UNSUPERVISED TOOL EVOLUTION FOR CODING AGENTS IN DATA-LAKE ANALYTICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents can answer analytical questions over a data lake by discovering, cleaning, and joining the relevant data. However, they discard this work after each question and redo it from scratch for the next one. Tool evolution, which automatically builds and refines reusable tools across tasks, is a natural fit for this repetition. Since a ground-truth label for a data-lake question requires an expert to solve it first, we consider the unsupervised setting. We propose TEAL (**T**ool **E**volution for **A**nalytics over data **L**akes), which evolves a *library of tools and usage skills* for a given Worker agent without labels. Two agents propose tools from complementary evidence, an Explorer from the data lake itself and a Critic from Worker trajectories. A proposed change is kept only if the Worker adopts the tool on a rerun without extra steps and any changed answer is supported by trajectory evidence. We also introduce TEAL-Q, which generates synthetic questions for tool evolution when real ones are unavailable. We evaluate on KramaBench and LakeQA with four Worker LLMs under step budgets, which cap the number of steps the Worker may take per question. With a tight budget of 10 or 15 steps, TEAL improves every Worker on both benchmarks, lifting Sonnet-5 at 15 steps by 9.3 points on KramaBench and 21.7 points on LakeQA. A tighter budget also yields better feedback for evolution. For example, Sonnet-5's LakeQA library evolved at 20 steps and deployed unchanged at 30 steps scores 68.3, above both the baseline (59.6) and a library evolved directly at 30 steps (64.2). Additionally, a library evolved on synthetic questions performs comparably to one evolved on real questions. These results show that useful tools can be evolved without labels, and we hope they inspire future work on what to evolve and how to obtain feedback in unsupervised agent evolution.

1 INTRODUCTION

A data lake (Nargesian et al., 2019; Hai et al., 2023) stores a large, heterogeneous collection of tables, documents, and raw files. Answering analytical questions over a data lake requires discovering relevant data (Fernandez et al., 2018; Castelo et al., 2021), transforming messy records into structured representations (Kandel et al., 2011; Li et al., 2023), and integrating data across sources (Freire et al., 2025; Zhang et al., 2026b). Coding agents equipped with a bash shell and a file editor (Wang et al., 2024a; 2025) can automate this workflow (Narayan et al., 2022; Fernandez et al., 2023). However, they typically treat each question over the same data lake in isolation and discard every parser, schema check, and join script as soon as the question ends.

Consider an analyst asking which metropolitan area accounts for the largest proportion of its state's fraud reports (Lai et al., 2026). The answer lies in 52 per-state CSV files of metropolitan-area counts with irregular headers, hidden among the data lake's thousands of files next to similar national and state summaries. Without a catalog or index, table granularity cannot be inferred from filenames, so a coding agent may pick a national summary table that lacks state breakdowns. Even when the agent locates the 52 state files, opening and cleaning them one by one exhausts a 30-step budget and fails sooner under a 15-step budget. When the next question comes, the agent repeats the same manual exploration and cleaning from scratch.

Database systems avoid repeated scans of raw data by maintaining catalogs (Halevy et al., 2016), indexes (Graefe, 2024), and views (Halevy, 2001). Coding agents can follow the same principle

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

with reusable *tools* that replace repeated step-by-step exploration with a single call. Some of these tools are backed by precomputed *data artifacts*, such as catalogs and indexes, and *usage skills* tell the agent when and how to invoke each tool.

However, existing methods cannot automatically generate such tools, for two reasons. First, catalog and ontology formats (Zhang et al., 2026a) store descriptive metadata rather than executable tools for cleaning and querying data, whereas code tool generators (Wang et al., 2024b; Li et al., 2026) build tools for the task at hand, not artifacts that pay off only across many questions. Second, supervised tool generators (Cai et al., 2024) and self-evolving ontologies (Chong et al., 2026) rely on ground-truth validation labels, which are costly to obtain in data-lake analytics because verifying even one answer requires an expert to manually discover, clean, and join the underlying data sets first. Without ground-truth labels, keeping every tool that runs without a runtime error (Li et al., 2026) is unreliable because a tool can execute cleanly while reading a table at the wrong granularity, dropping rows during parsing, or joining tables on unaligned keys.

To address these challenges, we introduce TEAL (**T**ool **E**volution for **A**nalytics over data **L**akes), an unsupervised framework that evolves a *library* for a fixed Worker agent. The library consists of executable tools, some backed by precomputed data artifacts, and usage skills. At test time, the library is frozen and provided to the Worker to answer new questions.

1.1 Contributions

Tool evolution from data exploration and Worker trajectories. TEAL builds the library offline from two complementary channels, which differ in the evidence they draw on. An *Explorer* agent inspects the data lake without seeing any question or trajectory and builds tools that a domain expert would prepare for the data lake, for example search and lookup tools backed by precomputed catalogs or cleaned tables. A *Critic* agent examines the Worker's trajectories on training questions to find where the Worker wastes steps or makes errors, and creates or revises tools accordingly. A *Skill Writer* agent then updates the usage skills to reflect every implemented change.

Tool verification without ground-truth answers. Because a newly proposed tool can still contain logic bugs or be ignored by the Worker, we introduce a rerun verification stage that accepts or rejects each change without answer labels. TEAL reruns the Worker with the updated library on the questions that motivated the change. The change is kept only if the Worker uses the changed tool, still finishes without extra steps, and, whenever its answer changes, the trajectory evidence supports the new answer over the old one.

Synthetic question generation for tool evolution. When real user questions are scarce for a data lake, we introduce TEAL-Q to generate questions that can be used by TEAL, using a small set of seed questions as demonstrations. TEAL-Q first drafts a question from a cluster of related files, then hardens it over several revisions by adding data processing and reasoning demands, such as multi-table joins, while ensuring it is answerable.

Empirical evaluation across models and step budgets. We evaluate TEAL on KramaBench (Lai et al., 2026) and LakeQA (Wang et al., 2026) with four Worker backbones (Gemini-3.8-Flash, Gemini-3.1-Pro, Sonnet-5, and Opus-4.8) under varying step budgets, i.e., the maximum Worker steps per question. We make three main observations. First, under tight 10- and 15-step budgets, TEAL improves accuracy for every Worker on both benchmarks. With Sonnet-5 at 15 steps, it gains 9.3 points on KramaBench and 21.7 points on LakeQA, and also outperforms the execution-only tool evolution method (Li et al., 2026). Second, libraries evolved under a tighter budget transfer to looser budgets. Sonnet-5's LakeQA library evolved at 20 steps scores 68.3 at 30 steps, above both the baseline (59.6) and a library evolved at 30 steps (64.2), suggesting that tighter budgets give better feedback for evolution. Third, evolving on 50 synthetic questions from TEAL-Q performs comparably to evolving on real questions, and our unsupervised rerun-based verification matches oracle ground-truth verification. The library evolved in the OpenHands harness also transfers unchanged to Google ADK, Codex CLI, and Prime Agent, improving accuracy by up to 16.3 points.

2 Related Work

Tools and Agents for Data Analytics. Data analytics over heterogeneous sources involves discovering relevant datasets (Paton et al., 2023), preparing and integrating their contents (Liu et al., 2024),

and performing analysis (Nam et al., 2025). Prior work supports this workflow with resources that humans prepare in advance, including data artifacts such as metadata catalogs and relationship indexes (Halevy et al., 2016; Fernandez et al., 2018) or materialized views and indexes (Agrawal et al., 2000), and tools such as user-defined plugins (Qiao et al., 2023) or a fixed toolset for searching and inspecting the lake (Wang et al., 2026). Recent work lets agents build such resources themselves. In EvoOntology (Chong et al., 2026), a builder agent constructs and refines an ontology layer that data agents query through tools, but a refinement is kept only if the score on a held-out labeled split improves. Closest to our setting, DataCOPE (Qiu et al., 2026) discovers skills without labels by scoring sampled trajectories through answer agreement, but its skills are procedural guidance, not callable tools backed by persistent artifacts over the data. We study fully automatic tool evolution, where no human writes tools, curates artifacts, or labels answers, and agents decide from data exploration and unlabeled trajectories which tools to build and how to guide their use with skills.

