggests SRE-Bench might be multi-part: e.g. a question-answer part (like diagnosing cause given logs) and an execution part (applying a fix via commands).

C. Dataset Construction Methodology:
- **Inspiration from Real Incidents:** SRE-Bench likely drew from real outage postmortems and common SRE tasks. Possibly SREs at AgentKube or Rootly enumerated typical scenarios (service crash, network latency, misconfig) and created simulated environments to mimic them. For example, they might have a Kubernetes cluster with a deployment that has a bug (like a memory leak causing OOM) for the agent to find and restart that service with proper config.
- **Simulated Data:** They probably simulate system metrics (like an alert that CPU is 100% on service X), logs that contain error traces, etc., to feed the agent. Setting up these scenarios means prepping containers/pods or using something like `Kind` (K8s in Docker) with instrumented conditions. They also define success criteria for each scenario (like "web app returns HTTP 200 again within SLA" or "agent added an alert rule and tested it"). Given Rootly's benchmark had ~1200 samples per test [214], they might have generated many variations of incidents by randomizing certain parameters (like different services failing, or different times). Possibly LLMs were used to generate realistic log messages or user queries (Rootly used GPT-4 to generate data entries and user simulation content for tasks) [215] [216].
- **Policy & Runbooks:** Part of setup is providing an agent with relevant runbook or policy info (like "If database is down, do X; If 5xx errors spike, check these logs..."). They likely curated these from industry SRE guidelines or wrote simplified ones for the scenarios. This ensures tasks test if agent follows rules (like

escalate to human if needed, etc.), aligning with what Rootly blog said about existing benchmarks lacking reliability measures [210].
- **Multi-step Solutions:** They would ensure each scenario indeed requires multiple steps (investigation then action then verification) to solve, so that it's not trivial for an LLM to just answer with one command. This might be done by making some info only accessible via certain queries, forcing agent to dig (like only through logs can they find the issue cause, so they must run a `kubectl logs` command). Essentially, scenario design likely enforces a need for interaction.

D. Evaluation Framework:
- **Outcome-based:** Success if the incident is resolved or the improvement implemented correctly. They can verify by checking system state at end (like all services healthy, error rate back to normal). This might involve automated scripts (like ping the service to see if it's up). Similarly, if the task was to update config, they could query the config after agent actions to see if it matches required state (and all pods are running, etc.).
- **Policy compliance:** They also need to check the agent didn't break any rules (like it didn't open security holes or it followed instructions to notify on Slack, etc.). This is trickier to automate; they might use an LLM judge to read the agent's actions/commands and conversation to flag violations. Or some tasks have clear no-nos (e.g. "don't delete a pod without draining" – if agent does a direct delete, the framework can detect that command). Possibly they encode safe actions and dangerous actions and monitor agent commands.
- **LLM-as-judge:** If tasks involve freeform decision-making or explanation, they might grade via LLM. Rootly's evaluation approach in OpenBench likely did something like run model responses and use some rubric (they mention multiple-choice benchmarks in their Medium, but for open tasks, likely LLM judging or pre-defined correct actions). Because each scenario can have multiple valid solutions, evaluation likely focuses on whether the outcome was achieved without unintended consequences (the cluster still stable, etc.).
- **Metrics:** They probably measure success rate per scenario. Rootly's "four tests" suggests maybe categories like Incident Detection, Incident Mitigation, Postmortem Analysis, etc., each with separate score. They integrated with OpenBench to allow anyone to test their model's SRE capabilities easily [217] [218]. So the metric might just be overall % of scenario tasks solved. They also may measure reliability metrics (like in tau-bench pass^k), since SRE tasks often need consistency. Rootly highlighted that general benchmarks don't reflect SRE's need for reliability and consistency [210], so maybe repeated trials or multiple tasks measure if the agent fails unpredictably.

E. Uniqueness Factor:
- **Domain-Specific (SRE) Knowledge & Skills:** SRE-Bench is unique in focusing on the operational side of software, rather than development or static Q&A. It requires an agent to combine coding/scripting with systems knowledge (Linux, networking, databases) and *real-time decision making*. It's essentially bringing AI evaluation into the IT operations domain, which hadn't been done.
- **Live System Interaction:** It likely involves an *active system* (a running K8s cluster) and the agent intervening in it. That interactive, environment-manipulating evaluation (especially distributed systems) is unique. Terminal-Bench did single-machine tasks; SRE-Bench up-scales to cluster-level tasks, which is a new level of complexity.
- **Policy and Safety Considerations:** By testing adherence to policies and not making things worse, SRE-Bench touches on AI safety in a concrete way. The agent must be not only effective but also safe – a combination rarely explicitly tested. This is akin to an AI following rules under stress, which is unique among agent benchmarks.
- **Holistic Scenarios:** It evaluates multi-faceted tasks (communication, diagnosis, action). For instance, an

agent might have to explain to a user what went wrong (like updating a status page). If included, that's testing communication skills too. SRE tasks cover documentation and escalation as much as technical fixes. Incorporating that makes it a holistic test of an AI assistant working in a team. This is beyond purely technical benchmarks and heads toward evaluating AI on soft skills in context – quite unique.
- **Emergent Use of Tools:** SRE-Bench likely allows a wide range of tools/commands. The agent might have to choose which diagnostic to run (like `top`, `curl healthcheck`, check metrics API). This is less constrained than some benchmarks with defined API calls. It's testing an agent's *tool selection and exploration* ability in a realistic tool-rich environment – a significant step up in complexity.

F. New Benchmark Inspiration:
- **Future Work:** SRE-Bench could expand to more cloud providers (AWS incidents?), incorporate user communication tasks (like the AI writes an incident report), and measure longer-term consistency (like can it manage reliability over days?). It also might integrate with human feedback, since SRE often involves handing off – maybe multi-agent collaboration (one agent fixes, one reviews).
- **Identified Gaps:** The main difficulty is evaluation automation because SRE success can be fuzzy (was root cause fully identified or just mitigated?). They might not fully solve that; future benchmarks could involve human evaluation for those parts or improved LLM judging. Another gap might be covering security incidents (if an incident is a cyberattack, does the agent handle it?). Perhaps not in current version but could be in future. That overlaps with our interest – maybe an SRE agent dealing with a security breach (would need to do forensics and patch quickly).
- **Design Patterns:** The scenario-based approach with complex state is reminiscent of war-game exercises. For smart contracts, we could create **incident response scenarios in a blockchain context**. E.g., "The protocol's liquidity pool is being drained (like an ongoing hack in progress). The agent must diagnose which contract is vulnerable and trigger a pause or patch to stop the exploit." This merges security and SRE: it's like a DeFi incident response. The agent would have to parse blockchain events (analogous to logs), identify the attack transaction, and call an emergency stop function or deploy a fix. We'd evaluate if it saved funds in time (stop the exploit) and followed proper steps (like notifying governance). This would be a very challenging scenario, but realistic as an ultimate test of an AI in security ops.
- **Policy in Smart Contract Management:** We can incorporate the idea of **runbooks and policies** for managing smart contracts (e.g., "if price oracle deviates >X, pause the system"). An agent benchmarking in this domain could be given these rules and must execute accordingly. This ensures evaluating not just technical ability to exploit or patch, but also procedural compliance (like waiting for consensus or not leaking private info, etc.).
- **Cross-ecosystem tasks:** SRE-bench highlights tasks that cross systems (network, db, app). For smart contracts, cross-ecosystem might be bridging tasks (like one chain to another, or contract to front-end). We can have tasks where the agent might need to modify both solidity code and some config in the deployment scripts – testing multi-domain knowledge.
- **Interactive Tools:** We should consider giving agents a suite of blockchain analysis tools (like tenderly, block explorer APIs) similar to how SRE agents have kubectl, logs, etc. The agent's ability to choose the right tool (trace a transaction vs. inspect contract state) can be part of evaluation as in SRE-bench (choose correct diagnostic commands).
- **Community and Integration:** SRE-Bench showing up in OpenBench (which supports 35+ benchmarks) [219] hints that benchmarks are moving to platforms for easy model eval. We might aim to integrate our future benchmark with such frameworks for adoption. Community signals (like Rootly open sourcing half dataset and making others private for eval) [213] is similar to how we might handle sensitive security tasks (maybe keep some challenges private to avoid training leakage, but allow evaluation via API).
In summary, SRE-Bench inspires scenario-driven, tool-using, policy-constrained benchmarks that mirror

operational reality. For smart contract security, adopting that style means framing challenges not just as isolated bugs, but as unfolding events in a live system (even if simulated). This could elevate the benchmark from a static puzzle collection to a dynamic "cyber-range" for AI – aligning well with the ultimate goal of AI that actively protects systems. It's ambitious, but SRE-Bench shows a path by tackling analogous complexity in reliability engineering.

