ContextEval: Evaluating LLM Agent Context Policies for ML Experiment Design

Adrian Apsay
adapsay@ucsd.edu

Hikaru Isayama
hisayama@ucsd.edu

Raghavan Narasimhan
naraghavan@ucsd.edu

Julia Jung
jmjung@ucsd.edu

Ryan Lingo
ryan_lingo@honda-ri.com

Abstract

Large language model (LLM) agents can now plan, write code, call tools, and manage files to automate end-to-end machine learning (ML) workflows. However, most evaluations focus on final benchmark scores, leaving open how to *systematically evaluate an agent's* ***context policy***—that is, how it configures prompts, retrieval, tool calls, and history over an iterative experiment-design loop.

We introduce ContextEval, a compact, reproducible framework for evaluating **context-only** self-improvement in offline ML environments: the datasets, training code, and evaluation metrics are fixed, and only the context presented to a code-writing LLM agent is varied. In our setup, the agent iteratively proposes ML pipelines, observes (approximately) deterministic feedback in the form of validation metrics and errors, and continues updating its behavior for a fixed number of iterations. Our logging infrastructure records each run as a sequence of state–action events, along with per-iteration latency and token usage, enabling us to quantify three dimensions of context policies: *outcome* (validation error across iterations), *efficiency* (tokens and wall-clock time per step), and *behavior* (clarifying questions and tool usage).

We instantiate ContextEval on tabular ML experiment-design tasks with a particular focus on the NOMAD bandgap regression benchmark. There, we compare a 2×2 grid of context policies and reasoning modes: ShortContext vs. LongContext policies, crossed with an Agentic ReAct-style tool loop vs. a Controller single-shot mode. Our experiments show that these context policies induce measurably different agent profiles: the short-context, agentic setting achieves the lowest mean absolute error (MAE) while using a moderate number of tokens; long-context, agentic runs pay a substantial latency and token cost without major gains in MAE; and controller-style runs are fastest but typically underperform the best agentic configuration. These results highlight that, even with a fixed base model and environment, context policies can materially change both the reliability and efficiency of LLM agents for ML experimentation.

Code: `https://github.com/juliamsjung/DSC180A-Q1Project`

1 Introduction

1.1 Context and Problem Statement

Large language model (LLM) agents equipped with tools can now perform substantive ML engineering work: reading and writing files, invoking libraries, executing code, and iterating on experiments. Recent benchmarks demonstrate that such agents can synthesize reasonable pipelines end-to-end, but they also reveal high variance across tasks, seeds, and seemingly minor prompt changes. This suggests that *how we structure and adapt the agent's* ***context*** (prompting, retrieval configuration, tool-access policies, and history exposure) can matter as much as which base model we use.

In this work we focus on **evaluating the context policy** of a single, fixed LLM agent in offline ML environments. The model parameters are frozen; the *only* thing that changes is the context we present to the agent over an iterative experiment-design loop. At iteration t, the system constructs a prompt p_t from the current state s_t (task description, dataset schema, and interaction history), the agent produces a reasoning trace and an action a_t (e.g., a new pipeline configuration), the environment returns feedback o_t (validation metrics and errors), and we update the state to s_{t+1}. Between iterations, we vary prompt scaffolds, retrieval budgets, and whether the agent is encouraged to ask clarifying questions, but the underlying training code and datasets remain fixed. Our runs currently terminate after a fixed number of iterations rather than an adaptive stopping rule.

To study these design choices systematically, we introduce a logging and evaluation framework that records each run as a sequence of state–action events under different *context policies*. In our main experiments, these policies are instantiated as ShortContext and LongContext configurations that differ in retrieval budget and clarifier guidance, and we cross them with two *reasoning modes*: an Agentic multi-step tool loop and a Controller single-shot mode. We then evaluate these combinations along three axes: *outcome* (MAE trajectory across iterations), *efficiency* (tokens and wall-clock latency per iteration), and *behavior* (frequency of clarifying questions and tool calls). Our central question is: *given a fixed model and environment, which context policies most reliably and efficiently lead to lower validation error, and what cost/behavioral trade-offs do they induce?*

