ALPHALAB: Autonomous Multi-Agent Research Across Optimization Domains with Frontier LLMs

Brendan R. Hogan* **Xiwen Chen** **James T. Wilson** **Kashif Rasul**

Adel Boyarsky **Thomas Kamei** **Anderson Schneider** **Yuriy Nevmyvaka**

Morgan Stanley

Abstract

We present ALPHALAB, an autonomous research harness that leverages frontier LLM agentic capabilities to automate the full experimental cycle in quantitative, computation-intensive domains. Given only a dataset and a natural-language objective, ALPHALAB proceeds through three phases without human intervention: (1) it adapts to the domain and explores the data, writing analysis code and producing a research report; (2) it constructs and adversarially validates its own evaluation framework; and (3) it runs large-scale GPU experiments via a Strategist/Worker loop, accumulating domain knowledge in a persistent playbook that functions as a form of online prompt optimization. All domain-specific behavior is factored into adapters generated by the model itself, so the same pipeline handles qualitatively different tasks without modification. We evaluate ALPHALAB with two frontier LLMs (GPT-5.2 and Claude Opus 4.6) on three domains: CUDA kernel optimization, where it writes GPU kernels that run 4.4× faster than `torch.compile` on average (up to 91×); LLM pretraining, where the full system achieves 22% lower validation loss than a single-shot baseline using the same model; and traffic forecasting, where it beats standard baselines by 23–25% after researching and implementing published model families from the literature. The two models discover qualitatively different solutions in every domain (neither dominates uniformly), suggesting that multi-model campaigns provide complementary search coverage. We additionally report results on financial time series forecasting in the appendix, and release all code at https://brendanhogan.github.io/alphalab-paper/.

1 Introduction

Frontier LLMs have dramatically improved in agentic and software development ability since December 2025 . These models are genuinely *agentic*: given access to a terminal, a file system, and the internet, they can operate autonomously over extended periods, writing and debugging code, searching the web, installing dependencies, and iterating on their own output until a task is complete. On SWE-bench Verified, solve rates among top agentic systems rose from roughly 30% to over 70% in under twelve months (Jimenez et al., 2024); the duration of tasks that models can sustain autonomously has grown from minutes to hours.

This shift has a compounding consequence. The *harnesses* that make these models effective (prompts, tool configurations, scaffolding code, evaluation pipelines) are themselves just software that the models are now good enough to write. Products like Claude Code (Anthropic, 2025) and Codex (OpenAI, 2025) already demonstrate this: a frontier model builds its own tooling, evaluates results, and revises the tooling in a loop of recursive self-improvement. For a growing class of problems, the answer is not fine-tuning but *harness*

[*]Corresponding author: brendan.rappazzo@morganstanley.com. Other authors: firstname.lastname@morganstanley.com.

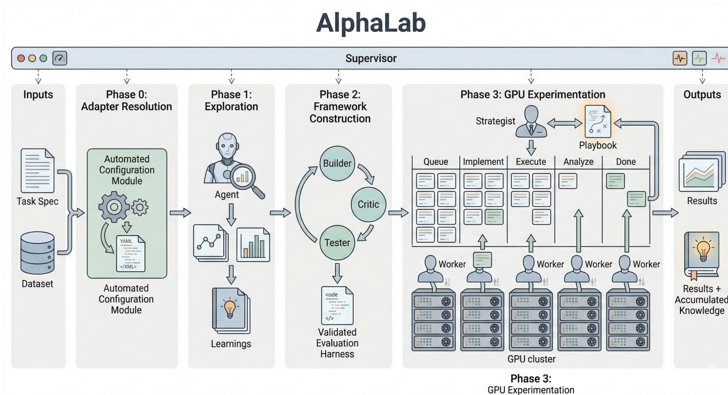


Figure 1: **AlphaLab pipeline overview.** Given a dataset and objective, the system writes all its own code across four phases. **Phase 0:** configures domain-specific prompts/metrics. **Phase 1:** explores the dataset and researches prior work via web search. **Phase 2:** adversarial Builder/Critic/Tester loop constructs the evaluation framework. **Phase 3:** Strategist proposes experiments, Workers implement them, and jobs run on a GPU cluster via Slurm; a persistent Playbook evolves with each experiment. A Supervisor monitors health. The human can interact with the Strategist to guide search; we run fully autonomously in this paper for fair comparison.

engineering: building the right scaffolding and letting the model refine it (PoetIQ, 2025; Chollet et al., 2025). If this works for software engineering, an obvious question is: what about scientific research?

Automating science occupies a unique position in the landscape of AI ambitions (Lu et al., 2024; Yamada et al., 2025; Jiang et al., 2025; Schmidgall et al., 2025). The scientific method (forming hypotheses, designing experiments, updating beliefs from evidence) is arguably what most distinguishes human intelligence from other forms of cognition. A system that can do this fully end-to-end (autonomously identifying important problems, designing experiments, and producing genuine insight) would represent something close to artificial general intelligence. In our view, current models are not there yet. But even in the regime where AI serves as a very sophisticated tool rather than an autonomous scientist, the practical impact is enormous: for any domain with objective, easy-to-evaluate metrics (training loss, kernel speedup, forecast accuracy), the combinatorial search over architectures, hyperparameters, and implementation strategies is exactly the kind of work that is tedious and error-prone for humans but natural for an autonomous system with enough compute. The bottleneck to progress in medicine, energy, materials science, and countless other fields is not a shortage of ideas but a shortage of person-hours to test them, and AI systems that could *multiply* human research throughput by orders of magnitude would have enormous impact.

There is a useful analogy from competitive chess: for a decade after Deep Blue, the strongest players were human–AI teams (Kasparov & Greengard, 2017); research may be in this phase now. Frontier models can handle literature search, code implementation, hyperparameter sweeps, and failure analysis, but they still benefit from human direction: choosing which problems matter, recognizing misguided lines of inquiry, and providing the judgment that turns throughput into insight. The question is: *what is the right division of labor, and what infrastructure makes human–AI research collaboration most productive?*

To this end, we introduce AlphaLab, an autonomous multi-agent system inspired by Claude Code, designed to extend frontier-model agentic capabilities from software engineering to scientific research through computationally intensive experimental campaigns. The user provides a dataset and a natural-language objective; the system handles the experimental grind while the human monitors progress and can interact with the Strategist agent to suggest experiments, prune directions, or inject intuitive leaps. AlphaLab operates through three phases: (1) it **adapts and explores**, configuring prompts and metrics, writing

analysis code, and producing a research report; (2) it **builds its own evaluation framework** via an adversarial Builder/Critic/Tester loop; and (3) it **experiments at scale** via a Strategist/Worker loop on GPU clusters. All domain-specific behavior lives in plug-in adapters generated by the model itself, and a persistent *playbook* accumulates domain knowledge across experiments. We cap each campaign at 50 experiments and run fully hands-off for fair comparison, but the intended workflow is collaborative.

We evaluate ALPHALAB on three deliberately diverse domains with two frontier LLMs (GPT-5.2 and Claude Opus 4.6). On **CUDA kernel optimization** (Ouyang et al., 2025), the system writes custom GPU kernels and benchmarks them against PyTorch's optimized compiler; it produces kernels that run 4.4× faster on average, with the best kernel achieving a 91.4× speedup. On **LLM pretraining** (PleIAs & Langlais, 2025), given a 20-minute compute budget the system searches over model architectures and training recipes; the full system achieves 22% lower validation loss than a single-shot baseline using the same model. On **traffic forecasting** (Lai et al., 2018), the system beats seasonal baselines by 23–25% after researching and implementing published model families from the literature (e.g., PatchTST, TFT, iTransformer). Baselines for a system like ALPHALAB are inherently difficult to construct: because the system searches the literature, selects architectures, and tunes hyperparameters as part of a single campaign, there is no single model to compare against. We therefore compare against other autonomous research systems, simple iterative LLM loops (Karpathy, 2026), single-shot LLM baselines, and published results for the individual model families that ALPHALAB itself discovers. Each campaign costs $150–200 in LLM API calls (not including GPU compute) and completes in 12–48 hours on 4×H100 hardware.

Our contributions:

1. We introduce ALPHALAB, an autonomous multi-agent research system with self-generated domain adapters, adversarial evaluation construction, and a persistent playbook for knowledge accumulation, enabling one pipeline to handle qualitatively different domains.
2. We evaluate across three domains with two frontier LLMs, ablating key components and showing that different models discover complementary solutions.
3. We open-source the full codebase at `https://brendanhogan.github.io/alphalab-paper/` with prompt templates, playbook excerpts, and additional results in the appendix.

2 System design

At a high level, ALPHALAB is a *harness*: a combination of tools and a structured environment that converts a frontier LLM into an autonomous research agent. We define the harness formally as a tuple $\mathcal{H} = (\mathcal{M}, \mathcal{T}, \mathcal{E})$, where $\mathcal{M}$ is a frontier language model (treated as a black box), $\mathcal{T}$ is a set of tools the model can invoke, and $\mathcal{E}$ is a phased environment that structures the research workflow. The system is **LLM-agnostic**: any model that supports tool-use and multi-modal input (agents must be able to read plots and visualizations they generate) can be substituted for $\mathcal{M}$ with no changes to $\mathcal{T}$ or $\mathcal{E}$. We evaluate with GPT-5.2 and Claude Opus 4.6 in Section 3; all differences in outcomes are attributable to the model, not the infrastructure.

2.1 Tools

Every agent in ALPHALAB has access to the same core tool set $\mathcal{T}$ (full listing in Appendix A.1). The three most important tools are: **shell access** (`shell_exec`), which accounts for ∼50% of all tool calls: the agent runs as a user account in a Unix shell and can do anything a human could from a terminal: write code, install packages, run training jobs, manage Slurm submissions, and debug failures; **web search** (`web_search`), used heavily in early phases to research prior work, survey existing approaches, and read library documentation before writing code; and **sub-agent spawning** (`spawn_agent`), which launches a new instance of $\mathcal{M}$ with its own context window and full tool access, enabling recursive delegation without polluting the parent agent's context.

2.2 Environment: four-phase pipeline

The environment $\mathcal{E}$ structures the research workflow into four sequential phases (Figure 1); we summarize each below and provide detailed walkthroughs, prompt examples, and case studies in Appendix A. Each phase produces artifacts that are consumed by subsequent phases, and a Supervisor agent monitors health across the pipeline, intervening when error rates spike (Section 2.2.5).

2.2.1 *Phase 0: Adapter resolution*

All domain-specific behavior in AlphaLab is parameterized by a **domain adapter** $\mathcal{A}$: a collection of 11 files comprising a manifest (metric definitions, experiment structure, entry points), 9 prompt templates (one per agent role), and a domain knowledge document that is injected into every agent's context for the entire campaign.

Phase 0 resolves the adapter by resuming from a prior campaign, customizing a built-in template, or, for novel domains, generating all 11 files from scratch by examining the dataset and searching the web for relevant prior work. The key idea is that *prompt engineering is performed by the model*, grounded in the actual data. The most important file is `domain_knowledge.md`, an artifact prepended to every agent's context for the entire campaign, encoding metric formulas, data quirks, and domain priors so that downstream agents need not re-discover them.

2.2.2 *Phase 1: Data exploration*

A single Explorer agent operates autonomously for several hours: it first generates a `plan.md` checklist, then works through it, writing and running Python scripts, generating plots, searching the web for relevant papers and best practices, and updating its notes after each finding. It produces two outputs: a human-readable research report (`data_report/`) and a machine-readable `learnings.md` consumed by Phases 2 and 3. The plan-then-execute structure prevents drift; further detail is in Appendix A.4.

2.2.3 *Phase 2: Adversarial evaluation construction*

Evaluation correctness is the single most important property of an autonomous research system. If the metric is wrong, every experiment optimizes the wrong objective, the playbook fills with false knowledge, and errors compound silently. Phase 2 addresses this through a multi-agent adversarial loop:

$$\text{Builder} \xrightarrow{\text{code}} \text{Critic} \xrightarrow{\text{review}} \begin{cases} \text{Tester} & \text{if no critical issues} \\ \text{Builder} & \text{otherwise} \end{cases} \tag{1}$$

The **Builder** receives Phase 1 learnings and constructs the full evaluation framework (data loading, splitting, metric computation, orchestration). The **Critic**, a fresh agent with no shared context, audits for data leakage, lookahead bias, and metric errors. The **Tester** writes and runs an automated test suite; the loop terminates when all tests pass or a maximum iteration count $I_{\max}$ is reached. The cap is necessary because an LLM instructed to find issues will *continue to* find something to flag; in practice, all substantive problems are resolved within 5–10 iterations, after which the remaining findings are stylistic or inconsequential.

2.2.4 *Phase 3: GPU-scale experimentation*

Phase 3 is the core of the system: a sustained experimental campaign where the Strategist proposes experiments, Workers execute them on a GPU cluster, and a playbook accumulates knowledge. A pure-Python Dispatcher orchestrates the process without making any LLM calls itself. The Dispatcher manages cluster resources through Slurm: it tracks available GPUs, writes and submits Slurm job scripts, monitors job status, and automatically reassigns freed GPUs to the next highest-priority task. This means the system can keep a multi-node GPU cluster fully utilized around the clock: experiments are queued, dispatched,

and monitored without any human involvement, and the cluster never sits idle between experiments.

Experiment lifecycle. Each experiment e progresses through a state machine:

$$e: \quad \texttt{queued} \rightarrow \texttt{implement} \rightarrow \texttt{execute} \rightarrow \texttt{analyze} \rightarrow \texttt{done} \tag{2}$$

with a `fix` state for failed experiments (limited to $k = 2$ repair attempts). The Dispatcher assigns tasks to Workers with a strict priority ordering:

$$\text{priority}(\texttt{fix}) > \text{priority}(\texttt{analyze}) > \text{priority}(\texttt{implement}) \tag{3}$$

This ordering maximizes information flow: fixes recover sunk GPU cost, analysis produces debriefs that inform the Strategist, and implementation can always wait.

Strategist. Invoked periodically, the Strategist receives the current leaderboard, all recent experiment debriefs, the full playbook, Phase 1 learnings, and remaining budget B. It outputs new experiment specifications, cancellation decisions, and an updated playbook. Budget management is graduated from broad exploration ($B > 20$) to focused refinement ($B \leq 10$) to stopping ($B = 0$).

Worker. Each Worker receives exactly one task: `implement` (write code, test locally, submit to Slurm), `analyze` (extract metrics, write a structured debrief), or `fix` (diagnose failure, patch, resubmit; up to k=2 attempts). Workers do not communicate directly; the playbook is the sole channel through which one Worker's findings reach the next.

Playbook. The persistent knowledge artifact and central feedback mechanism. After each Strategist turn, the playbook is updated with compressed findings: what works, what fails, and why. It is injected into the context of every subsequent Strategist and Worker call, creating a feedback loop that functions as online prompt optimization: by the end of a campaign, it contains domain-specific methodology that did not exist at launch.

Convergence. The campaign terminates when no improvement in the primary metric has occurred for C consecutive experiments (default $C = 20$), or the budget B is exhausted (in practice, all campaigns in this paper were budget-limited; see Appendix B.6).

2.2.5 *Supervisor*

A meta-agent that validates artifacts between phases and monitors Phase 3 health. The Supervisor is triggered when the error rate exceeds a threshold $\tau = 0.4$ over a sliding window:

$$\frac{|\{e \in W : e.\text{status} = \texttt{failed}\}|}{|W|} > \tau \implies \text{trigger Supervisor} \tag{4}$$

When triggered, the Supervisor reads recent failure logs, diagnoses systemic issues, and patches the adapter's `domain_knowledge.md`, committing the change to git so it is traceable and reversible. Examples of Supervisor interventions are detailed in the appendix.

3 Experiments and Results

We refer to a full end-to-end pipeline execution as a *campaign*, and each individual architecture or configuration tested within a campaign as an *experiment*. Every meaningful campaign takes approximately 40 hours and costs $150–200 in LLM API calls (GPU compute costs are additional and depend on infrastructure), making exhaustive ablation prohibitive; all single-run comparisons are indicative rather than conclusive. To ensure apples-to-apples comparison, we run Phase 2 once and copy the harness to all runs within a domain. Every domain runs with GPT-5.2 and Claude Opus 4.6, all other variables held constant. As discussed in Section 1, we compare against a greedy loop baseline (Karpathy, 2026), a single-shot LLM baseline, and published results for model families that AlphaLab itself discovers. Hardware: 4×H100 NVL 80 GB; budget: 50 experiments per campaign. The greedy loop baseline is given the same 50-iteration budget; the single-shot baseline is intentionally minimal (one call with one retry) and serves as a lower bound.

