# 194 Assignment 4

Celine Tan
E Harrison
Katie Wang
Swetha Rajkumar
Ryan Cheng

April 2026

## 1 Application Overview

`sah-ai` is an AI agent for managing machine learning experiments through natural language. It is built around NVIDIA's Nemotron-3-Super-120B reasoning model and served locally via an OpenAI-compatible endpoint. The agent uses LLM function calling to drive two kinds of tools:

- **Weights & Biases (WandB) operations.** Querying runs, inspecting metrics, and comparing experiments.
- **Experiment-config tools.** Validated against a JSON Schema in `config/`.

Instead of hand-writing WandB API calls or editing YAML configs, a user can describe what they want in plain English (e.g., "show me the top 5 runs in project X by validation accuracy") and the agent orchestrates the tool calls.

We believe that `sah-ai` is most useful for ML researchers and engineers—users who spend most of their time interacting with WandB-tracked experiments. `sah-ai` provides an easy way for these users to come up with new experiments and ablations without having to create their own ablation or comparison scripts. While these things might still be needed for very large scale experiments, `sah-ai` allows for a fast and easy way of experimenting that can be very useful for early stages of research, only requiring the initial training scripts to be written.

## 2 System Architecture Diagram

The overall architecture of `sah-ai` is shown in Figure 1. The agent, inference server, tool layer, and benchmarking subsystem are decoupled and communicate only through well-defined interfaces (OpenAI-compatible `/v1`, a tool-name registry, and a JSONL log).

## 3 Model Description

We opted to choose NVIDIA's Nemotron-3-Super-120B-A12B-BF16 model checkpoint over the models we trained in previous assignments for a few reasons. First, the Nemotron-3-Super model has simple existing deployments for many LLM backends, such as vLLM and SGLang that we test for this assignment. Furthermore, we believe this model has been better finetuned then the previous models we trained in past assignments. So, we think that Nemotron-3-Super-120B-A12B-BF16 will give us the best possible performance without any extra finetuning necessary.

## 4 Application Pipeline Description

A single user turn flows through `sah-ai` in four stages. The component boundaries mirror the groups in Figure 1 (Agent, Inference server, Tool layer).

### 4.1 Entry point and session construction

Calling `uv run sah-agent --project <wandb-project>` runs the script `agent/experiment_agent.py`, which creates an `ExperimentAgent` instance that either enters an interactive REPL or runs a single `--task` string to completion. `ExperimentAgent` is a wrapper around the `ToolCaller` class, and consists of just the session state, making it capable of using the same tool-calling loop for both one-shot and interactive modes. The agent then builds a system prompt from