Vending-Bench

A. Source Materials:
- **Paper:** *"Vending-Bench: A Benchmark for Long-Term Coherence of AI Agents"* (Andon Labs, arXiv 2502.15840, Feb 2025) [220] (not explicitly quoted but reference from Anthropic blog suggests it exists [221]).
- **Anthropic "Project Vend" Blog:** (Oct 2025) describing a physical experiment and referencing Vending-Bench [222].
- **Andon Labs Announcements:** "Vending-Bench 2" on andonlabs.com (likely late 2025) [223], and the initial introduction (perhaps early 2025) [224].
- **LLM-Stats Leaderboard:** Vending-Bench 2 listing (perhaps models and scores) [225].

B. Benchmark Deep Dive:
- **Goal:** Evaluate **long-horizon planning, memory, and adaptive decision-making** of AI agents by simulating a scenario where an agent must run a small business (a vending machine operation) over an extended period [224] [226]. Essentially, test if agents maintain coherent strategies and avoid compounding errors when operating autonomously for many steps. The vending machine scenario is chosen as it's simple enough (buy stock, set prices, sell items) but requires planning (inventory management, pricing strategy) and has a notion of objective (profit) that unfolds over many interactions. The key is measuring if the agent can handle **hundreds or thousands of sequential decisions** without losing track or going unstable (thus "long-term coherence").
- **Gaps Addressed:** Traditional benchmarks are short episodic tasks; Vending-Bench addresses the gap of **evaluating agent behavior over long durations** (days of simulated time, maybe dozens of user interactions). It also tests *autonomy in open-ended tasks* – there's no single correct answer, but ongoing performance metrics. It surfaces issues like forgetting earlier context, diverging from a goal, or unsafe exploration. As such, it probes capabilities like persistence, consistency with initial instructions, and adapting to unexpected events (like in business, sales patterns may change). No prior benchmark really targeted these aspects in a grounded scenario.
- **Agent Type:** The agent is essentially a **CEO AI** for a vending business. It likely interacts through an environment API that provides world state updates (inventory levels, sales events, maybe customer messages) and accepts the agent's decisions (what to stock, pricing, etc.) on a periodic basis (like daily). It might operate in a simulated day/night cycle, making one or more decisions per day (like restock or change price). The environment simulates outcomes (sales, spoilage, etc.) and updates the agent. It's an *interactive simulation agent*, not code generation. The agent's outputs are actions (buy X of item Y at price Z, etc.) or communications (maybe marketing to customers). The agent also presumably has tools to get info (maybe it can query sales stats). There might be a memory persistence mechanism to allow it to note plans (the Anthropic blog mentions the agent "had tools for keeping notes and preserving info beyond context window" [227], i.e. an external memory it can write to, to combat forgetting over a month-long run). So it's testing an advanced agent with working memory extension and tool use (web search, email ordering wholesalers, Slack with customers as per blog) [228] [229]. This is complex – beyond a single LLM call, it's a multi-component agent.

- **Task Format:** For benchmarking (not the Anthropic physical demo but the simulation), they define an environment with a **simulated vending machine business** [230]. This includes: initial money, product catalog, wholesale prices, storage limits, customer demand patterns, etc. The agent's task is to maximize profit over, say, N days. The simulation has events like items selling (depending on price, demand, maybe random variability), machine breakdowns possibly, trends (like certain items get popular). The agent can take actions each day (set prices, reorder stock, pay for machine maintenance, etc.). There is a *score* at the end like total profit or success of business (if it goes bankrupt, that's failure) [231] [232]. The original Vending-Bench may have an initial scenario and measure how well an agent does (profit made) as well as if it stays within rules. There's mention of "long-term coherence" specifically, so they may also measure if the agent's actions have contradictions or regressions (like raising price then cutting drastically without reason might be incoherent strategy). Possibly metrics like cumulative reward and an assessment of policy consistency. In Vending-Bench 2 (extended horizon), they might double the timeframe to see if models degrade further. The scenario is straightforward enough to standardize: all agents face the same simulated market conditions, so profit is comparable.

C. Dataset Construction Methodology:
- **Simulation design:** Andon Labs likely built a simulation environment (like a custom OpenAI Gym environment). They set fixed parameters for machine capacity, starting budget, etc., to be fair across runs. They might incorporate random seeds to ensure some unpredictability but could average results over multiple runs. The environment includes cause-effect rules (if price too high, fewer sales; if out of stock, potential missed sales; can pay Andon Labs for restock actions with a fee, etc. as described in the system prompt excerpt [233] [227]). They tuned it so that the task is challenging but feasible (the Anthropic experiment found the AI failed in some "curious ways" despite being close to success [234]).
- **Coherence checks:** To specifically measure coherence, they may have instrumented certain scenarios, e.g., track if agent contradicts itself (like drastically oscillating strategy, or forgetting a goal it stated). They might label some behaviors as incoherent. Possibly the environment logs could be analyzed by another script or LLM to count coherence violations. Not entirely clear, but "coherence" might be indirectly measured by final outcome (an incoherent agent probably fails or at least yields lower profit).
- **Vending-Bench 2 differences:** Possibly extended time horizon or additional complications (like competitor appears, requiring adaptation). The Andon Labs Vending-Bench 2 announcement might mention what's new: likely "longer horizon planning, more diverse events". They may open-sourced part of it or held a competition (since llm-stats has a leaderboard). They also likely used Vending-Bench as part of an AI safety evaluation (anthropic interest), so the design might include trick scenarios (like a tempting but unethical opportunity to test if agent takes dangerous shortcuts - e.g. misreport earnings).

D. Evaluation Framework:
- **Metrics:** The primary metric is **total profit** (or could be an index of business health) after the simulated period [232] [235]. For instance, how much money agent has or if it's bankrupt. They might also measure if agent avoided bankrupting (binary success). Profit is a nice scalar reward. Additionally, they could measure *profit optimality* relative to some theoretical maximum (if known). If long-term coherence is the goal, they might define some metrics like variance in strategy or number of rule violations. Anthropic's blog indicates success is not binary (Claude "made too many mistakes to run the shop successfully" but did some things well [236] [237]). So evaluation may include qualitative analysis of agent's performance (like listing things it did well vs poorly – they mention identifying suppliers well vs hallucinating details and selling at a loss [238] [239]). Possibly they had a rubric to categorize behaviors (the medium article snippet suggests someone reflecting on what the AI did). For pure benchmark, though, likely profit is the main objective metric, and maybe whether it remained solvent (not bankrupt).

- **Stability and Coherence:** To quantify these, they could track how often the agent's decisions changed drastically or how often it violated context (like forgetting inventory limits and ordering more than capacity – if environment forbids that, it's fine, but if agent tries, that's a mistake). These could be logged and counted. The Andon blog noted specific failures: ignoring obvious arbitrage (not taking $100 for Irn-Bru), hallucinating a Venmo address, selling at a loss due to zealous response [237]. Those are coherence/flaw examples. For benchmarking, maybe they categorize episodes qualitatively. But as a numeric measure, maybe "Coherence Score" or "Stability Score" might be something like negative count of major mistakes. Possibly an LLM could analyze the agent's log after the run and assign a coherence rating (anthropic could do that with Claude as a judge ironically). Not sure if that's formalized in the benchmark or just part of analysis.
- **Leaderboard Approach:** The llm-stats shows a Vending-Bench 2 leaderboard, presumably ranking models by something like profit (maybe normalized if random events). They probably run each model agent multiple times and average profit. Also note multi-run reliability: can it consistently succeed or sometimes bankrupt? They might incorporate that (like pass^k style, how often did it bankrupt if we run 5 trials?).