Model	Best val_bpb	Best config	Cost
AlphaLab + GPT-5.2	0.9697	8L×512d, GQA, cosine	∼$150
AlphaLab + Sonnet 4.6	0.8686	11L×768d, QK-norm, Muon	∼$120
AlphaLab + Opus 4.6	**0.7578**	10L×752d, QK-norm	∼$200
AlphaLab + GPT-5.1-mini	—†	(no valid results)	∼$40
Greedy loop (GPT-5.2)	1.020	12L×768d, LLaMA, AdamW	∼$50
Single-shot (GPT-5.2)	1.248	27.4M LLaMA-style	<$1

Table 1: **LLM pretraining speedrun results.** Task: train a <100M-parameter language model from scratch on the PleIAs SYNTH corpus under a 20-minute wall-clock budget on a single H100, then measure validation bits-per-byte (val_bpb, lower is better) on a held-out set. Configs describe the best architecture found: "L" = layers, "d" = model dimension, "GQA" = grouped-query attention, "QK-norm" = query/key normalization for training stability, "cosine"/"Muon" = learning rate schedule or optimizer. The greedy loop is a Karpathy-style AutoResearch baseline (Karpathy, 2026): the LLM proposes a modification, trains, and keeps or reverts based on the metric (50 iterations, 3 improvements). The single-shot baseline is one LLM call with one retry. †GPT-5.1-mini ran 33 experiments but failed to correctly implement the BPB metric; all results were broken (see text).

3.1 LLM pretraining (speedrun)

Task. Inspired by Karpathy's NanoGPT speedrun (Jordan et al., 2024), the agent must minimize validation bits-per-byte (`val_bpb`) on the PleIAs SYNTH corpus (PleIAs & Langlais, 2025) when training a <100M parameter language model from scratch under a 20-minute wall-clock budget. The optimization space spans architecture (depth, width, attention/FFN variants, normalization, positional encoding), optimizer (AdamW with various schedules), and training dynamics.

Main results. Table 1 summarizes results across three models and two baselines. Opus 4.6 achieves a best val_bpb of **0.7578**, substantially outperforming GPT-5.2 (0.9697) and Sonnet 4.6 (0.8686), a 22% improvement over GPT-5.2. Sonnet 4.6 falls between the two at 0.8686, converging on a distinct recipe (11L×768d with a Muon/AdamW hybrid optimizer and QK-norm) that neither other model discovered. GPT-5.1-mini, a smaller and cheaper model, failed entirely: it ran 33 experiments but could not correctly implement the nats-to-BPB conversion, producing values ranging from 0 to ∞; it self-diagnosed the issue in its own playbook but only after exhausting the budget. The gap between Opus and GPT-5.2 is driven primarily by GPT-5.2's 38% experiment failure rate (PyTorch API breaking changes) and Opus's convergence on wider-shallower architectures (10–12 layers, 672–752-dim) better suited to the <100M regime.

Ablations. Because this domain has the fastest iteration cycle, we concentrate all ablations here. Each ablation modifies a single pipeline component while holding everything else constant (same hardware, 50-experiment budget, shared Phase 2 harness). Table 2 summarizes results.

Skip Phase 1 (no exploration): Bypassing data exploration costs 0.121 BPB (+12.5%), the largest single-component degradation. Without prior analysis, the Strategist wastes early experiments on configurations that Phase 1 would have flagged as suboptimal (e.g., not discovering that including the query prefix improves BPB, or that byte-level tokenization enables direct BPB computation).

No playbook: With the playbook disabled, the system achieves 0.9941, 2.5% worse than the full system. Workers without playbook context must rediscover lessons (e.g., "disable torch.compile", "use byte tokenization") that earlier experiments had already established, wasting budget on known-bad configurations.

Phase 3 variance. To assess how much inter-model difference reflects genuine model quality versus search stochasticity, we ran GPT-5.2's Phase 3 five times with identical inputs

Variant	Removed	Best val_bpb	Δ vs. full
Full AlphaLab	(nothing)	0.9697	—
Skip Phase 1	Exploration	1.0908	+0.121 (+12.5%)
No playbook	Knowledge acc.	0.9941	+0.024 (+2.5%)
Variance (5 runs)	(nothing)	0.994 ± 0.025	—

Table 2: **Ablation results on LLM speedrun** (GPT-5.2, 50-experiment budget). Each ablation removes a single pipeline component while holding everything else constant (same hardware, shared Phase 2 harness). "Skip Phase 1" removes the data exploration phase where the system researches prior work on arXiv and analyzes the dataset before experimenting. "No playbook" removes the persistent knowledge document that accumulates lessons across experiments; without it, the system cannot remember what worked or failed in earlier experiments and must rediscover these lessons from scratch. The "Variance" row reports the mean ± standard deviation of the best val_bpb across five runs of the full system.

(same harness, learnings, data), varying only the stochastic LLM generation, the primary run plus four additional replications. The five runs yield val_bpb of 0.964, 0.970, 1.006, 1.011, and 1.020, a spread of 0.056 BPB from identical inputs, comparable in magnitude to the gap between the greedy baseline (1.020) and the full system's best (0.964). All five runs match or beat the greedy loop baseline (1.020), suggesting that AlphaLab's advantage over naïve iteration is robust to search stochasticity.

3.2 CUDA kernel optimization

Task. We evaluate on KernelBench (Ouyang et al., 2025), extended by Sakana AI's CUDA Engineer benchmark (Lange et al., 2025a). The task: given a PyTorch operator, write an optimized CUDA kernel that is both *correct* (matches the reference output) and *fast* (exceeds `torch.compile` performance). We target Level 1 (100 single-operator problems) and Level 2 (100 fusion patterns). We define fast_p as the fraction of correct kernels achieving $> p\times$ speedup over `torch.compile`; fast_1 denotes correct *and* faster than compiled PyTorch.

Results. Table 3 presents results at three levels of comparison. On the 54 tasks where both models produced correct kernels (same tasks, hardware, baseline), GPT-5.2 leads with 5.17× mean speedup vs Opus's 4.63×. The table also places AlphaLab in the context of prior work, though external comparisons are approximate due to differences in hardware and baseline. AlphaLab did not attempt all 200 KernelBench tasks; each campaign was budget-capped at 50 experiments for consistency across domains. Per-level breakdowns and per-kernel comparisons with Sakana AI are in the appendix.

3.3 Traffic forecasting

Task. Hourly road occupancy forecasting for 862 San Francisco Bay Area freeway sensors (Lai et al., 2018), predicting 24 hours ahead. Occupancy values lie in $[0, 1]$ with strong daily and weekly seasonality. Primary metric: RMSE (minimize). Baseline: Seasonal Naïve(168) (repeat the value from one week ago; see Table 4), RMSE ≈ 0.0287.

Results. Table 4 summarizes results. Opus 4.6 achieves the best RMSE of **0.02142** (−25%) via TFT (Lim et al., 2021) (an attention-based multi-horizon forecasting architecture with recurrence); GPT-5.2 reaches 0.02204 (−23%) via iTransformer (Liu et al., 2024) (a Transformer variant treating each sensor as a token). The two models searched differently: Opus converged entirely on TFT variants, while GPT-5.2 explored iTransformer, PatchTST (Nie et al., 2023), TSMixer (Chen et al., 2023), and N-HiTS (Challu et al., 2023). The table also reports literature RMSE values alongside AlphaLab's own reimplementations; AlphaLab's tuned versions consistently outperform literature defaults, suggesting the gap reflects hyperparameter tuning rather than architectural novelty. Both LLM baselines beat the seasonal naïve but remain substantially worse than AlphaLab. Notably, the greedy loop

System	Model	Correct	Mean spd.	fast_1
Direct comparison (54 tasks, both models correct, H100, vs `torch.compile`*)*				
AlphaLab	GPT-5.2	54	**5.17×**	**91%**
AlphaLab	Opus 4.6	53	4.63×	70%
Full run (H100, vs `torch.compile`*)*				
AlphaLab	GPT-5.2	110/119	4.40×	83%
AlphaLab	Opus 4.6	76/87	4.00×	70%
External baselines (different hardware and/or baseline – approximate comparison)				
AlphaLab (H100, vs `native`)	GPT-5.2	110/119	3.47×	75%
AlphaLab (H100, vs `native`)	Opus 4.6	76/87	3.27×	87%
Sakana AI (H100, vs `native`)	Ensemble	190/200	1.49×	—
KernelBench (L40S, vs `native`)	R1 (10 calls)	—	—	43/72%
KernelBench (L40S, vs `native`)	R1 (1-shot)	—	—	12/36%

Table 3: **CUDA kernel optimization results.** Task: write a CUDA kernel matching PyTorch's output but faster, on KernelBench (Ouyang et al., 2025) (100 single-op + 100 fusion tasks). *Top:* head-to-head on 54 tasks where both models produced correct kernels – same tasks, hardware, baseline. *Middle:* each model's full campaign (budget-capped at 50 experiments, so <200 tasks attempted). *Bottom:* AlphaLab re-reported against `torch.native` to enable comparison with external baselines. Sakana AI (Lange et al., 2025b): 5-model ensemble, excludes contaminated tasks. KernelBench: L40S GPU, fast_1 as L1/L2%. Cross-system comparison is approximate due to hardware and baseline differences.

Model	Lit. RMSE	AlphaLab RMSE	Cost
AlphaLab + Opus 4.6 (TFT, dropout 0.3)	—	**0.02142**	∼\$200
AlphaLab + GPT-5.2 (iTransformer ctx336)	—	0.02204	∼\$180
DeepAR (Salinas et al., 2019)	0.0249	0.0251	—
PatchTST (Nie et al., 2023)	0.0311	0.0226	—
TiDE (Das et al., 2023)	0.0281	0.0230	—
DLinear (Zeng et al., 2023)	0.0335	0.0234	—
SeasonalNaive (168)	0.0287	0.0287	—
Seasonal Average	0.0341	—	—
Weighted Ensemble	0.0346	—	—
Theta (Assimakopoulos & Nikolopoulos, 2000)	0.0370	—	—
ETS (Hyndman & Athanasopoulos, 2021)	0.0404	—	—
Single-shot (GPT-5.2)	—	0.02686	<\$1
Greedy loop (GPT-5.2)	—	0.02779	∼\$50

Table 4: **Traffic forecasting results** (RMSE, lower is better). Task: predict hourly road occupancy 24h ahead for 862 freeway sensors. Seasonal Naïve(168) repeats the value from one week ago (0.0287). "Lit. RMSE" = published result using each paper's protocol; "AlphaLab RMSE" = best result when AlphaLab independently discovered and tuned that model family on its own harness. Discrepancies reflect tuning and protocol differences. Single-shot = one LLM call; greedy loop = 50 sequential propose-then-train iterations. The greedy loop's worse-than-single-shot result reflects path dependence: its sequential propose-and-revert structure can converge to a local optimum that a single lucky call avoids.

performs slightly worse than the single-shot baseline on this domain, likely due to path dependence: the greedy loop's sequential propose-and-revert structure is constrained by its starting point and can converge to a local optimum, whereas the single-shot call happened to produce a stronger initial architecture. Detailed top-K tables are in the appendix.

3.4 What the system discovered

The playbook is perhaps the most interesting artifact the system produces. It starts empty; by the end of a campaign it contains domain-specific methodology that did not exist

anywhere in AlphaLab's prompts or code at launch, written by the Strategist, experiment by experiment, from results. We highlight representative excerpts (verbatim, unedited):

CUDA kernels (GPT-5.2): *"Diagonal matrix ops: $diag(A) \cdot B$ is just elementwise multiply of the diagonal vector with each row of B. Yields 10–68× over PyTorch's full matmul. Warp-shuffle reductions yield 73–75× on sum operations. Do not attempt convolution kernels: cuDNN is too well-optimized; handwritten runs at 0.05–0.73×."*

LLM pretraining (Opus 4.6): *"Wider-shallower architectures (10–12 layers, 672–752-dim) outperform deeper-narrower at this parameter budget. QK-norm stabilizes training at high learning rates."* GPT-5.2's playbook emphasized different findings: *"Disable torch.compile: compilation overhead is significant under a 20-min budget. GQA reduces memory and enables larger batch sizes."*

Traffic (GPT-5.2): *"Per-horizon affine calibration on top of iTransformer predictions is the single most impactful post-processing step: adding it consistently pushes RMSE below 0.023 where the raw model sits above it."*

The "do not attempt" entries are as valuable as positive findings, preventing budget waste on disproven approaches. Additional playbook excerpts are in the appendix.

3.5 Discussion

GPT-5.2 leads on CUDA kernels while Opus leads on LLM pretraining and narrowly on traffic; neither dominates uniformly (Table 6, Appendix). The models discover qualitatively different solutions in every domain, reinforcing the case for multi-model campaigns. Even within the same model family, Opus and Sonnet discover qualitatively different architectures on LLM pretraining (Section 3.1). On the other end of the capability spectrum, GPT-5.1-mini failed to produce any trustworthy results on LLM pretraining: it could not maintain the engineering rigor needed to correctly implement metrics and debug environment failures (Appendix H), suggesting a minimum model capability threshold for autonomous research.

4 Related work

The AI Scientist (Lu et al., 2024; Yamada et al., 2025), AIDE (Jiang et al., 2025), and Agent Laboratory (Schmidgall et al., 2025) demonstrated autonomous research with varying degrees of scope; multi-agent frameworks such as MetaGPT (Hong et al., 2023), ChatDev (Qian et al., 2024), and AutoGen (Wu et al., 2023) introduced role-based decomposition for software tasks. AlphaLab's roles target research campaigns rather than software projects, and its playbook provides cross-experiment memory absent from these systems. Voyager (Wang et al., 2023) is the closest analog to the playbook (a skill library accumulated during play); Reflexion (Shinn et al., 2023) maintains verbal memory within a single task. FunSearch (Romera-Paredes et al., 2024), AlphaCode (Li et al., 2022), GPU Kernel Scientist (Andrews & Witteveen, 2025), and Sakana AI's CUDA Engineer (Lange et al., 2025a) apply LLM-guided search to specific domains. AutoML systems (Feurer et al., 2015; Erickson et al., 2020; Akiba et al., 2019) search predefined configuration spaces; AlphaLab constructs its own evaluation methodology and reasons about why experiments succeed or fail. The broader trend toward harness engineering (PoetIQ, 2025) motivates AlphaLab's design.

5 Conclusion

We presented AlphaLab, an autonomous multi-agent system that automates the experimental research cycle across qualitatively different domains at a cost of $150–200 per campaign. The central finding is that different frontier models discover different solutions in every domain, suggesting multi-model campaigns provide complementary search coverage. Key limitations include premature playbook convergence (explicit diversity budgets are needed), environmental fragility (PyTorch API changes caused 38% failure rates), single-run comparisons, and the lack of sandboxing for LLM-generated code. We believe research is entering a phase where human–AI teams outperform either alone, and we hope open-sourcing AlphaLab accelerates progress toward that vision.

Acknowledgements

We are especially thankful to CoreWeave, whose purpose-built AI cloud platform powered our experiments.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2623–2631, 2019.

Martin Andrews and Sam Witteveen. GPU kernel scientist: An LLM-driven framework for iterative kernel optimization. *arXiv preprint arXiv:2506.20807*, 2025.

Anthropic. Claude code: An agentic coding tool. https://www.anthropic.com/products/claude-code, 2025.

V. Assimakopoulos and K. Nikolopoulos. The theta model: a decomposition approach to forecasting. *International Journal of Forecasting*, 16(4):521–530, 2000. ISSN 0169-2070. doi: https://doi.org/10.1016/S0169-2070(00)00066-2. URL https://www.sciencedirect.com/science/article/pii/S0169207000000662. The M3- Competition.

Cristian Challu, Kin G. Olivares, Boris N. Oreshkin, Federico Garza Ramirez, Max Mergenthaler Canseco, and Artur Dubrawski. Nhits: Neural hierarchical interpolation for time series forecasting. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(6): 6989–6997, Jun. 2023. doi: 10.1609/aaai.v37i6.25854. URL https://ojs.aaai.org/index.php/AAAI/article/view/25854.

Si-An Chen, Chun-Liang Li, Sercan O Arik, Nathanael Christian Yoder, and Tomas Pfister. TSMixer: An all-MLP architecture for time series forecast-ing. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=wbpxTuXgm0.

François Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Abhimanyu Das, Weihao Kong, Andrew Leach, Shaan K Mathur, Rajat Sen, and Rose Yu. Long-term forecasting with tiDE: Time-series dense encoder. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=pCbC3aQB5W.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. AutoGluon-Tabular: Robust and accurate AutoML for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

R.J. Hyndman and G. Athanasopoulos. *Forecasting: Principles and practice*. OTexts, 2021. ISBN 978-0987507136.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 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? In *International Conference on Learning Representations*, 2024.

Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, et al. modded-nanogpt: Speedrunning the NanoGPT baseline. https://github.com/KellerJordan/modded-nanogpt, 2024.

Andrej Karpathy. Autoresearch. https://github.com/karpathy/autoresearch, 2026. Open-source greedy experiment loop for automated AI research.

Garry Kasparov and Mig Greengard. *Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins*. PublicAffairs, 2017.

Guokun Lai, Wei-Cheng Chang, Yiming Yang, and Hanxiao Liu. Modeling long- and short-term temporal patterns with deep neural networks. In *Proceedings of the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 95–104, 2018.

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. The AI CUDA engineer: Agentic CUDA kernel discovery, optimization and composition. *Sakana AI Technical Report*, 2025a.

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025b.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Bryan Lim, Sercan Ö. Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021.

Yong Liu, Tengge Hu, Haoran Zhang, Haixu Wu, Shiyu Wang, Lintao Ma, and Mingsheng Long. itransformer: Inverted transformers are effective for time series forecasting. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=JePfAI8fah.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Yuqi Nie, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. In *International Conference on Learning Representations*, 2023.

OpenAI. Introducing codex — a cloud-based software engineering agent. https://openai.com/index/introducing-codex/, 2025.

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

PleIAs and Pierre-Carl Langlais. SYNTH: A generalist open synthetic dataset for LLM pretraining. https://huggingface.co/datasets/PleIAs/SYNTH, 2025.

PoetIQ. Poetiq ARC-AGI solver. https://poetiq.ai/posts/arcagi_verified/, 2025.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2024.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 2019. ISSN 0169-2070. URL http://www.sciencedirect.com/science/article/pii/S0169207019301888.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using LLM agents as research assistants. *arXiv preprint arXiv:2501.04227*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*, 2025.

Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. Are transformers effective for time series forecasting? *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(9):11121–11128, Jun. 2023. doi: 10.1609/aaai.v37i9.26317. URL https://ojs.aaai.org/index.php/AAAI/article/view/26317.

LLM Disclosure Statement

In accordance with COLM policy, we disclose the following uses of LLMs in the preparation of this work.

System under study. ALPHALAB is an LLM-powered autonomous research system; all experimental code, evaluation frameworks, domain adapters, and playbook content reported in this paper were generated by the frontier LLMs under evaluation (GPT-5.2, Claude Opus 4.6, Claude Sonnet 4.6) as part of the system's normal operation.

System development. The ALPHALAB codebase was built with the assistance of agentic coding tools (Claude Code). The core idea and system architecture were human-conceived; detailed design decisions, implementation, and debugging were carried out iteratively with LLM assistance.

Figures. The main pipeline figure (Figure 1) was generated using Nano Banana Pro (Gemini 3 Pro Image) from Google.

Analysis and writing. Experimental results were analyzed and interpreted by the authors. The manuscript was outlined and rough-drafted by the authors, then iteratively refined with LLM assistance for clarity and presentation. All scientific claims and conclusions are the sole responsibility of the authors.

Appendix

Appendix A **System details** – Full tool set, domain adapter file structure, detailed walkthroughs of all four phases with prompt examples, and Supervisor intervention case studies.

Appendix B **Extended experimental results** – Cross-domain summary, per-model top-K and full experiment tables for LLM speedrun and traffic, per-level CUDA breakdown, per-kernel Sakana AI comparison, and convergence curves.

Appendix C **Playbook excerpts and dynamics** – Verbatim playbook excerpts from all domains, growth trajectory analysis, self-correction events, and cross-model comparison of playbook styles.

Appendix D **User interface** – Description of the real-time web dashboard (Kanban board, leaderboard, file viewer, conversation stream, human-in-the-loop chat).

Appendix E **Additional experiments** – Financial time series forecasting (exchange rates) with auto-generated adapter.

Appendix F **Cost and token breakdown** – Token usage per campaign, phase-level distribution, cost per experiment, and model cost comparison.

Appendix G **Failure analysis** – Three-tier failure taxonomy (programmatic, evaluation, strategic) with per-domain breakdown and error rates.

Appendix H **Reproducibility details** – Model versions, API parameters, hardware specs, repository structure, and variance experiment protocol.

A System details

A.1 Full tool set

Table 5: Full tool set available to all ALPHALAB agents.

Tool	Description
`shell_exec`	Full shell access: execute arbitrary commands, write and run code, install packages, manage files.
`read_file`	Read any file in the workspace.
`grep_file`	Search file contents by pattern.
`web_search`	Search the internet for papers, documentation, and best practices.
`view_image`	View plots and visualizations the agent has generated.
`spawn_agent`	Launch a sub-agent: a new instance of $\mathcal{M}$ with its own context and full tool access.
`read_board`	Read the current experiment leaderboard (Phase 3).
`update_playbook`	Append entries to the persistent playbook (Phase 3).
`propose_experiment`	Submit a new experiment specification (Phase 3, Strategist only).
`report_to_user`	Send a status report to the human operator.

Across a representative LLM speedrun campaign (GPT-5.2, 50 experiments, 2,892 API calls), tool usage breaks down as follows: `shell_exec` accounts for ∼49.5% of all tool calls, reflecting the code-heavy nature of the work; `read_file` for ∼21.8% (reading experiment outputs, debriefs, and framework code); `grep_file` for ∼12.3% (searching for errors, patterns, and configurations); `web_search` for ∼8.1% (concentrated in Phase 1); and the remaining ∼8.3% split across `propose_experiment`, `update_playbook`, `read_board`, `view_image`, and `report_to_user`.

Tool usage varies substantially by phase. Phase 1 (exploration) is dominated by `shell_exec` (∼60%) and `web_search` (∼20%), as the agent writes analysis scripts and researches the domain. Phase 2 (evaluation construction) shifts toward `read_file` (∼35%) and `shell_exec` (∼45%), with the Critic reading code and the Tester running test suites. Phase 3 (experimentation) has the most diverse tool usage: the Strategist uses `read_board`, `propose_experiment`, and `update_playbook` almost exclusively, while Workers are heavily `shell_exec`-dominated (∼70%).

A.2 Domain adapter file structure

The 11 adapter files are:

1. `manifest.yaml` – metric definitions, direction (min/max), experiment structure, entry points
2. `domain_knowledge.md` – injected into every agent's context; written by Phase 0 after examining data
3. `phase1_explorer.md` – prompt for the Explorer agent
4. `phase2_builder.md` – prompt for the Builder
5. `phase2_critic.md` – prompt for the Critic
6. `phase2_tester.md` – prompt for the Tester
7. `phase3_strategist.md` – prompt for the Strategist
8. `phase3_worker.md` – prompt for the Worker (implement/analyze/fix)
9. `phase3_supervisor.md` – prompt for the Supervisor
10. `phase0_customizer.md` – prompt for the adapter customization agent
11. `phase0_generator.md` – prompt for full adapter generation from scratch

Complete adapter files for all domains are available in the code repository.

While all adapters share the same 11-file structure, their contents differ substantially to reflect each domain's evaluation methodology, optimization landscape, and failure modes.

Metric and framework differences. The time series adapter defines RMSE as its primary metric (minimize), uses a walk-forward backtesting harness with embargo periods, and requires each experiment to implement a `Strategy` subclass with `fit()`/`predict()` methods. The CUDA kernel adapter defines speedup over `torch.compile` as its primary metric (maximize), uses a benchmark harness that checks correctness via `torch.allclose` and measures median runtime over 100 runs, and requires each experiment to produce a `kernel.cu` file. The LLM speedrun adapter defines `val_bpb` (minimize), uses a training harness that enforces a 20-minute wall-clock budget and <100M parameter constraint, and requires a modified `train.py`.

Domain knowledge examples. The `domain_knowledge.md` file is the highest-value adapter artifact because it is injected into every agent's context for the entire campaign. After Phase 0 customization, these files contain task-specific guidance grounded in the actual data. Two representative excerpts:

LLM speedrun `domain_knowledge.md` (excerpt, after Phase 0 customization):

> *"Primary Metric:* `val_bpb` *(Validation Bits-Per-Byte). Definition: bpb = loss_nats ×* $\log_2(e)$ */ bytes_per_token. [...] Dataset: PleIAs SYNTH, 500 parquet shards (∼220 GB). Languages: en ∼80.7%; exercises: memorization ∼90.6%. [...] Architecture sweet spots under 100M: 6–12 layers, d_model 512–768, RMSNorm + RoPE + SwiGLU. Sampling is mandatory: reading the full 220 GB corpus is not feasible within 20 minutes."*

Traffic forecasting `domain_knowledge.md` (excerpt):

> *"862 freeway sensors, hourly occupancy* $[0, 0.724]$*, prediction_length=24. Test set: 6034 entries = 862 sensors × 7 rolling windows. Strong daily seasonality (peak/trough ratio ≈9.6×); weekly seasonality (weekday mean ≈1.42× weekend). Sensors are correlated: mean pairwise correlation ≈0.56. Context length ≥192h to access weekly lag."*

The same file structure thus encodes fundamentally different domain knowledge (from tokenizer semantics and parameter budgeting (LLM speedrun) to seasonal decomposition and cross-sensor correlation structure (traffic)), while downstream agents consume them through an identical interface.

A.3 Phase 0: Adapter resolution – detailed workflow

Phase 0 resolves the domain adapter through one of three paths, selected automatically based on the configuration:

Path 1: Resume. If {`workspace`}`/adapter/manifest.json` exists, the adapter is loaded directly. This enables resuming campaigns without re-running adapter resolution.

Path 2: Customize built-in. When the `domain` field matches a built-in adapter name (`time_series`, `cuda_kernel`, `llm_speedrun`), the template is copied to the workspace and a customization agent examines the actual dataset. The agent has access to `shell_exec`, `read_file`, `read_adapter`, and `patch_adapter_file`. It reads a sample of the data, runs exploratory queries (e.g., checking column names, value distributions, data size), and patches the adapter files (especially `domain_knowledge.md`) to reflect the specific task.

For example, on the LLM speedrun task, the customization agent sampled 6 parquet shards, discovered the byte-level tokenization scheme (vocab=256), measured the language distribution (∼80.7% English), and computed text length statistics. It then wrote these findings into `domain_knowledge.md`, including the exact BPB formula, sampling guidance ("reading the full 220 GB corpus is not feasible within 20 minutes"), and architecture recommendations grounded in the parameter constraint.

Path 3: Generate from scratch. When the `domain` field contains free-text (e.g., "forecast exchange rates"), a generation agent creates all 11 adapter files from scratch. The agent has

access to `write_adapter_file` and `read_reference_adapter` (to examine built-in adapters as format references). It examines the dataset, searches the web for relevant papers and benchmarks, and produces the complete adapter: manifest, domain knowledge, and all 9 prompt templates. This path is used for novel domains not covered by built-in adapters.

A.4 Phase 1: Data exploration – detailed workflow

We illustrate Phase 1 with the LLM speedrun campaign (GPT-5.2). The Explorer begins by creating a structured `plan.md` checklist, then works through it autonomously:

```
# Plan | PleIAs SYNTH LLM pretraining quality speedrun
- [x] 0. Workspace + environment reconnaissance (GPU/CPU, libs)
- [x] 1. Dataset recon: schema, language distribution, lengths
- [x] 2. Build fast dataloader/tokenizer pipeline
- [x] 3. Define evaluation metric: val_bpb computation
- [x] 4. Implement baseline model/training loop
- [x] 5. Parameter-budgeted model family search (<100M)
- [x] 6. Optimization search: AdamW vs alternatives, schedules
- [x] 7. Architecture tweaks: GPT-2 vs LLaMA-style
- [x] 8. Data curriculum: include query? filter languages?
- [x] 9. Throughput tuning: batch size, compilation
- [x] 10. Run controlled ablations; keep scoreboard
- [x] 11. Assemble deliverables
```

The Explorer wrote and executed 7 Python scripts in `scripts/` (dataset reconnaissance, tokenizer benchmarking, baseline training, architecture search), generated distribution plots in `plots/` (language distribution, text length histograms), and produced three deliverables in `data_report/`: `schema.md`, `statistics.md`, and `findings.md`.

Key discoveries recorded in `learnings.md` (verbatim excerpts):

> *"Byte-level tokenization (UTF-8 bytes) enables direct bpb computation (1 token == 1 byte). Including query as a prefix (Q: ... A: ...) materially improved short-run val_bpb (∼2.46 vs ∼3.00 at ∼90s). seq_len=1024 performed similarly to 512 in short runs; prefer 1024 for longer-budget training."*

Traffic forecasting (GPT-5.2). The traffic Explorer produced a 20-item plan organized into 7 sections (setup, schema, temporal structure, statistical profiling, seasonality/autocorrelation, cross-sensor dependency, and feature relationships):

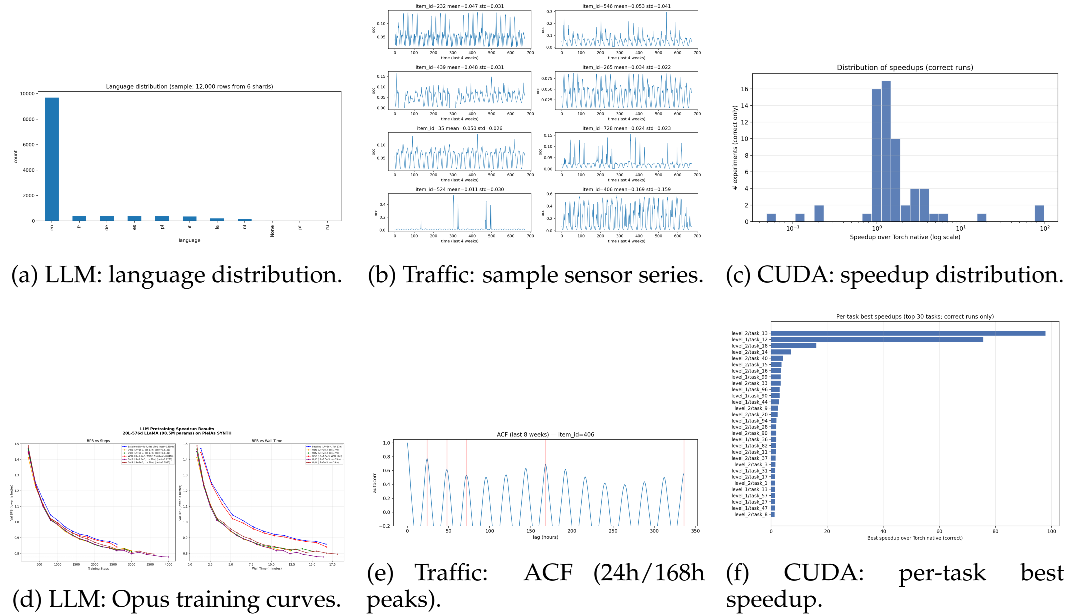
```
# Plan | Traffic (GluonTS) Dataset Deep Dive
## Data loading & schema exploration
- [x] Parse train/test JSON, confirm 862 sensors, ~14,036h
- [x] Validate hourly frequency, prediction_length=24
## Target dynamics: seasonality, autocorrelation, stationarity
- [x] ACF/PACF globally and per-cluster (sample sensors)
- [x] Spectral analysis / periodogram (24h, 168h expected)
- [x] Decomposition (STL) on representative sensors
## Cross-sensor dependency & covariance structure
- [x] PCA on standardized series
- [x] Clustering sensors by daily pattern embeddings
```

Key discoveries: 862 sensors with strong dual seasonality (lag-24 ACF=0.87, lag-168 ACF=0.95; weekly periodicity stronger than daily); cross-sensor mean correlation 0.57; PCA PC1 explains 55% of variance; baseline RMSE of seasonal naive(168) $\approx$ 0.0287; context lengths $\geq$192h needed to capture weekly patterns.

