SETUPX: CAN LLM AGENTS LEARN FROM PAST FAILURES IN FUNCTIONALITY-CORRECT CODE REPOSITORY SETUP?

Anonymous authors
Paper under double-blind review

ABSTRACT

Functionality-correct repository setup aims to configure execution environments (e.g., dependencies and build scripts) so that a repository's documented features execute successfully. This task is challenging because repository-specific failures include dependency incompatibilities, missing toolchains, incomplete installations, and verification-strategy mismatches. Existing LLM agents lack explicit cross-repository setup memory, safe multi-step recovery from state-changing repairs, and robust verification that separates setup-induced failures from repository defects. We introduce SETUPX, an experiential learning framework comprising a self-evolving eXPerience Unit (***XPU***) representation, LIFO snapshot-backed speculative execution, and a Prosecutor–Judge verification protocol. On a curated 100-repository benchmark, SETUPX achieves a 92% pass rate and outperforms the strongest baseline by 19 percentage points; it also handles multi-repository scenarios that coordinate interconnected services across containers.

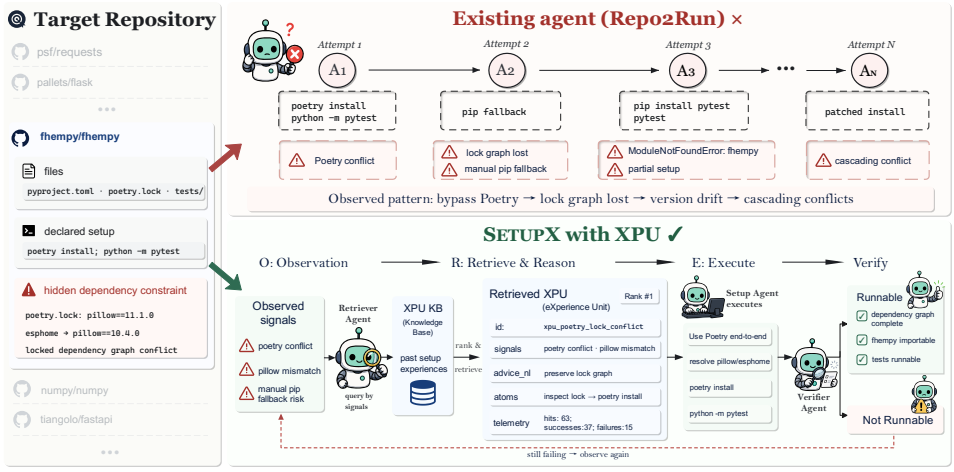


Figure 1: Comparison between existing setup agents (e.g., Repo2Run) and SETUPX.

1 INTRODUCTION

Large language model (LLM)-based agents demonstrate impressive performance on complex software engineering tasks (Hong et al., 2024; Yang et al., 2024; Zhang et al., 2024; Wang et al., 2025; Song et al., 2026). However, most benchmarks over these tasks assume pre-configured environments (Jimenez et al., 2024; Li et al., 2024; Yang et al., 2025), while real-world deployment requires building executable environments from scratch (i.e., functionality-correct repository setup). This process is notoriously complex, involving the resolution of intricate dependency conflicts, the installation of missing build tools, and the reconciliation of version incompatibilities.

Existing LLM-based coding agents (e.g., Claude Code (Anthropic, 2025), OpenHands (Wang et al., 2025)) and specialized setup tools (e.g., Repo2Run (Hu et al., 2025), ExecutionAgent (Bouzenia & Pradel, 2025)) show promise in automating this process. However, they can make significant

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.

setup mistakes in many scenarios. For instance, as shown in Figure 1, in one recorded run on FHEMPY (*Python extension for smart-home automation*), Repo2Run (Hu et al., 2025) repeatedly attempts related repairs: after `poetry install` fails due to a locked dependency conflict, it falls back to `pip install`, causing version drift and cascading dependency errors without extracting reusable diagnostic evidence.

Table 1: Native design comparison. ✓, △, and ✗ denote full, partial, and no support, respectively; partial support indicates generic persistent context or immediate failed-command rollback.

System	Cross-repo experience	Environment-state recovery	Native success signal
General-purpose coding agents	△	✗	Agent-selected checks
ExecutionAgent (2025)	✗	✗	Agent completion + artifact-presence checks
Repo2Run (2025)	✗	△	Image build + test execution
EnvBench (2025)	✗	✗	Static missing-import checks
SETUPX	✓	✓	Independent P–J adjudication

Moving beyond the FHEMPY failure example, we identify three fundamental challenges for effective automatic setup (Table 1): ❶ First, *Explicit Cross-Repository Failure Memory:* General-purpose agents may inherit broad debugging priors from pretraining, but existing setup agents lack an explicit, auditable, setup-specific memory whose entries can be traced to deployment evidence and corrected from subsequent outcomes. ❷ Second, *State Changes without Automatic Inverses:* Package-manager and system-level actions can leave persistent side effects after the command that caused them. Recovery is possible with virtual environments or container checkpoints, but a mutable shell does not provide a reliable per-action inverse unless recovery points are planned in advance. ❸ Finally, *Unreliable Success Signals:* There is no unified, trustworthy criterion for setup completion. Current systems either make their online completion decision within the same agent-driven workflow used for setup and execution (e.g., Claude Code (Anthropic, 2025), ExecutionAgent (Bouzenia & Pradel, 2025)), or rely on shallow objective proxies (e.g., Repo2Run's (Hu et al., 2025) `pytest`-oriented test-execution feedback and EnvBench's (Eliseeva et al., 2025) static missing-import checks) that cannot reflect functional usability or attribute failures to their root cause. These native signals therefore do not provide independent, per-failure causal attribution.

Collectively, these challenges motivate our research question: *Can a unified agentic framework systematically transfer cross-repository experience, safely explore complex setup states, and reliably verify setup outcomes within a single continuous loop?*

To answer this question, we introduce SETUPX, an experience-driven framework for autonomous repository setup. Our contribution is a setup-specific systematic design that connects explicit experience reuse, state-safe exploration, and independently verified outcomes.

In summary, SETUPX makes four contributions: (1) a structured, self-curating ***XPU*** representation that connects diagnostic signals, natural-language guidance, executable repairs, and delayed outcome telemetry; (2) snapshot-backed speculative execution that supports safe multi-step rollback; (3) a task-specific Prosecutor–Judge verification protocol that independently adjudicates setup-induced failures; and (4) evaluation on 100 single-repository tasks and 22 multi-repository families, where SETUPX reaches 92% pass rate and improves over the strongest baseline by 19 pp.

2 PROBLEM DEFINITION

Prior works (Hu et al., 2025; Bouzenia & Pradel, 2025) primarily verify the configured environment by executing the project's test suite, leaving documented user-facing commands unchecked. In this work, we consider a stricter setting, termed *functionality-correct repository setup*. Given a code repository $\mathcal{R} = (\mathcal{P}, S)$, where $\mathcal{P}$ denotes the source code and S specifies a commit, let $\mathcal{F}_{\mathcal{R}}$ denote the set of functionality-oriented execution targets declared by the repository (e.g., test suite, documented commands). The goal is to construct a valid execution environment $\mathcal{E}$ that (1) is realized as a reproducible container image produced either by building from a Dockerfile (as adopted by prior methods such as Repo2Run (Hu et al., 2025)) or by committing an interactively configured container, and (2) executing the targets in $\mathcal{F}_{\mathcal{R}}$ on this image does not produce any setup-induced failure:

$$\mathrm{Valid}(\mathcal{E}, \mathcal{R}) \iff \mathcal{E} \in \mathcal{D} \ \wedge \ \forall f \in \mathcal{F}_{\mathcal{R}},\ \mathrm{Run}(f, \mathcal{E}) \notin \mathcal{C}_{\mathrm{fail}} \tag{1}$$

where $\mathcal{D}$ denotes the space of reproducible container images, and $\mathcal{C}_{\mathrm{fail}}$ denotes the set of setup-induced failures: failures of a repository's declared core tests or documented entry points that are

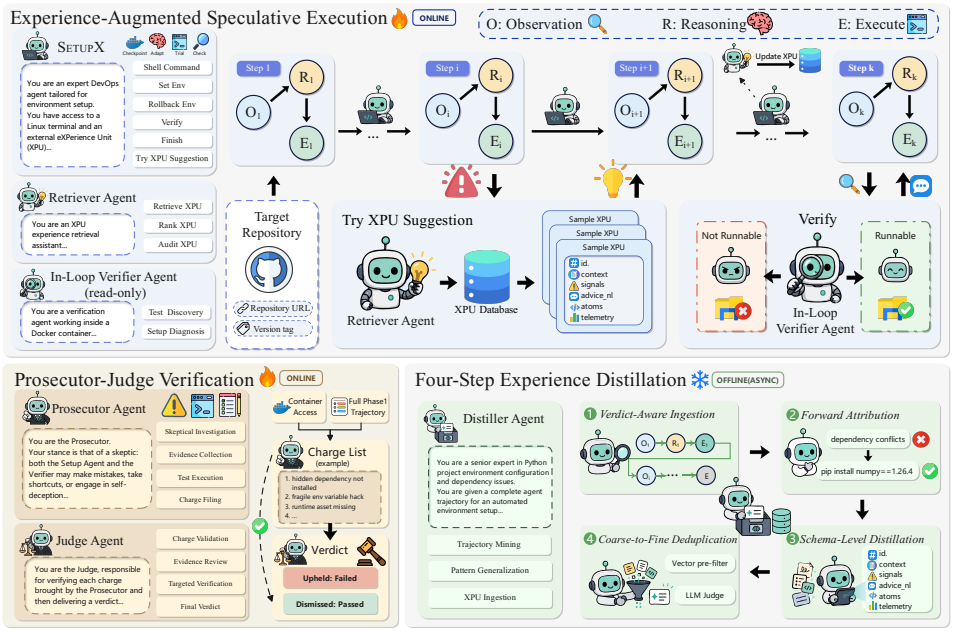


Figure 2: Overview of SETUPX. Environment setup proceeds in a ReAct loop with ***XPU***-guided speculative execution and in-loop verification; Prosecutor–Judge verification independently audits the result, and transferable lessons are ingested into the ***XPU Database***.

caused by missing or inconsistent environment configuration. Failures caused by source-code or test-logic defects, flaky tests, unavailable external services, optional dependencies, missing private data or credentials, and unsupported GPU or platform requirements are treated as project-intrinsic or externally constrained and are excused.

To address the three challenges for effective setup, SETUPX (Figure 2) relies on a self-curating knowledge base of *eXPerience Units* (***XPU***), which is continuously expanded by distilling offline execution trajectories for cross-repository deployments (Section 3). To safely apply this knowledge within the irreversible container space $\mathcal{D}$, the agent tests retrieved ***XPU***s using snapshot-backed *speculative execution* (Section 4). Finally, Prosecutor–Judge verification operationalizes this goal through executable checks (Section 5); a pass indicates no detected setup-induced failure, not a proof of correctness for every declared target.

