LMSysDojo: Building Scalable Analytical Agents for Optimizing LLM Serving Systems

Anonymous authors
Paper under double-blind review

Abstract

Complex real-world computing systems are hard to optimize because their performance emerges from many interacting components and depends on the hardware and workloads they serve. We introduce LMSysDojo, a toolkit that reconstructs reproducible optimization playgrounds from merged performance pull requests in vLLM and SGLang. Agents optimize live deployments on real GPUs, and each patch's improvement is normalized against the human expert's fix on the same setup. On the held-out splits of our LMSysDojo-100 benchmark, a Claude Code agent using Opus 5 achieves 57–60% of the expert's gain. A key bottleneck is that a patch's worth is known only after costly deployment and measurement, and a vanilla coding agent exposes no ex-ante signal for choosing among candidate ideas beforehand. Inspired by how systems researchers justify optimizations, we propose *gap analysis*, which separates diagnosis from implementation: an analyzer locates a bottleneck, proposes a mechanism, and predicts its gain before an implementer builds it. *Parallel gap analysis* selects among competing analyses by predicted gain; *structured gap analysis* expresses each prediction as a program over profiles, experimental evidence, and stated assumptions; and *iterative gap analysis* checks and refines these programs before selection. Our combined approach reaches 73–83% of the expert's gain, improving over the single agent by +16–23 percentage points. Across variants, predictive signals turn analysis into a new scaling axis under a fixed evaluation budget, with gains continuing through $K=10$ analyses. We will release the toolkit, benchmark, and 900-setup corpus.

1 Introduction

Can LLM agents research and improve real-world complex systems? A central ambition of AI is to tackle complex systems, such as economies, organizations, and computing infrastructure, whose behavior emerges from coupled components in a changing environment, yet most such systems cannot be replayed to test counterfactual interventions. Software systems are an exception: version control records every past state and human intervention (each merged pull request), allowing deployments to be replayed and interventions measured under controlled workloads. Open-source *LLM serving systems* such as vLLM and SGLang are a timely case: widely deployed in production, they coordinate model execution, memory management, and request scheduling across accelerators, where small latency or throughput gains carry large fleet-wide impact. Whereas existing benchmarks ask agents to resolve bug reports (Jimenez et al., 2023) or speed up code under unit tests (He et al., 2026; Shetty et al., 2026; Sehgal et al., 2026; Ma et al., 2026), optimizing a serving system requires finding where a live deployment loses performance on its workload and hardware, then implementing a mechanism that speeds up end-to-end serving without changing model outputs. We introduce LMSysDojo, an environment that measures such optimizations against human experts, and *gap analysis*, a framework that turns predictive analysis into a new scaling axis.

LMSysDojo. A serving system's performance depends jointly on the engine code, hardware, model, and workload. Because vLLM and SGLang are already heavily optimized, most arbitrary configurations offer little headroom; when an optimization attempt yields no speedup, one cannot tell whether the agent failed or the setup had no headroom. Evaluating optimization agents therefore requires (1) setups with known headroom, (2) an expert's measured fix on each setup to score against, and (3) diverse setups to test generalization. LMSysDojo obtains all three from the engines'

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.

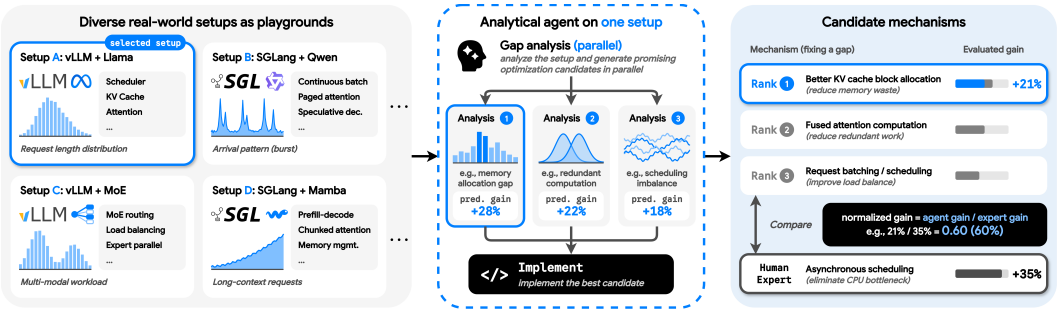


Figure 1: LMSysDojo and gap analysis at a glance. **Left:** Real serving setups rebuilt from merged vLLM and SGLang pull requests serve as reproducible playgrounds. **Middle:** Parallel gap analysis runs multiple independent analyzers; each locates a performance gap, proposes a mechanism, and predicts its gain, and only the highest-predicted mechanism is implemented. **Right:** The implemented patch is scored by its *normalized gain*, its improvement normalized against the expert's fix on the same setup.

development history, scanning all 39,411 merged PRs of vLLM and SGLang to identify 900 self-contained performance optimizations. It uses each PR only offline to rebuild the pre-PR deployment into a reproducible playground driven by GuideLLM (vLLM project, 2025) on real GPUs. With no internet access and all PR text, git history, and expert fixes sealed away, the agent receives only the deployment configuration and a brief scenario theme. Inside the playground, it explores performance gaps, implements a patch, and is scored by its *normalized gain*, its improvement normalized against the expert's fix on the same setup (100% matches the expert; Fig. 1).

Gap analysis. Measuring a serving patch requires a full-load benchmark on dedicated GPUs (§C.8), so an agent must decide which optimization mechanisms warrant implementation and measurement. Yet a code diff alone provides neither a quantitative prediction of its benefit nor an explicit account of the assumptions behind it. Inspired by how systems researchers justify optimizations (§B.4), *gap analysis* separates diagnosis from implementation (§3): an *analyzer* identifies a bottleneck, proposes a mechanism, and predicts its gain; an *implementer* then builds the selected mechanism from that analysis. The prediction makes competing proposals comparable before implementation, while its supporting evidence and assumptions make the reasoning open to inspection. Holding the analysis fixed sharply reduces run-to-run variability (Fig. 4), and using a smaller analyzer reduces held-out E2E normalized gain by 8–12 points on average across implementers, while using a smaller implementer on the same Opus analyses preserves average performance (§D.4). We develop three extensions around this prediction: *parallel gap analysis* selects the highest-predicted mechanism from multiple independent analyses; *structured gap analysis* expresses the prediction as an executable program, `analyze(C, E, A)`, whose inputs come from profile queries C, experimental evidence E, and stated assumptions A; and *iterative gap analysis* checks the program's sensitivity and audits its assumptions, returning the findings to a fresh analyzer when revision is needed.

LMSysDojo-100 benchmark. From the 900 measured setups, filtering for clean measurements and headroom above the 1.075× noise floor yields LMSysDojo-100: dense, MoE, and vision-language models across all major serving subsystems (Fig. 2), split into a development set and two held-out sets merged before and after Opus 5's knowledge cutoff. Because every item carries its merge date, the corpus supports analogous post-cutoff splits for future models. On the held-out splits, structured gap analysis with parallel selection reaches 71–77% of the expert's gain (+14–17 points over the single agent) and iterative gap analysis reaches 73–83%, with gains still climbing at $K=10$ across model families.

Contributions. We make three contributions. First, we introduce LMSysDojo, a toolkit that turns merged performance PRs of serving engines into sealed, expert-calibrated playgrounds, and LMSysDojo-100, a benchmark supporting temporal post-cutoff evaluation (§2). Second, we propose *gap analysis*, an agent framework in which an analyzer predicts a fix's gain before an im-

plementer builds it, with parallel, structured, and iterative extensions that select on, expose, and check that prediction (§3); structured gap analysis is, to our knowledge, the first formulation of an LLM agent's optimization claim as an executable, source-traced program. Third, we provide an empirical, theoretical, and qualitative study across model sizes, families, and mechanisms that beat the expert's fix, showing that predictive signals make analysis a new axis for test-time scaling with gains continuing through $K{=}10$ analyses (§4). We will release the open-source toolkit, benchmark, and 900-setup corpus.

2 LMSysDojo Toolkit and Benchmark

LMSysDojo reconstructs reproducible playgrounds from the engines' development history, letting agents optimize live deployments on real hardware and scoring their patches against human experts (Fig. 1; §A). Unlike static leaderboards that rank engines on fixed deployments, such as InferenceX (SemiAnalysis, 2026), LMSysDojo provides an interactive optimization environment with four properties: (1) *reference-calibrated evaluation*: every playground is anchored to a merged optimization PR whose gain we re-measure, grading each patch against that expert reference on the same hardware; (2) *reusable, diverse setups*: scanning all $39{,}411$ merged PRs yields 900 eligible measured setups across serving subsystems, each with its setup card, launch configuration, expert trace, and baseline profiles (Fig. 2, §A.1); (3) *unified setup and metrics*: every playground drives the live server with GuideLLM (vLLM project, 2025), an open-source LLM serving load generator that replays configurable prompt-length and concurrency or request-rate profiles and records five standard metrics: end-to-end latency (E2E), time to first token (TTFT), inter-token latency (ITL), output tokens per second, and requests per second (§A.3.1); and (4) *refreshable, sealed evaluation*: the constructor ingests new upstream PRs timestamped by merge date for temporal splits while sealing away the expert fix, commit hash, and git history.

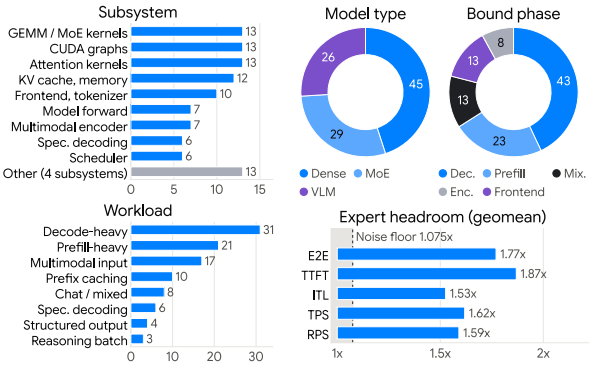
Normalized gain. Each benchmark item specifies three things: the system (the engine's source at a pre-PR commit with its launch configuration), the workload and hardware setup, and the reference fix (the expert's merged patch). A patch's speedup is its improvement over the unpatched baseline on the target metric, with $1\times$ meaning no change. We grade each patch by its increase in speedup above $1\times$, normalized by the expert's corresponding increase on the same setup,

$$\text{normalized gain} \;=\; \frac{\text{speedup of the patch} - 1}{\text{speedup of the expert's fix} - 1},$$

so 0% is no change, 100% matches the expert, and values above 100% beat the expert. A playground tests whether the agent speeds up the deployment, not whether it replicates the expert's mechanism: any valid improvement counts, even in different files. The expert's speedup sets the reference scale rather than a ceiling (item gains are capped at $3\times$ the expert's gain for stability), and patches touching no expert-modified files still beat the expert in 22–27$\%$ of cases (§D.1). Both speedups are measured against the same sealed baseline, and split scores aggregate item geometric-mean speedups (§A.4.3).

Playground constructor. Finding deployments with measurable headroom is itself non-trivial: in an open-ended exploration where agents chose the model, workload, and launch configuration on pinned engine releases, none of 150 attempts exceeded the $1.075\times$ noise floor (§A.6). Without a reference fix, this null result cannot distinguish agent failure from a lack of headroom. Across the engines' merged PRs, 2.3% meet our criteria for documented, reconstructable optimizations (Fig. 2; §A.3). The constructor turns each eligible PR into a self-contained *playground item* in two stages. Offline, agents read the PR thread, diff, and benchmark report to select candidate PRs (*Scan*), identify baseline and reference commits (*Segment*), extract a setup card with the deployment configuration and scenario theme (*Parse*), and check completeness (*Verify*; §A.3, Table 3). On target GPUs, *Measure* benchmarks the baseline and expert reference (§A.3.2), and *Profile* records baseline traces with the PyTorch profiler, `nsys`, and `py-spy` (Table 4) so every agent starts from the same reusable evidence (§A).

Evaluation environment. The evaluation environment runs an agent on a dedicated GPU node via Slurm, validates its delivery, and scores it (§A.4) in three parts. First, the *interaction sandbox* exposes the baseline source and environment, setup card, pre-recorded profiles, node GPUs, and a local benchmark script. Unlike issue-resolution benchmarks, the PR is used only offline: all PR text,



Census funnel	**vLLM**	**SGLang**	**Total**
Merged PRs scanned	20,969	18,442	**39,411**
Measured setups	607	293	**900**
LMSysDojo-100 suite	76	24	**100**
Suite Splits (N=100)	**Dev**	**Pre**	**Post**
Engine (vLLM / SGLang)	21/9	33/2	22/13
MoE / VLM share	43/20%	23/17%	31/40%
Expert E2E gain (geomean)	1.81×	2.25×	1.37×
Expert E2E gain (sd)	1.7	4.3	0.4
Expert patch (med. files)	3	4	4
Expert patch (med. LOC)	144	100	212
Hardware & Load (N=100)			**Coverage / Share**
GPUs (1/2/4/8× H100)			80/4/10/6%
Params (<4 / 4–10 / 10–40 / 40B+)			16/51/21/12%
Prompts (synth. / real / image+text)			73/13/14%

Figure 2: The LMSysDojo corpus and the LMSysDojo-100 benchmark. **Left:** Target subsystems and workloads across the 100 benchmark items. **Middle:** Model types, bottleneck phases, and the expert's geometric-mean speedup per metric against the 1.075× noise floor. **Right:** Filtering funnel from 39,411 merged PRs to 900 measured setups and 100 benchmark items, split sizes, and hardware configurations. Model types are mutually exclusive (an MoE vision-language model counts as VLM). Full breakdown in Table 7.

commit hashes, git history (repacked into a single-commit checkout), and hints about the gap or fix are stripped, internet access is disabled, and all tool calls and file reads are audited (§A.4.1). *From the agent's view, it is simply optimizing a standalone serving deployment.* Edits are confined to Python and Triton against the baseline's compiled extensions (95 of 100 reference fixes are reproducible in Python alone; §A.5), with a 1.5-hour wall-clock budget (§C.1). Second, the *checker* enforces workload and output equivalence (§A.4.2): static checks reject launch-flag or environment changes, request-bound runs must match the baseline request count within 5%, and greedy decoding on 100 fixed prompts must match the first 32 tokens with top-5 log-probabilities within 0.01. Third, the *scorer* deploys the validated patch once on a separate single-tenant node under the full benchmark and computes normalized gain. As in standard code-submission benchmarks, each run submits a *single final patch per item* for official end-to-end scoring on the target deployment (§B.2, §C.8).

LMSysDojo-100 benchmark. We select the 100-item benchmark from the corpus using two rules: *clean measurement* (at least 100 requests under matching load with uncorrupted metrics) and *headroom* (expert E2E speedup above the 1.075× noise floor; §A.1). All qualifying post-cutoff items plus a stratified sample of earlier items form LMSysDojo-100 (§A.1, §A.5). It spans both engines, 1–8 GPUs, dense, MoE, and VLM models from under 4B to over 40B parameters, prefill-, decode-, and encoder-bound regimes, and diverse serving workloads; the expert's E2E speedup has a geometric mean of 1.77× and a median of 1.43× (Fig. 2, Table 7). To control for pre-training leakage, we split the benchmark into 30 *dev* items and two 35-item *held-out* splits merged before and strictly after Opus 5's 2026-05-01 cutoff (§A.5).

3 Gap Analysis

On LMSysDojo-100, a single Claude Code agent on Opus 5 reaches only 57% and 60% of the expert's E2E gain on the two held-out splits (Table 1). In a single coding session, the agent entangles bottleneck diagnosis with code implementation, committing early to one hypothesis without a quantitative way to compare alternative optimization mechanisms or verify whether its bottleneck premises hold before the single deployed measurement. To design an agent that can evaluate and refine optimization hypotheses before coding, we examined how systems researchers justify optimizations across 50 recent LLM serving papers (§B.4, Table 9). Across papers, the argument follows a consistent structure:

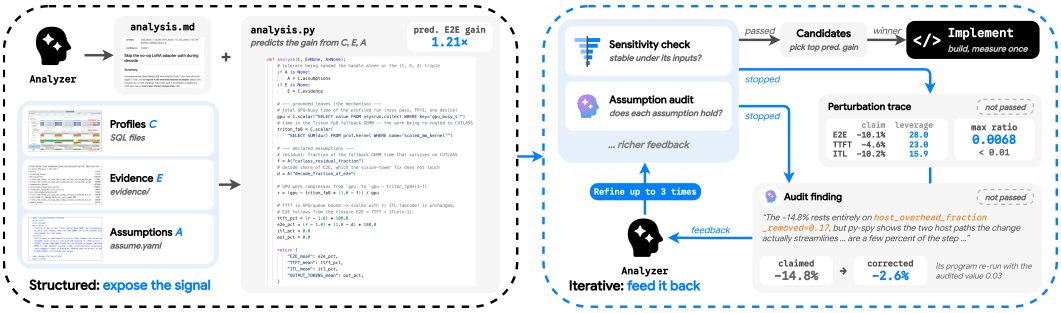


Figure 3: Structured and iterative gap analysis. **Left:** In structured gap analysis, alongside the analysis document, the analyzer submits `analysis.py`, a deterministic program that computes the predicted gain from SQL queries over the profiles (C), its own evidence (E), and stated assumptions (A) in `assume.yaml`. **Right:** In iterative gap analysis, a sensitivity check and an assumption audit examine each structured analysis before implementation; passing analyses enter parallel selection, while a rejected analysis goes back to a new analyzer with the check's findings (up to three times).

Gap analysis: locate the gap, find a fix, predict the gain. *Inspect the profiles to locate where the deployment loses performance; design a mechanism that removes wasted work or spends a resource with headroom to relieve the bottleneck; and predict the mechanism's gain from measured profile quantities, experimental evidence, and stated assumptions, independently of how the code is implemented.*

Gap analysis turns this structure into an analysis-centric agent by separating out implementation (Fig. 3): an *analyzer* locates the performance gap, explains why it costs that much, proposes a mechanism, and predicts its gain, and an *implementer* builds the mechanism from the analysis alone. Below, we show that outcome variance is concentrated in the analysis, and then extend gap analysis in three ways around this predicted gain: *parallel gap analysis* selects on it, *structured gap analysis* exposes what it rests on, and *iterative gap analysis* checks and refines it before coding. Following the bitter lesson (Sutton, 2019), each extension is minimal: we constrain only what the analyzer delivers and when it retries, leaving how it profiles, searches, and reasons to the model.

Why focus on the analysis. The analyzer and implementer run in separate sessions. The analyzer receives the sandbox environment (baseline source, setup, and pre-recorded profiles) and investigates freely: it may run probes, add instrumentation, and prototype edits, but its code changes are discarded when its session ends and it never sees the final score. It delivers an analysis document specifying where time goes by file and line, why it costs that much with measurement evidence, what mechanism to build, how to verify it, and a quantitative *predicted gain* on the scored metrics (§C.4). The repository is then reset, and the implementer builds the patch from the analysis alone without restarting diagnosis (§C.5).

This separation allows us to vary one role while holding the other fixed (Fig. 4). Re-running only the implementer on a fixed analysis barely changes an item's outcome: the median within-item standard deviation of log speedup is 0.010 for free-form analyses and 0.012 for structured ones, versus 0.067 and 0.072 when drawing a fresh analysis each run, a 97–98% reduction in squared median spread (§B). Swapping a smaller model into each role shows the same pattern: downgrading the analyzer hurts substantially, whereas downgrading the implementer does not (§D.4, Table 15). Search effort thus pays at the analysis stage: sampling *more* analyses and producing *better* ones.

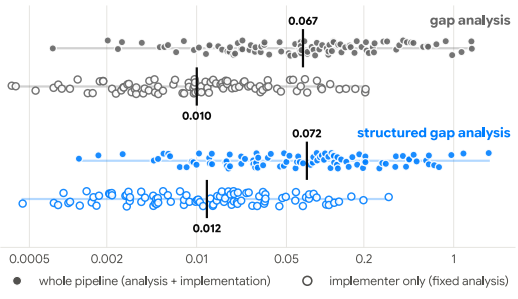


Figure 4: Within-item spread of log E2E speedup when re-running the whole pipeline versus only the implementer on a fixed analysis (ticks are medians); fixing the analysis reduces squared median spread by 97–98% (§B).

Parallel gap analysis. Parallel gap analysis runs K independent analyzers per item ($K{=}10$ in our experiments), selects the analysis with the highest predicted E2E gain (breaking ties on TTFT, then ITL, then at random), implements only that winner, and measures it once. At $K{=}1$, gap analysis shares the single agent's 1.5-hour budget and achieves comparable performance while providing a quantitative selection signal (Table 1). By predicting gain before implementing, gap analysis turns diagnosis into a scalable search axis (§B.2, §C.8): each analyzer explores a candidate mechanism in profile and microbenchmark space and outputs a comparable speedup estimate before any patch is built. A single agent (VANILLA) outputs no such estimate to compare candidate mechanisms; even an LLM ranker over K finished patches (+ *ranker* in Table 1) must run all K coding sessions and judge code diffs without a quantitative bottleneck model, while chaining K sequential VANILLA sessions serializes wall-clock latency to $O(K)$ versus $O(1)$ for parallel analysis (§C.8). We test the simplest form of parallel search, independent identically prompted analyzers ranked by one scalar prediction, though specialized or collaborative analyzers fit the same interface.

Structured gap analysis. In free-form gap analysis, predicted gains arrived as ranges, nested tables, or prose numbers (4% unparseable), and the premises behind a number cannot be checked before coding. For example, one free-form claim of -15% E2E latency rested on a false premise about vLLM's CUDA-graph modes that the implementer discovered only while coding, landing a 2% regression (§E.4); another measured a 75% kernel-time share ($4\times$ Amdahl ceiling) yet passed it through an unstated formula to claim $14\times$, beating valid analyses only to deliver $1.05\times$.

Structured gap analysis exposes what a prediction rests on. Alongside the document, the analyzer submits a deterministic program, `analyze(C, E, A)`, that computes the predicted percent change of every scored metric from three sources (Fig. 3): C, SQL queries over the baseline profiles and setup; E, experimental evidence from the analyzer, such as a microbenchmark; and A, stated assumptions in `assume.yaml`, each with a value and reason. Any other numeric literal (beyond 0, 1, and unit conversions like 100) counts as undeclared, and the document's prediction must match the program's output so every input can be traced for sensitivity (§D.3).

Iterative gap analysis. Iterative gap analysis turns this exposed structure into a feedback loop: each structured analysis is checked before implementation, and a failed analysis goes back to a fresh analyzer with the check's findings. We test the loop with the two simplest checks the program affords (§C.6): a *sensitivity check* perturbs the program's inputs and rejects predictions that fail to run, have zero input sensitivity, or swing overly under perturbation relative to the claimed gain; and an *assumption audit* runs a read-only agent session that checks each stated assumption against the codebase and profiles, rejecting the analysis if a load-bearing assumption is contradicted. Each of the K tracks regenerates rejected analyses up to three times and keeps its first passing analysis (or last retry) before parallel selection picks the highest-predicted winner, sharpening the predictive signal before implementation.

4 EXPERIMENTS

We evaluate agents on LMSYSDOJO-100 using Claude Code (`claude-opus-5`) on single-tenant GCP `a3-mega` nodes (8×H100 80GB GPUs). Every method is charged *one final deployed measurement per item* (§C.6). We report normalized gain on five serving metrics (NVIDIA, 2026): mean E2E latency (primary), time to first token (TTFT), inter-token latency (ITL), output tokens per second (TPS), and requests per second (RPS). To measure candidate pools and oracle ceilings, we also implement and score all unselected analyses post hoc; these outcomes never reach any analyzer, check, or selector, and failed or invalid deliveries count as $1\times$ speedup (0% gain; §C.6).

Compared methods. We compare seven setups (Table 1): **VANILLA**, a single Claude Code session delivering a patch; **+ ranker** ($K{=}10$), where Opus 5 ranks ten VANILLA diffs (with output artifacts and workspace context) to pick one; **Gap analysis** ($K{=}1$), one analyzer then one implementer; **+ parallel** ($K{=}10$), ten analyzers where only the highest-predicted is implemented; **Structured gap analysis** ($K{=}1$ and **+ parallel**), adding `analyze(C, E, A)`; and **+ iterative** ($K{=}10$), which applies the sensitivity check and assumption audit on each track with up to three retries before implementing one winner.