Tool Evolution for LLM Agents. Automatic tool creation for LLMs began with math and science questions that a single program can answer. A model writes a helper function, verified against labeled examples (Cai et al., 2024) or by execution tests (Lu et al., 2026; Shen et al., 2026), and reuses it across questions of the same type. Agent systems extended this to tools built during long-horizon tasks (Xia et al., 2025) and usually stored for later use. These agents create a tool when a task cannot proceed without one, such as a transcript downloader or a web page fetcher, so the tool is exercised immediately, and acceptance reduces to whether it executes (Li et al., 2026), whether the environment reflects completion (Wang et al., 2023), or whether the agent judges its output plausible (Qiu et al., 2025; Ding et al., 2026). In this work, we study unsupervised tool evolution for data-lake analytics, where neither assumption holds. Here tools rarely fill a gap, since the Worker is not blocked without them, and they cannot be credited for a correct answer without labels. TEAL therefore verifies tools by the Worker's behavior on a rerun, and we study how to make unsupervised tool evolution more effective, in particular by evolving under tighter step budgets.

3 METHOD

3.1 PROBLEM FORMULATION

Coding agents and tools. We consider coding agents equipped with a shell and a file editor, which enable them to read and modify files and execute code. Beyond these basic capabilities, we define a *tool* as a named, callable program added to the agent's action space and retained across tasks. Once implemented and debugged, a tool can be invoked in a single action without reconstructing its procedure. A tool may also rely on persistent artifacts, such as indexes or cached metadata, that are built once and reused across tasks.

Tool evolution for data-lake analytics. Given a natural-language question over a data lake, a coding agent, which we call the *Worker*, autonomously discovers the relevant files and performs the data preparation, integration, analysis, and reasoning needed to answer the question under a step budget. Without access to ground-truth answers, a tool evolution system observes the Worker's trajectories on training questions and produces a *library* $\mathcal{L} = (\mathcal{T}, S)$, consisting of a set of tools $\mathcal{T}$ and a skill file S, a natural-language document that tells the Worker when and how to use the tools. We then freeze the library and measure the Worker's accuracy with it on a disjoint set of test questions over the same data lake.

3.2 UNSUPERVISED TOOL EVOLUTION VIA TEAL

Alongside the ***Worker*** that answers questions over the data lake, TEAL coordinates five LLM agents to evolve $\mathcal{L}$. The ***Explorer*** inspects the data lake to propose data-specific tools and precomputed data artifacts. The ***Critic*** diagnoses execution bottlenecks and errors in the Worker trajectories to propose targeted tool changes. The ***Toolsmith*** implements proposals from the Explorer and Critic as executable code and validates them with unit tests. The ***Skill Writer*** maintains S that guides when and how to invoke each tool in $\mathcal{T}$, and the ***Analyst*** compares original and rerun trajectories during verification whenever their final answers differ.

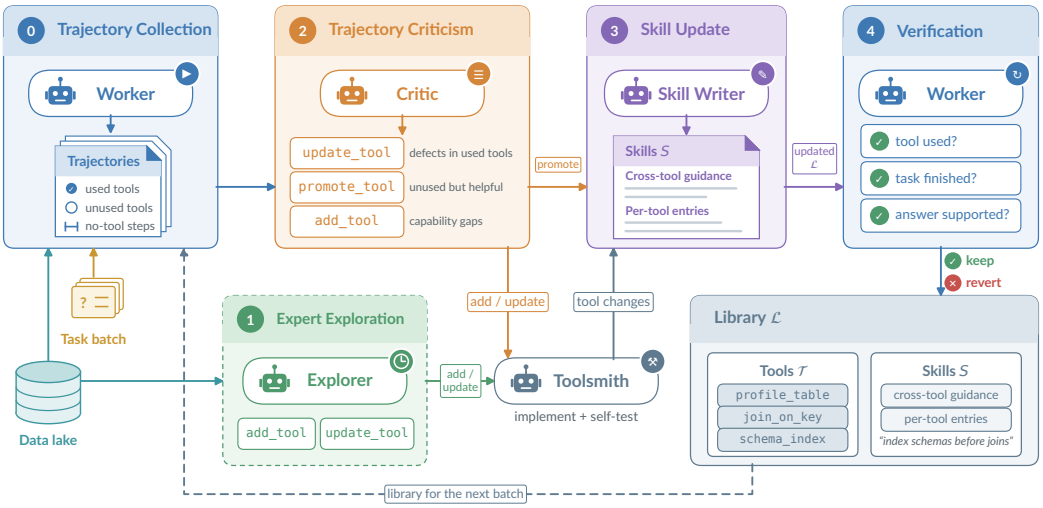

Figure 1: The TEAL evolution loop. The Worker first solves the questions with the current library (Stage 0). The Explorer proposes tools for the data lake as a whole (Stage 1), and the Critic proposes targeted fixes from the Worker's trajectories (Stage 2). The Skill Writer then updates the usage guidance (Stage 3), and a rerun of the Worker decides which changes to keep (Stage 4). Robot icons mark LLM roles. No stage uses ground-truth answers.

3.2.1 THE TOOL EVOLUTION PIPELINE

During offline tool evolution, which we refer to as training even though no model weights are updated, TEAL processes the training questions in batches and runs the five stages in Figure 1 on each batch to iteratively update $\mathcal{L}$.

In **Stage 0: Trajectory Collection**, the Worker executes a batch of questions with the current library $\mathcal{L}$. Each trajectory is recorded together with a summary of its tool usage as evidence for later stages. The summary covers which tools were used, which were unused, and where the Worker took many consecutive steps without any tool call. For example, on a question over a large directory or an Excel workbook, the summary flags that the Worker spent seven consecutive bash steps running file searches without calling any library tool and exhausted its step budget.

In **Stage 1: Expert Exploration**, an Explorer agent inspects the data lake and $\mathcal{L}$ without access to task questions or Worker trajectories. It *acts as a domain expert* preparing the tools for the data lake as a whole rather than for any specific task, and proposes `add tool` and `update tool` operations. It can also create tools backed by persistent artifacts that precompute information for reuse across tasks. A Toolsmith agent implements these proposals and checks them with executable unit tests. Unlike the other stages, this stage runs only once every E batches. For example, after surveying thousands of raw CSV and Excel files, the Explorer proposes a dataset search tool backed by a precomputed SQLite catalog of file formats, column headers, and sample rows.

In **Stage 2: Trajectory Criticism**, a Critic agent reviews the evidence collected in Stage 0 and the current tool set $\mathcal{T}$, including those just added or updated by the Explorer, to propose `update tool` operations for defects in used tools, `promote tool` operations for unused tools that could have helped, and `add tool` operations for capability gaps that the current tools do not cover. The `add tool` and `update tool` proposals are implemented by the Toolsmith, and the `promote tool` proposals are passed to the next skill update stage. For instance, when trajectories show the Worker failing on Excel sheets whose headers do not start on the first row, the Critic proposes `add tool` for a header-aware workbook loader, or proposes `update tool` when an existing catalog tool misses yearly editions of a table.

In **Stage 3: Skill Update**, a Skill Writer agent updates the skill file S by integrating the implemented changes from the Explorer and the Critic and the `promote tool` proposals. S contains two parts, the overall cross-tool guidance and the per-tool entries. For example, the Skill Writer adds a cross-

tool workflow rule instructing the Worker to query the catalog tool before running shell searches, and documents when to invoke the workbook loader on multi-sheet Excel files.

In **Stage 4: Verification**, the Worker reruns the questions with the updated library. Each rerun is compared with the original run on three checks, whether the changed tool was used, whether the task still finished, and whether any changed answer is better supported by the trajectory (Section 3.2.2). A change is kept if a rerun passes all three checks and reverted otherwise, after at most one round of repair. The resulting library $\mathcal{L}$ carries forward to the next batch and is frozen after offline training for evaluation.