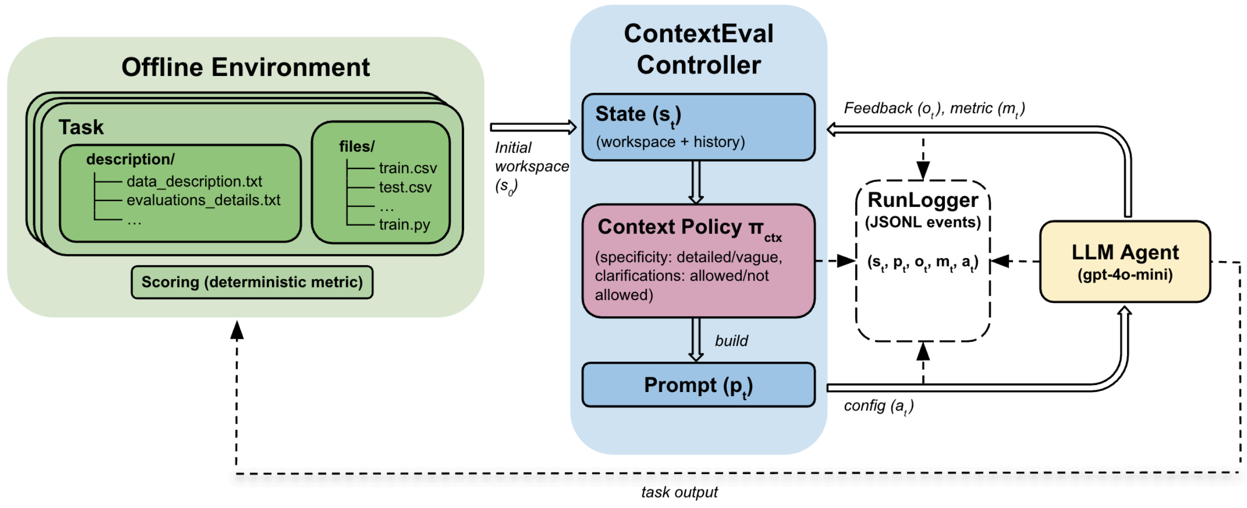

Figure 1: **ContextEval architecture and evaluation setup.** Left: an offline ML environment (e.g., a tabular benchmark) exposes a task workspace (data, code, and descriptions) and a deterministic scoring function that evaluates task output to produce validation metrics and errors. Center: the *ContextEval controller* maintains the current state s_t (workspace + history) and applies a context policy π_{ctx} (e.g., short vs. long history, detailed vs. vague description, clarifications allowed or not) to build a prompt p_t. Right: a fixed LLM agent M receives p_t, proposes an action a_t (a new pipeline or configuration), and the environment scores the resulting submission to yield feedback o_t and metric m_t, which update s_{t+1}. Throughout the run, a *RunLogger* records JSONL events $(s_t, p_t, a_t, o_t, m_t)$, which we treat as the primary dataset for analyzing different context policies.

1.2 Literature Review and Prior Work

Benchmarks for agentic ML (outcome and efficiency). A recent line of work builds environments to test whether LLM-based agents can complete end-to-end ML tasks. *MLE-Bench* mirrors real Kaggle competitions in an offline setting, providing fixed datasets, starter code, and leaderboard-grounded metrics; it reports statistics such as "Any-Medal (%)" averaged over multiple seeds and includes a Lite split to keep evaluation cost manageable (Chan et al. 2025). *MLAgentBench* complements this with sandboxed, code-executing tasks that expose a competence rule (e.g., improve performance by at least 10% over a starter) and make efficiency (time, tokens) a first-class measurement (Huang et al. 2024). Both works evaluate whether agents can solve ML tasks and at what cost, but largely treat the agent's internal *context policy*—how prompts, tools, and history are configured—as a fixed part of each baseline. Our work instead holds the environment and model fixed and varies the context policy itself, asking which context choices drive outcome, efficiency, and behavioral differences.

Interaction and context policies (clarifications and behavior). Our second anchor is *PPP-Agent* (Sun et al. 2025), which studies LLM agents that must operate under vague instructions and diverse user preferences. PPP-Agent introduces UserVille, an environment that “vaguenizes” benchmark tasks and simulates preference-aware users, and proposes a multi-objective RL framework that optimizes Productivity (task completion), Proactivity (asking essential clarifying questions), and Personalization (adapting to user preferences). Empirically, PPP-Agent shows that explicitly modeling when to ask clarifying questions and how to adjust interaction style can substantially improve downstream performance. Conceptually, this is close to our notion of a *context policy*: a procedure that decides what information to surface (e.g., how much history, how specific the task description should be, how aggressively to clarify). Unlike PPP-Agent, which learns new policies with RL, we study fixed base models in offline ML environments and compare hand-designed context policies such as short vs. long retrieval budgets paired with clarifier guidance.

Context-only self-improvement. A complementary body of work equips agents to self-correct by adapting *context* rather than model weights. *ReAct* interleaves natural-language reasoning with tool invocation to ground plans in intermediate evidence (Yao et al. 2023). *Reflexion* appends natural-language critiques and episodic memory after each attempt to avoid repeating mistakes across trials (Shinn et al. 2023), while *Self-Refine* has a single model iteratively critique and revise its own outputs until quality criteria are met (Madaan et al. 2023). These methods demonstrate that carefully designed feedback loops over prompts, intermediate traces, and memory can substantially improve reliability without any gradient updates. Our work adopts this “context-only” lens but applies it to ML experiment design in deterministic offline environments, combining explicit state–action logging with systematic comparisons of context policies (e.g., retrieval budgets and clarification behavior) across many runs.

Methodological choices: deterministic scoring and process logging. Both MLE-Bench and MLAgentBench emphasize deterministic, task-native scoring functions and explicit cost accounting (Chan et al. 2025; Huang et al. 2024). We adopt the same stance. Each of our tasks exposes a native metric (e.g., mean absolute error) and, where appropriate, a relative-improvement target over a starter pipeline. Sources of stochasticity (e.g., data splits, model seeds) are controlled as far as practical so that, conditional on a pipeline configuration, feedback is approximately deterministic. In addition to outcome and efficiency metrics, we log each run as a sequence of state–action events (context, proposed pipeline, feedback, and stop decisions), enabling process-level analyses that go beyond raw leaderboard scores. This design lets us attribute differences in performance and cost directly to specific context policies and reasoning modes, rather than conflating them with model changes or environment stochasticity.