Table 1: Main results on the held-out splits of LMSYSDOJO-100 (`claude-opus-5` on Claude Code). Entries are normalized gain in percent (100% matches the expert, higher is better), summarized over item geometric-mean speedups (§A.4.3). Rows without "+" use one sample (K=1); + *ranker* has Opus 5 rank ten VANILLA diffs; + *parallel* implements the highest-predicted of ten analyses; + *iterative* applies pre-selection checks and up to three retries per track. Bold is best per column, underline second (significance in Fig. 19; dev split in Table 12).

	held-out, pre-cutoff (n=35)					held-out, post-cutoff (n=35)				
setup	E2E	TTFT	ITL	TPS	RPS	E2E	TTFT	ITL	TPS	RPS
Vanilla	57.0	61.2	49.2	62.5	68.2	60.1	61.6	44.4	60.7	61.3
+ ranker	58.3	63.0	50.5	63.5	<u>69.0</u>	65.6	69.6	50.0	66.0	65.7
Gap analysis	55.1	58.3	45.5	61.4	67.1	62.2	65.6	51.3	61.5	62.9
+ parallel	64.7	64.0	57.5	67.4	**73.3**	73.2	72.9	59.1	71.6	72.3
Structured gap analysis	64.6	65.6	59.0	64.3	67.1	62.1	57.7	62.8	62.0	62.6
+ parallel	<u>70.7</u>	<u>72.7</u>	<u>71.4</u>	<u>68.6</u>	63.5	<u>77.3</u>	<u>85.7</u>	<u>73.2</u>	<u>75.4</u>	<u>75.4</u>
+ iterative	**73.1**	**75.0**	**74.5**	**71.4**	66.4	**82.7**	**92.5**	**79.0**	**81.0**	**81.1**

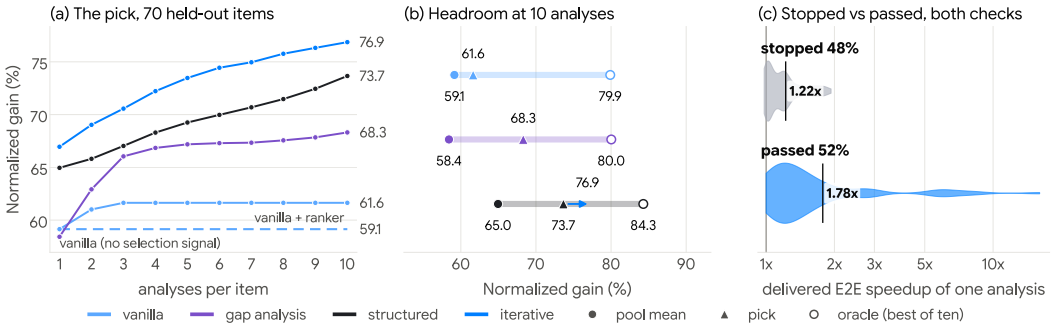


Figure 5: Gap analysis on the 70 held-out items, E2E. **(a)** Expected normalized gain of the selected candidate vs. sampled analyses K; the single agent has no selection signal and stays at its K=1 mean. **(b)** At K=10: the single-analysis mean, the prediction-selected candidate (*pick*), and the post-hoc best of ten (*oracle*); the arrow marks the iterative pick. **(c)** Delivered E2E speedups of structured analyses rejected (*stop*) vs. accepted (*pass*) by the checks, with group geometric means (per-split curves in Figs. 6, 14, and 18).

Gap analysis lifts a strong single agent. The single agent reaches 57.0% and 60.1% of the expert's E2E gain on the pre- and post-cutoff held-out splits (Table 1). Structured gap analysis with parallel selection raises these to 70.7% and 77.3% (p=0.003 pooled against VANILLA; Fig. 19), and iterative gap analysis raises them further to 73.1% and 82.7% (+16–23 points). The same progression holds for Gemini 3.1 Pro and Gemini 3.8 Flash on the Antigravity CLI (+7.0 and +8.3 E2E points; Table 2), and with a smaller Sonnet analyzer, parallel selection closes roughly half the gap to parallel Opus and beats single-session Opus by 10–13 points (Table 15).

More analyses give better picks. Without a selection signal, sampling multiple candidates under a single-evaluation budget stays at the single-run mean (dashed line in Fig. 5a). Having Opus 5 rank K=10 VANILLA code diffs (+ *ranker*) reaches only 58.3% and 65.6% E2E (Table 1), showing that our gains come not merely from 10× more compute, but from the predictive signal in gap analysis: selecting on predicted gain before coding lifts performance to 64.7%–73.2% (unstructured) and 70.7%–77.3% (structured) at K=10, with the structured curve still rising at K=10 (Fig. 5a; Fig. 14). Evaluating all ten candidates post hoc shows substantial headroom: pooled over the 70 held-out items, the oracle best-of-ten reaches 80.0% (unstructured) and 84.3% (structured), versus 68.3% and 73.7% when selecting on predicted gain alone (Fig. 5b; Fig. 6; §D.2).

Table 2: Cross-family and cross-CLI generalization on the held-out post-cutoff split (n=35). Normalized gain in percent for `gemini-3.1-pro-high` (left) and `gemini-3.8-flash-high` (right) on the Antigravity CLI (§D.5). Bold is best per column.

	Gemini 3.1 Pro (`gemini-3.1-pro-high`)					Gemini 3.8 Flash (`gemini-3.8-flash-high`)				
setup (held-out, post-cutoff)	E2E	TTFT	ITL	TPS	RPS	E2E	TTFT	ITL	TPS	RPS
Single agent (vanilla)										
Single agent	58.4	65.0	43.0	56.4	56.5	68.6	77.7	66.4	65.5	67.6
Gap analysis										
Analyzer → impl.	60.7	61.7	42.2	58.3	59.7	70.7	70.3	63.4	70.0	71.0
+ parallel	63.6	67.3	**57.6**	62.0	63.1	73.7	78.6	68.5	71.8	72.7
Structured gap analysis										
Analyzer → impl.	60.6	54.8	51.9	60.4	61.8	71.7	77.1	68.5	68.1	69.4
+ parallel	**65.4**	**76.7**	56.9	**67.2**	**68.6**	**76.9**	**91.6**	**69.8**	**75.8**	**76.5**

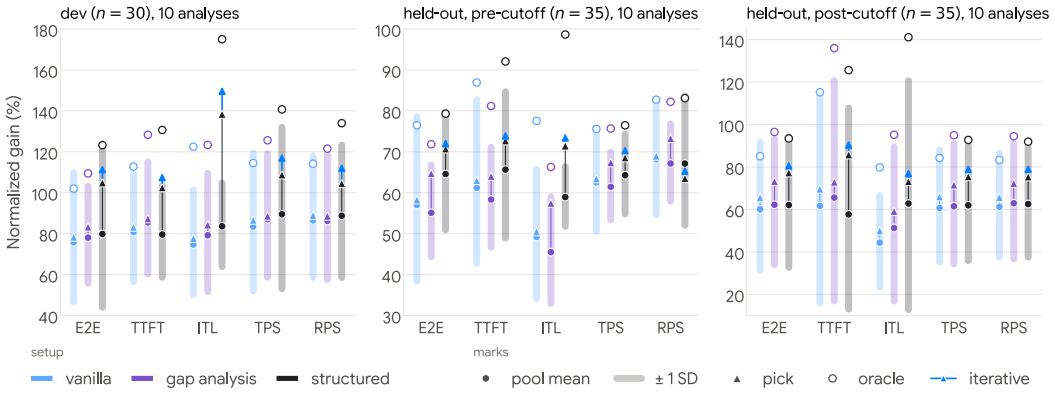


Figure 6: Candidate quality against selection per split and metric at K=10. For each method: the dot is the single-analysis mean, the band one within-item standard deviation, the triangle the parallel pick, the arrow the iterative pick, and the open circle the post-hoc best of ten (*oracle*).

Structure exposes signal that checks can use. Beyond beating unstructured gap analysis on 9 of 10 held-out metric–split cells under parallel selection (Table 1; 9 of 10 on Gemini, Table 2), the prediction program makes fragile analyses inspectable. Across the 1,000 submitted programs (10 analyses ×100 items), 848 execute cleanly and return a traceable E2E prediction; post-hoc AST provenance tracing shows that 58% of mean input sensitivity weight rests on undeclared literals and 21% of traceable predictions have zero sensitivity to the baseline profiles C (§D.3). Together, the sensitivity check and assumption audit filter out fragile analyses: on held-out items they reject 48% of candidates, with a geometric-mean speedup of 1.22× vs. 1.78× for those that pass (Fig. 5c).

Feedback lifts the pick. Returning a failed check's findings to a fresh analyzer repairs flawed analyses before selection, raising the pooled held-out E2E pick from 73.7% to 76.9% (Fig. 5b). On the post-cutoff split, combining both checks raises the pick from 77.3% to 82.7%, beating the sensitivity check (81.0%) or assumption audit (79.0%) alone (Figs. 17–18; all-split summary in Fig. 15).

Agents find mechanisms the expert missed. Across methods, 35–38% of delivered attempts exceed the expert's gain. Roughly a quarter of delivered patches share no modified files with the reference PR, and 22–27% of gradeable patches in that group still beat the expert (Table 13). Agents surpass the expert both by resolving a different bottleneck (e.g., on SGLang 8940, batching single-adapter LoRA decode requests as one segment reaches 1.78× E2E vs. the expert's 1.22× host-buffer fix) and by implementing a stronger algorithmic fix to the same bottleneck (e.g., on vLLM 4696,

chaining per-block prefix hashes reduces $O(L^2)$ hashing to $O(L)$, reaching $1.65\times$ vs. the expert's $1.29\times$; case studies in §E.1–E.2).

5 Related Work

Environments and benchmarks for discovery agents. MineDojo (Fan et al., 2022), SocioDojo (Cheng & Chin, 2024), and DiscoveryWorld (Jansen et al., 2024) provide sandboxes for embodied learning, societal analysis, and scientific discovery, while recent frameworks automate parts of the research loop (Lu et al., 2026; Tang et al., 2026; Gottweis et al., 2026). In software engineering, benchmarks mine repositories for bug fixing (Jimenez et al., 2023; Wei et al., 2026) or optimize competitive programs, CPU routines, GPU kernels, and inference setups (Shypula et al., 2024; Shetty et al., 2026; He et al., 2026; Ma et al., 2026; Sehgal et al., 2026; Ouyang et al., 2025; Yeon et al., 2026). LMSysDojo instead reconstructs live multi-GPU serving deployments from merged PRs, fixes the setup, and grades patches against the expert's speedup.

Predicting and selecting before executing. Coding agents typically select candidates by executing them via repeated sampling (Brown et al., 2024), generated tests (Ehrlich et al., 2025), tree search (Antoniades et al., 2025), or evolution (Novikov et al., 2025), or filter them when evaluation is costly via LLM review (Cheng et al., 2026a), fitness prediction (Wu et al., 2025), or simulated user feedback (Cheng et al., 2026c). Parallel gap analysis instead selects before coding so only one winner is built. To ground these estimates, structured gap analysis combines performance sizing and what-if modeling (Amdahl, 1967; Williams et al., 2009; Curtsinger & Berger, 2015; Zhu et al., 2020) with executable LLM reasoning (Gao et al., 2023; Cheng et al., 2026b) in a source-traced program, `analyze(C, E, A)`. To curb the optimizer's curse (Smith & Winkler, 2006) when ranking noisy estimates, iterative gap analysis audits each program's sensitivity and assumptions before selection.

LLM serving systems and agents for systems. AI-Driven Research for Systems (ADRS) evolves scheduling, placement, and routing heuristics with automated evaluators (Cheng et al., 2025), alongside agents that optimize code via RL (Du et al., 2026), search parallel mappings (Wei et al., 2025), or tune sharding and serving configurations from profiles (Ding et al., 2026). As Cheng et al. (2025) note, however, evolutionary search relies on isolated policy functions and fast simulators, entangling ideation with coding. We instead target full serving repositories via gap analysis, mirroring how systems research, from paged attention and prefix reuse to chunked prefill, disaggregation, offloading, pooling, and communication overlap (Kwon et al., 2023; Zheng et al., 2024; Agrawal et al., 2024; Zhong et al., 2024; Jiang et al., 2025; Qin et al., 2025; Zhu et al., 2025; Xie et al., 2026), trades slack resources or cuts wasted work to relieve a profiled bottleneck (§B.4).

6 Conclusion

We began by asking whether LLM agents can research and improve real-world complex systems, using production LLM serving engines as a replayable testbed. LMSysDojo turns the development history of vLLM and SGLang into reproducible environments for studying optimization agents, with measurable headroom and expert fixes as performance references. On its held-out benchmark splits, gap analysis raises performance from 57–60% to 73–83% of the expert's gain while holding the final deployed evaluation budget fixed. Separating diagnosis from implementation makes optimization hypotheses independently selectable; expressing their predictions as source-traced programs makes their premises inspectable; and checking those premises enables revision before implementation. Role ablations and mechanism studies further support investing search effort in analysis, with gains continuing through $K{=}10$ analyses and agent-discovered mechanisms beating the expert on over a third of delivered attempts. These results demonstrate *analysis-time scaling* in LLM serving: an agent can improve the intervention it ultimately tests by expanding and refining the quantitative analyses that precede it, offering a promising direction for research agents operating in real-world complex systems under limited experimental budgets.

AI USE STATEMENT

Generative AI is used in the benchmark construction pipeline and in the optimization agents studied in this paper. The construction agents assist with pull-request selection, setup extraction, and deployment reconstruction (Appendix A). The evaluated agents produce optimization analyses and code changes under the prompts, tools, and budgets in Appendix C. We also used LLMs to polish the writing and check typos.

ETHICS STATEMENT

Our study is foundational research for the automatic optimization of LLM serving systems. We see no potential negative impact on society.

REPRODUCIBILITY STATEMENT

We provide complete details of our benchmark construction and evaluation protocol in Appendix A, including the playground constructor pipeline (§A.3, Table 3), sandbox isolation and equivalence checks (§A.4, Table 5), normalized-gain aggregation (§A.4.3), and the full 100-item manifest with commits, models, and hardware configurations (§A.5, Table 7). Complete experimental and agent implementation details are given in Appendix C, including compute budgets and node configurations (§C.1), agent-visible sandbox artifacts (§C.2), verbatim prompts for all agent roles (§C.4), harness mechanics (§C.5), and the exact specifications of parallel selection, the sensitivity check, and the assumption audit (§C.6). We will release the LMSYSDOJO toolkit, the LMSYSDOJO-100 benchmark, and the corpus of 900 measured setups (§A.7).

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 117–134, 2024.

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *International Conference on Learning Representations*, volume 2025, pp. 64485–64515, 2025.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Shubham Agarwal, Mert Cemri, Bowen Wang, Alexander Krentsel, Tian Xia, Jongseok Park, et al. Let the barbarians in: How AI can accelerate systems performance research. *arXiv preprint arXiv:2512.14806*, 2025. URL https://arxiv.org/abs/2512.14806.

Junyan Cheng and Peter Chin. Sociodojo: Building lifelong analytical agents with real-world text and time series. In *International Conference on Learning Representations*, volume 2024, pp. 22153–22187, 2024.

Junyan Cheng, Peter Clark, and Kyle Richardson. Language modeling by language models. *Advances in Neural Information Processing Systems*, 38:171231–171331, 2026a.

Junyan Cheng, Kyle Richardson, and Peter Chin. Analytica: Soft propositional reasoning for robust and scalable LLM-driven analysis. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=9cFT6u82uh.

Junyan Cheng, Ankit Srivastava, Jessie Zeng, Milenko Drinic, and Jack W Stokes. Apeiron: A scalable llm-agentic framework for autonomous full-lifecycle demand-optimized application synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 3868–3899, 2026c.

Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 184–197, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815409. URL https://doi.org/10.1145/2815400.2815409.

Yuran Ding, Ruobing Han, Xiaofan Zhang, and Xinwei Chen. Prompts: Performance optimization via multi-agent planning for llm training and serving. *Proceedings of Machine Learning and Systems*, 8:1010–1039, 2026.

Mingzhe Du, Anh Tuan Luu, Yue Liu, Yuhao Qing, Dong Huang, Xinyi He, Qian Liu, Zejun Ma, and See-kiong Ng. Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization. *Advances in Neural Information Processing Systems*, 38:8979–9011, 2026.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35: 18343–18362, 2022.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, pp. 10764–10799. PMLR, 2023.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Petar Sirkovic, Artiom Myaskovsky, Grzegorz Glowaty, Felix Weissenberger, Alessio Orlandi, Dan Popovici, et al. Accelerating scientific discovery with co-scientist. *Nature*, pp. 1–3, 2026.

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? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=pNxw4i5Dp2.

Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. *Advances in neural information processing systems*, 37:10088–10116, 2024.

Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. Neo: Saving gpu memory crisis with cpu offloading for online llm inference. *Proceedings of Machine Learning and Systems*, 7, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. *URL https://arxiv. org/abs/2310.06770*, 7, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914–919, 2026.

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? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0pyFbZSfbT.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

NVIDIA. Metrics — NVIDIA NIM LLMs benchmarking guide. https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html, 2026. Accessed 2026-09-22.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *23rd USENIX conference on file and storage technologies (FAST 25)*, pp. 155–170, 2025.

Atharva Sehgal, James Hou, Akanksha Sarkar, Ishaan Mantripragada, Swarat Chaudhuri, Jennifer J. Sun, and Yisong Yue. Formulacode: Evaluating agentic optimization on large codebases. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=WArbqRUsAe.

SemiAnalysis. InferenceX: Open source continuous inference benchmark research platform (formerly InferenceMAX). https://github.com/SemiAnalysisAI/InferenceX, 2026. Accessed 2026-09-22.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=I5qDL315bQ.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

James E. Smith and Robert L. Winkler. The optimizers curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, March 2006. ISSN 0025-1909. doi: 10.1287/mnsc.1050.0451. URL https://doi.org/10.1287/mnsc.1050.0451.

Richard S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. Blog post, March 13, 2019.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. Ai-researcher: Autonomous scientific innovation. *Advances in Neural Information Processing Systems*, 38:9481–9520, 2026.

vLLM project. GuideLLM: Evaluate and optimize LLM deployments. https://github.com/vllm-project/guidellm, 2025. Open-source load generator for LLM serving; the version pinned per item is recorded in the item's environment file.

Anjiang Wei, Allen Nie, Thiago S. F. X. Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving parallel program performance with LLM optimizers via agent-system interfaces. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3h80HyStMH.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Xuan Wu, Di Wang, Chunguo Wu, Lijie Wen, Chunyan Miao, Yubin Xiao, and You Zhou. Efficient heuristics generation for solving combinatorial optimization problems using large language models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 3228–3239, 2025.

Zhiqiang Xie, Ziyi Xu, Mark Zhao, Yuwei An, Vikram Sharma Mailthody, Scott Mahlke, Michael Garland, and Christos Kozyrakis. Strata: Hierarchical context caching for long context language model serving. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1–16, 2026.

Jehyeok Yeon, Ben Rank, and Maksym Andriushchenko. Inferencebench: A benchmark for open-ended LLM inference optimization by AI agents. In *Second Workshop on Agents in the Wild: Safety, Security, and Beyond*, 2026. URL https://openreview.net/forum?id=UNudgkp1Jp.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pp. 193–210, 2024.

Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 337–352. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/zhu-hongyu.

Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, et al. {NanoFlow}: Towards optimal large language model serving throughput. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 749–765, 2025.

A LMSysDojo Details

The companion of §2. §A.1 gives the corpus of 900 setups and the 100-item benchmark in numbers, with the corpus diagnostic behind the $1.075\times$ noise floor; §A.2 the files that make up an item; §A.3 the playground constructor; §A.4 the evaluation environment, its interaction sandbox, checker and scorer; §A.5 how the 100 items were selected, split and labelled; §A.6 the open-ended study; and §A.7 how the benchmark is distributed and refreshed.

A.1 The corpus and the benchmark in numbers

Each of the 900 eligible optimization PRs (Fig. 2; §A.6) yields a setup with its setup card, reconstructed launch configuration, expert trace and baseline profiles. Fig. 7 shows when the reference fixes merged and what the setups contain. The corpus grows with the two engines, and the post-cutoff pool with it. Most setups run on one GPU and a model of 4–10B parameters; dense models

are the majority, with mixture-of-experts and vision-language models making up most of the rest; throughput and inter-token latency are the metrics the PRs most often declare; and the topics run from multimodal and long-context serving to prefix caching, structured output and speculative decoding.

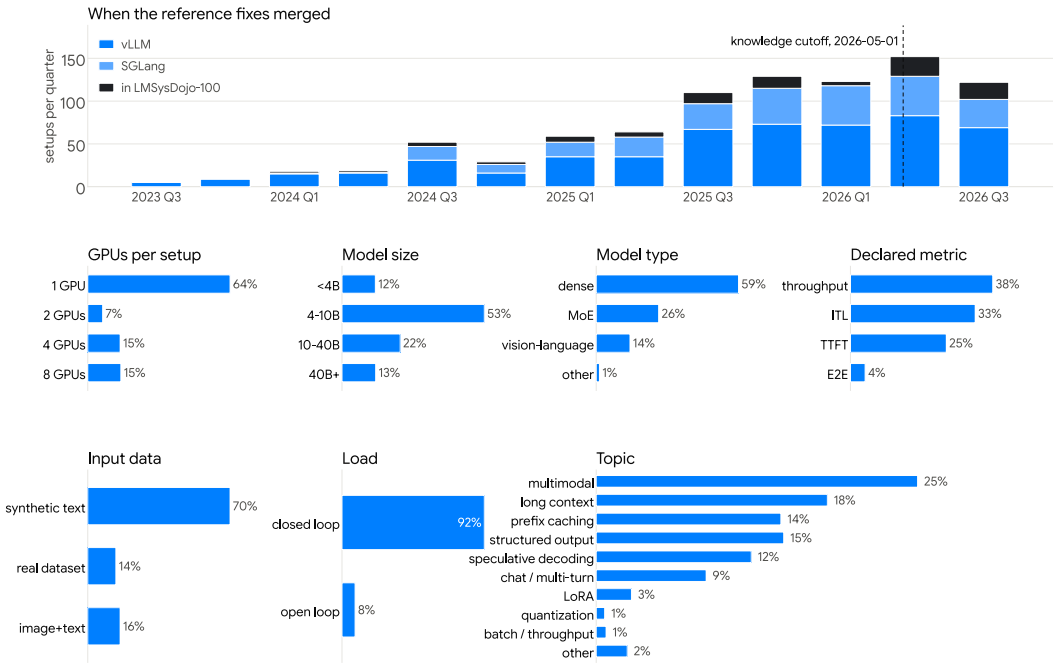


Figure 7: **The corpus.** Top: setups by the quarter their reference fix merged, vLLM and SGLang, with the 100 benchmark items marked and the knowledge cutoff of 2026-05-01. Bottom: the composition of the corpus as shares of the setups with the field recorded: GPUs, model size, model type, first declared metric, input data, load generator and topic.

Room for improvement. Fig. 8a compares the measured gain of the corpus setups on their declared metric with the expert's E2E gain on the 100 items. Room is concentrated: across the corpus the reference speedup has a geometric mean of $1.35\times$ and a median of $1.09\times$, and 47% of the setups gain at least $1.1\times$; the benchmark is drawn from the tail, where the expert's E2E gain has a geometric mean of $1.77\times$ and a median of $1.43\times$ and every item clears the noise floor. Time-to-first-token and end-to-end fixes are larger than inter-token and throughput fixes in the corpus (geometric means $1.60\times$ and $1.62\times$ against $1.22\times$ and $1.27\times$).

The noise floor. We use the corpus's 71 setups with measured ratios below $1\times$ as a diagnostic of measurement variation (Fig. 8b). Two summaries, the mirror ratio $\#\{r < 1/t\}/\#\{r > t\}$ against a keep threshold t (Fig. 8c) and a half-normal fitted to the left side only, $\sigma = \mathrm{median}(|\log r| : r < 1)/0.6745$, supported the $1.075\times$ threshold fixed when the suite was selected. Interpreting this tail as noise assumes approximately symmetric variation around unchanged performance. Workload mismatch and genuine regressions may also contribute, so this operational noise floor is a corpus-level selection rule rather than an independently calibrated error bound for every item or metric.

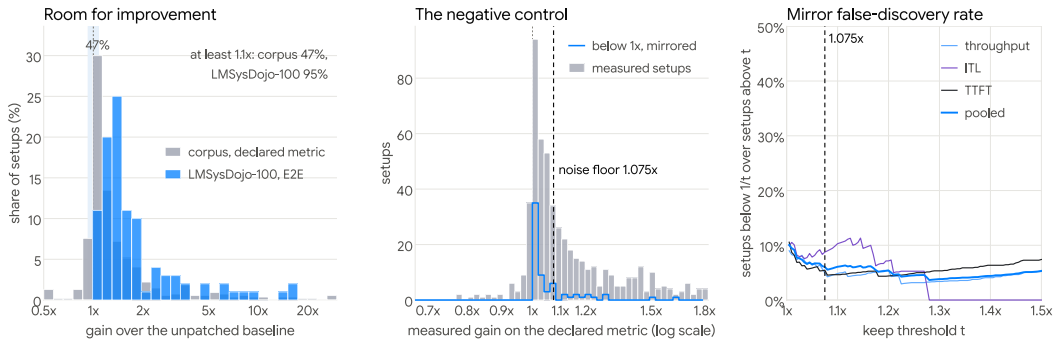


Figure 8: **Room for improvement and the noise floor.** (a) The measured gain of the corpus setups on their declared metric and the expert's E2E gain on the 100 items, as shares of each population; the band marks the noise floor. (b) The corpus diagnostic ("negative control"): gains on a log scale, with the part below $1\times$ mirrored to the right. (c) The mirror ratio ("mirror false-discovery rate"), setups below $1/t$ over setups above t, against the keep threshold t, pooled and by declared metric family.

A.2 The format of an item

An item is a directory. The agent sees `configs/` and the profiles; everything under `oracle/` is withheld and read only by the constructor and the scorer.

item layout

```
<engine>/<tier>/<item>/
  configs/                 what the agent is given
    setupcard.md           the setup in words: model, hardware, workload, purpose
    scenario.json          the GuideLLM workload W: data, sampling, load, bound, seed
    <engine>-config.yaml   the engine's launch configuration, the S side
    env.yaml, venv.txt     the environment and the frozen lock of the arm's interpreter
  profile/                 baseline profiles: guidellm, nsys, torch profiler, py-spy
  oracle/                  withheld
    trace/trace.md         the arms: baseline commit, reference commit, merge date
    trace.diff             the reference fix R
    arm_extra.yaml         reference-only flags, merged into the reference arm
    serving.yaml           the serving mode, engine or HTTP frontend
    verify.json            the eight static checks
    measurement/           the sealed baseline and reference runs
```

A.3 The playground constructor

The constructor of §2 has an offline stage that reads only the repositories and an online stage that runs where the agent will (Fig. 9, Table 3); every stage is a written protocol executed by a pool of agents, and the online numbers are specific to the machine that produced them (§A.4.3).