Two complementary channels for tool evolution. TEAL combines two complementary channels, the Critic and the Explorer, because neither suffices alone. Prior tool-evolution methods rely only on trajectory analysis similar to our Critic (Xia et al., 2025; Li et al., 2026; Ding et al., 2026). Trajectory analysis is effective for fixing specific execution errors, such as misread offset headers in Excel sheets or bugs in called tools. However, because the Critic only sees a small batch of trajectories at a time, it focuses on immediate fixes for those questions rather than building data lake-wide tools. The Explorer addresses this limitation. Without seeing any question or trajectory, it is instructed to act as a domain expert preparing tools for the whole data lake. It scans the lake's directory structures, file formats, and schemas and proposes tools that serve many questions, such as search and lookup tools, some backed by precomputed data artifacts such as metadata catalogs and value indexes. These tools let the Worker locate relevant data in a single tool call. In short, the Critic asks *what should change given how the Worker actually solved these tasks?*, while the Explorer asks *what would a domain expert build for this data lake?*

3.2.2 Verification without ground-truth answers

A proposed tool change can pass its unit tests and still fail to help the Worker. The tool may contain a logic error, such as reading a table at the wrong granularity, or the Worker may simply ignore it. *Verification* is the stage that decides whether to keep or revert each change. Because ground-truth answers are unavailable, TEAL cannot judge a change by whether it makes answers correct. Instead, it reruns the Worker with the updated library on the questions that motivated the change and compares each rerun with the original trajectory on three checks.

The first check confirms that the Worker actually calls the changed tool during the rerun, since a tool that is never called cannot improve any answer. When the Worker ignores the tool, the reason is usually unclear guidance rather than a broken tool, so TEAL asks the Skill Writer to revise the tool's usage instructions and reruns the question before rejecting the change.

The second check confirms that the change does not make the Worker worse at finishing the question. A rerun fails this check if it produces no answer where the original run produced one, or if it uses more than m steps beyond the n steps of the original run.

The third check applies only when the rerun produces a different answer from the original run. In this case, an Analyst agent compares the two trajectories and judges which answer is better supported by the execution evidence, for example whether the final value appears in a tool output or is merely asserted by the Worker. The Analyst does not assume that the original answer is correct. The rerun fails this check only if the evidence favors the original answer, and passes if the evidence favors the new answer or is inconclusive.

TEAL keeps a change if at least one rerun passes all three checks and reverts it otherwise. In this way, verification goes beyond checking whether a tool runs without errors and uses the Worker's own trajectories as evidence of whether the tool actually helped.

3.3 Synthetic Question Generation via TEAL-Q

Questions written by users are scarce and not always accessible. Existing data lake benchmarks contain only around a hundred questions each, and a newly deployed data lake may have none. Because TEAL does not need ground-truth answers, evolution needs only questions. We therefore propose TEAL-Q, which generates synthetic questions from the data lake, using a small set of seed questions as demonstrations of style and difficulty. As shown in Figure 2, TEAL-Q generates each question in four stages.

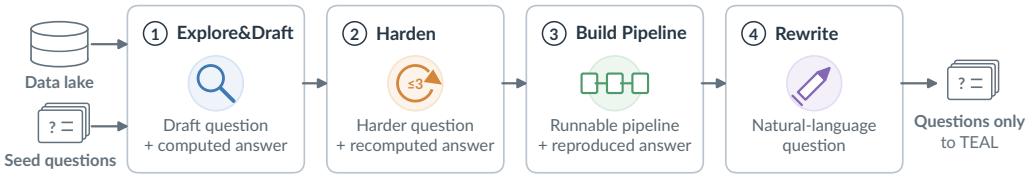

Figure 2: Synthetic question generation with TEAL-Q. Given the data lake and a small set of seed questions as demonstrations, TEAL-Q generates each question in four stages.

① **Explore and Draft.** TEAL-Q inspects a cluster of relevant data files and drafts a question with an answer, computed by running code on the data. ② **Harden.** In up to three rounds, TEAL-Q makes the question harder, for example by adding a join across tables or a hard-to-parse file, and recomputes the answer after each round. ③ **Build Pipeline.** TEAL-Q turns the final solution into an executable pipeline and checks that it reproduces the computed answer. This confirms that the question is answerable from the data. ④ **Rewrite.** TEAL-Q rewrites the question in natural language, preserving essential definitions that are needed to make it unambiguous but omitting details that the Worker can discover itself.

TEAL-Q passes only the questions to TEAL. The answers and solution pipelines serve only to keep each question well posed and answerable. As a result, TEAL-Q does not need to verify that its answers are correct, which is a major challenge in synthetic data generation (Ramrakhya et al., 2026; Gureja et al., 2025).

4 Experiments

Our experiments address two research questions (RQs). **RQ1: Are agent-evolved tools useful?** We ask whether a library evolved without labels improves a coding agent's accuracy on data-lake analytics, across Workers, datasets, and step budgets. **RQ2: How can feedback be used effectively without ground-truth answers?** We ask which signals, from step budget pressure to the proposal channels and rerun-based verification, make unsupervised evolution work.

4.1 Experimental Settings

Datasets. For KramaBench (Lai et al., 2026), we merge the data lakes of all six domains (archaeology, astronomy, biomedical, environment, legal, and wildfire) into a single unified data lake of 1,742 files and use stratified sampling to select 51 training and 53 test tasks.

For LakeQA (Wang et al., 2026), we use its tables-only variant, which keeps the 130 questions whose final reasoning step relies on tables and folds any fact from removed non-table sources into the question text, preserving the original answers. We randomly split them into 50 training and 80 test tasks. Each requires one to five reasoning hops over a lake of 4,180 tables.

We report LakeQA accuracy under its deterministic scorer and the official KramaBench score, which is exact match for most tasks and partial credit (F1 for lists, relative error for approximate numerics, an LLM judge for free-text answers) for the rest. We refer to both as accuracy.

Setup. We use OpenHands[1] as the harness for all agents. To assess generalization across Worker backends, we evaluate Gemini-3.8-Flash (Google DeepMind, 2026b), Gemini-3.1-Pro (Google DeepMind, 2026a), Sonnet-5 (Anthropic, 2026b), and Opus-4.8 (Anthropic, 2026a), while using Opus-4.8 for all evolution roles. Each Worker episode has a step budget of k: up to $k-3$ steps for the task, plus at most 3 steps to write an answer if none has been written by then. A step budget is the practical way to bound the cost and latency of answering a question, and how many steps a deployment can afford varies. We therefore treat it as an experimental variable. In all experiments, we use batches of 5 questions, run the Explorer on the first batch and every $E = 4$ batches after, and set $m = \max(2, 0.2n)$.

[1] `https://github.com/OpenHands/software-agent-sdk`.

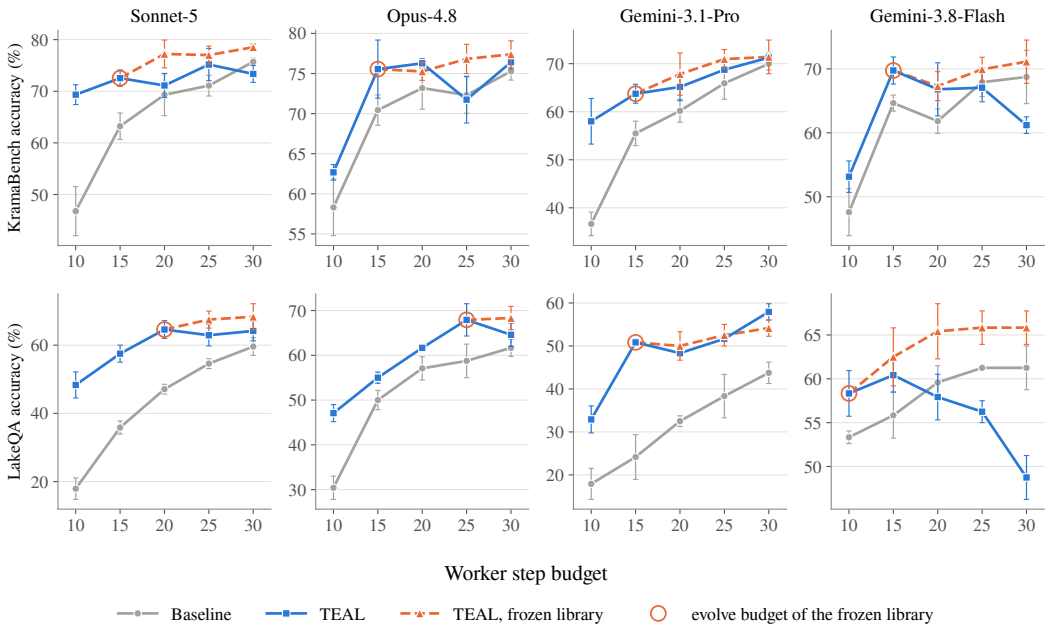

Figure 3: Accuracy versus Worker step budget in the real-question scenario. Rows correspond to benchmarks (top: KramaBench, bottom: LakeQA) and columns to Worker models. The horizontal axis is the step budget k at test time, from 10 to 30 steps. The grey line is the baseline Worker without evolved tools. The blue line is TEAL with a library evolved and tested at the same budget k. The orange dashed line is a frozen library: it is evolved once at the ringed budget and then tested unchanged at each looser budget.

