Praxium: A Knowledge-Grounded AI Framework for Iterative Experiment-Driven Software Synthesis and Optimization

January 16, 2026

Abstract

We introduce **Praxium**, a modular framework that converts a natural-language goal into runnable software through an iterative loop of **proposal**, **implementation**, **execution**, and **evaluation**. Praxium targets long-horizon failures common in coding agents, including lost experimental state, brittle debugging, and weak reuse of domain expertise, by integrating three tightly coupled components. First, a **git-native experimentation engine** isolates each attempt as a branch, producing reproducible artifacts and preserving provenance across iterations. Second, a **knowledge system** ingests heterogeneous sources, including repositories, benchmark artifacts (starter code and evaluation scripts), internal playbooks, and curated external resources such as documentation, scientific papers, and web search results, and organizes them into a structured representation that supports hybrid retrieval over workflows, implementations, and environment constraints. Third, a **cognitive memory layer** coordinates **cascaded retrieval** (WSR with PFR fallback, plus ERA augmentation) and maintains an **episodic store** of reusable lessons distilled from prior experiment traces (run logs, diffs, and evaluator feedback), enabling the system to reduce repeated error modes and accelerate convergence. Praxium supports evaluator-defined assessment ranging from automated metrics and test suites to LLM-based judges and qualitative preference rules. We evaluate Praxium on **MLE-Bench** (Kaggle-style ML competitions) and **ALE-Bench** (AtCoder heuristic optimization), and report end-to-end performance, cost, and ablations of the search, knowledge, and memory components.

1 Introduction

Domain experts often know what they want to build, but turning that intent into reliable, runnable software still requires repeated experimentation. In practice, successful development is an iterative process: propose an approach, implement it, run it in the real environment, inspect outcomes, and refine. This loop is especially visible in Data and AI programs, where progress depends on many measurable improvements and on careful management of code, data, and evaluation contracts. Importantly, iterations often succeed in the narrow sense of producing a working artifact, yet still fall short on quality, accuracy, robustness, or efficiency. Practical progress therefore requires repeated evaluation and targeted improvement, not only error fixing.

LLM-based coding agents reduce the cost of writing code, but they remain unreliable in long-horizon execution loops. Common failure modes include losing state across iterations, repeatedly triggering the same integration errors, and failing to reuse relevant engineering expertise even when it is available in repositories, documentation, internal playbooks, or prior attempts. In many real settings, the decisive advantage is not raw code generation, but the ability to consistently apply expert-grade best practices and high-leverage engineering workflows, including environment setup, data contracts, evaluation harnessing, debugging procedures, and performance tuning. A second

practical limitation is reproducibility. Without explicit experiment isolation and provenance, it is difficult to compare approaches, debug regressions, or reuse successful solutions as building blocks.

We present **Praxium**, a framework that turns knowledge into executable software through an execution-grounded loop of iterative improvement. Given a natural-language goal and an evaluation interface, Praxium repeatedly generates candidate solution specifications, applies code changes, executes the resulting artifact under a task-defined evaluator, and uses measured outcomes to guide subsequent iterations. The system integrates three tightly coupled components. First, a git-native experimentation engine isolates each attempt as a branch, capturing code changes, logs, and evaluation outputs as reproducible artifacts. Second, a knowledge system ingests heterogeneous sources, including repositories, benchmark artifacts, internal playbooks, documentation, scientific papers, and web-derived material, and organizes them into a structured representation that supports retrieval of workflows, implementations, heuristics, and environment constraints. This knowledge is hosted in **MediaWiki**, providing a familiar interface for human review, curation, and human-in-the-loop iteration. We release a complete knowledge package consisting of a MediaWiki dump, Neo4j and Weaviate snapshots, and Docker-based deployment scripts that bring up the MediaWiki instance and all indices in a reproducible configuration. Third, a cognitive memory layer coordinates **cascaded retrieval** (WSR with PFR fallback, plus ERA augmentation) and maintains an episodic store of reusable lessons distilled from experiment traces, such as run logs, diffs, and evaluator feedback, to reduce repeated error modes and accelerate convergence.

Praxium is intentionally modular. It supports pluggable evaluators, knowledge backends, and coding agents, enabling the same iteration loop to be applied across domains where progress is defined by executable outcomes and measurable objectives. We instantiate this design and evaluate it on two complementary benchmarks, MLE-Bench and ALE-Bench, and we use these instantiations to study end-to-end performance, cost, and ablations of the search, knowledge, and memory components. Beyond these benchmarks, the same interfaces generalize to additional tasks by swapping the evaluator and the knowledge sources.

Contributions. This paper makes the following contributions:

1. An end-to-end framework that converts heterogeneous knowledge into executable software via evaluator-grounded experimentation and iterative improvement.

2. A git-native experimentation engine that represents each attempt as an isolated, reproducible branch with explicit provenance.

3. A knowledge acquisition and representation pipeline hosted in MediaWiki that converts heterogeneous sources into a typed, workflow-oriented knowledge base usable at runtime.