Scan. There is no keyword filter, because a keyword filter never sees what it drops; agents read every thread in the window, and the threads it links, against five criteria. Under blind re-rating the eligibility verdict agreed on 96.8% of threads. The resulting corpus contains PRs meeting these documentation, hardware and build criteria; the agreement measures consistency of eligibility decisions.

The five scan criteria

A candidate satisfies all five: (1) a start and an end that bound a complete fix, both resolving in the mirror; (2) finding it required analysis, meaning the source records a localisation (a profile, a trace, a line-level attribution) before it names a remedy, however ordinary the remedy; (3) a quantified performance claim, on time spent in work the serving loop performs, read in arm order so that a regression repair is a speedup; (4) doable on one node, $\mathrm{tp} \times \mathrm{pp} \times \mathrm{dp} \leq 8$, where only the mechanism must be reproducible; (5) no compiling, tested on the arms rather than the diff.

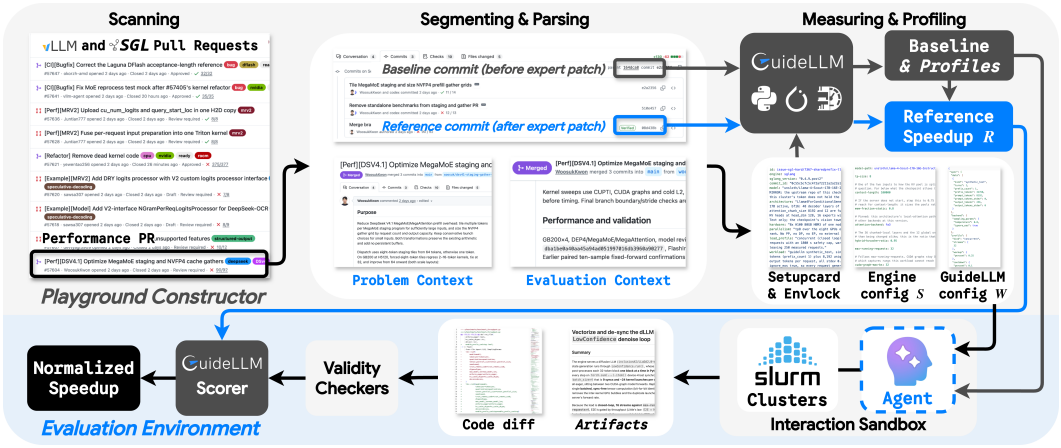


Figure 9: **The LMSysDojo pipeline.** Top: the playground constructor scans the 39,411 merged PRs of vLLM and SGLang, segments each performance PR into the commits before and after the expert's patch, parses its context into a setup card, an engine configuration and a GuideLLM workload, and on the user's hardware measures the speedup of the expert's patch and profiles the baseline. Bottom: the evaluation environment runs the agent in an interaction sandbox on a Slurm-allocated GPU node with the fix sealed away, checks its delivery, and scores it as a normalized gain against the reference.

Segment. The reference is the commit that landed on the default branch and the baseline its first parent, so the arms span exactly one pull request. Blind agreement on the pair is 100%.

Parse and verify. A published setup is used directly; otherwise the agent reads the diff for the conditions under which the changed path is taken and builds the smallest workload that enters it, translated to the target hardware. Items whose fix lives in the HTTP frontend, 8 of the 100, declare `mode: http` in `oracle/serving.yaml`. The card never names the arms, existing numbers, environment or oracle paths, or any prefix of the reference sha; a mechanism that ships behind a default-off flag is enabled through `oracle/arm_extra.yaml`, merged into the reference arm's view at measurement, so the reference arm always exercises its fix. Verify is the well-formedness check of §2, eight assertions filed in `oracle/verify.json`.

Measure. One virtual environment is built per arm from its frozen lock, the engine installed editable with the compiler disabled and the compiled extensions taken from the published wheel. A watchdog ends a run that has completed no request after 900 s.

The acceptance rule for a measurement

One agent owns the item until its measurement reproduces the source's claim or it files an evidenced reason: with $r = \text{measured}/\text{claim}$ it accepts at $r \geq 0.8$, accepts a higher result without tuning it back, accepts $0.5 \leq r < 0.8$ only when a recorded setup difference explains it, and otherwise refines the setup one change at a time, justified by the mechanism and never by moving the number toward the claim.

Because an item enters the benchmark only if its measured reference gain clears the headroom rule (§2), the sealed reference gain is an optimistic estimate of the expert's gain; normalized scores computed against it are correspondingly conservative.

Profile. Four passes run over the baseline arm's setup, one profiler per job because they perturb each other: an un-instrumented `guidellm` reference, `nsys`, a torch-profiler window opened once device allocation settles, and `py-spy`. These are the tools the PR authors themselves used (Table 4); nothing in the passes names the fix.

Table 3: **Constructor stages.** What each stage's agent reads, what it writes, and what happens on failure; the offline stages need no GPU, the online ones run where the agent will. The benchmarked agent receives `configs/` and baseline profiles; everything under `oracle/` is withheld.

stage	reads	decides / writes	on failure
offline: run once against the repositories			
scan	every PR thread in a commit window, with linked threads	candidate or not, by five criteria; `candidates[]`, `rejected[]` with a fixed reason; the window's bounds in `metadata.yaml`	a re-scan appends over `head_commit..HEAD`; built items are left alone
segment	the landing commit, its parents, the mirror	the arm pair, `pre`/`post` in `oracle/trace/trace.md`	a re-cut moves runs to `superseded-*/`; an unsegmentable PR is waitlisted
parse	the source thread, `trace.diff`, the target hardware	`configs/`: `setupcard.md`, `scenario.json`, `<engine>-config.yaml`, `env.yaml`, `venv.txt`; reference-only flags in `oracle/arm_extra.yaml`	a flat or negative measurement returns the item here
verify	`configs/`, the arm shas, the run command	eight static checks in `oracle/verify.json`; nothing is executed	routed back to parse or segment
online: run on the user's hardware			
measure	the built arms, `scenario.json`, `arm_extra.yaml`	both arms interleaved in one allocation; raw reports under `oracle/measurement/prefix/` and `postfix/`; `measured.json` on success, else `measure-attempt.json`; the reference speedup (§2)	a per-item agent refines the setup; an arm-job failed ten times is retired
profile	the measured baseline arm's setup	one profiler per job under `<item>/profile/`; `collect.json` judges each	re-judged from artifacts; ten attempts, then marked

Table 4: **The profiling tools the PR authors used**, over the 176 PRs whose thread or description names one; a PR can name several, so the shares do not sum to 100. Left, the wrap-and-record tools, which attach to a run without a person in the loop and are the ones the constructor runs ahead of time (GuideLLM is the load generator of every measurement, so it is not counted). Right, the interactive tools, which need a person at a REPL or a hand-picked kernel; the agent can still run them itself inside its sandbox, which has a GPU.

wrap-and-record			interactive		
tool	PRs	%	tool	PRs	%
guidellm	–	–	torch.cuda.Event	20	11.4
torch profiler	89	50.6	ncu	11	6.2
nsys	60	34.1	triton do_bench	9	5.1
py-spy	13	7.4	cProfile / line_profiler	3	1.7

A.3.1 THE WORKLOAD AND ITS CAPS

The GuideLLM workload drives both arms and later the delivery through one launcher. The scenario declares a load profile, closed loop at fixed concurrency (89 of the 100 items) or open loop at a rate (11); a bound, a request count with a safety cap or a duration alone (median request bound 506); and a warmup of 10–30% of requests. A global ceiling on the time an arm may run is merged into every arm as the smaller of it and the item's own cap; the scorer measures a delivery under the cap

recorded in the sealed baseline report, so the delivery and the arms it is compared with ran under the same bound.

A.3.2 THE SEALED MEASUREMENT

The measure stage produces the baseline and the reference of every item.

Two arms, one allocation, one tenant. The arms run interleaved in one Slurm allocation, one measurement per node. Before each arm the runner waits for free GPU memory to settle and records the reading; each arm runs against a private view of the item naming its own interpreter, with its side-specific keys merged in; node, job, visible GPUs, driver, torch and CUDA build and UTC time are recorded with every arm.

What is recorded. The run record (per arm: exit code, window, requests served, errored and dispatched, metrics, environment) is `measured.json` only when every arm exited cleanly, served and has a usable TTFT column. The five metrics are GuideLLM's request latency, time to first token, inter-token latency, output tokens per second and requests per second, as means over successful requests in the window; RPS is derived as served requests over the active span, first request start to last request end, because the load generator's own figure divides by the whole run including an idle tail that varies between runs.

The golden check. A measurement is data only once it passes the golden check: the run's own record says success, both arms served, neither errored more than 2% of its window, the arms ran the same prompts (token means within 1%), some metric's gain clears the noise floor, and where a second repeat exists the two agree within it. Served counts are reported but not required to match, because the faster arm completes more requests inside a bounded window.

A.4 THE EVALUATION ENVIRONMENT

The evaluation environment of §2 has three parts: the interaction sandbox and what keeps the agent from the answer (§A.4.1), the checker's validity rules and anti-cheat checks (§A.4.2), and the scorer, which turns a measurement into the normalized gain (§A.4.3).

A.4.1 THE INTERACTION SANDBOX

The agent receives what §2 lists; the eight rules and the 1.5-hour wall clock per attempt are in §C.2 and §C.1. Together with item construction (§A.3), the sandbox is designed to eliminate any information leakage from the originating PR: all PR text, issue threads, git history, and reference diffs are excluded, and all network access is blocked, so from the model's perspective it is simply evaluating and optimizing a standalone live deployment (while the post-cutoff held-out split of §A.5 controls for pre-training memorization). Containment is four layers, staging, tool restrictions, process isolation and audit, rather than an OS sandbox, so that the agent keeps direct access to the node's GPUs; the measurement environment is shared with the agent, since a changed environment would be a different measurement.

Staging and the redactor. The workspace is built from named pieces of the item: `configs/` copied, never linked; the dataset and profiler passes linked read-only; the baseline arm's interpreter as `venv`; a redacted corpus root; and the engine checkout. Nothing under `oracle/` is staged; the card's front matter drops every key whose name begins with `oracle`, `tier_note`, `type` or `tier`, comments are stripped from the staged YAML, and answer-bearing names (`trace`, `closure`, `postfix`, `solution.md`, `provenance.md`, `measurement`) are checked against every resolved path in the workspace, following symlinks, after staging and after the run.

The seal. Staging checks the engine out against the mirror's object store through an alternates file rather than cloning it. The seal then deletes every ref, writes `HEAD` to `.git/shallow`, repacks what is reachable into a local pack (one commit) and removes the alternates file, after which the reference sha resolves to no object in the workspace.

Denied paths and commands. The CLI runs under an allow list of ten tools and a deny list. Denied by name is every tool that routes around the environment (delegation, a second interpreter, worktree changes, deferred execution, URL fetch, web search, network requests, asking a human); denied to the read tools are the repository, the corpus, the item's directory, the run tree, every arm's withheld side, other agents' workspaces and the engine mirrors; and thirty-three shell prefixes are refused: network binaries, git's remote and history commands, package managers and the scheduler. What carries the weight is the answer's absence from the workspace and the audit.

Safe mode and credentials. Sessions start in safe mode with no setting sources but the environment's own settings file, so no operator-level instruction, skill or memory enters the measurement; every inherited variable whose name matches a credential shape is blanked, except the families the CLI authenticates with; and each session's process group is signalled TERM then KILL on timeout, so an engine holding a GPU does not outlive its CLI.

The audit. After every run the audit reads every tool call from the CLI's own transcript, resolves every shell command's paths as the kernel would and classifies them against the roots that hold the answer, re-checks the workspace for reachable answers, searches every transcript, delivery and analysis for the reference sha at every length git would abbreviate it, and asks the workspace's object store whether it can still resolve the reference commit. The scorer repeats the check, and a delivery whose transcript reached a withheld path is invalid; its attempt receives zero gain.

A.4.2 THE CHECKER

The checker of §2 decides whether a measurement happened and whether a delivery may be scored. No model judges anything: every check is arithmetic over file and hunk names, a regex over the diff, or a comparison of two reports.

Validity rules.

- *Served count.* The served count, not the exit status, decides whether a measurement happened, since the load generator writes a well-formed report even when the engine never serves.
- *Corrupt metrics.* A TTFT median of exactly 0.0 with requests completed is a missing event, not speed, and marks the column unusable.
- *Arm matching.* Request-bound runs must match the set of dispatched requests, time-bound runs the measurement window, which must lie inside the run.
- *Mechanism enabled.* Reference-only flags are stored with the withheld answer and merged into the reference arm at measurement, so a mechanism behind a default-off flag is exercised rather than reported at $1.00\times$.
- *Single tenancy.* One measurement per node: co-tenancy leaves TTFT within 0.2% but shifts per-token latency by 3.4%, three times the within-node noise.
- *Measured tree.* A startup hook puts the patched tree first in every interpreter that inherits the path, the load generator's worker and the engine core included, ahead of the arm's editable install.
- *Load shape.* Open-loop scenarios fix their arrival schedule at launch; an item whose sealed arms and evaluation runs disagree on arrival delay is excluded.

Anti-cheat checks. The static checks run in the scoring pass and the one live check beside the measurement; Table 5 lists them. `workload` catches "faster because it did less", which an output probe cannot see: a delivery that shortens its outputs improves mean ITL, and the pinned-length rule is what catches it. `configdrift` rejects pure knob tuning, while `activation` records the kind of source change, including rewiring and flag activation; mixed changes are retained and labelled. The sealed setup itself may not be edited (§C.2). These checks are calibrated so that all 100 upstream fixes pass. The apparatus rules exist because the agent's session measures with its own copy of `measure.sh`; the two bounds are relative to the item because some reference fixes reach double-digit speedups. The checks test workload matching and equivalence on the fixed output

probe; whether a change is one the upstream project would merge is a judgment the benchmark does not make (§E.4).

Table 5: **Checks on a delivery.** `validity` composes them into one verdict: `pass`, `fail` with the reasons recorded, or `no-delivery`.

check	what it computes	what fails
`workload`	the shape of the delivery's run against the baseline's over the dispatched requests: served, errored and incomplete counts, prompt and output tokens per request	fewer requests ($> 5\%$, request-bound baseline), shorter outputs or prompts ($> 2\%$ on a pinned length, 3σ of the baseline's spread otherwise), more errors, longer outputs on a pinned baseline; longer prompts are noted
`configdrift`	lines that only move a literal in a configuration file, or set a named engine knob or a `VLLM_*`/`SGLANG_*` variable anywhere	`knob-tuning`; `mixed` is reported and does not fail; an empty diff is `no-delivery`
`activation`	identifiers the diff newly defines against those it newly calls, binds, imports or names that the baseline tree already defined, and whether those are named in a configuration file	nothing; `new-implementation`, `reimplementation`, `rewiring` or `flag-flip` is reported beside the gain, and a delivery may not present one kind as another
`novelty`	Jaccard overlap of the delivery's and the reference fix's Python files and (file, function) hunks	nothing; `same-fix`, `same-region`, `disjoint` are indicators
`equivalence` (live)	100 frozen prompts at temperature 0, fixed seed, 32 tokens, `ignore_eos`, top-5 logprobs, sent before the benchmark; token ids and logprob deltas against the baseline's probe	a token differs or a logprob moves beyond tolerance
apparatus	digests of `configs/`, `measure.sh`, `hook/`, `pylib/` against what the environment wrote; import shadows; index entries marked unchanged; ignored-but-live files	any edit to the apparatus or the question; any change the diff does not show
bounds	the delivery's geometric-mean gain against the reference's; each scored metric against the unpatched baseline	`catastrophic-regression`, a metric the reference did not itself worsen made more than $5\times$ worse; `beyond-reference`, a gain above $3\times$ the reference's on a non-degenerate item, does not fail but is capped at $3\times$ for numerical stability

How outcomes are recorded. An attempt ends `delivered`, `abandoned` or `failed`; failed covers exceeded sessions and invalid deliveries. *Abandoned*, a manifest saying that nothing here should change, scores as an unchanged tree, a normalized gain of 0. *Exceeded* is a session that reached its wall clock, classified by the session record. *Invalid* is a delivery that `validity` fails: counted, with its reasons recorded, and given up to three chances in all to pass the checker; the chances are checker passes, not extra scored measurements, and apply to validity failures only. One still invalid after them scores as no change. Exceeded attempts likewise score as no change. An absent check record is reported as a rule that did not run, never as one that passed. Table 6 reports the rates.

A.4.3 THE SCORER

The scorer of §2 measures the delivery on a second node and turns the measurement into the normalized gain; this part records how, and what the score's noise depends on.

Measuring a delivery. Scoring is a second pass on a node the agent never held. The environment commits the delivered tree and writes `patch.diff` against the baseline commit with git's diff

Table 6: **Outcome rates.** Over all 100 items and every agent launch: *abandoned*, the agent finished without delivering; *exceeded*, the session reached its wall clock; *invalid*, delivered attempts that failed a check, as a share of delivered attempts; Δserved, mean absolute drift of the delivery's served count from the sealed baseline's.

setup	abandoned	exceeded	invalid	Δserved
Vanilla	10.3%	8.4%	0.3%	1.40%
Gap analysis	6.5%	6.8%	1.0%	1.51%
Structured gap analysis	7.3%	4.7%	0.3%	1.57%

drivers pinned; the scorer refuses a tree that is dirty or not at the committed sha, rebuilds `configs/` from the sealed item under the same redactions, restores `measure.sh` and the import hook, merges the baseline-side `arm_extra.yaml` keys and serving mode, asserts an empty node and GPU, and runs the workspace's own `./measure.sh` at full length under the sealed report's cap (§A.3.1). The import hook, which puts the patched tree first as the measured-tree rule requires, with the compiled extensions copied from the arm, also applies three translations to every measurement alike, restoring what the GuideLLM build that sealed the arms did: a renamed structured-output key spelled in the engine's vintage, a declared `ignore_eos` outranking a per-request override, and a multi-turn item's history replayed.

The score. The scorer divides the delivery's metrics against the sealed baseline's and the reference's, writing the normalized gain (§2) for every declared metric; a gain above three times the reference's is capped there for numerical stability, and an item's normalized gain on a metric is left ungraded when the reference's gain on that metric is under the noise floor; split entries use the speedups of all items. Per item the attempts' speedups, including $1\times$ for attempts without a valid delivery, are averaged (`mean@N`); a split's entry in Table 1 is $(S-1)/(H-1)$, with S the geometric mean over the split's items of that per-item speedup and H the geometric mean of the expert's speedup on the same items.

The metrics. Every item is measured on the five metrics NVIDIA's LLM benchmarking guide defines (NVIDIA, 2026), read from the load generator's report over the successful requests of one run. For request i with n_i output tokens, the run records the time to first token TTFT_i and the gaps between consecutive tokens, whose mean is the inter-token latency; the end-to-end latency is their sum,

$$\mathrm{E2E}_i \;=\; \mathrm{TTFT}_i + (n_i - 1)\,\mathrm{ITL}_i. \tag{1}$$

Arm-level latencies are means over requests. The throughputs are counts over the measurement window T, from the first request to the last response, $\mathrm{RPS} = M/T$ for M served requests and $\mathrm{TPS} = \sum_i n_i/T$ output tokens per second; under a closed loop of c concurrent streams Little's law gives $\mathrm{RPS} \approx c/\overline{\mathrm{E2E}}$ and $\mathrm{TPS} \approx \bar{n}\,c/\overline{\mathrm{E2E}}$, with $\bar{n}$ the mean output length, so they are the reciprocal of end-to-end latency up to constants of the workload, while under an open loop at arrival rate λ, $\mathrm{RPS} = \lambda$ and $\mathrm{TPS} = \lambda\bar{n}$ whenever the engine keeps up, whatever the patch did. Eq. 1 likewise makes E2E a share-weighted combination of the two phase latencies: a decode fix, which leaves TTFT where it was, moves E2E only through the decode share.

Measurement variation. The constructor runs the two sealed arms interleaved in one allocation; the same baseline measured 1–3% apart across nodes of one type and about 1% within a node. Single tenancy (§A.4.2) controls co-tenancy, but the agent's arm is measured later in a separate allocation of the same hardware type, so its comparison with the sealed arms retains node and run variation, which the headroom threshold does not remove. The normalization supplies an expert-relative scale on each setup; since a mechanism's relative benefit can change with the hardware, it does not guarantee hardware-invariant rankings. With independent relative noise of standard deviation ς on each arm, G and G_R the agent's and the expert's speedups and N the normalized gain,

$$\mathrm{sd}(N) \;\approx\; \frac{\varsigma\sqrt{G^2 + N^2 G_R^2 + (1-N)^2}}{G_R - 1}, \tag{2}$$

so the score's noise is the arm noise divided by the item's headroom. This is why the headroom rule of §2 is enforced on the reference and why items are averaged. The rule is enforced on end-to-end

latency, which every selected reference improves: on the other four, G_R is near 1 wherever the fix did not move that metric, an open-loop deployment for the throughputs, a decode fix for TTFT, and Eq. 2 makes those columns correspondingly noisier.

A.5 BUILDING THE 100-ITEM BENCHMARK

The selection of §2 is one set of rules applied identically on both sides of the cutoff, followed by the split, the labelling of the judged fields, and the description of the result by split.

The selection rules

One list, applied identically on both sides of the cutoff: both arms measured with a readable E2E; at least 100 requests served on each arm, because a mean over a few dozen requests is noisy; TTFT and ITL present on both arms; a measurement the constructor accepted; measured E2E gain at least the noise floor of $1.075\times$; exact deduplication on (workload, model, GPUs), with nothing on one side sharing a key with the other; and no held or waitlisted item. The bar is on our measured E2E only; the upstream claim is not used for selection.

Splits. The cutoff is 2026-05-01 on the reference commit's date. The post-cutoff side is a census of every post-cutoff item that passes the selection rules and the golden check. The pre-cutoff side is a stratified random sample of the qualified pool, strata being workload bucket $\times$ engine with targets from the post-cutoff marginals, seed 20260902; the qualified pre-cutoff pool held too few SGLang items to meet the engine targets, so the pre-cutoff side stays vLLM-heavy and its workload mix departs from the post-cutoff one where the pool fell short (Table 7). The 65 pre-cutoff items were divided at random into the 30-item development split, the only set on which prompts were tuned, and the 35 held out beside it. This is a temporal split by merge date. Earlier items may appear in the model's training data, while a later merge can still contain a PR discussion or proposed patch made public before the cutoff. Establishing absence of prior exposure would require an audit of the earliest public solution-bearing artifacts. We report both held-out splits separately.

The benchmark by split. Table 7 describes the 100 items by split, Fig. 10 their composition and Fig. 11 the reference fix. The pre-cutoff held-out split is almost entirely vLLM (33 of 35 items) and mostly dense; the post-cutoff split is richer in SGLang, mixture-of-experts and vision-language items, and its reference gains are smaller, which is why every result in the paper is the normalized gain on the same item and the splits are reported side by side. Fifteen of the 100 fixes touch CUDA or C++; 95 were judged reproducible in Python alone. The remaining five retain the expert's measured intervention as a reference, without an established Python-only implementation of that fix within the agent's edit scope.

The judged fields. The judged fields, the rows of Table 7 marked † together with the phase, fix-subsystem and Python-only labels behind Fig. 2 and §2 and the upstream-evidence, claimed-speedup and quantization labels, were labelled under a written protocol from the setup card, the scenario, the withheld problem and solution statements and the trace's file list, by two independent raters in batches of five with different batch compositions. Raters agreed on 98–100% of items for most fields and 94% for fix kind and claimed speedup; disagreements went to an adjudicator. A rater returns a category the paper counts, never a number it computes.