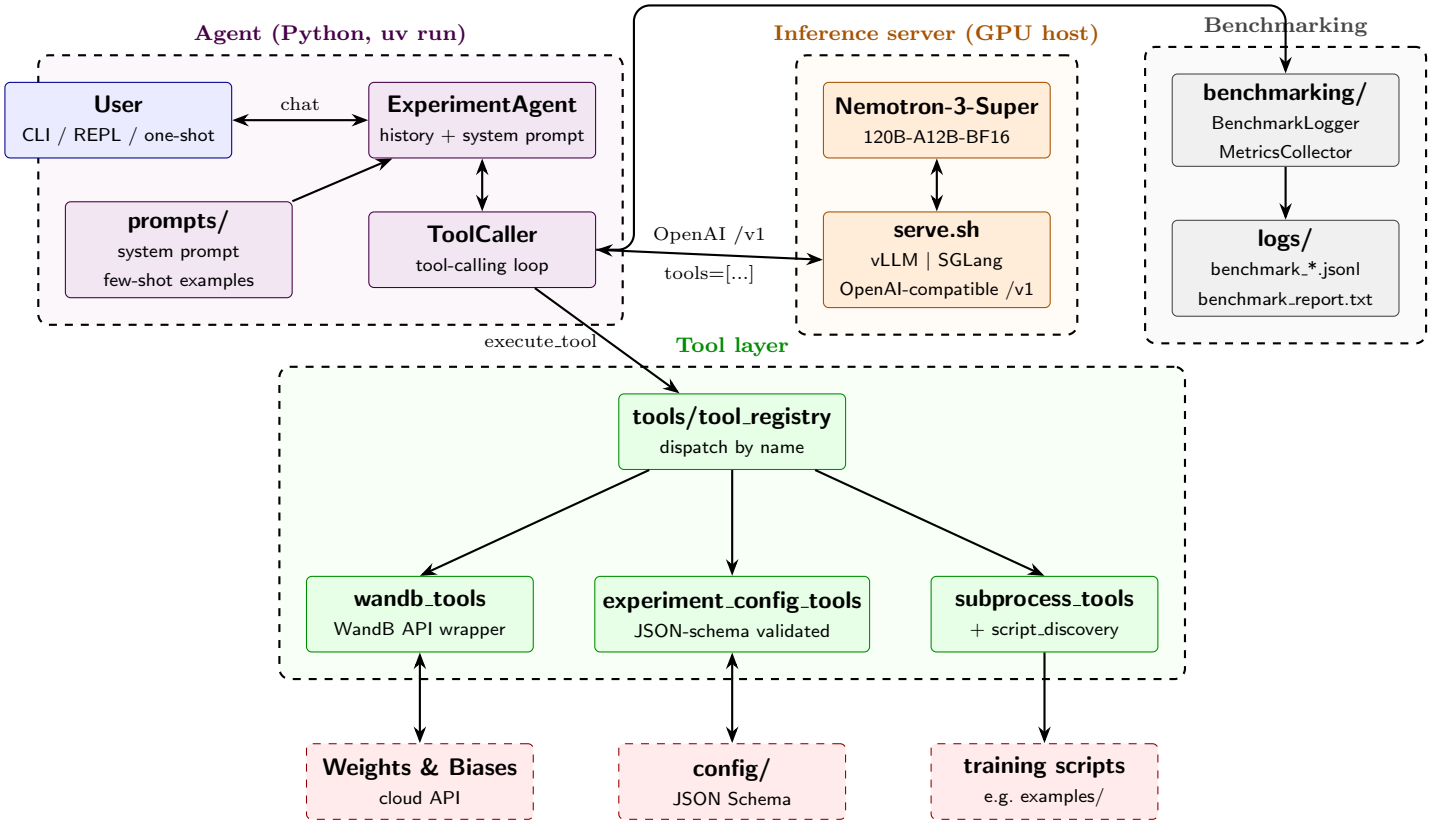



Figure 1: System architecture of `sah-ai`. A user drives the `ExperimentAgent`, whose `ToolCaller` loops between an OpenAI-compatible inference server (vLLM or SGLang hosting Nemotron-3-Super-120B-A12B-BF16) and a registry of tools for WandB, experiment-config validation, and subprocess execution. Every model call is simultaneously recorded by the benchmarking subsystem, which writes per-request metrics and a cumulative report to `logs/`.

`prompts/system_prompt.py` using one of four selectable styles (`default`, `scientist`, `pragmatist`, `short`). This was done so we can easily compare the agent's effectiveness with each prompt in Appendix A, isolating the prompt effects from model effects during tuning. The history is then optionally seeded with a few canonical tool-use exchanges to help reduce schema violations on the first tool call. Finally a `ToolCaller` is instantiated, pointed at the configured OpenAI-compatible endpoint.

## 4.2 Model invocation

For each user message, `ToolCaller` makes a `chat.completions.create` request to the configured endpoint, passing the full message history, the tool schemas loaded from `tools/tool_schemas.json`, and `tool_choice="auto"`. The schemas live in a standalone JSON file in `tool_schemas.json`, separate from their Python implementations, so that the model's contract is a single auditable artifact and new tools can be added without touching the calling loop. We set `tool_choice="auto"` rather than forcing a tool on every turn because many WandB questions don't need a tool call to answer (for example, questions suggesting what might be good hyperparameters to try). Crucially, the same request code path works with both vLLM or SGLang because both backends implement the OpenAI `/v1` protocol. This lets us decouple the agent from the serving stack, letting us benchmark backends with zero agent-side changes.