4.2 Main Results

Given the real training questions from the datasets, we sweep k over 10, 15, 20, 25, and 30 steps. Figure 3 plots accuracy against k for the baseline without the evolved library and for TEAL, reporting the mean and standard deviation over three test repeats. We have the following findings.

Under a tight step budget, TEAL improves accuracy substantially. With Sonnet-5 as the Worker, TEAL improves accuracy by 22.6 points on KramaBench and 30.4 points on LakeQA at a budget of 10 steps, and by 9.3 and 21.7 points at 15 steps. The gain holds for all four Workers on both lakes at 10 and 15 steps. In five of the eight panels, TEAL at 15 steps is on par with or above the baseline at 30 steps. We attribute the improvement to the fact that a single tool call replaces a multi-step procedure, such as locating, loading, and joining the relevant files, so the Worker can finish the task within a budget that the baseline exhausts on file discovery. Appendix A illustrates this with two test tasks, and Appendix B lists the most used tools.

Under a loose budget, a library evolved at that budget does not always help, but one evolved under a tighter budget does. At 20 to 30 steps, the library evolved at the deployment budget (blue line) sometimes falls below the baseline (grey line), whereas the library evolved under a tighter budget and deployed unchanged (orange line) stays above the baseline at every looser budget and is usually the best of the three. The difference lies in the feedback that evolution receives. Under a tight budget, the Worker fails in visible ways, through empty answers, truncated pipelines, and repeated processing work, which give the Critic concrete targets. A saved step then changes the rerun outcome, so verification can tell a useful tool from a useless one, and the short trajectories are also easier to analyze. Under a loose budget, the Worker solves most tasks inline, so a tool rarely changes the outcome, and the long trajectories make proposals harder to ground and reruns harder to compare, which leaves the quality of tools and skills unchecked. Step budget pressure thus acts as a training signal in place of gold labels, which makes evolving under a tighter budget and deploying under a looser one a practical recipe.

Appendix C further reports the number of steps each Worker actually uses and the resulting cost per episode, which reflect the practical value of TEAL in deployment.

4.3 Comparison with Existing Methods

We also compare TEAL with the agentic methods evaluated in KramaBench and LakeQA and with an existing tool evolution method. **DS-Guru** (Lai et al., 2026) writes a complete Python script from a data preview and repairs it with error feedback. **Smolagents DeepResearch** (Roucher et al., 2025; Lai et al., 2026) is a ReAct agent (Yao et al., 2023) that executes Python step by step with file inspection tools. **LakeQA Agent** (Wang et al., 2026) is the reference agent released with LakeQA, which calls human-designed tools to search, inspect, and run Python, and **LakeQA Agent + RAG** (Wang et al., 2026) adds a retrieval tool backed by a pre-built index of the lake's tables. **ISE** (Li et al., 2026) evolves tools without gold labels and keeps every tool that runs without errors, and we reproduce it from the released code and replace only its Worker with that of TEAL.

For each existing method, we use the hyperparameters from its original evaluation on the corresponding dataset, including a 20-step budget for Smolagents and a 25-step budget for LakeQA Agent, so its result is the same in both columns. Table 1 leads to two observations.

	k=15	k=30
KramaBench		
DS-Guru	39.73	
Smolagents	46.69	
Baseline	$63.23_{\pm 2.57}$	$75.71_{\pm 0.70}$
ISE	$66.81_{\pm 5.96}$	–
TEAL	$72.54_{\pm 1.59}$	$\mathbf{78.53}_{\pm 0.65}{}^{\dagger}$
LakeQA		
LakeQA Agent	55.00	
+ RAG	58.75	
Baseline	$35.83_{\pm 1.91}$	$59.58_{\pm 2.60}$
ISE	$25.83_{\pm 1.91}$	–
TEAL	$57.50_{\pm 2.50}$	$\mathbf{68.33}_{\pm 3.82}{}^{\dagger}$

Table 1: Accuracy (%) of existing methods and TEAL, Sonnet-5 as the Worker. †Library evolved at 15 (KramaBench) or 20 (LakeQA) steps, deployed unchanged (orange lines in Figure 3).

First, our baseline Worker is already a strong coding agent. On KramaBench, it outperforms DS-Guru and Smolagents by a wide margin even at 15 steps, fewer than the 20 steps Smolagents is allowed. On LakeQA, it surpasses LakeQA Agent at 30 steps, with or without RAG, even though LakeQA Agent comes with human-designed tools and a pre-built retrieval index. TEAL improves on this strong baseline on both datasets and at both budgets.

Second, we compare TEAL with ISE, another unsupervised tool evolution method whose released code allows a faithful reproduction. ISE retains a tool once it executes successfully. Under the same Worker and the same 15-step budget, ISE underperforms TEAL on both datasets (by 5.7 points on KramaBench), indicating that a tool that executes successfully does not necessarily help with the task. This issue is more pronounced on LakeQA, where ISE falls 10.0 points below the baseline. For instance, ISE evolved `search_file_content`, which scans the lake line by line. The tool was retained after a single successful run on a simple search, yet 121 of its 126 calls timed out at test time. The steps consumed by these calls prevented the Worker from reaching an answer within the step budget. Under execution-only verification, such a tool therefore does more harm than good.

4.4 Results in the Synthetic-Question Scenario

	KramaBench		LakeQA	
	Acc.	Δ	Acc.	Δ
Baseline	$63.23_{\pm 2.57}$	–	$35.83_{\pm 1.91}$	–
TEAL (real questions)	$72.54_{\pm 1.59}$	+9.31	$57.50_{\pm 2.50}$	+21.67
TEAL (synthetic questions)	$72.18_{\pm 3.22}$	+8.95	$56.25_{\pm 3.31}$	+20.42

Table 2: Accuracy (%) of TEAL evolved on real or synthetic questions with Sonnet-5 as the Worker. All configurations use a 15-step budget and the same held-out test sets.

In the synthetic-question scenario, TEAL-Q generates 50 questions from the lake of each dataset. Evolution sees only these questions, without answers, and the resulting library is evaluated on the same held-out test tasks as in the real-question scenario. In Table 2, the library evolved on synthetic questions improves Sonnet-5 over the baseline on both datasets and performs comparably to the library evolved on real questions. The same holds for all four Workers (Table 7 in Appendix D). Although the generated questions are never verified by a human, they exercise enough of the lake's

preparation and discovery work for TEAL to build a library that transfers to real questions, so tool evolution can begin before any user question has been asked.

4.5 Ablation and Generalization

Table 3 isolates the contribution of each component with Sonnet-5 as the Worker at a budget of 15 steps. Removing the Explorer leaves only the Critic channel, which still improves over the baseline but falls slightly short of the full TEAL. Evolution without the Explorer also no longer produces the artifact-backed tools shown in Appendix A, consistent with the complementary roles of the two channels (Section 3.2.1). Removing the evolved skills causes a larger drop, because the Worker no longer knows how to use the tools. A good tool alone therefore does not guarantee a gain, and the tools and their usage skills must be evolved together.

	KramaBench	
	Acc.	Δ
Baseline	$63.23_{\pm 2.57}$	–
TEAL	$72.54_{\pm 1.59}$	+9.31
− Explorer	$71.59_{\pm 2.28}$	+8.36
− Evolved skills	$68.69_{\pm 0.85}$	+5.46
+ Ground-truth answers	$72.73_{\pm 3.07}$	+9.50
+ Ground-truth solutions	$75.55_{\pm 3.04}$	+12.32
Google ADK: Baseline	$53.95_{\pm 4.41}$	–
Google ADK: TEAL	$70.20_{\pm 3.42}$	+16.25
Prime Agent: Baseline	$64.85_{\pm 1.11}$	–
Prime Agent: TEAL	$67.43_{\pm 2.41}$	+2.58
Codex CLI: Baseline	$66.22_{\pm 3.34}$	–
Codex CLI: TEAL	$70.57_{\pm 0.96}$	+4.35

Table 3: Accuracy (%) of ablations and reference configurations with Sonnet-5 as the Worker at a 15-step budget, in the real-question scenario.