Table 7: **The LMSYSDOJO-100 suite by split.** Cells are counts (percent) for shares, counts for slash rows, median [IQR] or geometric mean. † marks fields judged by two raters and adjudicated (§A.5). p compares the two held-out splits (Fisher exact for shares, Mann–Whitney for medians and geometric means).

	dev	held-out		
	(n=30)	pre-cutoff (n=35)	post-cutoff (n=35)	p
Provenance				
engine: vLLM / SGLang	21 / 9	33 / 2	22 / 13	**<.01**
fix merged (upstream PR), range	2024-03 – 2026-04	2024-03 – 2026-04	2026-05 – 2026-08	
model family: Llama / Qwen / other	7 / 10 / 13	16 / 11 / 8	5 / 20 / 10	**<.01**
Model and hardware				
total params (B), median [IQR]†	8.0 [6.7, 15.8]	8.0 [6.7, 8.0]	8.0 [4.0, 30.0]	0.30
params $\geq$ 10B†	12 (40)	6 (17)	15 (43)	**0.04**
MoE model†	13 (43)	8 (23)	11 (31)	0.59
vision / audio tower†	6 (20)	6 (17)	14 (40)	0.06
multi-GPU (TP/PP/DP $>$ 1)	6 (20)	6 (17)	8 (23)	0.77
Workload				
bucket: prefill / decode / multimodal†	8 / 10 / 5	5 / 12 / 6	8 / 9 / 6	0.54
bucket: prefix-cache†	0 (0)	3 (9)	7 (20)	0.31
bucket: chat/mixed†	2 (7)	6 (17)	0 (0)	**0.02**
bucket: other (reasoning, spec-decode, structured)†	5 (17)	3 (9)	5 (14)	0.71
open loop (Poisson / constant rate)	2 (7)	6 (17)	3 (9)	0.48
concurrency (closed loop), median [IQR]	26 [2, 112]	32 [2, 256]	13 [4, 64]	0.34
prompt tokens (synthetic text), median [IQR]	512 [100, 2048]	215 [94, 512]	512 [192, 2048]	**0.03**
ISL:OSL prefill-heavy / decode-heavy	10 / 7	11 / 6	19 / 2	0.09
Scored metric				
declares TTFT / ITL / E2E / throughput (p: throughput)	14 / 19 / 7 / 16	19 / 22 / 9 / 16	19 / 17 / 5 / 23	0.15
Human reference				
E2E gain, geomean	1.81x	2.25x	1.37x	**<.01**
TTFT / ITL / TPS gain, geomean	2.14 / 1.45 / 1.59	2.42 / 1.83 / 1.97	1.28 / 1.33 / 1.34	**0.03**
E2E gain, median [IQR]	1.50 [1.32, 2.28]	1.56 [1.26, 3.10]	1.31 [1.14, 1.52]	**<.01**
E2E gain, min – max	1.10 – 8.01	1.12 – 16.98	1.08 – 2.70	
E2E gain $<$ 1.15x	2 (7)	2 (6)	10 (29)	**0.02**
E2E gain $\geq$ 1.8x	12 (40)	14 (40)	3 (9)	**<.01**
Reference fix (the upstream PR)				
lines added + removed, median [IQR]	144 [27, 330]	100 [34, 283]	212 [114, 537]	**0.02**
patch touches tests	14 (47)	15 (43)	26 (74)	**0.01**
fix kind: algorithm / data structure†	2 (7)	0 (0)	7 (20)	**0.01**
closure anchored by a request issue	4 (13)	11 (31)	4 (11)	0.08

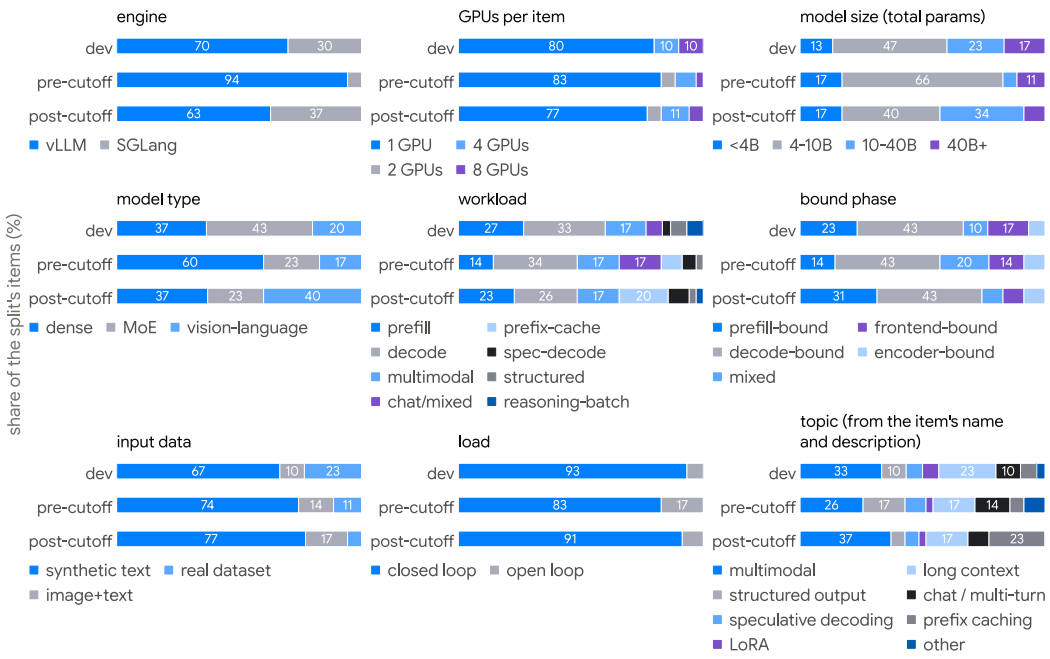


Figure 10: **Composition of the suite by split.** Share of each split's items by engine, GPUs, model size and type, workload, the phase that bounds the baseline, input data, load generator and topic.

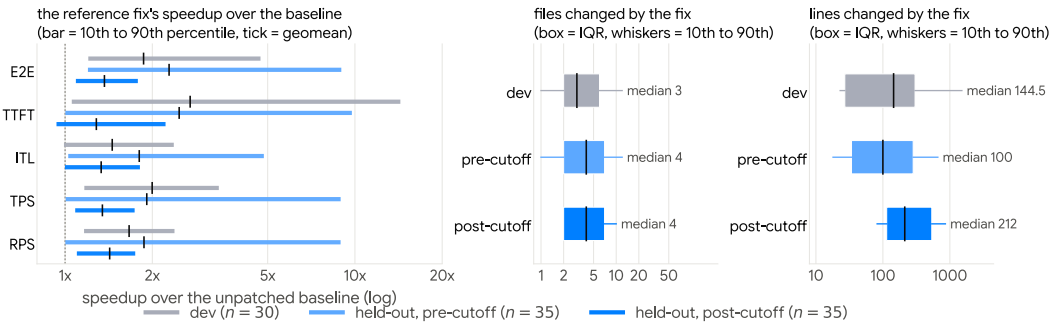


Figure 11: **The reference fix, by split.** Left: its speedup over the baseline on each metric, 10th to 90th percentile with the geometric mean marked. Right: files and lines changed, the interquartile range with 10th-to-90th whiskers and the median marked.

A.6 THE OPEN-ENDED STUDY

Before the constructor existed we asked what agents find when they choose the setup themselves. Each agent, Claude Code on Opus 5, was given a pinned engine release (vLLM v0.23.0 or SGLang v0.5.13) on an exclusive 8×H100 node, a benchmark script it could run at will, no time limit, and one instruction, to keep improving the engine's performance; it chose the model, the workload and the launch configuration itself and iterated on the code against them. An attempt is a patch the agent declared finished, scored as the benchmark scores an item: the mean E2E latency of the patched engine on the agent's own setup against the unpatched engine under its best configuration, so that configuration tuning earns nothing; an attempt counts as above noise only if it clears the $1.075\times$ noise floor of §A.1. The agents launched over a thousand GPU evaluations and wrote about two thousand commits; of the 150 attempts they declared finished, 121 on vLLM and 29 on SGLang, none cleared the floor (Table 8a), the best landing at $+2.6\%$ and $+2.4\%$. The null cannot be read without knowing whether the setups had room and without a reference to read a gain against, which is the gap the constructor closes by taking both from a merged fix (Table 8b; §2).

Table 8: **Open-ended optimization and eligible opportunities in engine history.** (a) Open-ended attempts on a pinned engine, scored on mean E2E latency against an optimally configured baseline; none of the 150 attempts clears the $1.075\times$ noise floor. (b) Every pull request merged into the two engines by 2026-09-05, and the share the scan admits as eligible optimization PRs (§A.3).

(a) Open-ended optimization

engine	attempts	median	best	above noise
vLLM	121	$+1.8\%$	$+2.6\%$	0
SGLang	29	-0.8%	$+2.4\%$	0
total	150			**0 (0%)**

(b) Upstream history

repo	merged PRs	eligible PRs
vLLM	20,969	607 (2.9%)
SGLang	18,442	293 (1.6%)
total	**39,411**	**900 (2.3%)**

A.7 DISTRIBUTION, LICENSING AND REFRESH

The corpus and the benchmark derive from the public pull requests of vLLM and SGLang, both released under the Apache 2.0 license; each item records the pull request, the two commits and the merge date, and ships the constructor's offline outputs, the sealed measurements and the baseline profiles, so an item can be rebuilt on other hardware by re-running the online stage. The toolkit, the benchmark and the corpus of 900 measured setups will be released after the review period. The constructor's offline stage ingests new merged PRs by extending its commit window and leaves built items alone (Table 3), so merge dates support refreshing the temporal split for other models.

B GAP ANALYSIS: FORMAL ACCOUNT AND THE 50-PAPER STUDY

The formal account behind gap analysis (§3): the benchmark's objective and an item-level model of choosing among analyses (Eqs. 3 and 7), which separates the two directions of more analyses and better analyses; when sampling the analysis pays (§B.2); a test of the model's selection curve on the held-out splits (§B.3); and the 50-paper study behind gap analysis and its structured form (§B.4, Table 9).

The sampling view. For each item an agent induces a distribution over analyses, and every method seeks to raise the quality of the candidate ultimately implemented; the oracle in Fig. 6 is the best outcome in the sampled pool. In the terms of §3, gap analysis lets us draw from that distribution without building a final patch; parallel gap analysis draws K times and picks by the prediction; structured gap analysis shifts the distribution and exposes what each prediction rests on; and iterative gap analysis requests a new analysis when a candidate fails a check. The distance between the pick and the oracle is the headroom that remains. The rest of this section makes the view precise.

B.1 OBJECTIVE AND THE VALUE OF PARALLEL GAP ANALYSIS

An item fixes a system S, an engine's source at a commit with its launch configuration, and a setup W, the workload it serves and the hardware it runs on (§2). An agent is a stochastic policy $\pi(\Delta \mid S, W)$ that produces a patch Δ. For a declared metric, write $G_i(\Delta)$ for the speedup the scorer assigns on item i after its validity rules and cap (§A.4.3), H_i for the expert's speedup, and $g_i(\pi) = \mathbb{E}_{\Delta\sim\pi}[G_i(\Delta)]$; a failed or abandoned attempt contributes $G_i = 1$. With $\mathcal{D}$ uniform over the split's scored items, the benchmark objective follows the aggregation of §A.4.3, averaging attempts' speedups within an item, taking their geometric mean across items and normalizing by the expert's geometric-mean gain:

$$\max_{\pi}\ J(\pi)\ =\ \max_{\pi}\ \frac{\exp\big(\mathbb{E}_{i\sim\mathcal{D}}[\ln g_i(\pi)]\big)-1}{\exp\big(\mathbb{E}_{i\sim\mathcal{D}}[\ln H_i]\big)-1}. \tag{3}$$

The model below works one item at a time. Define the item's normalized outcome $P_i = (G_i - 1)/(H_i - 1)$ and its expectation $J_i(\pi) = \mathbb{E}_{\Delta\sim\pi}[P_i]$; then $g_i(\pi) = 1 + (H_i - 1)J_i(\pi)$ links the item to Eq. 3. For a fixed item we drop the index, write $\mu_\pi := J_i(\pi)$ and $\sigma_\pi^2 := \mathrm{Var}_{\Delta\sim\pi}[P]$, and view the policy as a pool of candidate patches carrying the outcomes they would deliver.

Scope of the model. The account makes two approximations, stated once here. It is item-level: the objective of Eq. 3 is a nonlinear aggregate, not the mean of the J_i, so an item-level result reaches the benchmark score only through g_i and the geometric mean. And it treats score and outcome as jointly Gaussian, whereas measured outcomes are heavy-tailed and include no-change outcomes. We use the model for the shape of the selection curve and for the quantities that shape depends on, and test how well the shape transfers in §B.3.

Keeping the best of K. Draw K candidates independently from the policy and keep one. A *score* $\hat{s}(\Delta_k)$ available before evaluation lets the agent keep the highest-scored candidate k^* and evaluate only that one. Under the Gaussian model the kept candidate's expected outcome is

$$\begin{aligned} J_i^{(K)}(\pi, \hat{s})\ &=\ \mathbb{E}[P_{k^*}]\ =\ \mu_\pi + \rho\,\sigma_\pi\,c_K, \qquad k^* = \arg\max_{k\le K} \hat{s}(\Delta_k), \\ \rho\ &=\ \mathrm{corr}\big(\hat{s}(\Delta),\,P\big), \qquad c_K = \mathbb{E}\big[\max_{k\le K} Z_k\big],\ Z_k \overset{\text{iid}}{\sim} \mathcal{N}(0,1), \end{aligned} \tag{4}$$

where c_K is the expected largest of K standard normal draws: $c_1 = 0$, $c_2 = 0.56$, $c_3 = 0.85$, $c_5 = 1.16$, $c_{10} = 1.54$. This is the Gaussian regression of outcome on score, $P_k = \mu_\pi + \rho\,\sigma_\pi Z_k + \varepsilon_k$ with Z_k the standardized score and ε_k independent of every Z, evaluated at the largest Z. The term $\sigma_\pi c_K$ is the value of ordering the K candidates perfectly, and ρ, the *ranking precision* of the score, is the fraction of it the score collects. The single agent (VANILLA) states no predicted gain, so without a score its pick is random and $J_i^{(K)} = \mu_\pi$ at every K, the level the unselected running means of Fig. 14 settle toward. A patch's own measurement can serve as a score only once the patch has been built and measured, the measured-patches route of Eq. 8.

Gap analysis as a two-stage policy. Gap analysis (§3) draws a candidate in two stages: the analyzer π_A draws an analysis a, then the implementer π_I draws a patch Δ given a. Let $v(a)$ be the value of an analysis, the expected outcome of a patch implemented from it, and $\hat{v}(a)$ its predicted gain. The law of total variance splits the pool's spread into the analyses differing and the implementer differing on a fixed analysis:

$$\pi(\Delta \mid S, W)\ =\ \int \pi_I(\Delta \mid a, S, W)\,\pi_A(a \mid S, W)\,da, \quad \sigma_\pi^2\ =\ \underbrace{\mathrm{Var}_a[\,v(a)\,]}_{\sigma_A^2}\ +\ \underbrace{\mathbb{E}_a[\,\mathrm{Var}(P \mid a)\,]}_{\sigma_I^2}. \tag{5}$$

Fig. 4 gives a descriptive counterpart on the scale of $\ln G_i$. With $\mathrm{sd}_i^{\mathrm{fix}}$ the within-item standard deviation over three implementer runs on one fixed analysis and $\mathrm{sd}_i^{\mathrm{full}}$ that over three runs of the whole pipeline, the 97–98% reduction is

$$q = 1 - \left(\frac{\mathrm{median}_i\,\mathrm{sd}_i^{\mathrm{fix}}}{\mathrm{median}_i\,\mathrm{sd}_i^{\mathrm{full}}}\right)^2\ \approx\ 0.97\text{–}0.98. \tag{6}$$

The statistic q in Eq. 6 summarizes typical within-item spread through medians of log speedups; it is not an estimate of the share σ_A^2/σ_π^2 in Eq. 5, which averages conditional variances of the normalized outcome.

The score belongs to the analysis, $\hat{s} = \hat{v}(a)$, while the outcome is the analysis's value plus implementer noise, $P = v(a) + \eta$, with $\mathbb{E}[\eta \mid a] = 0$ and $\mathbb{E}[\eta^2] = \sigma_I^2$. The zero conditional mean makes η uncorrelated with the score, so $\mathrm{cov}(\hat{v}(a), P) = \mathrm{cov}(\hat{v}(a), v(a))$, hence $\rho\,\sigma_\pi = \rho_A\,\sigma_A$, and $\mu_\pi = \mu_A := \mathbb{E}_a[v(a)]$. Substituting into Eq. 4 gives the item-level value of parallel gap analysis (§3):

$$J_i^{(K)} = \underbrace{\mu_A}_{\text{pool quality}} + \underbrace{\rho_A}_{\text{ranking precision}} \sigma_A\, c_K, \qquad k^* = \arg\max_{k \le K} \hat{v}(a_k), \qquad \rho_A = \mathrm{corr}\big(\hat{v}(a),\, v(a)\big). \tag{7}$$

Eq. 7 separates the two directions of §3. More analyses act through c_K, which grows with K (parallel gap analysis). Better analyses act through the pool quality μ_A, the mean value of the analyses drawn, the spread σ_A of their values, and the precision ρ_A with which their predictions order them (structured and iterative gap analysis). Selecting on analyses also gives up little against selecting on built patches when the implementer's noise is small: a perfect score on patches exceeds a perfect score on analyses by $(\sigma_\pi - \sigma_A)\,c_K = \sigma_I^2 c_K/(\sigma_\pi + \sigma_A)$, second order in σ_I. The fixed-analysis experiment is consistent with this regime, although q does not identify the variance components.

B.2 Why sample the analysis

Eq. 5 factorizes the policy at the analysis. Within gap analysis the single-draw expectation is $\mu_\pi = \mu_A$ by construction; whether it matches the single agent's is empirical, close for Opus at $K{=}1$ (Table 1) and 10.5 points higher for Sonnet (Table 15). The analysis carries its signal, the predicted gain, before a final patch is built or evaluated, and fixing the analysis leaves far less run-to-run spread (Fig. 4). Both point to the analysis as the stage to sample and select.

Sampling the analysis pays when its predicted gain ranks outcomes and the extra analyses cost less than the implementations and evaluations they replace. Write κ_A, κ_I and κ_M for the cost of one analysis, one implementation and one measurement of a built patch on the target deployment, and $\rho_M \le 1$ for the ranking precision of that measurement used as a score, which its noise limits (Eq. 2). For candidates from the same policy, the item-level model gives the cost (left) and the expected outcome (right) of choosing among K analyses, of which only the chosen one is built and measured, and among B patches, each built and measured:

$$\begin{aligned} \text{analyses:} \quad & K\kappa_A + \kappa_I + \kappa_M, && \mu_A + \rho_A\,\sigma_A\,c_K, \\ \text{measured patches:} \quad & B\,(\kappa_A + \kappa_I + \kappa_M), && \mu_\pi + \rho_M\,\sigma_\pi\,c_B. \end{aligned} \tag{8}$$

In our setting ($B{=}1$; §3), each added analysis costs only κ_A while avoiding $\kappa_I + \kappa_M$ for unselected candidates. With $\sigma_A \approx \sigma_\pi$, the regime above, the two routes reach the same expected outcome when $\rho_A c_K = \rho_M c_B$. At $K{=}10$ the precision fitted on the development split (§B.3) puts $\rho_A c_{10}$ at about c_2: ten analyses picked by their predictions match the expected gain of the better of two noiselessly measured patches while requiring only one final implementation and evaluation, and the analysis route keeps improving with K (Fig. 14).

B.3 The selection model tested held-out

We test the curve shape of Eq. 7 at the split level. The ranking precision ρ_A is the only fitted quantity: it is estimated on the 30 development items and carried unchanged to the held-out splits, whose own measured pools supply μ_A and σ_A (Fig. 12). The test asks whether the selection response transfers given each split's pool; it does not predict the pools themselves.

The fitted response transfers across splits. Carried from the development split, ρ_A reproduces the held-out curves of structured gap analysis with RMS error 1.1 and 2.3 points on the pre- and post-cutoff splits. The development split itself fits worse, at 4.5 points; since only ρ_A is fitted, this error reflects the shape of the curve rather than overfitting: there, and to a lesser degree on the post-cutoff split, the observed curve keeps rising where the Gaussian best-of-K term flattens. Free-form gap analysis fits less well on the development and pre-cutoff splits (RMS 5.6 and 3.8 points) and

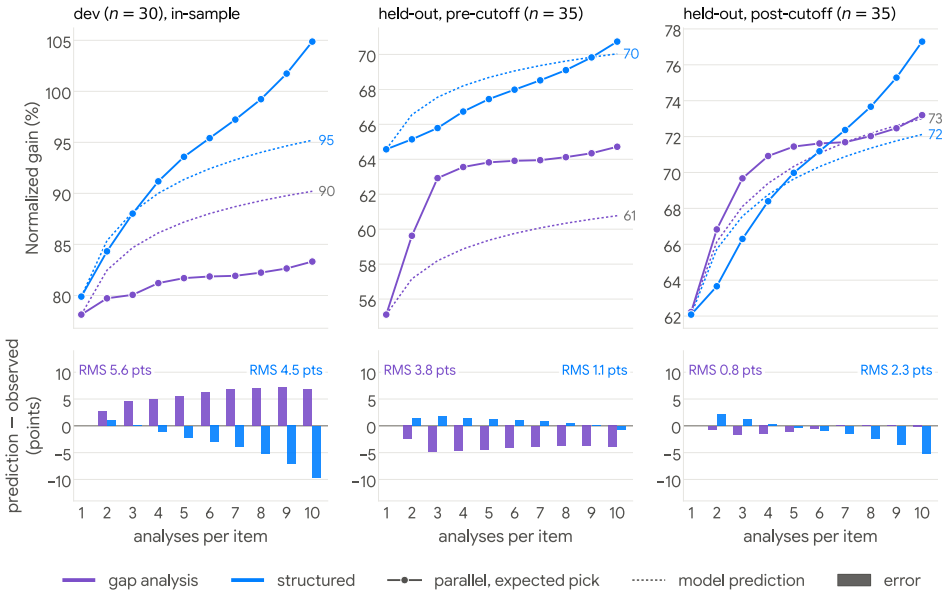


Figure 12: **Transfer of the fitted selection response, given each split's measured pool.** Dotted: the curve $\mu_A + \rho_A \sigma_A c_K$ of Eq. 7, with the ranking precision ρ_A estimated on the 30 development items only and each split's own μ_A and σ_A from measured outcomes. Solid: the observed pick of parallel gap analysis in expectation over the draw of K analyses, as in Fig. 14. Below: predicted minus observed, with the RMS error over K.

better on the post-cutoff split (0.8). The fitted response is thus a usable summary of selection within the observed pools.

Where structure acts. Paired per item at $K{=}10$, structure raises the weaker pool (pre-cutoff, $+6.7\%$ in log geometric mean of the sampled analyses) and sharpens the pick on the development split ($+3.7\%$ closer to the item's best). These are the two routes by which better analyses act in Eq. 7, pool quality and ranking precision, and they match the pool means and oracle gaps of Fig. 6.

B.4 Gap analysis in systems research: the 50-paper study

Resource trades. A deployed system uses memory capacity, execution time on its compute units, link and I/O bandwidth and provisioned capacity. We use the *resource trade* as an accounting view of its optimizations: relate a saving on a bottlenecked resource R_{save} and any cost on another resource R_{pay} to the target metric, using the baseline profile (C), experimental evidence (E) and declared assumptions (A). Schematically,

$$\text{Predicted gain} \;=\; \text{Saving}(R_{\text{save}} \mid C, E, A) \;-\; \text{Cost}(R_{\text{pay}} \mid C, E, A) \;>\; 0, \tag{9}$$

where savings and costs are expressed in common target-metric units through a performance model, rather than subtracting raw quantities of different resources. Two useful regimes appear in the study. Eliminating pure waste ($\text{Cost} = 0$ in the model) removes, for example, a redundant copy or accidental host–device synchronization. Spending a slack resource to relieve a bottleneck trades host memory for HBM or idle compute for a link at capacity, with positive net gain under the stated regime. Both motivate making the accounting of Eq. 9 explicit in the source-to-gain program `analyze(C, E, A)` of §3.

Method: the schema and the admission rule

We took 50 recent LLM serving papers from OSDI, SOSP, NSDI, EuroSys, ASPLOS, FAST and MLSys and restated each one's headline mechanism in a fixed schema: the gap located in the baseline profile, the slack resource spent and the bottlenecked resource relieved, the regime under which the trade pays, and the claim that follows. A row is admitted only if gap, trade and claim are each stated in the paper's own

numbers and the trade names no implementation. Each row was drafted by one agent from the PDF and checked by a second.

What the study shows. Every one of the 50 papers admits a restatement in the schema (Table 9); since a row is admitted only when the paper supports it in its own numbers, this shows that the schema fits these papers, not that every mechanism takes this form. Two observations bear on agent design. First, the trade names no implementation: the resource relieved, the resource spent and the regime can be stated without reference to how the system is built, the separation along which §3 isolates the analysis from the implementation. Second, the traded resource falls into four groups, memory capacity (15), execution time on a compute unit (25), link or I/O bandwidth (7) and provisioned capacity (3), the grouping of the table. Their costs and benefits can be related to profiles, hardware specifications and experiments, the inputs structured gap analysis asks the agent to connect in a source-to-gain program. Five mechanisms also give up output quality or drop requests (IC-Cache, CacheBlend, CacheSlide, PARD and Murakkab), a trade the benchmark's computation-preserving rule excludes.

How the sources map to claims. The three sources are the possible origins of an input, not a checklist every analysis must fill. The gap is quantified from the baseline resource profile (C) in 44 of the 50 rows; in the other six (NEO, vAttention, CacheSlide, HCache, TriInfer and LMETRIC) it is a hardware ratio, which the table marks as a declared assumption (A), or a microbenchmark the authors ran, which an analyzer supplies as experimental evidence (E). A declared assumption (A) enters 37 rows; the other 13 argue from measurement alone. The claim is a measured number in every row, since papers report what they built. The study reconstructs the link between those reported quantities; Fig. 13 illustrates how to express such an accounting argument as `analyze(C, E, A)`. The measured claims in the survey are distinct from the agent's ex-ante predictions.

```
# vLLM 4696: the agent's analysis
mechanism: memoised prefix hashes
gap: repeated whole-prefix hashing
baseline: 5.528 requests/s
evidence: 54.5 ms saved/request
assume: critical-path fraction 1.0
assume: CPU speed ratio 1.0
saved fraction: 0.0545 * 5.528
E2E prediction: -30.1%
final measurement: -39.2%
```

```
# analysis.py: abridged handover
def analyze(C, E, A):
  wall   = C.read("run seconds")
  served = C.read("served requests")
  saved  = E.read("saved seconds/request")
  frac   = A("critical_path_fraction")
  ratio  = A("cpu_speed_ratio")
  removed = saved * served / wall
  removed = removed * frac / ratio
  return {"E2E_mean": -100 * removed, ...}
```

Figure 13: A structured gap analysis from an agent delivery. **Left:** the prefix-hashing analysis on vLLM 4696. **Right:** its source-to-gain program, abridged from §E.2. The final measurement is shown for comparison and is not an input to the prediction. Colour marks a term's origin: blue the sealed corpus C, green the analyzer's own evidence E, orange a declared assumption A from `assume.yaml`. `C.read` and `E.read` abbreviate the corpus and evidence queries. The factor 100 converts a fraction to percent.