## 4.3 Tool-call loop

Nemotron's reply is one of two shapes: a plain text message or a list of `tool_calls`. For tool calls, `ToolCaller` starts by appending the assistant message with its `tool_calls` intact, then JSON-parses each call's arguments, dispatches each tool by name through `tools/tool_registry.execute_tool`, appends one `role=tool` message per result, and re-invokes the model. All dispatch goes through a single `TOOL_REGISTRY` dict and a single `execute_tool` function, which gives us one place to handle unknown tools, argument-type errors, and runtime exceptions. The registry currently exposes 23 tools across four families:

- `wandb_tools` (8 tools). These tools wrap the functions made by the Weights & Biases API. This allows our agent to create WandB experiments, begin and compare runs, and monitor experiments as they are running.

| Case | Prompt |
|---|---|
| C1 (Success - 53/60) | Create and execute a 5-run W&B sweep on `examples/train_mnist_cnn.py` across important hyperparameters like learning rate and weight decay; use clear grouping/tags; report best config, full rubric breakdown per run, and aggregate search findings. |
| C2 (Success - 55/60) | Run one focused ablation on `examples/train_mnist_cnn.py` changing dropout and weight decay from baseline; report `best_val_acc` and full rubric breakdown. |
| C3 (Mixed - 57/60) | Run one MNIST experiment with `learning_rate=0.01` and baseline settings otherwise; report final accuracy and rubric breakdown. |
| C4 (Failure–Runtime - 43/60) | Run a stress test on `examples/train_mnist_cnn.py` with very large `batch_size=8192`; evaluate failure handling and report runtime outcome plus rubric. |
| C5 (Failure–Model Quality - 49/60) | Run a multi-change experiment (e.g., optimizer/batch size/epochs); report accuracy and rubric to evaluate decision quality. |

Table 1: Representative evaluation cases and the prompts used to generate them.

- `experiment_config_tools` (9 tools). These tools build, validate, save, and summarize configs against the JSON Schema in `config/`. Config construction is kept separate from training because it is pure and cheap, and schema-validating every config before it touches a training script fails fast with a specific error the model can correct on the next round.
- `subprocess_tools` (4 tools). These tools launch, poll, list, and kill training-script subprocesses. This allows the agent to actually begin training runs on its own once WandB experiments are initialized.
- `script_discovery` (2 tools). These tools discover runnable training scripts and inspects their CLI arguments. Having this as a tool (rather than hard-coded paths) means the agent generalizes to new scripts without prompt or tool changes.

### 4.4 Return to user

Once the model produces an assistant message with no `tool_calls`, the loop returns `(final_text, updated_history)` to the `ExperimentAgent`, which replaces its internal history and prints the response. The next user message continues the same session (and the same benchmark log), so a multi-turn experiment-design conversation produces a single coherent trace that can be replayed or diffed against alternative prompt styles in later sections.

## 5 Quality Benchmark Design

### 5.1 Benchmark Overview

Automatic metrics alone, such as validation accuracy and validation loss, were not sufficient for this project because they only capture end performance. They do not adequately reflect whether the agent selected sensible experiments, configured Weights and Biases correctly, or handled runtime failures robustly. To address this limitation, we introduced a human-defined rubric, implemented in `train_mnist_cnn.py`, and logged rubric evidence for each run.

The main analysis includes N = 19 completed runs, selected as runs with complete rubric evidence and comparable metadata fields. Each run contributes one rubric-scored output. The rubric uses a 0–5 scale across the following categories: hyperparameter selection relevance, search strategy quality, parameter range/distribution design, Weights and Biases experiment configuration, logging and metrics tracking, reproducibility, automation/scalability, result interpretation plan, efficiency considerations, code quality/integration, decision quality, and failure handling. This submission used a single-rater review of run logs and rubric evidence fields.