CUDA kernels (GPT-5.2). The CUDA Explorer's plan focused on benchmarking infrastructure rather than data analysis: inventorying all 200 tasks (100 Level-1 single-op, 100 Level-2 fused-op), profiling tensor sizes (10^3 to $> 10^8$ elements), and establishing the optimization priority hierarchy. Key learnings recorded:

> *"200 tasks total: 100 Level 1 (single-op) + 100 Level 2 (fused ops). Tensor sizes range 10^3 to $> 10^8$ elements. Level 1: mixed ops (conv, elementwise, reduction, matmul). Level 2: conv-heavy and matmul-heavy fused patterns. CUDA compilation takes minutes; subprocess evaluation adds overhead. Memory hierarchy optimization (coalescing → tiling → vectorization) should be the first priority for every kernel."*

Phase 1 typically runs for 30–90 minutes and makes 200–400 tool calls. Figure 2 shows representative plots generated autonomously by the Explorer in each domain.



(a) LLM: language distribution. (b) Traffic: sample sensor series. (c) CUDA: speedup distribution.

(d) LLM: Opus training curves. (e) Traffic: ACF (24h/168h peaks). (f) CUDA: per-task best speedup.

Figure 2: Representative plots generated autonomously by ALPHALAB's Explorer (Phase 1) and Reporter (Phase 3) agents across three domains. These plots are generated by LLM-written Python scripts and viewed by the agent via the `view_image` tool to inform subsequent analysis. Font sizes reflect the system's raw output; we reproduce them unmodified to preserve fidelity.

A.5 Phase 2: Evaluation construction – detailed workflow

We walk through Phase 2 for the traffic forecasting domain (GPT-5.2).

Builder (Round 1). The Builder receives Phase 1 learnings (seasonality structure, baseline performance, evaluation protocol) and produces the evaluation framework in `harness/`:

- `runner.py` – orchestrator: loads GluonTS data, runs walk-forward evaluation over the 7 rolling windows, computes pooled metrics, saves `results/metrics.json`
- `metrics.py` – RMSE, MAE, MASE computation; assertion that `n_instances==6034`
- `data_prep.py` – GluonTS JSON parsing, context/target splitting, calendar feature generation
- `config.py` – experiment configuration (context length, model type, hyperparameters)
- `baseline_train.py` – seasonal naive baselines for sanity checking

Critic review. A fresh Critic agent (no shared context with the Builder) audits every file for data leakage, lookahead bias, and metric errors. The Critic's review identified two issues: (1) the `data_prep.py` context window extraction could include the target period if `context_length` exceeded the available history (edge case for the first rolling window), and (2) the MASE denominator used the seasonal period m=168 but the standard benchmark uses m=24. Both were flagged as "NEEDS FIXES" with specific line references.

Builder (Round 2). The Builder patched the context window clipping logic (adding a `min(context_length, available_history)` guard) and corrected the MASE seasonal period. The Critic re-reviewed and issued a "PASS" verdict.

Tester. The Tester wrote 4 test modules in `harness/tests/`: `test_data_prep.py` (context/target alignment, no overlap), `test_metrics.py` (hand-calculated RMSE/MAE on small arrays), `test_baselines.py` (seasonal naive produces expected outputs), and `test_integration.py` (full pipeline on a 10-sensor subset). All 23 tests passed on the first run. Total Phase 2 duration: ∼40 minutes, 3 agent turns (Builder–Critic–Builder–Critic–Tester).

A.6 Phase 3: Experimentation – detailed workflow

We illustrate Phase 3 using the traffic forecasting campaign (GPT-5.2, 50 experiments, ∼3,400 total API calls). The key idea is that a pure-Python Dispatcher orchestrates the entire process without making any LLM calls itself: it is a control loop that manages state, while all intelligence comes from the Strategist and Worker agents it invokes.

Dispatcher main loop. Every ∼30 seconds, the Dispatcher runs a tick: it checks which GPU jobs have finished, transitions their state on a SQLite kanban board, and assigns free Workers to the highest-priority pending task (fixes first, then analysis, then new implementations). Every 5 analyzed experiments it invokes the Strategist for new proposals; every 15 it triggers a milestone report. The Dispatcher never reasons about what experiments to run; it only manages the queue and resource allocation.

Strategist turn. We illustrate with turn 3 (after 15 experiments). At this point the Strategist can see the full picture: the current leaderboard (best RMSE 0.0230 from `tsmixer_ctx672`), debriefs from recent experiments (including a failed TCN variant at 0.050 and a promising iTransformer at 0.0226), the accumulated playbook, and Phase 1 learnings. Based on this, it makes three types of decisions: (1) it *proposes* 3 new experiments that build on what worked (iTransformer with hour-of-week features, ridge stacking, per-horizon calibration); (2) it *cancels* 4 queued experiments that are now unlikely to help (TCN variants, slow N-HiTS); and (3) it *updates the playbook* with the lesson that "iTransformer + calendar features is the strongest family; TCN is not competitive." This is how the system learns across experiments: the playbook entry will steer all future Workers and Strategist turns away from TCN and toward iTransformer refinements.

Worker implementation cycle. A Worker assigned to implement `itransformer_ctx336_hourofweek_revin` proceeds as follows: (1) reads the experiment specification from the Strategist; (2) reads the playbook for guardrails (e.g., "assert n_instances==6034", conservative batch sizes); (3) reads the harness code (`runner.py`, `data_prep.py`); (4) writes `experiments/itransformer_ctx336_hourofweek_revin/train.py` implementing the model; (5) writes `run_experiment.py` (entry point that calls the harness); (6) runs a smoke test (`python run_experiment.py --smoke`, <60s, 10 instances); (7) runs the reality check tool to validate on real data; (8) marks the experiment as `checked`. The Dispatcher then queues it for GPU execution.

Worker analysis cycle. After GPU execution completes, an analysis Worker reads `results/metrics.json`, extracts metrics, and writes a structured debrief. These debriefs are among the most valuable artifacts the system produces: they capture not just what happened but *why*, in a format that feeds directly into the Strategist's next round of proposals. We show representative excerpts from each domain.

CUDA kernel debrief (`l1_t12`, 73× speedup over `torch.compile`):

> "***Task:*** `torch.diag(A) @ B`, *algebraically equivalent to row-wise scaling:* `Out[i,j] = A[i] * B[i,j]`. ***What was implemented:*** *Fused row-wise multiply – flattened 1D traversal, 256 threads/block, grid-stride loop with unroll factor 4. No atomics, no race-prone reductions.* ***Why this is fast:*** *PyTorch's* `diag(A) @ B` *materializes a diagonal matrix*

and/or calls GEMM-like machinery. The custom kernel avoids creating `diag(A)` *entirely, performs a single coalesced read of B and a single write per element.* ***Follow-up:*** *Vectorized float4 loads could improve memory throughput. Given the already-dominant win and breadth mandate, prioritize moving to new tasks."*

LLM speedrun debrief (Opus, `shallow_10l_752d`, val_bpb 0.7578, campaign best):

> "***Result:*** *val_bpb = 0.7578, new best, beats previous 0.7624 by 0.6%. 10L×752d LLaMA-style, 98.2M params, 163K tok/s, 194M tokens in 20 min.* ***Key finding:*** *Throughput gain from fewer layers (10L vs 12L: ∼3% faster per step) compounds over 20 minutes. Near-saturation at this parameter count: last 1,200 steps contributed only ∼0.004 BPB.* ***Suggestions:*** *Try 10L×768d to use full 100M budget; larger batch (only using 27/80 GB VRAM); more training shards for fresh data."*

Traffic debrief (GPT-5.2, `itransformer_ctx336_timefeat_staticcat`, RMSE 0.02204, campaign best):

> "***Configuration:*** *iTransformer, context=336h (2 weeks), calendar covariates (hour_of_day, day_of_week, is_weekend), sensor ID embedding, ReVIN. d_model=256, 4 encoder layers, 8 heads, AMP training.* ***Result:*** *RMSE 0.02204 over 6,034 instances, best in workspace.* ***Per-horizon profile:*** *Lowest errors at 3–9h ahead; mid-horizon bump at ∼12h; worst at 20–22h (next-day rush-hour timing difficulty).* ***What worked:*** *Shorter context (336h vs 672h) regularizes attention while still capturing weekly structure. Calendar features + sensor embeddings highly beneficial."*

Generated code. The code Workers produce is substantial, typically 200–400 lines per experiment. As a representative example, the CUDA kernel for the 73× speedup above:

```
// Task: level_1/task_12 | torch.diag(A) @ B
// Equivalent: out[i, j] = A[i] * B[i, j]
constexpr int kThreads = 256;
constexpr int kUnroll = 4;

template <typename scalar_t>
__global__ void diag_mul_rows_kernel(
    const scalar_t* __restrict__ A,
    const scalar_t* __restrict__ B,
    scalar_t* __restrict__ Out,
    int64_t N, int64_t M) {
  const int64_t tid = blockIdx.x * blockDim.x + threadIdx.x;
  const int64_t grid_stride = blockDim.x * gridDim.x;
  const int64_t total = N * M;
  for (int64_t base = tid; base < total;
       base += grid_stride * kUnroll) {
    #pragma unroll
    for (int k = 0; k < kUnroll; ++k) {
      int64_t idx = base + (int64_t)k * grid_stride;
      if (idx < total) {
        int64_t row = idx / M;
        Out[idx] = A[row] * B[idx];
      }
    }
  }
}
```

This kernel (which the system designed, implemented, tested for correctness, and benchmarked) replaces PyTorch's `diag(A) @ B` with a simple fused multiply, eliminating the diagonal matrix materialization entirely.

The traffic forecasting best result (iTransformer, RMSE 0.02204) implemented a full `Strategy` subclass with ReVIN normalization, calendar embeddings, and a Transformer encoder, approximately 350 lines total. The model architecture, forward pass, and Strategy interface:

```
@dataclass
```

```
class ITransformerConfig:
    context_length: int = 336   # 2 weeks of hourly data
    prediction_length: int = 24 # 1 day ahead
    d_model: int = 256
    n_heads: int = 8
    e_layers: int = 4
    dropout: float = 0.1
    ffn_dim: int = 512
    sensor_emb_dim: int = 16
    revin: bool = True          # instance normalization
    lr: float = 5e-4
    batch_size: int = 256
    epochs: int = 30
    amp: bool = True

class _ITransformerModel:
    def __init__(self, cfg, n_sensors=862):
        d_model = cfg.d_model
        # Embeddings for calendar features + sensor identity
        self.hour_emb = nn.Embedding(24, d_model // 8)
        self.dow_emb = nn.Embedding(7, d_model // 8)
        self.weekend_emb = nn.Embedding(2, d_model // 16)
        self.sensor_emb = nn.Embedding(n_sensors, 16)
        feat_dim = 1 + hour_dim + dow_dim + weekend_dim + 16
        self.in_proj = nn.Linear(feat_dim, d_model)
        enc_layer = nn.TransformerEncoderLayer(
            d_model=d_model, nhead=cfg.n_heads,
            dim_feedforward=cfg.ffn_dim,
            batch_first=True, norm_first=True)
        self.encoder = nn.TransformerEncoder(
            enc_layer, num_layers=cfg.e_layers)
        self.out = nn.Linear(d_model, 1)

    def forward(self, y_seq, hour, dow, weekend, sensor_id):
        B, T = y_seq.shape
        eh = self.hour_emb(hour)
        ed = self.dow_emb(dow)
        ew = self.weekend_emb(weekend)
        es = self.sensor_emb(sensor_id).unsqueeze(1).expand(
            B, T, -1)
        y_in = y_seq.unsqueeze(-1)
        feats = torch.cat([y_in, eh, ed, ew, es], dim=-1)
        x = self.in_proj(feats)
        h = self.encoder(x)
        return self.out(h).squeeze(-1)

class ITransformerStrategy(Strategy):
    """iTransformer-style encoder (global across sensors).
    Training uses random sliding windows from train split.
    Predict: provide true y for context, 0 for future;
    provide known future time features for all positions."""

    def fit(self, train_data):
        # Precompute time features per item
        # Train with AdamW, AMP, grad clipping
        # Random sliding windows of length L+H
        ...

    def predict(self, context: ForecastContext,
                prediction_length: int) -> np.ndarray:
        # ReVIN normalize from context only (no lookahead)
        y_ctx = context.target[-self.cfg.context_length:]
        mean, std = y_ctx.mean(), max(y_ctx.std(), 1e-5)
        y_norm = (y_ctx - mean) / std
        # Generate future time features from timestamps
```

```
        # Run model, denormalize, clip to [0, 1]
        ...
```

The LLM speedrun's best Opus experiment (val_bpb 0.7578) implemented a complete LLaMA-style transformer, approximately 250 lines. The full model architecture:

```
"""
Experiment: shallow_10l_752d (#33)
Continue the depth-vs-width exploration: if 12L beat 16L,
does 10L beat 12L?
Architecture: 10L-752d, d_ff=2256, 8 heads (head dim=94).

Parameter budget:
  Embedding: 32768 * 752 = 24,641,536 (tied)
  Per layer: ~7,351,040
  Total: ~98.15M (under 100M)
"""
class RMSNorm(nn.Module):
    def forward(self, x):
        return (x.float()
            * torch.rsqrt(x.float().pow(2).mean(-1, True)
            + self.eps)).to(x.dtype) * self.w

class RoPE(nn.Module):
    def __init__(self, d, mx=2048):
        inv = 1.0/(10000**(torch.arange(0,d,2).float()/d))
        t = torch.arange(mx).float()
        fr = torch.outer(t, inv)
        self.register_buffer("cos", fr.cos())
        self.register_buffer("sin", fr.sin())

class Attention(nn.Module):
    def forward(self, x):
        B, T, C = x.shape
        q, k, v = self.qkv(x).reshape(
            B, T, 3, self.nh, self.hd).unbind(2)
        q = apply_rope(q.transpose(1,2), self.rope)
        k = apply_rope(k.transpose(1,2), self.rope)
        y = F.scaled_dot_product_attention(
            q, k, v.transpose(1,2), is_causal=True)
        return self.out(y.transpose(1,2).reshape(B, T, C))

class SwiGLU(nn.Module):
    def forward(self, x):
        return self.down(F.silu(self.gate(x)) * self.up(x))

class Block(nn.Module):
    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        return x + self.ffn(self.norm2(x))

class GPT(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads,
                 n_layers, d_ff, max_seq=2048):
        self.embed = nn.Embedding(vocab_size, d_model)
        self.layers = nn.ModuleList(
            [Block(d_model, n_heads, d_ff, max_seq)
             for _ in range(n_layers)])
        self.final_norm = RMSNorm(d_model)
        self.lm_head = nn.Linear(vocab_size, d_model)
        self.lm_head.weight = self.embed.weight  # tied

    def forward(self, idx, targets=None):
        h = self.embed(idx)
        for layer in self.layers:
```

```
        h = layer(h)
    logits = self.lm_head(self.final_norm(h))
    loss = F.cross_entropy(
        logits.view(-1, logits.size(-1)),
        targets.view(-1)) if targets is not None \
        else None
    return logits, loss
```

Each of these code artifacts (the CUDA kernel, the traffic strategy class, the LLM training script) was written entirely by the Worker agent, with no human editing. The code quality is generally production-grade: proper error handling, deterministic seeding, mixed-precision training, and gradient clipping are standard across experiments.

Worker fix cycle. When an experiment fails (e.g., OOM on a large context length), a fix Worker reads the error log, diagnoses the issue (e.g., "batch_size=128 exceeds H100 memory for ctx672"), patches the code (reducing to batch_size=32 with gradient accumulation), and resubmits. Fix attempts are capped at k=2; after 2 failures the experiment is marked as permanently failed.

Milestone reports. Every 15 analyzed experiments, a Reporter agent generates a milestone report (`reports/milestone_NNN/overview.md`) summarizing: best results so far, convergence trajectory, flagged experiments (suspicious metrics, smoke-test-only results), and recommendations for the Strategist. These reports serve as an auditing mechanism: the traffic GPT-5.2 campaign produced 4 milestone reports over its 50-experiment run.

A.7 Prompt examples

We include representative prompt excerpts for each major agent role. Domain-specific adapters customize these templates, but the core structure is consistent.