We also report two oracle variants that give TEAL ground-truth labels during evolution, which our unsupervised setting does not have. Given only the ground-truth answers, TEAL performs comparably to its label-free version, so our verification checks already obtain the feedback that the answers would provide. Given the full ground-truth solutions as well, the system knows how to solve the tasks and creates more targeted tools, such as `merge_entity_tables`, which joins tables on both state and metropolitan area and matches area names that differ only in punctuation, whereas the label-free library only standardizes state names with `normalize_join_state_keys`.

To test generalization across harnesses, we take the library evolved under OpenHands and deploy it unchanged under three other harnesses, Google ADK[2], Prime Agent[3], and Codex CLI[4], with Sonnet-5 as the Worker in each. The library improves the Worker over its baseline under every harness, so the evolved tools and skills generalize beyond the harness they were evolved in.

5 Conclusion

In this paper, we proposed TEAL, a framework that evolves tools and usage skills for a coding agent over a data lake without ground-truth labels, and TEAL-Q, which generates synthetic questions for evolution. TEAL proposes tools from both the data lake and the Worker's trajectories, and verifies each change by checking the Worker's tool use and trajectory evidence on a rerun, which performs comparably to verification with ground-truth answers. Our results show that unsupervised tool evolution is most effective under a tight step budget, which reveals where the Worker needs tools. Libraries evolved in this way improve all four Workers under tight budgets and remain effective under looser budgets, and libraries evolved on synthetic questions perform comparably to those evolved on real ones.

Limitations and future work. The evolved library of tools and skills introduces additional overhead in two forms. First, it lengthens the prompt that the Worker reads at every step, which increases the input tokens. Second, the Worker tends to reason more at each step, plausibly about which tools to call (Appendix C). Future work may reduce the input overhead by improving cache reuse of repeated prompt content, and the reasoning overhead by internalizing the skills into the Worker.

[2] `https://github.com/google/adk-python`.
[3] `https://github.com/PrimeIntellect-ai/prime-agent`.
[4] `https://github.com/openai/codex`.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods, and supporting qualitative and thematic data analysis. We have not used generative AI tools for proposing or refining hypotheses, or designing or providing feedback on research methodology or experiments, and the rest of the required disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for creating or modifying scientific figures or images, creating or editing software code, drafting parts of a research paper, summarizing or analyzing existing literature, discovering research topics or identifying gaps, editing a research paper to improve readability, and identifying relevant literature.

We have reviewed all AI-assisted work. We verified LLM-generated code and manually reviewed and revised all AI-edited text in this paper. Note that LLM agents are also the object of study in this work. We study how LLM agents can autonomously optimize their own tools and skills, so the tool code and skill text are intentionally written by the agents themselves without human review. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have detailed the experiment setup in this paper, including datasets, splits, Worker models, the model used for all evolution roles, and the step budget. To support reproduction, we will release the code for TEAL and TEAL-Q, along with the task splits and the synthetic questions.

REFERENCES

Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pp. 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607153.

Anthropic. System card: Claude opus 4.8. `https://www.anthropic.com/claude-opus-4-8-system-card`, May 2026a. Accessed: 2026-09-15.

Anthropic. System card: Claude sonnet 5. `https://www.anthropic.com/claude-sonnet-5-system-card`, June 2026b. Accessed: 2026-09-15.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations*, volume 2024, pp. 54067–54089, 2024.

Sonia Castelo, Rémi Rampin, Aécio Santos, Aline Bessa, Fernando Chirigati, and Juliana Freire. Auctus: A dataset search engine for data augmentation. *arXiv preprint arXiv:2102.05716*, 2021.

Meiduo Chong, Shaolei Zhang, Ju Fan, and Xiaoyong Du. Evoontology: A self-evolving ontology layer for data agents. *arXiv preprint arXiv:2609.15779*, 2026.

Haipeng Ding, Yuexiang Xie, Zhewei Wei, Yaliang Li, and Bolin Ding. From atomic actions to standard operating procedures: Iterative tool optimization for self-evolving llm agents. *arXiv preprint arXiv:2607.07321*, 2026.

Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International conference on data engineering (ICDE)*, pp. 1001–1012. IEEE, 2018.

Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. How large language models will disrupt data management. *Proceedings of the VLDB Endowment*, 16(11):3302–3309, 2023.

Juliana Freire, Grace Fan, Benjamin Feuer, Christos Koutras, Yurong Liu, Eduardo Peña, Aécio Santos, Cláudio T Silva, and Eden Wu. Large language models for data discovery and integration: Challenges and opportunities. *IEEE Data Engineering Bulletin*, 2025.

Google DeepMind. Gemini 3.1 pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026a. Accessed: 2026-09-15.

Google DeepMind. Gemini 3.8 flash model card. https://deepmind.google/models/model-cards/gemini-3-8-flash/, September 2026b. Accessed: 2026-09-15.

Goetz Graefe. More modern b-tree techniques. *Foundations and Trends in Databases*, 13(3):169–249, 2024.

Srishti Gureja, Elena Tommasone, Jingyi He, Sara Hooker, Matthias Gallé, and Marzieh Fadaee. Verification limits code llm training. *arXiv preprint arXiv:2509.20837*, 2025.

Rihan Hai, Christos Koutras, Christoph Quix, and Matthias Jarke. Data lakes: A survey of functions and systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(12):12571–12590, 2023.

Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 795–806, 2016.

Alon Y Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*, pp. 3363–3372, 2011.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, et al. Kramabench: A benchmark for ai systems on data-to-insight pipelines over data lakes. In *International Conference on Learning Representations*, volume 2026, pp. 142883–142912, 2026.

Haotian Li, Shijun Yang, Weizhen Qi, Silei Zhao, Rui Hua, Mingzhu Song, Xiaojian Yang, and Chao Peng. Yunjue agent tech report: A fully reproducible, zero-start in-situ self-evolving agent system for open-ended tasks. *arXiv preprint arXiv:2601.18226*, 2026.

Peng Li, Yeye He, Cong Yan, Yue Wang, and Surajit Chaudhuri. Auto-tables: Synthesizing multi-step transformations to relationalize tables without using examples. *Proc. VLDB Endow.*, 16 (11):3391–3403, July 2023. ISSN 2150-8097. doi: 10.14778/3611479.3611534. URL https://doi.org/10.14778/3611479.3611534.

Yurong Liu, Eduardo Pena, Aecio Santos, Eden Wu, and Juliana Freire. Magneto: Combining small and large language models for schema matching. *arXiv preprint arXiv:2412.08194*, 2024.

Jiaxuan Lu, Ziyu Kong, Yemin Wang, Rong Fu, Haiyuan Wan, Cheng Yang, Wenjie Lou, Haoran Sun, Lilong Wang, Yankai Jiang, Xiaosong Wang, Xiao Sun, and Dongzhan Zhou. Beyond static tools: Test-time tool evolution for scientific reasoning. *arXiv preprint arXiv:2601.07641*, 2026.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Raj Sinha, Jinwoo Shin, and Tomas Pfister. Ds-star: Data science agent for solving diverse tasks across heterogeneous formats and open-ended queries. *arXiv preprint arXiv:2509.21825*, 2025.

Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.

Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment*, 12(12):1986–1989, 2019.

Norman W Paton, Jiaoyan Chen, and Zhenyu Wu. Dataset discovery and exploration: A survey. *ACM Computing Surveys*, 56(4):1–37, 2023.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Zhisong Qiu, Kangqi Song, Shengwei Tang, Shuofei Qiao, Lei Liang, Huajun Chen, and Shumin Deng. Unsupervised skill discovery for agentic data analysis. *arXiv preprint arXiv:2606.06416*, 2026.

Ram Ramrakhya, Andrew Szot, Omar Attia, Bogdan Mazoure, Anh Nguyen, Yuhao Yang, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *International Conference on Learning Representations*, volume 2026, pp. 123630–123660, 2026.

Aymeric Roucher, A Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. smolagents: A smol library to build great agentic systems. *Hugging Face*, 2025.

Xintian Shen, Jiawei Chen, Lihao Zheng, Hao Ma, Tao Wei, and Kun Zhan. Evolving from tool user to creator via training-free experience reuse in multimodal reasoning. *arXiv preprint arXiv:2602.01983*, 2026.