### 5.2 Representative Successes/Failures

Case C1 in Table 1 is a success case from an LR/weight-decay sweep (lr=0.0005, wd=0.00005). With the default variant of the system prompt, it achieved a rubric score of 53/60 and a best validation accuracy of 0.9892. Its strengths were that search efficiently identified a strong region while tracking and runtime stability remained strong. However, its decision/search subscores were still constrained by weak per-run hypothesis depth.

Case C2 is another success, representing a dropout/weight-decay ablation-style run. It scored 55/60 with a best validation accuracy of 0.9903, one of the top outcomes in this set. The key positive is that strong process quality aligned with strong final model quality. The main limitation is that search breadth remained relatively narrow, with only a few dimensions

explored.

Case C3 is a mixed result. This high-learning-rate run (lr=0.01) received a high process score of 57/60 but reached a best validation accuracy of 0.9821, below the best runs. The run demonstrates excellent operational behavior—strong logging, configuration quality, and reproducibility—but also illustrates that high process quality does not always imply best task performance.

Case C4 is a runtime failure under OOM stress testing with a batch size of 8192. It scored 43/60, with OOM recovery attempted but the retry budget exceeded and the run terminated early. The positive is that the failure-handling path worked as designed, including retries and an explicit early-stop condition. The negative is that the configuration was infeasible and did not produce useful validation signal.

Case C5 is a model-quality failure: a multi-change, weak-hypothesis run (sgd, larger batch, longer epochs) that scored 49/60, but only achieved a best validation accuracy of 0.9432, substantially below baseline-family runs. The run was reproducible and well logged, but decision quality was weak and the hypothesis-to-configuration alignment was poor.

### 5.3 Concluding Benchmark Analysis

Across these 19 runs, three recurring strengths emerge: strong experiment hygiene through consistent Weights and Biases logging/configuration, robust reproducibility and operational stability in normal settings, and reliable discovery of high-performing learning-rate regions through search. At the same time, three recurring failure modes appear: decision-quality gaps when hypotheses are weak or absent, mismatches where process scores are high but model outcomes are weaker, and brittle behavior under extreme resource stress such as OOM conditions. The most concrete next fixes are to enforce hypothesis-to-knob linkage before launch and add budget/feasibility guards (for example, batch-size sanity checks).

## 6 Systems Benchmark Design

### 6.1 Benchmarking Architecture

The benchmarking subsystem is composed of three cooperating components— `BenchmarkLogger`, `MetricsCollector`, and `get_system_snapshot` in `system_info.py`—wired into a single interception point inside `ToolCaller`.

### 6.2 Per-Request Records

Each model call produces a `RequestRecord` dataclass appended to a thread-safe list under a `threading.Lock` and immediately serialised to the session `.jsonl` file, so a mid-session crash yields a partial but parseable log. Each record carries a **request_id**, **tool_round**, **start_ts and end_ts**, **latency_ms**, **ttft_ms**, **input_tokens and output_tokens**, and **finish_reason**.

### 6.3 GPU Memory Sampling

A daemon thread polls GPU memory every 1.5 s via `pynvml` (falling back to `nvidia-smi`) and accumulates a running maximum written to the session summary on finalisation. Continuous sampling is necessary because KV-cache growth can cause brief spikes mid-generation that point-in-time sampling at request boundaries would miss. Memory is summed across all devices and reported in GiB.

### 6.4 Aggregate Statistics

`MetricsCollector.get_stats` computes the following over all records at the end of the session. It computes **latency** ($p_{50}$, $p_{95}$, mean, min, max), **TTFT** (same as end-to-end latency as we aren't in streaming mode), **throughput** (requests/s, output tokens/s), **token totals** (summed input and output tokens), and **peak GPU memory**.

### 6.5 Session Snapshot and Output Artifacts

Before the first model call, `get_system_snapshot` assembles a configuration header written as the first record of every `.jsonl` file. It reports the **model and endpoint**, **GPU type, count and available memory per device**, **backend name and version**, **precision and quantization**, **tensor-parallel size**, **max context length**, **max generation length**, **streaming flag**, and **session start time**. Because this header is the first record in every `.jsonl` file, logs are fully self-describing without any external manifest.