4. A cognitive memory system that combines cascaded retrieval (WSR with PFR fallback, plus ERA augmentation) with episodic learning from experiment traces to reduce repeated failures and accelerate iteration.

5. A modular architecture with pluggable evaluators and knowledge sources, demonstrated through benchmark instantiations and ablations, together with a released knowledge package containing a MediaWiki dump, Neo4j and Weaviate snapshots, and Docker-based deployment scripts for reproducing the full stack. The released knowledge base is populated from over 2,000 widely used Data and ML repositories, with selection criteria defined later.

2 Framework Overview

Praxium is designed around a simple contract: given a natural-language goal, a knowledge base, and an evaluator, the system produces a runnable software artifact and improves it through measured iteration. Praxium exposes a user-facing `Expert` API with three operations:

- **`learn`**: ingest and curate knowledge sources into a unified knowledge base with both human-facing and retrieval-facing representations.
- **`build`**: run an evaluator-grounded experiment loop to synthesize and iteratively improve a runnable solution.
- **`deploy`**: adapt a selected solution into a target runtime strategy and return a unified runtime handle.

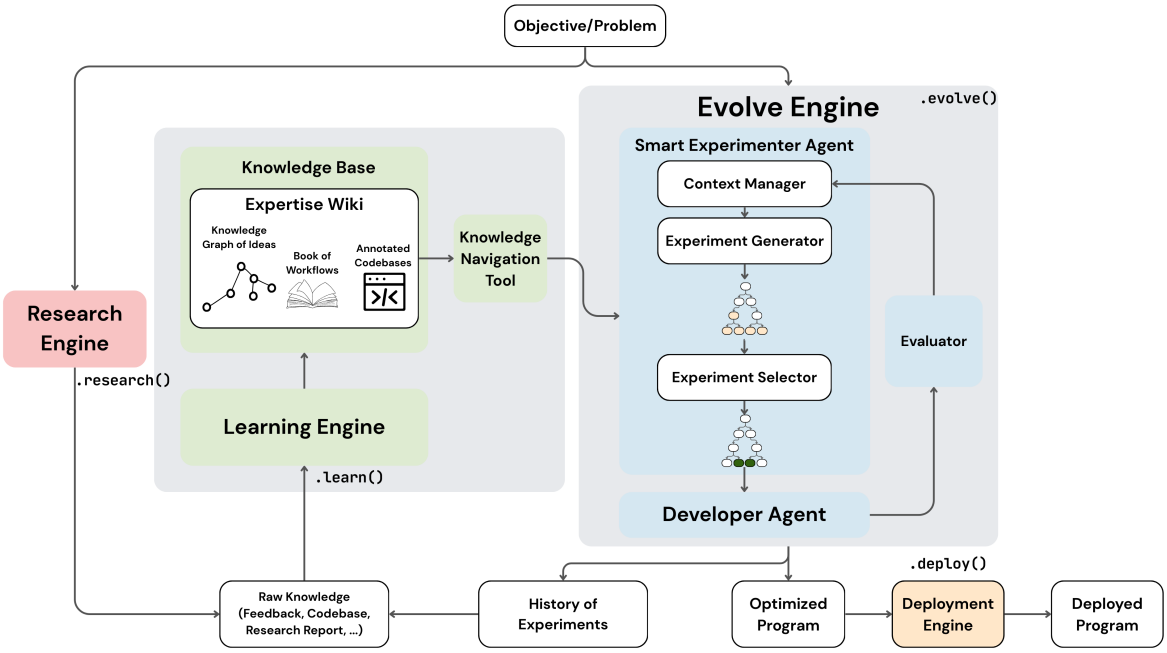

Figure 1: The overview of the framework architecture.

2.1 Knowledge plane vs. execution plane

Praxium separates *knowledge* from *execution*. The knowledge plane aggregates heterogeneous sources such as repositories, documentation, scientific papers, web-derived material, benchmark artifacts, and internal playbooks. It is hosted in **MediaWiki** for human review and curation, and it is indexed into **pluggable retrieval backends** (for example, graph and vector indices) to support machine retrieval at build time. In addition, Praxium generates self-knowledge from its own work by extracting lessons from experiment traces and storing them in an **episodic memory** store. Implementation details of ingestion, indexing, and the released knowledge package are described in Section 4.2.

The execution plane is anchored by an explicit evaluator boundary. A `ProblemHandler` (and optionally an `Evaluator`) defines the task contract: how the artifact is executed, what outputs

must be produced, and how quality is measured. Evaluators may be fully automated (metrics, tests, graders), may be stochastic, and may incorporate LLM-based judges or qualitative preference rules rather than a single scalar objective. This boundary allows the same build loop to generalize across domains by swapping the evaluator while keeping orchestration logic unchanged.

2.2 Build orchestration and modular subsystems

Internally, `Expert.build` is implemented by an `OrchestratorAgent` that composes four pluggable subsystems. These subsystems are configured per task and can be replaced independently:

- **SearchStrategy**: proposes candidate solution specifications and selects which candidates to evaluate next. Concrete strategies (for example, linear search and tree-based search) are defined formally later in the paper.
- **ContextManager**: renders the context passed into solution proposal and implementation, including the task description, constraints, retrieved knowledge, episodic insights, and summaries of prior experiments.
- **KnowledgeSearch**: retrieves workflows, implementations, heuristics, and environment constraints from the knowledge base, and returns structured knowledge packets with provenance.
- **CodingAgent**: applies code changes to implement or debug a candidate solution.

