SourceWorldBench: Can Coding Models Predict Test and Profiler Outcomes?

Anonymous authors
Paper under double-blind review

Abstract

Coding benchmarks increasingly focus on realistic repository-level software engineering tasks approached via agentic harnesses, where the models are allowed to explore repository contents and execute code. Supporting code execution for agentic harnesses requires sophisticated infrastructure, so *code world modeling*, *i.e.*, simulating the program's execution, has recently gained traction. Performance optimization, in particular, is a common software engineering task that often requires high-load or long-running workloads and could substantially benefit from simulation. However, while simulating the logical code state during execution and end-to-end performance optimization are widely studied, it remains unknown whether models can reliably, say, simulate a profiler. Thus, we introduce SourceWorldBench, a repository-level benchmark comprising six tasks from various aspects of modeling software performance: test outcome prediction, execution time and traced memory estimation, and runtime and memory hotspot localization. We derive instances from an end-to-end performance optimization benchmark SWE-fficiency, augment them with agent-generated patches, and execute test suites and performance workloads to construct the benchmark labels. We evaluate six proprietary and open-weight models under four context collection baselines, ranging from no retrieval to privileged execution-informed context. For all six tasks, the best configurations outperform simple task-specific heuristics marking the possibility of progress, while many of the considered configurations do not, highlighting a substantial room for improvement. For example, for the task of test outcome prediction, Claude Opus 5 and GPT-5.6 Sol score 45.1 and 41.5 Macro F1 respectively, while a random coin flip scores 49.5. These findings establish SourceWorldBench as a challenging testbed for studying and improving optimization-adjacent code world modeling capabilities in isolation.

1 Introduction

Widely used benchmarks measure AI programming abilities through test-based assessments of functional correctness. HumanEval (Chen et al., 2021) and SWE-bench (Jimenez et al., 2024) evaluate whether a generated function or patch passes the benchmark's tests. Passing these tests, however, does not guarantee the code's practical usability: a patch can still run too slowly, allocate more memory than the machine provides, or break behavior outside the tests' coverage. Using an LLM-based verifier, Yu et al. (2026) find that fewer than half of resolved issues fully satisfy design constraints mined from project pull requests. Recent benchmarks, including GSO (Shetty et al., 2025), SWE-Perf (He et al., 2025), SWE-fficiency (Ma et al., 2025), and FormulaCode (Sehgal et al., 2026), evaluate repository-level optimization under correctness and performance constraints.

End-to-end optimization outcomes combine candidate generation, repository exploration, tool use, and stopping decisions. They do not directly reveal what models can predict about execution from repository code before receiving execution feedback. SWE-fficiency's failure analysis identifies difficulties in preserving expected behavior, reasoning about execution across functions, and localizing optimization opportunities (Ma et al., 2025).

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

Execution and profiling feedback can improve optimization performance. Deng et al. (2026) introduce PerfAgent, a workflow combining profiler output, iterative refinement, and selective test feedback. With GPT-5.1, PerfAgent achieves hack-adjusted expert-matching patch rates of 39.2% on GSO and 74% on SWE-fficiency-Lite, compared with 19.6% and 26%, respectively, for OpenHands. Incorporating this feedback into the agentic loop, however, has preconditions: the workload must run at the required scale on available hardware, within the agent's latency and cost budget, with timing stable enough to attribute the difference to the patch. The ability to reason about the performance of candidate implementations can inform which changes to investigate and what to execute. We therefore evaluate this reasoning ability separately from patch generation and execution-based validation.

Code World Model (CWM) is trained on execution traces to simulate Python execution (Copet et al., 2025). Following Bogomolov & Zharov (2026), we use predictions of program behavior as behavioral probes of execution-related *Software World Model* (SWM) capabilities. We introduce SourceWorldBench, a repository-scale benchmark of six prediction tasks covering three capabilities relevant to software optimization. *Behavior preservation* tasks predict selected test outcomes, providing a proxy for behavioral changes observable through those tests. *Resource estimation* tasks predict workload execution time and peak increase in traced Python memory. *Bottleneck localization* tasks rank executed project-level functions by exclusive execution time or cumulative traced-memory growth. These rankings identify where measured costs concentrate and can inform further optimization analysis.

We construct the benchmark from 498 SWE-fficiency source tasks across nine Python repositories and their associated agent-generated candidate patches, and obtain prediction targets by executing repository tests and profiling performance workloads. At inference time, models must predict the targets without executing repository code, running tests or performance workloads, or invoking profilers. To estimate the performance across the model sizes and families we report lineup of OpenAI models (Luna, Terra, and Sol) together with singular models from other providers (Opus, GLM, and Qwen models). To examine how predictive performance depends on access to relevant repository code, we compare task inputs alone, source retrieval, read-only agent exploration, and an execution-trace oracle. The oracle supplies source code selected using prior execution, making it a privileged reference condition. Our results show that prediction remains challenging even with oracle context, while the effect of source context varies across tasks.

Contributions. Our contributions are as follows:

- SourceWorldBench, a repository-scale benchmark constructed from realistic agent-generated optimization patches, with execution-based labels for six independently evaluated prediction tasks.
- SourceWorldBench Lite, a fixed stratified subsets of 200 instances per task for lower-cost evaluation, reducing the number of evaluated instances by a factor of approximately 3–4 relative to the full datasets, accompanied by analyses of distributional alignment and sampling variability.
- An evaluation of frontier LLMs under four context conditions, examining how access to repository code affects their predictions and where limitations remain.

2 Related Work

Performance benchmarks. Each of the optimization benchmarks Shetty et al. (2025); He et al. (2025); Ma et al. (2025); Sehgal et al. (2026) reports one end-to-end quantity per task, the speedup of a patch that still passes the tests. That quantity aggregates over the agent scaffold, the search budget, the profiler access granted to the agent, and the timing behavior of the host machine. Chen et al. (2026) replayed the reference patches on four cloud machine types and found that they satisfied the original validity criteria under every cross-machine replay for portion of the tasks (39 of 102 GSO, 11 of 140 SWE-Perf, and 411 of 498 SWE-fficiency tasks); the official rankings of eight submissions common to GSO and SWE-fficiency disagreed on 9 of 28 pairwise comparisons. It also scores the patch rather

than the reasoning that produced it: a speedup does not record whether the agent identified the bottleneck that the expert identified, or arrived at the same file for an unrelated reason. Progress on these benchmarks is therefore difficult to attribute to a specific model capability.

Resource behavior across domains. Sarıkayak et al. (2026) instrument 102 repository-level optimization tasks with runtime, peak memory, and time-weighted memory usage, and report that model runtime gains are negligible and memory optimizations nearly non-existent, against expert patches that reduce peak memory by 171.3× in aggregate and improve it on 65.7% of tasks. For GPU code, models match PyTorch on fewer than 20% of the 250 KernelBench workloads (Ouyang et al., 2025); under a TF32 baseline and a hidden test suite that blocks value-specific shortcuts, the strongest model falls from 1.43× to 0.88× geometric-mean speedup and raises peak GPU memory on 28% of the kernels it writes (Zhang et al., 2026). For low-level systems code, where the crashes that fuzzing surfaces are predominantly memory violations, Mathai et al. (2025) repair 54.5% of Linux kernel crashes without localization and 65% when given the correct file, indicating that localization accounts for a significant part of the remaining difficulty. For resource ceilings, Ferreira et al. (2026) compare classical optimizers against an agent that tunes hyperparameters by editing training code, and attribute the advantage of CMA-ES and TPE to avoiding out-of-memory failures rather than to search diversity. Two further results suggest that the gap is not specific to optimization tasks: frontier models answer higher-order computer-architecture questions at 34% to 73% accuracy on QuArch (Prakash et al., 2025), and cues placed outside the task specification shift which algorithm family a model emits by up to 100 percentage points across 46,535 trials, while every candidate still passes the same tests (Narula et al., 2026). In the last case the resource profile of the generated code is determined by surface context rather than by any statement about cost. While stressing the importance of LLM's ability to reason about the resource consumption of code, these works do not test this reasoning directly.

Execution reasoning benchmarks. CRUXEval operates on the function level, and asks models to predict the input or the output of 800 short Python functions (Gu et al., 2024). Existing repository-level code reasoning benchmarks (Liu et al., 2025; Wang et al., 2026; Li et al., 2026; Taherkhani et al., 2026; Roy et al., 2026) primarily focus on evaluating program behavior and omit resource estimation entirely. In particular, the represented tasks include predicting input, output, and lower-level control flow properties (e.g., which branch within a conditional statement gets executed) for test functions inside Python repositories. Models perform substantially worse at repository scale than on isolated functions (Liu et al., 2025), and reasoning jointly about multiple tests is harder than reasoning about individual tests (Taherkhani et al., 2026). These benchmarks focus on behavior prediction in a single program state, and our behavior preservation tasks instead ask whether a patch changes test outcomes.

Resource reasoning benchmarks. BigO(Bench) asks models to infer and to target asymptotic time and space complexity classes over 3,105 problems (Chambon et al., 2025). EffiBench (Huang et al., 2024) and Mercury (Du et al., 2024) measure the runtime of code that a model generates for self-contained problems. RepoEffi-Bench (Yang et al., 2026) and Spotlights (Barzelay et al., 2026) focus on identifying the same location as edited by ground truth performance optimization patch. Phaedrus (Chatterjee et al., 2026) predicts function-level runtime hotspots, but in a specialized setting comprising 13 SPEC CPU2017 programs and compiler-derived artifacts for profile-guided optimization. Like these works, we want to check the model's understanding of the resource consumption during the runtime; however, unlike these works, we take a direct approach of estimating specific values grounded in the execution dynamics.

The closest existing result is code-to-metric regression: Akhauri et al. (2026) train a 300M-parameter model on a frozen LLM encoder to predict the memory footprint of Python and C++ programs and the latency of Triton kernels from source text, reaching Spearman rank correlations above 0.9 on competitive-programming submissions. That result establishes that the signal is recoverable from the source, and it is the reason we treat the agent failures above as a modeling gap rather than a limit on the information available. It does not, however, measure whether general-purpose models used as coding agents recover that signal, and it operates on self-contained submissions and single kernels rather than on repositories.

In contrast, SourceWorldBench evaluates workload-specific hotspot prediction in real-world repositories and covers both time and memory hotspots.

3 Data Collection

To augment SWE-fficiency with execution-derived supervision, we construct the benchmark in three stages. First, we assemble a core dataset by combining the original SWE-fficiency tasks with their corresponding agent trajectories. Second, we execute the test suite for each repository state and record the test outcomes. Third, we measure execution time and peak traced Python memory using the original and amplified performance workloads. In this section, we describe details of this procedure. In Section 3.1 we describe the data collection, in Section 3.2 we detail the collection of the observables that are used to construct the specific tasks, and in Section 3.3 we describe the sampling procedure we use to select the final datapoints.

3.1 Data

We use the SWE-fficiency dataset[1] as the source of performance optimization tasks. We call each SWE-fficiency entry a *source task*. A source task contains two repository states collected from GitHub: a *base state* before a performance optimization pull request and a *fixed state* after the pull request. It also contains a *container* that defines the execution environment, a *test command* that serves as a prefix for test execution, and human-curated performance *workloads*. These workloads are small programs designed to measure a particular performance improvement introduced by the pull request. The original workloads improvements in execution time.

For each source task, we attach the corresponding agentic trajectories provided by SWE-fficiency[2] and we use *patch* to refer to the resulting code diff. Applying a patch to the base state produces a *prediction state*. We use *repository state* as a common term for base, fixed, and prediction states. Each repository state forms a *data point* in the core dataset, and all data points derived from the same source task form a *group*.

To construct the core dataset, for each source task, we take its base state, fixed state, and all prediction states. To ensure data quality, we filter out the prediction states where the model failed during generation, produced an empty patch, or converged to the patch identical to already added. After the filtering, a typical group consists on median of 22 prediction states. Table 20 in Appendix L provides a breakdown of source tasks and states by repository.

3.2 Observables

To support the tasks of the benchmark, we collect three types of observables—test statuses, time profiles, and memory profiles. This section details the collection of these observables.

Test Status.

For each repository state in the core dataset, we execute the complete test suite and collect the result of each test. Because testing protocols differ across software projects, we normalize their outputs and retain three statuses for downstream sampling and analysis: `PASSED`, `FAILED`, and `ERROR`. We call them *in-scope statuses*. We assign `PASSED` when a test runs to completion and the observed result matches the expected result. We assign `FAILED` when a test runs to completion, but the observed result does not match the expected result. We assign `ERROR` when a test starts running but cannot complete because of a runtime error. We combine `FAILED` and `ERROR` statuses as `NOT PASSED` status.

We map all remaining statuses to `OTHER` and filter them out. In particular, such a status as `SKIPPED` often reflects the machine configuration, operating system, or availability of an optional feature rather than the algorithmic behavior of the code under analysis.

[1] `https://huggingface.co/datasets/swefficiency/swefficiency`

[2] `https://github.com/swefficiency/swefficiency/tree/swefficiency_base/predictions/converted`

Time Profiling. The original SWE-fficiency workloads generally use the `timeit` library to execute the measured code several times and report the mean of the recorded runtimes. Since we want to measure a more fine-grained reasoning, instead of using the original workload as is, we wrap it in a `pytest` test with a trivial assertion and profile the entire test with our custom `cProfile`-based tool. The tool provides the total execution time of the wrapped workload and the exclusive execution time of every project-level internal call. We provide a validation of our profiling methodology against SWE-fficiency in Appendix H.

Memory Profiling. Since the original SWE-fficiency workloads are designed to reveal differences in execution time, for memory profiling we introduce *amplified workloads* that exercise the same repository code with larger inputs. We generate these workloads by prompting an LLM to increase the input sizes, while preserving the workload operations. We wrap each amplified workload in a `pytest` test and use Python's standard-library `tracemalloc` module to collect memory measurements. This includes workload-level *peak*, the largest increase of traced memory, and per-internal-call *cumulative*, the sum of positive net increases in traced memory between entry and return across all calls. We provide the full comparison and workload-generation prompts in Appendix N.

3.3 Sampling

To reduce the number of points, while keeping the ones that provide important signal, we introduce *group-based sampling*. For each task-specific subset, we compute the relevant statistics over the entire group and use them to sample prediction states from that group, thus keeping the groups intact.

For test status tasks, in every group, we identify tests that have the `PASSED` status on the base state and one of the in-scope statuses on the fixed state. For all such tests we calculate the probability that the status changes after applying a patch (hereinafter *flip score*), and sample them proportional to this probability. After selecting a subset of tests, we sample two prediction states that have an in-scope status for every selected test.

In the profiler tasks group-based sampling, we use the same sampled prediction states for the time and memory data. For every group, we sample two prediction states based on their execution times: one is closer to the base state time, another—to the fixed state time. We define execution-time proximity and the resulting sampling probabilities in Appendix I.3. The dataset obtained through this process constitutes SourceWorldBench dataset.

To reduce evaluation cost further, we use fixed stratified subsets of 200 instances per task. We stratify by repository and binary label for Behavior Preservation tasks, and by repository and binned counts of executed functions for Resource Estimation and Bottleneck Localization. Metrics computed on these subsets are empirically indistinguishable from those computed on the full datasets. Subset construction and distributional validation are described in Appendix E. This subset constitutes SourceWorldBench Lite.

4 Task Design

In this section, we define six benchmark tasks using the data collected in the previous section. The tasks evaluate three capabilities relevant to performance-oriented software engineering: preserving observed behavior, estimating workload-level resource use, and localizing project-level bottlenecks. For each task, we specify the input, prediction target, and primary evaluation metric. Each stored dataset record contains a reference to an execution container and a patch represented as a string. We report dataset statistics in Appendix D, and detailed metrics analysis in Appendix A.

4.1 Behavior Preservation Tasks

Performance optimizations are useful only when they preserve intended program behavior. SWE-fficiency (Ma et al., 2025) reports that agent-generated patches often leave repositories with non-passing tests, motivating an evaluation of whether models can anticipate such outcomes. A naïve formulation would require predicting the outcomes of every test in a

repository state. Due to the sheer amount of tests (54,000 tests on average are affected by the golden patch), we split Behavior Preservation Tasks into two variants: single test and 100 tests with base state test statuses as an anchor.

Single Test Status. Each instance contains an execution container with the codebase in a prediction state and one selected test. The model predicts either `PASSED` or `NOT PASSED`. We evaluate predictions using *Macro F1* score, a mean of F1 scores for both labels

Test Status Anchored. Each instance contains an execution container with the codebase in its base state, a patch, and a selected subset of 100 tests. Given the information that all tests pass in the base state, the model predicts either `PASSED` or `NOT PASSED` status of each test after applying the patch. We use the same *Macro F1* score as in Single Test Status that is averaged across the subset of 100 tests.

4.2 Resource Estimation Tasks

Resource estimation evaluates whether models can anticipate the execution costs of a repository state. Such estimates are useful for an optimization agent to compare candidate implementations before selecting which approach to pursue. We define complementary tasks to predict execution time and memory use because an optimization may improve one resource while degrading the other.

Time Prediction. Each instance contains an execution container with the code base in a prediction state and one workload test obtained by wrapping the original workload. The model predicts the total elapsed time of this single workload test. We evaluate predictions using debiased mean absolute error (MAE) in log space.

We first estimate the systematic bias between the predictions and ground-truth values as $b = \text{median}_i \left(\log_{10} y_{\text{pred},i} - \log_{10} y_{\text{actual},i}\right)$ and then compute the MAE after subtracting b from the log-scale predictions. Debiased MAE measures the typical pointwise prediction error while still being insensitive to the additive shifts produced by machine-wide rescaling (Appendix C).

Memory Prediction. Each instance contains an execution container with the code base in a prediction state and one workload test obtained by wrapping an amplified workload. The model predicts the peak increase in traced memory during this test, measured relative to the level at test start. We evaluate predictions using the same debiased MAE in log space. as for Time Prediction.

4.3 Bottleneck Localization Tasks

SWE-fficiency shows that differences between LLM-generated and expert interventions persist below file granularity: LLM-generated and expert patches modify the same files in more than 55% of cases, more than 68% of expert-attributed performance gains occur in functions that the corresponding model does not edit. This finding motivates evaluating function-level localization directly. To identify hotspots, we aggregate the call-level profiling measurements by project-level function and order function with respect to this aggregate.

In both cases we use *HitRate@k* metric, which measures the fraction of reference top-k functions recovered by the predicted top-k. We use a tie-tolerant definition: when truncation at k separates functions with identical measured cost, any function tied at the boundary is considered correct.

Time Hotspots. Each instance contains an execution container with the code base in a prediction state, one workload test obtained by wrapping the original workload, and a list of project-level functions executed during the test. The model ranks the candidate functions, returns the top-20, and is evaluated with *HitRate@20* against the ground truth ranking by aggregated exclusive execution time. We chose $k = 20$, since top-20 functions capture 94% of profiled time on average (Appendix D).

Memory Hotspots. Each instance contains an execution container with the code base in a prediction state, one workload test obtained by wrapping an amplified workload, and a list

of project-level functions executed during the test. The model ranks the candidate functions, returns the top-5 and is evaluated with *HitRate@5* against the ground truth ranking by the cumulative memory consumption. We chose $k = 5$, since top-5 functions capture 97.4% of cumulative attributed memory on average (Appendix D).

5 Baselines

Using the six prediction tasks defined in Section 4, we study the *world-model reasoning* capabilities of Large Language Models (LLMs). However, an incorrect prediction may reflect missing execution-relevant information as well as limitations in reasoning, which motivates two main axes along which we vary the baselines: models and context collection scaffoldings. In this section, we describe these two axes, and additionally introduce a naïve heuristic to set the reference for the scores provided by LLMs. For further details, refer to Appendix B.

Models. We evaluate six models spanning capability tiers, model families, and deployment scales: GPT-5.6 Luna (OpenAI, 2026a), GPT-5.6 Terra (OpenAI, 2026c), GPT-5.6 Sol (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026a), GLM-5.3 (Z.ai, 2026a), and Qwen3.8-27B (Qwen Team, 2026). We use high reasoning effort for all models. For GLM-5.3, we switch to medium when the model fails to complete the task within the token budget. Further details and empirical results motivating the choice are presented in Appendix F.

Context collection baselines. All context collection baselines receive the same task specification and task-specific inputs. *No-Retrieval* adds no repository context beyond these inputs. *BM25* retrieves function-level chunks from the repository using BM25 queries derived from the task inputs. *Read-Only Agents* use the Codex and mini-SWE-agent harnesses as specified in Table 1 with access to a read-only repository sandbox without the ability to execute code, and must submit a valid prediction within 40 turns. Finally, as a privileged baseline, the *Execution Trace Oracle* supplies the source code of repository functions executed during the run of the target test or workload. For Test Status Anchored, the union of traced functions is capped to fit the context budget (Appendix B.2.4).

Heuristics. We include simple task-specific non-LLM heuristics as reference points: a random predictor for Single Test Status, an always-`PASSED` predictor for Test Status Anchored, a predefined constant for Time and Memory Prediction (5 s and 10^9 bytes, respectively), and call-count-based rankings for Hotspots Prediction.

6 Results and Discussion

Main experimental results are presented in Table 1. Scores are obtained on SourceWorldBench Lite. We estimate metric and model-rank uncertainty using a paired, stratified bootstrap over benchmark instances and report 95% bootstrap intervals; pairwise significance claims use Holm correction within each prespecified comparison family (Appendix E). Comprehensive per-task metrics are presented in Appendix A.

SourceWorldBench is challenging but tractable. On every task, the best configuration outperforms the naïve heuristics, indicating that the task inputs contain predictive signal beyond simple priors. However, even with frontier models and high reasoning effort, no configuration, including privileged baselines, nears the saturation. And, as colors in Table 1 indicate, most configurations fall below or are on par with heuristics. Even for the tasks with strict improvements, the error is far from being saturated. For example, Time Prediction errors are well above measurement noise: repeated runs on the reference machine have a mean coefficient of variation of 0.054 ($\approx$0.023 in $\log_{10}$ units), whereas the lowest observed $\mathrm{MAE}^{\mathrm{deb}}_{\log_{10}}$ is 0.221 (Appendix C).

Oracle context provides a significant performance boost. While the model improvements provide the most significant boost in multiple tasks, the Oracle context collection strategy still shows a significant impact on the performance. In the most significant case, relative to BM25, Oracle context significantly improves Single Test Status for four of six models (by up to 35.7 macro-F_1 for Opus), Time Prediction for four, and Memory Hotspots for five. It's worth noting that Oracle's effect is not uniformly positive: on Time Hotspots,

Table 1: Main baseline results on SourceWorldBench Lite. **Test Status**: macro F1 for Single and Anchored. **Time / Memory Pred.**: Debiased mean absolute $\log_{10}$ error (debiased $\log_{10}$ MAE; unitless; lower is better). **Hotspots**: HitRate@20 for time, HitRate@5 for memory. Cell shading marks strict improvement over Heuristics baseline.