3 SELF-EVOLVING EXPERIENCE REPRESENTATION

Existing memory-augmented agents struggle to efficiently accumulate and transfer setup experience, often resorting to storing verbose, non-generalizable trajectories. Moreover, natural-language and persistent memories (Shinn et al., 2023; Park et al., 2023; Zhao et al., 2024; Packer et al., 2023; Zhong et al., 2024) store experience that can be hard to index by exact setup evidence or calibrate from downstream utility, while executable skill and tool libraries (Wang et al., 2024; Cai et al., 2024; Wu et al., 2024; Chen et al., 2026) provide reusable actions but lack structured indexes for error matching. To resolve this, SETUPX maintains cross-repository setup experience in a self-evolving database: (1) the *Structured eXPerience Unit (**XPU**)*, a formalized representation mapping setup failures to illustrative advice and executable operations; (2) an offline *Four-Step Experience Distillation* pipeline that expands the ***XPU*** base by transforming raw trajectories into deduplicated records; and (3) an online *Anchor-Based Delayed Audit* that calibrates each ***XPU*** entry based on real deployment feedback.

► **(1) Structured eXPerience Unit (*XPU*).** To effectively bridge reasoning with execution, provide exact error context, and track historical success, we propose the *eXPerience Unit* (***XPU***), a structured knowledge record that natively integrates diagnostic context, natural language guidance, and executable code. Formally, an ***XPU*** is defined as: $\textbf{\textit{XPU}} = \langle id,\ signals,\ advice_nl,\ atoms,\ telemetry \rangle$, where *signals* indexes the exact error context for precise retrieval, including applicability conditions

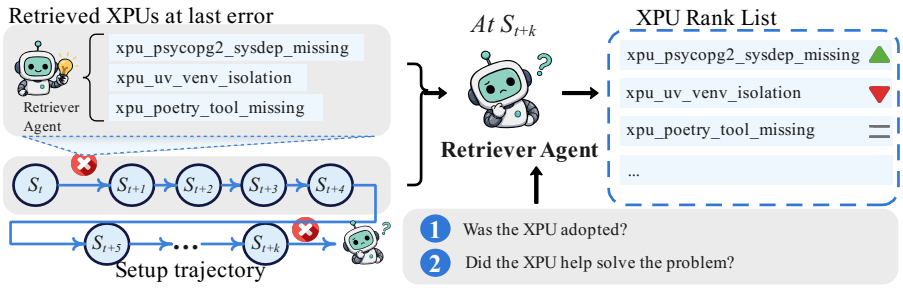


Figure 3: Delayed audit feedback loop. When a new retrieval is triggered, the Retriever Agent first audits the previous ***XPU*** recommendations against the subsequent setup trajectory, assigns verdicts, and updates per-***XPU*** telemetry that feeds back into future retrieval ranking.

(e.g., OS, build tools) and specific error messages; *advice_nl* and *atoms* form a dual representation to link reasoning with action, explaining the root cause in natural language and encoding the fix as executable operations, respectively; and *telemetry* enables online self-curation by tracking a deployment record ($\langle hits, successes, failures \rangle$) that the delayed audit updates and the retriever consumes.

► **(2) Four-Step Experience Distillation.** To systematically incorporate new ***XPU*** entries and drive continuous evolution, we propose an offline LLM-based distillation process. This process serves as the generation stage for new ***XPU*** entries in the database lifecycle and evaluates every execution trajectory together with its past evaluation signals through a four-step pipeline (Figure 2): ❶ *Verdict-Aware Ingestion:* A Distiller Agent analyzes the trajectory using evaluation signals, extracting corrective insights from actions, rather than relying exclusively on the agent's self-reported successes; ❷ *Forward Attribution:* Since error recovery is rarely linear and the true impact of a fix frequently emerges steps later, an attribution pass localizes concrete environment problems (e.g. dependency conflicts, build-tool failures) and traces them forward to the specific actions that resolved them. A trajectory is discarded when its root cause or the action responsible for recovery cannot be attributed reliably; ❸ *Schema-Level Distillation:* An LLM maps each problem-fix pair into the ***XPU*** schema, abstracting the transferable reasoning pattern instead of merely recording exact commands; ❹ *Coarse-to-Fine Deduplication:* For each newly distilled candidate, a vector pre-filter compares it against the existing entries in the ***XPU Database*** and retrieves potential duplicates whose cosine similarity is at least 0.85; an LLM judge then verifies whether the shortlisted entries are semantically equivalent. Confirmed duplicates trigger a merge (unioning signal sets and fusing advice), while candidates without an equivalent database entry are ingested as novel experiences.

► **(3) Anchor-Based Delayed Audit.** To further reliably assess ***XPU*** quality and capture delayed feedback (where real impact surfaces steps later when downstream dependencies resolve), we implement an online anchor-based retrospective evaluation. This mechanism serves as the audit stage for existing ***XPU*** entries in the database lifecycle. As shown in Figure 3, every retrieval records the recommended ***XPU*** IDs and the current trajectory position. At the next retrieval, the Retriever Agent audits the intervening execution steps and labels each previous recommendation as a *success*, *failure*, or *neutral*. These outcomes update the entry's telemetry, which influences its composite ranking score (Section 4.1). Repeatedly useful entries are promoted and receive greater retrieval weight, whereas consistently unhelpful entries move to the *Cold* tier and are down-weighted. Cold entries remain in the database and can still be retrieved when semantically relevant, allowing later positive evidence to restore their rank rather than permanently discarding potentially useful experience. Appendix A.2 details the audit protocol and update rules; Appendix B provides prompt excerpts.

4 Experience-Augmented Speculative Execution

Once SetupX has access to reusable ***XPU***s, the next challenge is to apply them safely inside a mutable container. To address this, SetupX augments the standard ReAct paradigm (Yao et al., 2023) with three mechanisms: (1) *Agentic Experience Retrieval* (Section 4.1); (2) *Speculative Execution* (Section 4.2); (3) *In-Loop Verifier* (Section 4.3). At each step, SetupX first observes the container and the last action's output; it automatically invokes the Retriever Agent to identify relevant ***XPU***s based on setup failures, which are added to the prompt. SetupX then reasons over recent history together with the retrieved natural-language guidance and executable atoms, and selects

one of six actions (provided in Appendix C), where three are specialized for experience-driven setup: `TRY_XPU_SUGGESTION`, `ROLLBACK_ENV`, and `VERIFY`. The setup loop terminates when SETUPX emits the `FINISH` action.

4.1 AGENTIC EXPERIENCE RETRIEVAL

Naive embedding-based search over raw error messages yields low retrieval precision since syntactically similar errors in setup may have entirely different root causes (e.g., `ModuleNotFoundError` for a missing system library *vs.* a missing pip package), while semantically related problems may produce syntactically different error text. To address this, SETUPX uses a dedicated *Retriever Agent* that operates in an independent context, shielding the main agent's prompt from retrieval noise. The Retriever Agent constructs a *hybrid situation* (an LLM-generated current state summary paired with the raw command output and error text) and runs coarse-to-fine retrieval against the knowledge base.

Coarse-to-Fine Retrieval. We first perform vector coarse filtering: the hybrid situation is embedded via `text-embedding-3-small` and queried against a pgvector index. The top-N (N=10) candidates are ranked by a composite score:

$$s_{\text{composite}} = s_{\text{sim}} \cdot (1 + r_{\text{success}}) \cdot b_{\text{tier}}, \qquad b_{\text{tier}} = \begin{cases} 1.5, & \text{if } \textit{hits} \geq 5 \text{ and } r_{\text{success}} \geq a \quad (\textit{Golden}), \\ 0.6, & \text{if } \textit{hits} \geq 5 \text{ and } r_{\text{success}} < b \quad (\textit{Cold}), \\ 1.0, & \text{otherwise} \quad (\textit{Normal}). \end{cases} \tag{2}$$

where s_{sim} is cosine similarity, $r_{\text{success}} = \textit{successes}/\max(\textit{hits}, 1)$ is the historical success rate, and b_{tier} is a tier boost. We set the thresholds to a=0.2 and b=0.1 based on current ***XPU*** telemetry, yielding a roughly balanced split across the three tiers while assigning entries with insufficient evidence to *Normal*. This score combines semantic relevance with empirical utility, so experiences with high similarity but low success rates are naturally deprioritized. Finally, an LLM refines the ranking by reading the full content and selecting the top-K most relevant entries, where K=3 in our experiment.

4.2 SPECULATIVE EXECUTION

Environment repair is *non-monotonic*: many package-manager and system actions have no automatic per-action inverse, and an earlier fix may appear successful but introduce a latent conflict that surfaces only several steps later, leaving the offending step buried beneath subsequent attempts. To safely explore candidate fixes under such delayed feedback, we give the agent *active*, multi-step control over the trajectory through a LIFO snapshot stack $\mathcal{S}$ exposed via two actions.

❶ **`TRY_XPU_SUGGESTION`** pushes a checkpoint of the current container onto $\mathcal{S}$ via Docker's copy-on-write `docker commit`, then trials the adapted ***XPU*** commands:

$$\mathcal{C}_{t+1} = \begin{cases} \mathcal{C}'_t & \text{if } \texttt{trial}(\mathcal{C}'_t) = \textit{success}, \\ \texttt{restore}(\mathcal{S}.\texttt{pop}()) & \text{otherwise.} \end{cases} \tag{3}$$

where the new state is retained if the trial succeeds; otherwise, the just-pushed snapshot is restored in place. The full five-stage trial protocol is provided in Appendix C.1.

❷ **`ROLLBACK_ENV`** lets the agent *actively* pop any number of frames off $\mathcal{S}$ and return the container to *any* earlier known-good checkpoint, not only the latest one. Once subsequent diagnostics reveal that several past attempts have collectively steered the trajectory into a dead end, the agent invokes `ROLLBACK_ENV` to retreat past all of them and explore an alternative path.

Unlike Repo2Run's immediate command-level rollback (Hu et al., 2025), SETUPX exposes an agent-controlled LIFO snapshot stack for safe multi-step exploration beyond immediate failure recovery, so accepted steps can be reverted when delayed diagnostics reveal dead ends.

4.3 IN-LOOP VERIFIER

The In-Loop Verifier is a lightweight read-only ReAct agent invoked when SETUPX issues `VERIFY`. It operates within the same container under strict constraints: *(1) Read Only*, prohibited from modifying environment variables or altering project files; *(2) Structured Protocol*, fixed sequence of structural reconnaissance, test suite location, test execution via the project's native runner, and failure analysis; *(3) Dual-Outcome Semantics*, distinguishes setup-induced failures (actionable) from inherent project limitations (non-blocking for `FINISH`). The result is a per-step, attribution-aware diagnostic signal that lets SETUPX drive targeted repair rather than blind retry.