Table 9: Each paper's major mechanism restated as a resource trade: the **gap** located in a profile, the **trade** of a conserved resource from a donor to a recipient under a regime, and the **claim** that follows. Numbers are coloured by provenance: blue is readable from a baseline profile (C), green the authors had to measure (E), orange is assumed (A). Grouped by the resource traded.

I. MEMORY CAPACITY 15 papers	
NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference (MLSys'25)	
ASYMMETRIC PIPELINING *offload some requests' cached attention state, and its decode-time computation, to the host*	**Gap:** HBM caps the decode batch, compute idles; GPU vs host is 600 vs 200 GB/s but 125 vs 1.2 TFLOPS, and decode attention is bandwidth-bound. **Trade:** some requests' KV and decode attention → host CPU, memory-bound. **Claim:** 7.5x on T4, 26% A10G, 14% H100.

Table 9, continued.

Marconi: Prefix Caching for the Era of Hybrid LLMs (MLSys'25)

JUDICIOUS ADMISSION *admit at most two recurrent-state snapshots per sequence, at a predicted branch point and at the tail*	**Gap:** block-32 checkpoints reuse 25.0% of KV blocks but 0.4% of SSM states, 65.3x apart; 10K tokens hold 17.4 GB. **Trade:** mid-sequence snapshots, at most 2 each → shared prefixes and chat tails, contended cache. **Claim:** hit rate 4.5-34.4x, P95 TTFT -71.1%.

Pensieve: Stateful Large Language Model Serving (EuroSys'25)

MULTI-TIER GPU–CPU CACHE *retain a conversation's past context across turns, in device and host memory*	**Gap:** each turn re-prefills the history; at 32-token prompts and 200 steps prefill overtakes generation past 3072 history tokens. **Trade:** live requests' concurrency → idle chats' resident context, 60 s think time. **Claim:** 1.14x-3.0x baseline throughput.

KunServe: Parameter-centric Memory Management for Efficient Memory Overloading Handling in LLM Serving (EuroSys'26)

PARAMETER DROP *drop part of the replicated weights, the now-partial holders serving cooperatively*	**Gap:** a 2x burst overflows HBM, TTFT rises 239x at 56.3% mean memory load; weights hold 34-74% of HBM. **Trade:** whole weight copies, a slice apiece until the burst ends → KV for queued requests, pipelined. **Claim:** P99 TTFT 12.7-72.2x lower, TPOT +15.8-22.7%.

FineMoE: Taming Latency–Memory Trade-Off in MoE-Based LLM Serving via Fine-Grained Expert Offloading (EuroSys'26)

FINE-GRAINED EXPERT OFFLOADING *predict per iteration which experts the next few layers activate, keep only those resident*	**Gap:** every expert stays resident though 72%-84% of parameters are inactive per step, 23-70 GB of idle HBM. **Trade:** lowest-ranked resident experts → host, the predicted few prefetched 3 layers ahead over 32 GB/s PCIe. **Claim:** hit rate +39%, latency -47%.

Jenga: Effective Memory Management for Serving LLM with Heterogeneity (SOSP'25)

LAYER-SPECIFIC ALLOCATORS *pool memory per layer, free it once tokens stop being read*	**Gap:** paging all tokens in all layers wastes 79.6% of KV memory (vision model), 38.2% (Ministral). **Trade:** tokens no layer still reads → more concurrent decoders, capacity-bound. **Claim:** decode batch 2.63 → 5.39, throughput up to 2.16x (1.46x avg).

eLLM: High Throughput and Low Latency LLM Serving via Adaptive KV Caching (EuroSys'26)

ADAPTIVE TOKEN-WISE CACHING *evict an early prefix of each running sequence, recompute it per layer during decode*	**Gap:** at 50 req/s memory is 50%-100% used, compute 0%-10%. **Trade:** the earliest 36%-47% of each running sequence's KV, recomputed every step → more concurrent decoders (avg 223.16 → 254.64, max 448), memory-bound. **Claim:** throughput 3.03x, TTFT 2.63x lower.

CRAFT: Fine-Grained Cost-Aware Expert Replication For Efficient Mixture-of-Experts Serving (MLSys'26)

BENEFIT-DRIVEN REPLICA ALLOCATION *allocate a fixed budget of duplicate expert copies per layer in proportion to estimated balance gain*	**Gap:** EPLB puts one replica per layer per GPU though gains plateau past 16 and even layers gain little, cutting the KV cache 19%-75%. **Trade:** replicas bound for even layers → KV cache and batch, 7.25x-7.5x fewer replicas. **Claim:** goodput 1.14x avg, up to 1.2x.

CacheWise: Understanding Workloads and Optimizing KVCache Management for Efficiently Serving LLM Agents (MLSys'26)

PREDICTIVE KVCACHE EVICTION *evict cached state from the sessions predicted to resume last*	**Gap:** agent sessions run a median 36 min, tool turns outnumber user turns 20x; at 10 sessions or fewer all policies tie. **Trade:** KV of sessions resuming last → those resuming next, under contention. **Claim:** evictions 2-2.6x fewer, session time 2.7-3.5x lower.

BatchGen: An Architecture for Scalable and Efficient Batch Inference (OSDI'26)

YIELD *park paused sequence state in host memory to enlarge the running batch*	**Gap:** saturating every expert needs a 16384-token batch; baselines hold 8-16 sequences, GPUs about 50% underused. **Trade:** paused sequences' KV → host, 1800+ sequences batch, completion time the goal. **Claim:** 1.31x-1.85x there, 2.3x at 128 GPUs, 9.6x on 24 GB.

ECHO: Efficient KV Cache Offloading with Lossless Prefetching for Serving Native Sparse Attention (OSDI'26)

DYNAMIC TOKEN-LEVEL OFFLOADING *offload cached context to host memory, with prefetch that preserves the result*	**Gap:** HBM holds 380K KV tokens per worker, at most 4 concurrent 100K requests; sparse kernel at 53% utilization. **Trade:** resident KV of each sequence → a 1.8M-token host pool, more requests per decode batch. **Claim:** effective batch 28 → 102, throughput 2.15x.

Table 9, continued.

Prism: Cost-Efficient Multi-LLM Serving via GPU Memory Ballooning (OSDI'26)

MEMORY BALLOONING *reclaim reserved-but-unused device memory from idle colocated models on demand*	**Gap:** models idle 70% of the time; static partitions leave 50% of GPU memory unused. **Trade:** reserved memory of idle models → the bursting model on the same GPU, 2 MB pages. **Claim:** 99% TTFT attainment on 16 GPUs vs 32 baseline: 2x cost, 3.5x more requests.

Symphony: Enabling Compute-Memory Disaggregation in LLM Serving Systems (NSDI'26)

ADVISORY REQUESTS *evict session state at turn end, prefetch it back during the user's idle gap*	**Gap:** recompute redoes 99% of ShareGPT tokens; sticky offload loads a node 3.1x the median. **Trade:** idle-session KV → host and disk at turn end, prefetched when a turn is signalled 11.3s ahead. **Claim:** latency 2.4x lower; 4x the users at 1.3% more latency.

Kairox: Adaptive GPU–CPU Hybrid LLM Inference via Online Neuron Balancing (OSDI'26)

ONLINE NEURON BALANCING *reassign device memory at runtime to the weights the workload is currently activating*	**Gap:** static placement leaves drifted activations on the CPU: 102.8 ms per step single-stream, 551.7 ms at batch 20. **Trade:** GPU slots of neurons no longer firing → host neurons now firing, a 2,000-neuron cache. **Claim:** up to 7.57x over llama.cpp, geomean 3.15x.

Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot (FAST'25)

MOONCAKE STORE *pool idle host memory cluster-wide to hold and reuse shared prefixes*	**Gap:** local DRAM caches 3M tokens, under 50% of the possible hit rate. **Trade:** idle DRAM and NICs cluster-wide (50M tokens) → prefill GPUs that stop recomputing prefixes, above 100 Gbps a node. **Claim:** hit rate 2.36x, prefill 48% less, capacity 59% to 498% more.

II. EXECUTION TIME ON A COMPUTE UNIT 25 papers

SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling (MLSys'25)

STATE-AWARE SCHEDULING *re-decide admission and ordering every iteration from each request's measured latency slack*	**Gap:** 35% of requests miss the 200ms TPOT SLO with TTFT far under 500ms; attainment 65%. **Trade:** iteration slots of requests inside both SLOs, sorted back or held → requests about to breach, co-located phases. **Claim:** attainment 98%, up to 1.27x more requests.

AdaGen: Workload-Adaptive Cluster Scheduler for Latency-Optimal LLM Inference Serving (EuroSys'26)

SELECTIVE DISTRIBUTED EXECUTION *move one phase of chosen requests onto another instance's devices*	**Gap:** brute-force search cuts tail TTFT 1.5x to 1.9x; Llumnix meets it for 45%. **Trade:** iterations of resident destination requests → requests waiting behind the moved phase, memory or not. **Claim:** one moved phase gives 1.4x; up to 3.6x attainment, 2x cost.

XY-Serve: End-to-End Versatile Production Serving for Dynamic LLM Workloads (ASPLOS'26)

TOKEN-WISE SCHEDULING *one fixed token budget per step, mixing all three kinds of work*	**Gap:** irregular tiles drop attention MFU 53% to 30%; per-stage batching wastes 50% on rearranging. **Trade:** fixed 4k-token steps: prompts cut into chunks → decode and verify tokens in reserved slots. **Claim:** attention 21.5% faster: 79% more QPS; mixed steps: 89%.

XGrammar: Flexible and Efficient Structured Generation Engine For Large Language Models (MLSys'25)

ADAPTIVE TOKEN MASK CACHE *precompute, per reachable constraint state, which vocabulary entries are valid*	**Gap:** each step re-parses the 128k vocabulary: grammar TPOT 44.2 ms vs 6.2 ms without. **Trade:** setup under prefill and 0.46 MB host memory → the CPU time of each step; 1134 tokens (under 1%) need the stack. **Claim:** mask 36 us a token, TPOT 6.3 ms, up to 80x.

CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion (EuroSys'25)

SELECTIVE KV RECOMPUTE *partially prefill only the cross-attentive tokens on top of fetched cached state*	**Gap:** 4K-token RAG prefill takes 3 s; prefix caching saves only the first chunk. **Trade:** prefill paid by earlier requests, on SSD → the arriving request and its queue; 15% of tokens redone. **Claim:** 3 ms a layer under a 16 ms KV load; TTFT 2.2x to 3.3x lower.

IC-Cache: Efficient Large Language Model Serving via In-context Caching (SOSP'25)

LIVE CAPABILITY AUGMENTATION *serve easy requests on a small model primed with cached large-model answers*	**Gap:** 70% of requests have a near-duplicate, but replaying its answer drops win rate 50% → 18%. **Trade:** easy arrivals cede the large model → hard ones and the operator, 70% primed. **Claim:** 2B plus examples holds 50%, 71% faster, 5.1x throughput → 1.4-5.9x.

Bidaw: Enhancing Key-Value Caching for Interactive LLM Serving via Bidirectional Computation–Storage Awareness (FAST'26)

I/O-AWARE REQUEST SCHEDULING *reorder requests by where their cached history resides and how large it is*	**Gap:** tiered KV loads put latency 3.8x above the in-memory ideal: host hits 20%, load-time CoV over 90%. **Trade:** history on SSD, largest last → history already in host memory. **Claim:** queueing 5.76 s → 2.45 s; eviction 1.25x → 3.58x latency, 1.83x throughput.

Table 9, continued.

vAttention: Dynamic Memory Management for Serving LLMs without PagedAttention (ASPLOS'25)

VIRTUAL TENSORS
reserve contiguous virtual addresses, commit physical pages on demand

Gap: paged prefill kernels run 37-42% slower than contiguous ones. **Trade:** physical memory, held in 64KB pages, reclaimed late → prefill attention, at P:D 356. **Claim:** contiguous kernels close the gap → 1.18x, 1.23x long-context throughput; decode ties.

TokenFlow: Responsive LLM Text Streaming Serving under Request Burst via Preemptive Scheduling (EuroSys'26)

BUFFER-AWARE SCHEDULING
preempt requests whose readers already hold a surplus of undelivered tokens

Gap: in a burst TTFT passes 20 s (limit 1.3 s) while served requests get 30 tokens a second, readers take 12. **Trade:** early requests with unread buffered tokens → later ones with nothing delivered. **Claim:** 82.5% more effective throughput, P99 TTFT 80.2% lower.

PARD: Enhancing Goodput for Inference Pipeline via ProActive Request Dropping (EuroSys'26)

PROACTIVE REQUEST DROPPING
kill a request before it is batched when its estimated end-to-end latency already exceeds its objective

Gap: under bursts reactive drops leave goodput at 30% of arrivals; 57.1-97.2% come late, GPU time spent. **Trade:** requests past deadline, or judged so at a 0.1 wait quantile → those with budget. **Claim:** drops 0.12-3.6%, waste 1.5-62x less → 16-176% goodput.

CacheSlide: Unlocking Cross Position-Aware KV Cache Reuse for Accelerating LLM Serving (FAST'26)

WEIGHTED CORRECTION ATTENTION
reuse cached attention state across position shifts, recomputing few tokens

Gap: a position shift cuts cached-key similarity over 90% under RoPE, only 28% under CoPE. **Trade:** retained state of fixed segments → queued agent turns, same order, shifted positions. **Claim:** recompute 26% of tokens → 3.11-4.3x latency, 3.5-5.8x throughput.

Eevee: Efficient Multimodal Serving via Module Multiplexing (EuroSys'26)

MODULE MULTIPLEXING
run modality-specific modules concurrently on one device, each with its own compute share and batch size

Gap: BLIP on Triton keeps active SMs below 10% over 50% of the time. **Trade:** light text modules holding all SMs idle → image encoding of other requests, alongside at 70% of SMs. **Claim:** strawman 22.2 → 34.4 req/s; near 90% active SMs → 157% more capacity.

HCache: Fast State Restoration in LLM Serving (EuroSys'25)

ACTIVATION RESTORATION*
keep half-size per-layer intermediates on the host, reproject them on the accelerator

Gap: state restores cost 20.0-26.0x the ideal TTFT by recompute, 6.5-13.0x by KV offload. **Trade:** GPU time of resident decoders → arrivals with long evicted history. **Claim:** half the bytes, 6x less compute → 1.93x vs offload, 5.73x vs recompute; TBT up 4%.

TriInfer: Hybrid Disaggregated Scheduling for Multimodal Large Language Model Serving (MLSys'26)

HYBRID EPD DISAGGREGATION
split a fixed pool of serving instances among the three stages in proportion to profiled demand

Gap: encode saturates at batch 6, decode at 512; even split 2.6 req/s, profiled split 3.9. **Trade:** instances from the slack stage pool → the bottleneck pool, load steady. **Claim:** split plus stage batching lift 2.2 → 3.9 req/s → up to 2.4x goodput at 90% SLO.

LAPS: A Length-Aware-Prefill LLM Serving System (MLSys'26)

LONG–SHORT PREFILL DISAGGREGATION
hold long and short prefill requests in separate queues within the prefill stage

Gap: 81% of later-turn prompts are under 256 tokens; batched with a long prefill, their TTFT inflates 29-31x. **Trade:** long prefills, chunked → short prompts and re-prefills, 82.9% of search load. **Claim:** TTFT 30% lower, RPS 33% up, SLO misses 28% fewer.

Agentix: An Efficient Serving Engine for LLM Agents as General Programs (NSDI'26)

PROGRAM-LEVEL ATTAINED SERVICE
prioritize each call by the service time its whole program has already accumulated

Gap: at load 0.9 programs spend 80.4% to 91.3% of their time waiting. **Trade:** GPU service time from programs with most attained service → short and new programs, once a queue forms. **Claim:** toy wait 18 to 12 units; 4-15x throughput at equal latency vs vLLM.

Libra: Flexible Request Partitioning and Scheduling for Serving Unbalanced and Dynamic LLM Workloads (NSDI'26)

FLEXIBLE PARTITION AND SCHEDULING
split each request at an arbitrary token boundary to equalize execution time

Gap: 8192-in/32-out: decode GPU at 0.19% MFU; colocation 352.93 ms tail TBT vs 100 ms SLO. **Trade:** GPU time of the instance idle after its prompt share → requests behind its partner, as load shifts. **Claim:** goodput up to 1.91x colocation, 1.61x disaggregation.

Table 9, continued.

JITServe: SLO-aware LLM Serving with Imprecise Request Information (NSDI'26)

MINIMUM SERVING BANDWIDTH *give each request only the generation rate its own objective needs, so it finishes just in time*	**Gap:** Sarathi misses 78.6% of mixed SLOs, Autellix 91.4%, 50.5% with precise lengths. **Trade:** token capacity from slack requests held to just hit their objective → tighter or no-SLO requests, if contended. **Claim:** goodput 1.4-6.3x or 28.5-83.2% fewer resources.

Revisiting PP: Revisiting Pipeline Parallelism for LLM Serving (OSDI'26)

DYNAMIC CHUNKED PREFILL *resize the prompt-token budget admitted per pipeline iteration each control interval*	**Gap:** PP bubbles break the 200 ms TPOT SLO; TP sustains 1.2x the rate. **Trade:** iteration GPU time from waiting prompts, chunked smaller → active decodes and idle stages, on PCIe, unsaturated. **Claim:** chunk 512: 8.28 vs TP 3.48 req/s; resizing cuts TPOT 35% more.

CPU–GPU Hybrid: Achieving Cloud-Grade SLOs for Local Mixture-of-Experts Inference through CPU–GPU Hybrid Design (OSDI'26)

STREAM-LOADING PREFILL *stream expert weights across the host link so the accelerator runs all prefill*	**Gap:** host prefill peaks at 500 tok/s: TTFT 20 s at 8K, some 100 s at 32K vs a 30 s target. **Trade:** prefill arithmetic from host cores → GPU compute fed expert weights over PCIe, past 4K tokens. **Claim:** prefill rises to 1,200 tok/s (2.4x), so 32K lands in 27 s.

EcoServe: Efficient LLM Serving on Commodity GPU Clusters with Data-Reduced Cross-Instance Orchestration (OSDI'26)

PARTIALLY DISAGGREGATED STRATEGY *give prefill and decode alternating time slots on one device, staggering sibling instances' slots*	**Gap:** colocated prefills stall decodes; disaggregation needs 39 GB/s of KV per node on 10 Gbps links. **Trade:** GPU time from decodes with slack under a 100 ms TPOT SLO → new prompts awaiting first tokens on tight TTFT. **Claim:** goodput 1.96x vLLM, 1.99x Sarathi.

NanoFlow: Towards Optimal Large Language Model Serving Throughput (OSDI'25)

NANO-BATCHING *split each batch into sub-batches and duplicate every operation, so differently-bound work overlaps*	**Gap:** ops hit 80% of their bottleneck, yet in sequence compute utilisation is 40%. **Trade:** units from compute-bound GEMMs → memory-bound KV reads and collectives on a slice of units, at large batch. **Claim:** overlap lifts utilisation 40% to 68.5%, 1.91x TRT-LLM.

Strata: Hierarchical Context Caching for Long Context Language Model Serving (OSDI'26)

GPU-ASSISTED I/O *the accelerator's own threads perform the cache copy in both directions, rather than issuing a transfer engine*	**Gap:** PCIe utilisation 22%; prefill is 0.74 copies, 0.26 compute. **Trade:** under 5% of colocated prefill throughput → requests stalled on host KV loads, on small pages. **Claim:** 75% link use runs copies 3.4x faster, stall 24%; I/O alone 2.3x peak throughput.

LMETRIC: Simple is Better: Multiplication May Be All You Need for LLM Request Scheduling (OSDI'26)

MULTIPLICATIVE COMBINATION *route to the instance with the smallest product of uncached prefill work and current batch occupancy*	**Gap:** KV$ awareness cuts TTFT 84%, but weight 0.9 vs 0.7 skews prefill 3.57 vs 2.17 s. **Trade:** capacity from emptier instances without the context → those holding it, at bigger batch, if held widely. **Claim:** P-token x batch size: TTFT down 92%, TPOT 24% vs vLLM.

FlexInfer: Flexible LLM Inference with CPU Computations (MLSys'25)

SPLITGEN *run decode-phase layers on the host processor instead of streaming their weights to the accelerator*	**Gap:** exposed PCIe copies are 91.6-97.6% of FlexGen runtime. **Trade:** idle host CPU cycles and memory bandwidth → memory-bound decode steps stalled on the host-to-device link. **Claim:** CPU decode moves 0.3 MB per token: TPOT down 77.7%, latency 75% and 76%.

III. LINK AND I/O BANDWIDTH 7 papers

DirectKV: No Buffer, No Bottleneck: Efficient Zero-Copy KV Cache Offloading for Long-Context LLMs (OSDI'26)

CPU-MEMORY-AWARE TILING *hold the host-resident operand stationary and re-read the partial results locally instead*	**Gap:** zero-copy GEMM on NVLink-C2C takes 106 ms vs 52 ms local, re-fetching the host operand: 33.5 GB. **Trade:** HBM traffic 21.1 GB → 61.8 GB + 33.9 GB, link 0.4 GB; HBM 4 TB/s vs 450 GB/s, prefill only. **Claim:** 54 ms; end to end 1.2x, 43% less GPU memory.

Seesaw: High-throughput LLM Inference via Model Re-sharding (MLSys'25)

DYNAMIC MODEL RE-SHARDING *re-partition weights and cached state across devices at each phase transition*	**Gap:** over PCIe, TP8 prefill is 0.77 all-reduce, 5x PP8's time; PP8 decode is 0.66 weight loads, 2x TP's. **Trade:** prefill's per-layer exchange → decode's, re-sharding PP→TP at the switch; offline batch, slow link. **Claim:** 1.36x avg, 1.78x max over vLLM.

Table 9, continued.

FlexPipe: Adapting Dynamic LLM Serving Through Inflight Pipeline Refactoring in Fragmented Serverless Clusters (EuroSys'26)	
INFLIGHT PIPELINE REFACTORING *switch pipeline depth at runtime as arrival burstiness changes*	**Gap:** 75% of peak always-on, GPUs at 17%; stall overhead 10x from CV 1 to 4. **Trade:** in-flight wire time 45 → 225 ms → the backlog; bursty only: deep pipes 3x faster at CV 4, 2.7x slower when smooth. **Claim:** latency -66.1%, 8.5x goodput per GPU, 75% → 30%.
HydraServe: Minimizing Cold Start Latency for Serverless LLM Serving in Public Clouds (NSDI'26)	
CROSS-SERVER MODEL PARALLELISM* *spread a model's layers across workers on several servers so each pulls only its share of the weights*	**Gap:** cold start over 40 s, 24.5 s fetching weights over 16 Gbps, vs 30 ms per token. **Trade:** idle servers' links, GPU memory-time → the cold instance; 4 workers each fetch a quarter, TPOT 1.06x. **Claim:** cold start 1.7-4.7x shorter, SLO attainment 1.43-1.74x.
PPC: Accelerating Model Loading in LLM Inference by Programmable Page Cache (FAST'26)	
INTERRUPTIBLE PREFETCHING *replay a read sequence recorded from a previous identical launch, reading ahead from the first access*	**Gap:** loading reads the SSD at 1.05 GB/s of a 5.93 GB/s peak, 17%, a 5.64x gap, and over 50% of startup. **Trade:** the IO-idle init phase → the critical-path load; local SSD, repeat launch. **Claim:** the gap caps the win at 5.64x; loading -79%, 4.8x, startup -38%.
Layered Prefill: From Tokens to Layers: Redefining Stall-Free Scheduling for MoE Serving with Layered Prefill (MLSys'26)	
LAYERED PREFILL *run prefill for exactly one contiguous layer group per iteration, co-scheduled with decode*	**Gap:** 512-token chunks make 18 expert passes per 9k prompt: 955 GB per request, MoE over 50% of a 500 ms step. **Trade:** prefill, one layer group per step → co-located decode, 1.3 → 1.6 req/s. **Claim:** loads -39%, TTFT 2.80 → 1.24 s, e2e -41%, energy -22%.
ADAngel: Accelerating Arbitrary-Precision Quantized LLMs with Adaptive Computing Mapping (OSDI'26)	
STRATEGY DISPATCHER *select per operation, from an offline latency profile keyed on shape, how both operands are partitioned into hardware-native widths*	**Gap:** padding reads 4-bit weights as 8-bit, 2x the bytes, and at batch 1 idles 93.75% of a tile. **Trade:** partial-sum shared memory, 32x, and issue slots → weight fetch at 4 bits, below prompt 8, batch 16. **Claim:** decode 1.95x vs TRT-LLM, 5.10x max vs llama.cpp.
IV. PROVISIONED CAPACITY 3 papers	
PiLLM: Resource-efficient LLM Inference Using Workload Prediction (EuroSys'26)	
ELASTIC INTER-GPU DISPATCH *scale the active device count per phase from predicted compute demand*	**Gap:** arrivals diurnal, compute demand spiky; tail lengths 3-10x the bulk, GPUs at peak, KV memory 45-70% used. **Trade:** peak-sized allocation → the provider; 1-4 instances a phase cap it at 4x. **Claim:** 1.62-3.06x fewer GPUs, decode 3.72-4.00x, SLO 97.9-100%.
Janus: Multi-LLM Serving at Production Scale (SOSP'26)	
STEADY POOL *colocate low-traffic models densely so devices are free for burst replicas*	**Gap:** 12% of apps send 80% of requests; 5-10x bursts in seconds vs a 0.4-2.7 s H2D reload. **Trade:** tail apps packed on shared devices → head apps' exclusive D2D-fork replicas in 388-783 ms. **Claim:** SLO 0.97-1.0 vs 0.80-0.92; 13,440 device-hours, 27% vs static.
Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms (OSDI'26)	
CROSS-WORKFLOW MULTIPLEXING* *choose knobs, model and hardware per objective tier, multiplex requests across workflows onto shared instances*	**Gap:** hand-tuned serving holds 2568 GPUs flat, 82.1 MWh, 211.7k USD; auto-scaling saves 1.8x. **Trade:** loose tiers drop 66.2% → 61.4% accuracy for 18.5k → 6.9k USD; cross-workflow pooling cuts 21.6% more. **Claim:** 912 GPUs, 2.8x; 22.1 MWh, 3.7x; 47.2k USD, 4.3x.