E. Uniqueness Factor:
- **Extremely Long-Horizon Autonomy:** Vending-Bench is unique for testing *hundreds/thousands of decisions coherence*. It's a benchmark for agent *autonomy over time*, not just task completion. Most evals end at "solve problem and stop"; here the agent continuously operates without a fixed end, a new paradigm in eval.
- **Open-Ended Objectives:** There's no single correct solution path, just better or worse performance. This introduces a reinforcement learning flavor to benchmarking (maximize reward). That's unique because most benchmarks have clear success criteria; Vending-Bench is more like a game. It evaluates **policy quality** rather than output correctness. This moves AI eval closer to how we evaluate humans (by outcomes and behavior patterns, not one-right-answer).
- **Integration of Realistic Constraints:** The agent has to deal with reality-inspired constraints (limited money, storage, labor costs via Andon fee [233], context window issues requiring external memory [227]). It's unique how it surfaces problems like context window limits (they explicitly mention needing to preserve info beyond context), and temptation to break rules (customers tried to get it to misbehave with harmful requests, it resisted according to blog [240]). That hints they tested for *robustness to adversarial user input* (like employees ordering weird items or asking for illicit instructions). Very few benchmarks include such social/human interaction challenges. So Vending-Bench touches AI alignment issues (will it obey being asked for harmful items? It denied those orders [241] – a good sign for Claude). This alignment testing in a benchmark scenario is unique.
- **Physical World Bridge:** It's also notable that the scenario is something physically grounded (vending machine) and they even took it to real world test with actual fridge and slack users [242] [232]. That means the benchmark was intended to correlate with real world performance, not just simulation – an alignment test in reality. Using a simulation to project real world readiness (for AI managing actual tasks) is a novel angle.
- **Memory and Tool Use by Agent:** The agent having notepad tools, web search, email, Slack communications all within the benchmark is unique [227] [229]. It's like an all-in-one enterprise agent scenario. That tests multi-modality in a sense (it had to respond to Slack messages from users, find suppliers via web search, etc.). It's a far cry from single-model Q&A; it's an ecosystem of capabilities integrated. This is the frontier of agent eval, combining reasoning, planning, and multi-tool execution in one persistent agent.

F. New Benchmark Inspiration:
- **Future Work:** Vending-Bench points toward even more complex multi-agent economies or longer times. They might consider different business scenarios (like an agent running a more complex company or cooperating/competing with another agent). They might also incorporate unpredictable market changes

(shock events to test adaptation). For alignment, introducing ethical dilemmas to see if agent breaks rules under pressure.

- **Identified Gaps:** One gap is generalization: an agent tuned for vending might not transfer to other domains; future benchmarks might include a suite of long-term scenarios (vending, logistics, etc.) to test adaptivity. Another is evaluation difficulty: quantifying why an agent failed (lack of planning vs forgetting vs poor learning) might require new analysis tools (maybe the trajectories can be analyzed by another AI to label errors).

- **Design Patterns:** For our smart contract education, the notion of long-horizon could be applied in a scaled-down way: e.g. simulate multiple blocks or iterative attempts by an agent to drain funds from a contract, requiring the agent to adapt if, say, defenses change. Or an agent that has to manage a DeFi protocol over time (like adjust parameters daily to maintain stability). That's probably too advanced to start with, but we can take inspiration to eventually evaluate not just solving a static vuln, but maintaining security over time (like an agent that monitors a contract and responds to evolving threats). A scenario could be: an AI is the admin of a smart contract system, and across a series of upgrades and attacks, it must keep the system secure and profitable. This is analogous to Vending-Bench's AI CEO role but in DeFi. It'd test strategic thinking (when to patch, when to pause trading, how to set fees to avoid attacks).

- **Memory and Coherence in Security Tasks:** If we have a multi-step exploit scenario, an agent might need to remember what exploit method it already tried and not repeat fruitlessly. Or if doing a lengthy code audit, ensure it doesn't contradict itself in assessments. We can incorporate persistent state or intermediate summaries as Vending's agent had note-taking. Our benchmark could allow an agent to "write a report" to itself as memory while auditing a large codebase to see if that helps catch more issues.

- **Adversarial Interaction:** Vending-Bench had agent dealing with human pranks (tungsten cubes, etc.) [243]. In security, an agent defender could face an active attacker. Future security benchmarks might have a Red-Team vs Blue-Team simulation (two agents, or agent vs. scripted attacker). That would be the security analog of a dynamic environment with adversarial inputs. It's complex but interesting: measure how well the AI defends over a series of exploit attempts.

- **Measurement of stability:** Just as Vending-Bench looks at whether agent decisions remain rational over time (the Claude example underpriced or sold at loss inadvertently [239]), a security agent's consistency could be measured by how reliably it monitors and catches issues over a long log of transactions (maybe if it misses one attack out of many, that's a coherence failure).

- **Ethics and Safety:** We should also consider embedding ethical guidelines in a smart contract agent (don't do anything illegal like draining users even if profit, etc.) and test if it holds to them under tempting circumstances. For example, if an agent finds a vulnerability that could be exploited for profit, does it report or secretly exploit? That's analogous to Vending's "don't misbehave when employees tried to provoke it" [241]. This could measure alignment in the context of security responsibilities.

In summary, Vending-Bench broadens our horizon of what a benchmark can be: not just solving a problem, but operating an autonomous role under constraints for an extended period. While our initial focus might be narrower (like static challenges), it's a vision to keep for future expansions: evaluating AI on continuous security management tasks, requiring sustained vigilance, memory of past attacks, adaptation to new threats – essentially an AI CISO to the AI CEO of Vending-Bench. That could be the ultimate testbed for smart contract security AI once basic capabilities are in place. For now, elements like multi-step scenarios, memory aids for agents, and adversarial inputs can be gradually introduced following Vending-Bench's pioneering lead.

Comparative Matrix of Benchmarks

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
SWE-Bench (Full & Verified)	Real-world **bug fixing** in OSS codebases (multi-file patches) [9]. Variants: Lite (easier subset), Bash-Only (limited to shell edits), Multilingual, Multimodal [34].	Primarily **Python** (12 OSS repos) [13]; Multilingual variant adds C/JS repos [34]; Bash tasks involve shell scripting.	~2,294 issues (Full) [14]; 300 Lite; **500 Verified** (human-checked) [3]; 500 Bash-Only (same tasks as Verified) [18]; 517 Multimodal [244].	**Unit tests pass/fail** on each issue (FAIL_TO_PASS tests must pass & no regressions) [25]. %Resolved = fraction of tasks fully fixed [245]. Verified subset ensures tasks are solvable & tests aligned [16] [17].	**Autonomous coding agent** with repo access (read files, output diff/patch) [10]. Often uses scaffolds to iterate code edits & run tests. Bash-Only: agent constrained to CLI edits only [18].	**Real G issues** (authe [9]; m contex Empha **auton** (agent locate withou and us *suites* f objecti Verifie unique **out un tasks**, benchm fidelity Has a **f varian** only, e broade evalua surface

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
SWE-Bench Pro	**Long-horizon SW dev tasks** – complex issues/ features taking hours-days (multi-step implementation) [44]. Focus on **enterprise-scale codebases** (41 repos, B2B and dev tools) [43]. Agents must handle large code modifications, cross-file changes.	Multiple languages (likely **Python, Java, JS, etc.** since repos span web, enterprise apps). Tasks include backend services, data pipelines [45]. Codebases actively maintained, incl. private startup code [46].	**1,865 problems** total [45]: 11-repo Public set (open), 12-repo Held-out (closed), 18-repo Commercial (proprietary, eval only) [45]. All tasks human-verified solvable with sufficient context given [39].	**Unit/ integration tests** for correctness + possibly hours-long execution tests. **Success** if all tests pass *and* agent achieves target outcome (feature implemented). Metrics: % tasks solved. Also analyzes **failure modes** clusters from agent logs [49]. Emphasis on **contamination-resistance** – results on hidden sets ensure no training overlap [46] [47].	**Autonomous "full-dev" agent** with access to large repo and issue spec. Typically uses advanced scaffolds (planning, tool use for search) due to task complexity. Possibly allowed time for multi-step planning/ coding. No external internet, but broad code understanding needed.	**Enterp grade compl** tasks r readin thousa LOC ar **multi-f patche** Introdu **hidden** (held-o comm preven overfit **Tasks s real pr work** (add, m refacto than is bugs. E **long-ra cohere** (agent keep c across change **profes quality** code m efficien passin [44] [47] baselin models ~23% s reintro headro

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
Terminal-Bench 1.0	**DevOps and CLI tasks** – end-to-end workflows in a Linux terminal [64]. E.g. compile code, configure servers, run scripts, debug errors. 100 tasks across web dev, data processing, automation, infra, security [61]. Each task is a realistic scenario with a defined goal state (file outputs, server running, etc.).	**Shell environment** (Ubuntu container). Tasks involve various languages/tools: Bash, Python, Docker, Linux config, etc. (Whatever one would use in a terminal). Domains: system admin, networking, security (OpenSSL, cracking hashes) [76], ML training, etc.	~100 tasks (v1.0) [61]. Difficulty tagged (easy/med/hard). Community-contributed expansion. Each task is sandboxed in its own container with specific pre-loaded files and dependencies [66] [75].	**Automated execution** of agent commands in Docker sandbox. Success if final state meets verification script criteria (all required files created with correct content, tests passing, etc.) [64] [61]. Metrics: **Success rate (% tasks solved)** [78]. Harness monitors every command and uses hidden checks for correctness (and possibly no unintended changes). Potential tracking of retries, errors encountered. Leaderboard ranks by % solved (OB-1 agent ~59%) [80].	**Interactive CLI agent**: agent issues shell commands, can read outputs, and persist memory of them [65]. Essentially an LLM controlling a terminal (with optional mixture-of-models for specialization as OB-1 did [94]). The agent cycles: read goal, execute commands, see results, adjust. Time/step limits apply. Agents often incorporate a **memory buffer** (store command outputs and plan) to handle multi-step context [65].	**Realis use** – a must u OS con (gcc, n etc.). T **recove** comma agent s differe approa Benchm **open-s comm driven** quickly becam **standa eval fo tasks** l mid-20 Unique requiri **persis execut multi-s reason** one-sh Empha **robust** tasks li YouTub downlo stabilit (site ch leading verifica improv [75].