5 PROSECUTOR–JUDGE VERIFICATION PROTOCOL

Existing verification either lets the setup agent judge its own success (Anthropic, 2025; Wang et al., 2025) or returns external binary signals without failure attribution (Eliseeva et al., 2025; Hu et al., 2025). To address this, we introduce a role-separated *Prosecutor–Judge protocol*: a structurally independent post-hoc audit that investigates and adjudicates outcomes separately from setup process.

Prosecutor Agent. The Prosecutor operates as a skeptical investigator with full container access, following a forced multi-step investigation protocol designed to prevent superficial assessments: (1) identify the project's language and build system by examining marker files; (2) review the full trajectory during setup; (3) verify that core dependencies are importable using the interpreter configured by the setup agent; (4) consult the README for documented entry points (e.g., example scripts, CLI commands) and verify a representative subset launches without setup-induced failure; (5) run the test suite independently and compare results; (6) formulate specific charges backed by concrete evidence, following the principle "when in doubt, prosecute".

Judge Agent. The Judge receives the Prosecutor's charges and performs *targeted verification*: for each charge, it executes 1–2 independent commands to confirm or refute the evidence. A charge is dismissed if the evidence is contradicted or the issue is attributable to source or test defects, flaky behavior, optional dependencies, unavailable external services, missing private data or credentials, or unsupported hardware and platform requirements. If any charge is upheld, the setup is ruled *guilty*; and *not_guilty* otherwise.

This separation creates an effective check: the Prosecutor finds possible failures, while the Judge verifies them independently, making the final verdict more reliable. The protocol serves as an independent evaluator of setup success and is applied uniformly to SETUPX and all baselines.

6 EXPERIMENTS

6.1 BENCHMARK CONSTRUCTION

Our evaluation is built on the EnvBench Python challenge pool, which contains 329 Python repositories (Eliseeva et al., 2025). We focus on Python because its dependency ecosystem is large and complex, its packaging practices are heterogeneous, and it frequently involves native-library interactions, all of which make automated setup particularly challenging (Bommarito & Bommarito, 2019; Wang et al., 2020). From this repository pool, we construct our benchmark, a curated benchmark of 100 repositories stratified into three difficulty levels for systematic comparison. The construction proceeds in three steps: we first retain repositories with detectable test assets; then run a preliminary agent-based trial to stratify repositories by setup difficulty; finally, sample across application domains and project maturity to obtain the final benchmark. For example, `lark` is a low-difficulty project because it has minimal dependencies and a straightforward installation process. In contrast, `acme` is a high-difficulty project because it involves optional ecosystems such as TensorFlow and JAX, with strict version coupling that causes the agent to fail even after multiple setup retries. The remaining 229 repositories in EnvBench Python challenge pool outside our benchmark form a disjoint development pool for SETUPX; their setup trajectories exclusively seed the initial ***XPU Database***.

The final benchmark contains 44 low-, 37 medium-, and 19 high-difficulty repositories. It covers nine major application areas and 148 domain tags, and each repository is pinned to a fixed revision. As shown in Figure 4, the benchmark covers diverse application domains and project scales.

We also construct a multi-repository track with 22 composite scenarios, covering four representative patterns: client–server, plugin–host, multi-service deployment, and platform–SDK. Because these tasks require workflow-level evidence across repositories, we validate them through manual review and use this track as a complementary stress test. The full rubric is placed in Appendix D. Each family is rated on a four-tier scale (Full / Mostly / Partial / Shallow) based on a five-check rubric covering clone integrity, editable install, module availability, pytest collection, and dependency consistency.

6.2 EXPERIMENTAL SETUP

We evaluate eight systems on our benchmark: SETUPX with and without ***XPU***; three CLI coding agents, Claude Code (Anthropic, 2025), Qwen Code (Qwen Team, 2025), and OpenCode (OpenCode Team, 2025); and three specialized setup tools, ExecutionAgent (Bouzenia & Pradel, 2025), Repo2Run (Hu et al., 2025), and EnvBench (Eliseeva et al., 2025). All use `qwen3.5-plus` for setup and unified P–J adjudication. Each run starts from a fresh `ubuntu:22.04` container with

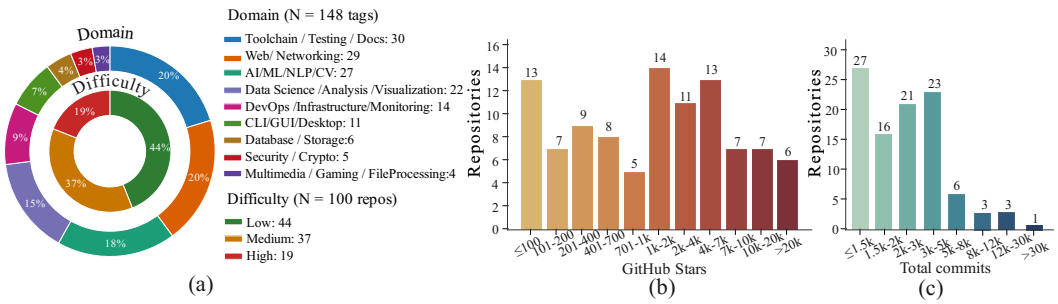


Figure 4: Statistics of the setup-specific benchmark.

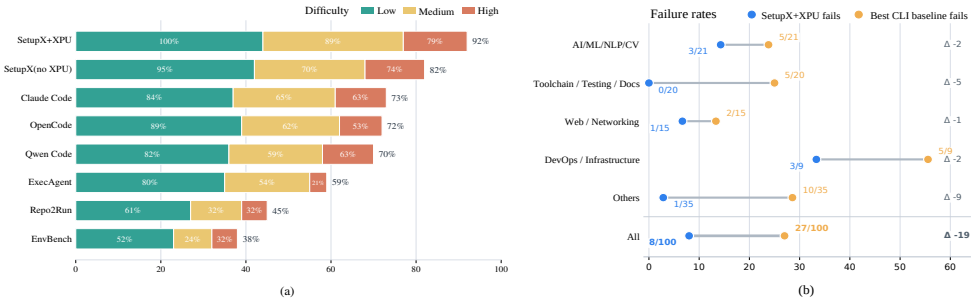


Figure 5: Main results on the single-repository track. (a) Setup pass rates by repository difficulty; (b) Domain-wise failure rates of SETUPX+XPU versus the best CLI baseline.

the same unrestricted root shell, full privileges, infrastructure, and 3600-second timeout; we allow baselines to checkpoint or restore through available shell and Docker mechanisms themselves.

The ***XPU Database*** is built only on the 229 development repositories, with online telemetry updates during experience generation. Distillation merges 705 candidates into 600 entries indexed by 1536-dimensional `text-embedding-3-small` embeddings (Appendix H). Entries and telemetry are frozen throughout benchmark evaluation, including repeated-seed runs; test trajectories do not update the database. A run passes if the Prosecutor files no accusation or the Judge rejects all accusations; timeout or any *guilty* verdict counts as failure.

6.3 Main Results

We evaluate two complementary capabilities: functionality-correct setup on the 100 single-repository tasks, and workflow-level completion on the 22 multi-repository families. The single-repository results are stratified by setup difficulty and further analyzed by repository domain (Figure 5).

On the single-repository track, SetupX+***XPU*** ranks first at 92%, exceeding Claude Code by 19 pp, ExecutionAgent by 33 pp, and the other specialized tools by 47–54 pp. The advantage is more pronounced on medium- and high-difficulty repositories: SetupX+***XPU*** succeeds on 48 of these 56 tasks versus 36 for Claude Code, a 12-task (21 pp) advantage. Compared with low-difficulty tasks, these repositories more often require dependency-version conflict resolution or toolchain repair alongside recovery from contaminated states, linking the gain to multi-step repair rather than routine installation. Across three runs, SetupX obtains $91.7 \pm 0.6\%$ versus $72.7 \pm 1.5\%$ for the strongest CLI baseline (Appendix G). We examine the contribution of each component in Section 6.4.

The multi-repository track evaluates setup completeness across interconnected repositories: every participating repository must be installed, importable, and dependency-consistent. Both SetupX and Qwen Code reach Full or Mostly on 16 of 22 families, but SetupX obtains 6 Full verdicts versus 1. Qwen Code often starts sibling services or registers plugins yet remains at Mostly because `PATH` is not linked to the virtual environment, `pytest` is absent, or required sibling services remain unavailable. The Full–Mostly gap thus reflects differences in setup completeness, particularly whether installation and dependency requirements are fully satisfied (Appendices E and F).

Across domains, SetupX+***XPU*** reduces the total number of failures from 27 to 8 and has no more failures than Claude Code in any domain. The largest gap is among 20 Toolchain repositories (0 vs.

5), where reusable patterns directly address missing compilers, build utilities, and native libraries. The remaining SETUPX failures concentrate in AI/ML and DevOps stacks whose repository-specific dependencies are harder to match (Figure 5(b)).

To characterize failure modes, we assign each failed run to one primary cause: incompatibility (C1), toolchain gaps (C2), incomplete installation (C3), or verification mismatch (C4). Claude Code's failures are relatively dispersed. Repo2Run concentrates 80% in C3 (47%) and C4 (33%), exposing weaknesses in setup completeness and success assessment. SETUPX+***XPU*** has fewer failures in every category; six of its eight failures are C3 (four) or C1 (two), leaving repository-specific installation and version compatibility as its main bottlenecks rather than toolchain discovery or verification (Table 2).

Table 2: Failure attribution on the 100-repository benchmark.

Category	Description	Claude Code (N=27)	Repo2Run (N=55)	SETUPX+***XPU*** (N=8)
C1	Dependency / runtime version incompatibility	6/27 (22%)	4/55 (7%)	2/8 (25%)
C2	Native / build toolchain gaps	4/27 (15%)	7/55 (13%)	1/8 (12.5%)
C3	Invalid or incomplete package installation	10/27 (37%)	26/55 (47%)	4/8 (50%)
C4	Verification strategy mismatch	7/27 (26%)	18/55 (33%)	1/8 (12.5%)

6.4 ABLATION STUDY

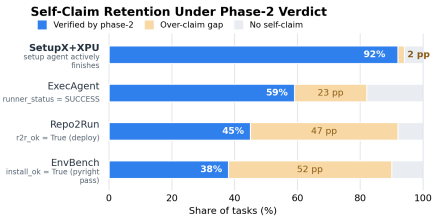
We examine four questions in this part: whether P–J provides reliable verdicts, which SETUPX mechanisms drive end-to-end performance, whether the ***XPU*** representation and reranking pipeline retrieve useful experience, and whether these gains require greater inference cost.