Groups (→) / Models (↓) / Tasks (→)		Test Status		Time		Memory	
		Single	*Anchored*	*Pred.*	*Hotspots*	*Pred.*	*Hotspots*
	Heuristics	49.5	48.3	0.628	0.456	0.506	0.254
No-Retrieval	GPT-5.6 Luna	35.1	55.0	0.452	0.243	0.713	0.371
	GPT-5.6 Terra	34.7	58.8	0.427	0.402	0.609	0.434
	GPT-5.6 Sol	35.8	65.4	0.418	0.51	0.335	0.544
	Claude Opus 5	48.4	63.7	0.45	0.456	0.455	0.456
	Qwen3.8-27b	54.3	58.2	0.541	0.278	1.148	0.203
	GLM-5.3	47.8	49.9	0.563	0.281	1.082	0.211
BM25	GPT-5.6 Luna	39.2	63.4	0.453	0.379	0.701	0.374
	GPT-5.6 Terra	40.2	61.5	0.407	0.42	0.675	0.439
	GPT-5.6 Sol	41.5	66.5	0.359	0.533	0.631	0.547
	Claude Opus 5	45.1	63.5	0.372	0.481	0.537	0.494
	Qwen3.8-27b	42.0	60.4	0.554	0.276	1.442	0.21
	GLM-5.3	46.0	52.0	0.561	0.298	0.851	0.214
Read-Only Agents	`Codex` GPT-5.6 Luna	53.5	51.8	0.515	0.296	1.307	0.215
	`Codex` GPT-5.6 Sol	34.0	55.7	0.477	0.361	0.845	0.427
	`mini-SWE` GPT-5.6 Luna	41.2	43.2	0.378	0.152	0.706	0.138
	`mini-SWE` GPT-5.6 Sol	48.5	63.5	0.313	0.214	0.618	0.475
	`mini-SWE` GLM-5.3	40.7	56.1	0.474	0.432	0.978	0.377
Execution Trace Oracle	GPT-5.6 Luna	58.3	57.2	0.296	0.283	0.67	0.521
	GPT-5.6 Terra	67.0	62.3	0.279	0.508	0.693	0.574
	GPT-5.6 Sol	73.5	65.8	0.221	0.67	0.631	0.683
	Claude Opus 5	80.8	65.1	0.295	0.451	0.608	0.429
	Qwen3.8-27b	52.9	59.2	0.497	0.271	1.536	0.34
	GLM-5.3	60.0	49.4	0.47	0.301	1.213	0.327

Oracle is significantly better than BM25 for Sol and Terra but significantly worse for Luna. And on Memory Prediction, it is significantly worse than No-Retrieval for Sol and Qwen. We treat this as a signal that better context collection is needed both through agentic model training and better retrieval tools.

Read-only agents do not provide a significant boost over BM25. The expectation is that agent-based pipelines will provide superior quality over classical retrieval pipelines. However, in our experiments, compared with the same model using BM25, mini-SWE-agent ends up significantly better in 3 of 18 comparisons and significantly worse in 5; the rest are not significant. Results also depend on the harness: GPT-5.6 Sol does better with mini-SWE-agent than with Codex on five of six tasks, whereas GPT-5.6 Luna does better with Codex on four. This reinforces the need for training to gather context and resolve the proposed tasks.

SourceWorldBench measures various model capabilities. Holding the context collection baseline fixed, the six SourceWorldBench tasks produce different model rankings. The median Kendall's τ between the rankings of two tasks is 0.6, and even with Oracle context, where ranks are most certain, it ranges from 0.33 to 1.0 (Appendix E). From this, we conclude that the tasks measure related but distinct capabilities, and a single aggregate score would obscure task-specific model performance.

Overlap with executed code is an incomplete proxy for context usefulness. We measure BM25 precision as the fraction of retrieved files or functions that are present in the Oracle context. File-level precision is positively associated with downstream quality for Test Status Anchored and Memory Prediction for most evaluated models, whereas associations for the remaining tasks and at the function level are mixed (Appendix J). Overall, these results suggest that overlap with executed code alone does not capture all properties that make the retrieved context useful to a model.

Small models struggle to benefit from extended context. The models on the smaller side of the spectrum, especially Qwen3.8-27B, rarely improve from the additional context. Qwen often has worse results with the additional context compared to the No-Retrieval baseline.

The Memory Prediction task proves itself significantly complicated. For the Memory Prediction task, models with the BM25 and Oracle baselines are more prone to underestimation than with No-Retrieval (Appendix M). A qualitative analysis of GPT-5.6 Sol provides a possible explanation: with additional context, the model more often mentions in its rationale excluding code objects that are, in fact, considered for the ground-truth memory footprint. We find that rationales with corresponding lexical patterns are associated with larger prediction errors (Appendix K).

7 Limitations and Future Work

The scope of SourceWorldBench is shaped by its reliance on SWE-fficiency, which contains optimization PRs from a selected set of Python repositories. However, performance optimization itself is relevant to a much broader range of software engineering projects, motivating an extension beyond the repositories considered here. We plan to broaden the benchmark's coverage across programming languages and application domains to better represent this diversity.

Our current approach to collecting performance workloads relies on workloads constructed for individual optimization tasks, while automatically generating suitable workloads remains challenging for LLMs (Ma et al., 2025). The future work should address this limitation by introducing better ways to collect and generate reliable workloads.

The workloads introduce a potential source of information leakage: because they are written with knowledge of the ground-truth patch, providing them to the evaluated model may reveal information about the targeted optimization. We plan to focus future data collection on repositories that already maintain performance benchmark suites.

A common limitation of benchmarks that use open-source data is data contamination, and SourceWorldBench is no exception. However, our benchmark design allows us to mitigate this problem in the future through periodic updates. If the benchmark data becomes contaminated, we can rerun SWE-fficiency to get new patches and construct the next version of the benchmark.

The observables for SourceWorldBench are derived from program execution, and particular values depend on hardware type and state, operating system processes, etc. We discuss this variability in Appendix C and mitigate its effects with our choice of metrics (Appendix A). This analysis remains preliminary and requires further development. Our baseline results, reported under multiple metrics, provide a starting point for a more systematic study of metric selection in code-specific tasks.

8 Conclusion

In this paper, we introduced SourceWorldBench, a novel benchmark that measures a model's ability to reason about program dynamics. It addresses three particular problems that were highlighted in the previous literature: behavior preservation, resource estimation, and bottleneck localization. We collect a dataset for these tasks and establish the performance baselines by varying both model families, model sizes, and context gathering strategies.

From our experiments, we conclude that while increasing model capacity and context relevance improve results, there are no clear winners among models, and no one-size-fits-all approach to context collection. Future research is needed to understand how to achieve targeted improvements in such capabilities in models and how to collect context for these tasks.

AI use statement

In this work, we used generative AI tools for the following tasks: design or provide feedback on research methodology or experiments, implement methods, support qualitative and thematic data analysis, generate synthetic data sets.

We have not used generative AI tools for the following tasks: help develop theoretical models or conceptual frameworks, propose or refine hypotheses, clean and reformat dataset, interpret results, and the following are not applicable to this work: formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation.

Additionally, we used generative AI tools for the following tasks: create or modify scientific figures or images, draft parts of a research paper, brainstorming, edit a research paper to improve readability, identify relevant literature.

We have reviewed all AI-assisted work.

- **Design or provide feedback on research methodology or experiments.** We used LLMs as a reviewer for research methodology designed by us. We applied feedback only after discussion in our group.
- **Implement methods.** We used AI-based coding assistants to help us develop the code base. Every change was organized as a PR to our repository, and each edit was reviewed by at least two people: an author of the PR and a reviewer.
- **Support qualitative and thematic data analysis.** We used LLMs to conduct qualitative data analyses for different intermediate data sets we produced while developing the benchmark. We checked all the useful insights manually and discarded others.
- **Draft parts of a research paper.** We used LLMs to draft parts of our paper, and every section was reviewed and edited manually.
- **Identify relevant literature.** We used AI search tools to help us find relevant work, every suggested paper was reviewed and discussed with the team.
- **Create or modify scientific figures or images.** We used AI-based coding assistants to refine the code for plotting our figures.
- **Brainstorming.** We used LLMs to prioritize some initial research ideas.
- **Edit a research paper to improve readability.** We used LLMs and AI-based writing assistants to refine the text of the paper. Final editing decisions and style checks were performed by the authors.
- **Generate synthetic data sets.** We used LLMs to amplify workloads for memory tasks (Appendix N). All the produced workloads were verified by one of the authors, and by running them in execution containers. Additionally, we reused SWE-fficiency patches that were generated by AI agents.

Reproducibility statement

Artifacts are released under an anonymous account. The Hugging Face collection at `https://huggingface.co/collections/anonymous-research-730875/sourceworldbench` contains the six task datasets defined in Section 4, the raw predictions of every evaluated model, and a public leaderboard. The evaluation harness, which computes all task metrics and implements the heuristic baselines, is available at `https://github.com/anonymous-research-730875/sourceworldbench-eval-harness`. All Docker images used to collect observables are available at `https://github.com/users/anonymous-research-730875/packages/container/package/containers`. The construction of the core dataset is described in Section 3.1 and Appendix L, and the construction of the SourceWorldBench Lite subsets is explained in Appendix E. Models, inference settings, context budgets, and the context collection baselines are specified in Section 5 and Appendix B, the full task prompts are listed in Appendix O, and the construction of the downsampled evaluation subsets is detailed in Appendix E.

References

Yash Akhauri, Xingyou Song, Arissa Wongpanich, Bryan Lewandowski, and Mohamed S. Abdelfattah. Regression language models for code. *arXiv preprint arXiv:2509.26476*, 2026. Accepted at ICML 2026.

Anthropic. Claude opus 5. `https://platform.claude.com/docs/en/models/opus-5/overview`, 2026a. Accessed 22 September 2026.

Anthropic. Messages api reference. `https://platform.claude.com/docs/en/api/messages/create`, 2026b. Accessed 22 September 2026.

Udi Barzelay, Ophir Azulai, Idan Friedman, Inbar Shapira, Foad Abo Dahood, Yevgeny Burshtein, Orit Prince, Michael Soloveitchik, Oshri Naparstek, Roi Pony, et al. Spotlights: Discovering improvement opportunities in software repositories. *arXiv preprint arXiv:2609.20446*, 2026.

Egor Bogomolov and Yaroslav Zharov. Towards evaluation of implicit software world models in coding llms. *arXiv preprint arXiv:2606.27406*, 2026.

Pierre Chambon, Baptiste Rozière, Benoît Sagot, and Gabriel Synnaeve. BigO(Bench): Can LLMs generate code with controlled time and space complexity? *arXiv preprint arXiv:2503.15242*, 2025.

Bodhisatwa Chatterjee, Neeraj Jadhav, and Santosh Pande. Phaedrus: Predicting dynamic application behavior with lightweight generative models and llms. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1):707–736, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are performance-optimization benchmarks reliably measuring coding agents? *arXiv preprint arXiv:2607.01211*, 2026.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Ryan Deng, Yuanzhe Liu, Bastian Lipka, Yao Ma, Xuhao Chen, Tim Kaler, and Jatin Ganhotra. PerfAgent: Profiler-guided iterative refinement for repository-level code optimization. *arXiv preprint arXiv:2607.19653*, 2026.

Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *arXiv preprint arXiv:2402.07844*, 2024.

Fabio Ferreira, Lucca Wobbe, Arjun Krishnakumar, Frank Hutter, and Arber Zela. Can LLMs beat classical hyperparameter optimization algorithms? a study on `autoresearch`. *arXiv preprint arXiv:2603.24647*, 2026.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Yin Zheng, Zejian Yuan, and Zejun Ma. SWE-Perf: Can language models optimize code performance on real-world repositories? *arXiv preprint arXiv:2507.12415*, 2025. Accepted at ICML 2026.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pp. 65–70, 1979.

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. EffiBench: Benchmarking the efficiency of automatically generated code. *arXiv preprint arXiv:2402.02037*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Jia Li, Yuxin Su, and Michael R Lyu. From laboratory to real-world applications: Benchmarking agentic code reasoning at the repository level. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8845–8869, 2026.

Changshu Liu, Alireza Ghazanfari, Yang Chen, and Reyhaneh Jabbarvand. Evaluating code reasoning abilities of large language models under real-world settings. *arXiv preprint arXiv:2512.14917*, 2025.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. SWE-fficiency: Can language models optimize real-world repositories on real workloads? *arXiv preprint arXiv:2511.06090*, 2025. Accepted at ICML 2026.

Alex Mathai, Chenxi Huang, Suwei Ma, Jihwan Kim, Hailie Mitchell, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. kAgent: An execution-guided crash resolution agent for the Linux kernel. *arXiv preprint arXiv:2504.20412*, 2025. Deep Learning for Code workshop at ICML 2026.

Akanksha Narula, Mofasshara Binte Rafique, and Laurent Bindschaedler. The invisible lottery: How subtle cues steer algorithm choice in LLM code generation. *arXiv preprint arXiv:2606.04057*, 2026.

OpenAI. GPT-5.6 Luna. `https://developers.openai.com/api/docs/models/gpt-5.6-luna`, 2026a. Accessed 22 September 2026.

OpenAI. GPT-5.6 Sol. `https://developers.openai.com/api/docs/models/gpt-5.6-sol`, 2026b. Accessed 22 September 2026.

OpenAI. GPT-5.6 Terra. `https://developers.openai.com/api/docs/models/gpt-5.6-terra`, 2026c. Accessed 22 September 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Shvetank Prakash, Andrew Cheng, Mark Mazumder, Arya Tschand, Varun Gohil, Jeffrey Ma, Jason Yik, Zishen Wan, et al. QuArch: A benchmark for evaluating LLM reasoning in computer architecture. *arXiv preprint arXiv:2510.22087*, 2025.

Qwen Team. Qwen3.8-27B. `https://huggingface.co/Qwen/Qwen3.8-27B`, 2026. Accessed 22 September 2026.

Monoshi Kumar Roy, Simin Chen, Benjamin Steenhoek, Jinjun Peng, Gail Kaiser, Baishakhi Ray, and Wei Le. Codesense: A real-world benchmark and dataset for code semantic reasoning. In *International Conference on Learning Representations*, volume 2026, pp. 13735–13768, 2026.

Ezgi Sarıkayak, Wenchao Gu, Hesham Ghonim, and Chunyang Chen. Evaluating LLMs on real-world software performance optimization. *arXiv preprint arXiv:2606.25530*, 2026.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. FormulaCode: Evaluating agentic optimization on large codebases. *arXiv preprint arXiv:2603.16011*, 2026. Accepted at ICML 2026.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-Agents. *arXiv preprint arXiv:2505.23671*, 2025.

Hamed Taherkhani, Mohammad Abdollahi, Melika Sepidband, Hridya Dhulipala, Tien N. Nguyen, and Hadi Hemmati. Can llms reason about runtime behavior? a repository-level dynamic benchmark, 2026. URL https://arxiv.org/abs/2609.28449.

Yanlin Wang, Suiquan Wang, Yanli Wang, Bowen Zhang, Daya Guo, Jiachi Chen, and Zibin Zheng. Reporeasoner: Evaluating repository-level code reasoning ability of long-context language models. *Proceedings of the ACM on Software Engineering*, 3(FSE):2790–2812, 2026.

Haowen Yang, Yun Peng, and Zishuo Ding. Effiholmes: Differential profiling-guided repository level time inefficiency fix localization. *arXiv preprint arXiv:2608.03558*, 2026.

Kai Yu, Zhenhao Zhou, Junhao Zeng, Ying Wang, Xueying Du, Zhiqiang Yuan, Junwei Liu, Ziyu Zhou, Yujia Wang, Chong Wang, and Xin Peng. Does pass rate tell the whole story? evaluating design constraint compliance in LLM-based issue resolution. *arXiv preprint arXiv:2604.05955*, 2026.

Z.ai. GLM-5.3. https://docs.bigmodel.cn/cn/guide/models/text/glm-5.3, 2026a. Accessed 22 September 2026.

Z.ai. Migrating to GLM-5.3. https://docs.bigmodel.cn/cn/guide/start/migrate-to-glm-new, 2026b. Accessed 22 September 2026.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Xiangjun Fan, Julian Reed, Azalia Mirhoseini, and Will Su. KernelBench-Verified: Do LLM-generated kernels actually beat PyTorch? *arXiv preprint arXiv:2607.16241*, 2026.

A Detailed Metrics Results

This appendix shows the evaluation run reported in the main text with a broader set of evaluation metrics. Table 2 reports detailed test-status results, including class-specific scores that distinguish performance on passing and non-passing tests. Tables 3 and 4 report detailed resource-estimation and hotspot-localization results, including log-scale prediction error, regression intercepts and slopes, accuracy within multiplicative tolerances, and complementary measures of hotspot selection, ordering, and captured cost. These complementary measures help distinguish aspects of predictive performance that a single aggregate score cannot capture. Both tables use the same stratified cohorts and evaluation conditions as the main results.

A.1 Test Status Metrics

All classification metrics are computed from a single pooled confusion matrix over the evaluation set, rather than by averaging per-instance scores. Precision, recall, and F_1 for class c are

$$P_c = \frac{TP_c}{TP_c + FP_c}, \qquad R_c = \frac{TP_c}{TP_c + FN_c}, \qquad F_{1,c} = \frac{2P_cR_c}{P_c + R_c},$$

where a class that has never been predicted receives precision and F_1 of zero.

We report the following metrics:

- **Macro-F_1.** The unweighted mean of the F_1 scores for the `PASSED` and `NOT PASSED` classes.
- **Macro precision and macro recall.** The unweighted means of the per-class precision and recall.
- **Balanced accuracy.** The average of the recalls for the two classes. For binary classification, this is identical to macro recall.

- **Accuracy.** The fraction of correctly classified scored units,
$$\frac{TP + TN}{TP + TN + FP + FN}.$$
This metric is reported for completeness.
- **Matthews correlation coefficient (MCC).** Computed with `NOT PASSED` as the positive class:
$$\frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$
- **Per-class F_1.** The individual F_1 scores for the `PASSED` and `NOT PASSED` classes reported individually.

Aggregation. For Single Test Status, the scored units are repository states, each containing one test. For Test Status Anchored, the scored units are repository-state/test pairs. We additionally report per-instance macro-F_1 for the anchored setting, which first computes macro-F_1 independently for each repository state and then averages across states. It is reported for completeness: single-class repository states inflate the across-state average, since correctly predicting the sole class present yields a macro-F_1 of 1 without evaluating discrimination between the two classes.

A.2 Time and Memory Prediction Metrics

Let $t_i > 0$ denote the measured quantity for instance i, wall-clock seconds for time prediction and peak bytes for memory prediction, and let $\widehat{t}_i > 0$ denote the corresponding prediction. We define the signed log-ratio residual $d_i = \log_{10} \frac{\widehat{t}_i}{t_i}$. Thus, $d_i = 0$ corresponds to an exact prediction, $|d_i| = 1$ corresponds to an error of one order of magnitude, and errors by the same multiplicative factor in opposite directions have identical magnitude. All metrics are computed in log space because the targets span several orders of magnitude. Instances with non-positive targets or predictions are excluded because the log-ratio is undefined.

- **Raw** $\mathrm{MAE}_{\log_{10}}$**.** The mean absolute log-ratio error,
$$\mathrm{MAE}_{\log_{10}} = \frac{1}{n} \sum_{i=1}^{n} |d_i|.$$
A value of v corresponds to a geometric mean multiplicative error factor of 10^v. This is the zero-shot metric that measures absolute prediction error and captures both systematic multiplicative bias and instance-level prediction scatter.
- **Debiased** $\mathrm{MAE}_{\log_{10}}$**.** The error remaining after removing a single global multiplicative offset from all predictions,
$$\mathrm{MAE}^{\mathrm{deb}}_{\log_{10}} = \min_{c \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^{n} |d_i - c| = \frac{1}{n} \sum_{i=1}^{n} |d_i - \mathrm{median}(\boldsymbol{d})|.$$
The equality follows because the sample median minimizes the mean absolute deviation. We report the fitted shift $c = \mathrm{median}(\boldsymbol{d})$. This metric removes only uniform bias while preserving the relative variation produced by the model.
- **Log–log regression slope and intercept.** Following Bogomolov & Zharov (2026), we fit the affine calibration
$$\log_{10} \widehat{t}_i = \alpha + \beta \log_{10} t_i + \epsilon_i$$
by ordinary least squares, and report calibration parameters α and β. Perfect agreement corresponds to $\alpha = 0$ and $\beta = 1$. The slope β measures whether predictions preserve the dynamic range of the target quantity, while α captures multiplicative offset when $\beta \approx 1$. These diagnostics characterize scaling behavior rather than prediction accuracy.

Table 2: Detailed results for Single Test Status (Single) and Test Status Anchored (Anchored) on the stratified 200-instance cohorts. Rows are grouped into naïve baselines, context conditions, and end-to-end agents. Columns report macro-F_1 (F1), macro-precision (Prec), macro-recall (Rec), balanced accuracy (BAcc), accuracy (Acc), Matthews correlation coefficient (MCC), and class-specific F_1 for `PASSED` (Pass F1) and `NOT_PASSED` (Fail F1). Anchored results additionally report mean per-instance macro-F_1 (Inst. F1). All scores are multiplied by 100; higher is better. Dashes indicate unavailable or undefined values. Superscript h denotes high reasoning effort. Metric definitions appear in Appendix A.1.

	Single								Anchored								
Model	F1	Prec	Rec	BAcc	Acc	MCC	Pass F1	Fail F1	F1	Prec	Rec	BAcc	Acc	MCC	Pass F1	Fail F1	Inst. F1
Heuristics	49.5	50.4	50.3	50.3	50.0	0.7	44.4	54.5	48.3	46.7	50.0	50.0	93.3	—	96.6	0.0	75.7
No context (Noop)																	
GPT-5.6 Luna	35.1	75.9	50.5	50.5	52.0	7.3	68.2	2.0	55.0	54.4	56.4	56.4	86.2	10.6	92.5	17.5	72.3
GPT-5.6 Terra	34.7	42.3	49.5	49.5	51.0	-3.7	67.3	2.0	58.8	58.1	59.9	59.9	88.6	17.8	93.8	23.8	75.2
GPT-5.6 Sol	35.8	50.8	50.1	50.1	51.5	0.4	67.6	4.0	65.4	67.6	63.8	63.8	92.3	31.3	95.9	35.0	78.5
Claude Opus 5	48.4	53.2	52.1	52.1	53.0	5.2	63.8	32.9	63.7	61.3	69.4	69.4	87.6	29.5	93.2	34.2	76.3
Qwen3.8-27b	54.3	54.9	54.7	54.7	55.0	9.6	59.8	48.9	58.2	58.8	57.8	57.8	90.2	16.5	94.8	21.7	75.7
GLM-5.3	47.8	51.0	50.8	50.8	51.5	1.7	61.7	34.0	49.9	52.7	58.5	58.5	71.5	9.5	82.8	16.9	61.7
Sparse retrieval (BM25)																	
GPT-5.6 Luna	39.2	68.0	52.1	52.1	53.5	12.3	68.7	9.7	63.4	61.9	65.9	65.9	89.3	27.5	94.2	32.7	75.3
GPT-5.6 Terra	40.2	69.3	52.6	52.6	54.0	14.2	68.9	11.5	61.5	60.3	63.2	63.2	89.0	23.4	94.1	28.9	76.3
GPT-5.6 Sol	41.5	76.7	53.6	53.6	55.0	19.6	69.6	13.5	66.5	70.4	64.0	64.0	92.9	33.8	96.3	36.6	79.4
Claude Opus 5	45.1	64.4	54.2	54.2	55.5	15.6	69.0	21.2	63.5	61.1	69.4	69.4	87.4	29.3	93.0	33.9	76.4
Qwen3.8-27b	42.0	53.9	51.3	51.3	52.5	4.4	66.7	17.4	60.4	60.4	60.4	60.4	90.1	20.9	94.7	26.1	75.5
GLM-5.3	46.0	53.8	52.0	52.0	53.0	5.4	65.4	26.6	52.0	54.1	63.0	63.0	73.1	14.5	83.8	20.3	64.8
End-to-end agents																	
`Codex` GPT-5.6 Luna	53.5	78.6	60.3	60.3	61.5	34.3	72.8	34.2	51.8	53.7	61.6	61.6	73.5	13.1	84.2	19.4	62.3
`Codex` GPT-5.6 Sol	34.0	25.8	50.0	50.0	51.5	—	68.0	0.0	55.7	54.9	58.9	58.9	84.4	13.3	91.4	20.1	70.7
`mini-SWE` GPT-5.6 Luna	41.2	70.3	53.1	53.1	54.5	15.9	69.2	13.3	43.2	52.2	58.9	58.9	56.3	8.9	70.4	15.9	46.6
`mini-SWE` GPT-5.6 Sol	48.5	71.7	56.8	56.8	58.0	24.2	70.6	26.3	63.5	62.0	65.9	65.9	89.4	27.6	94.3	32.8	76.4
`mini-SWE` GLM-5.3	40.7	61.3	52.2	52.2	53.5	9.9	68.3	13.1	56.1	56.1	66.9	66.9	78.4	20.4	87.4	24.9	66.9
Execution trace oracle																	
GPT-5.6 Luna	58.3	79.6	63.4	63.4	64.5	39.8	74.4	42.3	57.2	56.1	60.4	60.4	85.5	16.0	92.0	22.4	73.1
GPT-5.6 Terra	67.0	81.8	69.6	69.6	70.5	49.9	77.7	56.3	62.3	61.2	63.9	63.9	89.5	24.9	94.3	30.3	76.6
GPT-5.6 Sol	73.5	83.9	74.7	74.7	75.5	57.9	80.8	66.2	65.8	70.3	63.2	63.2	93.0	32.8	96.3	35.4	79.0
Claude Opus 5	80.8	85.6	81.0	81.0	81.5	66.4	84.5	77.0	65.1	62.8	69.4	69.4	89.0	31.5	94.0	36.2	77.3
Qwen3.8-27b	52.9	65.2	58.0	58.0	59.0	22.0	69.9	35.9	59.2	61.0	58.0	58.0	91.2	18.7	95.3	23.0	76.5
GLM-5.3	60.0	72.9	63.6	63.6	64.5	35.3	73.4	46.6	49.4	52.0	56.2	56.2	72.4	7.0	83.5	15.3	62.8

- **Calibrated** $\mathrm{MAE}_{\log_{10}}$**.** We fit calibration parameters a and b by minimizing the absolute error between measured and recalibrated predicted log values:

$$\mathrm{MAE}^{\mathrm{cal}}_{\log_{10}} = \min_{a,b\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} \left|\log_{10} t_i - (a + b\log_{10}\widehat{t}_i)\right|.$$

  Unlike debiasing, calibration corrects both multiplicative offset and scale mismatch. Because a and b are fitted using measured values from the target machine, this metric is a post-hoc diagnostic.