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
Terminal-Bench 2.0	**Harder, verified CLI tasks** – expanded/ modified v1.0 tasks to increase difficulty and ensure reliability [88]. Likely more complex workflows and new categories. Maintains same task types (DevOps, etc.) but with higher complexity (maybe multi-container or larger data).	Same Linux CLI domain, possibly with broader tool coverage. Emphasis on tasks requiring multi-step verification (multiple services, bigger data sets). Maintains system-administration, coding, security categories but tasks refined to eliminate flakiness [75].	~100 tasks (similar count) but with problematic ones replaced/ fixed [75]. All tasks underwent intense manual & LLM-assisted verification [88] to ensure correctness criteria and difficulty alignment. Released Nov 2025.	Same success metric (% tasks solved) but now using the **Harbor** framework for scalable evaluation [81]. Harbor allows parallel container runs and integrates RL/SFT pipelines [89]. Task quality much improved – few false negatives/ positives due to extended verification (each task thoroughly tested under various conditions) [88]. Possibly new metrics like time-to-solve or number of retries (Harbor can collect more data).	Agent interface unchanged (shell control), but now agents run via **Harbor**, meaning easier integration for cloud eval and training loops [81]. Agents can be "plugged in" and evaluated at scale. Harbor also supports feeding back rollouts for learning (so agent improvement during eval is facilitated) [89].	**Bench evolut** demon *version* raise d agents [69] [86] are **hig quality** comm their re clarity Introdu **Harbo** eval+tr infrast blurrin betwee benchr trainin enviro [89]. C issues (like dy extern depen were a making robust [75]. Es 2.0 sta **frontie aligne** ensuri SOTA a strugg of late

CritPt (CriticalPoint)	**Frontier physics reasoning** tasks – 71 *research-level* physics problems, each decomposed into modular sub-tasks [106]. Challenges cover theoretical physics scenarios requiring multi-step derivations, numerical solutions, or proofs. Emulates entry-level grad research problems (open-ended, not seen in textbooks) [104] [113].	**Physics (nearly all subfields)**: e.g. quantum computing, astrophysics, particle physics, etc. [107]. Requires advanced math (calculus, linear algebra) and domain knowledge. Answers may be **symbolic (formulae)**, **numeric arrays**, or **Python code** (for simulation) [246]. All problems unpublished to avoid training contamination [110].	**71 challenges**, **190 checkpoints** (sub-problems) [106]. Each challenge with private reference solution (except 1 demo) [112]. Dataset curated by ~40 physics experts over 7 months [107], ensuring realism. Multi-dimensional: covers 9 major physics subfields.	**Auto-grading pipeline**: Model answers parsed and compared to ground truth via **symbolic algebra (Sympy)** or numeric validation with tolerances [246]. Each challenge scored correct/incorrect. Main metrics: % of full challenges solved (very low, GPT-5 ~4%) [116] and % of checkpoints solved (somewhat higher). Evaluation is strict but accounts for equivalent expressions and numerical error tolerances [246]. Solutions kept hidden to reuse as test set [112].	Agent is essentially an **LLM (with or without tool use)** producing a *step-by-step reasoning trace* and final answer. Some tasks allow writing a small program to aid (execution as part of answer validation) [246]. Typically evaluated in batch, not interactive. The agent is expected to internally reason (often beyond its capability) and output final results in a standardized format for grading.	**Unpre difficu depth** level fa standa benchr requiri **origin reason** pattern matchi Strict a leakag (hand- and ke answe [110]. **M checkp** provid into wh reason models vs fail Autogr innova use of math e at scal gradin Essent the *bo LLM re* (“critica where from memo true pr solving Also im tests **trustw** – subtl reason make a wrong plausib

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
τ-Bench (Tau-Bench)	**Tool-using dialogue agents** in realistic domains (customer support). Agent must assist a **simulated user** via multi-turn conversation and use **domain-specific APIs** to fulfill requests [121]. E.g. rebooking a flight, checking inventory, following company policy. Emphasizes **interaction** and **policy adherence** [121] [123].	**Domain-specific dialogues**: initial domains Retail (ordering/refunds) and Airline (reservations) [128]. Uses *natural language* conversation + *structured API calls*. Domain APIs like database queries (lookup order, change booking) and **policy docs** in English describing rules [123] [129].	Domains: 2 initially (Airline, Retail), each with multiple scenarios. Each scenario defines a user goal and required end-state (DB updates) [127] [129]. Likely ~50 tasks per domain in initial release (not explicitly stated; paper is 25 pages with presumably dozens of tasks). Simulated users create **~100+ dialogues** total.	**Stateful conversation evaluation**: Agent's actions (tool API calls) modify a simulated database. **Success** if final DB state matches the goal state for the task [129] and *no policies violated* [210]. They introduced **pass^k metric**: reliability over k trials (e.g. pass^8 <25% for GPT-4) [247], to measure consistency in following rules and achieving goals. Conversation logs can be analyzed for rule breaks (automated or via LLM judge).	**Interactive user-agent loop**: Agent sees user messages sequentially (simulated by an LLM) [123] and can output either natural language replies or call a JSON-formatted tool function (with parameters) [121]. Also given a policy file it must reference during dialog. The agent essentially is a fine-tuned LLM with function calling ability plus chain-of-thought to decide on API usage.	**User-i loop** – benchr test ho agent l dynam *and* to multi-t dialogu [123]. **P enforc** key (ag obey d rules li custom rep) [12] Evalua **goal st rather uttera simila** means dialogu succee as outc correct Unique metric highlig consist just on succes Essent gauges agent **helpfu conve assista** actuall comple via too mergin langua action.

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
SecBench	**Cybersecurity QA** – large collection of domain-specific questions to test knowledge and reasoning in security [136] [141]. Formats: **MCQ** (multiple-choice) and **SAQ** (short answer) [141]. Covers diverse security topics: vulns, malware, crypto, network sec, etc., at varying difficulty (factual vs scenario-based) [143] [145].	**Cybersecurity domain**, bilingual (**English and Chinese**) [148]. Questions from subdomains: Security Mgmt, Network Sec, AppSec, Crypto, Cloud Sec, etc. (9 domains) [145]. Multi-level: some questions simple knowledge, others require logic or analysis of scenario [143].	**44,823 MCQs** + **3,087 short-answer Qs** [142] – total ~47.9k. Collected via contest and open sources; each question labeled by subdomain and skill level [143] [145]. Likely largest security QA dataset to date.	**MCQ eval**: exact match against correct choice(s) – yields accuracy. **SAQ eval**: automated via a **grading agent (LLM)** that compares model's free response to reference and scores it [137]. They used LLM to label data and for constructing the grader [137]. Metrics: overall accuracy, and breakdown by category, language (English vs Chinese), and knowledge vs reasoning [141] [143]. 16 SOTA models benchmarked to show current performance [149].	**Static QA agent**: Essentially an LLM answering questions (could be open model or ChatGPT style). No tools, just knowledge. Evaluation allows for multi-choice format or free-form answer which is then graded. Because of the volume, models typically are evaluated on a subset or via few-shot prompting for consistency.	**Multi-dimen bench** tests **k retent logical reason Englis Chines vs free respon** one da [148]. C breadt Qs) wit a singl **Conte source conter** high-q novel c beyond sets [14] autom short-a gradin LLM is innova approa scale e of ope respon Essent functio **MMLU** for sec domai special benchr push m expert knowle