We validate P–J from two complementary perspectives. First, two domain experts manually audit 700 final containers, 100 each from SETUPX+***XPU***, ExecutionAgent, Repo2Run, EnvBench, Claude Code, OpenCode, and Qwen Code, using the same blinded protocol with randomized presentation order. The experts independently assign pass/fail labels before resolving disagreements through discussion. On the 400 outputs from SETUPX+***XPU*** and the three specialized tools, P–J achieves 95–98% agreement with the resulting expert labels. Second, for each specialized setup tool, we compare its native success signal with P–J on that tool's own 100 outputs. Native self-claims exceed the corresponding P–J pass rates by 23–52 pp and reject only 14.3–45.0% of expert-confirmed failures, revealing a strong false-positive bias. P–J instead achieves 95–97% accuracy on each tool's outputs. For example, 92% of the environments configured by Repo2Run pass the `pytest -collect-only` check, whereas only 45% pass P–J and 44% pass expert audit. Together, these results show that P–J assesses functionality-correct setup substantially more accurately than single completion signals and shallow verification proxies (Figure 6a; Appendix I).

To analyze the contribution of each SETUPX component, we evaluate all combinations of ***XPU***, speculative execution, and the In-Loop Verifier on the 100-repository benchmark. Starting from the matched ReAct control (41%), ***XPU***, speculative execution, and the verifier alone reach 69%, 61%, and 58%, respectively, making ***XPU*** the strongest standalone component. Combining ***XPU*** with speculative execution produces the strongest pair (85%), suggesting that retrieved repairs benefit from rollback-protected trials. By contrast, ***XPU*** with the verifier is the weakest pair (78%), consistent with the verifier detecting unsuccessful setups but being unable to reverse state contamination. Speculative execution with the verifier reaches 82%, indicating that safe exploration and feedback remain effective without explicit retrieved experience; adding ***XPU*** raises performance to 92% by supplying repository-specific guidance. Together, these components enable SETUPX to apply past experience safely and verify the resulting environment rigorously (Table 3).

Retrieval ablations isolate the effects of reranking and ***XPU Database*** quality. Full-content selection reaches 92% versus 86% for direct top-3 vector retrieval on the Clean KB, and 88% versus 81% under 3:1 synthetic noise. Reranking therefore contributes 6–7 pp and buffers embedding-neighbor noise. Replacing `qwen3.5-plus` with Claude Opus 5.5 leaves the pass rate unchanged at 92%, suggesting that SETUPX's gains arise primarily from its architecture and the ***XPU Database*** rather than from using a stronger backbone (Figure 6b).

To further examine whether the setup-specific ***XPU*** representation is more effective than a general-purpose code-skill representation, we conduct an end-to-end representation-swap experiment on our benchmark. We replace the ***XPU Database*** with a noise-free Voyager-style code-skill library (Wang et al., 2024), built from the same development trajectories and matched in size. With all other setup and evaluation settings fixed, this configuration reaches 86% versus 92% for ***XPU*** (Figure 6b). The



(a) Self-claim vs. Prosecutor–Judge verdict.

Setting	Repr.	Rerank	KB	Backbone	Pass	Cost
No-XPU	–	–	–	qwen3.5-plus	82	1.0×
Selector + Clean ⋆	XPU	Selector	Clean	qwen3.5-plus	**92**	1.0×
Direct + Clean	XPU	Direct	Clean	qwen3.5-plus	86	1.0×
Selector + Code skill	Code skill	Selector	Clean	qwen3.5-plus	86	1.0×
Selector + Noisy	XPU	Selector	Noisy	qwen3.5-plus	88	1.0×
Direct + Noisy	XPU	Direct	Noisy	qwen3.5-plus	81	1.0×
Selector + Clean + Opus	XPU	Selector	Clean	Claude Opus 5.5	92	∼15×

(b) Retrieval and representation ablations. Repr.: experience representation; KB: library quality.

Figure 6: Trustworthiness of Prosecutor–Judge verdicts and retrieval ablation results.

Table 3: Component ablations of SETUPX (Spec.: speculative execution; Ver.: In-Loop Verifier).

Configuration	XPU	Spec.	Ver.	Pass	Configuration	XPU	Spec.	Ver.	Pass
Full SETUPX	✓	✓	✓	**92**	Only XPU	✓	✗	✗	69
−verifier	✓	✓	✗	85	Only speculative execution	✗	✓	✗	61
−XPU	✗	✓	✓	82	Only verifier	✗	✗	✓	58
−speculative execution	✓	✗	✓	78	Matched ReAct control	✗	✗	✗	41

gap suggests that setup-specific experience representation provides additional value beyond reusing executable code skills alone. Expert review additionally judges 98.7% of the 600 distilled XPUs technically accurate and 99.3% generalizable beyond their source repository (Appendix H).

In terms of efficiency, under the shared `qwen3.5-plus` backbone, SETUPX consumes 0.53M total tokens per repository, whereas the general coding agents consume 3.4–4.5× as many. SETUPX also requires comparable or less wall-clock time (18.3 vs. 18.6–24.7 minutes). Its performance gain therefore does not come from a larger token budget or longer runtime (Appendix G).

7 RELATED WORK

► **LLM Agents for Repository Setup.** Recent work such as ExecutionAgent (Bouzenia & Pradel, 2025), Repo2Run (Hu et al., 2025), and EnvBench (Eliseeva et al., 2025) has begun to study setup phase across test-execution, Dockerized repository setup, and research-repository reproduction settings. However, these existing setup methods and evaluations still treat each repository or task as an isolated episode, so a fix discovered for one project cannot directly inform future repositories.

► **Experience and Skill Reuse in LLM Agents.** Prior work explores several forms of reusable agent experience. Executable skill libraries such as Voyager (Wang et al., 2024) and SkillCraft (Chen et al., 2026) store programs or tool-use routines for retrieval and composition. Natural-language memories such as Reflexion (Shinn et al., 2023) and ExpeL (Zhao et al., 2024) distill trajectories into verbal feedback or transferable insights. More recent systems, including Agent-KB (Tang et al., 2025), EvolveR (Wu et al., 2025), and Mem^p (Fang et al., 2025), organize, refine, and maintain experience. These approaches primarily organize memory around general tasks or workflows rather than setup-specific diagnostic signals. ***XPU*** couples these signals with natural-language advice, executable repair atoms, and per-entry outcome telemetry.

► **Verification and Quality Assurance.** Existing tools assess setup through static import checks (Eliseeva et al., 2025), Docker build and test executability checks (Hu et al., 2025), or test execution results (Bouzenia & Pradel, 2025). These checks alone do not establish whether failures arise from setup errors, repository defects, or external constraints. CourtEval (Kumar et al., 2025) and Agent-as-a-Judge (You et al., 2026) explore role-separated or agentic evaluation of LLM outputs. SETUPX adapts this approach to repository setup, independently checking each charge through targeted container commands to establish failure attribution.

8 CONCLUSION

We presented SETUPX, combining explicit self-curating cross-repository experience, snapshot-backed speculative execution, and a structurally independent Prosecutor–Judge protocol for functionality-correct repository setup. Across 100 repositories, it substantially outperforms both general purpose and specialized baselines, turning environment setup from a brittle bottleneck into an experience-accumulating, independently verifiable component of the LLM-agent stack.

AI USE STATEMENT

Generative AI models are integral components of the proposed and evaluated agents as described in the method and experimental sections. Separately, we used generative AI tools to assist with organizing related works and formatting the manuscript. The benchmark repositories were selected through the stated construction procedure, and all reported quantitative results were computed from recorded executions. We reviewed and verified all AI-assisted content and take full responsibility for the final manuscript.

ETHICS STATEMENT

Our work evaluates automated environment setup on publicly available open-source repositories. It involves no human participants and does not collect private or personally identifiable data. Experiments execute third-party code in disposable containers and do not modify upstream repositories. Automated dependency installation and code execution can introduce security and software-supply-chain risks; SETUPX is therefore intended for authorized repositories and isolated environments, and generated commands and dependencies should be reviewed before deployment. Third-party repositories remain governed by their original licenses.

REPRODUCIBILITY STATEMENT

The experimental protocol, benchmark construction procedure, environment configuration, evaluation criterion, and component settings are described in Sections 6.1–6.4. Appendix K reports the complete 100-repository benchmark list and the per-repository outcomes of all evaluated systems, while the remaining appendices provide prompts, action definitions, ablations, verification analyses, and representative execution cases. An anonymized supplementary archive will accompany the submission with our implementation of the SETUPX agent architecture, prompts, container utilities, and scripts needed to reproduce the experiments.

REFERENCES

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2025. GitHub repository, accessed 2026-09-07.

Ethan Bommarito and Michael J. Bommarito, II. An empirical analysis of the Python Package Index (PyPI). *CoRR*, abs/1907.11073, 2019. URL http://arxiv.org/abs/1907.11073.

Islem Bouzenia and Michael Pradel. You name it, I run it: An LLM agent to execute tests of arbitrary projects. *Proc. ACM Softw. Eng.*, 2(ISSTA):1054–1076, 2025. doi: 10.1145/3728922. URL https://doi.org/10.1145/3728922.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/ed91353f700d113e5d848c7e04a858b0-Abstract-Conference.html.

Shiqi Chen, Jingze Gai, Ruochen Zhou, Jinghan Zhang, Tongyao Zhu, Junlong Li, Kangrui Wang, Zihan Wang, Zhengyu Chen, Klara Kaleb, Ning Miao, Siyang Gao, Cong Lu, Manling Li, Junxian He, and Yee Whye Teh. SkillCraft: Can LLM agents learn to use tools skillfully? *CoRR*, abs/2603.00718, 2026. doi: 10.48550/arXiv.2603.00718. URL https://doi.org/10.48550/arXiv.2603.00718.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. EnvBench: A benchmark for automated environment setup. *CoRR*, abs/2503.14443, 2025. doi: 10.48550/arXiv.2503.14443. URL https://doi.org/10.48550/arXiv.2503.14443.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory. *CoRR*, abs/2508.06433, 2025. doi: 10.48550/arXiv.2508.06433. URL https://doi.org/10.48550/arXiv.2508.06433.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=fZsd3KLMje.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Sandeep Kumar, Abhijit A Nargund, and Vivek Sridhar. CourtEval: A courtroom-based multi-agent evaluation framework. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 25875–25887, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1327. URL https://aclanthology.org/2025.findings-acl.1327/.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories, 2024. URL https://arxiv.org/abs/2405.19856.

OpenCode Team. OpenCode. https://github.com/anomalyco/opencode, 2025. GitHub repository, accessed 2026-09-07.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems. *CoRR*, abs/2310.08560, 2023. URL https://arxiv.org/abs/2310.08560.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (eds.), *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, pp. 2:1–2:22. ACM, 2023. doi: 10.1145/3586183.3606763. URL https://doi.org/10.1145/3586183.3606763.

Qwen Team. Qwen Code. https://github.com/QwenLM/qwen-code, 2025. GitHub repository, accessed 2026-09-07.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, Wayne Xin Zhao, Yang Song, Tao Zhang, and Ji-Rong Wen. SWE-Master: Unleashing the potential of software engineering agents via post-training. *CoRR*, abs/2602.03411, 2026. URL https://arxiv.org/abs/2602.03411v2.