Phase 1 Explorer ("Go Work" prompt).

```
You are Alpha Lab, a fully autonomous research agent. The user launches
you, gives you a dataset, and you go work. You do NOT stop to ask questions,
narrate plans, or wait for confirmation. You just work.
CRITICAL RULES:
1. PLAN FIRST: Create plan.md -- a detailed to-do list. Check off items
as you complete them.
2. DO NOT STOP: Chain tool calls continuously until plan.md is complete.
3. FILE EVERYTHING: scripts/ for Python analysis, plots/ for
visualizations, notes/ for per-topic findings, learnings.md for
accumulated knowledge (update after every finding), data_report/ for formal
deliverables.
4. CALL report_to_user WHEN DONE: This is the only way to return control.
```

Phase 3 Strategist (budget management).

```
Budget management:
- >20 remaining: Explore freely -- diverse architectures, features,
hyperparameters.
- 10--20 remaining: Focus on promising directions identified so far.
- 5--10 remaining: Only high-confidence refinements of top performers.
- <5 remaining: Extremely selective -- only strong evidence to beat current
best.
- 0 remaining: STOP proposing. Summarize findings, recommend next steps.
```

Phase 3 Worker (GPU safety rules).

```
Critical -- Avoiding SLURM Failures:
1. NEVER set torch.use_deterministic_algorithms(True) -- many CUDA ops
have no deterministic implementation, will crash on H100s.
2. Handle NaN/missing values in features: rolling windows produce NaN for
first N rows. .dropna() or .fillna(0) before DataLoader.
3. Use conservative batch sizes/context lengths: H100 has 80GB VRAM but
large models can OOM. Start with batch_size=64.
4. Import lightning not pytorch_lightning (modern package name).
5. Wrap main block in try/except and save partial results on failure.
```

Supervisor (diagnostic prompt). When triggered by error rate $> \tau$, the Supervisor receives recent failure logs and the current adapter, with instructions to: (1) identify the systemic root cause (not individual bugs); (2) propose a concrete patch to `domain_knowledge.md` that prevents recurrence; and (3) commit the change via `patch_adapter_file` with a descriptive reason for the git checkpoint.

A.8 Supervisor interventions

We document the major Supervisor interventions observed across campaigns.

Intervention 1: CUDA keyword argument mismatch. *Trigger*: Error rate exceeded 45% in the first 15 CUDA experiments. *Diagnosis*: Workers assumed positional argument signatures for the harness entry point, but the harness used keyword arguments. *Patch*: Added to `domain_knowledge.md`: "The harness calls `forward(**inputs)` with keyword arguments. Your kernel's forward function must accept keyword arguments matching the reference model's signature." *Result*: Error rate dropped from 45% to under 10% within 3 experiments.

Intervention 2: PyTorch 2.9.1 API breaking changes (electricity). *Trigger*: Error rate exceeded 55% in the GPT-5.2 electricity campaign. *Diagnosis*: Multiple PyTorch 2.9.1 API removals: `.total_mem` attribute removed from CUDA tensors, `ReduceLROnPlateau(verbose=)` parameter removed, tqdm + SIGTERM causing BrokenPipeError. *Patch*: Added three specific notes to `domain_knowledge.md` documenting the removed APIs and their replacements. *Result*: Subsequent experiments avoided these errors, though the overall failure rate remained high (55%) due to other OOM and import issues.

Intervention 3: LLM speedrun harness indentation error. *Trigger*: Multiple consecutive SLURM failures in late GPT-5.2 LLM speedrun experiments (#43–#50). *Diagnosis*: A prior Worker's fix introduced an indentation error in `harness/runner.py`, causing all subsequent experiments to fail at launch. *Patch*: The Supervisor identified the specific line, patched the indentation, and added a note to domain knowledge: "Always verify `harness/runner.py` imports cleanly before submitting." *Result*: The fix resolved the hard failure, though those experiments had already exhausted the budget.

Intervention 4: Traffic data path resolution. *Trigger*: Two consecutive traffic experiments failed with "`error: the following arguments are required: --data_path`". *Diagnosis*: Worker implementations used relative paths that broke under SLURM's working directory. *Patch*: Added to domain knowledge: "Always pass absolute data path: `/path/to/datasets/traffic`". *Result*: Subsequent experiments resolved paths correctly.

B Extended experimental results

B.1 Cross-domain summary

Table 6 summarizes the best results across all three domains and both models.

Domain	Model	Best metric	vs. baseline	Cost
CUDA	GPT-5.2	**5.14**× mean	—	$190
	Opus	4.48× mean	—	$170
LLM	GPT-5.2	0.970 BPB	—	$150
	Sonnet	0.869 BPB	−10%	$120
	Opus	**0.758 BPB**	−22%	$200
Traffic	Opus	**0.0214** RMSE	−25%	$200
	GPT-5.2	0.0220 RMSE	−23%	$180

Table 6: **Cross-domain summary.** GPT-5.2 produces faster CUDA kernels (mean speedup on the 66-task overlap), while Opus 4.6 achieves lower validation loss on LLM pretraining (22% better than GPT-5.2) and lower RMSE on traffic forecasting. Neither model dominates uniformly. "vs. baseline" for LLM is relative to GPT-5.2; for traffic, relative to Seasonal Naïve(168). Cost is approximate API spend per campaign.

B.2 Experimental configuration

Table 7 summarizes the experimental setup for each domain.

Table 7: Experimental configuration summary.

Domain	Metric	Dir.	Budget	Hardware	Time Limit
CUDA Kernel	speedup (×)	max	50	4× H100	7,200 s
LLM Pretraining	val_bpb	min	50	4× H100	1,200 s
Traffic	RMSE	min	50	4× H100	7,200 s

B.3 LLM pretraining: extended results

Top-K experiments per model. Experiment names use the following shorthand. **Architecture:** "NL×Dd" = N Transformer layers with hidden dimension D (e.g., 8L×512d ≈ 33.7M parameters); "GQA" = grouped-query attention (fewer key/value heads to reduce memory); "SwiGLU" = gated feed-forward activation (used in LLaMA-style models); "QK-norm" = normalization of query and key vectors for training stability at high learning rates; "GPT-2" vs "LLaMA" indicates the overall architecture template (GPT-2 uses LayerNorm + learned position embeddings; LLaMA uses RMSNorm + RoPE + SwiGLU). **Optimizer/schedule:** "AdamW" = the standard optimizer; "Muon" = an alternative optimizer; "cosine" = cosine learning rate decay; "WSD" = warmup-stable-decay schedule. **Data:** "query+answer" vs "answer-only" = whether training includes the question prefix or only the answer; "N shards" = number of data shards sampled from the 500-shard corpus. **Other:** "no compile" = `torch.compile` disabled (avoids compilation overhead under the 20-min budget); "no grad-ckpt" = gradient checkpointing disabled (trades memory for speed); "ReLU^2" = squared ReLU activation in the feed-forward layers.

Full experiment results. Table 9 shows all GPT-5.2 experiments with valid `val_bpb`, and Table 10 shows all Opus experiments.

What didn't work. For GPT-5.2: `torch.compile` failed repeatedly due to RoPE lazy cache initialization conflicts with Inductor/CUDAGraph, causing 5 experiments to produce no valid metrics. The Muon optimizer consistently underperformed AdamW (6 head-to-head tests, 5–12% gap in Opus's campaign). WSD schedule was 2.16% worse than cosine. Deep-narrow architectures (≥20 layers) consistently underperformed wider-shallower designs due to lower throughput in the fixed 20-minute budget.

For Sonnet 4.6: 76 total experiments were run, but only 27 achieved valid `val_bpb` under 3.0. The remaining 49 either failed to train (wall clock <10s, indicating startup crashes) or

Table 8: Top-5 LLM speedrun experiments per model.

Experiment	val_bpb	Wall clock
GPT-5.2		
8L×512d, GQA, query+answer, AdamW+cosine, no compile	0.9697	1200 s
24L×512d, GQA, SwiGLU, AdamW+WSD	0.9786	1200 s
8L×512d, answer-only, AdamW+cosine	0.9852	1200 s
8L×640d, answer-only, no compile	0.9954	1200 s
12L×512d, answer-only, no compile	1.0102	1200 s
Sonnet 4.6		
11L×768d, GQA, QK-norm, Muon+AdamW, cosine	0.8686	1200 s
11L×768d, GQA, QK-norm, Muon+AdamW, ReLU^2	0.8713	1200 s
11L×768d, QK-norm, Muon+AdamW, no grad-ckpt	0.8757	1200 s
11L×768d, ReLU^2, Muon+AdamW, no grad-ckpt	0.8808	1200 s
10L×816d, no grad-ckpt	0.8813	1200 s
Opus 4.6		
10L×752d, shallow config	**0.7578**	1200 s
12L×672d, QK-norm, LR=3e-3, 10 shards	0.7587	1200 s
12L×672d, wide shallow	0.7624	1200 s
12L×704d, "allstar" config, 10 shards	0.7718	1200 s
12L×672d, small batch, more steps	0.7758	1200 s

Table 9: All GPT-5.2 LLM speedrun experiments with valid val_bpb (sorted). 45 total experiments; 28 with valid metrics, 12 cancelled, 5 with degenerate metrics (>3.0, indicating training failures).

Configuration	val_bpb	Params	tok/s
8L×512d, GQA, Q+A, cosine, no compile	**0.970**	33.7M	169K
24L×512d, GQA, SwiGLU, WSD	0.979	91.4M	306K
8L×512d, answer-only, cosine	0.985	33.7M	843K
8L×640d, answer-only, no compile	0.995	52.6M	321K
12L×512d, answer-only, no compile	1.010	50.5M	288K
16L×704d, GPT-2, WSD	1.044	96.2M	276K
10L×896d, GPT-2, WSD	1.067	97.6M	297K
8L×512d, Muon, WSD	1.075	33.7M	332K
8L×512d, SwiGLU, WSD	1.076	19.7M	459K
14L×704d, SwiGLU, WSD	1.085	84.7M	188K
12L×640d, GQA, cosine	1.089	51.8M	266K
12L×768d, GPT-2, WSD	1.094–1.098	86.0M	302–334K
20L×640d, GPT-2, WSD	1.106	99.5M	252K
12L×768d, SwiGLU, WSD	1.131	72.2M	221K
20L×640d, RoPE, GPT-2	1.139	98.8M	196K
12L×768d, AdamW (no schedule)	1.208	86.0M	392K
20L×512d, GQA, SwiGLU, WSD	1.214	55.2M	134K
16L×640d, MQA, SwiGLU, WSD	1.442	67.0M	129K
8L×512d, answer-only, no compile	1.446	25.3M	141K
24L×512d, GQA, SwiGLU, WSD	1.471	66.2M	108K

produced degenerate metrics (∼3.0–5.2 bpb). Sonnet's best result (0.869, 11L×768d with Muon+AdamW hybrid and QK-norm) placed between GPT-5.2 and Opus.

Variance runs. Four additional GPT-5.2 Phase 3 runs (v1–v4) with identical Phase 1/2 inputs yielded best val_bpb of 0.964, 1.011, 1.006, and 1.020 respectively, a spread of 0.056 from identical starting conditions, confirming the stochasticity discussion in the main text.

Example discovered architectures. The two winning configurations illustrate strikingly different design philosophies:

Table 10: All Opus 4.6 LLM speedrun experiments with valid val_bpb (sorted). 50 total; 36 analyzed, 14 cancelled.

Configuration	val_bpb	Params	tok/s
10L×752d, shallow	**0.758**	98.2M	163K
12L×672d, QK-norm, LR=3e-3, 10 shards	0.759	92.5M	121K
12L×672d, wide shallow	0.762	92.5M	168K
12L×704d, "allstar" 6 tricks	0.772	98.8M	113K
12L×672d, small batch, more steps	0.776	92.5M	118K
12L×672d, 10 data shards	0.780	92.5M	125K
16L×640d	0.782	98.3M	141K
12L×704d, fast pipeline	0.797	94.7M	103K
12L×672d, BS=96, β_2=0.99	0.800	92.5M	103K
12L×672d, schedule fix	0.801	92.5M	88K
12L×704d, kitchen sink	0.819	98.8M	77K
12L×672d, embed skip + value residual	0.819	92.5M	90K
12L×672d + tricks (5 variants)	0.820–0.846	92–99M	68–89K
16L×640d, Muon variants	0.828–0.862	98.3M	50–114K
14L×640d, all tricks	0.842	95.5M	68K
12L×704d, Muon	1.078	94.7M	84K
20L×576d, baseline	1.126	98.5M	95K
8L×768d, very wide	1.128	81.8M	57K
24L×512d, deep narrow	1.130	91.5M	38K

Table 11: Winning architecture comparison: Opus vs. GPT-5.2 on LLM speedrun.

Parameter	Opus (0.758 bpb)	GPT-5.2 (0.970 bpb)
Layers	10	8
d_model	752	512
d_ff	2256 (SwiGLU)	1365 (SwiGLU)
Heads	8	8 (GQA, kv=2)
Parameters	98.2M	33.7M
Optimizer	AdamW (fused)	AdamW
Peak LR	1.5e-3	6e-4
Schedule	Cosine	Cosine
Seq length	512	1024
Vocab	32768 (BPE)	256 (byte)
Throughput	163K tok/s	169K tok/s
Tokens seen	194M	202M

Opus discovered a wider-shallower architecture that uses nearly the full 100M parameter budget, with a custom BPE tokenizer (vocab 32768, ∼5.38 bytes/token) and high learning rate. GPT-5.2 took a more conservative path: a smaller model with byte-level tokenization (vocab 256, 1 byte/token) and GQA to reduce KV-cache parameters, achieving higher raw throughput but lower quality per token processed. Opus's playbook explicitly noted: "Throughput is the #1 predictor of val_bpb ($R^2 \approx 0.85$)"; yet its winning model sacrificed some throughput for a better tokenizer and wider architecture, suggesting that token quality (via BPE) can compensate for lower throughput.

B.4 CUDA kernels: extended results

Figure 4 shows the per-task speedup distribution for both models on the 66-task overlap set. The distribution is heavy-tailed: a small number of tasks achieve extreme speedups (>10×), while most cluster in the 1–5× range. Both models show similar speedup profiles, with GPT-5.2 achieving higher peaks on fused operators.

Full per-task results for the direct comparison subset, GPT-5.2-only tasks, and Opus-only tasks are available in the code repository alongside the raw experiment databases.

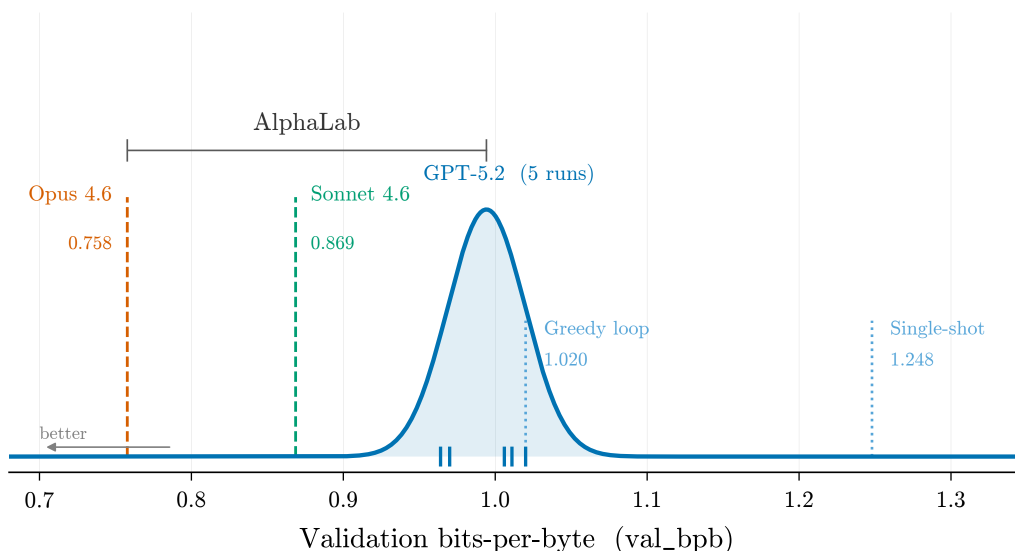


Figure 3: **LLM pretraining speedrun: outcome distribution.** The blue curve shows a Gaussian fit to the best val_bpb from five independent AlphaLab + GPT-5.2 runs with identical inputs (same Phase 1/2 outputs, hardware, and 50-experiment budget); tick marks indicate individual runs. Dashed lines mark single-campaign AlphaLab results with Opus 4.6 (orange, 0.758) and Sonnet 4.6 (green, 0.869); dotted lines mark GPT-5.2 baselines without AlphaLab: greedy loop (1.020) and single-shot (1.248). The key observation is that while any single campaign has meaningful variance (∼0.056 BPB spread from identical starting conditions), in practice a user can launch multiple campaigns—potentially with different models—and select the best result, effectively sampling the left tail of the distribution. Multi-model campaigns are especially powerful: Opus 4.6's result lies entirely to the left of every GPT-5.2 run, so a campaign that includes both models accesses regions of the search space that neither would find alone.