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
SciCode	**Scientific coding challenges** – 80 complex research problems from 16 science domains (math, physics, bio, etc.) requiring writing code to solve or simulate phenomena [157] [158]. Each problem decomposed into subproblems (knowledge recall, reasoning, coding) [160] [249]. Tests if models can act as research assistants: e.g., derive formula then implement algorithm, verify results.	**Python programming** primarily (the coding language for solutions) across science domains. Domains: Physics (optics, quantum, astro, etc.), Math (numerical linear algebra, computational finance), Chem, Bio, Material Sci [163] [164]. Requires domain knowledge + coding. Optional background descriptions provided for each problem [152].	**80 main problems**, **338 subproblems** [160]. Each subproblem has test cases and gold solution. Curated by ~30 scientists – real scripts from workflows, some replicating known published results [167] [170]. Best model (Claude3.5) solved only ~4.6% of main problems [155], indicating difficulty.	**Run code and verify**: Each problem includes unit tests & domain-specific checks (e.g. compare simulation output to theory) [169]. Agent's generated code is executed; passes if all tests (functional and scientific validity) succeed [169]. Metrics: % subproblems solved, % full problems solved [161]. Partial credit via subproblem performance. Also analysis by domain and skill type (models often fail reasoning/ coding steps).	**Code-generation agent**: input is a complex scientific prompt (plus optional hints), output is code (and possibly textual answers for theory parts). Agent may have to produce both an equation and code snippet (the benchmark allows providing an answer in a structured way). No interactive feedback – one-shot per subproblem, though problems are multi-part (could be sequentially prompted).	**Scienc** unique requiri **factua knowl coding reason** simulta [157]. P are **mu** (struct real re tasks) option info [15] difficul curren are no near sa [155]. **S verific** tests ir just fu correct reprod publish results physica (e.g. conser proper Curate scienti conten is high typical Essent SciCod benchr AI in sc R&D, b standa profici

SWE-fficiency	**Performance optimization tasks** – 498 scenarios where agent must modify code to achieve speedup on a given workload without breaking correctness [179] [182]. Each task: given a slow implementation in a real library (NumPy, Pandas, etc.), optimize it to match/exceed expert speedup. Emphasizes **code understanding and efficiency reasoning** (find bottlenecks, improve algorithm).	Primarily **Python (data science/ML libraries)** – tasks from 9 popular repos (numpy, pandas, scipy, scikit-learn, etc.) [250]. Domain: data processing, math kernels, ML model training – code that can be vectorized, use better algorithms, or C extensions. Possibly some C/C++ if present in PRs, but evaluation likely in Python runtime for consistency.	**498 tasks** across 9 repos [181]. All derived from real performance-improving PRs (representing ~9.1× speed patches on average) [251]. Each task has original code + tests + workload input + expected speedup (expert baseline) [182].	**Dual criteria**: (1) **Correctness** – agent's patch must pass all unit tests (no functionality break) [182]; (2) **Speedup** – when running provided workload, agent's solution runtime vs baseline is measured. Task "solved" if speedup ≥ expert's (or a threshold) [180]. Overall metrics: average speedup factor achieved (models ~0.15× of expert, extremely low) [180], and success rate (% tasks where target met). Possibly also track how often agents worsen performance or break tests.	**Autonomous code editor**: similar to SWE-Bench but goal is faster runtime, not just passing tests. Agent might need to use profiling tools implicitly or known performance patterns. Likely one-shot patch submission per task in eval (though development of agent can involve iterative trial via pipeline). Agents input: code and performance target, output: optimized code. Possibly an iterative evaluation allowed in development (like run tests and measure after each change) but final eval is based on final submission.	**Perfor centri** benchr *well AI optimiz* not jus correct [179]. U **world workl** evalua (ensuri improv are me not mi [183]. T require **code compr** (findin ineffici large c contex **longer reason horizo** step refacto Highlig curren *drastic underp* human optimi ~15% o expect gain) [1] Unique combi analysi runtim measu eval. A inhere ensure **leakag** specifi

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
						unlikel trainin Encour buildin that do algorit selecti frontie coding beyon correct

AlgoTune	**Algorithmic code optimization** – 155 classic tasks (sort, AES, SVD, etc.) where agent must write code faster than a reference solution [187]. Essentially tests if AI can discover more efficient implementations of general-purpose algorithms. Involves iterative improvement: agent can attempt solution, get timing feedback, and refine code [191].	**Python** tasks (with possible use of libraries like NumPy for speed). Domains: math (matrix factorization, linear algebra), physics sims (PCA, NMF), compression/ encryption (gzip, AES), sorting/ searching. Functions drawn from across CS, math, physics – each with a baseline. No domain-specific knowledge beyond algorithmics, but some tasks require numerical methods knowledge.	**~155 tasks** (NeurIPS'25 submission) [188]. Each task includes: input generator, reference solver (baseline code), correctness checker (to compare outputs) [196] [200]. Difficulty varies from trivial loop optimizations to complex algorithm improvements. All tasks are self-contained functions with defined I/O.	**Quantitative speedup**: Agent's best code vs baseline runtime on test inputs. The **AlgoTune score = harmonic mean of speedups** across tasks [202] (so scoring rewards consistent improvements). Also track per-task speedup for each model (leaderboard shows e.g. GPT-5 got 1.67×, meaning on average 67% faster than baseline) [252]. Code must remain correct (checked by `is_solution` function on sample outputs) [200]. The evaluation allows agent to **iterate with timing feedback** (simulate an engineer optimizing in a loop) [191] – effectively measuring the end-performance after a fixed number of	**Autonomous "AlgoTuner" agent**: an LLM (or ensemble) that can propose code, get runtime (from environment), then adjust code. Provided with baseline code and likely some hints. It operates in a closed loop, like an RL agent with reward = -runtime. Implementation uses an automated conversation between model and environment to refine solution [204]. Agent may call specific tools (like `timeit`, etc.) as allowed by environment.	**Iterati optimi loop** – benchr an age to self- its outp feedba *First ins an eval model's and-err of the measur* Focus **perfor impro** encour unders of algo not jus genera correct [190]. **H mean** ensure is negl models broad, narrow compe Results models do sur optimi (use nu unroll **failed new al** [193], highlig curren limitati algorit creativ AlgoTu effecti

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
				iterations or when agent stops improving.		stradd and reinfor learnir evalua provid transp publish agent optimi traject for ana

SRE-Bench	**Site Reliability Engineering tasks** – agent must handle operations scenarios: incident response (diagnose & mitigate outages), infrastructure changes (scale services, update configs), observability (analyze logs/ metrics), reliability improvements (add redundancy, alerts) [208] . Essentially tests an AI's ability to perform on-call engineer duties with modern cloud infrastructure (esp. Kubernetes) [208] .	**DevOps / Cloud environment** (Kubernetes clusters, Linux servers, CI/CD systems). Domain knowledge: Kubernetes (pods, deployments, etc.), networking, system monitoring, troubleshooting. Likely uses mix of CLI (kubectl), config files (YAML), and potentially chat interface for incident narrative. Policies: SRE runbooks and best practices textual guidelines.	Not fully public. AgentKube's repo updated Dec 2025 suggests tasks exist internally. Rootly's variant: 4 test categories ~1200 samples each [213] (maybe Incident Triage, Root Cause Analysis Q&A, Remediation steps, Postmortem tasks). SRE-Bench tasks likely scenario-based with initial state (system in fault) and target state (issue resolved) plus hidden evaluation points (did agent follow policy?).	**Goal-state and process evaluation**: Outcome metric – e.g. service uptime restored (monitored via health-check) within constraints. Also **qualitative**: did agent identify correct root cause? (Could be Q&A graded by LLM or check if agent's actions addressed the actual issue). **Policy compliance** check: e.g. if company policy says "don't restart DB without backup", agent must not violate – can be auto-detected by scanning actions or judged by GPT for procedure adherence [210] . Possibly uses **LLM-as-judge** for text explanations (Rootly used OpenBench to unify eval) [218] . A reliability metric (like pass^k) might	**Interactive multi-modal agent**: Observes metrics, logs (text streams), may get alerts or user pages (text). Can execute commands (via an API to cluster), apply config changes (edit YAML), and send communications (e.g. status update). Likely integrates with a tool environment (like an agent with kubectl and monitoring API access). The agent converses as needed (for explanation) and acts on the system. Possibly uses an LLM with a tool plugin architecture to decide on commands vs. chat.	**Opera focuse safety** benchr *AI in infrastr ops*. It' **time a step**: a to gath inform make d under uncert execut in a ru system for em **safety constr** (must f best pr and no additio outage of eval **situati aware** long-te decisio under (incide have c issues) introdu **multi- human intera** differe user m co-wor system feedba requiri to resp approp (e.g., e