Xiangru Tang, Tianrui Qin, Tianhao Peng, Ziyang Zhou, Daniel Shao, Tingting Du, Xinming Wei, Peng Xia, Fang Wu, He Zhu, Ge Zhang, Jiaheng Liu, Xingyao Wang, Sirui Hong, Chenglin Wu, Hao Cheng, Chi Wang, and Wangchunshu Zhou. Agent KB: leveraging cross-domain experience

for agentic problem solving. *CoRR*, abs/2507.06229, 2025. doi: 10.48550/arXiv.2507.06229. URL https://doi.org/10.48550/arXiv.2507.06229.

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. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=ehfRiF0R3a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, et al. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. Watchman: monitoring dependency conflicts for Python library ecosystem. In Gregg Rothermel and Doo-Hwan Bae (eds.), *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pp. 125–135. ACM, 2020. doi: 10.1145/3377811.3380426. URL https://doi.org/10.1145/3377811.3380426.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, and Botian Shi. EvolveR: Self-evolving LLM agents through an experience-driven lifecycle. *CoRR*, abs/2510.16079, 2025. doi: 10.48550/arXiv.2510.16079. URL https://doi.org/10.48550/arXiv.2510.16079.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-Copilot: Towards generalist computer agents with self-improvement. *CoRR*, abs/2402.07456, 2024. URL https://arxiv.org/abs/2402.07456v2.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In Amir Globerson, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

John Yang, Carlos E. Jimenez, Alex Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. SWE-bench Multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/07d6332ae36730707fddddba736d7b6c-Abstract-Conference.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Runyang You, Hongru Cai, Caiqi Zhang, Qiancheng Xu, Meng Liu, Tiezheng Yu, Yongqi Li, and Wenjie Li. Agent-as-a-judge, 2026. URL https://arxiv.org/abs/2601.05111.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In Maria Christakis and Michael Pradel (eds.), *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pp. 1592–1604. ACM, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam

Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2024, February 20-27, 2024, Vancouver, Canada*, pp. 19632–19642. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29936. URL https://doi.org/10.1609/aaai.v38i17.29936.

Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. MemoryBank: Enhancing large language models with long-term memory. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19724–19731. AAAI Press, 2024. doi: 10.1609/aaai.v38i17.29946. URL https://ojs.aaai.org/index.php/AAAI/article/view/29946.

A XPU AND RETRIEVAL IMPLEMENTATION

A.1 EXAMPLE XPU

Example XPU Entry

```
{
  "id": "xpu_poetry_lock_conflict",
  "signals": {
    "keywords": ["poetry.lock", "pyproject.toml", "dependency conflict"],
    "regex": ["Because .* depends on .*", "version solving failed"],
    "situation_triggers": [
      "Poetry-managed project where manual pip fallback risks losing the lock graph"
    ]
  },
  "advice_nl": [
    "Do not bypass Poetry with manual pip installation.",
    "Preserve the locked dependency graph and resolve the conflict inside Poetry.",
    "Run Poetry installation end-to-end after checking the lock file."
  ],
  "atoms": [
    {"name": "inspect_file", "args": {"path": "pyproject.toml"}},
    {"name": "inspect_file", "args": {"path": "poetry.lock"}},
    {"name": "shell", "args": {"cmd": "poetry install --no-interaction"}}
  ],
  "telemetry": {
    "hits": 63,
    "successes": 37,
    "failures": 15
  }
}
```

A.2 ANCHOR-BASED RETROSPECTIVE EVALUATION PROTOCOL

During experience generation on the development pool, each Retriever Agent invocation first audits the outcome of the previous retrieval. The database entries and telemetry are frozen during benchmark evaluation; the updates below apply only to experience generation. For each previously recommended ***XPU***, the Retriever Agent assigns a categorical verdict $v \in \{\textit{success}, \textit{failure}, \textit{neutral}\}$ based on the subsequent setup trajectory.

1. **Anchor.** At retrieval time, record the current trajectory length and the identifiers of the recommended ***XPU*** entries, establishing a temporal reference point.
2. **Extract.** On the next retrieval call, extract up to five subsequent steps from the main agent's trajectory after the anchor point.
3. **Judge.** An LLM judge determines whether each recommended ***XPU*** contributed to resolving the observed problem. The verdict is *success* if the advice was adopted and the problem was resolved or improved; *failure* if the advice was adopted but the problem persisted or worsened; and *neutral* if adoption or causal contribution cannot be determined.
4. **Update.** The telemetry counters of each recommended ***XPU*** are updated atomically according to the verdict:

$$(\textit{successes}, \textit{failures}) \mathrel{+}= \begin{cases} (1,0), & \text{if } v = \textit{success}, \\ (0,1), & \text{if } v = \textit{failure}, \\ (0,0), & \text{if } v = \textit{neutral}. \end{cases} \tag{4}$$

B AGENT PROMPT EXCERPTS

This appendix provides selected excerpts from the role-specific prompts used by SETUPX. We include only the parts that define each agent's role, action boundary, and output contract. The complete prompts are provided in the supplementary code artifact.

Setup Agent Prompt Excerpt

```
You are an expert DevOps agent tailored for environment setup.
You have access to a Linux terminal and an external eXPerience Unit (XPU).

## Action Types -- Purpose and When to Use
```

```
### SHELL_COMMAND
Execute any shell command directly in the container.

### TRY_XPU_SUGGESTION
Apply a proven fix from the XPU knowledge base inside a snapshot sandbox.

### SET_ENV
Persist an environment variable across all subsequent commands.

### ROLLBACK_ENV
Pop any number of frames off the snapshot stack and return the container to
any earlier known-good checkpoint, not only the latest one.

### VERIFY
Trigger the full pytest verification pipeline.

### FINISH
Signal that the task is complete. ONLY call after a successful VERIFY.

You MUST respond in JSON format with this schema:
{
  "thought": "Analyze the current state and the cause of the error, then explain why you
  chose this action...",
  "action_type": "SHELL_COMMAND | TRY_XPU_SUGGESTION | SET_ENV | ROLLBACK_ENV | VERIFY |
  FINISH",
  "content": {
    // For SHELL_COMMAND:
    "command": "pip install numpy",

    // For TRY_XPU_SUGGESTION:
    "xpu_suggestion_id": "suggestion_123",
    "command": "pip install numpy==1.23.5",
    "reasoning": "The XPU suggests downgrading numpy; this matches the error closely."

    // For SET_ENV:
    "env_key": "VAR_NAME",
    "env_value": "value"

    // For ROLLBACK_ENV:
    // "n_frames": 1   // default 1; pass >=2 to go back to an earlier checkpoint.
    //
    // For VERIFY / FINISH:
    // (VERIFY needs no extra fields)
    // FINISH requires: "message": "environment setup complete"
  }
}
```

Retriever Agent Prompt Excerpts

```
You are an XPU experience-retrieval assistant. Given a list of candidate XPU experiences,
your task is to pick the Top-K that best match the current deployment situation.

## Selection rules
1. Exact match first: XPUs whose advice_nl directly addresses the current problem rank
highest.
2. Telemetry as reference: pay attention to each XPU's historical hit / success / failure
counts, but do not discard one merely because it has many failures -- judge whether the
previous failure scenarios are similar to the current one. If they are not, the XPU may
still be effective.
3. Drop the irrelevant: if an XPU's advice is completely unrelated to the current problem,
do not pick it.
4. Pick at most {k}.

You are an XPU experience-audit assistant. Your task: judge whether the XPU experiences
recommended to the Agent last time helped the deployment.

## Decision rules (judge each XPU separately)
- success: the Agent adopted the XPU's idea (possibly with a different command but the same
approach), and subsequent steps show the problem was solved or clearly improved.
- failure: the Agent adopted the XPU's idea, but the problem was not solved or new problems
were introduced.
- neutral: cannot tell whether the Agent adopted the suggestion, or the suggestion is
unrelated to the subsequent steps.
```

In-Loop Verifier Agent Prompt Excerpt

```
You are a verification Agent working inside a Docker container.
Your task: examine whether the environment configured by the Setup Agent is acceptable, and
report the result truthfully.

You are not in charge of fixing anything; you only run tests, observe results, and pass
judgment.

## Verification procedure
1. Structure reconnaissance: `ls` the project root, locate pyproject.toml / setup.cfg /
pytest.ini / tox.ini, etc.
2. Locate the test suite: confirm the test directory and framework. (pytest / unittest /
tox, ...).
3. Run the tests in the project's native way and collect results.
4. Analyze failure causes and make a judgment.
5. If the project has no tests at all, write a smoke test under /tmp/ to verify basic
environment usability.

## Hard constraints (violation invalidates the verdict)
- Install no packages.
- Modify no environment configuration.
- Modify no file under /workspace/repo.
- write_file may only write into /tmp/.
```

Prosecutor and Judge Prompt Excerpts

```
You are the prosecutor. Your stance is skeptical: both the Setup Agent and the Verifier may
make mistakes, take shortcuts, or deceive themselves.
You must verify independently using actual evidence inside the container, not trust their
self-reports.

## Mandatory investigation procedure (in order, no skipping)
Step 0: identify the project language and build tool.
Step 1: verify that core dependencies are available.
Step 2: exercise the entry commands declared in README.
Step 3: run the test suite yourself.
Step 4: adjudicate each failure category.
Step 5: cross-check the credibility of the Verifier's conclusion.

You are the Judge. Your job is to verify the prosecutor's charges one by one and then issue
a verdict.

You are not the prosecutor -- you do not conduct open-ended investigation. Your
responsibility: for each charge raised by the prosecutor, execute 1-2 verification commands
to confirm whether that charge holds.

## Final verdict
- >=1 charge upheld after your verification -> guilty
- All charges dismissed -> not_guilty
```

XPU Distiller Prompt Excerpt

```
You are a senior expert in Python project environment configuration and dependency issues.
You will be given a complete agent trajectory (executed commands and error logs) from an
automated environment setup of a repository, and (when available) the Phase 2
Prosecutor-Judge adjudication signal phase2_context for that trajectory.

Mandatory four-step distillation procedure:
[Step 1: Verdict-Aware Ingestion] Eat the verdict first, then read the trajectory.
[Step 2: Forward Attribution] Trace each problem forward to the action that actually fixed
it.
[Step 3: Schema-Level Distillation] Map each problem-fix pair onto the XPU schema.
[Step 4: Coarse-to-Fine Deduplication] Use a vector pre-filter to shortlist existing XPUs
with cosine similarity >= 0.85 to each candidate. Judge whether the shortlisted entries are
semantically equivalent to the candidate. Merge confirmed duplicates by unioning their
signal sets and fusing their advice; otherwise, ingest the candidate as a new experience.

Distillation principles:
- prosecution_charges are the cleanest causal knowledge source -- distill from them first.
- Even when verdict=guilty, distill the generalizable patterns within.
- One XPU = one root cause; never mix several unrelated problems.
- Do not produce an id field; the system assigns a unique ID automatically.
```

C ACTION SPACE

The agent executes the selected action from the action space $\mathcal{A}_{\text{act}} = \{a_1, \ldots, a_6\}$:

- `SHELL_COMMAND`: Execute an arbitrary shell command—the primary action for diagnosis and repair.
- `TRY_XPU_SUGGESTION`: Speculatively apply a retrieved ***XPU*** entry under the snapshot protection described in Section 4.2.
- `SET_ENV`: Persistently set an environment variable in the container.
- `ROLLBACK_ENV`: Roll back the container to the most recent snapshot on the LIFO stack, or to a specific earlier checkpoint.
- `VERIFY`: Invoke an in-loop Verifier Agent (Section 4.3) to assess environment readiness.
- `FINISH`: Declare setup complete and proceed to Phase 2 Prosecutor–Judge verification.