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

Haonan Wang, Jiaxiang Liu, Yurong Liu, Austin Senna Wijaya, Tianle Zhou, Eden Wu, Yijia Chen, Wanting You, Reya Vir, Daniela Pinto, et al. Lakeqa: An exploratory qa benchmark over a million-scale data lake. *arXiv preprint arXiv:2606.10460*, 2026.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable Code Actions Elicit Better LLM Agents. In *International Conference on Machine Learning*, 2024a. URL https://arxiv.org/abs/2402.01030.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Haoxiang Zhang, Yurong Liu, Aécio Santos, Wei-Lun (Allen) Hung, and Juliana Freire. Autoddg: Automated dataset description generation using large language models. *Proc. ACM Manag. Data*, 4(1), April 2026a. doi: 10.1145/3786626. URL https://doi.org/10.1145/3786626.

Jiani Zhang, Sercan O Arik, Cosmin Arad, Fatma Ozcan, and Alon Halevy. An agentic approach to metadata reasoning. *arXiv preprint arXiv:2604.20144*, 2026b.

A Qualitative Analysis

Tools with build artifacts. As introduced in Section 3, the Explorer may create a tool backed by a persistent artifact that is constructed once, stored alongside the tool, and consulted at call time. We illustrate this mechanism with `search_dataset_catalog`, the discovery tool of the Sonnet-5 LakeQA library evolved under a budget of 10 steps.

Its artifact is a catalog of the lake, built in a single pass over every directory that infers the true format of each file from its bytes, records the header, an estimated row count, per-column samples, and data-quality flags, and embeds the name of each directory. The pass takes approximately four minutes of wall-clock time and roughly forty batched embedding calls, and the resulting 93 MB SQLite catalog is stored beside the tool and never rebuilt inside a task. The tool itself is a ranked search over this catalog. It takes a natural-language description of the data sought and returns the best-matching directories. The tool was revised fourteen times, each revision traceable to a Worker trajectory. For instance, after a Worker had to fall back to `ls | grep` to collect the yearly editions of a table, the tool was revised to return all editions of a matched table together.

Across all experiments, most artifacts are catalogs, which answer which file is relevant, or value indexes, which answer which file contains a given value. What exactly is built varies from one evolution run to another. For example, one value index inverts the cell values themselves and reaches 6 GB, whereas another is a full-text index over names, headers, and sampled cells. Other artifacts also occur, such as manifests that record which files belong together and in what order, and decoded copies of archives that were mis-stored as text.

How tools help. First, tools save steps, which is why TEAL is particularly effective under a tight step budget. At test time the Worker invoked `search_dataset_catalog` in all 80 held-out tasks. On the task requesting the 2018 violent crime rate of Albany County, every TEAL repeat finished within four steps. One catalog query located the county-level crime-rate table, one SQL call through the companion `query_tabular_file` tool returned 359.8, and the Worker submitted the answer. Without such a tool, the baseline used nine to ten steps in every repeat, most of them on `find` and `grep` over the lake. The additional steps did not translate into accuracy, since in two of the three repeats the search selected an incorrect crime table and the baseline did not answer correctly. Figure 6 shows one repeat of each.

Second, tools make accuracy more robust, because a tool's output is consistent across repeats while the Worker's manual work is not. On the task asking for the calcium reading at the smallest dust-proxy value in a paleoclimate workbook, `inspect_excel_workbook` detected the header on the sixth row and returned the columns under their real names, so every TEAL repeat filtered the two relevant columns with short scripts and answered correctly. The baseline, with nine steps in every repeat but without reusable tools, did not reach a correct answer. In one repeat it parsed the workbook manually and misaligned the header, and in another it located the file but failed to parse the header. Figure 7 shows one repeat of each.

B Tool Statistics

Tables 4 and 5 list the five most used tools in the libraries evolved for each lake with Gemini-3.1-Pro as the Worker. We run each test task three times and call each run an episode. The coverage of a tool is the percentage of test episodes in which the Worker called that tool at least once. On both lakes the most used tools answer where the data lives.

As shown in Table 4, three of the KramaBench tools help the Worker find data. One maps a topic to the files that hold it, one searches for a column name or a value across all files, including Excel workbooks that `grep` cannot read, and one lists the files in the lake. The other two load files that standard readers parse incorrectly, namely CSVs that stack several tables in one file and Excel sheets whose header is not on the first row. The LakeQA lake is larger, with over four thousand files. As shown in Table 5, its library therefore indexes the lake ahead of time. One tool looks up a pre-built catalog that records where each dataset is and which columns and format it has, and another searches a pre-built full-text index of dataset names and column headers. Its compute tool runs a filter or an aggregate over a large file through DuckDB, which streams the file instead of loading it

Tool	Coverage	Role	What it does
`resolve_concept_to_datasets`	55.3%	concept lookup	Maps a natural-language concept or a few keywords to the lake files that hold the corresponding measurements.
`locate_field_in_lake`	39.6%	field search	Finds every file, sheet, and column in which a query string appears as a header or a cell value, including inside Excel workbooks that `grep` cannot read.
`catalog_data_lake`	37.7%	enumeration	Lists the files under a lake root, following symlinks and optionally filtered by substring, glob, or regex, with a one-line schema per file.
`load_report_csv`	36.5%	CSV loading	Loads one or more report-style CSVs in one call, detecting stacked data sections separated by title rows and blank rows.
`inspect_excel_workbook`	13.8%	workbook loading	Lists every sheet of a workbook with its shape and detected header row, or loads one sheet as a clean typed table.

Table 4: The five most used tools in the KramaBench library evolved with Gemini-3.1-Pro as the Worker under a 10-step budget. Coverage (%) is the share of the 159 test episodes (53 tasks, three repeats) that call the tool at least once.

Tool	Coverage	Role	What it does
`query_lake_catalog`	97.1%	metadata lookup	Answers where a dataset is, which columns and format it has, how big it is, what its known traps are, or which sibling files it has, from a pre-built SQLite catalog of the lake.
`aggregate_over_files`	82.1%	streaming compute	Runs one filter, count, distinct, projection, group-by, or aggregate over a large delimited file or a union of sibling files through DuckDB, without loading the file into memory.
`preview_candidate_columns`	40.4%	schema preview	Shows the columns of a set of candidate files given as paths, a glob, or a dataset slug.
`resolve_filter_values`	32.9%	value matching	Maps a fuzzy category term to the exact distinct values stored in a column, with the row count of each value.
`search_lake_content`	30.8%	full-text search	Finds which dataset, file, or column contains a keyword, phrase, or value through a pre-built full-text index over dataset names and column headers.

Table 5: The five most used tools in the LakeQA library evolved with Gemini-3.1-Pro as the Worker under a 20-step budget. Coverage (%) is the share of the 240 test episodes (80 tasks, three repeats) that call the tool at least once.

into memory. The Worker calls the catalog tool in nearly every LakeQA episode, so almost every solution starts by querying the catalog rather than walking the directory tree.

C STEPS AND COST

C.1 STEP COUNTING

The number of steps a Worker actually uses can differ from its step budget. Figure 4 therefore plots the accuracy of TEAL and the baseline against the steps they actually use under each step budget. The horizontal axis is the mean number of steps per episode over three test repeats, and the vertical axis is the accuracy, so a curve closer to the upper left offers a better trade-off between accuracy and steps. As expected, the evolved tools let the Worker reach higher accuracy with fewer steps.