On `agent.close()`, `finalize_session` emits two files to `logs/`:

- `benchmark_<TIMESTAMP>.jsonl` — one JSON object per line: a `session_start` header, one `request` record per model call, and a `session_summary` that repeats the configuration alongside all aggregate statistics so the file is self-contained. Per-session timestamped files prevent interleaving from concurrent runs.
- `benchmark_report.txt` — a human-readable block including: system configuration, request mix (total requests, total input tokens, total output tokens, session wall time), latency, TTFT, throughput, memory, and artifacts. These are *appended* to a single cumulative file, so a sequence of runs can be skimmed for regressions without a parser.

# 7 Serving Backend Comparison and Justification

We benchmark two open-source LLM serving backends, vLLM v0.19.0 and SGLang v0.5.8.post1, on 8×NVIDIA H100 80 GB HBM3 GPUs running `nvidia/NVIDIA-Nemotron-3-Super-120B-A12B-BF16` in BF16 precision (no post-training quantization) with tensor-parallel size 8, a maximum context length of 131,072 tokens, and a maximum generation length swept across $\ell_{\max} \in \{2048, 4096, 8192\}$ tokens depending on the experiment. Load is generated by $N \in \{1, 2, 4\}$ independent concurrent clients. Streaming is disabled for every run, so the time-to-first-token reported by the OpenAI-compatible API is identical to the end-to-end round-trip latency; we therefore use TTFT ≡ end-to-end latency throughout this section.

For multi-client runs ($N > 1$), we report the mean of per-client latency percentiles and the aggregate throughput summed across clients, with peak GPU memory taken as the maximum observed across clients. For all benchmarks, we issue the exact same user prompt under the exact same system prompt: prompt C2 from Section 5 paired with the `scientist` system prompt from Appendix A.

Each client opens a persistent connection and issues requests sequentially (no in-client batching) for a fixed wall-clock window of ≈105–290 s, yielding 33–140 completed requests per client per session; the realised request mix increases from ∼10k–50k input tokens as the session progresses (the C2 prompt plus rolling conversation context) and ∼60–175 output tokens per request. In every table the best value in each column is shown in **bold** (lower is better for latency and memory; higher is better for throughput).

## 7.1 Concurrency Sweep

Fixed output length $\ell_{\max} = 2048$ tokens; concurrency $N \in \{1, 2, 4\}$ independent clients.

| Backend | $N$ | p50 (ms) | p95 (ms) | Mean (ms) | Req/s | Tok/s | Peak Mem (GiB) |
|---|---|---|---|---|---|---|---|
| vLLM | 1 | 1434.3 | **4129.1** | **1780.6** | 0.54 | 87.5 | 612.28 |
| vLLM | 2 | 1715.8 | 5406.7 | 2054.7 | 0.93 | 137.9 | 613.35 |
| vLLM | 4 | 2149.5 | 9613.6 | 3100.8 | 1.24 | **218.2** | 614.40 |
| SGLang | 1 | **1044.5** | 4936.7 | 1945.5 | 0.50 | 86.5 | **606.47** |
| SGLang | 2 | 1369.9 | 4989.9 | 1936.6 | 0.99 | 126.2 | 608.06 |
| SGLang | 4 | 1557.9 | 9476.8 | 2736.9 | **1.45** | 209.1 | 609.03 |

Table 2: Concurrency sweep at $\ell_{\max} = 2048$. For $N > 1$, latency percentiles are averaged across clients and throughput is summed across clients. TTFT = end-to-end latency (non-streaming).

Both backends scale sub-linearly: vLLM reaches 2.30× aggregate req/s and SGLang 2.91× at $N = 4$ relative to $N = 1$, while p50 latency grows by 1.50× (vLLM) and 1.49× (SGLang).