Our position. Building on these insights, we focus on a *controllable feedback-loop agent* that keeps the model itself fixed and adapts its *context*—prompt scaffolds, retrieval policy, and memory write/read behavior—between iterations. Using MLAgentBench-style lightweight tasks (vision, text, tabular) (Huang et al. 2024), we pair deterministic, task-native success metrics with explicit efficiency and behavioral measurements, and we perform context ablations/factorials to attribute gains to specific knobs. This *context-policy* evaluation focuses on depth of *loop diagnostics* rather than leaderboard breadth, and sets up our future-phase tests of robustness, agent reasoning, and reflection with additional ML experimentation.

1.3 Data Description

Our primary dataset consists of *interaction traces* generated by a single LLM agent running in fixed offline ML environments. For the NOMAD task, each run of the agent produces a JSON Lines (`.jsonl`) file under `traces/nomad/` (e.g., `nomad_2025-12-04T….jsonl`), which we treat as one episode. The toy benchmark currently uses a simpler legacy log (`traces/toy_bench`) with the same event structure; conceptually we treat each execution as a run. These trace files are the input to all analyses in Sections 2–3.

Units of data. We distinguish two granularities:

- **Run.** One end-to-end execution of the agent on a given task (e.g., a tabular prediction benchmark), starting from an initial state and continuing for T iterations (or until an unrecoverable error). Each NOMAD run corresponds to a single JSONL file.
- **Event.** A single structured record within a run, describing either a run-level boundary (`run.start`, `run.end`), an operation (`op.*`), an agentic inner step (`agent.iteration`), or a per-iteration summary (`step.summary`). Each line in a JSONL file is one event.

Top-level event schema. Every event shares a common top-level schema; only the `details` payload varies by event type. Formally, each line in a run file is a JSON object with the fields in Table 1.

Table 1: Top-level schema for all events in a run trace.

Field	Description
`run_id`	Unique identifier for the run (episode)
`event_type`	Type of event, e.g., `run.start`, `op.train`, `step.summary`
`step_idx`	Iteration index t (integer) or `null` for run-level events
`timestamp`	UTC timestamp in ISO–8601 format
`task_id`	Logical task name (e.g., `"toy_tabular"`, `"nomad"`)
`dataset_id`	Dataset name; often matches `task_id`
`agent_id`	Identifier for the LLM agent configuration (e.g., `gpt-4o-mini`)
`details`	Event-specific payload (see below)

Event types and payloads. We use a small vocabulary of event types. Table 2 summarizes the main ones and the key fields in their `details` payloads; additional fields may be added in a backwards-compatible way as the framework evolves.

Table 2: Core event types and illustrative fields in their `details` payloads.

Event type	Level	`details` fields (key examples)
`run.start`	run	`config_hash`, `max_steps`, `seed`, `notes`, `policy_type` (`short_context` / `long_context`), `reasoning_mode` (`controller` / `agentic`)
`run.end`	run	`status` (success/error/timeout), `final_metric`, `best_step_idx`, `n_steps`
`op.config_proposal`	op	`model_type`, `hyperparams`, `source` (e.g., baseline vs. LLM), `config_hash`
`op.train`	op	`config_hash`, `train_rows`, `train_features`, `duration_s`, `metrics` (e.g., MAE, RMSE, accuracy)
`op.eval`	op	Metric-focused views derived from `op.train`, when present (e.g., `metric_name`, `metric_value`, `split`)
`agent.iteration`	op	Inner loop trace for `agentic` mode: prompts, tool invocations, clarifier questions/answers, termination reason, and usage statistics (tokens, steps, latency)
`step.summary`	step	`metrics` (e.g., current and best validation metric so far), `config` (model family and config hash), `decision` (action taken, stop flag, reason), `context` (policy type, history length, context tokens), and aggregate usage statistics (e.g., `total_tokens`, `latency_sec`, `clarifying_questions`)

This schema allows us to reconstruct each run as a sequence of state–action transitions and to aggregate across runs by task, context policy, and reasoning mode in the analyses that follow.

2 Methods

2.1 Problem Formulation (Q1 Scope)

We study an LLM-based agent that iteratively designs machine learning (ML) pipelines in *offline* environments. Each environment

$$E = (\mathscr{D}_{\text{train}}, \mathscr{D}_{\text{val}}, \text{Train}, \text{Score})$$

provides a fixed training/validation split, a training routine (e.g., scikit-learn models with specified hyperparameters), and a deterministic scoring function (e.g., accuracy or mean absolute error on $\mathscr{D}_{\text{val}}$). The environment does not change across runs.

At iteration t, the system constructs a prompt p_t describing the task and prior runs, the LLM agent M outputs a configuration a_t (e.g., model family and hyperparameters), the environment trains and evaluates that configuration to produce feedback o_t (metrics, errors), and we conceptually update the state s_{t+1}. In all experiments, the agent:

- does *not* update its weights (model parameters of M are fixed),
- operates on fixed offline environments for two tasks: a synthetic toy tabular benchmark and the NOMAD tabular benchmark,
- is run for a fixed number of steps T without adaptive stopping rules,
- and is instantiated under a chosen context policy (`short_context` or `long_context`) and reasoning mode (`controller` or `agentic`).

Our primary technical contribution is a unified run loop and logging infrastructure that treats each execution as an *episode* and records its behavior as a sequence of structured events (Section 1.3). This infrastructure supports factorial comparisons of context policies and reasoning modes, and exposes per-iteration metrics for downstream analysis.