The `OrchestratorAgent` drives an iterative loop that alternates between (i) constructing context from evaluator state, knowledge retrieval, episodic memory, and experiment history, and (ii) executing one or more isolated experiments to obtain measured outcomes.

2.3 Deploy and the unified runtime interface

Today, `deploy` returns a Python `Software` handle with a unified `run()` interface, backed by an adapted copy of the solution repository. The `Software` handle exposes `run(inputs)` and returns a normalized dictionary (for example, `{"status": "success", "output": ...}` or `{"status": "error", "error": ...}`), plus lifecycle methods such as `start`, `stop`, `logs`, and `is_healthy`. Depending on the selected strategy it executes locally (import-and-call) or via an HTTP or remote endpoint (for example, Docker, Modal, BentoCloud, or LangGraph Platform), while preserving the same `run()` contract for callers. Implementation details of repository adaptation and strategy adapters are described in Section 4.4.

2.4 Experiment artifacts and provenance

Praxium represents each experiment as an isolated branch and persists run artifacts sufficient to reproduce, debug, and reuse successful attempts. The experiment artifact model and branch publication behavior are implemented by the experimentation engine described in Section 4.1.

3 Formalization (Notation and Algorithms)

This section introduces notation and formal operators for the Praxium framework. The goal is to make the system comparable to prior work in program synthesis, agentic search, and black-box optimization by stating (i) the objects Praxium manipulates, (ii) the evaluation contract it relies on, and (iii) the algorithms it runs. Benchmarks such as MLE-Bench and ALE-Bench are treated later as concrete instantiations of the evaluator contract.

3.1 Build instance and evaluator contract

A Praxium run is defined by a natural-language goal g, a budget specification B (time, iterations, and/or cost), and an evaluator contract $\mathcal{E}$. The evaluator contract defines how artifacts are executed, measured, compared, and when the run should stop:

- **Budget progress**: $\beta_i \in [0, 1]$ is the normalized budget progress at outer iteration i.
- **Problem context**: $P(\beta)$ returns the context shown to the system at budget progress β (optionally budget-aware).
- **Execution and measurement**: $\text{Run}(c; \theta)$ executes a code artifact c under evaluator configuration θ and returns a measurement record.
- **Selection rule**: either (i) a scalar utility mapping U over measurement records, or (ii) a preference relation $\succ$ over measurement records. This rule may be implemented by automated metrics, test suites, rule-based comparators, an LLM-based judge, or a human-in-the-loop policy.
- **Stochastic aggregation**: Agg_K^R and Agg_K^J aggregate K stochastic rollouts into an aggregated record and an aggregated utility estimate, respectively.
- **Stopping**: $\text{Stop}(\beta, H)$ optionally terminates the loop based on budget progress and experiment history.

A **code artifact** $c \in \mathcal{C}$ denotes an executable repository state together with sufficient entry-point/configuration to run under the evaluator. A single evaluator execution returns a measurement record

$$R(c) = (\text{status},\ M(c),\ F(c),\ A(c)),$$

where `status` indicates success or error, $M(c)$ denotes quantitative measurements (a scalar or vector of metrics), $F(c)$ denotes qualitative feedback (free-form text or structured diagnostics, including judge rationales), and $A(c)$ denotes auxiliary artifacts such as logs, traces, or produced files.

3.2 Objective, feasibility, and stochastic evaluation

Praxium improves artifacts according to the evaluator's selection rule. In the common case, the evaluator provides a scalar utility mapping $U(R(c)) \in \mathbb{R}$. When execution is stochastic, the measurement record $R(c)$ is a random variable induced by evaluator randomness, and the conceptual optimization target is the **expected utility**

$$J^\star(c) := \mathbb{E}[U(R(c))]\,.$$

This definition avoids ambiguity between $U(\mathbb{E}[R])$ and $\mathbb{E}[U(R)]$ when U is nonlinear or when R contains more than a single scalar.

Praxium does not observe $J^\star(c)$ directly. Instead, the evaluator runs K rollouts, where K is part of evaluator configuration θ. Let $R^{(k)}(c)$ be the record returned by rollout k. The evaluator defines an aggregation operator that produces an aggregated record

$$\widehat{R}_K(c) := \text{Agg}_K^R\Big(R^{(1)}(c), \ldots, R^{(K)}(c)\Big)\,,$$

and an aggregation operator that produces an aggregated utility estimate

$$\widehat{J}_K(c) := \text{Agg}_K^J\Big(U(R^{(1)}(c)), \ldots, U(R^{(K)}(c))\Big)\,.$$

In many instantiations, Agg_K^J is the sample mean and $\widehat{J}_K(c)$ is a Monte Carlo estimator of $J^\star(c)$.