C EXPERIMENT DETAILS

The setup of §3 and §4 as run: the constants of every run (Table 10), what every agent is given and the eight rules, the three configurations of Table 1, prompt excerpts, the handover between the stages, how the pool was sampled and how selection and the two checks were applied to it, and the ablation setups. Every attempt has the same 1.5-hour wall clock and pinned model. The workspace contains the setup and profiles; the reference patch and its measured gains are withheld.

C.1 SETUP AT A GLANCE

An attempt has 1.5 hours of wall clock in total, whichever configuration it belongs to: the single agent spends it in one session, and a gap-analysis attempt divides it equally between analyzer and implementer, so the second stage is paid for out of the same envelope. The wall clock is the only bound enforced from outside the CLI on an agent session, with no turn cap and no spend cap; a session that exceeds it ends the attempt, which counts as no change, a transport failure is retried once, and every session runs the same pinned model, `claude-opus-5`, with no fallback. Table 10 collects these and the other constants shared by every run of §4.

Table 10: **Setup at a glance.** Constants shared by every run of §4; the model-size ablation changes only the model row (§C.7).

item	value
node	GCP `a3-mega`, 8×H100 80GB; the agent measures on the node's GPUs through `./measure.sh`; scoring on a separate single-tenant node of the same type
wall clock per attempt	1.5 hours; one session for the single agent, 45 minutes each for analyzer and implementer
turn cap, spend cap	none for agent sessions; the assumption audit runs under a 25-minute wall clock and a $5 cap
session exceeding its wall clock	process group terminated, any engine it launched included; the attempt ends and counts as no change
transport failure	one retry, recorded
model	`claude-opus-5`, pinned, no fallback; ablation: `claude-sonnet-5` in one or both roles
sessions per item and configuration	ten analyzer sessions, each handed to one implementer on a fresh tree; ten single-agent sessions for VANILLA; under iterative gap analysis, up to three further analyzer sessions per stopped slot, each handed to an implementer likewise
K	parallel gap analysis at K=10; $K = 1, \ldots, 10$ read over the item's pool of analyses
selection metric	predicted E2E gain; ties on predicted TTFT, then ITL, then the mean over the tied set; a missing prediction sorts as 0

C.2 WHAT EVERY AGENT IS GIVEN

The workspace. Every session starts from the staged workspace of §2, with `./measure.sh` and an empty `deliverable/` beside the engine tree. The structured analyzer additionally gets `corpus/`, the same configuration and profiles behind the SQL handle C of §3; the implementer additionally gets `analysis/`, the analyzer's handover, on a reset tree. The single agent and the implementer write the engine tree and `deliverable/DELIVERY.yaml`; the analyzer writes `output/analysis/`. The reference fix and everything that reveals it are never copied in (§A.4.1).

The eight rules. A delivery that still breaks a rule after the checker's three chances (§A.4.2) receives no performance credit: its attempt remains in the evaluation at 1× speedup, or zero normalized gain.

1. `configs/` is the question: it may not be edited and is rebuilt from the sealed item before measurement.
2. The delivery is the working-tree diff against the baseline commit; `ready` over an empty diff is refused, and "nothing here should change" is `abandoned` with evidence.
3. Patches are confined to Python and Triton; a diff touching native source or build files is refused, since the patched engine runs on the baseline's binaries.
4. The environment is untouched; the package managers are refused by name.

5. The engine must still compute the same thing, checked by the probe of §2; a change known to alter output must also be declared (`semantics_changed`), and declaring it does not waive the probe.
6. No network and nothing read outside the workspace; every tool call and shell command is audited afterwards from the transcript.
7. The tree is left buildable: no branches or resets, no deleting files the agent did not create.
8. The kind of change is declared (`change_kind`) and checked; no kind disqualifies a delivery, but one may not be presented as another.

Measuring. `./measure.sh` is a local diagnostic helper inside the agent's sandbox, distinct from the actual evaluator (§2, §A.4). It runs against the agent's working tree on the sandbox node's GPUs (queueing when busy) and supports short diagnostic slices (`--window 25%` or `--window 120s`) alongside any profiling or microbenchmarks the agent writes; analyzers may only run such short probes or microbenchmarks and are not allowed to run the full benchmark. Unlike `./measure.sh`, the actual evaluator runs only once after the session ends on a separate, dedicated single-tenant node: it rebuilds `configs/` from the sealed item, enforces the anti-cheat and 100-prompt output-equivalence checks, and runs the full-length certification pass (requiring served requests within 5% of the baseline) to compute the official normalized gain.

C.3 THE THREE CONFIGURATIONS

Table 11: **The three configurations** of Table 1. Only the listed artifacts cross the reset between the stages; everything else the analyzer left is quarantined.

configuration	sessions (roles)	crosses between stages	delivers
VANILLA	1 (single agent)	none	patched tree, `DELIVERY.yaml`
Gap analysis	2 (analyzer, implementer)	`analysis.md`	analysis document, patched tree, manifest
Structured gap analysis	2 (analyzer, implementer)	`analysis.md`, `analysis.py`, `assume.yaml`, `evidence/`	analysis document, program, assumptions, patched tree, manifest

VANILLA is one session given the workspace and the rules (Table 11); it delivers the patched tree and `DELIVERY.yaml` (schema in §C.4), and an abandonment with a reason is a valid answer that scores as no change. Under gap analysis, the analyzer of §3 delivers `analysis.md`, whose front matter carries a predicted percent change for every metric of a fixed block and a confidence, and the implementer's manifest adds `follows_analysis` (`full`, `partial` or `departed`). Under structured gap analysis the analyzer additionally submits `analysis.py`, the deterministic `analyze(C, E, A)` program returning the predicted percent change of every metric in that block, and `assume.yaml`, declaring every number the program uses that it did not read from the corpus, with a value, a description and a rationale each; the number the document states in `predicts:` must equal the number the program returns.

C.4 THE PROMPTS

Composition. Every prompt is one file, `laws.md`, with the item's facts substituted and the role's delivery block appended under "What to hand back": `direct.md` for the single agent, `analyst-report.md` or `analyst-program.md` for the analyzer, and `implementer.md`. The prompt files call the analyzer role the "analyst"; the quotations keep that spelling. The analysis document has four parts: where the time goes, by file and line; why it costs that much, with evidence; what to change, specifically enough to implement without redoing the analysis; and how to tell that it worked.

The frame (`laws.md`). It opens with the task and the item's facts, then the workspace, the eight rules, how to measure, and what to hand back:

Prompt excerpt `laws.md`

"You have a serving setup and the engine that runs it. **Make this setup faster.** [...] Nobody is telling you what is slow here or where to look. `configs/` is the whole of the question and there is no answer written anywhere you can reach. Several directions through a setup like this usually work; find one and take it as far as it goes. You are one agent, working alone, with no way off this machine."
"**Read it before you theorise.** The most common way to waste a session here is to reason from what the code looks like it should cost and never check what it actually cost. If the profile says the GPU is busy 25% of the window, no kernel change is going to matter."

The single agent (`direct.md`). The block is the manifest schema and the instruction to "write this file EARLY and keep it current [...] a session that leaves no manifest is recorded as having delivered nothing, however good the tree is."

`DELIVERY.yaml`, the schema as quoted in `direct.md`

```
status: ready        # `ready` or `abandoned`; nothing else counts
reason: >            # REQUIRED when `abandoned`
summary: >           # what the engine did, what changed, why faster
mechanism: >         # the thing that costs less now; name it
change_kind: new-code    # new-code | reused-existing | rewired |
                         # default-changed | mixed
semantics_changed: false # true if the output changed (law 5)
risk: >              # where this could be wrong; what went unchecked
```

The analyzer, free-form analysis (`analyst-report.md`). Three parts: the handover, the analysis document, the prediction.

Prompt excerpt `analyst-report.md`

"**A report. You are not the one who implements this.** Somebody else, with no memory of this session and nothing but your report and the same workspace, will make the change. Everything you worked out has to survive in the document, because nothing else does: **the engine tree is reset to the start commit before they see it.** Edit it, instrument it, break it – experimenting is what it is for, and none of it carries. Only the report does."

The document carries the four parts above, and "if you looked and there is nothing here worth changing, say that, with the evidence. That is a result." The prediction lives in the front matter, in a block whose shape is a constant of the suite,

`analysis.md` front matter, the `predicts` block

```
predicts:
  E2E_mean:           <signed percent change>  # REQUIRED; selection reads it
  TTFT_mean:          <signed percent change>  # recorded
  ITL_mean:           <signed percent change>  # recorded
  OUTPUT_TOKENS_mean: <signed percent change>  # recorded
confidence: medium                             # low|medium|high
```

and is to be what the built implementation "will ACTUALLY measure [...] not the size of the effect you found." Parallel gap analysis selects on the predicted E2E gain; deliveries are scored on all five metrics of §4. Four rules govern the prediction:

Prompt excerpt `analyst-report.md`

"**The sign is arithmetic, not editorial: negative means the number went DOWN.** [...] A bare number is a PERCENT – `-0.5` is half a percent, not a half." "**`E2E_mean` is the one required entry and the one selection reads.** [...] The other three are recorded beside it; the delivery itself is scored on all five metrics. They are what make a right answer distinguishable from a lucky one." "**Omission is not free and `0` is not a cop-out.** A missing `E2E_mean` is read as 'no change' and sorts below every stated improvement, so staying quiet to avoid being wrong loses to guessing honestly." "**The three latency**

axes are not independent, so do not state them as three opinions. For a request emitting n tokens, $\text{E2E} = \text{TTFT} + \text{ITL} \times (n-1)$, and that closure holds on this corpus within 5% – over the MEANS."

The analyzer, source-to-gain program (`analyst-program.md`). The document section and the prediction rules are reproduced word for word; the prompt adds the program of §3, handed over as four names under `output/analysis/`, and `analysis.py` "has to stand on its own. A helper module beside it does not travel."

The `analyze()` contract

"A report can say 'this should be roughly 30% faster' and nobody can tell whether that came from arithmetic or from optimism. Here the number is **computed**, from inputs that each have a source, so a reader can see which quantities it rests on and how hard it moves when they are wrong." The program is `analyze(C, E, A) -> {metric: percent change}` over the metric names of the block above; it "must be **deterministic**. Nothing else is required and nothing is forbidden: import what you want, loop, use numpy." The prompt's own example is the whole contract:

`analysis.py`, the example given in `analyst-program.md`

```
def analyze(C, E, A):
    # two numbers you READ, with lineage, one you could not read
    cost   = C.scalar("SELECT AVG(dur) FROM prof.kernel "
                      "WHERE name LIKE ?", ("<pattern>",))
    budget = C.scalar("SELECT value FROM setup.config "
                      "WHERE key = ?", ("<a declared key>",))
    saved  = budget * A("share_removed")
    return {"<a latency in your set>": -100.0 * saved / cost,
            "<one you expect not to move>": 0.0}
```

C exposes the schemas `setup`, `prof`, `nsys`, `nsysrun` and `bench` (the configuration, the profiler passes and the guidellm output) and `evidence` ("whatever you submitted alongside. Never certified."). `C.scalar(sql, params) -> float` is "the only way a number reaches the prediction with lineage"; the program may also read the evidence schema through its own handle, the E of `analyze(C, E, A)`. "Any literal that is not 0 or 1 is an assumption, wherever you put it", a query included; every such number is declared in `assume.yaml` and read back as `A("name")`:

`assume.yaml`, one entry

```
- name: share_removed
  value: 0.70
  kind: assumed      # assumed | definition | external
  description: >     # what the number IS, in one sentence
  rationale: >       # why THIS value and not 1.0, or 0.3
```

Self-scoring is refused ("An analysis tuned against its own score is measuring the score."), the program is to derive E2E from TTFT, ITL and the corpus's n ("the arithmetic, not a fourth opinion"), and "`predicts:` must be what `analyze()` returns."

The implementer (`implementer.md`).

Prompt excerpt `implementer.md`

"**`analysis/` holds an analysis of this setup, written by somebody who is gone.** Read it first. It is the reason you are here: your job is to make what it describes real, not to start the investigation over. [...] Read it as a hypothesis with evidence behind it, not as an instruction. Where it points at a function, go and read that function. Where its reasoning does not survive contact with the code, **say so in `DELIVERY.yaml` and do the thing the code supports** – an implementation faithful to an argument that was wrong is worth nothing. Where it is silent on a detail, that detail is yours. What you must not do is throw it away and analyse the setup yourself from scratch. That is a different experiment, and this run cannot tell the two apart afterwards."

The manifest is the single agent's with `follows_analysis: full | partial | departed` and `departure` added. The implementer builds what the analysis proposes first ("That is your floor"), then may improve on it: adding to it is `full`, substituting its own idea is

partial, and "if you depart, you own the number." Independent bottleneck exploration is excluded by design. The implementer can resolve code details, test the proposed change and report a contradicted premise; it cannot run a second analysis search. This holds the handover fixed when implementation behavior is studied.

C.5 Handover and mechanics

One attempt at one item is one process: it stages and seals the workspace (§A.4.1) and runs the configuration's stages in order as headless CLI invocations. Between the analyzer and the implementer the engine tree is reset to the start commit, the four handover names (`analysis.md`, `analysis.py`, `assume.yaml`, `evidence/`) are copied out of `output/analysis/` into `analysis/`, and everything else the analyzer left is quarantined beside the record, so the implementer has no access to the analyzer's session. Every attempt runs its own analyzer, so an attempt is an independent draw of analysis and implementation. The delivery is the diff against the start commit, taken with the repository's diff configuration pinned off and after `git add -N`. A delivery is accepted if and only if a manifest exists with status `ready` and the diff is non-empty, applies to the baseline and touches no native source; the tree is then committed and measured by the scorer of §2. A session whose failure is the transport is retried once; the outcome states of an attempt and their rates are in Table 6. The tool allow and deny lists and the transcript audit are those of the interaction sandbox (§A.4.1); a denied call costs the agent a turn and nothing else.

C.6 Sampling, selection and checks as run

Sampling and post-hoc evaluation. For each item and gap-analysis configuration, we independently generate ten analyses, implement each once on a fresh reset tree, and evaluate the resulting deliveries post hoc; VANILLA supplies ten single-agent attempts. These additional implementations and measurements characterize the candidate pool and allow the same data to support the selection and oracle comparisons. No final scorer outcome is returned to an analyzer, implementer, check or selector. The analyzer's own profiling, prototypes and measurements remain part of its permitted investigation.

Every original attempt remains in the pool. A valid, comparable delivery receives its measured speedup; a failed, invalid, timed-out, unmeasured or abandoned attempt receives $1\times$ speedup, hence zero normalized gain. A failed selected candidate is not replaced by another candidate. The transport retry specified in Table 10 is a separate infrastructure rule. The solid curves in Fig. 14 average over every K-subset of the ten attempts; the faded curves use the first K in run order.

Selection rule

Parallel gap analysis (§3). Among the K analyses of an item, the one with the largest predicted end-to-end gain (the most negative predicted `E2E_mean` percent change) is selected and its implementation's measured outcome taken; ties are broken on predicted `TTFT_mean`, then `ITL_mean`, and a tie that survives is scored as the mean over the tied set, the expectation of breaking it at random. The prediction is the analysis's front-matter value; with the source-to-gain program it is the value `analyze()` computes, and the prompt requires the two to agree. A missing prediction sorts as 0.

Patch ranker (+ *ranker*). For VANILLA + *ranker* (K=10 in Table 1), we pass the item's setup card, workspace and codebase context, and all K candidate VANILLA code diffs (along with any output artifacts, such as `DELIVERY.yaml`) to `claude-opus-5` in a read-only call without GPU execution and evaluate the top-ranked patch:

Prompt excerpt `patch ranker (+ ranker)`

System: "You review performance patches to LLM serving engines. You cannot run anything; judge from the code. Answer with the JSON object asked for and nothing else."
User: "Engineers were each given the same serving setup, alone, and asked to make it faster. Each delivered a patch to the engine. Rank the patches by how much each will improve {`metric`} when measured under this setup. [...] The score is {`metric`} on every item, even where the card's `metric:` line names others. A patch that breaks serving, or only helps a metric that is not scored, gains nothing. `## Setup card` {`card`} {`patches`} Rank ALL {`n`} patches ({`labels`}) from the

largest expected {metric} improvement to the smallest. Reply with only this JSON: {"ranking": [{example}], "reason": "<two sentences at most>"}"

Why evaluating all candidates preserves the policy. Let $Z_{1:K}$ contain the analyses and all information available before final implementation, and let $I = s(Z_{1:K}, U)$ be the prediction-based selected index, where U breaks any remaining tie uniformly. Write Y_k for candidate k's final speedup, with $Y_k = 1$ on failure, and $p_k(Z) = \Pr_U(I = k \mid Z)$. The replay averages only this tie-breaking randomness:

$$G_{\text{replay}}(Z, Y) = \sum_{k=1}^{K} p_k(Z) Y_k = \mathbb{E}_U[Y_I \mid Z, Y]. \tag{10}$$

Measuring the unselected candidates cannot change the selection probabilities: their outcomes enter neither s nor candidate generation. With a unique winner the replay returns exactly that candidate's outcome; averaging tied winners preserves the expected per-item speedup used in the aggregate score. Thus replay and implementing only the selected analysis have the same expected speedup under the same implementation and measurement protocol. This is the one-final-evaluation policy reported in Table 1; the post-hoc experiment spends extra measurements to estimate it. Oracle selection explicitly uses measured outcomes and is reported separately (Fig. 6).

The tracer. The sensitivity check and the source-tracing figures (§D.3) rest on one procedure, the trace of §3. The submission, `analysis.py` and any sibling `.py`, is compiled with every undeclared numeric literal, unit conversions such as the $\times 100$ of a percentage excepted, AST-lifted into a substitutable slot. Every number that reaches `analyze()` is a *leaf*, read through `C.scalar` with its SQL, the (schema, table, column) it touched and the hash of the file underneath, or an *orphan*, a lifted literal. Each is perturbed in turn and the elasticity $\partial \ln v / \partial \ln x$ of the prediction taken by central difference; a literal inside a query string is perturbed in the query text and its elasticity netted off the leaf it lives in. The *leverage* is the sum of the absolute elasticities, and the four sensitivity shares split it by source: ϕ_C (certified schemas), ϕ_E (`evidence`), ϕ_A (`assume.yaml`) and the residual, every orphan plus the in-query weight; ϕ_C alone is derived from which schema a query actually read. A number that leaves the surface, a non-deterministic run, a 0% prediction or a query that reads nothing is reported as a flag rather than a share.

The sensitivity check. The sensitivity check (left box of Fig. 16) gates on total perturbation leverage rather than the source shares (ϕ_C, ϕ_E, ϕ_A), which are reported separately as post-hoc diagnostics in §D.3. A program that fails to execute, is non-deterministic, or whose trace does not finish is stopped. Otherwise, for each metric $m \in \{\texttt{E2E_mean}, \texttt{TTFT_mean}, \texttt{ITL_mean}\}$, let claim_m be the predicted percent change and $\text{lev}_m = \sum_i |\partial \ln v_m / \partial \ln x_i|$ be the total leverage across all perturbed inputs (both declared leaves from C, E, A and AST-lifted undeclared literals). The check stops the analysis if $\max_m \text{lev}_m = 0$, or if the maximum claim-to-leverage ratio across metrics with $\text{lev}_m > 0$, $\max_{m:\,\text{lev}_m > 0} |\ln(1 + \text{claim}_m/100)| / \text{lev}_m$, is below $\tau = 0.01$ (fitted on the development split beforehand and held fixed). Taking the maximum across m credits phase-specific mechanisms (e.g., prefill-only TTFT or decode-only ITL). Because this rule checks claim-to-leverage conditioning rather than thresholding ϕ_C or the undeclared share directly, an undeclared constant prediction (e.g., -20%) whose literal is AST-lifted has $\text{lev}_m \approx 1$ and ratio $|\ln 0.8| \approx 0.223 \geq \tau$, and a well-conditioned TTFT or ITL claim can pass even if E2E is unsupported; catching ungrounded or contradicted premises relies on the assumption audit, while stricter per-source sensitivity gates ($\phi_C > 0$, bounded orphan share, and E2E-specific gating) are natural extensions.

The assumption audit. The assumption audit (right box of Fig. 16) is one additional session per analysis, staged like an analyzer's but read-only, with the sealed baseline tree linked as `engine/`, no GPU, and a `probe.py` that re-runs `analysis.py` as shipped or with named assumptions overridden. The implementer's patch, manifest and every measurement stay out of its workspace, so a verdict cannot be a reading of the outcome. The question is the one in Fig. 16; the rules for answering it:

Prompt excerpt `assumption audit`

> "Verdict per assumption: `holds` (the evidence supports the value, within the slack the rationale itself allows), `doubtful` (you could not confirm or refute it), or `wrong` (the code or the profile contradicts it). Name the file and line, or the profile number, you checked it against. An unsupported guess is not `wrong`; a contradicted one is." "Decide `load_bearing`: does the E2E claim move materially when this value is corrected? `probe.py` answers that directly." "**`fail`** when a load-bearing assumption is `wrong` – with your corrected values the claimed E2E gain shrinks below half of itself (in log terms) or changes sign – or when the assumed mechanism is not on this setup's path at all. **`pass`** otherwise, including when the values are uncertain but nothing you found contradicts them."

The verdict is the last fenced JSON block of the final message. The session runs the same pinned model with a 25-minute wall clock and a $5 spend cap; one that returns no parseable verdict passes the analysis, because an unfinished audit is not evidence against it. Every structured analysis in the pool was audited, one session each.

Checks and iterative gap analysis as run. Checks are computed once per analysis and joined to the pool by attempt; they apply to structured analyses only. *Filtering* keeps, for an item and a K-subset, the best-predicted analysis among those the checks pass, and falls back to the best-predicted of all K when none passes; the conjunction stops an analysis if either check does. *Iterative gap analysis* returns the verdict of a stopped analysis to a fresh analyzer session, staged like the original, which produces a new analysis; the new analysis is checked in turn, and the loop runs up to three times per slot, after which the slot's last analysis stands, even if the checks still stop it. Each slot therefore uses at most four analyzer sessions. Re-attempts depend on the checks' feedback alone. For post-hoc evaluation each is also handed to an implementer on a reset tree and scored under the same failure rule; these outcomes do not influence the retry decision or the final selection.

Two statistics describe different parts of this procedure. The *passed-set mean* filters the original K analyses: it is the expected delivered gain of a uniform draw from those passing the checks, or from all K if none passes. It is the original-pool diagnostic shown in Figs. 15b, 17 and 18; its dependence on K comes from the fallback probability. The *iterative pick* instead selects by predicted gain among the K final slots after retries. It includes newly generated analyses, whose outcomes are looked up only after selection, as in Eq. 10.

C.7 Ablation setups

The model-size ablation of §D.4 runs the 35 items of the held-out post-cutoff split with `claude-sonnet-5` in one or both roles, under the prompts, rules, wall clock and measurement of the main comparison, whose attempts on those items are the Opus rows. Three stages, one model assignment each: every role on Sonnet in all three configurations; a Sonnet analyzer handing to an Opus implementer in the gap-analysis configurations; and an Opus analyzer handing to a Sonnet implementer (Vanilla has no second role to mix and appears only in the all-Sonnet stage). Each attempt is an independent analyzer–implementer pair; Table 15 reads the pools with the conventions above, and each arm has ten samples per item.