- **Within-factor accuracy.** The fraction of predictions within a multiplicative factor f of the measurement,

$$\mathrm{Acc}_f = \frac{1}{n}\left|\{i : |d_i| \le \log_{10} f\}\right|,$$

  reported for $f \in \{1.25, 2\}$. These metrics measure the fraction of practically accurate predictions and complement MAE by characterizing the distribution of errors.

Cross-machine metric variability. To identify which metric is more robust to the choice of hardware, we conduct the following experiment. We run label collection on four different machines (Table 8) and evaluate prediction against these labels.

Table 3: Detailed results for time prediction and hotspot localization on the stratified 200-instance cohorts. Rows are grouped into naïve baselines, context conditions, and end-to-end agents. Prediction columns report raw, debiased, and calibrated mean absolute $\log_{10}$ error (MAE; lower is better), the percentages of predictions within factors of 1.25 and 2 of measured wall-clock time, and the log–log regression slope and intercept (ideal values: 1 and 0, respectively). Hotspot columns report Hit Rate (Hit) and macro/micro nDCG at $k \in \{1, 5, 20\}$, with Somers' D, Kendall's τ_b, and reference hotspot cost captured (Cost) at $k \in \{5, 20\}$. Hit Rate is shown on a 0–1 scale; all other hotspot scores are multiplied by 100. Higher within-factor accuracy and hotspot scores are better. Dashes indicate unavailable or undefined values. Superscript h denotes high reasoning effort.

	Pred.							Hotspots																
Model	Raw MAE	Debiased MAE	Calibrated MAE	Within 1.25×	Within 2×	Log-log slope	Log-log intercept	Hit@1	$nDCG_{macro}@1$	$nDCG_{micro}@1$	Hit@5	$nDCG_{macro}@5$	$nDCG_{micro}@5$	Somers' D@5	τ_b@5	Cost@5	Hit@20	$nDCG_{macro}@20$	$nDCG_{micro}@20$	Somers' D@20	τ_b@20	Cost@20
Heuristics	0.666	0.628	—	8.5	26.5	0	0.699	0.105	25.0	37.5	0.239	30.8	41.9	25.2	—	25.7	0.456	37.8	49.4	29.4	—	45.8
No context (Noop)																								
GPT-5.6 Luna	0.479	0.452	0.445	21.0	50.0	0.532	0.216	0.175	28.3	28.0	0.232	32.2	36.0	6.6	6.5	31.5	0.243	34.0	35.6	10.0	10.0	37.1
GPT-5.6 Terra	0.474	0.427	0.42	21.5	53.0	0.554	0.114	0.3	42.9	52.7	0.393	51.9	60.2	27.2	27.2	52.1	0.402	55.0	60.6	29.8	29.7	61.4
GPT-5.6 Sol	0.448	0.418	0.409	18.5	52.0	0.583	0.168	0.36	50.6	56.6	0.473	59.4	63.5	36.3	36.3	58.5	0.51	63.2	65.5	37.2	37.2	69.7
Claude Opus 5	0.466	0.45	0.435	14.5	45.0	0.604	0.431	0.34	47.2	62.8	0.434	55.7	67.2	27.0	27.0	54.0	0.456	58.4	68.5	30.3	30.3	62.3
Qwen3.8-27b	0.627	0.541	0.469	12.5	37.0	0.64	-0.114	0.175	29.0	29.5	0.252	35.0	32.3	21.1	21.1	34.7	0.278	39.0	35.2	20.2	20.2	46.1
GLM-5.3	0.567	0.563	0.513	13.5	36.5	0.434	0.465	0.165	27.6	21.3	0.259	34.4	35.0	12.5	12.5	35.6	0.281	39.0	37.6	15.4	15.4	47.3
Sparse retrieval (BM25)																								
GPT-5.6 Luna	0.47	0.453	0.445	22.5	51.5	0.546	0.202	0.305	44.0	50.5	0.372	49.9	58.9	27.5	27.5	48.2	0.379	53.2	59.0	28.6	28.6	58.1
GPT-5.6 Terra	0.46	0.407	0.392	20.0	49.0	0.63	0.048	0.315	47.0	55.3	0.407	55.2	58.8	27.8	27.8	53.9	0.42	58.2	59.1	32.5	32.5	62.9
GPT-5.6 Sol	0.378	0.359	0.353	25.5	57.0	0.679	0.142	0.43	57.8	65.0	0.494	64.6	66.9	34.9	34.9	62.2	0.533	68.0	69.8	38.8	38.8	72.5
Claude Opus 5	0.398	0.372	0.358	17.5	61.0	0.71	0.451	0.36	49.8	58.3	0.466	59.6	67.6	30.5	30.5	59.0	0.481	62.6	69.7	36.6	36.6	68.0
Qwen3.8-27b	0.66	0.554	0.446	14.5	34.0	0.756	-0.251	0.195	32.4	36.9	0.262	35.7	40.3	24.9	24.9	34.3	0.276	39.2	40.6	24.1	24.1	45.4
GLM-5.3	0.569	0.561	0.528	11.5	42.0	0.362	0.539	0.225	35.4	29.4	0.285	40.5	35.2	21.7	21.7	39.2	0.298	43.6	36.7	24.5	24.5	48.9
End-to-end agents																								
`Codex` GPT-5.6 Luna	0.603	0.515	0.489	10.0	42.0	0.489	0.065	0.21	32.4	31.9	0.269	37.6	39.6	22.4	22.3	37.2	0.296	40.7	41.2	20.6	20.5	46.8
`Codex` GPT-5.6 Sol	0.538	0.477	0.437	18.5	45.5	0.648	-0.047	0.195	30.2	38.2	0.313	38.2	47.7	18.8	18.8	39.6	0.361	42.2	48.8	19.7	19.7	49.2
`mini-SWE` GPT-5.6 Luna	0.451	0.378	0.377	16.5	47.5	0.703	-0.057	0.095	16.3	16.1	0.139	19.2	22.2	-10.0	-10.0	18.6	0.152	20.9	22.7	-0.4	-0.4	22.9
`mini-SWE` GPT-5.6 Sol	0.343	0.313	0.302	31.0	59.5	0.859	-0.124	0.175	24.7	39.9	0.22	28.1	39.5	-9.9	-9.9	27.0	0.214	28.9	39.1	5.0	5.0	29.7
`mini-SWE` GLM-5.3	0.499	0.474	0.399	16.0	40.5	0.844	-0.109	0.345	49.7	59.1	0.436	57.7	60.9	28.3	28.2	56.5	0.432	60.3	62.4	35.2	35.2	64.3
Execution trace oracle																								
GPT-5.6 Luna	0.302	0.296	0.295	27.5	68.0	0.672	0.188	0.255	33.3	28.1	0.267	35.1	29.9	1.5	1.5	32.7	0.283	36.3	30.0	9.6	9.6	36.6
GPT-5.6 Terra	0.295	0.279	0.278	30.0	67.0	0.695	0.122	0.435	56.1	65.4	0.484	63.7	70.0	28.1	28.1	62.1	0.508	66.7	74.1	33.9	33.9	70.0
GPT-5.6 Sol	0.225	0.221	0.221	39.0	77.0	0.816	0.102	0.55	69.6	80.0	0.616	77.3	80.8	42.4	42.4	75.7	0.67	80.3	83.8	45.8	45.8	84.1
Claude Opus 5	0.318	0.295	0.291	21.5	59.0	0.804	0.267	0.365	48.8	54.6	0.412	53.6	57.7	18.6	18.6	51.2	0.451	55.8	59.9	28.3	28.3	57.5
Qwen3.8-27b	0.548	0.497	0.406	15.0	40.0	0.875	-0.255	0.2	32.9	48.4	0.29	38.0	48.5	8.6	8.6	36.6	0.271	38.9	50.6	21.0	21.0	41.1
GLM-5.3	0.471	0.47	0.437	16.5	48.0	0.597	0.325	0.195	29.0	36.5	0.275	36.1	41.2	5.9	5.9	35.6	0.301	39.7	42.0	15.3	15.3	45.8

Debiased $\log_{10}$ MAE varies little across machines (0.460–0.483; Table 5). Relative to c3, rankings are unchanged on e2 and n2 and shift by at most two places on t2d (Table 6). Debiasing removes constant offsets in log space, making the metric exactly invariant to uniform runtime scaling. Calibrated $\log_{10}$ MAE, which fits a slope as well as an offset, is uniformly lower (0.402–0.429) at comparable stability: its orderings are identical to c3's on e2 and shift by at most two places elsewhere.

In contrast, raw $\log_{10}$ MAE increases by 37% between c3 and e2 (0.515 to 0.707), despite identical predictions (Table 5). Although rankings remain relatively stable ($\rho \geq 0.925$; Table 6), raw absolute error values remain machine-dependent.

A.3 Time and Memory Hotspot Metrics

For instance i, let $\mathcal{F}_i$ denote the set of profiled project functions and let $w_i : \mathcal{F}_i \to \mathbb{R}_{\geq 0}$ denote their measured cost (exclusive execution time for time hotspots and cumulative attributed memory for memory hotspots). A submission produces an ordered list P_i, truncated to its first k unique entries. Functions not contained in $\mathcal{F}_i$ contribute zero gain.

To handle profiler ties, set-based metrics are evaluated against a tie-tolerant expected set $E_i^{(k)}$. This set replaces missed reference functions with equally costly functions selected by the model, with at most one replacement per missed slot. Thus, predictions are not penalized for choosing between functions with indistinguishable measured costs.

Table 4: Detailed results for memory prediction and hotspot localization on the stratified 200-instance cohorts. Rows are grouped into naïve baselines, context conditions, and end-to-end agents. Prediction columns report raw, debiased, and calibrated mean absolute $\log_{10}$ error (MAE; lower is better), the percentages of predictions within factors of 1.25 and 2 of measured peak memory, and the log–log regression slope and intercept (ideal values: 1 and 0, respectively). Hotspot columns report Hit Rate (Hit) at $k \in \{1, 5\}$, reference hotspot cost captured (Cost) and macro/micro nDCG at $k = 5$, and rank agreement measured by Somers' D and Kendall's τ_b. Hit Rate is shown on a 0–1 scale; all other hotspot scores are multiplied by 100. Higher within-factor accuracy and hotspot scores are better. Dashes indicate unavailable or undefined values. Superscript h denotes high reasoning effort.

	Pred.							Hotspots						
Model	Raw MAE	Debiased MAE	Calibrated MAE	Within 1.25x	Within 2x	Log-log slope	Log-log intercept	Hit@1	Hit@5	Cost@5	$\text{nDCG}_{\text{macro}}$@5	$\text{nDCG}_{\text{micro}}$@5	Somers' D	τ_b
Heuristics	0.516	0.506	—	12.5	34.5	0	9.000	0.175	0.254	18.1	20.5	14.8	49.0	—
No context (Noop)														
GPT-5.6 Luna	1.013	0.713	0.482	11.0	25.0	0.393	4.510	0.435	0.371	57.2	57.8	20.1	36.1	35.9
GPT-5.6 Terra	0.646	0.609	0.489	34.5	57.0	0.34	5.368	0.455	0.434	62.3	63.2	21.0	43.0	43.0
GPT-5.6 Sol	0.346	0.335	0.326	44.5	68.5	0.727	2.164	0.525	0.544	71.2	72.3	63.3	50.1	50.0
Claude Opus 5	0.484	0.455	0.405	37.5	60.0	0.625	3.009	0.5	0.456	60.9	62.5	59.9	41.1	40.9
Qwen3.8-27b	1.496	1.148	0.498	11.0	24.5	0.353	4.383	0.215	0.203	39.6	36.8	12.4	18.1	18.0
GLM-5.3	1.268	1.082	0.501	15.5	32.0	0.452	3.727	0.255	0.211	39.1	37.5	15.4	25.3	25.1
Sparse retrieval (BM25)														
GPT-5.6 Luna	0.941	0.701	0.472	16.5	30.0	0.479	3.790	0.42	0.374	60.2	59.4	20.8	33.2	33.1
GPT-5.6 Terra	1.123	0.675	0.479	9.0	20.5	0.41	4.254	0.48	0.439	63.2	64.1	63.4	41.1	41.0
GPT-5.6 Sol	0.706	0.631	0.462	29.5	50.5	0.557	3.312	0.615	0.547	72.0	74.4	66.5	51.5	51.4
Claude Opus 5	0.58	0.537	0.418	30.5	53.0	0.688	2.345	0.535	0.494	66.1	66.8	65.4	43.4	43.2
Qwen3.8-27b	1.796	1.442	0.492	10.5	24.5	0.689	1.068	0.285	0.21	39.6	39.6	14.0	28.4	28.0
GLM-5.3	0.973	0.851	0.501	19.0	38.5	0.415	4.378	0.22	0.214	39.8	36.0	16.0	15.0	14.7
End-to-end agents														
`Codex` GPT-5.6 Luna	1.853	1.307	0.506	5.5	20.0	0.032	6.906	0.245	0.215	39.3	37.2	8.2	16.8	16.6
`Codex` GPT-5.6 Sol	0.954	0.845	0.501	24.0	38.0	0.232	6.028	0.345	0.427	60.9	57.0	63.2	31.2	31.1
`mini-SWE` GPT-5.6 Luna	1.016	0.706	0.483	15.0	24.5	0.325	5.138	0.2	0.138	26.4	26.2	11.2	1.0	0.9
`mini-SWE` GPT-5.6 Sol	0.698	0.618	0.453	29.5	46.0	0.554	3.365	0.52	0.475	69.0	70.3	65.4	52.0	52.0
`mini-SWE` GLM-5.3	1.046	0.978	0.497	25.5	49.5	0.427	4.175	0.455	0.377	61.3	62.1	22.5	41.4	40.9
Execution trace oracle														
GPT-5.6 Luna	1.065	0.67	0.48	10.0	23.5	0.41	4.268	0.545	0.521	72.6	73.3	72.9	49.2	49.2
GPT-5.6 Terra	0.854	0.693	0.486	22.5	37.5	0.402	4.565	0.605	0.574	74.3	76.7	67.5	53.8	53.8
GPT-5.6 Sol	0.751	0.631	0.461	29.0	40.5	0.562	3.215	0.66	0.683	81.8	83.6	68.9	65.3	65.2
Claude Opus 5	0.672	0.608	0.442	30.0	47.0	0.656	2.512	0.445	0.429	58.3	58.4	22.1	30.7	30.5
Qwen3.8-27b	2.025	1.536	0.502	6.0	20.5	0.222	5.040	0.435	0.34	55.9	56.5	44.9	37.3	37.0
GLM-5.3	1.373	1.213	0.496	16.0	34.5	0.568	2.619	0.365	0.327	56.9	54.3	59.1	30.4	29.8

- **HitRate@k.** The fraction of expected hotspot functions recovered:

$$\text{HitRate}\,@k = \frac{|P_i \cap E_i^{(k)}|}{|E_i^{(k)}|}.$$

  Since both sets contain k functions, this is equivalent to precision and recall at k. It measures retrieval only and ignores function cost and ordering.

- **CostCaptured@k.** The fraction of reference cost recovered:

$$\text{Cost}\,@k = \frac{\sum_{f \in P_i \cap E_i^{(k)}} w_i(f)}{\sum_{f \in E_i^{(k)}} w_i(f)}.$$

  Unlike HitRate@k, this weights functions by their contribution to the measured bottleneck.

Table 5: Time prediction metrics averaged over 19 model runs. Each run's predictions are evaluated against wall-clock runtimes measured on four different 4-vCPU GCP VMs. Predictions are identical across columns; only the ground truth differs. Arrows indicate the preferred direction.

Metric	c3	e2	n2	t2d
Raw MAE ↓	0.515	0.707	0.579	0.553
Debiased MAE ↓	0.461	0.462	0.460	0.483
Calibrated MAE ↓	0.413	0.402	0.410	0.429
Log-log slope ↓	0.406	0.396	0.405	0.426
Log-log intercept ↓	0.186	0.199	0.168	0.184
Within 1.25x ↑	0.181	0.062	0.123	0.164
Within 2x ↑	0.463	0.244	0.394	0.427

Table 6: Rank agreement with the c3 ordering across 19 model runs. For each VM, ρ denotes Spearman correlation and τ denotes Kendall correlation. The final column reports the largest rank change of any single run, measured in places.

Metric	e2		n2		t2d		Max. rank
	ρ	τ	ρ	τ	ρ	τ	change
Raw MAE	+0.925	+0.813	+0.991	+0.953	+0.998	+0.988	5
Debiased MAE	+1.000	+1.000	+1.000	+1.000	+0.991	+0.953	2
Calibrated MAE	+1.000	+1.000	+0.998	+0.988	+0.982	+0.918	2
Log-log slope	+0.998	+0.988	+0.998	+0.988	+0.996	+0.977	1
Log-log intercept	+0.135	−0.006	+0.709	+0.567	+0.991	+0.942	12
Within 1.25x	+0.037	+0.031	+0.815	+0.655	+0.952	+0.861	16
Within 2x	+0.754	+0.611	+0.959	+0.885	+0.981	+0.926	10

- **Macro and micro nDCG@k.** For an ordered list S, define discounted cumulative gain as

$$\mathrm{DCG}(S) = \sum_{p=1}^{|S|} \frac{w_i(S_p)}{\log_2(p+1)}.$$

Let

$$A_i = \mathrm{DCG}(P_i), \qquad I_i = \mathrm{DCG}(\mathrm{sort}_{\downarrow w_i}(E_i^{(k)}))$$

denote the achieved and ideal gains. Metrics are undefined for instances with empty expected sets or zero total reference cost. We use cost-weighted nDCG:

$$\mathrm{nDCG}^{\mathrm{macro}}\,@k = \frac{1}{n}\sum_i \frac{A_i}{I_i}, \qquad \mathrm{nDCG}^{\mathrm{micro}}\,@k = \frac{\sum_i A_i}{\sum_i I_i}.$$

Macro nDCG normalizes each instance independently and, therefore, weights instances equally. Micro nDCG aggregates gains before normalization, and therefore weights instances by their total hotspot cost. Both evaluate whether high-cost functions are ranked early, rather than only whether they are retrieved.

- **Somers' D.** measures rank agreement between the predicted ordering of the model and the ordering implied by the measured costs, calculated as $(C - D)/(n_0 - T_y)$, where C and D count concordant and discordant pairs. A pair of functions is concordant if the predicted ordering agrees with their ordering by measured cost, and discordant if the predicted ordering is reversed. n_0 is the total number of pairs, and T_y is the number of pairs with identical measured costs in the ground truth. Normalizing over pairs ordered by the ground truth excludes profiler-induced ties and allows a perfectly ordered ranking to achieve 1.0. Because every candidate function carries a measured cost, Somers' D evaluates the ordering of the functions

Table 7: Documented model context windows. Prompt fitting uses the configured endpoint input limit.

Model	Context window (tokens)
GPT-5.6 Luna / Terra / Sol (OpenAI, 2026a;c;b)	1,050,000
Claude Opus 5 (Anthropic, 2026a)	1M
GLM-5.3 (Z.ai, 2026a)	1M
Qwen3.8-27B (Qwen Team, 2026)	262,144

returned by the model rather than only the intersection with the reference top-k. Unlike HitRate@k, which measures membership, Somers' D measures relative ordering; together they separate retrieval quality from ranking quality.

- **Kendall's τ_b.** Pairwise rank correlation between the predicted and reference orderings, with corrections for ties in both rankings:

$$\tau_b = \frac{C - D}{\sqrt{(C + D + T_x)(C + D + T_y)}},$$

where C and D denote concordant and discordant pairs, and T_x and T_y denote pairs tied only in the prediction and reference, respectively. The metric ranges from -1 (inverse ordering) to 1 (perfect agreement). Unlike Somers' D, Kendall's τ_b accounts for ties in both rankings, making its maximum attainable value depend on the number of profiler-induced ties.

We report hotspot metrics at $k \in \{1, 5, 20\}$ for time prediction and $k \in \{1, 5\}$ for memory prediction.

Metric comparison. HitRate@k and CostCaptured@k evaluate retrieval quality. Macro and micro nDCG evaluate retrieval with position-discounted, cost-weighted gain, whereas Somers' D and Kendall's τ_b measure pairwise rank agreement. Somers' D normalizes only over pairs ordered by the reference and therefore always attains a maximum value of 1, whereas the maximum attainable value of Kendall's τ_b decreases in the presence of ground-truth ties. Unlike Somers' D and τ_b, nDCG is non-negative and therefore cannot distinguish an inverted ranking from an uninformative one.

B Experimental Setup: Further Details

This appendix records the implementation details needed to reproduce context construction and model inference in Section 5. An overview of our pipeline is presented in Figure 1.

B.1 Heuristics

For Single Test Status, *Heuristics* samples uniformly with seed 42 from `PASSED`, `FAILED`, and `ERROR` for each test, then maps the latter two to `NOT_PASSED`. Thus, under the binary formulation evaluated here, it predicts `PASSED` with probability 1/3. For Test Status Anchored, Heuristics always returns `PASSED`. For Time and Memory Prediction, *Heuristics* returns 5 s and 10^9 bytes, respectively. For Time and Memory Hotspots, *Heuristics* ranks the profiled candidate functions executed by each workload functions by descending call count, without using the measured time or memory costs that define the reference ranking. We score the top-20 time and top-5 cumulative-memory prefixes with HitRate@20 and HitRate@5.

B.2 Context Collection Baselines

B.2.1 Shared task inputs

All providers receive the same task specification and fixed task inputs, including any task-provided patch or entry point, the workload command, and resolved source code. The No-retrieval baseline adds no repository-derived context.