The `status` field defines feasibility under the evaluator contract. Errors correspond to infeasible artifacts. The evaluator selection rule handles infeasibility by ensuring feasible artifacts are ranked above infeasible artifacts, for example by assigning a sentinel utility to errors in U (such as $-\infty$ in a maximize setting) or by defining $\succ$ to always prefer feasible records over error records.

When an evaluator does not naturally provide a scalar utility, it instead provides a preference relation $\succ$ over (aggregated) measurement records, potentially using multi-metric and qualitative feedback. In that case Praxium selects artifacts by comparing $\widehat{R}_K(\cdot)$ under $\succ$.

3.3 Experiments, provenance, and history

Praxium maintains an explicit history of executed experiments. Each experiment corresponds to an isolated execution in an experiment branch and records the motivating specification, the produced artifact, and the measured outcomes. Let b denote a branch identifier. We define the corresponding artifact as the repository state checked out from that branch:

$$c := \mathrm{RepoState}(b).$$

After iteration i, the experiment history is

$$H_i = \{e_1, \ldots, e_{i-1}\},$$

with

$$e_j = (b_j,\ u_j,\ \beta_j,\ c_j,\ K_j,\ \widehat{R}_{K_j}(c_j),\ \widehat{J}_{K_j}(c_j)), \quad \text{where } c_j := \mathrm{RepoState}(b_j).$$

Here u_j is the solution specification used to generate edits for branch b_j, β_j is the budget progress at which the experiment was executed, and K_j is the rollout count used by the evaluator. When selection is defined purely by a preference relation $\succ$, $\widehat{J}$ can be omitted and Praxium selects using $\widehat{R}$ directly.

3.4 Knowledge grounding as cascaded retrieval

Let $\mathcal{K} = (\mathcal{V}, \mathcal{E}_K)$ be a typed knowledge graph where each node $v \in \mathcal{V}$ corresponds to a typed wiki page and has an ID, a title, a type in $\{\text{Workflow}, \text{Principle}, \text{Implementation}, \text{Environment}, \text{Heuristic}\}$, content text (and optionally code snippets), and typed edges in $\mathcal{E}_K$ (for example `STEP`, `IMPLEMENTED_BY`, `USES_HEURISTIC`, `REQUIRES_ENV`).

Praxium retrieves knowledge using a cascaded (backoff) policy with failure-conditioned augmentation. Let s denote an optional signal derived from the most recent experiment record (for example, an error trace, a contract violation, or qualitative evaluator feedback). The retrieval policy produces a structured knowledge packet:

$$\mathrm{RetrieveCascade}(g, s) \to K.$$

The cascade is implemented by three retrieval objectives:

Workflow-Structured Retrieval (WSR). Given g, retrieve a best-matching workflow page and expand it into a structured bundle by traversing workflow step links and collecting associated principles, implementations, heuristics, and environment constraints.

Principle Fallback Retrieval (PFR). If no workflow match exists, retrieve relevant principles and implementations directly and present them as an unordered guidance set (no workflow synthesis).

Error-Recovery Augmentation (ERA). After a failed experiment or repeated contract violations, retrieve error-specific heuristics and alternative implementations conditioned on s, and attach them to the current knowledge packet with provenance.

We treat $\mathrm{WSR}(g)$ as returning $\emptyset$ when no workflow match is found, and we treat $s = \emptyset$ when no failure or contract-violation signal is available. We can express the cascade as:

$$K_{\text{base}}(g) := \begin{cases} \mathrm{WSR}(g) & \text{if } \mathrm{WSR}(g) \neq \emptyset, \\ \mathrm{PFR}(g) & \text{otherwise,} \end{cases}$$

$$K := \begin{cases} \mathrm{ERA}(g, s, K_{\text{base}}(g)) & \text{if } s \neq \emptyset, \\ K_{\text{base}}(g) & \text{otherwise.} \end{cases}$$

The output K is a **knowledge packet** (implemented as `KGKnowledge`). It contains a mode label in $\{\mathrm{WSR}, \mathrm{PFR}, \mathrm{ERA}\}$ indicating how the packet was formed, anchor pages and a structured payload (a step-ordered workflow in WSR, otherwise a set of principles and implementations), confidence scores and provenance metadata (including `query_used` and `source_pages`), and optional recovery attachments (heuristics and alternative implementations) when ERA is applied.

3.5 Core solve loop (Orchestrator)

At the top level, Praxium repeatedly builds context and executes experiments until a stop condition fires. In the codebase this is `OrchestratorAgent.solve()` plus a chosen `SearchStrategy`.

Algorithm 1: Praxium solve loop (orchestrator level)

```
Inputs:
  - evaluator contract E = (P, Run, Stop, Select, Agg)
  - search strategy S (linear or tree)
  - context manager M
  - budgets B

Initialize i = 0
Initialize history H_0 = empty
while i < N:
  beta_i = budget_progress(B, i)
  if Stop(beta_i, H_i) or beta_i >= 1:
    break

  x_i = M.get_context(beta_i)
  # x_i includes P(beta_i), knowledge, episodic memory, and H_i

  new_experiments = S.run(x_i, beta_i)
  H_{i+1} = H_i union new_experiments
  i = i + 1

return best artifact in H_i under evaluator selection rule
```

This formalization makes explicit that Praxium is a search over executable artifacts driven by repeated experiments and evaluator-defined assessment.