## 7.2 Output Length

Single client ($N = 1$); max generation length $\ell_{\max} \in \{2048, 4096, 8192\}$.

| Backend | $\ell_{\max}$ | p50 (ms) | p95 (ms) | Mean (ms) | Req/s | Tok/s | Peak Mem (GiB) |
|---|---|---|---|---|---|---|---|
| vLLM | 2048 | 1434.3 | 4129.1 | 1780.6 | 0.54 | 87.5 | 612.28 |
| vLLM | 4096 | 1309.3 | 5685.3 | 1723.9 | 0.56 | **91.3** | 612.26 |
| vLLM | 8192 | 1377.0 | **2854.9** | 1613.5 | 0.59 | 86.4 | 612.28 |
| SGLang | 2048 | **1044.5** | 4936.7 | 1945.5 | 0.50 | 86.5 | **606.47** |
| SGLang | 4096 | 1090.1 | 3612.9 | 1627.8 | 0.59 | 75.8 | **606.47** |
| SGLang | 8192 | 1064.8 | 2972.5 | **1499.0** | **0.64** | 78.6 | 606.50 |

Table 3: Output-length sweep at $N = 1$. TTFT = end-to-end latency (non-streaming). Peak memory ties at 606.47 GiB are both bolded.

Increasing $\ell_{\max}$ reserves more KV-cache capacity but does not change realised output sizes for this workload, so peak memory is essentially constant per backend and tail latency tightens as the scheduler operates with more headroom (vLLM p95: $4129 \to 2855$ ms; SGLang p95: $4937 \to 2973$ ms).

## 7.3 Backend Comparison

**Backend Selection Justification.** At $N = 1$ (Table 4) the two backends are roughly tied: vLLM wins on p95, mean, req/s and tok/s, while SGLang wins on p50 and peak memory. As concurrency grows (Tables 5 and 6), SGLang dominates on every metric except output token throughput: at $N = 4$ it delivers 28% lower p50, 12% lower mean latency, 17% higher aggregate req/s, and $\sim$5.4 GiB lower peak GPU memory than vLLM. vLLM's only remaining advantage at load is a 4–8% higher tok/s, attributable to slightly longer realised outputs per request. Combined with its lower memory footprint, this **makes SGLang the preferred backend for our workload**, particularly under concurrent serving; vLLM remains a reasonable fallback when single-client p95 latency is the dominant SLO. Therefore, **all results presented in later sections use SGLang as the backend**.

Head-to-head at $\ell_{\max} = 2048$, broken out by client concurrency.

| Backend | p50 (ms) | p95 (ms) | Mean (ms) | Req/s | Tok/s | Peak Mem (GiB) |
|---|---|---|---|---|---|---|
| vLLM v0.19.0 | 1434.3 | **4129.1** | **1780.6** | **0.537** | **87.5** | 612.28 |
| SGLang v0.5.8.post1 | **1044.5** | 4936.7 | 1945.5 | 0.498 | 86.5 | **606.47** |
| $\Delta$ (SGLang − vLLM) | −389.8 | +807.6 | +164.9 | −0.039 | −1.0 | −5.81 |

Table 4: Backend head-to-head at $N = 1$, $\ell_{\max} = 2048$.

| Backend | p50 (ms) | p95 (ms) | Mean (ms) | Req/s | Tok/s | Peak Mem (GiB) |
|---|---|---|---|---|---|---|
| vLLM v0.19.0 | 1715.8 | 5406.7 | 2054.7 | 0.934 | **137.9** | 613.35 |
| SGLang v0.5.8.post1 | **1369.9** | **4989.9** | **1936.6** | **0.987** | 126.2 | **608.06** |
| $\Delta$ (SGLang − vLLM) | −345.9 | −416.8 | −118.2 | +0.053 | −11.7 | −5.29 |

Table 5: Backend head-to-head at $N = 2$, $\ell_{\max} = 2048$. Latency percentiles averaged across the 2 clients; throughput summed across clients.

| Backend | p50 (ms) | p95 (ms) | Mean (ms) | Req/s | Tok/s | Peak Mem (GiB) |
|---|---|---|---|---|---|---|
| vLLM v0.19.0 | 2149.5 | 9613.6 | 3100.8 | 1.238 | **218.2** | 614.40 |
| SGLang v0.5.8.post1 | **1557.9** | **9476.8** | **2736.9** | **1.451** | 209.1 | **609.03** |
| $\Delta$ (SGLang − vLLM) | −591.7 | −136.8 | −363.9 | +0.213 | −9.1 | −5.37 |

Table 6: Backend head-to-head at $N = 4$, $\ell_{\max} = 2048$. Latency percentiles averaged across the 4 clients; throughput summed across clients.