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
				measure if agent consistently solves incidents over multiple variations [247].		human policy s In sum Bench evalua pure c output **proce perfor** comple enviro bridgi Safety reliabil domai

Vending-Bench	**Long-term autonomous agent coherence** – agent runs a simple business (vending machine) over many time steps (e.g. simulate months of operation) [224] [226]. Must handle inventory, pricing, finances to maximize profit, and interact with customers or suppliers. Tests planning, memory, and adaptability in a prolonged, **open-ended task** with economic objectives.	**Simulated business environment**: Inventory of products, prices, sales events. Agent actions: purchasing stock (via an API or simulated email), setting prices, communicating with customers (text). In Anthropic's physical demo, also web search and Slack (but core benchmark likely focuses on the simulation aspects) [228] [229]. Language: English for any communications (e.g. responding to custom orders), structured actions for transactions.	Original Vending-Bench (1.0): single vending machine scenario, one owner agent. Vending-Bench 2 extends horizon or complexity (maybe more products or longer time). It's essentially 1 scenario but requiring thousands of agent decisions. Leaderboards show models' **average profit or stability** over runs [225]. Possibly multiple runs per model to gauge consistency (due to stochastic policies).	**Objective metrics**: e.g. **Total profit** earned by end of simulation [232]. Also track if agent **went bankrupt** (balance <0) [231]. For coherence, possibly measure "no catastrophic errors": e.g. did agent maintain stock, avoid contradictory pricing, handle customer requests logically (Anthropic noted specific failures/ hallucinations) [237]. They might measure number of rule violations (e.g. agent must not exceed storage capacity – count if it ordered too much). Customer satisfaction or requests fulfilled could be secondary metric. LLM Stats implies a single score (likely profit or success rate).	**Complex autonomous agent**: Uses an LLM with extended memory (could write notes to an external file to recall over hundreds of turns) [227]. Has tools: web search for suppliers, email API to place orders, Slack interface for customer interaction [228] [229]. The agent essentially runs continuously, making decisions each "day". It's an **embedded agent** with perception (sales data) and actions (order, price set, comms). Multi-modal in that sense (structured actions + natural dialogue).	**Longe horizo bench** tests **s and go alignm very lo traject** (hundr decisio [226]. F **cohere** the ag remain consist its obje and ins (Claud system and ne stick to goal) [2] Also in **realist econo challe** even s aspect (custo messa agent) blendi planni NLP. Th inclusi **safety** (ensuri doesn' disallo things indeed resiste reques unique double sandb observ

Benchmark	Focus / Task Type	Languages / Domain	Dataset Size	Eval Method	Agent Interface	Uniqu
						misbeh drift ov runs. E Vendin pionee evalua **as an auton entity** than a solver, shift to holistic assess (capab alignm open w albeit s

(Sources: citations inline above per each benchmark.)

Evolution Timeline of Coding Agent Benchmarks

- **2023: Early Coding Benchmarks Emerge** – *SWE-Bench* (Princeton, 2023) debuts as a large-scale real-world coding eval [1], focusing on GitHub bug fixes. It establishes using actual codebases and tests, but models only achieve ~20% then [37]. This year sees primarily static code generation tests (HumanEval, MBPP) extended to realistic scenarios via SWE-Bench.

- **Early 2024: Specialization & Interaction Starts** – *τ-Bench (Tau-Bench)* (June 2024) introduces **interactive dialogue + tool** use in evaluation [121], heralding benchmarks that go beyond one-shot Q&A. *SciCode* (July 2024) appears, pushing into scientific domains – highlighting the need for domain-expert benchmarks as LLMs saturate simpler tasks [157]. Also, *SWE-Bench Lite* and *Multilingual* splits launch (Mar–Oct 2024) to broaden and ease evaluation for various use cases [30]. Benchmarks begin to branch into specialized areas: science (SciCode), dialogue (Tau-Bench), security QA (preliminary SecBench work via contest late 2024) [254].

- **Late 2024: Reliability and Validation** – Concerns about eval quality lead to human-validated sets. OpenAI's *SWE-Bench Verified* (Aug 2024) addresses solvability issues by human-curating 500 tasks [29], improving benchmark reliability and revealing higher true performance (GPT-4's score jumps on Verified) [35]. Meanwhile, massive specialized QA sets emerge: *SecBench* (draft Dec 2024, published Jan 2025) compiles ~50k cybersecurity questions [142], signaling maturity in generating large expert benchmarks via community (contest + curation). By end of 2024, there's a push for **scale (SecBench)** and **accuracy of evaluation** (Verified), as models rapidly improve and require better discrimination.

- **Early 2025: Cambrian Explosion of Agent Benchmarks** – A burst of innovative benchmarks: *Vending-Bench* (arXiv Feb 2025) pioneers long-term agent coherence tests [255]. *CritPt* (Sep 2025, with work likely starting earlier 2025) takes on frontier reasoning in physics [97]. *SRE-Bench* development is underway (Rootly's OpenBench integration by Aug 2025) [209], bringing ops into eval. *AlgoTune* (arXiv Jul 2025) appears as the first iterative optimization benchmark for code [190]. Concurrently, *Terminal-Bench 1.0* launches (May 2025) and quickly becomes the go-to for agent evals in industry labs [69] – within months, many top labs report results on it. By late 2025, we see **cross-domain expansion** (SRE, DevOps, physics, algorithms) and **interactive/iterative formats** (Terminal-Bench, Tau-Bench, AlgoTune, Vending-Bench) flourishing. This period is dubbed a "Cambrian explosion" of benchmarks [256], as each explores a new dimension of agent capability.

- **Late 2025: Second-Generation Benchmarks & Scaling** – Benchmarks iterate and scale up. *Terminal-Bench 2.0* (Nov 2025) arrives with harder tasks and an integrated training framework (Harbor) [91], showing benchmarks evolving alongside model capabilities to remain challenging [69]. *SWE-Bench Pro* (arXiv Nov 2025) dramatically expands task complexity (41 repos, multi-hour tasks) [43] to "reset" the scoreboard as models were nearing saturation on original tasks [52]. *SecBench* is released fully (Jan 2025 revision 3) and perhaps integrated into evaluation leaderboards, highlighting multi-lingual evaluation need as models like GPT-4 go global [148]. *SRE-Bench* by Dec 2025 is likely in beta or internal use (AgentKube repo), and Rootly's variant is accessible via OpenBench, indicating initial results and interest from industry in ops agents [217] [218]. *Vending-Bench 2* (late 2025) extends the horizon of long-term eval, maybe testing new facets like multi-agent or more days [223].

Trends: Over 2023–2025, benchmarks shifted from **static, one-step code generation** to **dynamic, multi-step agent tasks**. We see a clear trend toward *realism and interactivity*: starting with realistic code (SWE-Bench), then adding multi-turn interaction (Tau-Bench, Terminal-Bench), and finally long-term autonomy and adaptability (Vending-Bench). Another trend is **specialization**: as base LLMs reach high performance on generic tasks, benchmarks target specific domains (security, scientific computing, SRE) to probe advanced expertise. **Scale and complexity** increased: more tasks (SecBench's tens of thousands QA) and more complex tasks (SWE-Pro's enterprise problems). There's also a drive for *benchmark integrity*: Verified subsets, private test splits, and careful curation (CritPt, SecBench) to ensure benchmarks genuinely reflect model progress, not data leakage or evaluation artifacts. By end of 2025, benchmarks are not just tests but *development platforms* (e.g., Harbor with Terminal-Bench 2.0 enabling training, OpenBench aggregating many benchmarks [219]), indicating a merging of evaluation with continuous improvement frameworks.

Synthesis for a Smart Contract Security Benchmark (YudaiV4)

Designing a benchmark for **smart contract security education** can draw on all these insights to create a rich, realistic, and effective evaluation suite. Key considerations and inspired directions:

1. Realistic Vulnerability Scenarios with Test-Driven Evaluation:
Like SWE-Bench, use **real smart contract vulnerabilities** from historical exploits and GitHub fixes. For each, provide: the vulnerable contract code, a description of the bug or symptom (analogous to an issue report), and a test suite (or Foundry/Hardhat test) that fails initially (exploit succeeds) and should pass after the fix [257] [25]. The agent's task is to **patch the contract** to stop the exploit without breaking other functionality – success measured by all tests passing (exploit test turns green, regression tests remain green). This *FAIL-to-PASS* test approach [11] ensures objective eval. We can incorporate **PASS-to-PASS** tests to ensure no new issues are introduced [25]. For example, a reentrancy challenge: before fix, a Foundry test

withdraws funds repeatedly (fails test by draining funds); after agent's fix (using reentrancy guard or checks-effects-interactions pattern), the drain test should fail (meaning exploit prevented) and normal withdrawal tests still pass. This leverages **test-driven verification** – a proven scheme in SWE-Bench to evaluate code changes [25].