C.1 `TRY_XPU_SUGGESTION`

The speculative execution protocol proceeds as follows:

1. **Checkpoint**: Snapshot the current container via `docker commit`, pushing the image onto the LIFO stack $\mathcal{S}$.
2. **Adapt**: The agent reads the ***XPU***'s *advice_nl*, leverages its full conversation context (recent history, observed versions, repository structure), and generates concrete commands tailored to the current repository. If the agent produces no command, the system falls back to rendering the ***XPU***'s *atoms* via a type-aware *Atom Rendering Engine* that maps 12 predefined atom types (e.g., `pip_install`, `apt_install`) to executable bash commands.
3. **Trial**: Execute the adapted commands sequentially; halt on any non-zero exit code.
4. **Verify**: Compare the error state before and after execution to determine the trial outcome.
5. **Decision**: Retain changes on success trial; roll back to the snapshot otherwise.

D Multi-repository Track Evaluation Rubric

D.1 Overview

For the 22 non-atomic families, a verdict is assigned by direct in-container inspection rather than by the Prosecutor–Judge protocol used on the single-repository benchmark, because family-level usability spans multiple containers, services, and configuration consistency that no single adjudication protocol covers.

Each family is independently rated by three evaluators using the rubric below. Disagreements are resolved through discussion until consensus is reached. Evaluators apply the same family-specific tests and rubric to both systems' final containers. Verdicts are based on these final-container checks, rather than on whether the setup agent itself invoked a particular test. Setup trajectories provide contextual evidence, not a substitute for the common evaluation procedure.

D.2 Five-Check Rubric

For each family's host container, the following five checks are performed:

C1. Clone integrity. The host repository is fully cloned and contains all expected source files. (Verified via `ls /workspace/repo` and `git status`.)

C2. Editable install. The host package is installed in editable mode and resolves correctly. (Verified via `pip list -e`.)

C3. Module / CLI availability. The primary module of the host package is importable, and the CLI entry point (if any) responds to `-version` or `-help`. (Verified via `python -c "import <module>"` and `<cli> -version`.) Required commands are also checked in the delivered execution environment; missing `PATH` configuration is recorded as an entry-point configuration gap, separately from package installation or module import failure.

C4. Pytest collection. The host repository's test suite can be collected by `pytest -co` without fatal errors. The number of collected tests is recorded.

C5. Dependency consistency. `pip check` reports no broken or inconsistent dependencies in the venv.

For families involving cross-container protocol layers (sibling services, Kubernetes deployments, plugin entry-point registration), additional protocol checks are performed inside the relevant container, e.g., service connectivity via socket connection, `kubectl get pods` for Kubernetes families, or `importlib.metadata.entry_points()` for plugin-host families.

D.3 FOUR-TIER VERDICT

Verdicts are assigned based on the worst-evidence + overall main-path usability principle:

Verdict	Criteria
Full	All five checks pass; protocol layer (if any) is fully realized: sibling services running, Kubernetes pods Up, plugin entry points registered.
Mostly	Host setup is complete and the main path is functional, but with one or two bounded minor gaps (e.g., a few collection errors, one sibling missing, an unfinished helm step).
Partial	Host installation is partial; some core dependencies missing or main module imports with errors. The intended workflow does not run end-to-end.
Shallow	Host repository is cloned but core dependencies are not installed; `import` fails immediately, or `git clone` itself failed.

The **Full–Mostly boundary** is determined by whether one independently fixable minor gap exists. The **Shallow** tier qualitatively differs from the other three: it represents "did not actually install" rather than "installed with limitations."

Pytest collection serves as a key signal: successful collection of the majority of tests indicates that host setup is substantively complete.

E PER-FAMILY VERDICTS OF MULTI-REPOSITORY

This appendix reports per-family verdicts for the non-atomic ecosystems used in our multi-repository experiment. For each family we run two independent setup pipelines—SETUPX and the QWEN CODE baseline—and rate the resulting container against the five checks of Appendix D (clone integrity, editable install, module/CLI availability, `pytest -co`, and `pip check`) plus protocol-layer checks where relevant (sibling-service ports, Kubernetes pods, plugin entry-point registration). Verdicts use the four tiers defined there: **Full**, **Mostly**, **Partial**, **Shallow**. The *primary repo* column gives the entry repository in the family's selection record; the *Pattern* column summarises the cross-repository dependency type; the *Notes* column gives a one-line contrast of the two pipelines' behaviour on that family.

Table 4: Per-family verdicts on the 22 non-atomic ecosystems. SETUPX reaches Full on 6/22 and Mostly on 10/22; QWEN CODE reaches Full on 1/22 and Mostly on 15/22. Both systems clear the ≥Mostly bar on 16/22, but the internal composition of the verdicts differs (see Appendix F for a worked example).

#	Family (primary repo)	Pattern	SETUPX	QWEN
1	`terminusdb-client-python`	client/server	Full	Mostly
Notes: Both complete core installation and establish server connectivity; QWEN leaves the default `PATH` unconfigured for its venv, whereas SETUPX provides directly available execution commands.				
2	`ansible/awx-operator`	K8s deployment	Mostly	Mostly
Notes: Both bring all AWX pods (web/task/postgres) Up on a KIND cluster; QWEN ships no editable Python install for any package.				
3	`HumanSignal/label-studio`	full platform	Shallow	Partial
Notes: SETUPX hits a 5 min `git clone` timeout on the 1 GB monorepo, leaving an empty checkout; QWEN clones 81 packages but the host package is not editable and the CLI is absent.				
4	`scrapy/scrapy`	deploy + plugin host	Mostly	Mostly

continued on next page

Table 4 – continued

#	Family (primary repo)	Pattern	SETUPX	QWEN
	Notes: SETUPX covers the host only (4 siblings missing); QWEN installs `scrapyd`/`splash`/`redis`/`scrapy-redis`, but `pytest` is absent and the Scrapyd process is a zombie.			
5	`saleor/saleor`	Django commerce	Partial	Partial
	Notes: Both collect 17,228 tests with no import errors; in both, the `saleor` host package itself is not truly editable and `import saleor` fails.			
6	`odoo/odoo`	framework + OCA addons	Mostly	Mostly
	Notes: SETUPX runs 1,245 base tests but ships no OCA siblings; QWEN clones the three OCA repos and keeps `odoo-bin` alive on `:8069`, but no editable install.			
7	`intake/intake`	plugin host	Mostly	Mostly
	Notes: SETUPX ships core editable + 9 driver registrations; QWEN adds `intake-xarray`/`sql`/`geopandas` as editable siblings (794 tests collectable across packages).			
8	`robotframework/robotframework`	plugin host	Mostly	Full
	Notes: QWEN adds `SeleniumLibrary`/`RequestsLibrary`/`Browser` siblings + 2,318 utests OK; SETUPX keeps the host alone, none of the three siblings installed.			
9	`napari/napari`	Qt plugin host	Full	Mostly
	Notes: Both instantiate a headless `Viewer` under `QT_QPA_PLATFORM=offscreen`; QWEN reports one dev-version warning and 3 collection errors.			
10	`apache/airflow`	client/server + deploy	Mostly	Shallow
	Notes: SETUPX ships `core`+`client`+`cncf-kubernetes provider` all editable, both Postgres/Redis sibling ports up; QWEN has only the client editable, `import airflow` fails.			
11	`matrix-org/synapse`	multi-service homeserver	Mostly	Mostly
	Notes: SETUPX switches to a Poetry venv and rebuilds the Rust extension on aarch64; QWEN runs the homeserver process plus `matrix-client`/`sydent` sibling editables.			
12	`ckan/ckan`	runtime + plugin host	Full	Mostly
	Notes: SETUPX collects 3,358 tests and runs alembic Upgrade DB to completion; QWEN keeps CKAN listening on `:5000` but `pytest` is missing and Solr `:8983` is unreachable.			
13	`flyteorg/flyte`	Go backend + Py SDK	Partial	Mostly
	Notes: SETUPX picks the wrong language path (Python image vs. Go monorepo) and crashes on embedded-postgres root pollution; QWEN ships Flyte 2 SDK + `flyteidl2` as editable.			
14	`MISP/MISP`	PHP server + Py SDK	Partial	Partial
	Notes: SETUPX skips the PHP server entirely (Python base image), only `PyMISP` is in place; QWEN's rerun left `misp-modules` cloned only.			
15	`bitcart/bitcart`	multi-repo workflow	Partial	Mostly
	Notes: SETUPX hits the Python 3.10 vs. `>=3.12` floor; `:5432`/`:6379` both refuse. QWEN uses `uv` + Python 3.14.4 and brings 4 sibling containers + Backend `:8000` up.			
16	`bottlecapdave/homeassistant-octopusenergy`	HA custom integration	Mostly	Partial
	Notes: SETUPX `apt`-installs `python3.13-venv` matching the manifest pin (759 tests, 0 error); QWEN's venv pins HA `2024.3.3` against a `2025.11+` requirement, yielding 96 collection errors.			
17	`frappe/erpnext`	framework + app	Partial	Shallow
	Notes: SETUPX pip-installs `frappe` + editable `erpnext` but ships no `bench`/MariaDB; QWEN's Python 3.11 is rejected by source-level PEP 695 syntax (`type ConfType = ...`).			
18	`internetarchive/brozzler`	archiving workflow	Full	Mostly
	Notes: Both install `brozzler`+`warcprox` as editables and bring up a RethinkDB sibling on `:28015`; SETUPX additionally installs Chromium 147 with the full sandbox stack.			
19	`jazzband/wagtailmenus`	Wagtail plugin	Mostly	Mostly
	Notes: Both run `runtests.py` cleanly with all 175 tests OK; QWEN additionally installs `wagtail-localize`, `-localize-git`, and `-grapple` as editable siblings.			
20	`jupyterhub/jupyterhub`	Hub plugin host	Full	Mostly
	Notes: SETUPX ships `oauthenticator` editable with 11 OAuth-provider entry points registered and a full `pytest` pass; QWEN keeps Hub running and adds `kubespawner`/`dockerspawner`/`batchspawner`.			
21	`mov-cli/mov-cli`	CLI plugin host	Mostly	Mostly
	Notes: Both load three plugins via the `mov-cli-*` module-naming convention; SETUPX additionally executes a live YouTube search returning 13 hits dated 2026-05.			
22	`qiboteam/qibolab`	quantum backend driver	Full	Mostly
	Notes: SETUPX instantiates `create_platform("dummy")` into a 5-qubit `Platform` with the qibolab backend entry point registered; QWEN ships no `pytest`, no CLI, with the host repo at `/workspace` not the expected `/workspace/repo`.			