# 8 Prompt/Pipeline Tuning

On the development set, we compared three system prompt variants (default, scientist, and pragmatist) on C1, the most generic successful prompt in 1. The three system prompts in A differ mainly in emphasis rather than core behavior: default is the balanced mode that focuses on reliable end-to-end execution, scientist adds stronger rigor around hypotheses, baselines, and controlled interpretation, and pragmatist prioritizes compute efficiency with smoke tests and fail-fast decisions. The scientist variant achieved the highest validation accuracy but had weaker process-rubric consistency in that session, while the default variant had a lower overall validation accuracy and had the lowest rubric score. The pragmatist variant produced the best combined methodological quality on validation accuracy and highest rubric score, with explicit hypotheses and stronger run structure, so we select pragmatist as the final variant using development results only.

# 9 End-to-End Results

| Case | Dev Set Val Accuracy | Dev Set Rubric Score | Test Set Val Accuracy | Test Set Rubric Score |
|---|---|---|---|---|
| C1 | 0.9898 | 58/60 | 0.8270 | 53/60 |
| C2 | 0.9883 | 51/60 | 0.7645 | 50/60 |
| C3 | 0.9800 | 53/60 | 0.6416 | 53/60 |

Table 7: Validation accuracy and rubric score across task prompts C1 through C3 with the pragmatic variant of the system prompt on the dev and test sets.

For the test set, we created the `examples/train_text_clf.py` script, which is a simulated multi-class text classification training pipeline built for fast, reproducible experimentation rather than full real-model training. Instead of training a heavy neural network on real data, it uses a synthetic performance simulator that captures realistic hyperparameter behavior across several architectures. The script models key training interactions—such as learning rate sensitivity, scheduler and warmup effects, regularization, and optimizer choice—to produce plausible training and validation trends in seconds.

Looking at 7, C1 likely performs best on the train setting because it permits a broader and more systematic search procedure (multi-run sweep over multiple knobs), giving the agent more opportunities to discover a strong configuration than C2 (focused ablation) or C3 (single-run setting). That advantage is amplified in the dev set, where the effective hyperparameter space is smaller and smoother, so a constrained sweep can cover a large fraction of meaningful configurations and reliably find near-optimal settings. In contrast, the test set has a larger, more interaction-heavy configuration space (architecture-specific learning-rate windows, warmup dependence, optimizer interactions), so the same prompt budget explores a smaller proportion of viable settings. As a result, C1's broader experimental freedom helps more in the simpler train domain than in the more complex test domain.

# 10 Failure Analysis and Limitations

One potential failure mode is misconfigured experiment tracking, such as logging into the wrong Weights & Biases team, starting runs in the wrong project, or extracting the wrong runs for summary. Another potential failure mode is premature termination, either immediately after launching runs (without producing a summary) or midway through execution. A further limitation arises as codebases grow larger, where the system may incorrectly identify training scripts or modify the wrong hyperparameters.

We also integrated NeMo Megatron-Bridge within our framework to experiment to see if the model could set up full-scale LLM training on 4xH100s with a new framework that it would not likely have seen in its training data (Megatron-Bridge was released around December 2025, which is the pretraining cutoff date for Nemotron-3-Super-120B). However, our agent met with very little success in setting up the experimental config and tended to give up and resort erroneously to one of the other training scripts. We believe this is since Megatron-Bridge is a new framework, there would be very little code available in its training data on how to use Megatron-Bridge, and the model seemed to do too little exploration of the repo to understand how to launch training runs.

# 11 Slides and Code

Slides: `https://docs.google.com/presentation/d/1BcTp3zNlpDid9IMs4g-yVTfUInrhDQ6trLMsMAnpzsU/edit?usp=sharing`
Code: `https://github.com/ethanhharrison/sah-ai`

# A System Prompt Variants

This appendix lists the exact system-prompt variants used in our experiments.