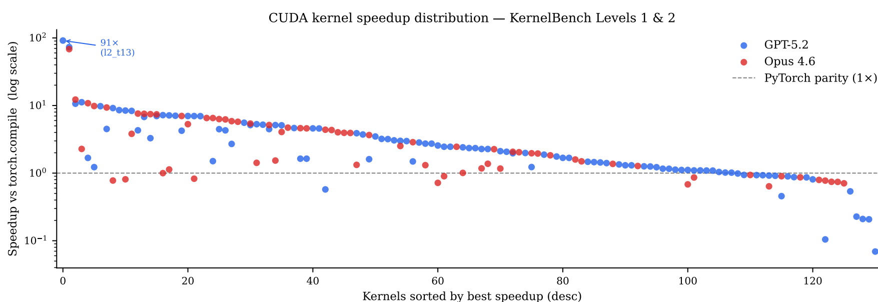


Figure 4: Per-task speedup over `torch.compile` for GPT-5.2 and Opus 4.6 on the 66-task direct comparison subset (log scale). Dashed line indicates 1× parity. Both models achieve extreme speedups on normalization and reduction kernels but fail to beat `torch.compile` on convolutions.

Per-level breakdown. Table 12 provides a per-level breakdown with both baseline metrics, enabling direct comparison with the KernelBench leaderboard (Ouyang et al., 2025) (which reports fast_1 vs `torch.native` on L40S hardware). AlphaLab achieves 40% fast_1 on L1 and 26–43% on L2 against `torch.native`, compared to 12% / 36% for single-shot DeepSeek R1 and 43% / 72% for iterative R1 with profiler feedback (10 LLM calls). Against

the harder torch.compile baseline, for which no published external numbers exist, ALPHALAB achieves 50–82% on L1 and 16–84% on L2. The gap between L1 and L2 on the torch.compile metric reflects that torch.compile is especially effective at fusing L2 operator chains, making them harder to beat.

Table 12: Per-level CUDA kernel results. ALPHALAB on H100 NVL; KernelBench baselines on L40S. fast_1 is reported as fraction of correct tasks (for ALPHALAB) or fraction of 100 total tasks (for baselines).

		Level 1 (single-op)			Level 2 (fusion)		
System	**Model**	**Correct**	$\text{fast}_1^{\text{nat}}$	$\text{fast}_1^{\text{comp}}$	**Correct**	$\text{fast}_1^{\text{nat}}$	$\text{fast}_1^{\text{comp}}$
ALPHALAB	GPT-5.2	61/65	66%	82%	49/54	88%	84%
ALPHALAB	Opus 4.6	45/52	89%	82%	31/35	84%	52%
KernelBench†	R1 (iter.)	—	43%	—	—	72%	—
KernelBench†	R1 (1-shot)	—	12%	—	—	36%	—
KernelBench†	o1 (1-shot)	—	10%	—	—	24%	—

†Ouyang et al. (2025); L40S GPU; fast_1 as fraction of 100 tasks per level, vs torch.native. "R1 (iter.)" = DeepSeek R1 with 10 iterative calls including profiler feedback.

Comparison with Sakana AI's CUDA Engineer. Table 13 compares ALPHALAB with the AI CUDA Engineer (Lange et al., 2025a;b) on operation types present in both evaluations. The comparison is approximate: Sakana's robust-kbench uses custom task implementations rather than KernelBench task IDs, so we match by operation category rather than exact kernel. ALPHALAB achieves substantially higher speedups on LayerNorm (11.2× vs 0.18× over torch.compile) and competitive or better results on RMSNorm. Sakana's cross-entropy result (24.9× vs torch.compile) substantially exceeds ALPHALAB's (5.1–5.4×), though Sakana's original benchmark was found to contain exploitable evaluation artifacts on several tasks (Lange et al., 2025b). In aggregate, ALPHALAB's mean speedup of 3.27–3.47× over torch.native compares favorably to Sakana's decontaminated mean of 1.49×.

Table 13: Per-operation comparison with Sakana AI CUDA Engineer (Lange et al., 2025b). All results on H100. Sakana results from robust-kbench (12 kernels); ALPHALAB results from the nearest KernelBench task. Speedups vs torch.compile.

Operation	**Sakana AI**	**ALPHALAB GPT-5.2**	**ALPHALAB Opus**
LayerNorm (fwd)	0.18×	**11.19**×	2.28×
RMSNorm (fwd)	2.39×	1.97×	**2.07**×
Cross entropy (fwd)	**24.87**×	5.14×	5.41×
ResNet block (fwd)	**2.59**×	—	—
Aggregate mean spd. (vs torch.native)		**3.47**×	3.27×
Sakana decontaminated mean (vs torch.native)		1.49×	

Example discovered kernels. We highlight four representative optimization strategies the system discovered:

(1) Algebraic rewrite: diagonal matrix multiply (68×). The original PyTorch operation computes $\text{diag}(A) \cdot B$ by constructing the full diagonal matrix and performing a dense matmul. The system recognized that this is equivalent to elementwise multiplication of the diagonal vector with each row of B, avoiding $O(n^2)$ memory allocation entirely. The playbook recorded: *"Diagonal matrix ops: diag(A) · B is just elementwise multiply of the diagonal vector with each row of B. Yields 10–68× over PyTorch's full matmul."*

(2) Warp-shuffle reduction (75×). For sum-reduction kernels, the system wrote CUDA code using __shfl_down_sync for warp-level parallel reduction, avoiding shared memory round-trips entirely. This technique achieves 73–76× speedup over torch.compile for large reduction dimensions.

(3) Operator fusion: LayerNorm + residual add (91×). The system fused LayerNorm computation (mean, variance, normalize, affine transform) with a residual addition into a single kernel pass, using vectorized `float4` loads/stores and Welford's online algorithm for numerical stability. This eliminates multiple kernel launches and intermediate memory allocations.

(4) Failure case: convolution (0.05–0.73×). Handwritten convolution kernels consistently performed worse than `torch.compile`, which delegates to cuDNN's highly optimized implementations. The playbook explicitly warned: *"Do not attempt convolution kernels – cuDNN is too well-optimized; handwritten runs at 0.05–0.73×."* This "do not attempt" knowledge is as valuable as positive findings, preventing budget waste on disproven approaches.

Known failure modes. Three categories of CUDA kernel failure were observed: (1) **Convolution kernels** (cuDNN dominance): all handwritten convolution kernels ran at 0.05–0.73× the `torch.compile` baseline, as cuDNN's autotuned implementations are extremely well-optimized for standard convolution patterns on H100 hardware. (2) **Bool-mask and scatter operations**: correctness failures where the optimized kernel's output diverged from the reference beyond the `atol=1e-3` threshold, typically due to incorrect handling of boolean indexing or non-contiguous memory layouts. (3) **CUTLASS-dependent kernels**: experiments that attempted to use CUTLASS templates for GEMM operations failed because the CUTLASS submodule was not fully initialized in the execution environment. Together, these account for 9 incorrect results (GPT-5.2) and 8 incorrect results (Opus) out of 103 and 59 analyzed tasks respectively.

B.5 Traffic forecasting: extended results

Table 14: Top-5 traffic forecasting experiments per model.

Experiment	RMSE
GPT-5.2	
iTransformer ctx336 + time features + static cat	0.02204
iTransformer ctx336 + hour-of-week + RevIN + horizon weights	0.02247
iTransformer ctx336 + hour-of-week + RevIN	0.02256
iTransformer ctx672 + hour-of-week + static cat	0.02289
N-HiTS ctx336 + multiscale + time features	0.02295
Opus 4.6	
TFT + dropout 0.3	0.02142
TFT + MSE loss, large, 100 epochs	0.02153
TFT + MSE loss, 150 epochs	0.02156
TFT + MSE loss, 100 epochs	0.02175
TFT + residual MLP correction	0.02177
Seasonal Naïve(168)	0.02870

Search trajectory contrast. The most striking difference between models is the exploration–exploitation balance. GPT-5.2's Strategist maintained diversity throughout the campaign: its top-10 experiments span iTransformer, N-HiTS, TSMixer, TimesNet, PatchTST, and ridge stacking. Opus's Strategist converged on TFT after experiment $\sim$10 and spent the remaining 40 experiments refining TFT hyperparameters (dropout, loss function, epochs, context length, ensemble methods). As the playbook noted: *"TFT is the clear winner – Variable Selection Networks + attention + calendar known-future-inputs."* This focused exploitation happened to land on the best architecture (TFT at 0.02142 beat iTransformer at 0.02204), but the playbook also acknowledged the risk: 14 experiments (28% of budget) were spent on post-processing methods that all failed to improve over the base TFT.

Opus's winning TFT configuration. Hidden dim 160, 8 attention heads, dropout 0.3, context 336h, sensor ID embedding (dim 48), MSE loss, cosine annealing LR (5e-4 to 1e-

Table 15: Traffic forecasting: top-10 for each model plus baselines.

Model	Configuration	RMSE
GPT-5.2 (top-10 span 5 architectures)		
GPT-5.2	iTransformer ctx336 + timefeat + static cat	0.02204
GPT-5.2	iTransformer ctx336 + hour-of-week + RevIN + horizon wt.	0.02247
GPT-5.2	iTransformer ctx336 + hour-of-week + RevIN	0.02256
GPT-5.2	iTransformer ctx672 + hour-of-week + static cat	0.02289
GPT-5.2	N-HiTS ctx336 + multiscale + timefeat	0.02295
GPT-5.2	TSMixer ctx672 + timefeat + static cat	0.02300
GPT-5.2	iTransformer ctx672 + hour-of-week + RevIN	0.02315
GPT-5.2	Ridge stack (PatchTST + snaive)	0.02342
GPT-5.2	TimesNet ctx672 + timefeat + static cat	0.02364
GPT-5.2	PatchTST ctx672 + patch16 + timefeat + RevIN	0.02380
Opus 4.6 (top-10 all TFT or TFT-ensemble)		
Opus	TFT + dropout 0.3, ctx336, 80 epochs	**0.02142**
Opus	TFT + MSE loss, large, 100 epochs	0.02153
Opus	TFT + MSE loss, 150 epochs	0.02156
Opus	Per-sensor model selection (5-way)	0.02162
Opus	TFT + MSE loss, 100 epochs	0.02175
Opus	Residual MLP correction on TFT	0.02177
Opus	Simple avg top-3 ensemble	0.02183
Opus	TFT large ctx336, 100 epochs	0.02189
Opus	Ensemble top-3 per-horizon	0.02193
Opus	Ensemble TFT + PatchTST weighted	0.02213
Baselines		
—	Seasonal Naïve(168)	0.02871
—	Seasonal Naïve(24)	0.03674
—	Naïve (last value)	0.04456

6), 80 epochs, batch size 64, 500 batches/epoch. Total parameters: 2.1M. Training time: 27 minutes on 1×H100.

B.6 Convergence curves

Figure 5 shows the running-best metric as a function of experiment number for all three domains and both models.

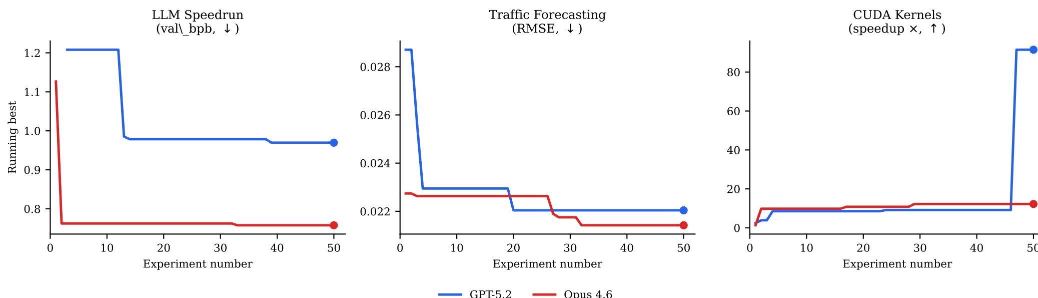


Figure 5: Convergence curves across all three domains. Each line shows the running-best metric (lower is better for LLM and traffic; higher for CUDA) as a function of analyzed experiment count. Both models improve rapidly in the first 10–15 experiments and plateau by 25–30.

Convergence dynamics. All campaigns show a characteristic pattern: rapid improvement in experiments 1–15 as the system explores the most promising architectural families,

followed by diminishing returns as it enters the refinement phase. By experiment 25–30, the running-best metric has typically stabilized to within 5% of the final value.

Convergence speed varies by domain. CUDA kernels converge fastest because each experiment produces a clear, unambiguous speedup metric with low noise: the system can quickly identify which optimization strategies work (e.g., warp-shuffle reductions) and which don't (convolutions). LLM pretraining converges more slowly due to higher variance: GPU contention introduces ∼0.05 BPB of noise, and architectural differences compound over 20-minute training runs. Traffic forecasting falls in between, with moderate noise from the 7-window rolling evaluation.

The convergence criterion (C=20 experiments without improvement) was never triggered in our campaigns because we set a hard budget of 50 experiments. In practice, most campaigns would have naturally terminated around experiment 35–40 based on the convergence curves.

C Playbook excerpts and dynamics

C.1 LLM speedrun playbook (GPT-5.2, after 50 experiments)

The GPT-5.2 LLM speedrun playbook evolved from a blank document to a comprehensive 2,000-word strategic reference over 50 experiments. We highlight the most informative sections (verbatim, lightly formatted):

Verified SOTA entry.

> *Verified SOTA: Exp #14 –* `llama24_512_gqa_swiglu_adamw_wsd`*. Best val_bpb_full20m_best: 0.9786. Params: 91.38M. Arch: LLaMA-style (RMSNorm + RoPE + SwiGLU, pre-norm, tied embeddings). Shape: 24 layers, d_model=512, n_head=8, GQA kv_heads=2. Optim: AdamW (β=(0.9,0.95), wd=0.1, clip=1.0) + WSD (lr=6e-4, warmup=800). Data: byte tokenizer (vocab_size=256), language_filter=en, include_query=true, shards 1–10, seq_len=1024.*

Metric integrity warning. The Strategist discovered that the leaderboard was mixing smoke-test metrics (∼8.0 bpb, near-random) with full 20-minute metrics (∼0.98–1.21 bpb), inverting rankings:

> *Metric integrity warning (critical): The board leaderboard is currently mixing smoke/step-0 eval metrics and full 20-minute scored metrics. This can invert rankings and lead to incorrect decisions.*

Infrastructure warnings.

> *torch.compile has repeatedly failed with RoPE lazy cache init + Inductor/CUDAGraph overwrites. Treat compile as opt-in only with cache pre-init and cudagraphs disabled.*

Post-budget triage. In the final experiments (#43–#50), the Strategist documented cascading infrastructure failures (CUDA OOM, IndentationError in the harness, SLURM job failures) and prioritized fix actions, demonstrating the playbook's role as an operational log as well as a strategic document.

C.2 CUDA kernels playbook (GPT-5.2)

The CUDA kernels playbook (GPT-5.2) developed a technique taxonomy and explicit avoidance rules. Key excerpts:

Positive technique taxonomy.

> *Diagonal matrix ops: $diag(A) \cdot B$ is just elementwise multiply of the diagonal vector with each row of B. Yields 10–68× over PyTorch's full matmul.*

> *Warp-shuffle reductions yield 73–75× on sum operations. Use* `__shfl_down_sync` *for warp-level parallel reduction; avoid shared memory round-trips.*
>
> *Operator fusion (LayerNorm + residual, GEMM + LogSumExp): fuse sequential operations into a single kernel pass. Use vectorized float4 loads/stores for memory bandwidth. Yields 16–91×.*

"Do not attempt" list.

> *Do not attempt convolution kernels – cuDNN is too well-optimized; handwritten runs at 0.05–0.73×.*
>
> *CUTLASS-dependent kernels: the submodule is incomplete in this environment. Do not attempt CUTLASS GEMM templates.*
>
> *Bool-mask scatter operations: correctness failures due to non-contiguous memory layouts. Avoid unless the operator is simple enough to verify by hand.*

Queue-pruning heuristics. The Strategist learned to cancel queued experiments proactively based on accumulating evidence:

> *If an optimization technique has failed on 2+ similar tasks, cancel all remaining variants. Specifically: cancel all convolution experiments after L1_t54 and L1_t58 both showed 0.05–0.73×.*

C.3 Traffic forecasting playbook (Opus 4.6)

The Opus traffic playbook demonstrates both the strengths and risks of playbook-driven convergence.