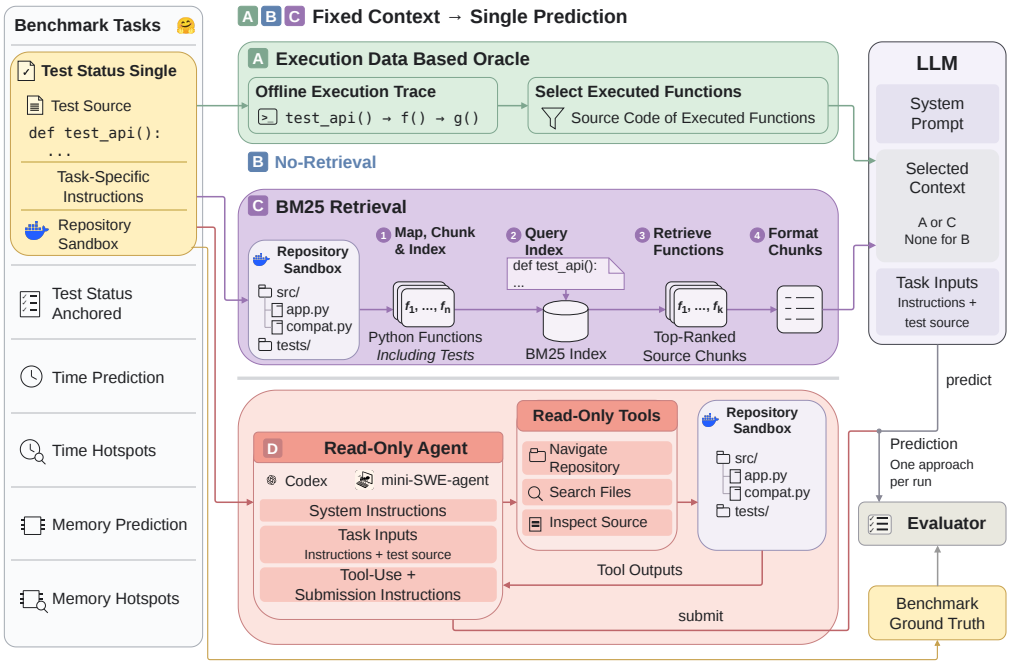


Figure 1: **Overview of the evaluation pipeline.** All approaches receive the same task-specific inputs, illustrated here for Test Status Single. Fixed-context approaches make a single zero-shot LLM prediction using source code of functions identified by offline execution (A), no additional repository context (B), or BM25-retrieved source code (C). The read-only agent (D) gathers context through repository inspection and submits its prediction within a single rollout. We evaluate each prediction independently against benchmark ground truth using task-specific metrics (Section 4).

B.2.2 Execution-trace oracle

We collect a runtime trace from the Python interpreter and identify the repository functions invoked by the target tests or workload. The model receives the source code of those functions and must infer the workload's behavior from the supplied source.

B.2.3 BM25

We use the `BM25Okapi` implementation from `rank-bm25`, version 0.2.2. We index function-level chunks extracted from Python files, including test files, following the chunking strategy described in SWE-bench (Jimenez et al., 2024).

Query construction depends on the task. For resource estimation and bottleneck localization, we perform a single query consisting of the workload command concatenated with the source code of any resolvable Python files named in the command. For Single Test Status, the query is the test command concatenated with the source code of the selected test.

Test Status Anchored uses multiple queries based on the patch, the test command, and the resolved source code of the 100 target tests. The tests are divided into ten groups, each containing the source code of ten tests. We merge the query results using Reciprocal-Rank Fusion (RRF). Retrieved functions are returned in descending BM25 score order for single-query retrieval and descending RRF score order for Test Status Anchored.

Table 8: Descriptions of different machines used for time tracer stability experiments.

Machine type	CPU model	Microarch.	GHz	vCPUs
c3-standard-4	Intel Xeon Platinum 8481C	Sapphire Rapids	2.70	4
n2-standard-4	Intel Xeon (Cascade Lake)	Cascade Lake	2.80	4
e2-standard-4	Intel Xeon (Broadwell-EP)	Broadwell-EP	2.20	4
t2d-standard-4	AMD EPYC 7B13	Milan	2.45	4

Reciprocal-Rank Fusion (RRF) We use RRF to merge the results of multiple BM25 queries as follows:

$$\mathrm{RRF}(c) = \sum_{j=1}^{m} \frac{1}{k + \mathrm{rank}_j(c)}$$

where $k = 60$ by convention, c is a retrieved chunk, m is the number of rankers, and $\mathrm{rank}_j(c)$ is the rank assigned to c by ranker j.

B.2.4 Context budgets and overflow

During prompt construction, providers with context-window fitting enabled add chunks in provider order and recount the rendered context after each addition. They stop before the first chunk that exceeds the budget, discarding it and the remaining suffix. This preserves descending BM25 or RRF order for retrieval and the oracle provider's deterministic order. The context budget is

$$B = \max(0, \lfloor (1 - m)C \rfloor - R - H),$$

where C is the configured endpoint input limit (or LiteLLM's catalog value), m is the safety margin, R reserves completion tokens, and H counts the fixed system and user inputs. The provided fitting configurations use $m = 0.1$. Token counts use `tiktoken`'s `cl100k_base` encoding. If the first chunk cannot fit, context collection records an error; fixed task inputs are not truncated.

For frozen replay, we fit the stored ranked chunks to the consuming model's budget and re-render the prompts from the retained context and stored task inputs. Any endpoint context-length error is recorded as a failed prediction, as it relates to an inherent limitation of the model.

B.2.5 Agentic baseline

The agent receives the task specification, shared task inputs, and harness-specific instructions for tool use and prediction submission. It iteratively searches and inspects a read-only repository, using its observations to guide subsequent reasoning and exploration. The agent may not execute the workload and must submit a valid prediction within a budget of 40 turns.

We evaluate GPT-5.6 Luna and GPT-5.6 Sol with both Codex and mini-SWE-agent, and GLM-5.3 with mini-SWE-agent.

B.3 Model Selection and Inference Settings

Unless otherwise noted, fixed-context runs use high reasoning effort; the low/high comparison is reported in Appendix F. For fixed-context prediction calls, the evaluation configuration sets temperature to 1.0 for all six models. This matches the documented GLM-5.3 default and Qwen3.8-27B thinking-mode setting (Z.ai, 2026b; Qwen Team, 2026). Claude Opus 5 accepts 1.0 for compatibility but does not support changing temperature (Anthropic, 2026b).

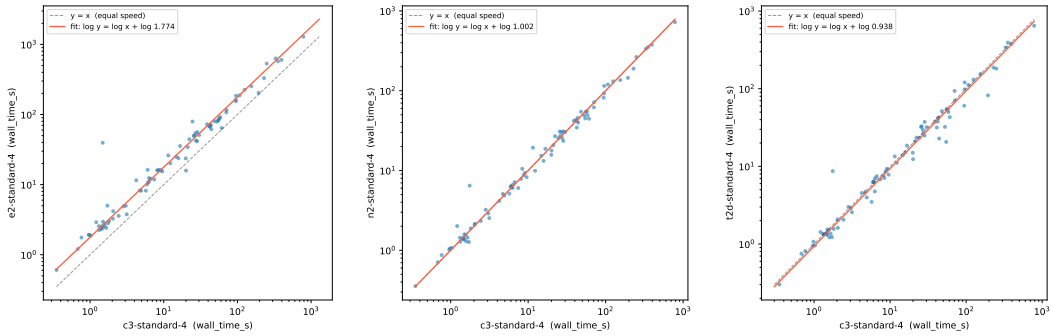


Figure 2: Wall time on `c3` vs. each other machine type (log scale).

Table 9: Repository composition within each task dataset (percentages).

Repository	Single Test Status	Test Status Anchored	Time Prediction	Memory Prediction	Time Hotspots	Memory Hotspots
pandas-dev/pandas	59.5	59.5	60.7	61.2	62.0	62.2
scipy/scipy	10.6	10.6	10.9	10.9	10.4	10.6
astropy/astropy	8.4	8.4	8.6	8.6	8.2	8.3
scikit-learn/scikit-learn	6.6	6.6	5.8	5.9	5.2	5.2
matplotlib/matplotlib	4.9	4.9	4.6	3.9	4.7	4.0
numpy/numpy	4.9	4.9	5.1	5.3	5.2	5.4
pydata/xarray	2.7	2.7	2.5	2.6	2.6	2.7
sympy/sympy	1.5	1.5	0.8	0.8	0.8	0.8
dask/dask	1.0	1.0	1.0	0.8	1.0	0.8

C Time tracer stability

The runtime of a program depends strongly on the machine on which it is executed, as well as on several other factors. In this section, we investigate how these factors affect workload runtime estimates.

Different Machines. We evaluate the same subset of datapoints on four different machines (Table 8). Figure 2 compares workload runtimes measured on *c3-standard-4* with the corresponding runtimes on each of the other machine, using a loglog scale. The solid red line represents a fitted linear regression with a fixed slope, while the dotted $y = x$, is included as a reference. For every pair of machines, Spearman's correlation exceeds 0.98, indicating that the relative ordering of workload runtimes remains highly consistent across machines.

Multiple Runs. We also evaluate the stability of runtime measurements across repeated runs on the same *c3-standard-4* machine. Specifically, we measure 100 data points over four runs and compute the coefficient of variation, CV = std/mean, for each data point. The average CV is 0.054 , indicating low variability across runs.

D Task Dataset Distributions

We describe the task datasets by the three task families defined in Section 4.

Dataset sizes depend on task-specific sampling and on which examples have usable test outcomes or profiling measurements. We use the same computational resources to run test suites and workloads. Test suites run without profiling. We profile the original workloads to measure execution time. For memory measurements, we trace allocations in the amplified workloads. These procedures have different resource demands, so an example may yield usable data for one task while exceeding the time or memory limits for another.

All six datasets span nine repositories; Table 9 reports their repository composition.

Behavior Preservation Tasks. We construct the test-status datasets by sampling prediction states from the core dataset. Before sampling, we exclude states for which container failures or incompatibilities with older test frameworks prevent us from collecting test outcomes. The resulting Single Test Status and Test Status Anchored datasets each contain 814 instances.

Figure 3(a) shows their outcome distributions. Single Test Status has nearly balanced prediction labels: 51.2% are `PASSED` and 48.8% are `NOT PASSED`, with the latter combining `FAILED` and `ERROR` outcomes. Each Test Status Anchored instance contains 100 selected tests. Although 91.8% of individual test outcomes are `PASSED`, 57.4% of instances have all selected tests passing.

Resource Estimation Tasks. We construct Time Prediction by sampling prediction states from the core dataset, excluding states whose wrapped workload tests fail or time out. Time Prediction contains 788 instances, with elapsed-time targets ranging from 0.04 to 3,612.46 seconds and a median of 11.06 seconds.

For Memory Prediction, we profile amplified workloads on the states sampled for Time Prediction. We exclude states for which no workload variant completes successfully under memory profiling, including cases of timeouts or out-of-memory errors. Memory Prediction contains 760 instances, with a median peak increase in traced memory of 1,098.41 MiB. Figure 3(b) shows the prediction-target distributions for both tasks.

Bottleneck Localization Tasks. We use the time and memory measurements to construct the hotspot datasets. We retain only examples with at least 20 profiled repository functions. Time Hotspots contains 771 instances with a median of 292 candidate functions. Memory Hotspots contains 747 instances with a median of 283 candidate functions.

Figure 3(c) shows the candidate-set sizes and cost concentration. In Time Hotspots, the highest-ranked 20 functions capture a median of 94.0% of exclusive execution time attributed to the candidate pool. In Memory Hotspots, the highest-ranked five functions capture a median of 97.4% of cumulative attributed memory.

E SourceWorldBench Lite: Further Details

SourceWorldBench Lite is introduced in Section 3.3. We assess its distributional alignment (Section E.1), metric variability across cohort sizes (Section E.2), and paired comparison uncertainty across cohort sizes (Section E.3). Finally, Section E.4 examines model-rank stability across context collection baselines and tasks, together with Oracle and Agent comparisons on SourceWorldBench Lite.

E.1 Distributional Alignment

Tables 10 and 11 compares the directly controlled stratification dimensions with their population distributions. Repository shares and complexity-bucket or label proportions differ by at most 1.5 percentage points. This agreement primarily reflects the stratified sampling procedure rather than an independent validation of metric stability.

We additionally examine covariates not directly controlled by stratification using normalized quantile–quantile comparisons at 50 matched percentiles (Figure 4). The selected cohorts closely track the eligible population distributions of oracle-context length, available executed functions, and scalar prediction targets, with normalized deviations from the diagonal of approximately 0.05 or less at the evaluated percentiles. These validations support distributional alignment along the measured dimensions, although they do not establish equivalence for unmeasured characteristics or guarantee identical evaluation scores, as in the full set.

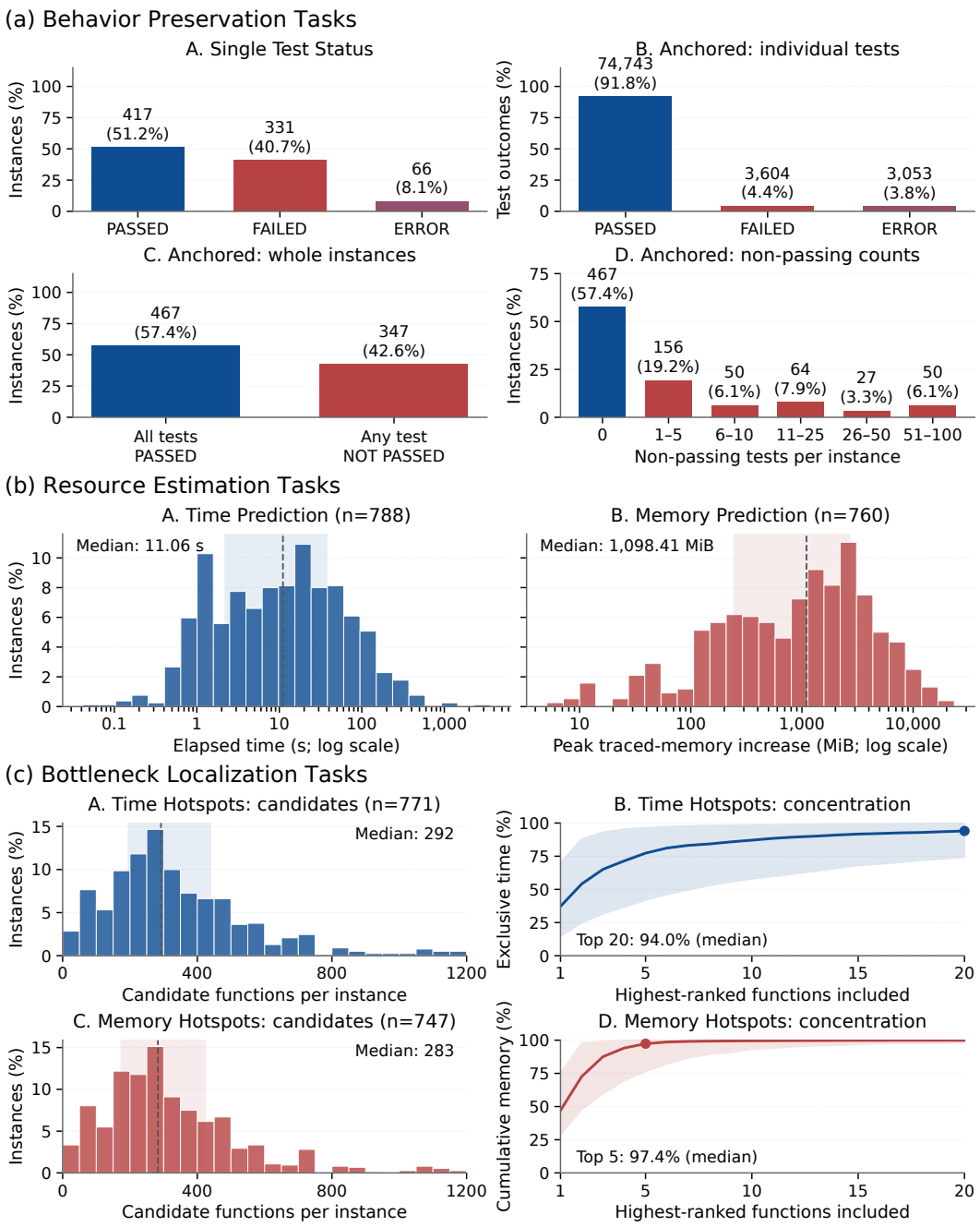


Figure 3: **Task dataset distributions.** **(a)** Behavior preservation: A, outcomes for Single Test Status instances; B, outcomes Test Status Anchored tests; C, whether all 100 selected tests pass; D, `NOT PASSED` tests per instance. C–D use Test Status Anchored instances. **(b)** Resource estimation: elapsed time and peak traced-memory. **(c)** Bottleneck localization: candidate counts (left) and cost concentration (right), for Time Hotspots and Memory Hotspot. Histograms show percentages, dashed medians, and shaded interquartile ranges. Curves show median cost captured with interquartile bands and marked task cutoffs, normalized over each complete reported candidate pool.

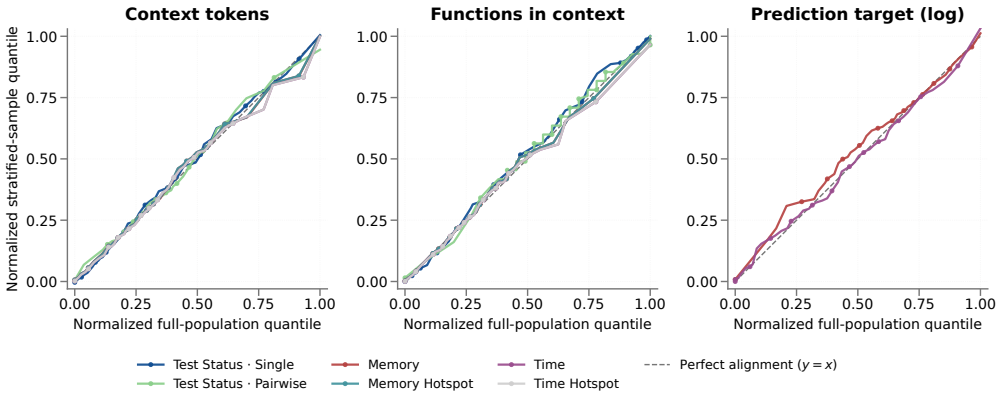


Figure 4: **Distributional alignment of SourceWorldBench Lite with SourceWorldBench.** Normalized quantile–quantile comparisons show: **(left)** oracle-context length in tokens, **(middle)** the number of executed functions available as context, and **(right)** continuous prediction targets (log-scaled `traced_delta_bytes` for memory and `wall_time_s` for time). Each line represents one task, and each point compares the eligible population and cohort quantiles at a matched percentile. Agreement with the diagonal indicates similar distributions along the measured dimension.

Table 10: Repository share (%) in the full dataset versus the n=200 stratified sample, for each of the three cohort manifests (Section E). Repository is one of the two axes stratification is performed on, so agreement here is largely by construction; we report it for completeness and because it is the dominant source of task heterogeneity (repository size and code style).

	Cohort A (resource)		Single test status		Pairwise test status	
Repository	Full	Sample	Full	Sample	Full	Sample
pandas-dev/pandas	62.7	62.5	58.7	58.5	63.8	63.5
scipy/scipy	10.3	10.0	10.5	10.5	9.8	10.0
astropy/astropy	8.1	8.5	9.7	9.5	8.2	8.5
numpy/numpy	5.4	5.5	4.5	4.5	4.2	4.0
scikit-learn/scikit-learn	5.3	5.0	6.4	6.5	4.6	4.5
matplotlib/matplotlib	4.1	4.0	5.1	5.0	5.2	5.0
pydata/xarray	2.6	2.5	3.3	3.5	2.8	2.5
dask/dask	0.8	1.5	0.5	1.0	0.5	1.0
sympy/sympy	0.8	0.5	1.4	1.0	0.8	1.0

E.2 Evaluation-Metric Sensitivity to Cohort Size

Distributional alignment does not by itself establish that a reduced cohort leads to stable evaluation scores. We therefore examine how much the reported score varies with the choice of instances at a fixed evaluation budget. We analyse repeated subsampling metrics sensitivity to cohort selection across alternative subsets of the same size.

We perform this analysis using GPT-5.6 Luna with BM25 retrieval on all six tasks. For each task, let P denote the evaluated reference pool (full dataset) and $M(P)$ its task-specific metric functions reported in the main table (Table 1). Holding the collected predictions fixed, we repeatedly draw subsets $S_n \subseteq P$ of n instances without replacement and recompute $M(S_n)$. Figure 5 reports the relative sampling standard deviation,

$$r(n) = 100 \cdot \frac{\widehat{\mathrm{SD}}[M(S_n)]}{|M(P)|}.$$

Normalizing by the reference-pool score shows cohort-selection variability on a common relative scale across tasks with different evaluation metrics. The resulting curves quantify

Table 11: Stratification balance of SourceWorldBench Lite. The resource cohort uses executed-function-count buckets, while the two test-status cohorts use binary labels. The table compares each category's share in the full dataset with its share in the n=200 sample. For each cohort, this variable was used alongside repository for stratification (Table 10), so close agreement is expected. Small differences remain because each stratum must contain a whole number of instances and the percentages are rounded.

Cohort	Stratum	Full (%)	Sample (%)
Cohort A resource	small (executed fns. < 100)	10.7	11.0
	medium (100–299)	41.9	41.5
	large (≥ 300)	47.4	47.5
Test status single	`NOT_PASSED`	48.4	48.5
	`PASSED`	51.6	51.5
Test status pairwise	`ALL_PASSED`	55.8	56.0
	`ANY_FAIL`	44.2	44.0

how much additional evaluation reduces this variability, informing the trade-off between evaluation cost and score precision.

At $n = 200$, the relative sampling standard deviation is 6.3% for Single Test Status, 8% for Test Status Anchored, 4.3% for Memory Prediction, 4.0% for Memory Hotspots, 6% for Time Prediction, and 2.9% for Time Hotspots. Across all tasks, increasing the cohort size reduces sampling variability, with substantial reductions between the smallest cohorts and $n = 200$. Nevertheless, the curves continue to decline beyond 200: the selected cohort size $n = 200$ is the cohort size at which we empirically observed stabilization of our evaluation metrics, and therefore a practical cost–precision compromise, and not a point at which additional evaluation stops improving precision.

E.3 Paired Comparison Uncertainty Across Cohort Sizes

To complement the single-model sensitivity analysis, we examine whether the chosen evaluation budget can distinguish model performance. We compare GPT-5.6 Luna and GPT-5.6 Sol with BM25 retrieval on SourceWorldBench Lite. For each resample size n, we draw n instance indices with replacement from this cohort, use identical indices for both models, and recompute their task metrics. We orient the metric difference Δ so that positive values favor Luna. Figure 6 shows the mean and central 95% interval of the resulting differences.

The intervals generally narrow with increasing resample size. At $n = 200$, the paired percentile-bootstrap intervals lie entirely below zero for Time Prediction, Memory Prediction, and both hotspot tasks, supporting Sol's advantage under this resampling procedure. In contrast, both test-status intervals include zero, consistent with the modest Luna–Sol differences in Table 1. The chosen budget therefore distinguishes the models on the four resource tasks under this resampling procedure, while leaving their smaller test-status differences unresolved.

Together, distributional alignment, single-model sampling variability, and paired comparison uncertainty motivate $n = 200$ as a practical cost–precision compromise.

E.4 Model-Rank Stability

In this section, we detail metric and model-rank uncertainty for results from Table 1.