3.6 Implement-and-debug loop (SearchStrategy)

Both linear and tree search ultimately execute the same inner loop: create an isolated branch, implement a solution, run it, and optionally debug it for a bounded number of tries. The debug loop is intended to repair execution failures and evaluator-detected contract violations (for example, incorrect output format or missing required files), not to perform objective optimization.

Algorithm 2: Implement-and-debug loop (branch level)

```
Inputs:
  - solution spec u
  - context x (problem + knowledge + episodic memory + history)
  - debug budget D
  - branch name b and parent branch p

session = create_experiment_session(branch=b, parent=p)
r = implement_solution(u, x, session)   # codegen + Run

for k in 1..D:
  if r.has_error_or_contract_violation:
    r = debug_solution(u, x, r, session)
  else:
    break

finalize_session(session)               # commits + push + cleanup
return r
```

3.7 LLM-steered tree search (one concrete instantiation)

For completeness, we formalize an LLM-steered tree search instantiation used in some Praxium configurations. Let $T = (\mathcal{N}, \mathcal{A})$ be a tree of nodes $\mathcal{N}$, where each node stores a solution specification $u(n)$ and an optional experiment outcome derived from executing the corresponding artifact.

At each outer iteration, the strategy:

- **Prune**: optionally terminates some leaf nodes using an LLM conditioned on $(P(\beta_i), K_i, H_i)$.
- **Expand**: chooses nodes to expand (exploration versus exploitation) and generates new child solution specifications via an LLM ensemble.
- **Select**: selects top-k leaf nodes to execute as experiments, conditioned on the rendered context.

This can be viewed as a learned proposal distribution over solution specifications u combined with black-box evaluation through $\mathrm{Run}(\cdot)$ and selection under U or $\succ$.

3.8 Cognitive memory (cascaded retrieval, episodic learning, decisions)

Praxium maintains an episodic memory store E of reusable lessons extracted from experiment traces (run logs, diffs, and evaluator feedback). After each experiment, the system updates episodic memory, retrieves relevant prior lessons, and decides whether to continue iterating on the current workflow or pivot to alternative knowledge.

Let the controller state be C_i containing the goal g, the current knowledge packet K_i, the latest experiment record e_{i-1}, retrieved episodic insights Z_i, and meta statistics (for example, consecutive failures).

The controller defines:

- RetrieveCascade$(g, s) \to K_i$: cascaded retrieval using WSR with PFR fallback, plus ERA augmentation when s indicates failure or contract violation,
- UpdateEpisodic$(e_{i-1}) \to E$: store generalized lessons from errors and qualitative feedback,
- $\pi(C_i) \in \{\textsc{Retry}, \textsc{Pivot}, \textsc{Complete}\}$: a policy over iteration-level actions.

Algorithm 3: Cognitive controller step (per experiment)

```
Inputs: goal g, current knowledge K, episodic memory E, last experiment e

if e.has_error_or_contract_violation:
  E.add(ExtractIssue(g, e))
  K = ERA(g, e, K)   # recovery heuristics + alternatives with provenance
else if e.feedback is non-empty:
  E.add(ExtractInsight(g, e.feedback))

Z = RetrieveEpisodic(E, g, e)
a = DecideAction(g, K, e, Z)   # RETRY / PIVOT / COMPLETE

if a == PIVOT:
  K = RetrieveCascade(g, s=None)   # exclude current workflow in implementation

return (a, K, Z)
```

The key property is that both the coding agent and the decision maker are grounded in the same rendered context, and ERA explicitly records provenance via `query_used` and `source_pages`.

4 System Implementation

This section describes the implementation of the mechanisms defined in Section 3. The key design principle is that framework semantics (artifact, evaluator contract, experiment history, cascaded retrieval, and controller loops) are fixed, while concrete implementations of execution, storage, indexing, and adapters remain modular.

4.1 Experimentation Engine

The experimentation engine is the mechanism that realizes Praxium's experiment history and provenance model from Section 3. Its primary role is to isolate attempts, make outcomes reproducible, and enable reuse of successful attempts as parents for subsequent experiments.

4.1.1 Experiment sessions as git branches

Praxium represents each experiment as a git branch inside an `ExperimentWorkspace`. An `ExperimentSession` starts from a parent branch, creates a new branch identifier b, applies edits using the configured `CodingAgent`, and executes the evaluator through the active `ProblemHandler`. The session then commits the resulting artifact state and run outputs to the branch. This makes each experiment concrete and inspectable: a branch can be checked out and re-executed to reproduce the same evaluator interaction and artifacts.

To support downstream reuse, `ExperimentSession.close_session()` always attempts `git push origin <branch>`. In typical deployments, `origin` is not a network remote. Sessions are cloned from `file://<main_repo.working_dir>`, so pushing primarily publishes the branch back

into the local `ExperimentWorkspace` repository. This ensures that newly created branches are immediately available as parents for child experiments in tree-based exploration. It is not guaranteed to publish to GitHub or any external remote.