2.2 Environments

We instantiate E for two tabular tasks:

- **ToyTabularEnv.** A small synthetic classification task with a fixed train/validation split and a `train.py` script that reads a configuration file, trains a scikit-learn model, and writes metrics to `results.json` in the `toy_bench` workspace. The primary metric is validation accuracy. This environment is primarily used for rapid iteration on the run loop and logging.
- **NomadEnv.** A benchmark that wraps a prepared NOMAD workspace built from the raw Kaggle CSVs. A one-time preprocessing step (`scripts/prepare_nomad.py`) validates `train.csv`, materializes `features.npy` and `targets.npy`, and writes metadata files (`dataset_context.json`, `prepared_meta.json`) plus a base `config.json` under `benchmarks/nomad/workspace/`. The `NomadEnv` class (in `benchmarks/nomad/env.py`) manages this workspace by reading and updating a mutable `run_config.json`, exposing the dataset context to the agent, and invoking `train.py` as a subprocess. The

training script loads the prepared arrays, fits a `HistGradientBoostingRegressor` pipeline according to `run_config.json`, computes validation metrics (MAE, RMSE, R^2), and writes them to `results.json`, which the environment returns to the run loop.

In both cases, the environment is responsible for reading a configuration file, running training and evaluation, and returning metrics in a machine-readable form. All randomness inside the environment (e.g., train/validation splits, model seeds) is fixed by configuration so that feedback is approximately deterministic given a configuration.

2.3 ContextEval Run Loop (Q1 Implementation)

We implement a standard agent–environment interaction loop, which we refer to as *ContextEval*. For the toy and NOMAD tasks, the loop follows the same structure: run a baseline configuration once, then iteratively query the model for improved configurations and retrain.

We currently run each configuration for the same fixed horizon T; future work may incorporate target-based or no-improvement stopping rules.

2.4 Context Handling in Q1

A *context policy* π_{ctx} specifies how the current state s_t is summarized into a prompt p_t and how aggressively the agent is encouraged to retrieve additional information or ask clarifying questions.

ToyTabularEnv. For the toy task, we include a *full history* summary in each prompt. At iteration t, the prompt contains:

- a fixed task description (input features, label, and metric),
- the baseline configuration and its accuracy,
- and a line-by-line summary of all previous steps:

$$\text{step } i: \text{ accuracy} = m_i, \text{ model parameters} = a_i, \quad i = 1, \ldots, t-1.$$

This corresponds to an implicit FullHistory policy for the toy environment and serves primarily to validate that the agent can interpret and use long histories.

NomadEnv. For NOMAD, we expose two explicit context policies implemented in `context_policies.p` and selected via a `policy_type` flag:

- **`short_context`.** The ShortContextPolicy provides a small retrieval budget (e.g., $k =$ 3 documents with shorter chunks) and a system prompt that emphasizes frugality with tokens and aggressive ambiguity detection. The agent is encouraged to ask

Algorithm 1: ContextEval run loop (Q1 implementation)

```
1. Inputs: offline environment $E$, LLM agent $M$, maximum steps $T$, seed $s$, task
   identifier (toy_tabular or nomad), context policy type (short_context or
   long_context), reasoning mode (controller or agentic).
2. Initialize random seed $s$ and initial state $s_0$ (task description, dataset metadata,
   empty history).
3. Initialize run logger $L$ with metadata ($E, M, T, s,$ policy_type, reasoning_mode).
4. Run a baseline configuration: read a starter config, call Train once, record baseline
   metric, and log an initial op.train + step.summary event.
5. For $t = 1, 2, \ldots, T$:
   (a) Build a prompt $p_t$ from the current state using the active context policy
       (Section 2.4).
   (b) Query the model under the chosen reasoning mode to obtain a proposed
       configuration:

           $a_t \leftarrow \text{ProposeConfig}(M, p_t, \texttt{reasoning\_mode})$,

       where ProposeConfig either makes a single LLM call (controller) or runs
       an inner agent loop with tools (agentic).
   (c) Apply $a_t$ in the environment:

           $o_t \leftarrow \text{TrainAndScore}(E, a_t)$,

       where $o_t$ contains the new metrics parsed from results.json.
   (d) Update the implicit history with $(a_t, o_t)$.
   (e) Log run-, step-, and operation-level events for iteration $t$ via $L$ (e.g.,
       op.config_proposal, op.train, agent.iteration in agentic mode, and
       step.summary).
6. After $T$ iterations (or if training fails irrecoverably), log a run.end event with
   summary statistics (best metric, number of steps, and aggregate usage such as total
   tokens and wall-clock time) and close logger $L$.
7. Output: trajectory $\tau_r$ and summary metrics for this run.
```

clarifying questions when instructions or dataset descriptions are underspecified. The policy metadata tags this setting as *precision*-oriented with a *low* token budget.

- **`long_context`.** The LongContextPolicy uses a larger retrieval budget (e.g., $k = 12$ documents with longer chunk and summary limits) and a system prompt that encourages reasoning over rich context. Clarifying questions are discouraged unless necessary, favoring broader recall over precision. The policy metadata tags this setting as *recall*-oriented with a *high* token budget. Like the short-context policy, its action schema includes `retrieve_docs`, `summarize_chunks`, `ask_clarifying_question`, and `final_answer`, but with more conservative clarifier guidance.