Methodology. All intervals and tests are obtained via pairwise bootstrap over the 200 instances of each SourceWorldBench Lite task, stratified by repository. Each replicate resamples instances with replacement within each repository to retain the original per-repository counts, and then recomputes the metric for a given pair on the resampled set. We use 200,000 Monte Carlo replicates. We orient each difference Δ so that positive values favor the first-named method. We report the percentile 95% interval and the two-sided

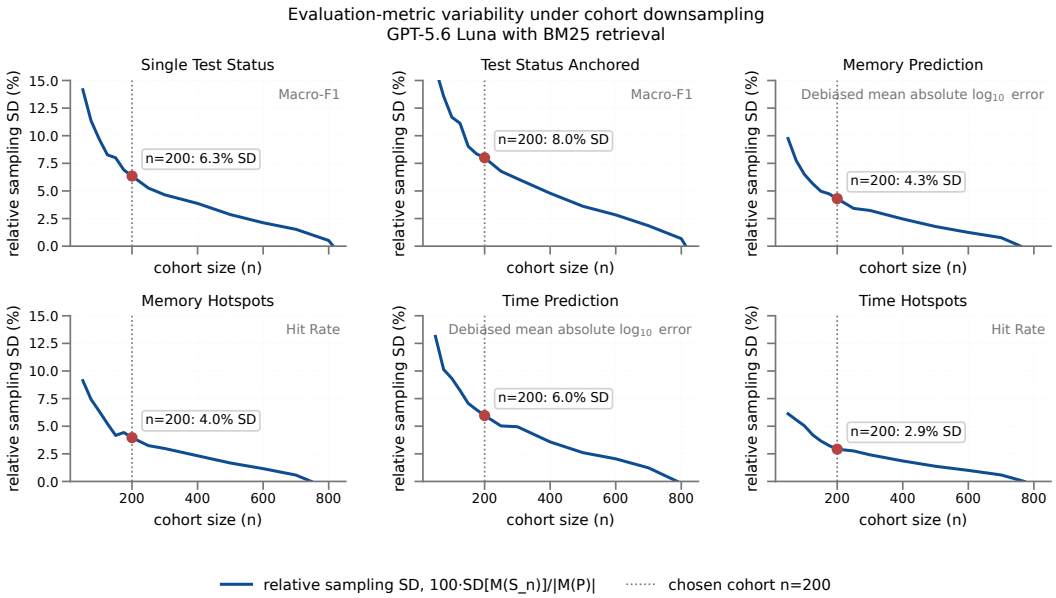


Figure 5: **Sampling variability of evaluation metrics by cohort size for GPT-5.6 Luna with BM25 retrieval.** Curves show the standard deviation of metrics recomputed on repeatedly sampled subsets of n instances, normalized by the absolute metric on the evaluated reference pool. Dotted vertical lines and red markers indicate the main evaluation cohort size, $n = 200$; annotations report relative sampling standard deviations. Predictions are held fixed, and instances are sampled without replacement. Consequently, variability reaches zero when a subset contains the entire reference pool. This endpoint is a property of finite-population sampling, not evidence of improved model performance.

bootstrap p-value $p = 2\min\{P(\Delta^* \leq 0), P(\Delta^* \geq 0)\}$. In the reported results, an audit confirmed that exclusion of zero by the pointwise interval agreed with $p < 0.05$ for every contrast.

Multiple comparisons. We apply Holm correction (Holm, 1979) separately to two comparison families. The Oracle family compares Oracle with BM25 and No-Retrieval ($K = 72$; Figures 16 and 17). The Read-Only Agent family compares mini-SWE-agent with BM25 ($K = 18$; Figure 18). We call a contrast significant when its Holm-adjusted p-value is below 0.05. The correction does not apply to bootstrap rank distributions or Kendall rank correlations.

E.4.1 Within-Baseline Model Rankings

For model comparisons, we assess the uncertainty of rankings observed on SourceWorldBench Lite. For each task and context collection baseline, we rank the considered models on each bootstrap-produced replicate and study the rank distributions. The results are available in Figure 13.

Figure 7 compares model rankings across tasks using Kendall's τ. The median pairwise τ is 0.73 for both No-Retrieval and Best Matching 25 (BM25) (range $[-0.33, 1.0]$ for each) and 0.60 for Oracle (range $[0.33, 0.87]$). Thus, model rankings show lower median agreement across tasks with Oracle context.

Table 1 shows that Oracle effects vary by model and task, which may contribute to the lower cross-task agreement. On Single Test Status, `Claude Opus 5` gains 32.4 macro-F_1 points over No-Retrieval, whereas `Qwen3.8-27b` loses 1.4 points. On Test Status Anchored, only `GPT-5.6 Luna` and `GPT-5.6 Terra` gain more than two points (2.2 and 3.5, respec-

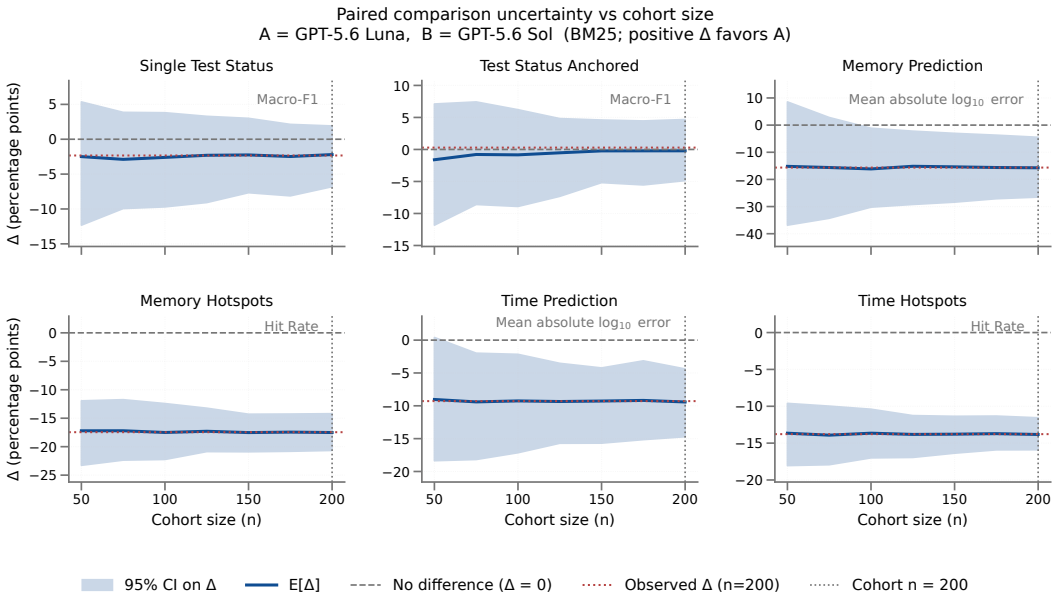


Figure 6: **Paired comparison uncertainty for GPT-5.6 Luna versus GPT-5.6 Sol with BM25 retrieval.** At each resample size n, instance indices are sampled with replacement from SourceWorldBench Lite, using identical indices for both models. Positive differences favor Luna: we subtract Sol's score from Luna's for macro-F_1 and HitRate, but reverse the subtraction for prediction error, where lower is better. Solid blue lines show mean bootstrap differences, and shaded bands show their central 95% intervals.

tively); gains for the other models are smaller or negative. Oracle context also lowers Time Prediction error for all six models, but lowers Memory Prediction error only for Luna.

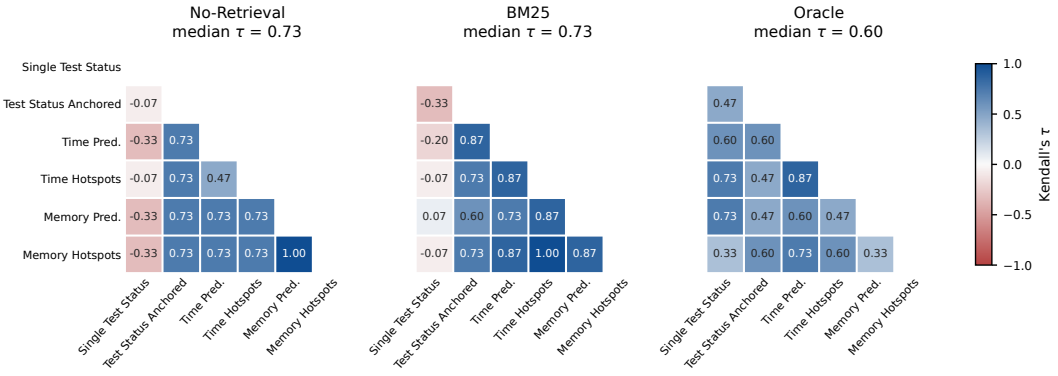


Figure 7: Pairwise Kendall's τ between observed model rankings induced by SourceWorldBench Lite tasks, computed separately for No-Retrieval, BM25, and Oracle baselines. Panel titles report the median τ across the 15 task pairs.

E.4.2 Oracle Comparisons

We investigate whether oracle context significantly impacts downstream task performance compared to BM25 (Figure 16) or No-Retrieval (Figure 17) across all models and tasks. For most Single Test Status and Time Prediction oracle context significantly improves task performance (excluding open-weights models), however, it does not improve task performance on All Test Status Anchored. For Memory Prediction, the oracle does not significantly improve task performance, we discuss this result further in Appendix K. Time and Memory

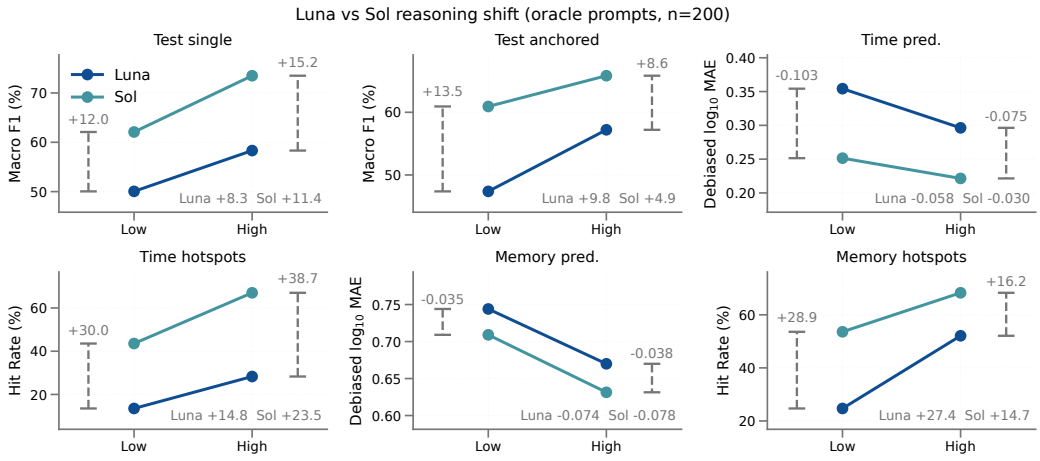


Figure 8: Effect of reasoning effort under execution-trace oracle context on the stratified $n = 200$ cohort for each task. Panels report macro-F1 for test-status prediction, top-set Hit Rate for hotspot localization, and debiased $\log_{10}$ MAE for resource estimation. Dashed brackets show the `Sol`–`Luna` gap, not uncertainty intervals; bottom annotations show each model's gain from low to high effort. F1 and Hit Rate differences are in percentage points. Slopes closer to one indicate better matching of predicted and measured cost variation, not necessarily lower pointwise error.

Hotspot Prediction, the results are mixed. The most notable result is that `GPT-5.6 Sol` significantly benefits from Oracle context on both tasks, while `Claude Opus 5` does not.

E.4.3 Agent Comparisons

Next, we investigate whether there is a significant difference in downstream task performance between our practical context collection baselines BM25 and Read-Only Agents. Specifically, we compare `GPT-5.6 Luna`, `GPT-5.6 Sol`, and `GLM-5.3` with mini-SWE-agent against the same model with BM25. In Figure 18, we find that the differences between these context collection baselines are mostly insignificant. The notable exception are Bottleneck Localization tasks, for which the Read-Only Agent outperforms BM25 for `GLM-5.3` on both time and memory, but performs worse than BM25 for GPT models. Overall, we surprisingly find that an agentic context collection approach does not significantly nor consistently outperform a simple RAG baseline.

F Effect of Reasoning Effort

We compare low and high reasoning effort for `GPT-5.6 Luna` and `GPT-5.6 Sol` across all six tasks, using the same execution-trace oracle prompts on SourceWorldBench Lite. Figure 8 shows that high effort improves every displayed metric for both models. Macro-F1 gains range from 4.9 to 11.4 percentage points across the test-status tasks, while hotspot HitRate gains range from 14.7 to 27.4 points. For resource estimation, debiased $\log_1 0$ MAE values move closer to zero, decreasing by 0.030–0.058 for time and 0.074–0.078 for memory.

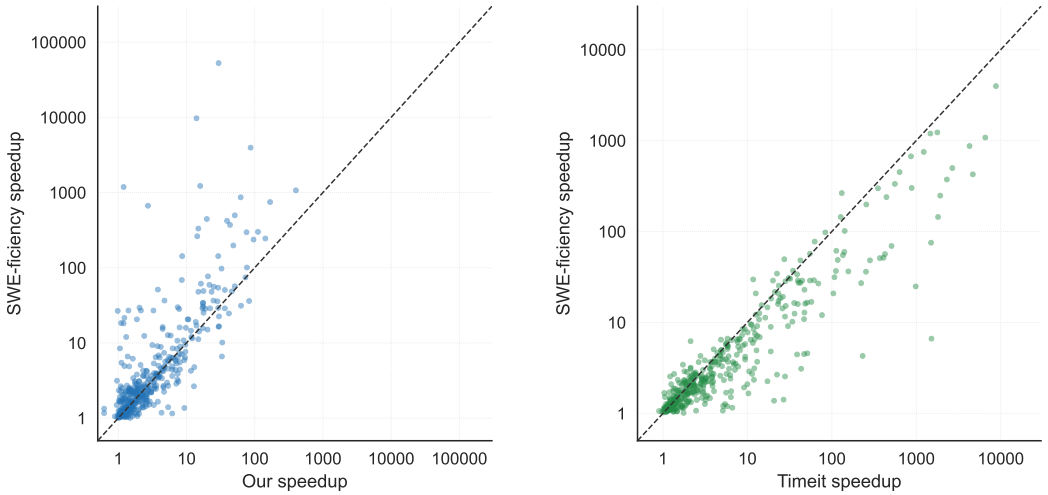
`Sol` outperforms `Luna` on every displayed metric at both effort levels, so the model ordering is preserved. The gains are consistent with extra effort helping models integrate execution information across repository functions. We therefore use high reasoning effort in the main experiments to reduce the risk of conflating limited predictive capability with insufficient reasoning effort.

Table 12: Inference costs in USD on the stratified 200-instance cohorts. For each model and context condition, we report the average cost per instance (/dp), total cost per task (/task), and total across all six tasks. The bottom row aggregates costs across the evaluated configurations.

Groups (→)	**Test Status**				**Time**				**Memory**				**Total**
Models (↓) / Tasks (→)	*Single*		*Anchored*		*Pred.*		*Hotspots*		*Pred.*		*Hotspots*		
	/dp	*/task*	*/dp*	*/task*	*/dp*	*/task*	*/dp*	*/task*	*/dp*	*/task*	*/dp*	*/task*	
No context (Noop)													
GPT-5.6 Luna^{h}	$0.0045	$0.9	$0.021	$4.3	$0.0078	$1.6	$0.018	$3.7	$0.0098	$2	$0.019	$3.8	$16
GPT-5.6 Terra^{h}	$0.014	$2.7	$0.19	$39	$0.048	$9.6	$0.13	$27	$0.069	$14	$0.15	$30	$120
GPT-5.6 Sol^{h}	$0.043	$8.5	$0.31	$62	$0.11	$22	$0.29	$58	$0.17	$35	$0.28	$56	$240
Claude Opus 5^{h}	$0.14	$29	$0.73	$150	$0.062	$12	$0.61	$120	$0.14	$28	$0.5	$100	$440
Qwen3.8-27b^{h}	$0.013	$2.5	$0.034	$6.9	$0.0092	$1.8	$0.023	$4.7	$0.016	$3.1	$0.023	$4.6	$24
GLM-5.3^{m,h}	$0.021	$4.2	$0.097	$19	$0.013	$2.5	$0.015	$2.9	$0.008	$1.6	$0.012	$2.3	$33
Sparse retrieval (BM25)													
GPT-5.6 Luna^{h}	$0.0097	$1.9	$0.033	$6.5	$0.015	$3	$0.026	$5.1	$0.013	$2.7	$0.024	$4.8	$24
GPT-5.6 Terra^{h}	$0.08	$16	$0.3	$60	$0.12	$23	$0.21	$41	$0.12	$23	$0.19	$38	$200
GPT-5.6 Sol^{h}	$0.14	$28	$0.5	$100	$0.23	$45	$0.39	$78	$0.25	$49	$0.37	$73	$370
Claude Opus 5^{h}	$0.36	$73	$0.96	$190	$0.33	$65	$0.73	$150	$0.37	$74	$0.66	$130	$680
Qwen3.8-27b^{h}	$0.013	$2.6	$0.039	$7.8	$0.017	$3.4	$0.032	$6.4	$0.023	$4.6	$0.031	$6.2	$31
GLM-5.3^{m,h}	$0.056	$11	$0.15	$30	$0.053	$11	$0.056	$11	$0.037	$7.4	$0.051	$10	$81
End-to-end agents													
`Codex` GPT-5.6 Luna	$0.019	$3.9	$0.033	$6.6	$0.0037	$0.73	$0.0075	$1.5	$0.0035	$0.71	$0.0054	$1.1	$15
`Codex` GPT-5.6 Sol	$0.097	$19	$0.21	$41	$0.081	$16	$0.32	$65	$0.087	$17	$0.3	$60	$220
`mini-SWE` GPT-5.6 Luna	$0.0037	$0.74	$0.012	$2.4	$0.0032	$0.63	$0.0082	$1.6	$0.0037	$0.74	$0.0059	$1.2	$7.4
`mini-SWE` GPT-5.6 Sol	$0.062	$12	$0.15	$29	$0.068	$14	$0.2	$40	$0.074	$15	$0.15	$30	$140
`mini-SWE` GLM-5.3	$0.021	$4.2	$0.13	$26	$0.0071	$1.4	$0.11	$21	$0.037	$7.3	$0.081	$16	$76
Execution trace oracle													
GPT-5.6 Luna^{h}	$0.013	$2.6	$0.027	$5.5	$0.021	$4.1	$0.029	$5.9	$0.021	$4.2	$0.029	$5.7	$28
GPT-5.6 Terra^{h}	$0.12	$23	$0.25	$50	$0.18	$36	$0.24	$48	$0.18	$35	$0.23	$46	$240
GPT-5.6 Sol^{h}	$0.23	$46	$0.5	$99	$0.37	$74	$0.51	$100	$0.37	$73	$0.45	$90	$490
Claude Opus 5^{h}	$0.56	$110	$0.93	$190	$0.63	$130	$0.93	$190	$0.69	$140	$0.85	$170	$920
Qwen3.8-27b^{h}	$0.021	$4.2	$0.039	$7.8	$0.03	$6.1	$0.045	$9	$0.033	$6.5	$0.041	$8.2	$42
GLM-5.3^{m,h}	$0.083	$17	$0.15	$30	$0.11	$22	$0.1	$21	$0.093	$19	$0.11	$22	$130
Total	—	$430	—	$1200	—	$500	—	$1000	—	$560	—	$910	$4600

G Evaluation Cost

Table 12 reports inference costs for each model–context configuration across the six benchmark tasks, including the average cost per instance, total cost per task, and aggregate cost across tasks. Evaluating the complete matrix of models and context conditions is costly even on the stratified 200-instance cohorts. Running the same experiments on the full task datasets would require approximately three to four times as many instances, with a corresponding increase in inference cost if average per-instance costs remain similar. These costs motivate further research of test-time scaling for software behavior prediction: how best to allocate additional inference budget between repository exploration and reasoning to improve predictive accuracy.



(a) Speedup measured by our tracer implementation vs SWE-fficicency.

(b) Speedup measured by our tracer implementation with timeit vs SWE-fficicency.

Figure 9: Comparison of workload speedups measured by our tooling vs results reported in the SWE-fficicency paper.

H Speedup comparison

Our profiling methodology differs from that used in SWE-fficiency (Ma et al., 2025) and accounts also for setup time needed for workloads. For that reason we validate our approach by comparing the speedups computed by our profiler with those reported by SWE-fficiency. Figure 9a compares the SWE-fficiency speedups with those obtained for our wrapped workloads. The results show a strong correlation, with Spearman's $\rho = 0.79$.

Additionally, we repeat the measurements reported by SWE-fficiency on our own machine (Figure 9b). These measurements show even stronger correlation, with Spearman's $\rho = 0.90$.

I Group-Based Sampling: Further Details

I.1 Single Test Status

For the Single Test Status task, we sample one test and two prediction states from each group. We describe the sampling procedure for both stages below.

First, we sample a single test per group. We restrict the selection to tests whose status is `PASSED` in the base state. For such tests, the flip score is defined as the proportion of states in the group in which the status of the given test changes from `PASSED` in the base state to any other in-scope status after a patch is applied. If all considered tests have a flip score of zero, we sample a test uniformly. Otherwise, we sample a test with probability equal to its flip score divided by the total sum of all flip scores. In this way, the flip score serves as a proxy for the informativeness of a test with respect to the prediction states in the group. This also prevents us from sampling tests with no variability whenever possible.

Second, we divide the prediction states in the group into two lists according to the status of the selected test. If both lists are empty, the group is skipped. If one of the lists is empty, we sample both states from the other list. Otherwise, sampling is done uniformly from each list. In this way, we obtain one state in which the selected test has `PASSED` status and another in which it has `NOT PASSED` status whenever possible.

Table 13: **Overlap with Oracle context for BM25 and mini-SWE-agent baselines on SourceWorldBench Lite tasks.** With B denoting a set of files/function in baseline context, and O — in Oracle context, (1) Precision is $\frac{|B \cap O|}{|B|}$; (2) Recall is $\frac{|B \cap O|}{|O|}$; (3) Jaccard is $\frac{|B \cap O|}{|B \cup O|}$. Two Time tasks share instances and Oracle context, so we report a single overlap metric. Same with Memory tasks.

Tasks (→)	Single Test Status			Test Status Anchored			Time			Memory		
Scaffolds (↓) / Metrics (→)	Precision	Recall	Jaccard	Precision	Recall	Jaccard	Precision	Recall	Jaccard	Precision	Recall	Jaccard
File-level												
BM25	12.4	20.8	6.7	39.3	10.0	8.6	7.1	7.5	3.6	5.1	4.8	2.4
mini-SWE GPT-5.6 Sol	63.1	23.5	18.1	85.1	4.7	4.6	52.0	8.0	7.0	68.0	8.1	7.4
mini-SWE GLM-5.3	50.4	16.3	10.8	75.4	5.7	5.4	65.6	3.9	3.5	68.4	5.6	4.7
Function-level												
BM25	1.6	4.4	0.7	10.6	1.0	0.9	0.7	0.4	0.2	0.7	0.4	0.2
mini-SWE GPT-5.6 Sol	19.0	7.2	3.9	27.0	0.4	0.4	28.9	2.2	2.0	31.4	2.2	2.0
mini-SWE GLM-5.3	25.6	3.4	2.4	28.6	0.5	0.5	38.4	1.0	1.0	36.7	1.7	1.5

I.2 Test Status Anchored

For the Test Status Anchored task, we sample up to 100 tests for each prediction state and two prediction states per group. Similarly to the Single Test Status task, we restrict sampling to tests whose status is `PASSED` in the base state and calculate their flip scores. If a test has an original flip score of zero, we assign a small positive value to it instead. If the number of tests with in-scope statuses is at most 100, all of them are selected. Otherwise, we sample 100 tests for each prediction state without replacement, with probabilities proportional to their flip scores. We then sample two prediction states uniformly from the group.

I.3 Resource Estimation and Bottleneck Localization

From each group, we select two distinct prediction states for the resource estimation and bottleneck localization datasets. We measure the wall-clock time of the base state t_{base}, the fixed state t_{fixed}, and each of the n prediction states $\{t_i\}_{i=1}^{n}$ on the group's workload. Groups with at most two prediction states require no sampling, so we retain all available prediction states. For groups with $n > 2$, we define a probability distribution over pairs of prediction states based on their execution-time proximity to the base–fixed pair.

We quantify the proximity between two execution times with a Laplacian kernel in log space,

$$k(x, y) = \exp\left(-|\log x - \log y|\right),$$

and evaluate it for every base–prediction pair $k(t_{\text{base}}, t_i)$ and every fixed–prediction pair $k(t_{\text{fixed}}, t_i)$. Using a logarithmic scale ensures scale invariance across groups and is consistent with the debiased MAE used to evaluate resource estimates (Section 4.2).