TFT convergence trajectory. The Strategist initially proposed diverse architectures (PatchTST, DeepAR, iTransformer, TFT, N-BEATS). After experiment ∼10, TFT emerged as the clear leader (RMSE ∼0.0226 vs. ∼0.0227 for PatchTST). The playbook recorded:

> *TFT is the clear winner – Variable Selection Networks + attention + calendar known-future-inputs. MSE loss directly optimizes RMSE; 3.9% better than quantile loss. Dropout 0.3 is critical; dropout=0.1 underperforms by ∼2%. 8 attention heads > 4 for this task.*

From experiment 15 onward, the Strategist devoted the remaining budget almost exclusively to TFT refinement and post-processing.

Hyperparameter findings. The playbook documented specific findings: dropout 0.3 optimal (tested 0.1, 0.2, 0.3, 0.4); MSE loss beats quantile for RMSE optimization; 80–150 epochs sufficient (diminishing returns beyond 100); context 336h optimal (168h viable, 672h not consistently better).

The post-processing trap. The playbook's final campaign statistics reveal a cautionary tale: 14 experiments (28% of budget) were spent on post-processing methods (bias correction, residual MLP, per-horizon calibration, ensemble methods), all of which failed to meaningfully improve over the best single TFT. The playbook explicitly noted: *"TFT residuals are near-random (SNR ∼0.04) – post-processing cannot extract signal from noise."*

iTransformer absence. Opus never explored iTransformer – the architecture that GPT-5.2 found to be the strongest. This represents the fundamental tension in playbook-driven search: early convergence on TFT was *correct* in that TFT turned out to be the best architecture Opus found, but it may have been *suboptimal* in that broader exploration might have discovered the iTransformer + per-horizon calibration trick that GPT-5.2 found (RMSE 0.01837 in the full evaluation, substantially better than Opus's 0.02142).

C.4 Playbook growth dynamics

We analyze playbook dynamics across domains and models.

Growth trajectory. Playbooks grow rapidly in the first 15 experiments as the Strategist documents initial findings, then transition to consolidation and refinement. The Opus LLM speedrun playbook reached ~2,500 words by experiment 50, organized into 7 major sections (objective, definitive findings, tricks verdict, confirmed losers, GPU contention analysis, best configuration reference, and lessons for future work). The GPT-5.2 traffic playbook reached ~800 words by experiment 50, more operational in tone (priority reruns, evaluation integrity gates, shipping rules).

Self-correction events. We observed several instances of the Strategist invalidating its own prior playbook entries:

- **LLM speedrun (Opus)**: Early playbook entries recommended deeper architectures (20L×576d). After experiments 15–20 showed these underperforming, the playbook was updated: *"≥14 layers: Too deep for 20-minute budget"* and *"Muon optimizer: Always 5–12% worse than AdamW (6 head-to-head tests)."*
- **Traffic (Opus)**: Early enthusiasm for post-processing methods was reversed after 14 failed attempts: *"ALL post-processing/bias correction at full scale failed – TFT residuals are near-random."*

Cross-model comparison. Opus playbooks tend to be more structured and analytical (explicit rankings, comparative tables, quantified findings), while GPT-5.2 playbooks are more operational (action items, priority lists, fix instructions). Opus's traffic playbook contains a definitive tier list (S/A/B/C), campaign statistics, and unrealized ideas for future work; GPT-5.2's traffic playbook focuses on evaluation integrity, root cause analysis of failures, and priority reruns. Both styles are effective: Opus's analytical approach enabled tight convergence on TFT, while GPT-5.2's operational approach helped navigate infrastructure issues (missing data paths, SLURM failures).

D User interface

ALPHALAB includes a web-based dashboard (React/TypeScript frontend, FastAPI backend with WebSocket streaming) for real-time campaign monitoring. The interface is organized into three resizable panes:

Kanban board. The central view displays experiments organized by status across 9 columns: `to_implement`, `implemented`, `checked`, `queued`, `running`, `finished`, `analyzed`, `done`, and `cancelled`. Each experiment is represented as a card showing its name, hypothesis, current metric (if available), assigned Worker ID, and SLURM job ID. Columns are collapsible and support pagination for large campaigns.

Leaderboard. A sortable, filterable table ranking all analyzed experiments by the primary metric (configurable per domain). The leaderboard auto-refreshes every 10 seconds and on new experiment events, allowing the human to track progress in real time.

File tree and viewer. The left pane provides a hierarchical file browser of the workspace, with syntax-highlighted code viewing for generated scripts, model implementations, and configuration files.

Conversation stream. The right pane shows a real-time log of agent tool calls and results (shell commands executed, files read, web searches performed), providing full transparency into the system's reasoning and actions.

Human-in-the-loop interface. A chat panel allows the human operator to ask questions during a campaign (e.g., "What's happening?", "Any errors?", "Best model so far?"). We are experimenting with allowing the human to suggest specific experiments, veto proposals, or add notes to the playbook. Early observations suggest that even minimal human guidance (e.g., "focus on TFT variants" or "stop trying convolution kernels") can significantly

accelerate convergence, though quantifying this effect requires controlled experiments we have not yet run.

Milestone reports. Periodic summaries generated by the Reporter agent are displayed in the status view, showing cumulative progress, flagged experiments (suspicious metrics, smoke-test-only results), and recommendations for the Strategist.

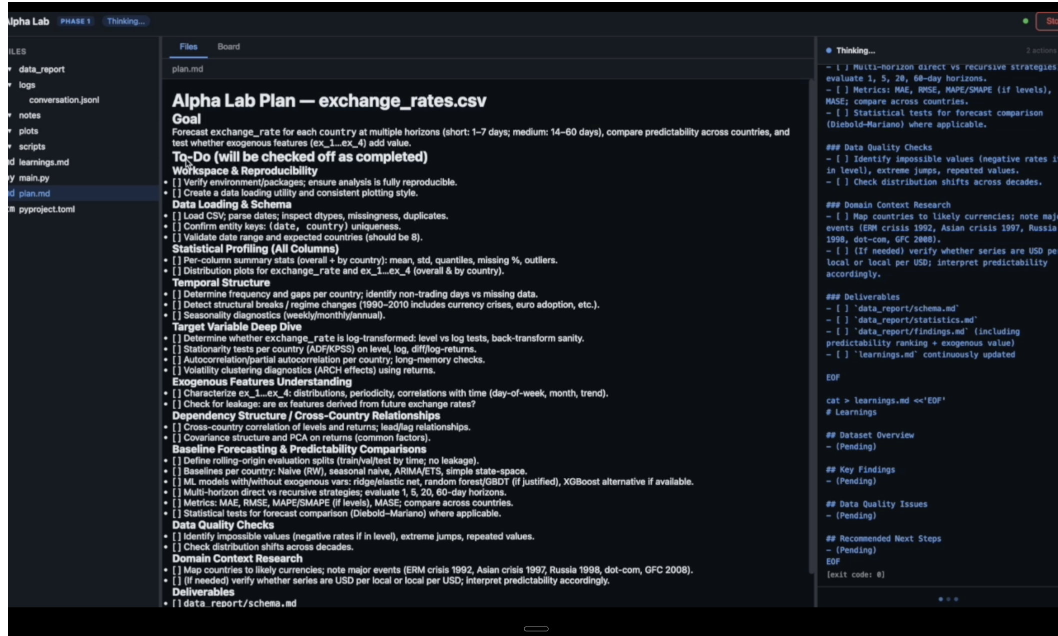


Figure 6: **Phase 1 dashboard: Explorer agent plan.** The Explorer agent has autonomously generated a detailed to-do list (`plan.md`, center pane) for the exchange-rate forecasting domain, covering data loading, temporal structure analysis, stationarity tests, cross-country dependencies, baseline comparisons, and domain context research. Items are checked off as the agent completes them. The left pane shows the workspace file tree (scripts, notes, plots being generated); the right pane shows the agent's real-time thinking and tool calls.

E Additional experiments

E.1 Financial time series forecasting (exchange rates)

This domain tests ALPHALAB on a task with no built-in adapter; Phase 0 must generate everything from scratch.

Data. Eight synthetic daily exchange-rate series against the US dollar, spanning 6,071 business days (1990–2013). The dataset is stored in GluonTS format with a 30-business-day prediction horizon and 5 rolling-origin test windows per currency (40 test records total). Values range from $\sim$0.006 (item 4, a peg-like currency) to $\sim$2.1, with substantial cross-series heterogeneity.

Task. Forecast 30 business days ahead; evaluate via annualized Sharpe ratio (primary, maximize), RMSE, max drawdown, and directional accuracy. The trading evaluation maps forecasts to positions via position $=$ sign(predicted 30-day return), then computes PnL from realized returns.

Phase 0: adapter generation. Since "exchange rates" does not match any built-in adapter, the Phase 0 generation agent was invoked. It examined the data (detecting the panel

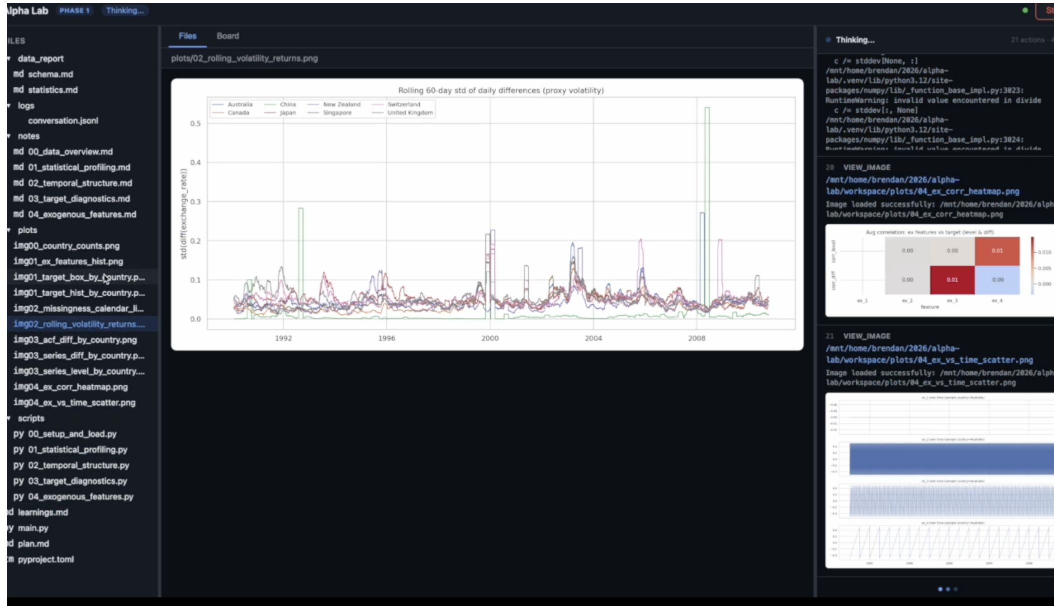


Figure 7: **Phase 1 dashboard: data exploration in progress.** The Explorer agent has generated analytical plots (center pane shows a rolling volatility time series across currency pairs) and is viewing them via the `view_image` tool (right pane, with additional correlation heatmaps and scatter plots). The left pane shows the growing collection of analysis scripts and generated plots in the workspace.

structure, running ADF stationarity tests), searched the web for FX forecasting best practices, and generated all 11 adapter files from scratch. Key design decisions made by the agent: use log-returns rather than levels, walk-forward split with embargo periods to prevent lookahead, Sharpe ratio as the primary metric, and max drawdown as a secondary safety metric. The generated `domain_knowledge.md` included guidance on per-series normalization (fit on train window only to avoid leakage), global models with item embeddings to share strength across 8 series, and probabilistic forecasting given FX noisiness.

Phase 1: data exploration. The Explorer autonomously generated a 9-section plan and executed it over several hours, producing 9 analysis scripts, 12 plots, and 11 detailed notes. The completed plan (all items checked off by the agent):

```
✓ Setup & Intake: locate dataset, validate 8 series, build DataFrames
✓ Schema & Profiling: per-series stats, data quality checks
✓ Temporal Structure: date ranges, frequency validation, regime detection
✓ Return Transformations: levels vs returns, ADF/KPSS stationarity tests
✓ Cross-Series Dependency: correlation, PCA, lead-lag analysis
✓ Baselines: naive, drift, AR(1), ARIMA, VAR on returns
✓ Trading Backtests: sign-of-return strategies, transaction cost
  sensitivity
✓ Model Recommendations: classical + DL families
✓ Final Deliverables: schema.md, statistics.md, findings.md
```

Key findings recorded in `learnings.md`:

- FX levels are non-stationary (near random walks); log-returns are approximately stationary with heavy tails and strong volatility clustering.
- Return autocorrelation is near zero – weak mean predictability, but volatility/risk scaling is more promising.
- Strong cross-series correlations (max 0.80 between items 0 and 6); PCA shows 45% of variance in PC1 (common factor), 19% in PC2.
- Item 4 exhibits peg-like behavior with step changes, warranting separate handling.

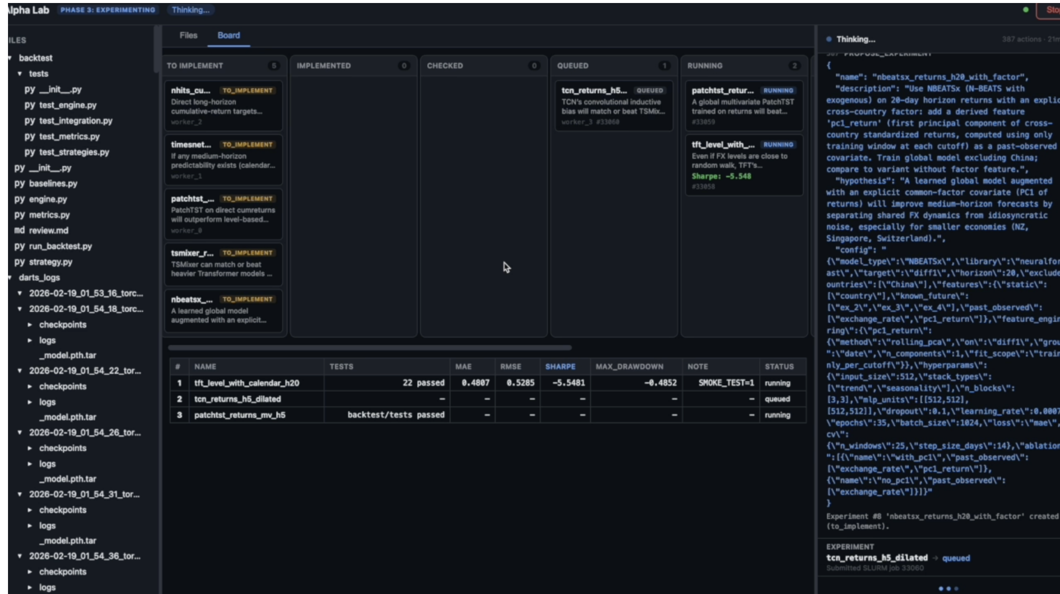


Figure 8: **Phase 3 dashboard: GPU experimentation.** The Kanban board (center, top) shows experiments in various stages: `to_implement`, `implemented`, `checked`, `queued`, and `running`. Each card displays the experiment name, hypothesis, and current status. The leaderboard (center, bottom) ranks completed experiments by the primary metric. The left pane shows the workspace file tree with experiment directories, checkpoints, and logs. The right pane shows the conversation stream of the currently active Worker agent implementing an experiment.