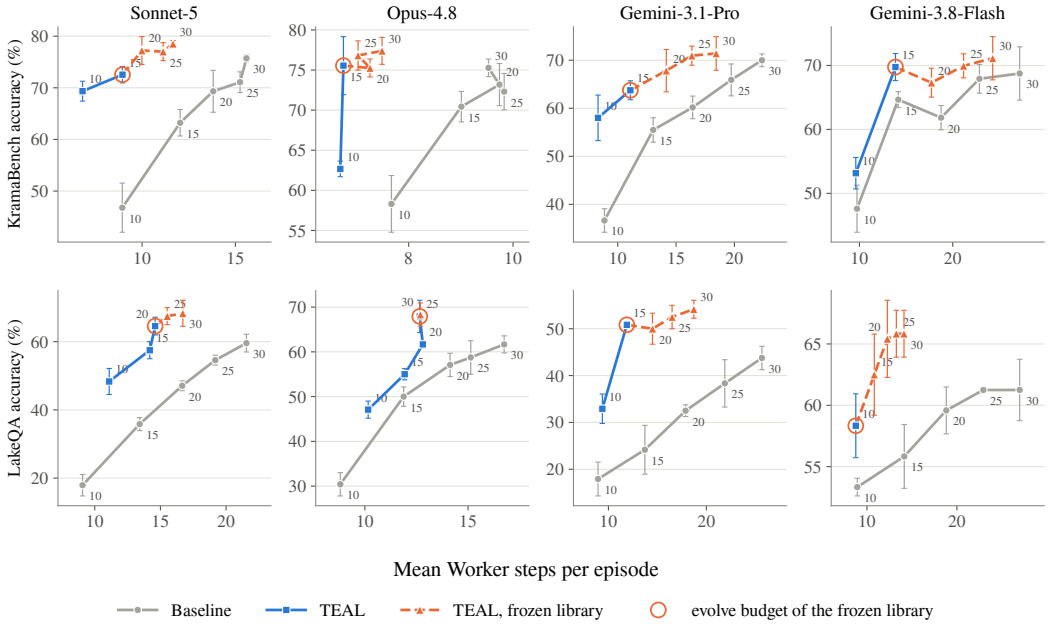

Figure 4: Accuracy versus mean Worker steps per test episode.

C.2 COST ACCOUNTING

A step budget is a practical way to bound the cost and latency of an agent in deployment. A budget stated directly in tokens or in dollars is less practical, since the agent would spend extra tokens tracking how many it has already used. However, a step budget does not reflect the actual cost of an episode. We therefore also plot accuracy against the estimated cost per test episode in Figure 5.

Worker	Uncached input	Cached input	Output
Sonnet-5	2.50	0.20	10.00
Opus-4.8	6.25	0.50	25.00
Gemini-3.1-Pro	2.00	0.20	12.00
Gemini-3.8-Flash	0.75	0.075	3.75

Table 6: Vertex AI list prices (USD per million tokens) used to estimate the cost of a test episode.

We compute the cost of each test episode from three token counts. The *uncached input* is the number of tokens written to the prompt cache for the first time, the *cached input* is the number of tokens read back from the cache on later turns, and the *output* is the number of completion tokens, including hidden reasoning, which both providers bill as output. We assume an optimal caching policy. The prefix, consisting of the system prompt, the tool schemas, and the skills, stays the same across all test episodes with the same method, Worker, lake, and step budget, so it is always read from the cache. Each turn then writes only the tokens added since the previous turn and reads the rest from

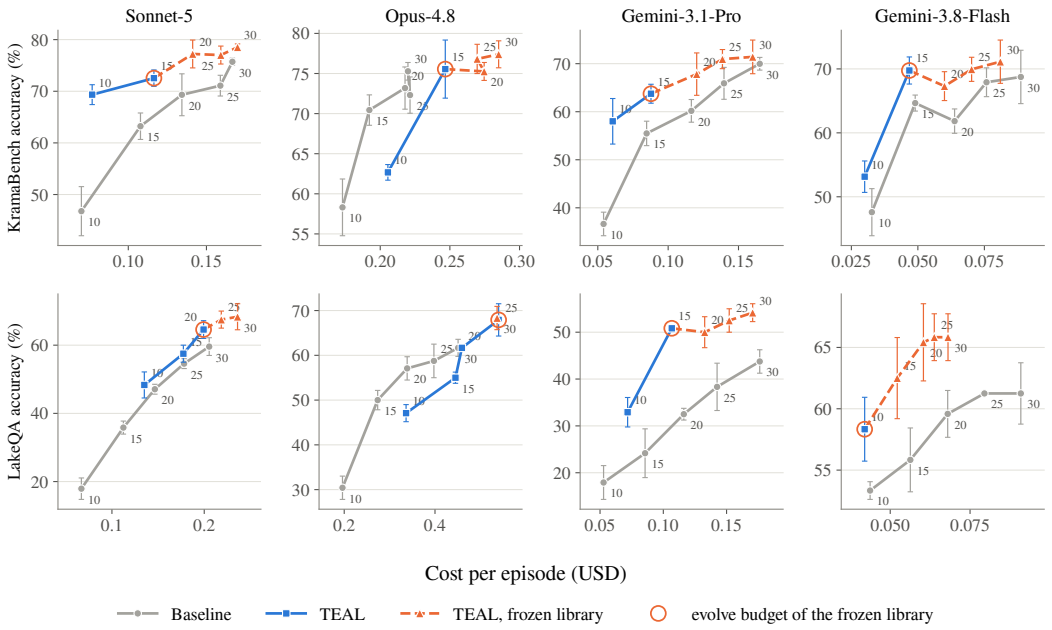

Figure 5: Accuracy versus estimated inference cost per test episode.

the cache. The cost of an episode is then

$$C = n_{\mathrm{u}}\, p_{\mathrm{u}} + n_{\mathrm{c}}\, p_{\mathrm{c}} + n_{\mathrm{o}}\, p_{\mathrm{o}}, \tag{1}$$

where n_{u}, n_{c}, and n_{o} are the uncached input, cached input, and output token counts of the episode, and p_{u}, p_{c}, and p_{o} are the corresponding Vertex AI list prices of the Worker, given in Table 6. Since it assumes no cache misses, this estimate is a lower bound. We report this estimate instead of the billed cost because prompt caching behaved differently across Workers in our runs, so the billed cost mostly reflects the caching configuration of the client, whereas the estimate applies the same accounting to every Worker and method.

Figure 5 is read in the same way as Figure 4, except that the horizontal axis is the mean cost per episode, so a curve closer to the upper left again offers a better trade-off. With Sonnet-5, Gemini-3.1-Pro, and Gemini-3.8-Flash as the Worker, TEAL generally reaches higher accuracy at the same cost, or the same accuracy at a lower cost. Opus-4.8 has the most expensive input tokens, which limits the cost advantage of TEAL, but TEAL remains the better choice when the goal is higher accuracy.

Three factors shape the cost of TEAL. First, TEAL adds several tools and their skills to the prompt, so its fixed prefix is roughly two to three times as long as the baseline's, and the Worker reads this prefix again at every step. Prompt caching absorbs much of this overhead, so we recommend enabling it when deploying TEAL, and we hope that future work on more effective caching will further reduce the cost of such repeated input. Second, perhaps surprisingly, when TEAL shortens an episode, the saving appears in uncached input rather than in output. The result of a tool or shell command is billed as new input at the next step, and an evolved tool returns a compact result in place of the raw intermediate output that the baseline passes through the shell. On KramaBench at 15 steps, for example, TEAL cuts the uncached input per episode by 37% to 63% for Sonnet-5, Gemini-3.1-Pro, and Gemini-3.8-Flash. Third, the output shrinks much less than the number of steps, since the output per step grows for most Workers. We attribute this to the Worker reasoning more at each step, plausibly about which of the additional tools to call.

D Synthetic-Question Scenario Across Workers

Table 7 extends Table 2 to all four Workers.

Model	Methods	KramaBench		LakeQA	
		Acc.	Δ	Acc.	Δ
Sonnet-5	Baseline	$63.23_{\pm 2.57}$	–	$35.83_{\pm 1.91}$	–
	TEAL (real questions)	$72.54_{\pm 1.59}$	+9.31	$57.50_{\pm 2.50}$	+21.67
	TEAL (synthetic questions)	$72.18_{\pm 3.22}$	+8.95	$56.25_{\pm 3.31}$	+20.42
Opus-4.8	Baseline	$70.44_{\pm 1.90}$	–	$50.00_{\pm 2.17}$	–
	TEAL (real questions)	$75.55_{\pm 3.62}$	+5.11	$55.00_{\pm 1.25}$	+5.00
	TEAL (synthetic questions)	$75.73_{\pm 1.26}$	+5.29	$55.42_{\pm 1.91}$	+5.42
Gemini-3.1-Pro	Baseline	$55.48_{\pm 2.54}$	–	$24.17_{\pm 5.20}$	–
	TEAL (real questions)	$63.74_{\pm 1.97}$	+8.26	$50.83_{\pm 0.72}$	+26.67
	TEAL (synthetic questions)	$64.22_{\pm 4.86}$	+8.74	$47.92_{\pm 4.39}$	+23.75
Gemini-3.8-Flash	Baseline	$64.65_{\pm 1.25}$	–	$55.83_{\pm 2.60}$	–
	TEAL (real questions)	$69.75_{\pm 2.12}$	+5.10	$60.42_{\pm 1.91}$	+4.58
	TEAL (synthetic questions)	$70.50_{\pm 2.80}$	+5.85	$59.17_{\pm 1.91}$	+3.34

Table 7: Accuracy (%) of TEAL evolved on real or synthetic questions, compared with the same Worker without evolved tools. All configurations use a 15-step budget and the same held-out test tasks.