We then form the $n(n-1)$ ordered pairs of distinct prediction states and assign each pair a weight based on its joint proximity to the base and fixed states,

$$w(t_i, t_j) = k(t_{\text{base}}, t_i) \cdot k(t_{\text{fixed}}, t_j).$$

Normalizing the weights by their total sum yields a probability distribution over all prediction-state pairs, from which we sample a single pair per group.

We apply the described procedure only to the construction of the time prediction dataset. The Time Hotspots and Memory datasets use the same prediction states to maintain consistency and reduce the cost of memory profiling of unsampled states.

Table 14: Correlations between overlap with Oracle (precision) and downstream task quality (metrics from Table 1) for practical baselines. We use the rank-biserial correlation for Test Status (Single and Anchored) and the Spearman correlation for the remaining tasks. **Bold values** indicate statistically significant associations after global Benjamini–Hochberg correction ($q < 0.05$).

Groups (→)	**Test Status**		**Time**		**Memory**	
Configurations (↓) / Tasks (→)	*Single*	*Anchored*	*Prediction*	*Hotspots*	*Prediction*	*Hotspots*
File-level						
BM25 + GPT-5.6 Luna	0.003	**0.291**	0.006	0.002	-0.112	0.015
BM25 + GPT-5.6 Terra	0.007	0.202	0.047	0.050	-0.200	0.012
BM25 + GPT-5.6 Sol	-0.001	**0.301**	0.049	0.038	**0.249**	0.019
BM25 + Claude Opus 5	-0.013	**0.262**	0.081	0.021	0.176	-0.080
BM25 + Qwen3.8-27B	-0.020	**0.270**	0.032	0.020	0.080	-0.009
BM25 + GLM-5.3	-0.055	0.170	0.001	0.030	0.069	0.106
mini-SWE GPT-5.6 Sol	0.063	-0.001	-0.083	-0.134	0.067	-0.104
mini-SWE GLM-5.3	0.008	0.095	0.027	-0.165	-0.022	-0.137
Function-level						
BM25 + GPT-5.6 Luna	0.142	-0.061	0.033	0.112	-0.111	-0.044
BM25 + GPT-5.6 Terra	0.154	-0.118	0.081	0.183	-0.006	-0.150
BM25 + GPT-5.6 Sol	0.129	-0.079	0.099	0.164	0.158	-0.136
BM25 + Claude Opus 5	0.128	-0.131	0.003	0.181	0.084	**-0.269**
BM25 + Qwen3.8-27B	0.065	-0.177	0.017	0.166	-0.101	0.064
BM25 + GLM-5.3	0.054	-0.139	-0.030	0.179	-0.075	0.018
mini-SWE GPT-5.6 Sol	0.048	0.181	-0.008	**0.253**	0.137	-0.035
mini-SWE GLM-5.3	-0.028	0.176	0.090	-0.022	-0.001	-0.035

J Do Practical Baselines Overlap with Oracle, and Does It Matter

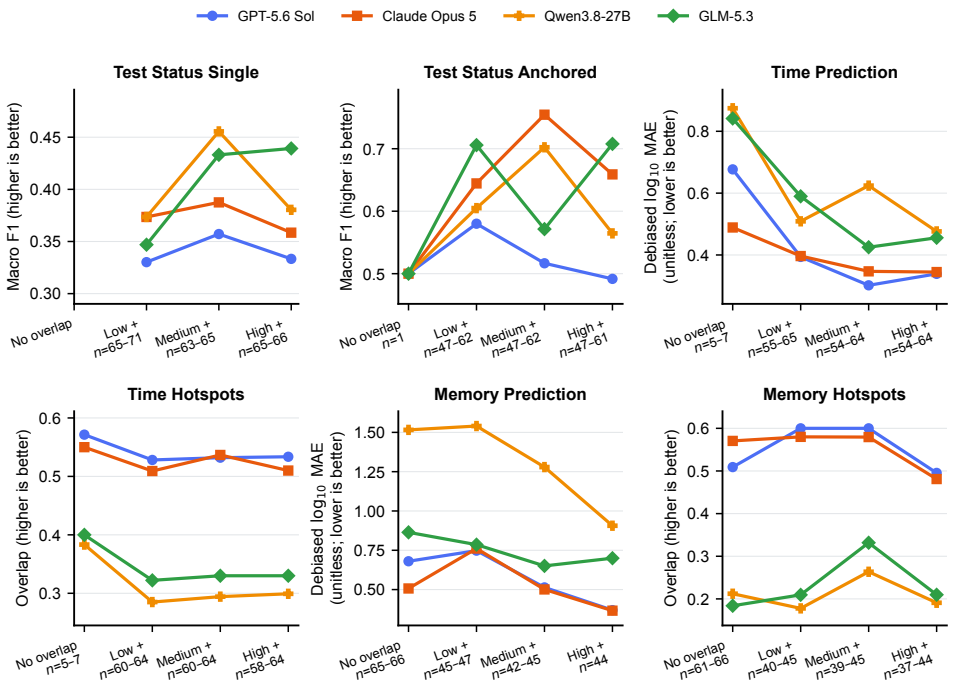
To assess which context practical baselines retrieve (BM25) or observe (agents), we study their file-level and function-level overlap with Execution Trace Oracle context. For each task instance, let R denote a set of files or functions in retrieved (BM25) or observed (agents) context, and O — a set of files or functions in Oracle context. We consider three overlap metrics: (1) Precision is $\frac{|R \cap O|}{|R|}$; (2) Recall is $\frac{|R \cap O|}{|O|}$; (3) Jaccard is $\frac{|R \cap O|}{|R \cup O|}$. We report the metrics in Table 13. Overall, BM25 shows low overlap with Oracle context at both the file and function levels. For agents, function-level Precision remains relatively low (from 19.0 for Single Test Status to 38.4 for Time), but file-level Precision is substantially higher (up to 85.1 for Test Status Anchored). Recall and Jaccard are usually lower than precision; the only exception is BM25 on Single Test Status.

To study whether overlap with Oracle context can be a useful proxy for context usefulness for downstream tasks, we report the correlations between Precision and the downstream metrics in Table 14. The results are mixed. For BM25, there is a statistically significant positive correlation with file-level Precision for the Test Status Anchored and Memory Prediction tasks, but in other tasks and for function-level overlap in all the tasks there is no significant positive association. Similarly, agents display only a single positive correlation: function-level Precision for GPT-5.6 Sol on Time Hotspots Predictions tasks.

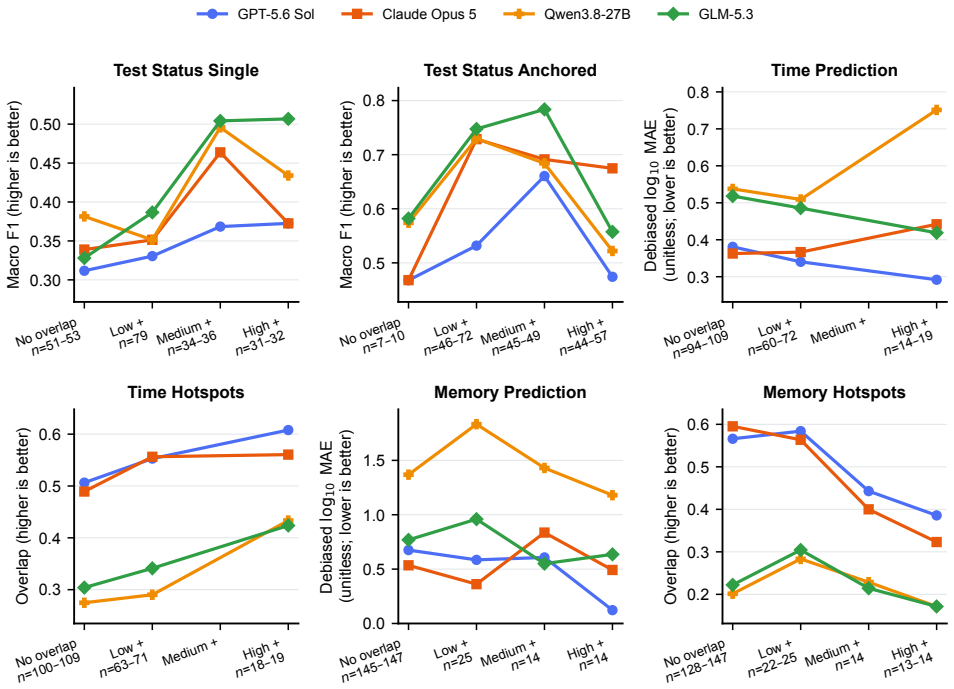
We also visualize the relationship between overlap and downstream quality by stratifying datapoints based on Precision inFigure 10a (file-level), and Figure 10b (function-level). The resulting plots show that dynamics differ substantially across models and tasks.

K Why More Context Can Hurt Memory Prediction

Memory Prediction is an exception to the otherwise generally positive effect of richer context on SourceWorldBench Lite. As shown in Section M, most models have higher Memory Prediction error with BM25 or Oracle context than with No-Retrieval.



(a) File-level overlap.



(b) Function-level overlap.

Figure 10: **BM25 downstream task quality stratified by overlap with Oracle context (Precision).** Panels show (a) file-level and (b) function-level overlap. Colors and markers distinguish models. Strata are constructed separately for each model–task pair using per-datapoint Precision at the corresponding granularity; all zero-precision datapoints form a "No overlap" stratum, while positive-precision datapoints are divided into up to three approximately equal-sized rank-based strata without splitting tied values. The main task metric from Table 1 is then recomputed within each stratum.

Table 15: Blinded single-annotator validation of the exclusion-language detector. The left panel gives confusion counts in the stratified annotation sample; the right panel gives population-weighted estimates with approximate 95% stratified-bootstrap intervals.

Context	TP	FP	FN	TN
No-Retrieval	17	3	1	19
BM25	20	0	3	17
Oracle	20	0	2	18
Total	57	3	6	54

Measure	Estimate	95% interval
Precision	97.4%	94.9–100.0%
Sensitivity	92.5%	86.7–97.8%
Specificity	97.0%	93.8–100.0%
Accuracy	94.5%	90.6–97.8%

Table 16: Counts of Memory Prediction rationales containing exclusion language for GPT-5.6 Sol under all context collection baselines.

Baseline	Rationales with exclusion language
No-Retrieval	54/200 (27.0%)
BM25	122/200 (61.0%)
Oracle	140/200 (70.0%)

We qualitatively study the rationales produced by GPT-5.6 Sol and identify potential confusion about memory-tracing semantics, particularly categorical statements that large NumPy or native buffers are excluded, even though such buffers can be visible to the tracer. The measurement semantics appear twice in our prompts (Prompt 7): first in the detailed profiling protocol before the additional context, and again in the task instructions after the context. We therefore investigate (1) the frequency of exclusion language under all context-collection baselines and (2) the association between exclusion language and observed error, measured using debiased $\log_{10}$ MAE.

Exclusion language detection. Following initial inspection, we devised a set of simple static rules to detect exclusion language. A rationale is positive only when the same sentence contains (i) an array or buffer term and (ii) explicit exclusion language such as "not counted", "outside the Python allocator", or "invisible to `tracemalloc`". We validated this detector on a sample of 20 detector-positive and 20 detector-negative rationales for each of the three context-collection baselines (120 total), annotated by one paper author who was blinded to condition and detector output (Table 15). Results on SourceWorldBench Lite in Table 16 show a substantial increase in rationales containing exclusion language for BM25 and Oracle compared with No-Retrieval.

Associations between exclusion language and observed errors. The detector-positive group has substantially higher observed error under every context-collection baseline (Table 17). Next, in Table 18, we partition matched pairs of the same instance under No-Retrieval and another baseline according to how the detector outcome changes. When the comparison baseline introduces exclusion language, debiased $\log_{10}$ MAE increases sharply. Removals move in the opposite direction, particularly for BM25, although these subgroups are small. However, the absent-in-both groups transitions also worsen slightly.

Possible interpretation. These observations suggest two possible contributors to the deterioration in Memory Prediction quality from No-Retrieval to the context-rich baselines: (1) the detailed profiling protocol appears before the additional context (Prompt 7), and (2) the additional context supplies more memory-allocation cues to which the exclusion rule can be applied. This exploratory analysis highlights an association; it does not establish that exclusion language caused the errors.

Table 17: Observed error conditional on detected exclusion language for GPT-5.6 Sol under all context-collection baselines. Debiasing uses the median signed error over all SourceWorldBench Lite Memory Prediction instances within each baseline; each group value averages the resulting per-instance absolute errors.

Baseline	Has Exclusion Language	n	Debiased $\log_{10}$ MAE
No-Retrieval	Yes	54	0.655
No-Retrieval	No	146	0.216
BM25	Yes	122	0.803
BM25	No	78	0.361
Oracle	Yes	140	0.683
Oracle	No	60	0.511

Table 18: Instance-matched exclusion-language transitions from No-Retrieval to other context collection baselines. With $e_c = \log_{10}(\widehat{y}_c/y)$ and $r_c = e_c - \widetilde{e}_c$, $\Delta|r|$ is baseline minus No-Retrieval and directly decomposes the change in the primary debiased log10 MAE. Brackets give paired 95% bootstrap intervals; each replicate resamples within transition groups and recomputes both condition-level medians.

Baseline Transition	Exclusion Language Transition	n	Δ debiased $\log_{10}$ MAE [95% CI]
No-Retrieval → BM25	Introduced	82	+0.744 [0.599, 0.881]
No-Retrieval → BM25	Absent in both	64	+0.130 [0.005, 0.283]
No-Retrieval → BM25	Present in both	40	−0.023 [−0.227, 0.168]
No-Retrieval → BM25	Removed	14	−0.658 [−1.154, −0.097]
No-Retrieval → Oracle	Introduced	92	+0.559 [0.440, 0.675]
No-Retrieval → Oracle	Absent in both	54	+0.274 [0.095, 0.467]
No-Retrieval → Oracle	Present in both	48	−0.094 [−0.277, 0.069]
No-Retrieval → Oracle	Removed	6	−0.400 [−0.868, 0.037]

L Core Dataset Statistics

The core dataset contains 11,940 repository states from 498 source tasks across nine Python repositories: 498 base states, 498 fixed states, and 10,944 prediction states. Table 20 provides a breakdown by repository.

The prediction states originate from trajectories generated with OpenHands, SWE-agent, and Cursor CLI. Table 19 provides a breakdown by agent framework and model.

M Resource Estimation Calibration Scatter Plots

In this section, we provide additional plots comparing predicted and measured runtime and peak memory on $\log_{10}$ axes for Time and Memory Prediction tasks.

Figure 12 presents plots for fixed-context baselines. A fitted slope $s < 1$ means that the predictions vary less than the measurements: low-cost and high-cost workloads receive estimates that are too similar. In these plots, this pattern often appears as overprediction at the low end and underprediction at the high end. Oracle context drastically improves Time Prediction, consistent with lower debiased $\log_{10}$ MAE across all models in Table 1. For GPT-5.6 Sol, for example, the error falls from 0.418 with No-Retrieval to 0.221 with Oracle, a 47% reduction. Memory Prediction does not show the same benefit: Oracle lowers the error relative to No-Retrieval for only one of the six models. For GPT-5.6 Sol, the error rises from 0.335 to 0.631, and the slope falls from 0.727 to 0.562. The memory plots also show frequent underprediction at the high end of the measured range. We examine this exception further in Appendix K.

The read-only agent results (Figure 11) are heterogeneous and depend on both the specific agentic harness used and the resource being estimated. Codex's memory predictions vary

Table 19: Agent frameworks and models that generated the prediction states in the core dataset.

Agent framework	Model	Prediction states
OpenHands	Claude 3.7 Sonnet	473
	Claude Opus 4.1	479
	Claude Opus 4.5	478
	Claude Sonnet 4.5	464
	Claude Opus 4.6	476
	DeepSeek-V3.1	868
	Gemini 2.5 Flash	457
	Gemini 2.5 Pro	446
	Gemini 3 Flash Preview	469
	Gemini 3 Pro Preview	480
	GLM-4.6	463
	GLM-5	478
	GPT-5	474
	GPT-5 mini	448
	GPT-5.1	474
	GPT-5.2	478
	Kimi K2	419
	MiniMax-M2	282
	Qwen3-Coder-Plus	480
SWE-agent	Claude 3.7 Sonnet	479
	Gemini 2.5 Flash	465
	GPT-5 mini	436
Cursor CLI	Composer 1	478
Total		**10,944**

Table 20: Composition of the core dataset.

Repository	Source tasks	Base	Fixed	Prediction	Total
pandas-dev/pandas	279	279	279	6,049	6,607
scipy/scipy	45	45	45	964	1,054
sympy/sympy	39	39	39	850	928
astropy/astropy	35	35	35	830	900
scikit-learn/scikit-learn	27	27	27	577	631
numpy/numpy	23	23	23	524	570
matplotlib/matplotlib	20	20	20	470	510
dask/dask	19	19	19	439	477
pydata/xarray	11	11	11	241	263
Total	498	498	498	10,944	11,940

particularly little with measured memory: the fitted slopes are 0.03 with GPT-5.6 Luna and 0.23 with Sol. The mini-SWE time predictions track changes in measured runtime closely (slopes from 0.7 with GPT-5.6 Luna to 0.86 with Sol), but struggle more with memory predictions. This pattern is also associated with worse observed errors: debiased $\log_{10}$ MAE for agents ranges from 0.31 to 0.51 for time and from 0.62 to 1.3 for memory, generally exceeding the errors with the same backbones under the fixed-context baseline. Thus, agent runs do not merely exhibit a uniform scale offset; their shallow fitted slopes also indicate limited sensitivity to differences in measured workload costs, especially in terms of memory.

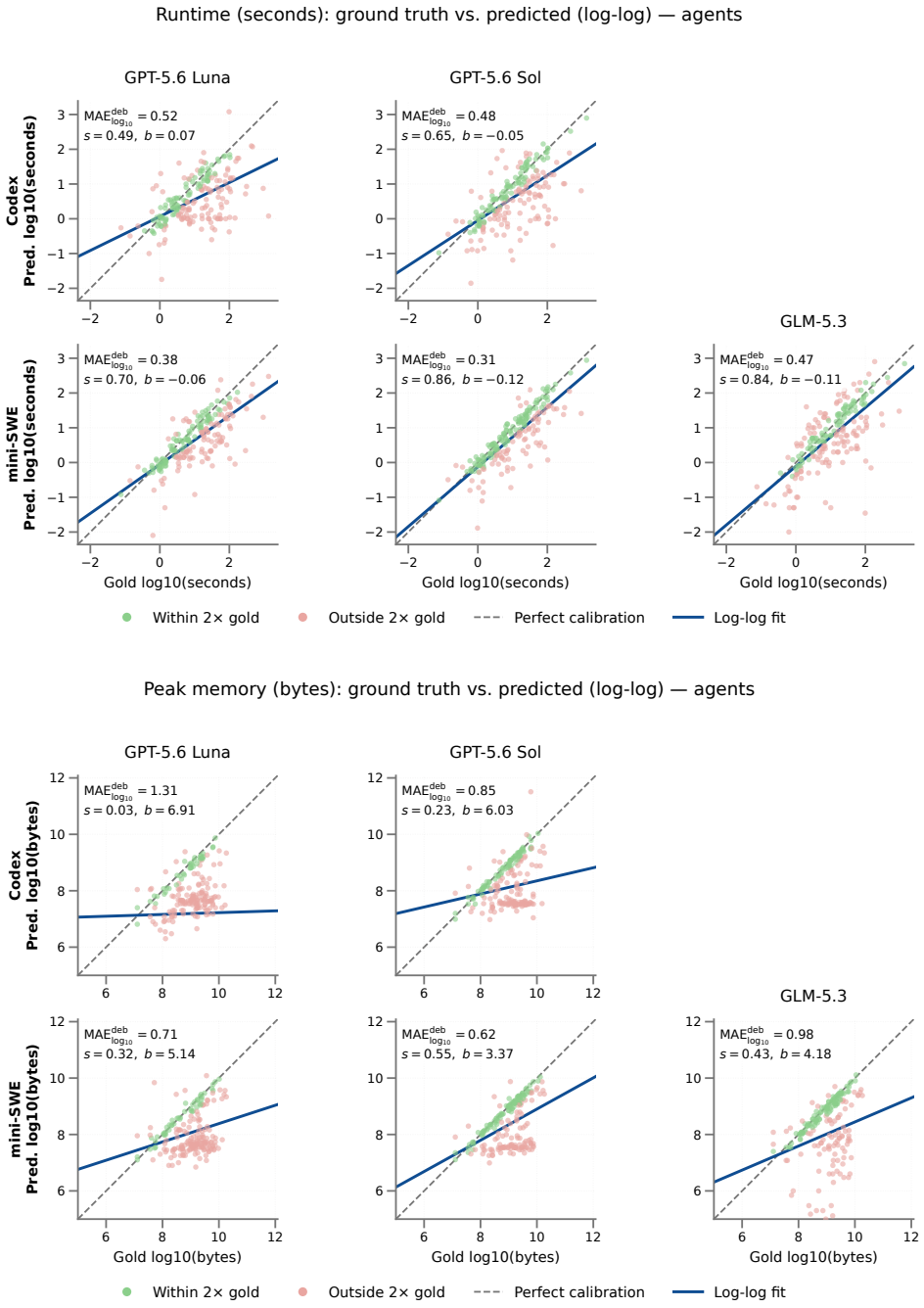


Figure 11: **Resource estimation with read-only agents.** The dashed diagonal denotes exact predictions, while the solid blue line is an ordinary least-squares fit of predicted log value on measured log value. The green points are within a factor of two of the measurement; the pink points are outside that range. The points and colors use the original, unadjusted predictions. Only the displayed debiased $\log_{10}$ MAE removes the median log-ratio offset.

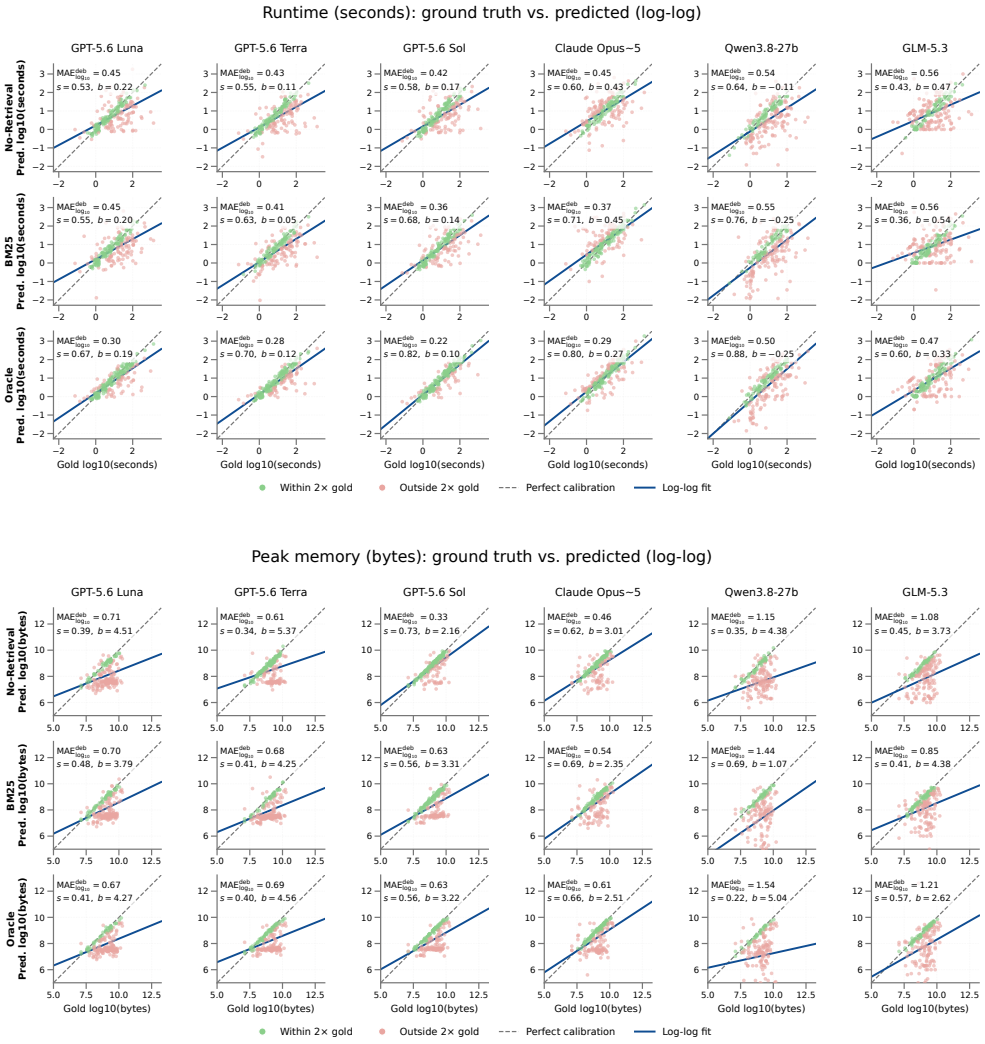


Figure 12: **Resource estimates under fixed-context baselines.** Notation as in 11.