Each experiment persists a lightweight, reproducible bundle sufficient for debugging and audit. Concretely, the committed artifacts include:

- the code changes relative to the parent branch (diffs and commit identifiers),
- evaluator configuration used for the run (including rollout count K when stochastic aggregation is enabled),
- run logs and structured diagnostics, and
- evaluator-produced artifacts (for example output files and traces).

4.1.2 Execution environment and scaling

Experiments execute inside the active `ProblemHandler`. In the default path, `GenericProblemHandler` runs the artifact via local subprocess, and no containerization is introduced by the experimentation engine itself. Containerized execution is used only when the evaluator requires it. For example, ALE evaluation is performed by an external evaluation harness that runs solutions in Docker, while MLE runs the Python entrypoint locally inside the benchmark environment. This division keeps the experimentation engine evaluator-agnostic while still supporting evaluators with strict runtime requirements.

The engine supports executing multiple `ExperimentSession`s concurrently, which is used by search strategies that evaluate multiple candidates in parallel. While the current implementation primarily runs locally (or via evaluator-required containers), the design admits a pluggable execution backend. In particular, the same session and artifact model can be paired with a remote executor to run resource-intensive workloads (for example GPU-heavy training) on appropriate machines, while preserving the branch-based provenance that downstream search and reuse depend on.

4.2 Knowledge System

The knowledge system converts heterogeneous sources into a typed wiki and indexed retrieval backends. MediaWiki provides a familiar interface for human review, curation, and editing, while the same content is served to agents through retrieval APIs. The retrieval semantics, including cascaded retrieval (WSR with PFR fallback, plus ERA augmentation), are defined in Section 3. This subsection describes how those semantics are implemented and how knowledge is acquired, represented, and indexed.

Knowledge acquisition is orchestrated by a `KnowledgePipeline` that produces typed wiki pages (`WikiPage`) and merges them into a canonical store. For repositories, `RepoIngestor` follows a two-branch pipeline that balances top-down structure with coverage. First, it performs workflow-based extraction by identifying workflows from READMEs and examples, tracing code paths, and emitting linked pages spanning workflows, principles, implementations, environments, and heuristics. Second, it performs orphan mining by identifying useful files not reached by workflow tracing, triaging them deterministically, optionally requesting agent review for ambiguous cases, and emitting pages for approved orphans. This combination reduces knowledge holes that arise when extraction is driven only by high-level workflows.

Knowledge is represented as typed pages with explicit, typed links. Praxium uses five page types: `Workflow`, `Principle`, `Implementation`, `Environment`, and `Heuristic`. Links between pages

become typed edges (for example `STEP`, `IMPLEMENTED_BY`, `USES_HEURISTIC`, and `REQUIRES_ENV`). This structure is critical for returning bounded, compositional knowledge bundles. In particular, workflow expansions can be returned as ordered step bundles, while fallback retrieval returns a non-workflow guidance set without synthesizing structure that is not present in the knowledge base.

For machine retrieval, the wiki is indexed into pluggable backends. The reference implementation ships with a typed graph index (default Neo4j) and a vector index (default Weaviate), while allowing alternative graph stores and vector databases behind the same interfaces. Praxium supports two retrieval implementations: (i) LLM-guided graph navigation for lightweight setups, and (ii) hybrid graph and vector retrieval where vector search retrieves candidate pages, graph traversal enriches them with linked structure, and optional reranking improves precision. The retrieval service implements the cascaded policy in Section 3 by routing requests to workflow expansion routines, principle-level fallback routines, and failure-conditioned recovery augmentation. Across all modes, the system records provenance into the returned knowledge packet (including `query_used` and `source_pages`) to support auditability and reproducibility.

We release a complete knowledge package consisting of a MediaWiki dump, Neo4j and Weaviate snapshots, and Docker-based deployment scripts that bring up the MediaWiki instance and all indices in a reproducible configuration. The released knowledge base is populated from over 2,000 widely used Data and ML repositories, with selection criteria defined later.

4.3 Cognitive Memory System

The cognitive memory system implements episodic memory and controller decisions as formalized in Section 3. Its goal is to reduce repeated error modes, preserve high-value lessons from prior attempts, and make long-horizon iteration more stable by conditioning future proposals and debugging on prior experience.

Episodic memory entries are derived from experiment traces and are designed to be reusable across tasks. Each entry includes a compact trigger description, a generalized lesson, recommended actions, and provenance linking back to the originating experiment branch and its artifacts (for example logs, diffs, and evaluator feedback). This provenance makes the memory auditable and supports inspection when a retrieved lesson influences a new change.

After each experiment, Praxium performs lesson extraction. When the experiment indicates an execution error or contract violation, the system generalizes the failure into a reusable fix pattern grounded in the observed trace and validator feedback. When the experiment succeeds or improves quality, the system extracts best-practice insights from measured outcomes and qualitative feedback, including judge rationales when an LLM-based evaluator is used. Extracted lessons are stored in a vector database (default Weaviate) with a JSON fallback, enabling semantic retrieval by goal and by failure signal.