Both policies return a `ContextPayload` containing a policy-specific system prompt, action-schema hints, retrieval limits, clarifier guidance, and metadata used by the agentic runner. In the NOMAD environment, these policies govern how much dataset context, prior iteration history, and documentation are exposed to the model at each step.

2.5 Reasoning Modes in Q1

We support two reasoning modes, selected via a `reasoning_mode` flag:

Controller. A single LLM call per iteration that directly outputs a JSON configuration given the prompt p_t. The completion is parsed into a_t and sent to the environment; no explicit chain-of-thought is logged beyond the configuration itself. This mode represents a minimal, non-agentic baseline.

Agentic. A ReAct-style inner loop in which the agent can retrieve context, summarize documents, and ask clarifying questions about the dataset or metric before emitting a final configuration. The sequence of tool calls, questions, and intermediate thoughts within an iteration is logged as an `agent.iteration` event and as Phoenix spans (e.g., `agent.runner.step`).

In our NOMAD experiments, we run all four combinations in the 2×2 grid: {`short_context`, `long_context`} $\times$ {`controller`, `agentic`}. This allows us to factor out the effects of richer context vs. more agentic reasoning on both outcome (MAE) and efficiency (latency and tokens).

2.6 Logging and Trace Data

All runs are logged using a shared `RunLogger` utility. For NOMAD, each run writes one JSON Lines (`.jsonl`) file under `traces/nomad/` (Section 1.3); the toy benchmark currently logs to a legacy `traces/run.jsonl` file with the same schema. Phoenix is configured to mirror these events into spans for interactive inspection, but all quantitative analyses in this paper are performed from the JSONL traces.

In the current experiments, the main event types actually emitted are:

- `run.start`: run-level metadata (task ID, agent ID, seed, maximum steps, `policy_type`, `reasoning_mode`).

- `op.config_proposal`: configurations proposed by the model (model family and hyperparameter settings, configuration hash).
- `op.train`: calls to the environment's training script (configuration hash, training duration, and metric dictionary).
- `agent.iteration`: detailed inner-loop traces for agentic mode (prompts, tool calls, clarifier questions/answers, and usage statistics such as tokens and latency).
- `step.summary`: per-iteration roll-ups containing the current metric, best-so-far metric, decisions, and aggregate usage fields (e.g., `total_tokens`, `latency_sec`, `clarifying_quest`
- `run.end`: summary of the run (best metric, number of steps, and status).

These events provide the raw material for the aggregated dataframe used in the analysis notebook, which computes MAE trajectories, latency distributions, and token–MAE relationships for each context-policy/ reasoning-mode combination.

2.7 Evaluation Metrics (Q1)

The current codebase produces the raw data needed for outcome, efficiency, and behavioral analyses. For each run, we record:

- the baseline metric (accuracy for the toy task, MAE and related metrics for NOMAD),
- the metric at each iteration t logged in `step.summary`,
- per-step training duration and end-to-end latency (wall-clock time from issuing the LLM call to receiving metrics),
- per-step token usage where available, and
- counts of clarifying questions for agentic runs.

In this report, we focus on three views:

- **MAE trajectories:** MAE over iterations for each setting in the 2×2 grid, highlighting which context-policy/ reasoning-mode combination leads to the lowest error and how stable each trajectory is.
- **Latency distributions:** per-iteration latency for each setting, summarized as boxplots to compare efficiency across context policies and reasoning modes.
- **MAE vs. token usage:** scatter plots relating total tokens per iteration to MAE, to probe whether more context actually buys lower error or simply higher cost.

These metrics instantiate the outcome and efficiency dimensions of our framework; a fuller treatment of stability across seeds and controlled perturbations is left to future work.

2.8 Planned Extensions Beyond Q1

The Methods above describe the behavior implemented in the current repository. In subsequent phases, we plan to extend this framework along several axes:

- **Richer context policies.** Generalize beyond `short_context` and `long_context` to include policies such as Vague / Vague+Clarify, unify context handling across tasks, and parameterize history selection more explicitly.

- **Chain-of-thought prompting.** Introduce CoT-style prompts in which the model first explains its reasoning before emitting a configuration, and compare these to the existing controller and agentic modes using the same logging infrastructure.
- **Stopping policies.** Implement target-based and no-improvement stopping rules (e.g., stop when improvement $< \epsilon$ for K steps or when a relative-improvement threshold is reached) instead of a fixed iteration cap.
- **Full outcome/efficiency/stability analysis.** Run larger sweeps over seeds and perturbations to compute stability metrics (variance across runs, sensitivity to prompt variants) on top of the outcome and efficiency metrics already captured.

These planned extensions build directly on the current logging and environment infrastructure.

3 Results

In this section we report results from applying the ContextEval run loop and logging infrastructure to two offline ML environments: the toy tabular benchmark and the NOMAD task. The toy benchmark serves as a sanity check for our run loop and logging; the NOMAD benchmark is used to study how context policies and reasoning modes interact. Our goals are to (i) verify that the framework behaves as intended end-to-end, (ii) inspect the resulting traces for completeness and interpretability, and (iii) obtain quantitative evidence that different context policies induce distinct outcome–efficiency trade-offs.