N Memory Workload Amplification

To assess workload amplification, we generate three variants of each original workload: *light*, *medium*, and *heavy*. We compare their memory consumption in the base and fixed states of 326 source tasks for which measurements succeed at all three scales.

The *light* variant retains the original input sizes with timing repetitions removed. The *medium* and *heavy* variants use larger inputs while preserving the original workload operations and calls to repository code. For these variants, the prompt asks the LLM to target approximately 10× and 100× the light variant's peak memory, respectively. Each variant is executed once per state, with all three variants collected in a single `pytest` invocation for that state.

For each prediction state in the task-specific memory datasets, we select the largest variant that completes successfully within the timeout and without running out of memory, preferring *heavy* over *medium* over *light*.

We use `claude-sonnet-5` with the templates in Prompts 1, 2, and 3.

Prompt 1: Amplifying an existing workload

```
You are given a performance-benchmark workload from the SWE-fficiency
dataset: a single pytest test that builds inputs in `setup()` and times a
`workload()` call with `timeit.repeat`. We re-run these workloads under a
memory profiler to measure how a performance patch changes memory
consumption. Small inputs often hide the memory effect, so we want the same
workload at three input scales.

Rewrite the file so it contains exactly three tests, in this order:

1. `test_workload_light` | the original workload verbatim (same inputs, same
logic).
2. `test_workload_medium` | identical logic, input sizes scaled so the
workload's peak memory is roughly 10x the light version, or at least ~10 MB
of workload data, whichever is larger.
3. `test_workload_heavy` | identical logic, input sizes scaled so peak memory
is roughly 100x the light version, or at least ~100 MB of workload data,
whichever is larger.

The absolute floors matter: if the light workload only touches a few KB (tiny
arrays), a literal 10x/100x is still tiny and useless for memory profiling |
scale far enough to reach the floors instead, and say so in the header
comment. The floors are subject to the same budget as everything else: if
reaching a floor would break the memory or runtime budget below, get as close
as the budget allows.

Rules:
- Three plain test functions, NOT parametrized, no fixtures, no imports of
anything the original didn't import (plus nothing outside the standard
library / the profiled package itself).
- Scale only the DATA: array lengths, row counts, grid resolutions,
n_samples, etc. Never change which functions are called or add new code
paths. Reason about complexity: if memory grows quadratically with a knob,
scale that knob by sqrt(10)/sqrt(100).
- Strip the timing harness: do NOT use `timeit` (we measure memory, not time,
and the peak is per-call). Each test builds its inputs | call `setup()` once
if the original defines one, keep inline setup inline | then calls the
workload function exactly once. Drop the runtime mean/stddev prints; keep the
trailing `assert True`.
- Memory budget (hard): every single test must stay under ~8 GB peak RSS.
Estimate from the data sizes (element count x itemsize), then be
conservative: temporary copies, format conversions, and intermediates
routinely make the true peak 3-5x the naive estimate, so keep the NAIVE
estimate under ~1.5 GB. If the nominal 100x heavy would exceed that, scale it
as far as the budget allows and say so in the header comment. Each test
should also plausibly finish within a few minutes on one CPU core.
- Shared helpers may be module-level, but each test must be runnable in
isolation and the three tests must not share mutable state (re-run setup per
test; if the original uses `global`, keep that pattern per test with distinct
global names or re-assignment).
- First line of the file must be a comment: `# amplification: <one-line note
on what was scaled and by how much>`.

If the workload has NO meaningful data-size knob (e.g. it measures a
fixed-size operation, import time, or a scalar API call), do NOT write any
code: reply with the single line `NOT_AMPLIFIABLE: <one-line reason>` and
nothing else.

Otherwise reply with ONE fenced python code block containing the complete
file, nothing else.

Original workload file:
```

````
```python
{workload}
```
````

Prompt 2: Generating a workload from a patch

````
Here is a performance-optimization diff from the {repo} repository. We
measure how such patches change MEMORY consumption: the same workload runs
under a memory profiler on the code before and after the patch, and we
compare peaks. No existing benchmark exercises this patch at meaningful data
sizes, so write one from scratch.

Write a single self-contained pytest file with exactly three tests, in this
order: `test_workload_light`, `test_workload_medium`, `test_workload_heavy`.
All three run the SAME code | a realistic call into the public API that
reaches the code changed in the diff | at three input scales:

1. light | small, fast sanity scale (sub-second, a few MB at most).
2. medium | inputs sized so the workload allocates at least ~10 MB.
3. heavy | inputs sized so the workload allocates at least ~100 MB.

Rules:
- Three plain test functions, NOT parametrized, no fixtures. Import only the
standard library and the profiled package (plus numpy if the package's API
takes arrays).
- Each test builds its inputs deterministically (seed any randomness), then
calls the workload exactly once. No timeit, no timing prints. End each test
with `assert True`.
- The workload MUST execute the functions changed in the diff | reach them
through the public API a real user would call, and say in the header which
changed function(s) the workload reaches and how.
- The three tests must differ only in input sizes. Reason about complexity:
if memory grows quadratically with a knob, scale the knob by the square root.
- Memory budget (hard): every single test must stay under ~8 GB peak RSS.
Estimate from data sizes, then be conservative: temporary copies and
intermediates routinely make the true peak 3-5x the naive estimate, so keep
the NAIVE estimate under ~1.5 GB. Each test should finish within a few
minutes on one CPU core.
- Use only APIs that exist at this commit (the diff shows the code as it is
being changed | match its imports and call signatures, not today's versions).
- First line of the file must be a comment: `# from-scratch: <one line | what
the workload does, which changed functions it reaches, what is scaled>`.

If the changed code CANNOT have size-dependent memory behavior (e.g. it
operates on O(1) state regardless of input size), do NOT write code: reply
with the single line `NOT_POSSIBLE: <one-line reason>` and nothing else.

Otherwise reply with ONE fenced python code block containing the complete
file, nothing else.

The diff:

```diff
{patch}
```
````

Prompt 3: Retaining the original input scale

```
You are given a performance-benchmark workload from the SWE-fficiency
dataset: a single pytest test that builds inputs in `setup()` and times a
`workload()` call with `timeit.repeat`. We re-run these workloads under a
memory profiler; the timing harness only adds runtime (memory peaks are
per-call), so we want the same workload with it stripped.

Rewrite the file so it contains exactly one test, `test_workload_light`: the
original workload verbatim | same inputs, same logic, no scaling. Build the
inputs (call `setup()` once if the original defines one, keep inline setup
inline), then call the workload function exactly once. Do NOT use `timeit`;
drop the runtime mean/stddev prints; keep the trailing `assert True`. Import
nothing the original didn't import.

First line of the file must be a comment: `# light-only: <one-line note on
what the workload does>`.

Reply with ONE fenced python code block containing the complete file, nothing
else.

Original workload file:

```python
{workload}
```
```

O Benchmark Task Prompts

We reproduce the fixed instructions for all six benchmark tasks and show where provider output enters the fixed-context user prompts. We omit instance-specific inputs: repository and revision identifiers, source code, test and workload commands, patches, and candidate-function lists. We also omit provider- and harness-specific instructions. Ellipses replace retrieved context, while the CWM delimiter lines come from the user-prompt templates. Bracketed headings are editorial; the captions explain how each baseline supplies or gathers repository context. For the two memory tasks, we show all three question variants (`heavy`, `medium`, and `light`), which otherwise share the same instructions.

Prompt 4: Single Test Status

```
[SYSTEM INSTRUCTION]
You predict one explicit test's outcome in a single evaluated repository
state.
Classify the observed behavior using exactly one of: PASSED, NOT_PASSED.

Reason carefully, then follow the structured-output contract exactly.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

[QUESTION]
Assign the test one of two labels: 'PASSED' if it completes successfully, or
'NOT_PASSED' if it fails or errors.

[OUTPUT INSTRUCTION]
Return a rationale and exactly one outcome allowed by the structured-output
contract.
```

Context delivery. User message: after the selected test and source, before the question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.

Prompt 5: Test Status Anchored

```
[SYSTEM INSTRUCTION]
You predict selected test outcomes in a candidate repository state. You
receive reference-state
test sources and the reference-to-candidate change. Infer each selected
test's outcome in the
candidate state only; never report reference-state labels.

Reason carefully, then follow the requested structured-output contract
exactly.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

[CANDIDATE-STATE REMINDER]
All labels in your answer must describe the candidate state after the change,
never the reference
state.

[QUESTION]
After the provided patch is applied at the base commit, assign each of the
provided tests one of two labels: 'PASSED' if it completes successfully, or
'NOT_PASSED' if it fails or errors.

[OUTPUT INSTRUCTION]
Return one outcomes entry for every unit_id, using each unit_id exactly once.
Allowed outcomes are PASSED, NOT_PASSED.
```

Context delivery. User message: after selected tests, patch, and sources, before the candidate-state reminder and question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.

Prompt 6: Time Prediction

```
[SYSTEM INSTRUCTION]
Your task is to predict the cProfile-derived wall-clock runtime of a given workload script from a
given Python repository.
Ground-truth timings measure Python execution via cProfile and exclude shell or environment setup
overhead that is not part of the profiled workload.

[GROUND-TRUTH PROFILING PROTOCOL]
The ground truth profiling works as follows:

- The clock is read immediately before the test function is called and immediately after it
  returns. The reported time is the interval between the two readings.
- This is elapsed time, not CPU time, so sleeps, disk and network I/O, and waits on worker
  processes or threads all count.
- The interval covers everything the test function does, including its own setup and every
  repeat if it measures an operation repeatedly. It does not cover process startup, plugin
  loading, test collection, fixtures, or environment setup before the run.
- The test function runs under an active cProfile profiler, and the profiler's per-call
  overhead is part of the measured time. That overhead inflates call-heavy pure-Python code by
  a few percent. It does not affect time spent inside compiled extension code.

[PROFILING HARDWARE]
The code is executed using the following hardware:

- The run happens in a container on a Google Kubernetes Engine node of Google Cloud machine
  type c2d-standard-16: the C2D compute-optimized machine family, 16 vCPUs and 64 GB of memory
  per node, 4 GB of memory per vCPU.
- The CPU is a 3rd generation AMD EPYC Milan (AMD EPYC 7B13), x86-64, vendor AuthenticAMD,
  family 25, model 1.
- Google lists its base frequency as 2.45 GHz, its effective all-core frequency as 2.8 GHz,
  and its max boost frequency as 3.5 GHz. Inside the container /proc/cpuinfo reports 3050 MHz.
- Simultaneous multithreading is enabled. One vCPU is one hardware thread, and a core runs two
  threads.
- C2D instances have a fixed virtual-to-physical core mapping and expose the NUMA cell
```

```
  architecture to the guest OS.
- The container has a CPU limit of 8 vCPUs (8000 millicores), equal to its CPU request, and
  32 GiB of memory, also equal to its request. os.cpu_count() reports 8 inside it. Ephemeral
  storage is capped at 20 GiB.
- The container runs under the gVisor sandbox. The kernel it reports is Linux 4.4.0 x86_64 and
  the C library is glibc 2.35.
- 77 CPU feature flags are visible inside the sandbox.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

[QUESTION]
The provided workload is a single test, `test_workload.py::test_workload`, profiled with
`cProfile`. Predict the total elapsed time in seconds of that test function | the whole call, its
setup and every repeat included.

[OUTPUT INSTRUCTION]
Return `seconds` as one positive finite numeric runtime estimate.
```

Context delivery. User message: after workload inputs and profiling details, before the question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.

Prompt 7: Memory Prediction

```
[SYSTEM INSTRUCTION]
Your task is to predict the memory measurement requested by the supplied dataset question for a
workload from a given Python repository.
Preserve the measurement semantics stated in the question; do not substitute a different memory
definition.

[GROUND-TRUTH PROFILING PROTOCOL]
The ground truth profiling works as follows:

- Each row's selected workload scale runs once in a fresh Python process. Measurement covers
pytest's test-call phase,
  including the imports, input construction and workload operation inside the test function.
- The measured memory increase is the highest traced memory reached during the test minus the
  traced memory at its start. This includes Python allocations and native buffers explicitly
  registered with `tracemalloc` by extension libraries.
- The peak includes temporary allocations held during this execution, including memory freed before
  the test returns.
- Heap readings cover all threads in the test process.
- The function-level memory profiler is active during the measurement. Its Python bookkeeping
  allocations remain part of the recorded values.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

[QUESTION: HEAVY VARIANT]
The provided workload is a single test, `test_workload.py::test_workload_heavy`. Predict the peak
increase in memory in bytes of that test function | the whole call and its setup included. The peak
increase is the highest level reached during the test minus the level at test start. Memory usage
means the additional memory the test allocates on the Python heap above the level in use when it
started, measured with tracemalloc: the high-water mark during the test minus the level at test
start. It counts objects allocated through the Python allocator; it does not count memory that C
extensions allocate outside the Python allocator, such as large NumPy or BLAS buffers. Memory
allocated and freed again during the test counts towards the peak: a loop that allocates and frees
80 MB a hundred times peaks at about 80 MB.

[QUESTION: MEDIUM VARIANT]
The provided workload is a single test, `test_workload.py::test_workload_medium`. Predict the peak
increase in memory in bytes of that test function | the whole call and its setup included. The peak
increase is the highest level reached during the test minus the level at test start. Memory usage
means the additional memory the test allocates on the Python heap above the level in use when it
started, measured with tracemalloc: the high-water mark during the test minus the level at test
start. It counts objects allocated through the Python allocator; it does not count memory that C
extensions allocate outside the Python allocator, such as large NumPy or BLAS buffers. Memory
allocated and freed again during the test counts towards the peak: a loop that allocates and frees
80 MB a hundred times peaks at about 80 MB.

[QUESTION: LIGHT VARIANT]
```

```
The provided workload is a single test, `test_workload.py::test_workload_light`. Predict the peak
increase in memory in bytes of that test function | the whole call and its setup included. The peak
increase is the highest level reached during the test minus the level at test start. Memory usage
means the additional memory the test allocates on the Python heap above the level in use when it
started, measured with tracemalloc: the high-water mark during the test minus the level at test
start. It counts objects allocated through the Python allocator; it does not count memory that C
extensions allocate outside the Python allocator, such as large NumPy or BLAS buffers. Memory
allocated and freed again during the test counts towards the peak: a loop that allocates and frees
80 MB a hundred times peaks at about 80 MB.

[OUTPUT INSTRUCTION]
Return `bytes` as one positive finite numeric memory estimate in bytes.
```

Context delivery. User message: after workload inputs and profiling details, before the question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.

Prompt 8: Time Hotspots

```
[SYSTEM INSTRUCTION]
Your task is to inspect a given workload script in a given repository and determine hotspot
functions,
using the runtime metric named in the question.

[GROUND-TRUTH PROFILING PROTOCOL]
The ground truth profiling works as follows:

- The test function runs under an active cProfile profiler, enabled immediately before the test
  function is called and disabled immediately after it returns.
- The profiled interval covers everything the test function does, including its own setup and
  every repeat if it measures an operation repeatedly. It does not cover process startup,
  plugin loading, test collection, fixtures, or environment setup before the run.
- Times are elapsed time, not CPU time, so sleeps, disk and network I/O, and waits on worker
  processes or threads all count. A wait is charged to the function that waits.
- Only the thread that enabled the profiler is traced, so work in other threads and in child
  processes is invisible.
- Only functions defined in `.py` files inside the repository checkout are reported. This
  excludes the standard library, installed packages, and the workload file itself, and includes
  vendored third-party code inside the repository tree.
- A function is identified by its path relative to the repository root, by the first line of
  its definition, which for a decorated function is the line of its first decorator, and by its
  qualified name, which includes the enclosing class on Python 3.11 and later and is the bare
  function name on earlier versions. Comprehensions, generator expressions and lambdas are
  separate entries named `<listcomp>`, `<genexpr>` and `<lambda>`.
- The exclusive (self) time of a function is its total time minus the time of the calls the
  profiler was notified about. It is notified when a Python function is entered, and when a
  built-in function or a method of a C type is called by name. Everything else reaches compiled
  code silently, including operators, indexing, iteration, context managers, and calls to
  compiled callables that are not plain built-ins. Time spent that way stays in the self time
  of the Python function whose line triggered it, however long it lasts, so a short function
  can carry seconds of self time it did not spend on its own bytecode. Whether a given library
  call is silent depends on how that library is implemented, not on how the calling code is
  written, and the same operation spelled two ways can land on either side. Self times of the
  reported functions do not add up to the run time, because time subtracted into built-ins and
  installed packages is then dropped from the report.
- The profiler's per-call overhead is part of the measured time. It inflates code that makes
  many calls, whatever the callee, and does not inflate the interior of compiled code.

[PROFILING HARDWARE]
The code is executed using the following hardware:

- The run happens in a container on a Google Kubernetes Engine node of Google Cloud machine
  type c2d-standard-16: the C2D compute-optimized machine family, 16 vCPUs and 64 GB of memory
  per node, 4 GB of memory per vCPU.
- The CPU is a 3rd generation AMD EPYC Milan (AMD EPYC 7B13), x86-64, vendor AuthenticAMD,
  family 25, model 1.
- Google lists its base frequency as 2.45 GHz, its effective all-core frequency as 2.8 GHz,
  and its max boost frequency as 3.5 GHz. Inside the container /proc/cpuinfo reports 3050 MHz.
- Simultaneous multithreading is enabled. One vCPU is one hardware thread, and a core runs two
  threads.
- C2D instances have a fixed virtual-to-physical core mapping and expose the NUMA cell
  architecture to the guest OS.
- The container has a CPU limit of 8 vCPUs (8000 millicores), equal to its CPU request, and
  32 GiB of memory, also equal to its request. os.cpu_count() reports 8 inside it. Ephemeral
  storage is capped at 20 GiB.
- The container runs under the gVisor sandbox. The kernel it reports is Linux 4.4.0 x86_64 and
  the C library is glibc 2.35.
```

```
- 77 CPU feature flags are visible inside the sandbox.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

[QUESTION]
The provided workload is a single test, `test_workload.py::test_workload`, profiled with
`cProfile`. You are given a list of candidate functions from this repository that ran during that
call. List the 20 with the highest exclusive time, `cProfile's tottime`, the time in the function's
own frame excluding the calls the profiler recorded separately | most expensive first. Identify
each function by its file path, its first line number and its qualified name.

[OUTPUT INSTRUCTIONS]
Return exactly 20 functions in `functions`, ordered from highest to lowest profiler time. Each item
must use `file_path`, `firstlineno`, and `qualname`. Do not include duplicates.
Every returned locator must exactly match one listed candidate (`file_path`, `firstlineno`, and
`qualname`). Use the available static evidence only to rank the listed candidates; do not return
any locator that is not on the list.
```

Context delivery. User message: after workload inputs, candidate functions, and profiling details, before the question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.

Prompt 9: Memory Hotspots

```
[SYSTEM INSTRUCTION]
Your task is to inspect a workload in a repository and determine its Python memory-hotspot
functions using the allocation metric named verbatim in the question.

[GROUND-TRUTH PROFILING PROTOCOL]
The ground truth profiling works as follows:

- Each row's selected workload scale runs once in a fresh Python process.
- Profiling covers pytest's test-call phase, including the imports, input construction and workload
  operation inside the test function.
- Memory is measured with `tracemalloc`, which records Python allocations and native buffers
  explicitly registered by extension libraries.
- The candidate list contains recorded repository function frames. Module and class frames are
  filtered out after memory has been attributed. A function is identified by its
repository-relative
  file path, first line number and qualified name. On Python versions before 3.11, the name falls
  back to the bare function name.
- Both rankings use exclusive (self) memory: memory changes attributed to a function after
  subtracting contributions from other profiled functions it calls. Memory changes inside
  unprofiled library calls stay with the caller.
- The **peak ranking** orders functions by the largest peak increase in memory attributed to any
  one of their calls. Temporary allocations can contribute even if freed before the call returns.
  Different functions can reach their peaks at different times.
- The **cumulative ranking** orders functions by the sum of net memory increases attributed to
  their calls during the workload. Each call contributes the difference between memory at its
  return and at its start, after the same subtraction described above. Negative contributions
  count as zero. A call that allocates and frees memory can contribute little or nothing.
- Function events come from the thread that enables the profiler. Heap readings cover all threads
in
  that process, so concurrent allocations can affect the values attributed to the active frame.
- The values describe heap changes measured around calls, and include the profiler's Python
  bookkeeping allocations.

### <<< BEGIN CWM RETRIEVED CONTEXT >>>
...
### <<< END CWM RETRIEVED CONTEXT >>>

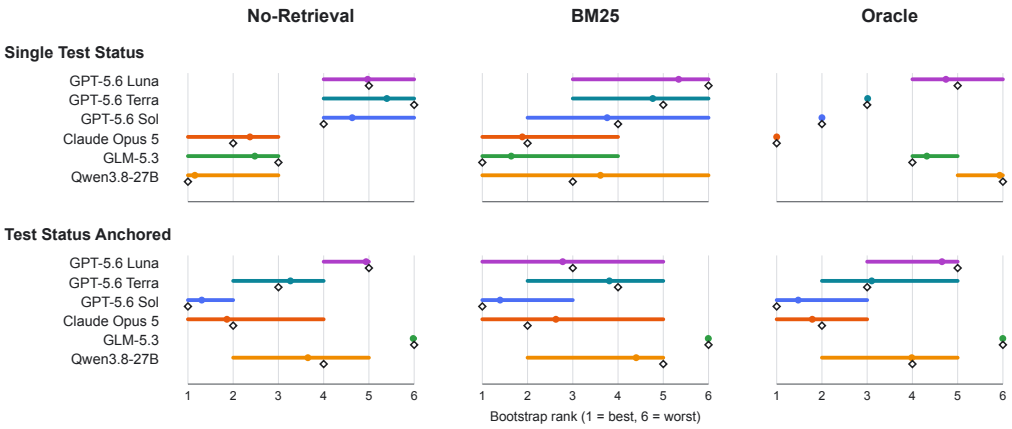
[QUESTION: HEAVY VARIANT]
The provided workload is a single test, `test_workload.py::test_workload_heavy`. You are given a
list of candidate functions from this repository that ran during that call. List the 5 with the
highest cumulative memory allocation | largest first. A function's cumulative memory allocation is
the memory added between entry and return, summed over every call; memory freed and re-allocated
between calls counts each time. A function that allocates 80 MB and returns it, called a hundred
times while the caller drops the previous result, counts about 8 GB, even though no more than about
80 MB is ever held at once | this is cumulative allocation, not memory still held when the test
ends. Memory is measured as the Python heap as seen by tracemalloc. A function is credited with the
memory it allocates itself, including allocations made by library code it calls. It is not credited
with memory allocated by the functions it calls that appear in this list | that memory counts
towards their entries instead. Identify each function by its file path, its first line number and
its qualified name.
```

```
[QUESTION: MEDIUM VARIANT]
The provided workload is a single test, `test_workload.py::test_workload_medium`. You are given a
list of candidate functions from this repository that ran during that call. List the 5 with the
highest cumulative memory allocation | largest first. A function's cumulative memory allocation is
the memory added between entry and return, summed over every call; memory freed and re-allocated
between calls counts each time. A function that allocates 80 MB and returns it, called a hundred
times while the caller drops the previous result, counts about 8 GB, even though no more than about
80 MB is ever held at once | this is cumulative allocation, not memory still held when the test
ends. Memory is measured as the Python heap as seen by tracemalloc. A function is credited with the
memory it allocates itself, including allocations made by library code it calls. It is not credited
with memory allocated by the functions it calls that appear in this list | that memory counts
towards their entries instead. Identify each function by its file path, its first line number and
its qualified name.

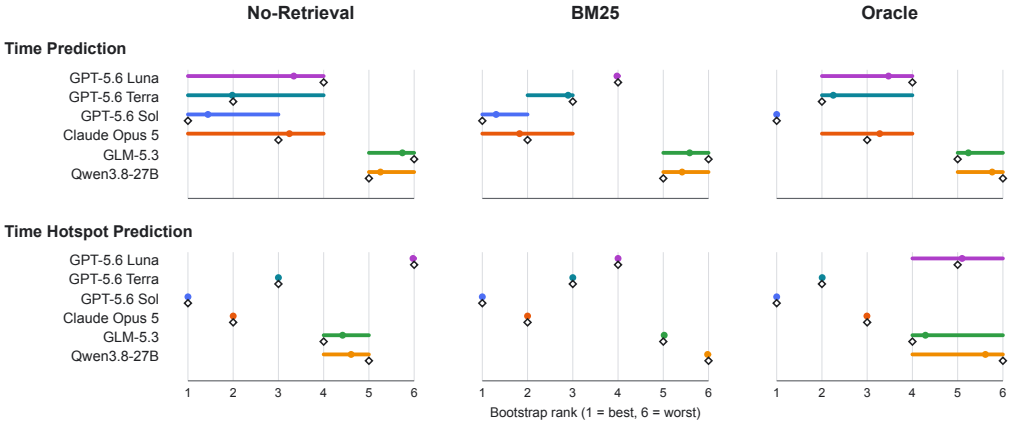
[QUESTION: LIGHT VARIANT]
The provided workload is a single test, `test_workload.py::test_workload_light`. You are given a
list of candidate functions from this repository that ran during that call. List the 5 with the
highest cumulative memory allocation | largest first. A function's cumulative memory allocation is
the memory added between entry and return, summed over every call; memory freed and re-allocated
between calls counts each time. A function that allocates 80 MB and returns it, called a hundred
times while the caller drops the previous result, counts about 8 GB, even though no more than about
80 MB is ever held at once | this is cumulative allocation, not memory still held when the test
ends. Memory is measured as the Python heap as seen by tracemalloc. A function is credited with the
memory it allocates itself, including allocations made by library code it calls. It is not credited
with memory allocated by the functions it calls that appear in this list | that memory counts
towards their entries instead. Identify each function by its file path, its first line number and
its qualified name.

[OUTPUT INSTRUCTIONS]
Return exactly 5 functions in `functions`, ordered from highest to lowest memory value. Each item
must use `file_path`, `firstlineno`, and `qualname`. Do not include duplicates.
Every returned locator must exactly match one listed candidate (`file_path`, `firstlineno`, and
`qualname`). Use the available static evidence only to rank the listed candidates; do not return
any locator outside the list.
```