C.8 Evaluation regime and baseline scope

Four design choices define our evaluation regime and the scope of baselines. First, other coding-agent scaffolds: gap analysis is a policy laid over an off-the-shelf CLI agent, not a scaffold, so the informative baseline is the same CLI agent without it (Vanilla), and the framework's effect is shown to hold across two CLIs and two model families (§D.5); a different scaffold would change both arms together. Second, search and evolutionary methods such as AlphaEvolve (Novikov et al., 2025): they evaluate every candidate patch on the target benchmark, whereas every configuration here is held to one final deployed measurement per item (and at $K{=}1$ also shares Vanilla's 1.5 h wall-clock envelope) to study pre-deployment analysis and selection (§B.2). Third, scaling Vanilla via parallel or sequential runs under a single final evaluation: gap analysis scales effectively because it exposes quantitative pre-implementation signals that Vanilla lacks, namely the predicted gain for parallel selection and the sensitivity and assumption checks on `analyze(C, E, A)` for iterative refinement. Without a predicted gain, picking one of K unmeasured Vanilla patches at random stays at the single-run mean (Fig. 14), and having an Opus 5 judge read and rank the $K{=}10$

VANILLA code diffs and output artifacts against the workspace and codebase (+ *ranker* in Table 1) yields only $+1.3$ and $+5.5$ E2E points on the pre- and post-cutoff splits (58.3% and 65.6%), far below parallel gap analysis (64.7%–73.2% unstructured and 70.7%–77.3% structured) while still requiring K full coding sessions; likewise, because modern coding agents already self-prompt, test, and refine within a single session, blindly prompting VANILLA to "refine the code further" across K sequential sessions without a new external signal or deployed measurement may leave the analysis-stage spread untouched while serializing wall-clock latency to $O(K)$ versus $O(1)$ for parallel gap analysis. Fourth, existing performance benchmarks: kernel and micro-benchmark suites score a different task than a deployed serving system graded against a sealed expert fix, and the normalized gain is not comparable with their metrics; the related work positions them.

D ADDITIONAL RESULTS

The companion of §4: the development-split results (Table 12), the per-split versions of Fig. 5 (Figs. 14 and 18), the two checks and what they stop and pass, and pairwise significance; then five checks of the findings: that the benchmark rewards routes the expert did not take (§D.1), how accurately predicted gains track and rank delivered gains (§D.2), what a structured analysis rests on (§D.3), a model-size ablation of the two roles (§D.4), and the same progression on a second model family and CLI (§D.5).

Table 12 reports the results on the 30-item development split alongside the held-out results of Table 1. The development split was used to iterate on the role prompts, the `analyze(C, E, A)` handover contract, and the sensitivity check's scalar threshold (τ=0.01; §C.6), after which all prompts, rules, and thresholds were frozen before evaluating on the pre- and post-cutoff held-out splits. Because prompt instructions and check formatting were tuned on *dev* (where VANILLA also starts at a higher baseline of 75.8% E2E), absolute scores on *dev* are higher than on the untouched held-out splits; we therefore report only the held-out splits in the main text and include *dev* here for completeness. Figs. 14 and 18 break Fig. 5 down by split. Fig. 16 specifies the sensitivity check and the assumption audit of §3; Fig. 15 shows both checks over all 100 items, Fig. 18 what each check stops and passes per split, and Fig. 17 the pick with feedback per split; Fig. 19 gives the paired tests behind §4.

Table 12: **Results on the 30-item development split of LMSYSDOJO-100** (`claude-opus-5` on Claude Code). Prompts, handover rules, and τ=0.01 were developed on this split before evaluating on the held-out splits of Table 1. Entries are normalized gain in percent.

	dev (n=30)				
setup	E2E	TTFT	ITL	TPS	RPS
Vanilla	75.8	80.9	74.7	83.5	86.5
Gap analysis	78.1	85.6	79.3	87.1	86.3
+ parallel	83.3	87.3	84.3	88.6	88.6
Structured gap analysis	79.9	79.6	83.6	89.5	88.7
+ parallel	104.9	102.5	138.4	108.8	104.6
+ iterative	**113.5**	**109.6**	**151.8**	**119.2**	**114.2**

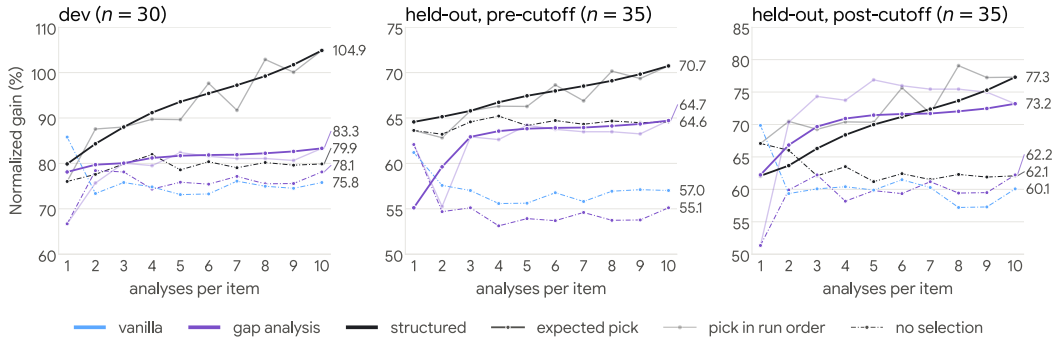


Figure 14: **The pick against the number of analyses**, per split, E2E. Solid: the expected pick, in expectation over which analyses are drawn; faded: the pick in run order; dash-dotted: the mean of the first K runs in run order, without selection. VANILLA states no prediction and is shown without a selector. End labels give the value at ten analyses.

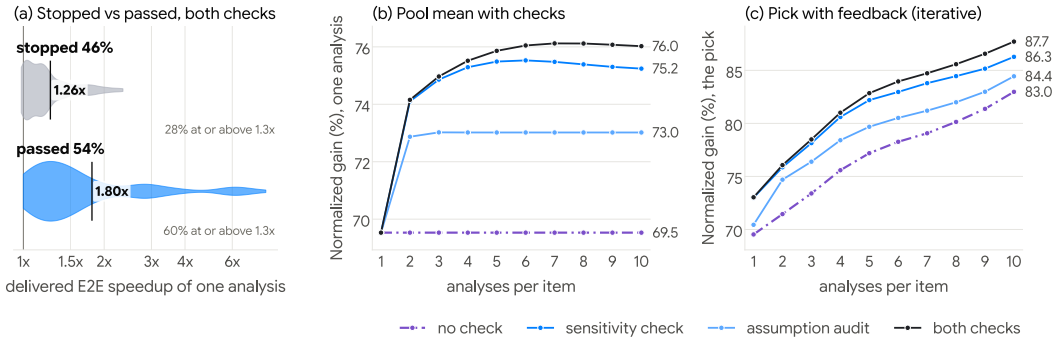


Figure 15: **Checks and iterative gap analysis**, all 100 items, E2E. (a) Delivered gain of the original analyses that both checks stop and pass, with each group's geometric mean. (b) The passed-set mean of the original pool against the pool mean without checks, over the number of analyses, per check (§C.6). (c) The pick of iterative gap analysis against the pick without checks, per check.

```
SENSITIVITY_CHECK(trace):
  for m in {E2E, TTFT, ITL}:
    claim[m], lev[m] = trace[m]
    # % change, and its leverage
  if max(lev) == 0: return STOP
  for m with lev[m] > 0:
    r[m] = |ln(1+claim[m]/100)|/lev[m]
  return STOP if max(r)<tau else PASS
  # tau = 0.01, fitted on the dev set
```

```
ASSUMPTION_AUDIT(analysis, corpus):
  v = agent(model = "claude-opus-5",
      tools = [read corpus, re-run analyze],
      prompt = "... decide whether each value
        in assume.yaml holds for THIS engine,
        configuration and workload ...")
  for a in v:      # holds | doubtful | wrong
    if a.wrong and a.load_bearing:
      return STOP
  return PASS
```

Figure 16: **The two checks on structured analyses.** Left: the sensitivity check, which perturbs all traced inputs (declared leaves and AST-lifted literals) to compute total leverage lev_m and stops programs that fail to run, have $\max_m \mathrm{lev}_m = 0$, or have maximum claim-to-leverage ratio $\max_m |\ln(1 + \mathrm{claim}_m/100)|/\mathrm{lev}_m < \tau$ (τ=0.01). Right: the assumption audit, a single read-only agent session that reads the corpus and re-runs the program to judge whether each declared assumption holds for this engine, configuration, and workload.

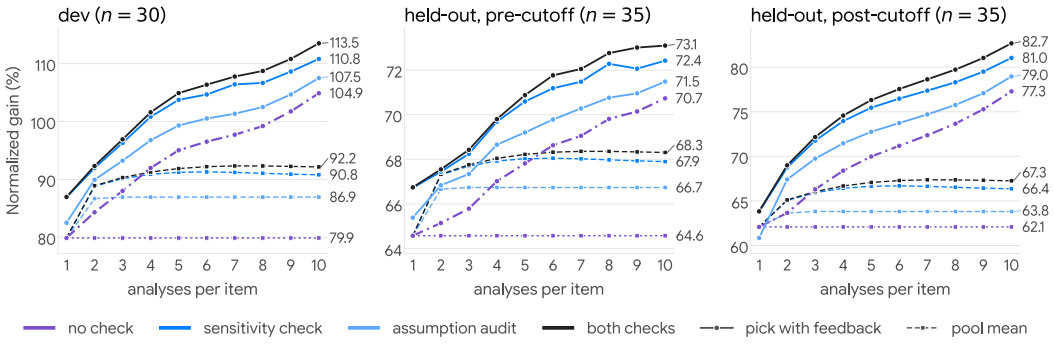


Figure 17: **Iterative gap analysis on the structured agent, per split.** The verdict of an analysis a check stops is returned to a fresh analyzer session, with up to three retries per slot. Each slot retains the first passing analysis or, when retries are exhausted, the last analysis. Dashed: the passed-set mean of the original pool (§C.6), against its mean without checks. Solid: the pick after retries, against the pick without checks. Sensitivity check, assumption audit and both; on the post-cutoff split both checks lift the pick from 77.3% to 82.7% of the expert's gain at ten analyses. Stop and pass distributions: Fig. 18.

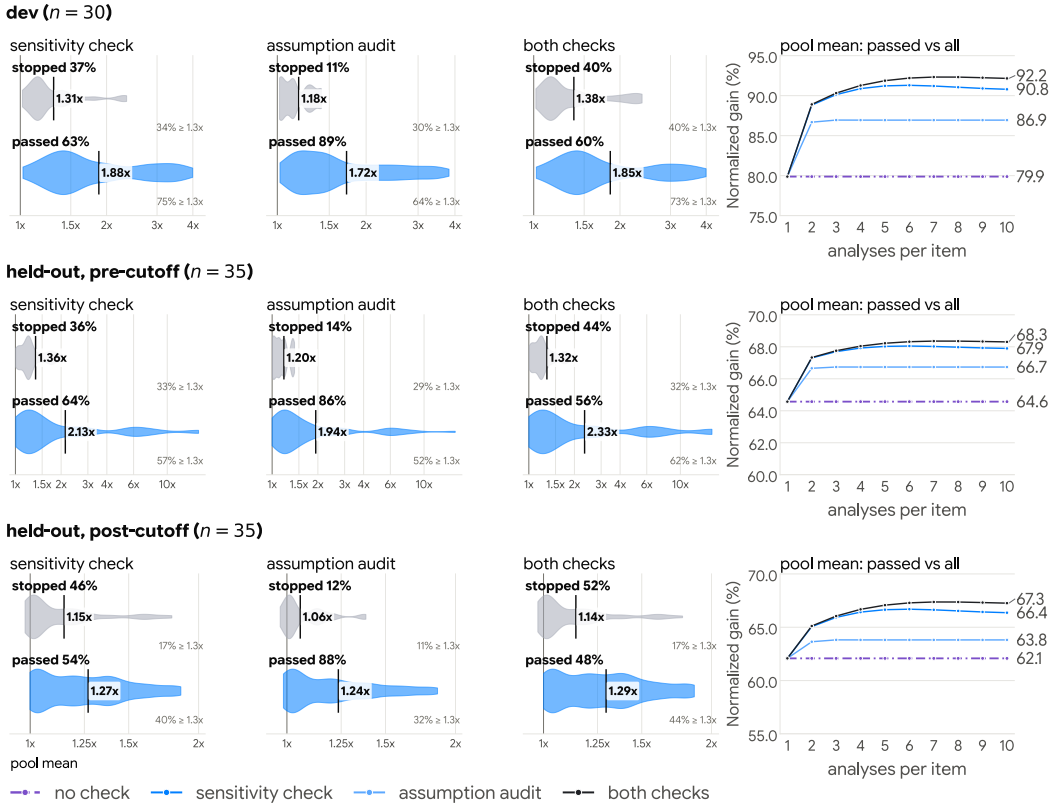


Figure 18: **What the checks stop and pass.** Per split (rows) and check (columns), the realized E2E gain of the original analyses the check stops and of those it passes, with each group's geometric mean, share and share above 1.3×. Right: the expected gain of an analysis drawn from the passed set against the number sampled, compared with no check.

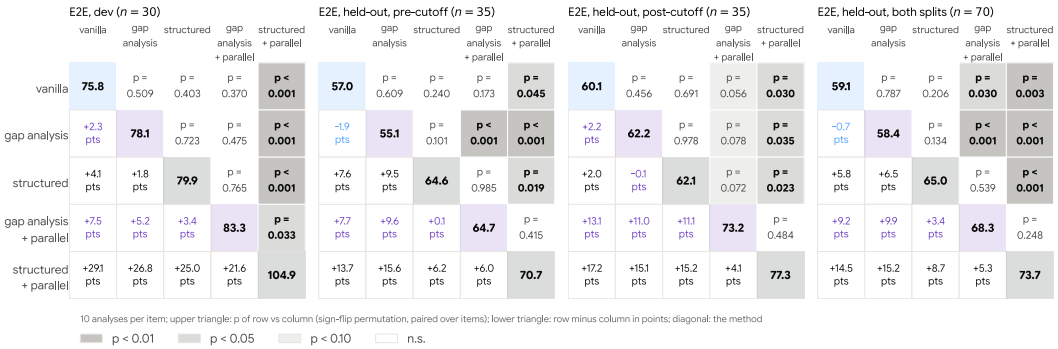


Figure 19: **Pairwise significance at** $K{=}10$, E2E, on the development split, on each held-out split, and on both held-out splits pooled. Upper triangle: p of row against column under a sign-flip permutation test paired over items, uncorrected for the ten comparisons per panel; lower triangle: the difference in points; diagonal: the method's value.

D.1 Beyond the expert

§2 treats the expert fix as a reference, not a target, and credits an agent for whatever it measures; Table 13 checks that the benchmark rewards routes to the headroom the expert did not take. The expert's fix is not a ceiling: 35–38% of delivered attempts exceed it. The share counts attempts, so large outliers cannot inflate it, and a delivery's gain enters the score capped at three times the reference's on a non-degenerate item (Table 5); on items whose reference gain sits near the noise floor, however, measurement noise alone can put a delivery above the reference (Eq. 2). Nor is it a template: about 40% of delivered patches touch the functions the expert touched, a third other functions in the same files, a quarter no file in common, and that last group beats the expert on 22–27% of gradeable attempts. Landing where the expert landed still predicts success (46–52% against 22–31%), the expected signature of a benchmark built from real fixes.

Table 13: **Where the agents' patches land, and how often they beat the expert.** Delivered attempts on all 100 items by overlap with the expert's fix: the same functions in the same files (hunk-level Jaccard ≥ 0.5), the same files but other functions, or no file in common. *ratio*: share of delivered attempts in the group; *>human*: share of the group's gradeable attempts beating the expert's E2E gain; *all*: over every gradeable attempt.

	same file + func		same file + diff func		diff file		all
setup	ratio	>human	ratio	>human	ratio	>human	>human
Vanilla	42.6%	52.3%	32.6%	30.6%	24.8%	21.9%	37.5%
Gap analysis	38.9%	46.5%	31.8%	31.4%	29.3%	24.7%	35.1%
Structured gap analysis	39.4%	48.0%	33.7%	28.3%	26.9%	26.7%	35.2%

D.2 Prediction accuracy and ranking consistency

Parallel gap analysis selects one candidate per item by ranking its $K{=}10$ analyses by predicted gain (§3). Table 14 and Fig. 20 evaluate how accurately predicted E2E speedups match delivered speedups and how consistently predicted rankings agree with measured rankings across all delivered, measured analyses that state a predicted E2E change.

Predicted versus delivered speedup. Across all individual analyses (Table 14, left), predicted E2E speedup correlates strongly with delivered E2E speedup on both held-out splits (Spearman $\rho = 0.76$–0.77 on pre-cutoff and 0.74–0.75 on post-cutoff; 0.52–0.55 on dev). Typical calibration is tight: the median multiplicative error factor, $\exp(\text{median}\,|\ln(\text{predicted}/\text{delivered})|)$, is $1.14\times$–$1.15\times$ on the held-out pre-cutoff split and $1.02\times$–$1.04\times$ on the post-cutoff split ($1.14\times$–$1.18\times$ on

dev). Aggregated to item-level geometric means (Fig. 20), predicted and delivered speedups track the $y{=}x$ diagonal across more than an order of magnitude of item headroom, with item-level rank correlations of $\rho = 0.70$–0.79 on pre-cutoff and $\rho = 0.82$–0.85 on post-cutoff.

Within-item ranking consistency and top-predicted selection. To test whether predicted gains order competing candidates for the *same* item in the same direction as real measurements, Table 14 reports pairwise *ranking consistency*: the fraction of within-item analysis pairs whose delivered gains differ by more than the $1.075\times$ measurement noise floor (§3) that the predictions order in the same direction as measured (50% is random chance). On the two held-out splits, predictions order separated within-item pairs correctly 65%–76% of the time (60%–62% on dev). Because within-item ranking is well above chance, selecting the single candidate with the highest predicted gain (*pick*) consistently improves over the pool *average* and captures a large share of the post-hoc *best*-of-ten ceiling (Table 14, right).

Table 14: **Prediction quality and pool selection across splits.** Evaluated over all analyses whose implementation was delivered and measured and which state a predicted E2E change (brackets: 95% item-bootstrap confidence intervals). *correlation*: Spearman ρ of predicted versus delivered E2E speedup across all analyses; *error*: median multiplicative error factor $\exp(\text{median}\,|\ln(\text{predicted}/\text{delivered})|)$; *ranking consistency*: share of within-item analysis pairs (with delivered gains more than $1.075\times$ apart) ordered by prediction in the same direction as measured (50% chance); *pool quality*: geometric-mean delivered E2E speedup over items of the pool *average*, the highest-predicted *pick*, and the post-hoc *best* of ten.

	prediction quality			pool quality (E2E speedup)		
setup	correlation ↑	error ↓	ranking consistency ↑	average	pick	best
Dev						
Gap analysis	0.52 [0.21, 0.73]	1.14× [1.11, 1.25]	62% [45, 80]	1.63×	1.65×	1.77×
Structured	0.55 [0.29, 0.73]	1.18× [1.12, 1.29]	60% [40, 80]	1.65×	1.71×	1.83×
Held-out, pre-cutoff						
Gap analysis	0.77 [0.56, 0.89]	1.14× [1.06, 1.20]	76% [56, 92]	1.72×	1.82×	1.84×
Structured	0.76 [0.52, 0.90]	1.15× [1.08, 1.18]	65% [46, 82]	1.84×	1.86×	1.96×
Held-out, post-cutoff						
Gap analysis	0.75 [0.60, 0.83]	1.04× [1.03, 1.08]	73% [61, 84]	1.24×	1.27×	1.34×
Structured	0.74 [0.55, 0.87]	1.02× [1.01, 1.05]	70% [58, 81]	1.23×	1.28×	1.34×

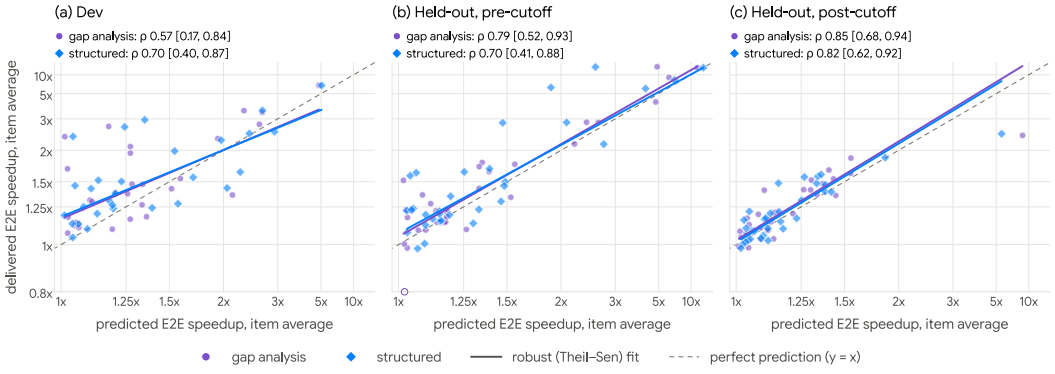


Figure 20: **Delivered versus predicted E2E speedup per item.** Each point is one item's geometric-mean delivered versus predicted E2E speedup over its delivered, measured analyses (log–log scale). Solid lines show robust Theil–Sen fits on log-speedup; the dashed diagonal marks perfect prediction ($y{=}x$). Legend entries give item-level Spearman rank correlation ρ and 95% bootstrap confidence intervals.

D.3 What a structured analysis rests on

The two checks of §3 rest on the trace of §C.6. Fig. 21 shows per-metric coverage, individual program compositions, accuracy by source, failures by unaccounted share and outcomes by assumption weight. The tracer AST-lifts undeclared literals into substitutable slots (§C.6), making their influence observable: among 848 traceable E2E claims, on average 58% of a claim's sensitivity weight rests on undeclared literals, and 21% credit no sensitivity weight to the profiles. Sensitivity weight measures the dependence of the prediction on an input, not the share of realized speedup physically caused by that input.

Profile-sensitivity groups. The ranking comparison uses 791 claims with delivered gains. The zero-weight group has $\phi_C < 10^{-9}$, where ϕ_C is the fraction of total sensitivity credited to the profiles, the sealed corpus C; a program in it may still have read the profiles, since a value can be read but unused, locally insensitive, or credited to an in-query literal. Among programs with nonzero profile weight, the tracer's *offset flag* is set when $\sum_{i\in C}|e_i| - |\sum_{i\in C} e_i| > 1$, using the signed elasticities e_i of certified profile leaves; the third group has nonzero profile weight without that flag. The threshold is a diagnostic: the valid estimate $r\,t_{\text{kernel}}/t_{\text{total}}$, a kernel's share of the run scaled by a constant r, has elasticities $+1$ and -1 in its two measured times and triggers the flag while depending meaningfully on their ratio.

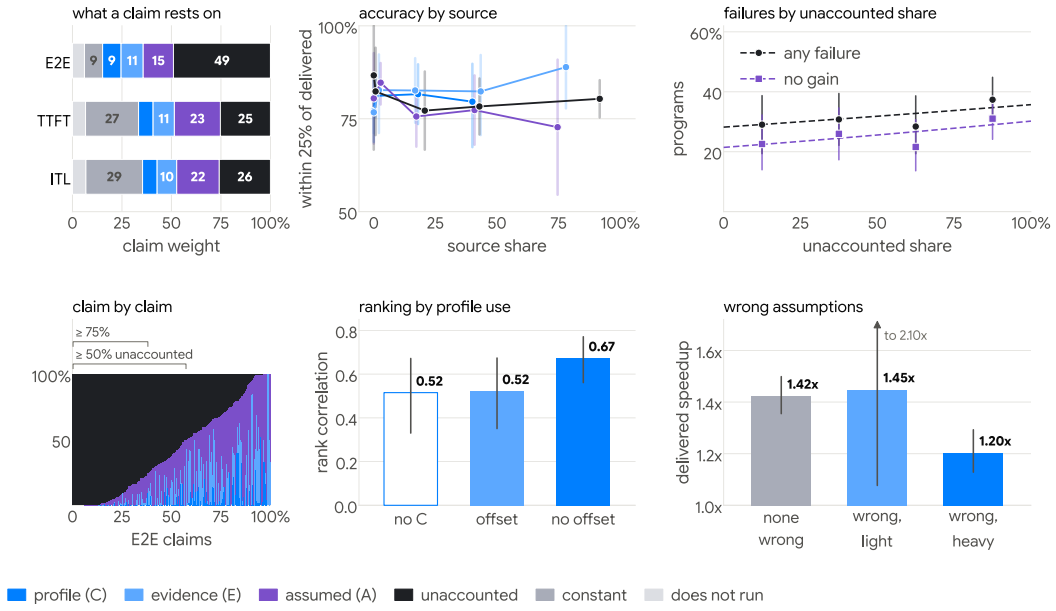


Figure 21: **Detailed source-tracing diagnostics for structured analyses.** All 1000 structured submissions are traced. Top left: per-metric coverage and input-source composition, including programs that do not run or return input-independent constants. Bottom left: sensitivity shares for each of the 848 traceable E2E claims. Middle: prediction accuracy by source share and rank correlation by profile-sensitivity group. Right: outcomes by unaccounted sensitivity share and delivered speedup by the heaviest assumption judged wrong. Intervals are 95% bootstrap intervals.

What the trace and the audit show

Typical prediction error is similar across source compositions: about four claims in five fall within 25% of their delivered gains. The three profile-sensitivity groups have observed rank correlations of 0.52 (zero weight), 0.52 (offset flag) and 0.67 (no offset flag), at the same rounded median error of 8%. Analyses whose heaviest wrong assumption has sensitivity weight ≥ 0.5 deliver 1.20× against 1.42× when no assumption is judged wrong ($p = 0.013$, attempt-level permutation test); wrong assumptions with little sensitivity weight show no comparable drop. The auditor's load-bearing label agrees with the trace's weight (AUC 0.76 over 1,882 audited assumptions). Together these motivate using the trace and the audit as pre-selection checks.

Checks on the pool: filtering and iterative gap analysis. Filtering keeps the passing analyses among the K drawn, so it enriches the pool it selects from but also shrinks it (Fig. 18). Iterative gap analysis instead returns the verdict of a stopped analysis to a fresh analyzer session, which produces a new analysis, up to three times per slot (Fig. 17). Every check lifts the passed-set mean of the original pool at every K above one, from 62.1% to 67.3% of the expert's gain for both checks on the post-cutoff split at $K{=}10$, and iterative gap analysis lifts the pick with it: 82.7% with both checks, 81.0% with the sensitivity check and 79.0% with the assumption audit, against 77.3% without checks; both checks together lead on every split, and over all 100 items they lift the pick from 83.0% to 87.7% (Fig. 15).