On subsequent iterations, Praxium retrieves relevant episodic memories conditioned on the current goal and the latest experiment signal. Retrieved lessons are rendered into the unified context passed to the search strategy and coding agent, alongside the current knowledge packet produced by cascaded retrieval. This ensures that both proposal and debugging steps are grounded not only in domain knowledge (the wiki and indices) but also in the system's own prior experience on similar issues.

Finally, the controller implements an iteration-level decision policy as in Algorithm 3 in Section 3. It uses the goal, the current knowledge packet, the latest experiment record, and retrieved episodic insights to decide whether to retry, pivot, or complete. On pivot, the implementation re-runs cascaded retrieval while excluding the currently selected workflow candidate, encouraging exploration of alternative workflows or principle-level guidance.

4.4 Deployment Interface

Praxium provides a unified deployment interface that packages a selected solution artifact into a runnable form while preserving a stable invocation contract. The user-facing contract is described in Section 2.3; this subsection describes the implementation mechanics that realize that contract across deployment strategies.

`Expert.deploy(...)` returns a Python object implementing the `Software` interface. The returned handle exposes a stable `run(inputs)` method and lifecycle methods (for example `start`, `stop`, `logs`, and `is_healthy`). The key implementation goal is to keep this contract invariant while allowing strategy-specific execution details to vary.

Deployment is implemented via repository adaptation. Given a selected solution repository, the deployment adapter creates an adapted copy at a path of the form `<solution.code_path>_adapted_<strategy>`. The adapter injects strategy-specific runtime wrappers (for example, container or service scaffolding or platform configuration) and emits a run interface descriptor that specifies how to invoke the artifact (for example an endpoint URL, a module path, or a callable reference). The `Software` handle uses this descriptor to route `run()` to the appropriate local or remote mechanism.

The current implementation supports multiple strategies under the same `Software` interface. In **LOCAL** mode, the runner imports and calls a function inside the adapted repository (default `main.predict`). In **DOCKER** mode, the system builds or reuses a Docker image and runs a local container exposing an HTTP endpoint (default `http://localhost:8000/predict`). In **MODAL** mode, it generates a `modal_app.py` and invokes a remote Modal function. In **BENTOML** mode, it generates BentoML service files and can optionally deploy to BentoCloud, returning an HTTP endpoint. In **LANGGRAPH** mode, it generates LangGraph deployment files and the runner connects to a LangGraph Platform URL to invoke the deployed agent. Across all strategies, the caller interacts only with `Software.run()`, while lifecycle methods expose health and logs in a strategy-appropriate way.

The deployment layer is extensible. New strategies can be added by implementing an adapter that (i) produces the required runtime wrapper files, (ii) emits the run interface descriptor, and (iii) registers how the `Software` handle should execute `run()` and lifecycle methods for that strategy.

5 Evaluation

We evaluate Praxium on two benchmarks that capture distinct real-world software-building regimes.

5.1 MLE-Bench (Kaggle-style ML competitions)

Task: produce a Python solution that trains on provided competition data and writes a `final_submission.csv` in the expected format.

Execution protocol (as implemented in the benchmark handler):

- run **debug mode** first (`python main.py -debug`) with a strict runtime cap,
- validate the submission file format,
- run **full mode** (`python main.py`) with a longer runtime budget,
- grade the output submission on the competition's train/test split of public train dataset.

- **Stop condition**: Run for 24 hours or until a maximum budget of $200 is reached. Also, stop early if the run achieves **any medal** according to the MLE-Bench grading library.

Reported metrics:

- **Private score**: returned by the benchmark grader.
- **Medal rate**: fraction of competitions where any medal is achieved in 4 categories: low, medium, hard, and all.

Results:

Table 1 indicates that Leeroo consistently outperforms other agents, and its advantage becomes more evident as the difficulty of tasks increases. Although performance in Low-difficulty tasks is comparable between agents (68.18%), Leeroo achieves substantially higher accuracy on Medium and Hard problems, reaching 44.74% and 40.00%, respectively, and thus generalizes more reliably across various MLE scenarios. In comparison, the closest baseline, MLE-Star, attains only 34.21% on Medium and 33.33% on Hard tasks. These results suggest that Leeroo's capabilities transfer more effectively to complex tasks involving larger data scales and higher levels of specialization, which better reflect real-world machine learning challenges.

Agent	Low (%)	Medium (%)	Hard (%)	All (%)
Leeroo	**68.18 ± 2.62**	**44.74 ± 1.52**	**40.00 ± 0.00**	**50.67 ± 1.33**
MLE-Star (CAIR)[1]	68.18 ± 2.62	34.21 ± 1.52	33.33 ± 0.00	44.00 ± 1.33
InternAgent[2]	62.12 ± 3.03	26.32 ± 2.63	24.44 ± 2.22	36.44 ± 1.18
R&D Agent[3]	68.18 ± 2.62	21.05 ± 1.52	22.22 ± 2.22	35.11 ± 0.44
AIRA[4]	55.00 ± 1.47	21.97 ± 1.17	21.67 ± 1.07	31.60 ± 0.82

Table 1: Performance of open-source and publicly available agents on MLE Bench.