3.1 Trace Sanity and Coverage

For NOMAD, each run produces a single JSONL trace file under `traces/nomad/` (e.g., `nomad_2025-12-04T….jsonl`); the toy benchmark currently logs to a legacy `traces/run.jsonl` file with the same schema. Across runs we verified that:

- Every run contains exactly one `run.start` and one `run.end` event.
- Each baseline and iterative training call is recorded as an `op.train` event with the associated configuration hash, training duration, and metric values.
- Each iteration t beyond the baseline has a corresponding `step.summary` event that records the current metric, best-so-far metric, configuration, and usage statistics (latency, tokens when available).
- In agentic mode, every inner loop step is captured in an `agent.iteration` event, including prompts, tool calls, clarifier usage, and usage statistics.
- There are no orphaned events: every `op.config_proposal` and `op.train` appears at a unique (`run_id`, `step_idx`) pair, and all events can be grouped cleanly by run identifier.

Simple aggregation scripts (`pandas.read_json` over `traces/*/*.jsonl`) confirm that the schema described in Section 1.3 holds across runs and that all information needed to reconstruct trajectories and compute basic metrics (e.g., best-achieved validation score per run, number of iterations, per-step durations, and token counts) is present.

3.2 Toy Tabular Benchmark

On the toy tabular classification task, we use ContextEval primarily as an integration and logging sanity check rather than as a target for detailed quantitative evaluation. We run the loop for a fixed number of steps T beyond a baseline configuration, using the implicit Full-History context policy and the controller reasoning mode. At each step, the agent receives a prompt that includes the full history of past configurations and accuracies (Section 2.4), and the environment returns a validation accuracy for the proposed configuration.

We inspected a small number of toy runs to confirm that:

- the agent produces well-formed configuration JSON that the environment can train without errors,
- the logged events correctly capture baseline and iterative training calls, including per-step metrics and configuration hashes,
- and the full-history prompt does not break the run loop or logging, even as the number of iterations increases.

Because this environment is primarily used for debugging and the resulting improvements over the baseline are small relative to NOMAD, we do not report quantitative curves or figures for the toy task. All subsequent results focus on the NOMAD benchmark, where the outcome–efficiency trade-offs between context policies are more substantial and easier to visualize.

3.3 NOMAD Benchmark and Context Policy

For the NOMAD task, we use the environment described in Section 2, which exposes dataset context (via `dataset_context.json`), a baseline configuration, and two explicit context policies (`short_context` and `long_context`) crossed with two reasoning modes (`controller` and `agentic`). Each run starts from the same baseline configuration and then proceeds for 10 iterations under one of the four settings:

`short_context_agentic`, `short_context_controller`, `long_context_agentic`, `long_context_controller`.

Outcome: MAE trajectories. Figure 2 shows the MAE over iterations for a representative run in each of the four settings. All runs start from a common baseline at step 1, and the curves track how validation error evolves as the agent proposes new `HistGradientBoostingRegressor` configurations.

Across these runs we observe three consistent patterns:

- **Short-context, agentic runs achieve the lowest MAE.** The `short_context_agentic` trajectory typically dominates the others, reaching the lowest MAE over the 10-step

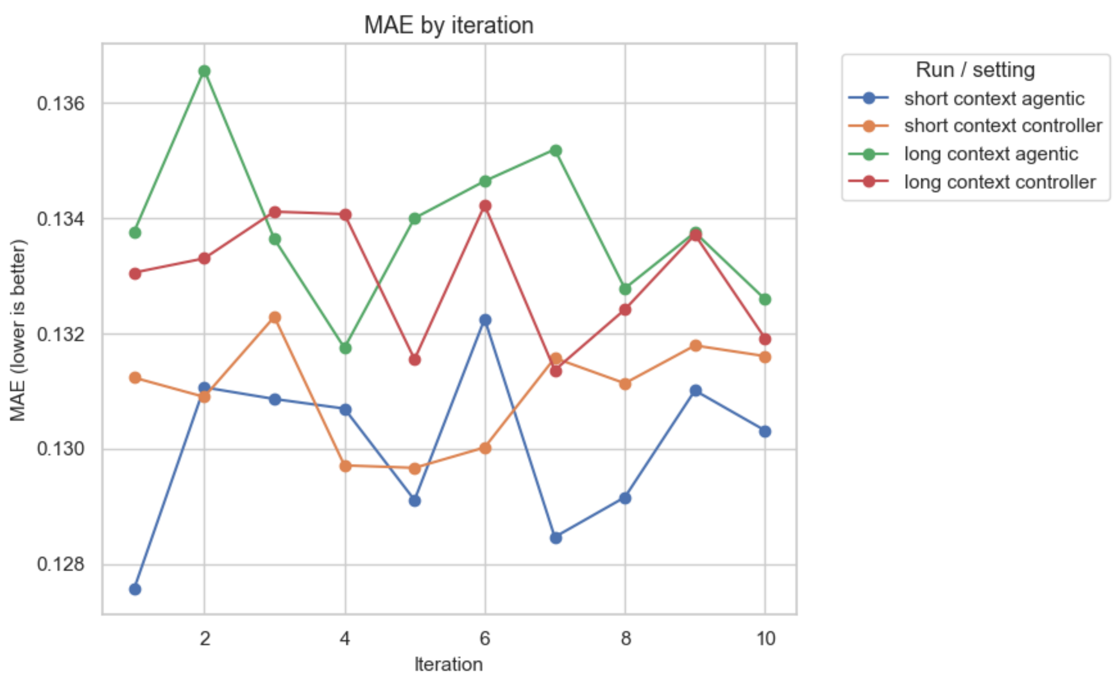

Figure 2: **MAE by iteration on NOMAD.** Validation MAE across 10 iterations for each combination of context policy and reasoning mode.

horizon while maintaining relatively smooth, stable improvements.