Aggregate. SETUPX's distribution is Full $= 6$, Mostly $= 10$, Partial $= 5$, Shallow $= 1$, with $\geq$Mostly on $16/22$. QWEN CODE's distribution is Full $= 1$, Mostly $= 15$, Partial $= 4$, Shallow $= 2$, also with $\geq$Mostly on $16/22$. The two systems tie on the $\geq$Mostly bar but their verdict structure differs: SETUPX's six *Full* verdicts are concentrated on families where the protocol layer is end-to-end observable (real sibling containers, real protocol handshakes, real integration tests), while QWEN CODE's fifteen *Mostly* verdicts reflect remaining gaps in setup completeness or protocol realization

(Appendix F develops the `terminusdb-client-python` family in detail as a worked example of this asymmetry).

Failure modes. SETUPX's six Partial/Shallow cases concentrate on (i) the 5-minute `git clone` ceiling failing on ∼1 GB monorepos (`label-studio`); (ii) cross-language families colliding with the default Python base image (`flyte` vs. Go, `MISP` vs. PHP, `frappe` via the wrong strategy); (iii) Python-version floors above the base image (`bitcart` 3.10 vs. 3.12+ requirement); and (iv) incomplete host installation and failed main-module import (`saleor`). QWEN CODE's six Partial/Shallow cases come predominantly from (i) missing editable install of the host package leading to `import` failure (`airflow`, `frappe`, `saleor`); (ii) mis-pinned host versions (`homeassistant-octopusenergy` pinned to HA 2024.3 against a 2025.11+ requirement); and (iii) incomplete component installation (`MISP`: the rerun left `misp-modules` cloned only). Beyond the shared host-installation failure in `Saleor`, the cases illustrate bottlenecks in *protocol adaptation* (SETUPX-side) and *dependency-assembly semantics* (QWEN-side) at different layers.

F REPRESENTATIVE MULTI-REPOSITORY CASES

We zoom into two of the 22 family verdicts in Appendix E as worked examples. The two cases are chosen to cover contrasting dependency structures and setup outcomes: **terminusdb-client-python** is the only pure client/server family in the 22 (its cross-repository dependency is a single HTTP protocol against a sibling server, with no plugin-host or multi-service confounders), and **frappe/erpnext** is one of the families on which SETUPX+XPU itself underperforms (Partial), chosen so that the case pairing covers both ends of the verdict scale rather than only families where SetupX wins.

TERMINUSDB-CLIENT-PYTHON (CLIENT/SERVER, SETUPX+XPU=FULL / QWEN CODE=MOSTLY)

Run identifiers. SETUPX+XPU on container `bfd7bd0e6b8f`; QWEN CODE on container `4462896bd3c7`.

terminusdb-client-python is the Python SDK for TerminusDB. Its integration tests (e.g., `Client.connect`, `db_create` / `db_delete`) all require sending HTTP requests to a running TerminusDB server and parsing the response. A standalone `pip install -e .` only covers unit tests over the client's internal data structures (schema / document / WOQL AST construction); the full integration suite requires a TerminusDB server running in parallel.

SetupX+XPU trajectory (17 steps). The agent inspects `pyproject.toml` and `conftest.py`, identifies that `integration_tests/` target `localhost:6363`, then runs `apt install docker.io` followed by `docker pull/run terminusdb-server` to bring up the sibling. `curl /api/info` confirms HTTP 200. After installing the client and test dependencies, `pytest integration_tests/test_client.py` reports 122 passed / 5 skipped, and the agent halts on its own assessment that setup is complete.

Qwen Code trajectory (setup completed). Qwen Code likewise recognizes the client/server protocol, brings up `terminusdb-server` on port 6363, and performs an editable install of the client SDK in a venv. `pytest -co` collects 1499 tests and `pip check` reports clean. However, the setup terminates at the "tests can be collected" stage without invoking the integration suite. Separately, the container's default `PATH` is not linked to `/workspace/.venv/bin/`, so bare `python` / `pytest` commands return `executable not found` and the full venv path must be used explicitly.

Recorded observations. The table summarizes container checks and, separately, integration-test execution during setup. The final-container entry-point configuration distinguishes the two ratings under Appendix D.

Check	SetupX+XPU	Qwen Code
Clone integrity	PASS	PASS
Editable install (host package in `pip list -e`)	PASS	PASS
`pip check` clean	PASS	PASS
Protocol: sibling server up + HTTP handshake	PASS	PASS
Protocol: integration tests invoked during setup	122 passed	**not invoked**
Bare commands on default `PATH`	available	require full venv path

Verdict justification. Both systems complete core installation and establish server connectivity. Qwen Code, however, leaves the default `PATH` unconfigured for its virtual environment, so `python` and `pytest` require explicit venv paths. This bounded, independently fixable entry-point configuration gap yields **Mostly**, rather than **Full**. SETUPX+XPU satisfies the corresponding checks without additional environment adjustment and receives **Full**.

FRAPPE/ERPNEXT (MULTI-SERVICE DEPLOYMENT, SETUPX+XPU=PARTIAL / QWEN CODE=SHALLOW)

Run identifiers. SETUPX+XPU on container `3bc8bfb590f8`; QWEN CODE on container `7b071e5b5237`.

frappe/erpnext is an ERP business application. Its standard deployment relies on the `bench` toolchain (`bench init` / `bench install-app`) to host multiple sibling apps, with parallel MariaDB (port 3306) and Redis siblings providing persistence and cache. The latest source's `pyproject.toml` requires Python `>=3.14,<3.15` (the source already uses PEP 695 `type` aliases such as `type ConfType = ...`); the container's default base image ships Python 3.11.

SetupX+XPU trajectory (53 steps). The agent inspects the source and identifies the Python version constraint, then takes a *sidestep* strategy rather than satisfying the constraint directly: it runs `apt install python3.13` and creates a Python 3.13 venv at `/workspace/venv`, then installs the released v15 tag via `pip install git+https://github.com/frappe/frappe@version-15` (non-editable, frappe 15.107.0; the v15 release has a lower Python floor than the latest dev source) and installs erpnext editable from `/workspace/repo` (15.106.0). Both `import frappe` and `import erpnext` succeed. `pytest -co` lists 260 tests but reports 171 collection errors (missing frappe site context). After creating a placeholder `sites/test_site/site_config.json`, the agent attempts `apt install docker.io` to start a MariaDB sibling; the MariaDB sibling does not come up, the bench CLI is never installed, and the bench toolchain pipeline is not invoked.

Qwen Code trajectory. Qwen Code uses the container's default Python 3.11.2 without upgrading. `pip install -e /workspace` immediately raises a `SyntaxError` on the source's PEP 695 `type` alias, and the host package fails to install. `import frappe` then raises `ModuleNotFoundError`. The bench environment contains only `pip` and `setuptools`; `bench -version` prints a version, but `bench -help` / `bench list-apps` are blocked by environment protections. MariaDB on port 3306 refuses connection (Redis on 6379 is reachable). Fourteen sibling app repositories have been git-cloned, but no `bench install-app` step is performed.

Verdict justification.

Check	SetupX+XPU	Qwen Code
Clone integrity	PASS	PASS
Host install (Python compatibility)	PASS (Python 3.13 venv + v15 tag)	**FAIL** (Python 3.11 rejects PEP 695 syntax)
Module import (`import frappe / erpnext`)	PASS	**FAIL** (`ModuleNotFoundError`)
Protocol: bench CLI available	not installed	installed but blocked
Protocol: MariaDB sibling running	not started	not started
Protocol: bench install-app triggered	not triggered	not triggered

Neither run completes the full bench toolchain pipeline (the three protocol-layer rows are tied at zero). The verdict difference is forced by the host-install row, which Appendix D defines as the boundary between Partial and Shallow: **Shallow** applies when "host repository is cloned but core dependencies are not installed; `import` fails immediately," and **Partial** applies when "host installation is partial ... intended workflow does not run end-to-end." SETUPX+XPU recognizes the source's Python version constraint early in setup and sidesteps it by pinning to the v15 release tag on a Python 3.13 venv; the host package and erpnext install successfully, `import` succeeds, and partial pytest collection works, but the bench/MariaDB workflow does not run end-to-end — this matches Partial. QWEN CODE uses the default Python and fails at the syntax level during host install, leaving the host module unimportable; sibling app sources are cloned but never integrated — this matches Shallow. The core behavioral difference between the two paths lies at the *runtime-constraint recognition step* early in setup: whether the agent verifies the source's required Python version before attempting to install, and whether it has a recovery strategy (here, pinning to an older release tag) when the constraint is not satisfied by the base image.

G STABILITY, BACKBONE, AND EFFICIENCY ANALYSIS

G.1 RUN-TO-RUN STABILITY AND BACKBONES

We repeat the four principal systems with two additional random seeds. Table 6 reports the three runs, their mean and standard deviation, and the 95% Wilson interval over repository-level pass outcomes. We additionally run the full pipeline with two stronger backbones; the result varies by at most 1 pp across the three backbones.

Table 6: Run-to-run stability on the 100-repository benchmark.

System	Three runs (%)	Mean $\pm$ std	95% Wilson CI
SETUPX	92 / 91 / 92	91.7 ± 0.6	[84.6, 95.7]
Claude Code	73 / 71 / 74	72.7 ± 1.5	[63.2, 80.4]
OpenCode	72 / 73 / 72	72.3 ± 0.6	[62.9, 80.2]
Qwen Code	70 / 71 / 69	70.0 ± 1.0	[60.4, 78.1]

The intervals are separated: SETUPX's lower bound (84.6%) exceeds the upper bound of every CLI baseline (at most 80.4%), so the observed advantage is not explained by run-to-run variation.

Table 7: Full SETUPX under different backbone models.

Backbone	Pass
`qwen3.5-plus`	92
Claude Opus 5.5	92
GPT-5.6-sol	93

G.2 DEPLOYMENT COST

Table 8 measures only the setup main loop that a user runs. For SETUPX, this includes the independent-context Retriever Agent, top-K reranking, and In-Loop Verifier. The one-shot P–J evaluator is excluded for every system because it is an evaluation protocol rather than part of deployment.

Table 8: Mean deployment cost per repository with a shared `qwen3.5-plus` backbone.

System	Input tokens	Output tokens	API cost	Wall-clock
SETUPX	0.47M	0.055M	¥0.64	18.3 min
Claude Code	2.37M	0.008M	¥1.93	19.2 min
OpenCode	1.77M	0.006M	¥1.45	24.7 min
Qwen Code	2.06M	0.007M	¥1.68	18.6 min

The Retriever Agent absorbs raw-error context and returns a small structured experience to the main setup agent. Although retrieval and reranking introduce additional short calls, they avoid repeatedly injecting a growing monolithic trajectory into the main context; this primarily reduces input tokens, which dominate the reported API cost.