Intervals. Fig. 15 reports point values. The detailed diagnostics of Fig. 21 retain their attempt-level intervals and permutation test. The pooled test of §4 applies the sign-flip permutation of Fig. 19 to the per-item differences of the two held-out splits taken together.

D.4 A WEAKER ANALYZER OR IMPLEMENTER

The role ablation tests how changing either model affects the constrained analyzer–implementer pipeline (§3), and whether parallel gap analysis recovers gains lost with a weaker analyzer. Table 15 tests this on the held-out post-cutoff split with `claude-sonnet-5` in place of `claude-opus-5` in one or both roles (setup: §C.7).

Table 15: **Model-size ablation on the held-out post-cutoff split.** Normalized gain in percent, relative to the expert's gain, with `claude-opus-5` (Opus) and `claude-sonnet-5` (Sonnet) in each role; a row names the analyzer's model, then the implementer's. Left block: gap analysis; right: structured gap analysis. Rows without a "+": one analysis; + parallel: the best-predicted of the item's pool of ten, implemented, one measurement. The Opus-analyzer pool is the main experiment's; each Sonnet-analyzer arm has its own independent pool of ten samples per item. Every cell is measured directly. Bold is best per column.

	Gap analysis					Structured gap analysis				
setup	E2E	TTFT	ITL	TPS	RPS	E2E	TTFT	ITL	TPS	RPS
Single agent (vanilla)										
Opus	60.1	61.6	44.4	60.7	61.3			—		
Sonnet	46.9	53.7	37.3	49.0	49.5			—		
Analyzer → implementer										
Opus analyzer → Opus impl.	62.2	65.6	51.3	61.5	62.9	62.1	57.7	62.8	62.0	62.6
+ parallel	**73.2**	72.9	**59.1**	71.6	72.3	77.3	85.7	**73.2**	75.4	75.4
Sonnet analyzer → Sonnet impl.	57.4	62.1	47.3	56.5	57.7	51.4	44.9	40.9	52.1	54.2
+ parallel	72.7	66.6	52.6	**74.2**	**74.5**	70.0	57.2	63.0	72.2	73.8
Sonnet analyzer → Opus impl.	49.5	51.4	38.6	49.9	53.2	50.0	55.5	35.6	51.2	53.3
+ parallel	64.4	55.8	43.8	67.4	69.8	68.4	68.1	57.3	71.2	72.9
Opus analyzer → Sonnet impl.	61.5	66.9	45.8	59.7	61.6	63.5	74.0	47.9	64.3	65.9
+ parallel	72.5	**74.2**	53.5	69.7	71.0	**78.8**	**103.2**	57.9	**77.9**	**78.9**

Key takeaways from the model-size ablation

First, downgrading the single agent from Opus to Sonnet drops E2E normalized gain by 13.2 points ($60.1\% \rightarrow 46.9\%$). Second, in gap analysis, model capacity matters primarily in the analyzer role: averaged across implementers, replacing an Opus analyzer with Sonnet costs 8.4–12.1 points at $K{=}1$, whereas replacing an Opus implementer with Sonnet on the same Opus analyses costs no gain on average (the two Sonnet-analyzer arms draw independent pools of ten Sonnet analyses per item). Third, parallel selection over $K{=}10$ analyses closes roughly half of the smaller analyzer's gap to parallel Opus and enables an all-Sonnet parallel pipeline (70.0–72.7%) to outperform the single-session Opus baseline (60.1%) by 10–13 points.

D.5 Generalization across model families and CLIs

To verify that the findings are not specific to the Claude model family or the Claude Code CLI, Table 2 evaluates `gemini-3.1-pro-high` and `gemini-3.8-flash-high` running on the Antigravity CLI across the held-out post-cutoff split. The split is post-cutoff for Opus 5; the Gemini models' training data may extend past that date, so for them it is a held-out split rather than a guaranteed post-cutoff one (`gemini-3.8-flash-high` is also a newer model generation and completes turns faster within the fixed 1.5-hour wall-clock budget). Across both models the same progression holds: separating the analyzer at $K{=}1$ improves over the single agent ($58.4\% \rightarrow 60.7\%$ on Gemini 3.1 Pro; $68.6\% \rightarrow 70.7\%$ on Gemini 3.8 Flash), parallel gap analysis lifts E2E to 63.6% and 73.7%, and structured parallel gap analysis achieves the best performance on nine of the ten columns, reaching $\mathbf{65.4}\%$ E2E ($+7.0$ pp over Vanilla) and $\mathbf{76.7}\%$ TTFT ($+11.7$ pp) on Gemini 3.1 Pro, and $\mathbf{76.9}\%$ E2E ($+8.3$ pp) and $\mathbf{91.6}\%$ TTFT ($+13.9$ pp) on Gemini 3.8 Flash.

E Qualitative Studies and Examples

Individual deliveries rather than averages: seven mechanisms the structured agent found (Table 16), one delivery in depth (vLLM 4696, $1.65\times$ against the expert's $1.29\times$), the two analyzer designs on matched items, what an analysis can get wrong, and two deliveries end to end. The thing to take from it: the mechanism classes recur across items and engines, and were usually found in several independently sampled attempts using the supplied setup and profiles.

The examples are deliveries from the main run (§4). Gains are E2E speedups over the sealed baseline; "expert" is the reference fix measured on the same hardware.

E.1 A gallery of mechanisms

Table 16 groups seven representative mechanisms across three recurring bottleneck classes, most of which were independently rediscovered across multiple sampled attempts.

item	where is the gap, and the mechanism the agent implemented	agent	expert
Quadratic bookkeeping on the engine thread			
vLLM 4696	Prefix-cache keys were hashes of the whole prompt prefix, recomputed for each of 938 blocks of a 15k-token request. Chain each block's hash on the previous one and memoise.	1.65×	1.29×
vLLM 3469	Prompt-logprob detokenisation re-converted the whole prefix at every position. Carry the incremental state forward.	8.45×	8.97×
vLLM 44212	Three per-step loops re-scanned a request's 200 images to find nothing to do, on 2000 decode steps. Skip once decoding is past the last image.	1.87×	1.78×
Kernel geometry built for many tenants, paid by one			
SGLang 8940	LoRA kernels launch one block group per request; in decode each group fills a 16-row tile with one row and reloads the same adapter. Describe the batch as one segment.	1.78×	1.22×
vLLM 12185	Int4 MoE launched one kernel per expert, 2880 per token at batch one. Dequantise experts once at load.	2.35×	1.90×
A cache that was off, or an invariant fetched per request			
vLLM 16725	The outlines grammar cache was disabled with nothing in its place; the same 20-second automaton was rebuilt for each request. Add a bounded in-memory cache.	6.44×	7.12×
vLLM 14223	The API server validated the model's generation config against the Hugging Face hub on every request, blocking the event loop. Cache it once per process.	17.2×	17.0×

Table 16: Seven mechanisms across three recurring bottleneck classes. "agent" and "expert": E2E speedups over the sealed baseline for the best structured attempt and for the reference fix. Two beat the expert by a different root cause (8940, 12185); one improves the algorithmic complexity of the expert's own function (4696); the rest tie or closely trail it.

E.2 One delivery in depth: prefix-cache hashing (vLLM 4696)

One structured delivery from profile to measurement: the bug the upstream PR fixed, solved differently.

Setup and bug. vLLM 0.4.2 serving longchat-7b-16k on one H100 with prefix caching on; closed loop, four streams, 400 requests, every prompt 14 800 shared tokens followed by 200 unique ones, 16 output tokens. Baseline latency is 0.758 s per request, of which 55 ms is spent naming the prompt's 938 blocks on the single engine thread before the first token; the GPU is busy under 40% of the time. The shipped code rebuilt each 16-token block's whole-prefix hash from scratch: block i touches $16(i{+}1)$ tokens, so a request copies about seven million integers three times in Python (Fig. 22).

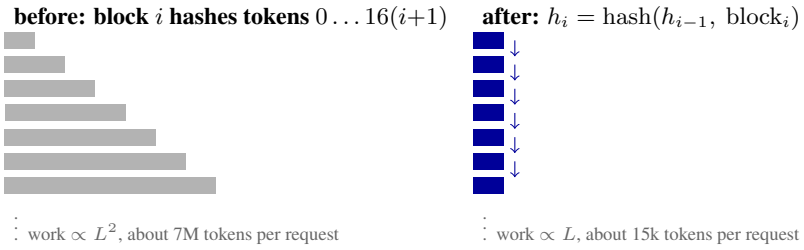


Figure 22: The per-block work before and after the agent's change. Full blocks never change, so each chained hash is memoised. Identical prefixes through block k produce the same chained key; as with the original hash-based cache, this identity is subject to hash collisions.

Reference fix versus agent fix. Both edit `hash_of_block` in `vllm/sequence.py`.

Expert, PR #4696 ($+14/-2$ lines, $1.29\times$)

PR #4696: vllm/sequence.py

```
# store the prompt as a tuple once
self._prompt_token_ids_tuple = \
    tuple(prompt_token_ids)

def hash_of_block(self, i):
    n = self.num_hashed_tokens_of_block(i)
    return hash((
        self.data.get_prefix_token_ids(n),
        self.lora_int_id))
# get_prefix_token_ids slices 0..n
```

Removes two of three copies per call; the per-block slice of the whole prefix remains.

Agent, structured attempt 3 ($\approx$70 lines, $1.65\times$)

structured attempt 3: patch.diff

```
def hash_of_block(self, i):
    cache = self._cached_block_hashes
    if i < len(cache):
        return cache[i]          # O(1)
    ids, B = self.data.get_token_ids(), 16
    prev = cache[-1] if cache \
           else hash((self.lora_int_id,))
    for k in range(len(cache), len(ids)//B):
        prev = hash((prev,
            tuple(ids[k*B:(k+1)*B])))
        cache.append(prev)
    ...                  # partial block
```

Each block costs 16 tokens plus the previous hash.

Result. Table 17 gives the three arms on the same hardware; the agent's normalized gain (§2) on E2E is 221% ($2.2\times$ the expert's gain). All three audited structured attempts found this mechanism ($1.65\times$, $1.65\times$, $1.72\times$); eight of nine audited attempts across the three agents (3 per configuration) landed 1.65–$1.73\times$, and the ninth, which removed only the tuple copy, landed $1.37\times$, consistent with the additional benefit of removing repeated whole-prefix work.

metric	baseline	agent	expert	agent gain	expert gain
E2E mean (s)	0.758	0.461	0.587	$1.65\times$	$1.29\times$
TTFT mean (ms)	360	173	252	$2.08\times$	$1.43\times$
ITL mean (ms)	25.2	17.9	21.0	$1.41\times$	$1.20\times$

Table 17: vLLM 4696, structured attempt 3: 400 requests served in every arm, 16 output tokens in every arm, no metric regressed. A same-node paired A/B gives $1.77\times$ against the reference's paired $1.29\times$.

The analysis. Four links. (1) *Profile:* py-spy on the serving node shows the engine thread CPU-saturated with 34.5% of its samples in `hash_of_block`. (2) *Measure the fix:* a microbenchmark on the real `Sequence` class gives $54.8 \to 0.33$ ms per request, 30.1% of the run's wall time. (3) *Model:* closed-loop at fixed concurrency, so mean latency falls by the fraction of wall time removed; Little's law reproduces the baseline to the digit ($3.714 = 5.528\ \mathrm{req/s} \times 0.672\,\mathrm{s}$). (4) *Predict:* the claim ships as the program of §3 (abridged):

vLLM 4696, structured attempt 3: analysis.py (abridged)

```
def analyze(C, E, A):   # corpus, evidence, declared assumptions
    wall   = C.scalar("SELECT value FROM bench.collect "
                      "WHERE key='run_bench_seconds'")
    served = C.scalar("SELECT value FROM bench.collect "
                      "WHERE key='run_served'")
    saved  = E.scalar("SELECT ms2s(value) "
                      "FROM evidence.hash_microbench "
                      "WHERE metric='saved_ms_per_request'")  # 0.0545 s
    removed = saved * served / wall \
              * A("critical_path_fraction")   # 1.0
              / A("cpu_speed_ratio")          # 1.0
    return {"E2E_mean": -100 * removed, ...}  # -30.1 %
```

`assume.yaml` declares the two knobs with their rationale: `critical_path_fraction` is 1.0 because the engine is enforce-eager and single-threaded and the hashing runs before the forward pass; `cpu_speed_ratio` is 1.0 because the serving-node py-spy share (34.5%) brackets the agent-node microbenchmark share (30.1%). Confidence was declared medium, with the TTFT/ITL split named as the soft part.

	predicted	measured	|log error|
E2E	−30.1%	−39.2%	0.14
ITL	−30.1%	−29.2%	0.01
TTFT	−30.1%	−51.9%	0.37

Table 18: Prediction against measurement for vLLM 4696, structured attempt 3. Signs right on every metric. The E2E under-claim partly reflects different baseline runs: the model used the analyzer's own profile run (0.672 s), while the sealed baseline is slower (0.758 s); TTFT moved more than E2E because the hashing sits entirely before the first token.

Checks on this delivery

The implementer followed the analysis in full, with one improvement; its paired same-node run gave $0.758 \rightarrow 0.429$ s, providing an additional paired measurement. Validation includes 279 checks on real `Sequence` objects with zero observed partition mismatches; decoding is greedy and output tokens are identical in all arms. The mechanism differs from the expert's: a chained, memoised hash, where the expert keeps the whole-prefix hash and removes copies. Later vLLM versions key their prefix cache by chained block hashes, however, and the item predates the cutoff, so recall cannot be ruled out.

E.3 What the program changes about the analysis: free-form versus structured

A free-form analyzer states its prediction in prose; a structured analyzer ships it as the program of §3. Over all deliveries the two are close: capped absolute log error on E2E 0.246 against 0.271, sign accuracy 0.889 against 0.894, both under-claiming on average (mean over-claim −0.25 and −0.28), and implementers followed the analysis in full in 89% and 85% of attempts (893 and 842). Both designs predict E2E with similar typical error; what structure adds is that the estimate is reproducible and an error is auditable afterwards; two matched items show it (Table 19).

item	agent	mechanism the analyzer named	predicted	measured	|log|
vLLM 4696	free-form #1	O(L^2) block hashing; microbench $57.8 \rightarrow 1.2$ ms	−30%	−41%	0.18
	free-form #2	same	−25%	−42%	0.25
	free-form #3	same	−19%	−41%	0.31
	structured #1	same; Little's-law closure	−25%	−42%	0.26
	structured #2	same; three measurements bracket each other	−29%	−39%	0.16
	structured #3	same; knobs declared	−30%	−39%	0.14
SGLang 8940	free-form #1	per-forward buffer rebuild (the expert's cost)	−13%	−19%	0.07
	free-form #2	same	−6.5%	−19%	0.15
	free-form #3	LoRA tile retune	−10%	−27%	0.21
	structured #1	per-forward buffer rebuild (the expert's cost)	−6.3%	−19%	0.15
	structured #2	per-request kernel geometry	−2.7%	−44%	0.55
	structured #3	per-request kernel geometry	−1.7%	−44%	0.56

Table 19: The same two items under both analyzer designs. Percentages are E2E change; "measured" is the delivery built on that analysis.

When the profile points at the function, the program makes the estimate reproducible. On 4696 the baseline profile puts a third of the engine thread's samples in the hashing path, and all six analyzers land on it with a similar microbenchmark. The structured central estimate is reproducible and its soft parts are named: attempt 3's only free parameters are two declared ratios set to 1.0 with a stated reason, and a reader who disagrees can change one number and re-run.

When the mechanism is larger than it first looks, the program makes the error auditable. On 8940 two of the three free-form analyzers chose the host-side buffer rebuild and predicted it within 0.07–0.15 log units. Two structured analyzers instead read the kernels and found the per-request

launch geometry, a cost twice as large. Their program summed the LoRA kernels over a CUDA-graph capture window: declared window bounds, a query over the sealed profile, correct arithmetic, and wrong, because graph capture runs the LoRA kernels with empty adapter ids, so the analyzer had priced the launch cost of an oversubscribed grid as weight traffic. Because every leaf was a query, re-running it exposes the mistake in one line. The delivery still landed at $1.78\times$ because the implementer measured before shipping (§E.4, first row).

E.4 What an analysis can get wrong

An analysis can name the right mechanism and mis-size it, rest on a premise the code refutes, or optimise a metric the card does not score. Table 20 gives cases of each and how they surfaced; in each, the implementer's own measurement or reading caught the error before delivery.

Failure mode	Case	How it surfaced
Right mechanism, wrong size	SGLang 8940 (structured #3): LoRA kernel geometry sized from a CUDA-graph capture window in which the kernels ran empty; predicted -1.7%, measured -44%.	Implementer's interleaved A/B ($39 \rightarrow 15.7$ ms ITL); re-running the corpus query exposes the error.
Premise refuted at implementation	SGLang 10286 (#2): analysis called the server GPU-bound and proposed a tile retune; implementer found it host-bound, swapped in a CUDA-graph batch-cap change, and landed $0.94\times$. vLLM 48144: analysis claimed a backend reorder would move mixed batches to full CUDA graphs (-15% E2E); the code routes them to piecewise graphs unconditionally, landing $0.98\times$. vLLM 44858 (#3): analysis patched the V1 model runner while the V2 runner executed; implementer abandoned.	`follows_analysis: partial` with a written departure; an honest abandon.
Wrong target	vLLM 41992 (#4): the card scores TTFT; the analysis optimised E2E and abandoned. The implementer re-targeted and landed $1.24\times$ against the expert's $1.38\times$.	Implementer read the card.

Table 20: Representative cases of three ways an analysis can go wrong. Attempt numbers index the item's ten analyses (§4).

What the checks do not judge. The checks of §A.4.2 establish that a delivery serves the same work with equivalent output; whether the change is one the upstream project would merge is a judgment the benchmark does not make. Two valid deliveries sit at that boundary: vLLM 50992 raises a documented threshold so that the CPU KV tier never stores on a no-reuse workload ($1.57\times$), and vLLM 45757 removes the per-step retry of a KV store that always fails ($1.64\times$, the ceiling of disabling the store); upstream would be unlikely to merge either as written. The analysis-side errors of Table 20 were surfaced by the implementer, who must measure before delivering.

E.5 Two deliveries, end to end

One excerpt each from the analysis and the delivered diff shows the mechanism in the agent's own words for two end-to-end deliveries, summarized side by side in Table 21.

SGLang 8940. The analysis, and the patch, which touches no kernel but describes the batch as one segment of length `bs` instead of `bs` segments of length 1:

SGLang 8940, structured #3: analysis.md

```
1. BLOCK_S=16 over seg_len=1. Every LoRA kernel tiles the token
   dimension at BLOCK_S = 16 but a decode segment is a single row,
   so 15/16 of each M tile is masked away.
2. Weights re-read once per sequence. The launch grid's Z axis is
   the batch size, and batch_id = program_id selects the adapter.
   With only one adapter loaded, every sequence's program re-streams
   the same weights[w_index] from HBM: the read-once 55 us is paid
   ~batch times.
```

SGLang 8940, structured #3: patch.diff

```
+  if (getattr(self, "enable_dense_decode", False)
+          and forward_batch.forward_mode.is_decode()
+          and len(cur_uids) == 1):
+      dense = self.cuda_graph_dense_batch_info
+      dense.seg_lens.fill_(bs)
+      dense.seg_indptr[1:].fill_(bs)
+      dense.bs = 1
+      dense.max_len = bs
+      batch_info = dense
```

vLLM 41868. The analysis quotes the dispatch check; the patch zero-pads the CUDA case to the next multiple of 16 before CUTLASS and slices the padding off, the same GEMM on a different kernel, as the expert's PR also argues:

vLLM 41868, structured #2: analysis.md

```
cutlass_compatible_b = b.shape[0] % 16 == 0 and b.shape[1] % 16 == 0
if current_platform.is_rocm() or not cutlass_compatible_b:
    out = triton_scaled_mm(...)        # slow naive GEMM

Qwen2.5-VL's vision tower trips the gate: intermediate_size = 3420,
which is not a multiple of 16 (3420 = 16*213 + 12). The vision SwiGLU
MLP has a fused gate_up_proj (N = 2*3420 = 6840) and a down_proj
(K = 3420); both are FP8-quantized and both fail the divisibility
test.
```

vLLM 41868, structured #2: patch.diff

```
-  if current_platform.is_rocm() or not cutlass_compatible_b:
+  if current_platform.is_rocm():
       out = triton_scaled_mm(a, b, scale_a, scale_b, out_dtype, bias)
+  elif not cutlass_compatible_b:
+      K, N = b.shape
+      K_pad = (K + 15) // 16 * 16
+      N_pad = (N + 15) // 16 * 16
+      ...   # zero-pad a, b along K; b, bias, scale_b along N
+      torch.ops._C.cutlass_scaled_mm(out, a, b, scale_a, scale_b, bias)
+      out = out[:, :N].contiguous()
```

Stage	SGLang 8940 (structured #3)	vLLM 41868 (structured #2)
Deployment	SGLang 0.5.0rc0, Llama-3.1-8B-Instruct with one $r{=}64$ LoRA adapter, $1{\times}$H100; open loop at 8 req/s, 2400 streamed chat requests, prompts $\sim$290 tokens, outputs $\sim$170.	vLLM at `6881c754`, Qwen2.5-VL-7B-Instruct with `--quantization fp8`, $1{\times}$H100; open loop at 36 req/s ($3\times$ overloaded), 1320 requests of 512 text tokens plus one $512{\times}512$ image, 60 out.
Mechanism	LoRA kernels tile the token axis at 16 rows and launch one block group per sequence; in decode each segment is one row, so $15/16$ of every tile is masked and the single adapter is re-read once per sequence (192 launches/step).	The FP8 dispatcher picks CUTLASS only if both weight dimensions are multiples of 16 and otherwise falls back to Triton with no log; the vision MLP width is 3420, taking 81.4 of 107.7 s of GPU time (75.6%).
Prediction	E2E -1.7%, ITL -1.7%, TTFT 0%; low confidence (sized from a capture window in which the kernels ran empty).	E2E -59.1%, TTFT -60.9%, ITL -53.5%; sized from a short local probe ($52.8 \to 21.6$ s; analyzers are not allowed to run the full benchmark).
Delivered	E2E $4.794 \to 2.692$ s ($1.78\times$, -43.8%); ITL $1.79\times$; TTFT $1.27\times$; 2400 served in all arms.	E2E $55.6 \to 20.5$ s ($2.72\times$, -63.2%, $\lvert\log\rvert$ error 0.105); TTFT $2.84\times$; ITL $2.33\times$; 1320 served in all arms.
Expert fix	Hoisted a per-forward rebuild of fused LoRA-B host buffers: $1.22\times$ E2E, $1.22\times$ ITL, $1.19\times$ TTFT. Different root cause; the two fixes compose.	Same diagnosis and pad-to-16 fix, placed in the CUTLASS kernel class rather than the shared dispatcher: $2.27\times$ E2E, $2.39\times$ TTFT, $2.14\times$ ITL.
Notes	4 of 9 attempts across agents land the collapse at 1.64–$1.78\times$, reproduced in a same-node A/B; restricted to one adapter as written.	8 of 10 attempts land 2.58–$2.74\times$; the $1.20\times$ edge over the expert reflects $3\times$ overload compounding small throughput differences.

Table 21: Two end-to-end deliveries side by side ("Expert fix" is the reference fix measured on the same hardware). On SGLang 8940 the analyzer identified the right mechanism but mis-sized it; on vLLM 41868 it sized the gain from a short local probe (never running the full benchmark, which is not allowed) and came within 4 percentage points.

F Limitations and Future Work

LMSysDojo is built for optimizing recognized, fixed setups, which makes it a clean environment for evaluating and developing agents and also marks its limits and the directions ahead. (1) **Dynamic setups.** Each item holds one workload on one hardware configuration fixed while the agent works, whereas real deployments shift by the hour, share an engine among workloads and change hardware underneath; discovering mechanisms that hold across such a mix, or adapt to it, is the natural next task, and the constructor can pose it by attaching several setups to one item. Most optimizations pay off only in some regimes (§B.4), so a setup-dependent gain is expected; the score does not, however, distinguish a mechanism whose benefit depends on the setup from code special-cased to it, such as the single-adapter fast path in SGLang 8940 (§E.5), and whether upstream would merge a change is not part of the score (§E.4). (2) **Setup discovery.** LMSysDojo takes the choice of setup away from the agent, since that choice rests on knowledge outside the system layer and would confound the measurement of optimization skill; yet knowing where room exists is itself a hard and valuable problem, one the experts behind our PRs solved before writing code, and the corpus of 900 measured setups, each with its motivating context, is a starting point for it. (3) **Beyond optimization.** Optimization is the activity of systems research easiest to measure, since a speedup on a fixed setup is a number; designing systems, setting abstractions and deciding what to build have slower, less crisp measures of success. Our analysis–implementation decomposition motivates studying whether analyses can be sampled and judged in these settings too; a useful scorer for these broader tasks remains open. (4) **Construction and scope.** The constructor's stages are carried out and checked by agents (§2), every result is on one node type ($8{\times}$H100), and most experiments use one model family, so the transfer of scores across hardware and the pipeline's robustness to construction errors remain to be tested. Within the evaluated protocol, the results motivate improving analysis quality and the selection of analyses for implementation.