See the official MLE-Bench repo for the most up-to-date results.

5.2 ALE-Bench (AtCoder heuristic contests)

Task: produce a C++ (`cpp23`) solution in `main.cpp` to maximize/minimize a contest-defined score under a strict runtime limit of each competition.

Execution protocol:

- compile and run in the benchmark's Docker environment,
- if the solution is accepted, run it multiple times and average the public evaluation score to reduce the effect of randomness.
- Run the private evaluation for the experiment with the highest public score and report final performance and rank percentile.

Reported metrics:

- **Final performance**: Final ELO rating of the submission for the competitions.
- **Rank percentile (RP)**: `final_rank` / `number_of_contestants`, Compares the agent's performance with the human-level performance.

- **Private absolute score** (contest objective value).
- **Cost**: cumulative LLM usage cost.

Results:

Table 2 shows that Leeroo achieves the highest final performance on the ALE Bench, scoring 1909.4 with a rank percentile of 6.1%, outperforming the original ALE-Agent (1879.3, 6.8%). Notably, Leeroo does this while keeping total cost relatively low at $914.8, compared to ALE-Agent's $1003.3, indicating a more cost-efficient approach. The competition-level results in Table 3 further highlight Leeroo's advantage: it surpasses ALE in most AHC competitions, sometimes by large margins—e.g., ahc016 (2022 vs. 1457) and ahc026 (2040 vs. 1965)—while maintaining comparable rank percentiles. These outcomes demonstrate that Leeroo not only achieves higher absolute performance but also generalizes more reliably across diverse competitions, combining effectiveness with efficiency, which makes it a more impactful agent for real-world ALE tasks.

One limitation of the current results is that the relatively low number of competitions in ALE-Bench may introduce noise, as the performance of LLMs and agents can vary across tasks. For example, in competition ahc039, ALE-Agent achieves a notably high score, but this performance is not consistently reflected in other similar short competitions. Future studies including a larger set of competitions, running with multiple seeds, could provide a more robust and reliable comparison between agents.

Agent	Final Performance	Rank Percentile (%)	Total Cost ($)	Model
Leeroo	**1909.4**	**6.1**	914.8	Gemini-2.5-pro
ALE-Agent [5]	1879.3	6.8	1003.3	Gemini-2.5-pro
ALE Sequential [5]	1198	54.1	111.0	Gemini-2.5-pro
ALE one-shot [5]	832	88.4	4.7	Gemini-2.5-pro

Table 2: Aggregated results on ALE Bench.

Competition	ALE		Leeroo	
	Final Performance	RP (%)	Final Performance	RP (%)
ahc008	1189	52.06	1221	49.03
ahc011	1652	20.30	1607	22.35
ahc015	2446	3.85	2528	2.82
ahc016	1457	33.05	2022	9.17
ahc024	1980	13.10	1980	13.10
ahc025	1331	47.00	1353	46.30
ahc026	1965	16.00	2040	12.43
ahc027	1740	18.12	1839	14.91
ahc046	2153	8.95	2194	8.09
ahc039	2880	0.73	2310	5.27

Table 3: Comparison of ALE and Leeroo performance across AHC competitions.

6 Conclusion

Praxium is a framework for building software by running evaluator-grounded experiments, using structured knowledge, and learning from episodic experience. Its core contributions are a git-native experimentation engine, a scalable knowledge acquisition and retrieval system with cascaded retrieval (WSR with PFR fallback, plus ERA augmentation), and a workflow-aware cognitive memory layer that stores and reuses lessons from experiment traces. The framework is designed to be modular and auditable, with a clear evaluator contract and reproducible artifacts. We provide a clear execution protocol for MLE-Bench and ALE-Bench to enable reproducible evaluation.

References

[1] J. Nam, J. Yoon, J. Chen, J. Shin, S. Ö. Arık, and T. Pfister, *MLE-STAR: Machine Learning Engineering Agent via Search and Targeted Refinement*, arXiv:2506.15692, 2025. https://arxiv.org/abs/2506.15692

[2] InternAgent Team et al., *InternAgent: When Agent Becomes the Scientist – Building Closed-Loop System from Hypothesis to Verification*, arXiv:2505.16938, 2025. https://arxiv.org/abs/2505.16938

[3] X. Yang et al., *R&D-Agent: An LLM-Agent Framework Towards Autonomous Data Science*, arXiv:2505.14738, 2025. https://arxiv.org/abs/2505.14738

[4] E. Toledo et al., *AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench*, arXiv:2507.02554, 2025. https://arxiv.org/abs/2507.02554

[5] Y. Imajuku, K. Horie, Y. Iwata, K. Aoki, N. Takahashi, and T. Akiba, *ALE-Bench: A Benchmark for Long-Horizon Objective-Driven Algorithm Engineering*, arXiv:2506.09050, 2025. https://arxiv.org/abs/2506.09050