[step] step **tool** evolved tool `input` Worker input *Thought* Worker thinking `-> output` tool or shell output

TEAL ✓

```
[step 1] search_dataset_catalog
  query: "violent crime rate by county New York 2018"
  -> {"best_match":
        "index-violent-property-and-firearm-rates-by-county-beginning-1990",
      "dataset_geography_sample": [{"column": "County", "value": "Albany"}, ...],
      "score": 10.104, ...}
[step 2] query_tabular_file
  Thought: This is exactly the dataset. Buffalo (1832) is earlier than
  Rochester (1834), so Buffalo is the terminus city; the other terminus is
  Albany, in Albany County.
  path: "data/datagov/index-violent-property-and-firearm-rates-by-county-beginning-1990/
         files/rows.txt"
  mode: "query"
  sql:  "SELECT County, Year, "Violent Rate" FROM {src}
         WHERE County='Albany' AND Year=2018"
  -> {"rows": [{"County": "Albany", "Year": 2018, "Violent Rate": 359.8}],
      "columns": ["County", "Year", "Violent Rate"], "n_returned": 1, ...}
[step 3] file_editor  create ./answer.txt  "359.8"
```

Baseline ✗, where it wasted steps on data discovery.

```
[step 1] $ find ./data -iname "*crime*"; find ./data -iname "*albany*"
  -> (no output)
[step 2] $ find ./data -type f | head -100
  -> (no output)
[step 3] $ pwd; ls -la; find . -maxdepth 3 -type d; find . -type f | wc -l; ...
  -> lrwxrwxrwx  data -> /.../lake/lakeqa_tables/unified
[step 4] $ find -L ./data -type f | wc -l; find -L ./data -iname "*crime*"; \
           find -L ./data -iname "*albany*"
  -> 4180
     ./data/datagov/crime-incidents-in-2010
     [... 20 more datasets ...]
     ./data/datagov/violent-crime-property-crime-by-county-1975-to-present
     [... 16 more datasets ...]
     ./data/datagov/city-of-albany-five-year-capital-plan-beginning-2013
[step 5] $ find -L \
           ./data/datagov/violent-crime-property-crime-by-county-1975-to-present -type f
  -> ./data/datagov/violent-crime-property-crime-by-county-1975-to-present/
     files/rows.txt
[step 6] $ head -1 ".../rows.txt"; \
           grep -i "^\"Albany\"\|,Albany," ".../rows.txt" | grep ",2018,"
  -> JURISDICTION,YEAR,POPULATION,MURDER,RAPE,ROBBERY,AGG. ASSAULT,B & E,
     LARCENY THEFT,M/V THEFT,GRAND TOTAL,PERCENT CHANGE,VIOLENT CRIME TOTAL,[...]
     (grep matched nothing, exit code 1)
[step 7] $ grep -i "Albany" ".../rows.txt" | head -20
  -> (no output)
[warning] All 7 task steps are used. Please write your answer now.
[step 8] file_editor  create ./answer.txt  "421.8"
```

Figure 6: Trajectories on the Albany County task from Section A, both with Sonnet-5 as the Worker under a 10-step budget. Tool outputs and shell observations are abridged, with elisions marked by `[...]`. The final `finish` step is omitted.

[step] step tool evolved tool input Worker input *Thought* Worker thinking -> output tool or shell output

TEAL ✓

```
[step 1] resolve_concept_to_datasets
  concept: "dust proxy OR wet-dry index OR calcium"
  -> {"candidates": [{"path": "./data/climateMeasurements.xlsx", "relevance": 0.4,
      "kind": "data_table", "ext": "xlsx", "recommended_loader": "inspect_excel_workbook",
      "why": {"content_evidence": {"sheet": "Sheet1", "column_index": 0, "column_name":
      "Data Supplement to: \"A 3 million year index for North African humidity/aridity
      [...]\"", ...}}}, [... 7 more candidates ...]], "n_files_walked": 1742, ...}
[step 2] inspect_excel_workbook
  path: "./data/climateMeasurements.xlsx"   sheet: "Sheet1"
  -> {"artifact_path": ".../inspect_excel_workbook/cache/climateMeasurements__Sheet1.parquet",
      "artifact_format": "parquet", "artifact_rows": 8360, "artifact_cols": 30,
      "headerless": false, "header_index": 5, "header_depth": 1, "score": 1.0,
      "columns": ["Site", "Hole", "Core", "Section", "Interval_cm", "Depth below seafloor_m",
      "Splice depth_m", "Age_ky", "Al", "Si", "S", "K", "Ca", "Ti", "Mn", "Fe", "Rb", "Sr",
      "Zr", "Ba", "col_20", "Age_ky.1", "PC1", "PC2", "col_24", "Age_ky.2",
      "ODP 967 Dust proxy", "col_27", "Age_ky.3", "ODP 967 wet-dry index"],
      "dtypes": {"Site": "str", "Hole": "str", "Core": "float64", ..., "Ca": "float64", ...},
      "preview": [...]}
[step 3] run_python_script
    df = pd.read_parquet(".../climateMeasurements__Sheet1.parquet")
    min_dust = df['ODP 967 Dust proxy'].min()
    subset = df[df['ODP 967 Dust proxy'] == min_dust]
    min_wet_dry = subset['ODP 967 wet-dry index'].min()
    final_subset = subset[subset['ODP 967 wet-dry index'] == min_wet_dry]
    print(final_subset['Ca'].iloc[0])
  -> 207878.877358207
[step 4] run_python_script
    sorted_df = df.sort_values(by=['ODP 967 Dust proxy', 'ODP 967 wet-dry index'])
    print(sorted_df[['ODP 967 Dust proxy', 'ODP 967 wet-dry index', 'Ca']].head(10))
  ->       ODP 967 Dust proxy  ODP 967 wet-dry index             Ca
     5370               0.001               0.059866  207878.877358
     5369               0.001               0.396778  197141.431580
     [...]
[step 5] $ echo "207878.8774" > ./answer.txt
```

Baseline ✗, where it failed to parse the header.

```
[step 1] $ find ./data -type f
  -> (no output)
[step 2] $ ls -la
  -> lrwxrwxrwx  data -> /.../lake/kb_all/unified
[step 3] $ grep -rilE "calcium|dust proxy|wet-dry" ./data
  -> (no output, exit code 1)
[step 4] $ grep -RiL "calcium" ./data || grep -Ril "calcium" ./data
  -> ./data/noaa_wildfires_variabledescrip.csv
     ./data/environmental-justice-populations.csv
     [... 393 more files, climateMeasurements.xlsx among them ...]
[step 5] $ cat << 'EOF' > search.py
           import os
           import pandas as pd
           for root, _, files in os.walk('./data'):
               for f in files:
                   if f.endswith('.csv'):
                       df = pd.read_csv(path, nrows=5)  [...]
                   elif f.endswith('.xlsx'):
                       df = pd.read_excel(path, sheet_name=sheet, nrows=5)
                       cols = [str(c).lower() for c in df.columns]
                       if any('calcium' in c for c in cols) or any('dust' in c ...):
                           print("Found in:", path, "Sheet:", sheet)  [...]
           EOF
           python3 search.py
  -> [An error occurred during execution.] Cannot execute multiple commands at once.
[step 6] $ cat << 'EOF' > search.py && python3 search.py
           [... same script, os.walk('./data', followlinks=True) ...]
  -> (no "Found in:" line, exit code 0)
[step 7] $ grep -ril "dust proxy" ./data
  -> (no output, exit code 1)
[warning] All 7 task steps are used. Please write your answer now.
[step 8] $ echo '0.0000' > ./answer.txt
```

Figure 7: Trajectories on the paleoclimate workbook task from Section A, both with Gemini-3.1-Pro as the Worker under a 10-step budget. Tool outputs and shell observations are abridged, with elisions marked by `[...]`. The final `finish` step is omitted.