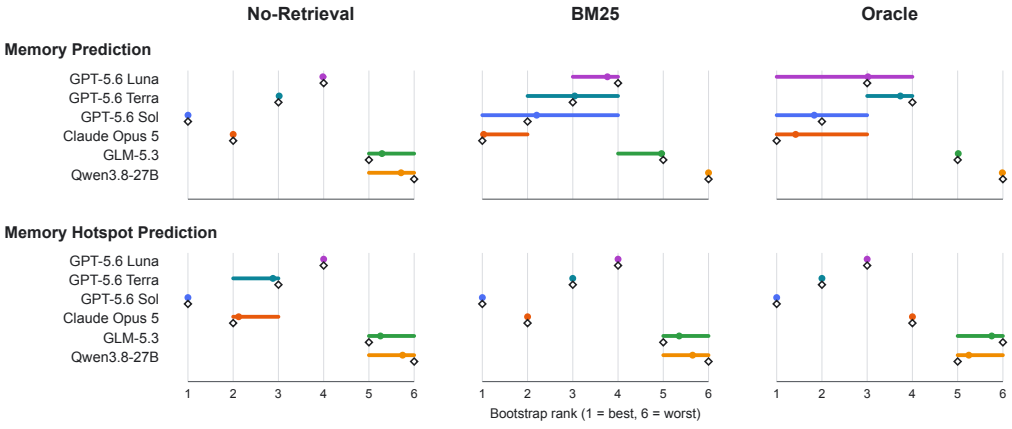
Context delivery. User message: after workload inputs, candidate functions, and profiling details, before the question. BM25 and Oracle fill the block when nonempty; No-Retrieval and agent prompts omit it. Codex and mini-SWE-agent collect repository context through read-only tools.



(a) Test Status tasks.



(b) Time tasks.



(c) Memory tasks.

Figure 13: Instance-bootstrap uncertainty in within-baseline model ranks. Horizontal lines show central 95% rank intervals, • marks the mean bootstrap rank, and ◇ marks the observed rank; rank 1 is best.

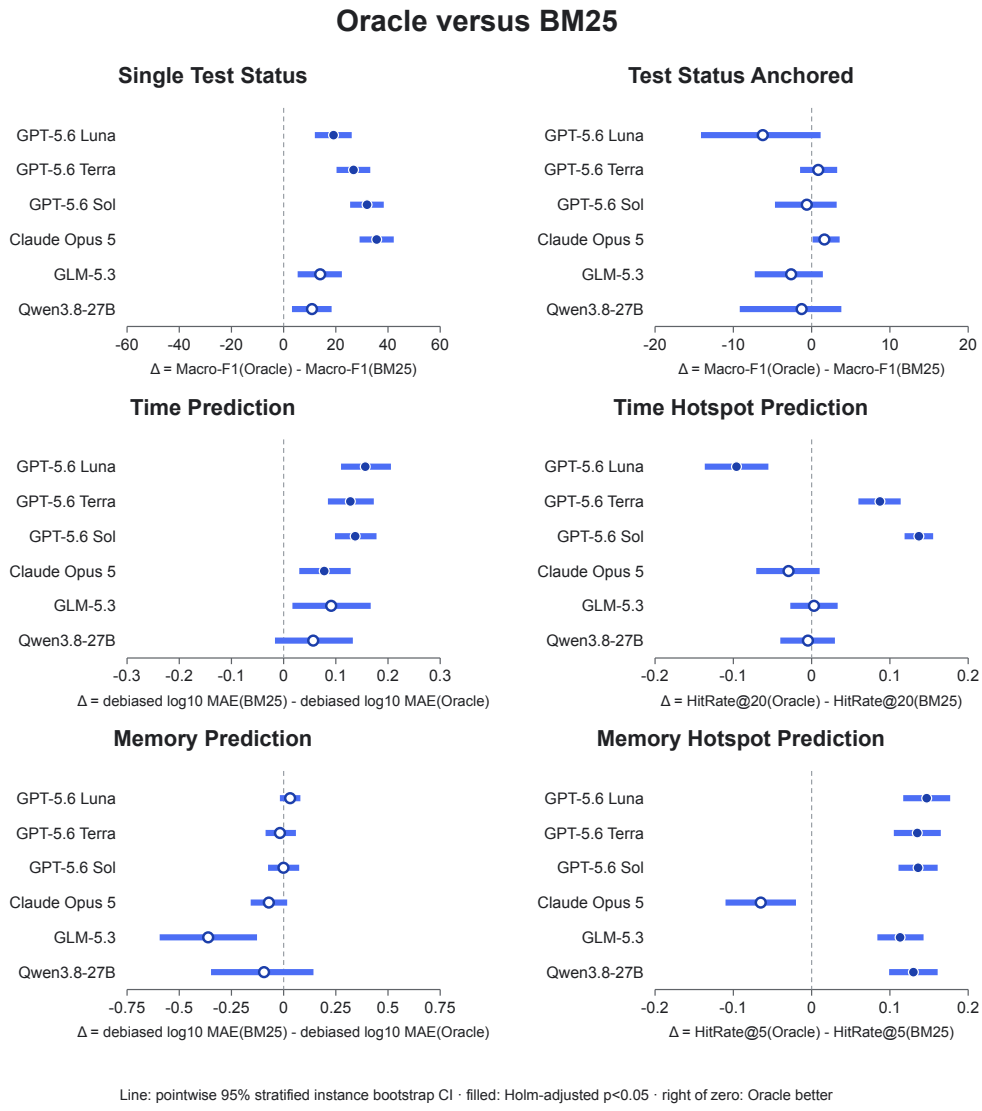


Figure 14: Oracle vs. BM25. Points show observed differences, with positive values favoring Oracle. For debiased $\log_{10}$ MAE, the difference is BM25 minus Oracle because lower error is better. Lines show pointwise 95% stratified bootstrap intervals; filled points are significant after Holm correction (K=72, including comparisons from Figure 17).

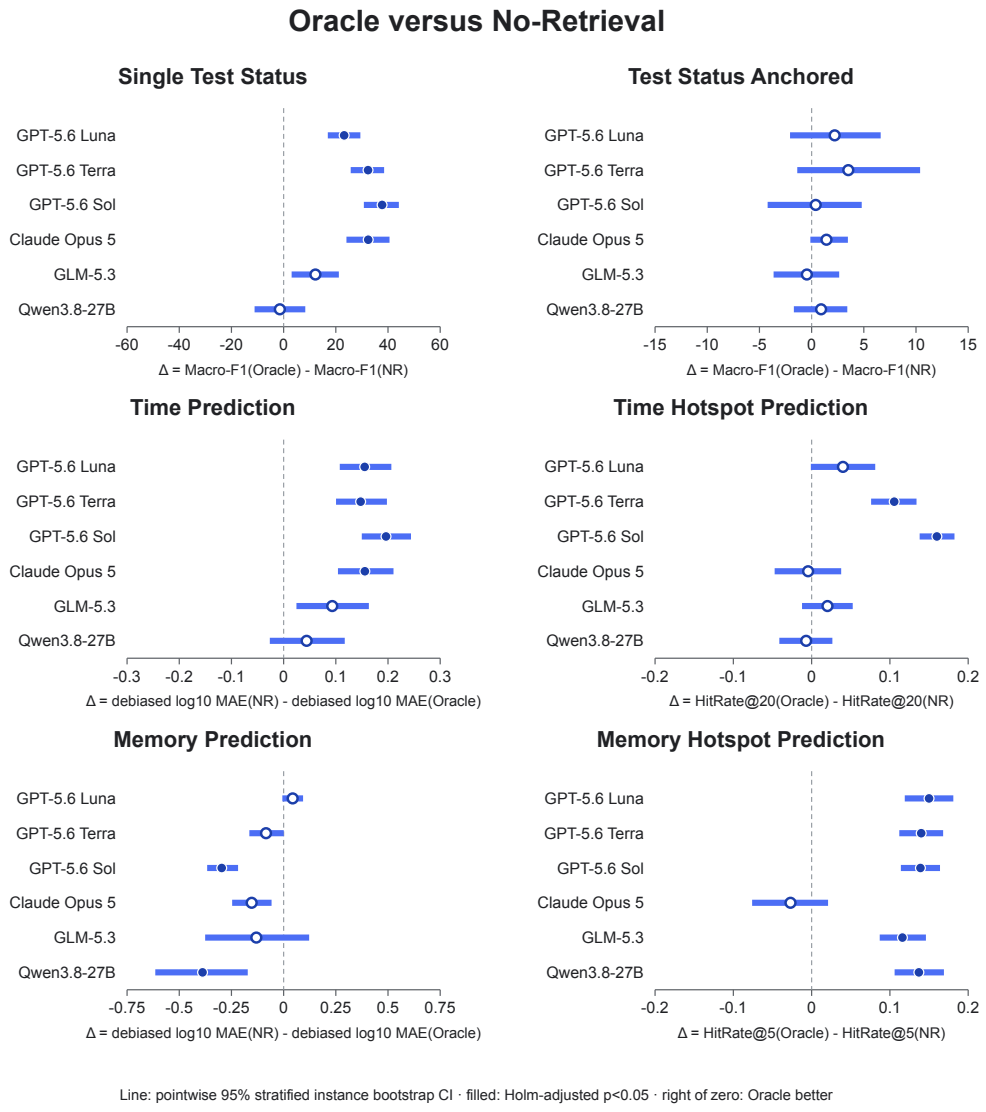


Figure 15: Oracle vs. No-Retrieval. Differences are signed as in Figure 16, so positive values favor Oracle.

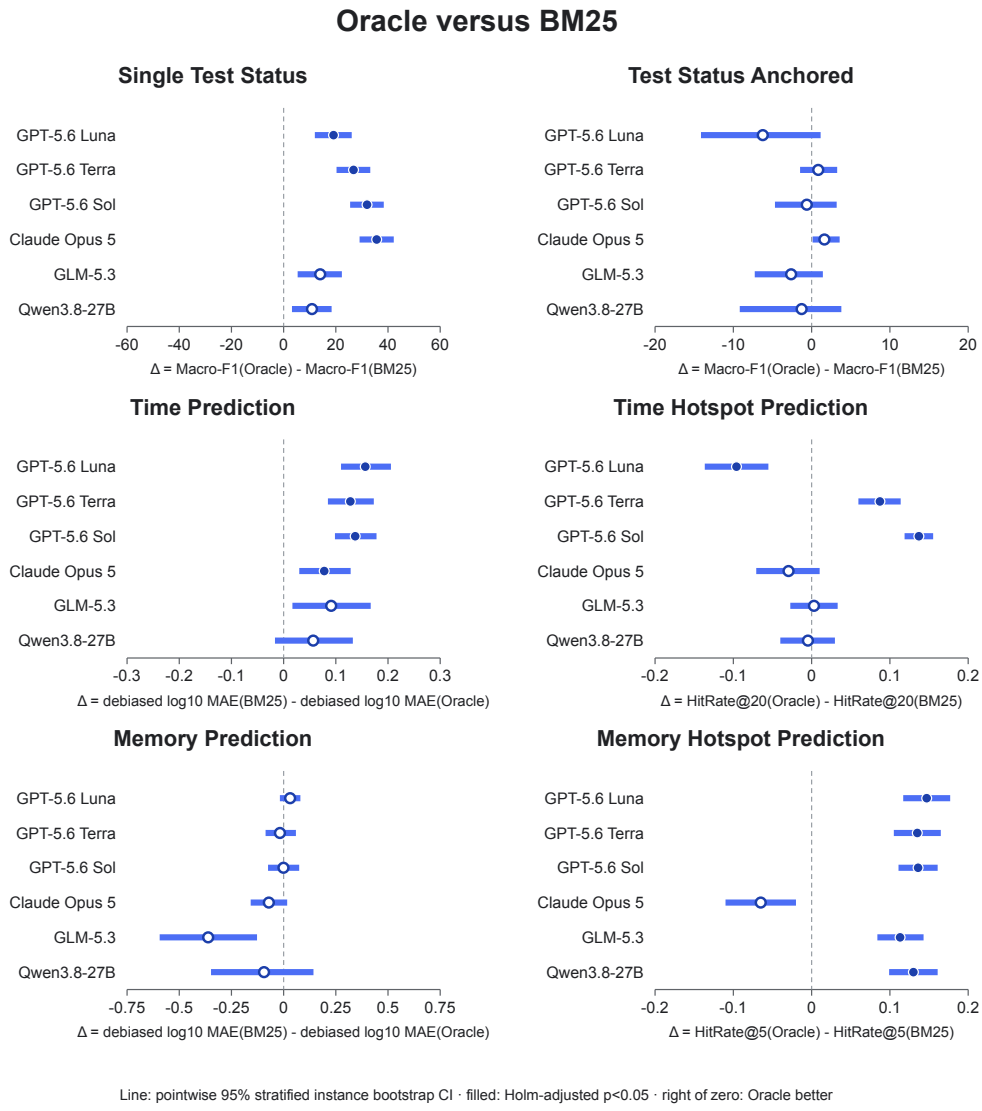


Figure 16: Oracle vs. BM25. Points show observed differences, with positive values favoring Oracle. For debiased $\log_{10}$ MAE, the difference is BM25 minus Oracle because lower error is better. Lines show pointwise 95% stratified bootstrap intervals; filled points are significant after Holm correction (K=72, including comparisons from Figure 17).

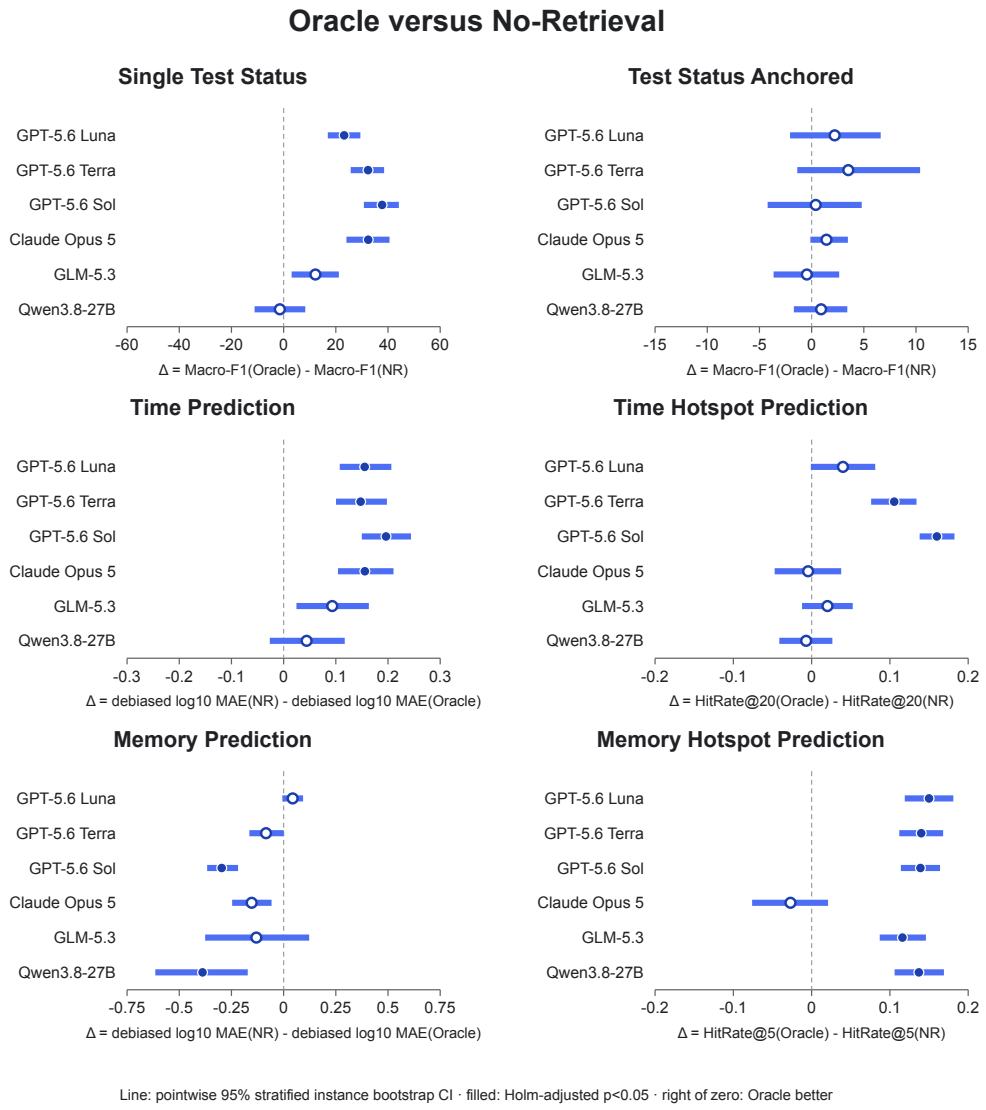


Figure 17: Oracle vs. No-Retrieval. Differences are signed as in Figure 16, so positive values favor Oracle.

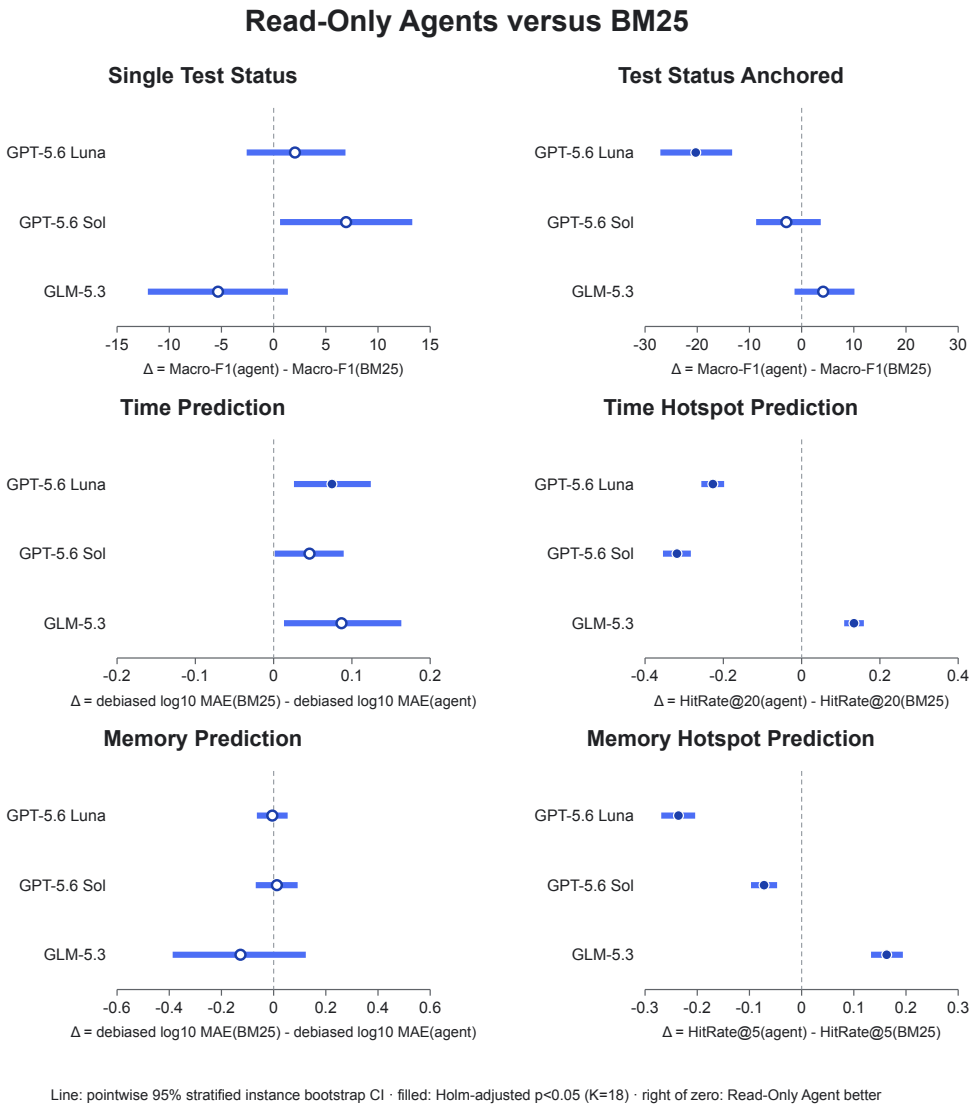


Figure 18: Read-Only Agents vs. BM25. Points show observed differences, with positive values favoring the Read-Only Agent. For debiased $\log_{10}$ MAE, the difference is BM25 minus the agent because lower error is better. Lines show pointwise 95% stratified bootstrap intervals; filled points are significant after Holm correction ($K = 18$).