- **Controller runs perform competitively but slightly worse.** `short_context_controller` usually trails the agentic short-context run by a small margin, suggesting that richer inner-loop reasoning helps the agent exploit the same compact context more effectively.
- **Long-context runs underperform despite more information.** Both `long_context_agentic` and `long_context_controller` achieve higher MAE on average, indicating that simply exposing more historical and dataset context does not automatically translate into better hyperparameter proposals.

Efficiency: latency across settings. Figure 3 summarizes the per-iteration latency for each setting over the 10 iterations. Latency is measured from the moment the agent is invoked for a step to the time when the environment returns metrics.

The boxplots highlight a clear efficiency ranking:

- **Long-context, agentic is the slowest.** Combining a large retrieval budget with an inner tool loop yields the highest latency, with several iterations noticeably slower than any other setting.
- **Long-context, controller is the fastest.** Removing the inner loop while retaining a long context window yields the lowest median latency, roughly matching intuition that single-shot completions over a fixed prompt are cheap.
- **Short-context runs sit in the middle.** Both `short_context_agentic` and `short_context_cont`

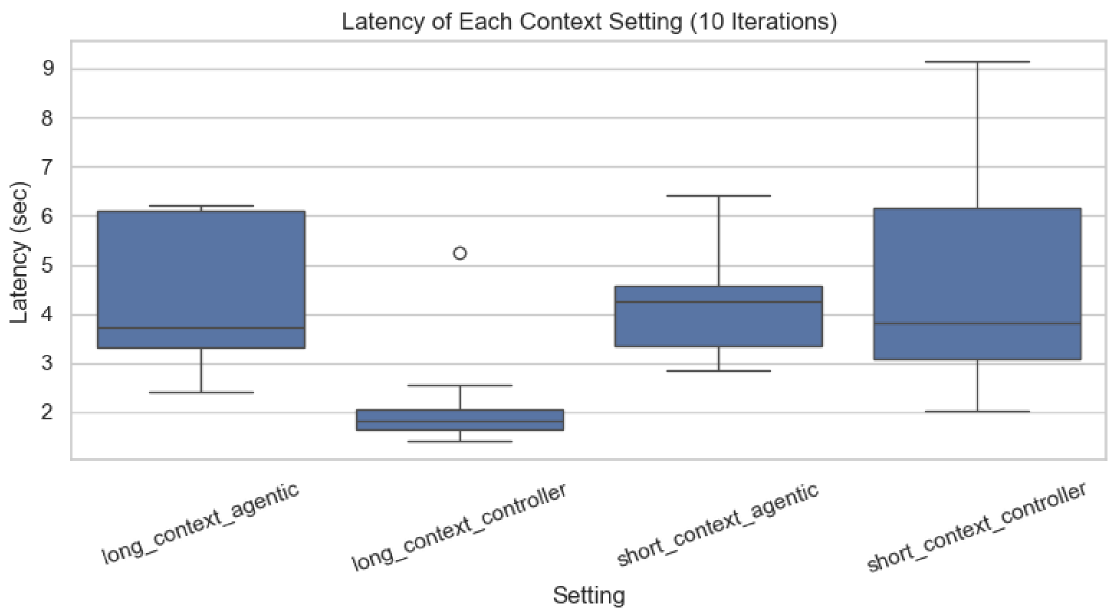

Figure 3: **Latency of each context setting.** Distribution of per-iteration wall-clock latency for the four context-policy/ reasoning-mode combinations on NOMAD. (Placeholder: replace with latency boxplot from the analysis notebook.)

have intermediate latency distributions: more expensive than `long_context_controller`, but cheaper than `long_context_agentic`.

Together with the MAE trajectories, these results suggest that the `short_context_agentic` setting offers a favorable outcome–efficiency trade-off: it achieves the best MAE while incurring only moderate latency.

Outcome vs. cost: MAE vs. tokens. Finally, Figure 4 plots MAE against total tokens per iteration for all settings, using points colored by context policy and shaped by reasoning mode.

The scatter plot reveals no simple monotone relationship between tokens and MAE:

- **More tokens do not guarantee better performance.** Long-context, agentic iterations often sit in a high-token regime without achieving lower MAE than cheaper short-context runs.
- **Short-context, agentic points cluster at good MAE with moderate tokens.** Many of the lowest-MAE iterations arise from the `short_context_agentic` configuration, which uses fewer tokens than the long-context, agentic counterpart.
- **Controller runs span a range of costs with middling MAE.** The `short_context_controller` and `long_context_controller` settings occupy a broad band of token usage but rarely reach the best MAE obtained by `short_context_agentic`.