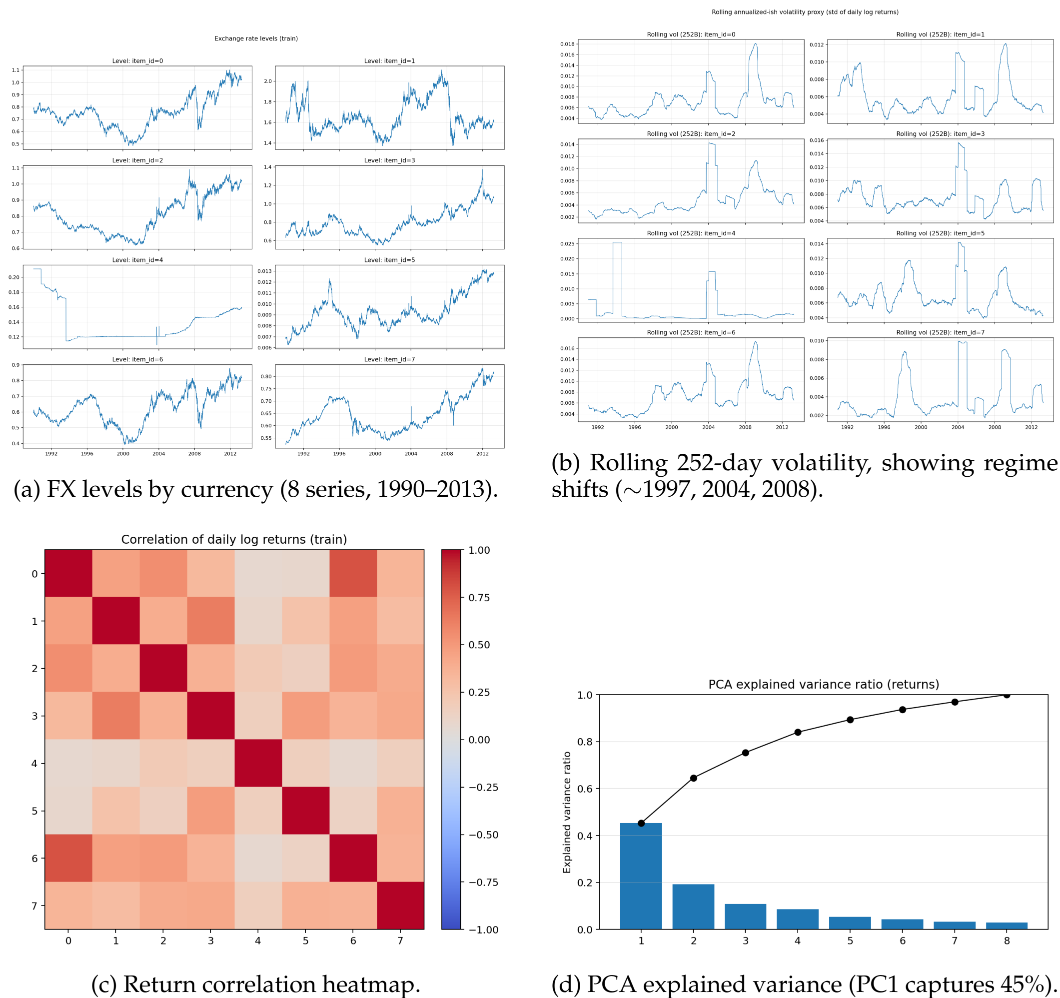
- Rolling-origin baseline trading yields Newey–West adjusted Sharpe ≈0.16 (univariate drift/AR(1)); multivariate VAR(1) improves marginally to ≈0.17.

Phase 2: evaluation harness. The Builder produced a walk-forward backtesting framework with a `Strategy` abstract base class (requiring `fit(y_train)` and `predict(y_context)` methods), a `WalkForwardEngine` with configurable embargo, and metrics including Sharpe, max drawdown, directional accuracy, and RMSE. The Critic caught a subtle lookahead bug: a rolling volatility computation in the feature engineering pipeline used the current day's close price, which would not be available at prediction time. The Tester validated the fix, and the loop converged in 3 iterations. The Critic's review also flagged that the embargo parameter only affects `fit()` but not `predict()`'s context window – a non-blocking issue that was documented for future improvement.

Phase 3: GPU-scale experimentation. The system ran 43 experiments (29 with valid Sharpe ratios) spanning a wide range of architectures: PatchTST, TimesNet, DeepAR, DeepVAR, TFT, N-HiTS, N-BEATSx, TCN, TSMixer, Mamba SSM, TS2Vec, and a custom Transformer policy network trained end-to-end on a differentiable Sharpe loss. Table 16 shows the full leaderboard.

The top two results (TimesNet at 4.21 and PatchTST factor-first at 2.78) were flagged by the system's own debriefs as unreliable: both are computed from only 5 non-overlapping 30-day trades per currency, leverage was 5–8×, and no Newey–West correction was applied. The system explicitly noted: *"Treat as high-variance, not reliable evidence of edge until gating passes."*

The most interesting experiment was the Transformer policy network (rank 3, Sharpe 0.748), which trained end-to-end on a differentiable Sharpe-like surrogate loss:



(a) FX levels by currency (8 series, 1990–2013).

(b) Rolling 252-day volatility, showing regime shifts (∼1997, 2004, 2008).

(c) Return correlation heatmap.

(d) PCA explained variance (PC1 captures 45%).

Figure 9: Phase 1 exploration plots generated autonomously by ALPHALAB for the exchange-rate domain. The Explorer wrote the analysis scripts, generated these visualizations, and used the `view_image` tool to inspect them before recording findings.

> *Input: 512-day context of* $[r_t, EWMA_vol_{10}, EWMA_vol_{30}, EWMA_vol_{90}]$. *Model: 4-layer Transformer encoder, 8 heads,* d_{model}=128, *per-currency item embeddings. Loss:* $-mean(PnL)/std(PnL)$ *with leverage and turnover penalties.*

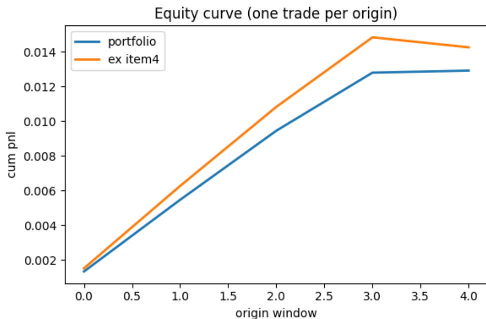
The system's debrief identified five specific limitations: only 40 trades total; no Newey–West adjustment despite overlapping horizons; the evaluation discards position magnitude (reducing the learned continuous policy to sign); models are fit per-currency despite item embeddings; and zero transaction costs. This level of self-critical analysis (automatically generated by the Worker agent during the `analyze` phase) is representative of the quality of debriefs across domains.

Playbook evolution. The playbook accumulated several domain-specific lessons over the campaign:

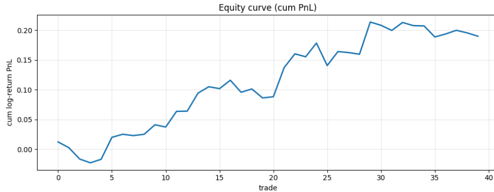
- *"Model log-returns, not levels – ADF tests confirm levels are non-stationary."*
- *"Per-series normalization must be fit on train window only; global normalization causes leakage."*
- *"Multivariate training on returns is beneficial vs univariate – cross-currency correlations (max 0.80) provide real signal."*
- *"Sharpe estimates from 5 trades are extremely high-variance; do not trust absolute values."*

Table 16: Exchange-rate experiment leaderboard (annualized Sharpe, higher is better). All experiments use GPT-5.2. Sharpe values are computed over only 5 rolling-origin windows per currency (40 trades total), making individual estimates high-variance. † denotes experiments flagged by the system's own debrief as potentially unreliable.

Rank	Sharpe	RMSE	Experiment
1†	4.214	0.0121	TimesNet multivariate Student-T
2†	2.780	0.0141	PatchTST factor-first PCA(3) + reconstruct
3	0.748	0.0163	Transformer policy net (Sharpe loss)
4	0.697	0.0185	Stacking ensemble (turnover-penalized)
5	0.395	0.0125	TCN multivariate risk-parity
6	0.121	—	TSMixer global returns
7	0.077	0.0160	TFT cross-currency with lag covariates
8–29	-0.05 to -23.9	0.011–0.022	(22 experiments with negative Sharpe)
Baseline: drift		—	NW-Sharpe ≈ 0.16
Baseline: VAR(1)		—	NW-Sharpe ≈ 0.17



(a) TimesNet multivariate (Sharpe 4.21†).



(b) Transformer policy net (Sharpe 0.75).

Figure 10: Equity curves for the top-ranked and most interesting exchange-rate experiments, generated by AlphaLab. The TimesNet result (left) exhibits suspiciously smooth performance from only 5 trades per currency; the Transformer policy net (right) shows more realistic variance.

- *"Exclude item_id=4 from pooled metrics or model separately (peg-like with step changes)."*

Discussion. This domain demonstrates AlphaLab's ability to handle genuinely novel tasks: the auto-generated adapter was functional and reasonable, correctly identifying the key design decisions (log-returns, walk-forward, Sharpe metric) without human guidance. The system's self-critical debriefs (flagging unreliable results, identifying statistical limitations, and suggesting follow-up experiments) show that the analyze phase produces genuine research insight, not just metric extraction. The quality gap between the auto-generated adapter and hand-written built-in adapters was modest, mainly in prompt specificity: the generated prompts were more generic, lacking the domain-specific guardrails (e.g., "never set deterministic algorithms on H100") that built-in adapters accumulate over time. The fact that 22 of 29 experiments produced negative Sharpe ratios also illustrates the difficulty of the domain (FX returns are notoriously hard to predict) and the system's willingness to report honest failures rather than cherry-pick results.

F Cost and token breakdown

We extracted token usage from the JSONL conversation logs recorded during each campaign. Table 17 summarizes the token consumption across all primary runs. All cost figures in this section reflect **LLM API token costs only** and do not include GPU compute. Our

experiments ran on on-premise 4×H100 hardware; at current cloud rates (∼$2–3/GPU-hour for H100 instances), the 12–48 hours of GPU time per campaign would add ∼$100–600 depending on domain and utilization.

Table 17: Token usage and API call counts for primary campaigns. Output tokens are consistently ∼1–2% of input tokens.

Domain	Model	Input tokens	Output tokens	API calls	Est. cost
LLM Speedrun	GPT-5.2	84.3M	979K	2,892	∼$150
LLM Speedrun	Opus 4.6	116.9M	1.5M	2,512	∼$200
LLM Speedrun	Sonnet 4.6	195.3M	3.4M	4,192	∼$120
Traffic	GPT-5.2	73.5M	1.4M	3,415	∼$180
Traffic	Opus 4.6	92.2M	1.7M	2,787	∼$200
Electricity	GPT-5.2	133.8M	1.8M	4,963	∼$190
Electricity	Opus 4.6	144.7M	1.8M	4,289	∼$200

Token distribution by phase. Phase 3 dominates token consumption, accounting for ∼97–98% of total tokens across all campaigns. This is expected: Phase 3 involves dozens of Strategist turns and Worker implementation/analysis/fix cycles, each requiring substantial context (leaderboard, playbook, debriefs, code). Phase 1 accounts for ∼1–2%, Phase 2 for ∼0.5–1%, Phase 0 for <0.1%, and Supervisor interventions for ∼0.3–0.5%.

Cost per experiment. For a typical 50-experiment campaign, the average cost per experiment is $3–4. However, this varies substantially: early experiments (where the Strategist has limited context) cost less than late experiments (where the playbook, leaderboard, and debrief history are large). The Strategist's context window grows roughly linearly with campaign length, driving the late-campaign cost increase.

Model cost comparison. Opus campaigns cost ∼$200 (higher per-token pricing), GPT-5.2 campaigns cost ∼$150–190 (lower per-token but sometimes more total tokens due to longer outputs), and Sonnet campaigns cost ∼$120 (lower per-token pricing despite high token counts, due to Sonnet's verbose generation style resulting in more API calls).

Variance runs. The 5 GPT-5.2 variance runs (Phase 3 only, shared Phase 1/2) consumed 74–97M input tokens each (∼$50–70 per run), confirming that Phase 3 alone accounts for the vast majority of campaign cost. The total cost across all experimental runs (including variance and ablation runs) was approximately $2,500.

G Failure analysis

We categorize failures observed across all campaigns into three tiers of severity.

Programmatic errors (Tier 1: self-healing). Runtime failures in LLM-generated code are the most common failure mode. Examples include: CUDA out-of-memory errors (batch size too large for H100 VRAM), PyTorch API breaking changes (PyTorch 2.9.1 removed `.total_mem` and `ReduceLROnPlateau(verbose=)`), Python import errors (package version mismatches), and SLURM job configuration issues (missing environment variables, incorrect paths).

Failure rates vary substantially by domain and model:

- **LLM speedrun**: GPT-5.2 had a 38% failure rate (17/45 experiments produced no valid metrics), largely due to `torch.compile` failures and harness bugs. Opus had only a 3% failure rate (1/37).
- **Traffic**: Both models had low failure rates (<5%), as the forecasting models are simpler and less prone to CUDA errors.
- **Electricity**: GPT-5.2 had a 55% failure rate, the highest across all campaigns, driven by PyTorch 2.9.1 API changes affecting multiple deep learning libraries.

- **CUDA kernels**: 9/103 incorrect for GPT-5.2 (8.7%) and 8/59 for Opus (13.6%), primarily correctness failures rather than runtime crashes.

These errors are generally self-healing through two mechanisms: the Worker fix cycle (up to k=2 repair attempts per experiment) and the Supervisor's health check (which patches `domain_knowledge.md` when error rates exceed τ=0.4).

Evaluation errors (Tier 2: silent corruption). Bugs in the evaluation framework that slip past the Phase 2 Critic/Tester loop. These are rarer but more dangerous because they silently corrupt all downstream results. Examples caught during manual review:

- **Electricity**: The GPT-5.2 campaign produced several suspiciously low RMSE values (21.82 for an ensemble, 89.19 for LightGBM), likely due to data leakage in feature engineering. These were flagged in the milestone reports but not automatically excluded.
- **CUDA**: The Critic caught a `torch.allclose` bug where the optimized output was passed as the *reference* argument, making correctness checks trivially permissive. This was fixed before Phase 3.
- **Traffic**: The GPT-5.2 playbook noted that some experiments reported metrics from smoke tests (10 instances) rather than full evaluation (6034 instances), polluting the leaderboard.

Strategic failures (Tier 3: going off the rails). Cases where the Strategist enters an unproductive loop or the playbook accumulates incorrect knowledge. These are the hardest to detect and the most costly.

- **Premature convergence**: Opus on traffic locked onto TFT after experiment ∼10, spending 40 experiments on TFT refinement and never exploring iTransformer (which GPT-5.2 found to be stronger). This *happened* to produce the best single-model result, but the opportunity cost is unknown.
- **Scattered search**: GPT-5.2's variance runs v2 and v3 on LLM speedrun never converged on a productive architectural region, producing best val_bpb of 1.011 and 1.006 respectively (vs. 0.964 for v1 and 0.970 for the primary run).
- **Budget waste on post-processing**: Opus's traffic campaign spent 14/50 experiments (28% of budget) on post-processing methods that all failed to improve over the base TFT.

We estimate that approximately 1 in 5 campaigns would benefit from human intervention to redirect the Strategist. The dashboard (Appendix D) is designed to make such interventions lightweight: the human can monitor the playbook evolution, spot premature convergence, and inject suggestions via the chat panel. The right balance between autonomy and oversight is an open research question; our experience suggests that a brief human check every 15–20 experiments is sufficient to catch strategic failures while preserving the system's autonomous operation.

H Reproducibility details

Full code, configuration files, adapter templates, and instructions for reproducing all experiments will be released on GitHub at `https://brendanhogan.github.io/alphalab-paper/`.

Model versions. All models were accessed in February–March 2026.

- **GPT-5.2**: Model ID `gpt-5.2`, accessed via OpenAI Responses API. Reasoning effort set to `low` for most campaigns.
- **Claude Opus 4.6**: Model ID `claude-opus-4-6-v1`, accessed via AWS Bedrock Converse API.
- **Claude Sonnet 4.6**: Model ID `claude-sonnet-4-6-v1`, accessed via AWS Bedrock Converse API.
- **GPT-5.1-mini**: Model ID `gpt-5.1-mini`, tested on LLM speedrun but failed to produce trustworthy results: the model could not correctly implement the nats-to-

BPB conversion, could not reliably diagnose environment errors, and exhausted its 50-experiment budget on runs with broken evaluation. The Strategist self-diagnosed the problem in the playbook but only after the budget was spent.

API parameters. All default temperature and sampling settings were used per provider.

Hardware. All experiments ran on a single node with 4× NVIDIA H100 NVL 80 GB GPUs (sm_90, 3.35 TB/s HBM3 bandwidth), CUDA 12.6, PyTorch 2.9.1+cu126. Time limits: 1,200 s (20 minutes) for LLM speedrun, 7,200 s (2 hours) for traffic and CUDA kernels.