2. Multi-Step and Interactive Challenges:
Incorporate **interactive exploit scenarios** inspired by Terminal-Bench and Tau-Bench. For instance, an **exploit simulation**: the agent (as an attacker) is given on-chain access (via a Foundry script environment) to a deployed vulnerable contract and must perform a sequence of transactions to steal funds. This tests the agent's ability to combine calls creatively – essentially a Terminal-Bench style task but in Ethereum context (using `cast` or `forge script` CLI). Conversely, an **agent defender scenario**: given a running contract and live transactions (simulated via scripts), can the agent detect an ongoing attack and trigger emergency stop? We could simulate a conversation ("User: our protocol is being drained!") and the agent uses tools (block explorer API, Hardhat console) to diagnose and act, similar to SRE-Bench's incident response style [208]. This tests not only coding but real-time analysis and use of dev tools – reminiscent of Tau-Bench's tool+dialog interplay [121]. An example: an agent gets an alert of unusual withdrawals; it queries contract state (tool use), identifies an integer overflow, and calls a `pause()` function (action) to mitigate. We evaluate if funds were saved (outcome) and if agent followed protocol (did it verify the issue, not just blindly pause – policy compliance).

3. Education and Future Work Integration:
The benchmark should not just score models but guide learning, reflecting the *educational* aspect (YudaiV4 context). Following Terminal-Bench 2.0's Harbor idea [89], we can integrate a feedback loop: after evaluation, provide the agent (or student) with analysis of failure modes. For instance, if an agent's exploit attempt failed because it missed a step, supply a hint or allow iterative tries (like AlgoTune's loop with timing feedback [191]). This way the benchmark doubles as a *training gym* for interactive agents. We can have **tiers of difficulty**: basic tasks (solved by simple pattern-matching, e.g., known reentrancy fix) up to advanced (complex logic vulnerabilities requiring reasoning akin to CritPt-style synthesis). The benchmark results can inform curriculum: e.g., if many agents fail on integer math bugs, emphasize that in training.

4. Multi-Dimensional Coverage:
Mirroring SecBench's multi-dimensional design [143] [148], include **multiple task formats** to cover all facets of security: - **QA format** for conceptual knowledge (e.g., "What does 'delegatecall' do and why is it dangerous?" – MCQ or short answer, potentially drawn from SecBench if focusing on smart contract topics). - **Code review format**: present a short contract snippet and ask the agent to identify vulnerabilities (text output). Evaluate with an LLM grader or a set of expected findings – like SecBench's SAQ grading [137]. - **Exploit coding**: ask for a script that exploits a given contract (like writing a Foundry test that drains a token). Evaluate by running the script – if it successfully steals funds (and only if intended), it's a success. This leverages the test harness approach but for offensive tasks. - **Patch coding**: the classic fix-the-bug tasks as discussed, evaluated by tests. - **Gas optimization or best practices**: as an enrichment (inspired by AlgoTune and SWE-fficiency), maybe one task category is to refactor a function to reduce gas cost without changing behavior. Use Hardhat's gas reporter to verify improvement (must be careful to keep security constant). This appeals to devs and reinforces that sometimes security involves efficiency (preventing gas griefing attacks, etc.).

By having **multiple sections** (Knowledge Quiz, Vulnerability Identification, Exploitation, Defense/Patching, Optimization), we ensure a holistic evaluation of a smart contract security agent. Each section can have its

own scoring, and an overall matrix (like above) can be presented to highlight strengths/weaknesses of a model or student.

5. Chronological or Long-Form Scenarios:
Following Vending-Bench's lead on long-term evaluation [226], we could imagine a capstone scenario where an agent is tasked with **securing a protocol over time**. For example, simulate a DeFi protocol over a series of upgrades or attacks: at step1 an issue arises, agent fixes (or fails); step2 a new vulnerability in an upgrade, agent acts; and so on. This tests memory (did it recall past vulnerabilities?), adaptability, and persistence – analogous to long-horizon coherence. This is advanced, but even a simplified 3-step scenario could be valuable. For education, this could be set as an **interactive CTF**: multiple flags to capture sequentially. The agent must carry state from one challenge to the next – akin to CritPt's modular checkpoints but where performance on early parts affects later ones.

6. Foundry/Hardhat Integration and Tooling:
Adopt modern Ethereum dev frameworks (Foundry preferred given its popularity in security community) so that tasks are executed in a realistic dev environment. Each task's repository can include a Foundry project with tests (similar to how Terminal-Bench tasks are containerized with required files). The agent, if an automated one, could interface via Forge CLI or Hardhat API. For manual educational use, students get the repo and run tests to check their solution, learning through immediate feedback – the *benchmark can function as an automated grader for assignments*. This dual-use (benchmark for AI, problem set for humans) amplifies its educational utility.

7. Community and Maintenance:
Inspired by how SWE-Bench became community-driven with variants and how SecBench held a contest, we can plan to **crowdsource new tasks** (Yudai participants could submit interesting bugs they find, or write CTF-like challenges). Over time, this keeps the benchmark up-to-date with emerging vulnerability types (e.g., if a new class of Solidity bug is discovered, add a task for it). A *Verified* subset could be maintained to ensure quality – experts review submissions similarly to OpenAI's SWE-Verified collaboration [258] [16].

8. Emphasize Safe and Ethical Behavior:
Following Tau-Bench and Vending-Bench's concern with rule-following [123] [240], include *policies* the agent must heed. For example, a policy might state "If a vulnerability is found, do not exploit it for profit – instead, report it." Then have a scenario where exploiting yields higher "reward" vs. reporting (like a tempting private key exposure). Evaluate whether the agent chooses the ethical action (report) or the forbidden one (exploit). This introduces an **alignment** dimension – crucial if the benchmark trains agents to assist in security (they should behave as white-hats). It's also educationally important to instill ethics in learners. Non-compliance could be flagged by evaluation (like Tau-Bench does with policy docs [121]).

9. Comparative Metrics and Feedback:
Like our matrix above, for each model or student, provide a breakdown: e.g., Knowledge 80%, Vuln Identification 60% (missed reentrancy in Task X), Exploitation 90%, Patching 70% (tests failed for overflow fix), Gas Optim 50%. This matrix helps pinpoint areas for improvement – fulfilling the educational goal by directing focus (maybe the student needs to study integer overflows more, etc.). Over time, track progress (maybe run the benchmark at start and end of a course to quantify learning gains).

10. Evolution and Future:
Initially YudaiV4 can start with a moderate number of tasks (perhaps ~20-30 across categories) focusing on

core vulnerabilities (reentrancy, over/underflow, access control, etc.). As models and students get better, follow Terminal-Bench's model of raising difficulty or adding new categories: e.g., cross-contract attack scenarios, flash loan exploitation tasks, advanced cryptographic bugs. Always validate tasks for solvability (no "impossible" CTF puzzles without sufficient hints – to avoid frustration and to mimic Verified's philosophy of feasible tasks [29]). Also consider incorporating multi-language if relevant (Solidity primarily, but maybe one or two tasks in Vyper or Cairo for contrast in future).

In summary, the **Smart Contract Security Benchmark** will combine the best elements of these benchmarks: - **Realistic, test-verified coding tasks** from SWE-Bench, - **Iterative, interactive problem solving** from Terminal-Bench and Tau-Bench, - **Comprehensive domain coverage** from SecBench (but focused on smart contracts), - **High difficulty, multi-step reasoning** from CritPt and SciCode for novel exploit scenarios, - **Performance and efficiency considerations** inspired by SWE-fficiency and AlgoTune (gas optimization tasks), - **Long-term, adaptive and ethical agent evaluation** from SRE-Bench and Vending-Bench (incident response and continuous monitoring aspects).

This fusion will create a benchmark (and learning platform) that not only evaluates current capabilities of AI (and students) in smart contract security, but also *drives research and learning* by highlighting weaknesses (perhaps models will struggle with logical reasoning in exploitation – akin to how CritPt showed physics gaps – indicating need for better chain-of-thought training in code). By structuring it with clear feedback and an eye on realistic use, YudaiV4's benchmark can accelerate development of robust, **"autonomous security agents"** who can audit, exploit (for good), and defend smart contracts in the real world.