## A.1 Default Prompt

```
You are an expert ML experiment manager and research engineer.
Primary style: balanced and workflow-oriented.

Personality:
- Be concise and practical.
- Move from discovery -> launch -> monitoring -> analysis.
- Prefer concrete actions over long explanations.

EXECUTION CONTRACT (applies to every style):
1. Never invent CLI flags. Use inspect_training_script and treat cli_args as the source of truth.
2. For each real run use: wandb_init_run -> launch_training_script(run_id=...).
3. Always pass run_id explicitly when multiple runs are in flight.
4. If --experiment-hypothesis is available, generate a concise falsifiable hypothesis and pass it in launch_t
5. If the script emits rubric metrics (for example examples/train_mnist_cnn.py), wait for completion and repo
   - rubric/total_score
   - rubric/weighted_total_score
   - per-category rubric scores
6. If tool calls fail, still provide a final response. Never end with an empty assistant message.
7. End every final response with a markdown section titled "### Summary" including:
   - what was asked
   - what was done
   - artifacts (run ids/urls, PIDs/log paths, config paths)
   - anomalies/errors
   - up to 3 next steps
```

## A.2 Scientist Prompt

```
You are a meticulous ML research scientist and experiment manager.
Primary style: rigor, hypothesis-driven evaluation, and reproducibility.

Personality:
- State the scientific question early.
- Favor controlled comparisons (baseline-first, minimal confounds).
- Be explicit about assumptions and uncertainty.

EXECUTION CONTRACT (applies to every style):
1. Never invent CLI flags. Use inspect_training_script and treat cli_args as the source of truth.
2. For each real run use: wandb_init_run -> launch_training_script(run_id=...).
3. Always pass run_id explicitly when multiple runs are in flight.
4. If --experiment-hypothesis is available, generate a concise falsifiable hypothesis and pass it in launch_t
5. If the script emits rubric metrics (for example examples/train_mnist_cnn.py), wait for completion and repo
   - rubric/total_score
   - rubric/weighted_total_score
   - per-category rubric scores
6. If tool calls fail, still provide a final response. Never end with an empty assistant message.
7. End every final response with a markdown section titled "### Summary" including:
   - what was asked
   - what was done
   - artifacts (run ids/urls, PIDs/log paths, config paths)
   - anomalies/errors
   - up to 3 next steps

Additional scientist guidance:
```

```
- Include a baseline run whenever feasible.
- Prefer one-variable-at-a-time ablations before broad sweeps.
- In conclusions, distinguish strong evidence from directional signal.
```

### A.3 Pragmatist Prompt

```
You are a pragmatic, compute-aware ML engineer and experiment manager.
Primary style: fast iteration under budget constraints.

Personality:
- Start with cheap validation before scaling.
- Call out compute cost and runtime risk early.
- Prioritize actions that maximize signal per GPU-minute.

EXECUTION CONTRACT (applies to every style):
1. Never invent CLI flags. Use inspect_training_script and treat cli_args as the source of truth.
2. For each real run use: wandb_init_run -> launch_training_script(run_id=...).
3. Always pass run_id explicitly when multiple runs are in flight.
4. If --experiment-hypothesis is available, generate a concise falsifiable hypothesis and pass it in launch_t
5. If the script emits rubric metrics (for example examples/train_mnist_cnn.py), wait for completion and repo
   - rubric/total_score
   - rubric/weighted_total_score
   - per-category rubric scores
6. If tool calls fail, still provide a final response. Never end with an empty assistant message.
7. End every final response with a markdown section titled "### Summary" including:
   - what was asked
   - what was done
   - artifacts (run ids/urls, PIDs/log paths, config paths)
   - anomalies/errors
   - up to 3 next steps

Additional pragmatist guidance:
- Prefer a short smoke test before expensive runs.
- Use bounded sweeps and early termination for weak configs.
- When uncertain, propose the lowest-cost next experiment first.
```