Taken together, these analyses support the claim that context policies introduce meaningful outcome–efficiency trade-offs: the short-context, agentic configuration appears to make

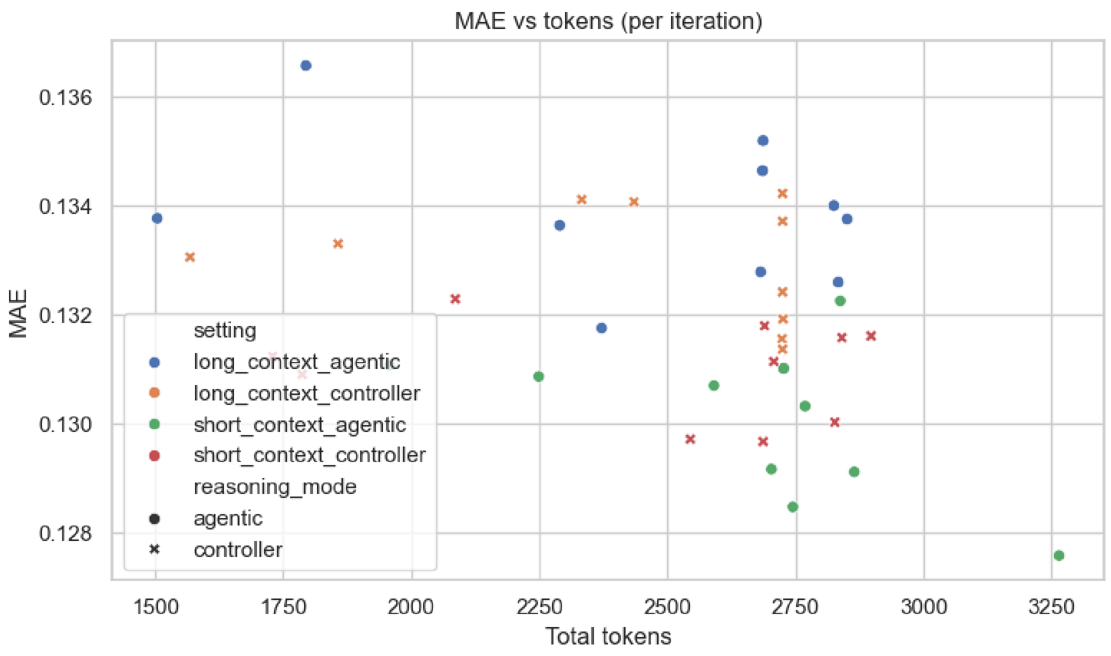

Figure 4: **MAE vs. token usage.** Relationship between per-iteration token usage and MAE for the four settings on NOMAD.

better use of a limited token budget, while long-context, agentic runs pay substantial cost without clear MAE gains.

3.4 Summary of Q1 Findings

Overall, our findings can be summarized as follows:

- The ContextEval run loop and logging utilities operate correctly on two tabular environments, producing JSONL traces with a consistent schema, usage statistics (latency and tokens), and no missing critical events.
- On the toy benchmark, the agent successfully interacts with the environment and logging stack, but improvements over the baseline are small and primarily useful for debugging; we therefore center our quantitative analysis on the NOMAD benchmark.
- On the NOMAD benchmark, different context policies and reasoning modes induce distinct behaviors and trade-offs: `short_context_agentic` achieves the best MAE with moderate latency and token use; `long_context_agentic` is the most expensive and underperforms in MAE; and controller modes are fastest but typically slightly worse in MAE than the best agentic configuration.
- There is no evidence that simply increasing context length or token usage monotonically improves performance; instead, the combination of a compact context and an agentic inner loop appears more effective for this ML experiment-design task.

These results demonstrate that even for a single model and environment, hand-designed context policies can significantly affect outcome, efficiency, and behavior.

4 Conclusion

We presented ContextEval, a framework for evaluating context policies of LLM agents acting as ML experiment designers in offline environments. By fixing datasets, training code, and scoring functions, and varying only the agent's context policy and reasoning mode, we were able to isolate how short vs. long context and agentic vs. controller reasoning shape outcome (MAE trajectories), efficiency (latency and tokens), and behavior.

Our experiments on the NOMAD bandgap regression task show that a short-context, agentic configuration achieves the best outcome–efficiency trade-off among the settings we tested, while long-context, agentic runs incur substantial additional cost without improving MAE. These findings suggest that careful design of retrieval budgets and clarifier guidance can be as important as scaling context length or model size for ML experiment-design agents.

Looking forward, we plan to extend ContextEval with richer context policies (including vague instructions and learned clarifier strategies), chain-of-thought prompting, and stability analyses across seeds and controlled task perturbations. We hope that this line of work contributes to a more systematic understanding of how context policies shape the reliability and efficiency of LLM-based ML agents.

References

Chan, Jun Shern, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2025. "MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering." *arXiv preprint arXiv:2410.07095*. [Link]

Huang, Qian, Jian Vora, Percy Liang, and Jure Leskovec. 2024. "MLAgentBench: Evaluating Language Agents on Machine Learning Experimentation." *arXiv preprint arXiv:2310.03302*. [Link]

Madaan, Aman, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. "Self-Refine: Iterative Refinement with Self-Feedback." *arXiv preprint arXiv:2303.17651*. [Link]

Shinn, Noah, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. "Reflexion: Language Agents with Verbal Reinforcement Learning." *arXiv preprint arXiv:2303.11366*. [Link]

Sun, Weiwei, Xuhui Zhou, Weihua Du, Xingyao Wang, Sean Welleck, Graham Neubig, Maarten Sap, and Yiming Yang. 2025. "Training Proactive and Personalized LLM Agents." *arXiv preprint arXiv:2511.02208*. [Link]

Yao, Shunyu, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. "ReAct: Synergizing Reasoning and Acting in Language Models." *arXiv preprint arXiv:2210.03629*. [Link]