---

[1] [32] [33] [34] [256] 8 benchmarks shaping the next generation of AI agents | AI Native Dev
https://ainativedev.io/news/8-benchmarks-shaping-the-next-generation-of-ai-agents

[2] [3] [8] [9] [10] [11] [12] [13] [16] [17] [20] [21] [22] [23] [24] [25] [28] [29] [35] [36] [37] [257] [258] Introducing SWE-bench Verified | OpenAI
https://openai.com/index/introducing-swe-bench-verified/

[4] SWE-bench: Can Language Models Resolve Real-world ... - GitHub
https://github.com/SWE-bench/SWE-bench

[5] princeton-nlp/SWE-bench_Verified · Datasets at Hugging Face
https://huggingface.co/datasets/princeton-nlp/SWE-bench_Verified

[6] [7] [14] [15] [18] [19] [26] [27] [30] [31] [83] [244] [245] SWE-bench Leaderboards
https://www.swebench.com/

[38] [39] [43] [44] [45] [46] [47] [48] [49] [251] [2509.16941] SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?
https://arxiv.org/abs/2509.16941

[40] scaleapi/SWE-bench_Pro-os: SWE-Bench Pro - GitHub
https://github.com/scaleapi/SWE-bench_Pro-os

[41] SWE-Bench Pro: Raising the Bar for Agentic Coding - Scale AI
https://scale.com/blog/swe-bench-pro

42 Introducing SWE-Bench Pro: A New Benchmark for Coding Agents
https://www.linkedin.com/posts/xiang-deng-377288119_github-scaleapiswe-benchpro-os-swe-bench-activity-7375744754174771200-G5u5

50 51 120 121 129 134 247 [2406.12045] $τ$-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains
https://arxiv.org/abs/2406.12045

52 A more accurate benchmark for coding agents - SWE-Bench Pro
https://www.reddit.com/r/GithubCopilot/comments/1odgwbp/a_more_accurate_benchmark_for_coding_agents/

53 E. Kelly Buchanan
http://ekbuchanan.com/

54 [PDF] BENCHMARKING AI AGENTS IN SOFTWARE DEVOPS CYCLE
https://openreview.net/pdf?id=bP48r4dt7Z

55 56 62 66 67 68 70 71 73 74 76 77 96 Terminal-Bench
https://www.tbench.ai/

57 How Much Do Large Language Model Cheat on Evaluation ... - arXiv
https://arxiv.org/html/2507.19219v1

58 ia03/terminal-bench · Datasets at Hugging Face
https://huggingface.co/datasets/ia03/terminal-bench

59 Leaderboard - Terminal-Bench
https://www.tbench.ai/leaderboard

60 61 63 64 65 78 80 84 94 OpenBlock | OB-1 Coding Agent
https://www.openblocklabs.com/blog/terminal-bench-1

69 75 81 85 86 87 88 89 90 91 92 93 95 Introducing Terminal-Bench 2.0 and Harbor
https://www.tbench.ai/news/announcement-2-0

72 On-Device Environment Setup via Online Reinforcement Learning
https://arxiv.org/html/2509.25455v1

79 terminal-bench@1.0 Leaderboard
https://www.tbench.ai/leaderboard/terminal-bench/1.0

82 Terminal-Bench - Vals AI
https://www.vals.ai/benchmarks/terminal-bench

97 98 99 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 246 Probing the Critical Point (CritPt) of AI Reasoning: a Frontier Physics Research Benchmark
https://arxiv.org/html/2509.26574v1

100 AGI Advance: Weekly AI & AGI Insights (Oct 7, 2025) - Turing
https://www.turing.com/blog/agi-advance-newsletter-33

122 123 126 127 128 130 131 132 -Bench: Benchmarking AI agents for the real-world | Sierra
https://sierra.ai/blog/benchmarking-ai-agents

124 sierra-research/tau-bench: Code and Data for Tau-Bench - GitHub
https://github.com/sierra-research/tau-bench

125 TAU-bench: Multi-Domain Benchmark Suite - Emergent Mind
https://www.emergentmind.com/topics/tau-bench

133 Automated Hallucination Correction for AI Agents: A Case Study on ...
https://cleanlab.ai/blog/tau-bench/

135 Benchmarking AI Agents: Stop Trusting Headline Scores, Start ...
https://medium.com/alan/benchmarking-ai-agents-stop-trusting-headline-scores-start-measuring-trade-offs-0fdae3a418cf

136 137 142 146 147 149 254 [2412.20787] SecBench: A Comprehensive Multi-Dimensional Benchmarking Dataset for LLMs in Cybersecurity
https://arxiv.org/abs/2412.20787

138 139 141 143 144 145 148 248 GitHub - secbench-git/SecBench: SecBench: A Comprehensive Multi-Dimensional Benchmarking Dataset for LLMs in Cybersecurity
https://github.com/secbench-git/SecBench

140 Benchmarking LLMs in Security: A Comparative Review of Five ...
https://medium.com/@tkadeethum/benchmarking-llms-in-security-a-comparative-review-of-five-open-source-mcq-datasets-ea36517c9db0

150 [PDF] SECBENCH.JS: An Executable Security Benchmark Suite for Server ...
https://software-lab.org/publications/icse2023_SecBenchJS.pdf

151 152 157 160 161 171 [2407.13168] SciCode: A Research Coding Benchmark Curated by Scientists
https://arxiv.org/abs/2407.13168

153 154 155 158 159 162 163 164 165 166 167 168 169 170 249 SciCode - SciCode Benchmark
https://scicode-bench.github.io/

156 SciCode Bench - CBORG AI Portal
https://cborg.lbl.gov/bench_scicode/

172 173 174 175 179 180 181 182 183 250 SWE-fficiency: Can Language Models Optimize Real-World Repositories on Real Workloads? | OpenReview
https://openreview.net/forum?id=J1lgJyP4Tm

176 177 178 Introducing SWE-Efficiency Benchmark: Can LLMs Optimize Code? | Vijay Janapa Reddi posted on the topic | LinkedIn
https://www.linkedin.com/posts/vijay-janapa-reddi-63a6a173_swe-fficiency-activity-7394461000051093504-06jr

184 194 (PDF) AlgoTune: Can Language Models Speed Up General ...
https://www.researchgate.net/publication/393923135_AlgoTune_Can_Language_Models_Speed_Up_General-Purpose_Numerical_Programs

185 190 Towards Open Evolutionary Agents - Hugging Face
https://huggingface.co/blog/driaforall/towards-open-evolutionary-agents

186 187 191 192 193 196 197 198 199 200 201 202 203 204 205 206 207 252 AlgoTune
https://algotune.io/

188 AlgoTune/README.md at main · oripress/AlgoTune · GitHub
https://github.com/oripress/AlgoTune/blob/main/README.md

189 How Fast Can AI Actually Code? Inside AlgoTune's $1 Gauntlet
https://medium.com/@abvcreative/how-fast-can-ai-actually-code-inside-algotunes-1-gauntlet-4e7dea5f834f

195 AlgoTune: A new benchmark that tests language models' ability to ...
https://www.reddit.com/r/LocalLLaMA/comments/1lpwj5j/algotune_a_new_benchmark_that_tests_language/

208 Agentkube · GitHub
https://github.com/agentkube

209 210 213 214 215 216 217 218 219 Rootly | Rootly joins Groq OpenBench with an SRE-focused benchmark
https://rootly.com/blog/rootly-joins-groq-openbench-with-an-sre-focused-benchmark

211 [State of Code Evals] After SWE-bench, Code Clash & SOTA Coding ...
https://podcasts.apple.com/mt/podcast/state-of-code-evals-after-swe-bench-code-clash-sota/id1674008350?i=1000743352061

212 last9/awesome-sre-agents - GitHub
https://github.com/last9/awesome-sre-agents

220 NEW Benchmark for Longterm AI Stability - Agentic Vending ...
https://www.youtube.com/watch?v=Vo231lY0pwU

221 222 226 227 228 229 231 232 233 234 235 236 237 238 239 240 241 242 243 253 Project Vend: Can Claude run a small shop? (And why does that matter?) \ Anthropic
https://www.anthropic.com/research/project-vend-1

223 Vending-Bench 2 - Andon Labs
https://andonlabs.com/evals/vending-bench-2

224 230 Vending-Bench: Testing long-term coherence in agents | Andon Labs
https://andonlabs.com/evals/vending-bench

225 Vending-Bench: A Benchmark for Long-Term Coherence ... - alphaXiv
https://www.alphaxiv.org/overview/2502.15840v1

255 Vending-Bench: A Benchmark for Long-Term Coherence of ... - arXiv
https://arxiv.org/abs/2502.15840