H XPU QUALITY AND REPRESENTATION EVALUATION

H.1 DISTILLATION AND RUN-LEVEL UTILITY

From deployment trajectories on the 229 development repositories, the four-step distillation pipeline extracts 705 candidate XPUs and merges 105 near-duplicates, yielding 600 entries. Re-embedding the final store leaves only 54 of 179,700 entry pairs (0.03%) above the 0.85 similarity threshold. Two software-engineering experts reviewed disjoint halves of the store (300 XPUs each): 98.7% were judged technically accurate, corresponding to a 1.3% unsound-advice rate, and 99.3% were judged generalizable beyond the source repository.

At deployment time, 84 of 100 repositories adopt at least one retrieved XPU, and 81 of these 84 reach a successful setup (96.4%); the remaining 3.6% bounds adoption-conditioned non-resolution. Of the 16 runs without adoption, 11 complete without triggering retrieval and five encounter niche errors for which the store contains no matching entry; all five fail. Because a run often adopts several XPUs, we report adoption-conditioned final resolution rather than attributing success causally to a single suggestion. Failed trials can be contained by speculative rollback; telemetry is updated during experience generation but remains frozen during benchmark evaluation.

H.2 END-TO-END REPRESENTATION COMPARISON

We compare two experience representations within the same SETUPX pipeline: the ***XPU Database*** and a repository-domain Voyager-style code-skill library (Wang et al., 2024). Both libraries contain 600 entries built from the same development trajectories drawn from the 229-repository development pool, disjoint from the 100 benchmark repositories. Neither library contains added synthetic noise. This experiment replaces the experience representation, rather than evaluating the original Voyager agent.

Both configurations use the same `text-embedding-3-small` embedding model, top-10 candidate retrieval, and Selector-based reranking to select three entries. The `qwen3.5-plus` backbone, speculative execution, In-Loop Verifier, and setup and evaluation settings remain fixed. Each configuration is evaluated on the same 100 repositories at the same pinned revisions, using fresh `ubuntu:22.04` containers, a 3600-second timeout, and the unified Prosecutor–Judge protocol described in Section 6.2.

The Voyager-style replacement passes 86 of 100 repositories, compared with 92 of 100 for the ***XPU*** configuration, an observed difference of 6 percentage points. These results correspond to the *Selector + Code skill* and *Selector + Clean* rows in Figure 6b, respectively, and measure end-to-end setup success rather than retrieval relevance.

I VERIFICATION VALIDATION

Two domain experts independently inspect 700 configured environments: 100 final containers from each of SETUPX+***XPU***, ExecutionAgent, Repo2Run, EnvBench, Claude Code, OpenCode, and Qwen

Code. The same review protocol applies to all seven systems: containers are presented in randomized order with system identities masked, and neither expert sees the other's annotation during the initial review. Each expert first assigns a pass/fail label to every container; disagreements are then resolved through discussion to produce the adjudicated expert labels. Table 9 reports confusion matrices for the 400 outputs from SETUPX+***XPU*** and the three specialized tools. Table 11 uses the same adjudicated labels for SETUPX+***XPU***, Repo2Run, and the three CLI agents; outputs appearing in multiple tables are the same final containers, not additional runs.

Table 9: P–J verdicts against adjudicated expert labels. Each system contributes 100 final containers.

System	P–J verdict	Expert pass	Expert fail	Total
SETUPX+***XPU***	Pass	90	2	92
	Fail	0	8	8
	Total	90	10	100
ExecutionAgent	Pass	58	1	59
	Fail	2	39	41
	Total	60	40	100
Repo2Run	Pass	42	3	45
	Fail	2	53	55
	Total	44	56	100
EnvBench	Pass	36	2	38
	Fail	1	61	62
	Total	37	63	100

P–J agrees with the experts on 98/100 SETUPX outputs, 97/100 ExecutionAgent outputs, 95/100 Repo2Run outputs, and 97/100 EnvBench outputs; the corresponding Cohen's κ values are 0.878, 0.938, 0.899, and 0.936. Its estimated pass count is also well calibrated, differing from the expert count by only two repositories for SETUPX and by one for each specialized tool. For SETUPX, P–J retains all 90 expert-confirmed successes and adds two false-positive passes. For the specialized tools, both error directions remain small: P–J produces only 1–3 false-positive passes and 1–2 false-negative failures per system.

We next examine the native completion signals exposed by the three specialized tools: `runner_status=SUCCESS` for ExecutionAgent, `r2r_ok=True` after deployment for Repo2Run, and `install_ok=True` based on `pyright` for EnvBench. Table 10 compares each signal with the expert labels on the same final containers.

Table 10: Native self-claim verdicts against adjudicated expert labels. Each system contributes 100 final containers.

System	Native verdict	Expert pass	Expert fail	Total
ExecutionAgent	Pass	60	22	82
	Fail	0	18	18
	Total	60	40	100
Repo2Run	Pass	44	48	92
	Fail	0	8	8
	Total	44	56	100
EnvBench	Pass	37	53	90
	Fail	0	10	10
	Total	37	63	100

The native signals are uniformly optimistic on their respective 100-output evaluations. All three retain every expert-confirmed success, but they also accept 22 expert-confirmed failures for ExecutionAgent,

48 for Repo2Run, and 53 for EnvBench. Their pass recall is therefore 100%, while failure specificity falls to 45.0%, 14.3%, and 15.9%, respectively; overall accuracy correspondingly declines from 78% to 52% and 47%. P–J is substantially more balanced on the same per-system outputs, attaining 94.6–97.5% failure specificity and 95–97% overall accuracy. Thus, the native criteria are useful indicators that a pipeline reached its nominal endpoint, but their absence of false-negative failures is achieved by accepting many unusable environments.

We also compare P–J with `pytest --collect-only`, the deterministic proxy used by Repo2Run. Collection checks whether test modules import sufficiently to enumerate tests, but neither executes those tests nor launches README-declared entry points. Table 11 shows that collection therefore over-reports success by 5–48 pp relative to experts, while P–J differs by 1–4 pp.

Table 11: Verification outcomes on identical final containers. Inflation is measured relative to expert labels.

System	Collect-only	Expert	P–J	Collect inflation	P–J inflation
SETUPX	95	90	92	+5	+2
Repo2Run	92	44	45	+48	+1
Qwen Code	90	66	70	+24	+4
OpenCode	89	71	72	+18	+1
Claude Code	88	71	73	+17	+2

J SYNTHETIC NOISE COMPOSITION

The clean ***XPU*** knowledge base contains 600 advice entries distilled exclusively from the 229-repository development pool, which is disjoint from the 100 repositories in our benchmark. To construct the noisy variant used in Section 6.4, we add 1,800 synthetic perturbation entries on top of the 600 real ones, yielding a noisy KB of 2,400 total entries (a 3:1 noise-to-real ratio). All noise entries share embedding neighborhoods with real entries, making them difficult to filter via vector similarity alone, and carry zero telemetry counts.

The 1,800 noise entries fall into four classes:

Context perturbation (600 entries, 33.3%). Each real entry is duplicated with its `context.python` randomly rewritten to a Python version in 3.8–3.13, its `os` field expanded, and its `tools` field augmented with one of `conda`, `poetry`, `pdm`, `hatch`, `uv`, or `mamba`. The same advice is replicated under multiple plausible context configurations.

Cross-grafting (450 entries, 25%). Three real entries A, B, C are sampled at random; the synthetic entry combines A's `context`, B's `signals`, C's `advice`, and a random subset of `atoms` from one of the three. This produces entries that are internally consistent in form but inconsistent in provenance.

Generalization blur (450 entries, 25%). Concrete advice is replaced with one of 12 generic templates such as "check the Python version" or "install project dependencies"; `signals.keywords` is replaced with one of 6 generic keyword groups; and `atoms` is cleared. This emulates advice extracted at excessive abstraction.

Action perturbation (300 entries, 16.7%). The problem signals and natural-language advice are retained, while the executable actions in `atoms` are perturbed. This produces entries that remain relevant to the apparent problem but contain incorrect or incomplete operational guidance.

K PER-REPOSITORY MATRIX

This section reports all 100 benchmark repositories and their system-level outcomes. Repository names are shown in compact form for readability.

Table 12: Per-repository pass/fail outcomes for the eight systems on the 100-repository benchmark (compact repository names are used here; canonical identifiers and pinned commit SHAs are provided in the supplementary manifest). A check mark indicates that the system passes the Prosecutor–Judge protocol; a cross (×) indicates failure, including timeout, runtime errors that abort the run, or a guilty verdict. CC, OC, QC, EA, R2R, and EB denote Claude Code, OpenCode, Qwen Code, ExecutionAgent, Repo2Run, and EnvBench, respectively.

Repository	SetupX *+XPU*	SetupX *–XPU*	CC	OC	QC	EA	R2R	EB
cookiecutter	✓	✓	✓	✓	✓	✓	✓	✓
datasets	✓	✓	×	✓	✓	×	×	✓
kedro	✓	✓	✓	✓	✓	✓	✓	✓
luigi	✓	✓	×	✓	✓	×	×	✓
mopidy	✓	×	×	×	✓	×	×	×
pip	✓	✓	✓	✓	✓	✓	×	✓
platformio-core	✓	✓	×	×	✓	×	×	×
supervision	✓	✓	✓	✓	✓	✓	×	×
guardrails	✓	✓	✓	✓	✓	×	×	×
lark	✓	✓	✓	✓	✓	✓	✓	✓
nonebot2	✓	×	✓	×	✓	✓	×	×
pip-tools	✓	✓	✓	✓	✓	✓	✓	✓
tortoise-orm	✓	✓	✓	✓	✓	✓	×	×
trax	×	✓	✓	✓	✓	×	×	×
umap	✓	✓	✓	✓	✓	✓	✓	✓
acme	✓	×	✓	✓	✓	✓	×	×
connexion	✓	✓	✓	✓	✓	✓	✓	×
giskard	✓	✓	✓	✓	✓	✓	×	×
neuralforecast	×	✓	✓	×	✓	×	×	×
plotnine	✓	✓	✓	✓	✓	✓	✓	×
tablib	✓	✓	✓	✓	✓	✓	✓	✓
tmuxp	✓	✓	✓	✓	✓	✓	×	×
great-tables	✓	✓	✓	✓	✓	✓	×	×
mteb	✓	✓	×	×	✓	×	✓	×
plugin.video.netflix	✓	✓	✓	✓	✓	✓	×	✓
py-evm	✓	×	✓	✓	✓	✓	✓	×
python-control	✓	✓	✓	✓	✓	✓	✓	×
smart_open	✓	✓	✓	✓	✓	×	✓	✓
androidviewclient	✓	✓	✓	✓	✓	×	✓	✓
dj-stripe	✓	✓	✓	✓	✓	✓	×	×
fastapi-pagination	✓	✓	✓	✓	✓	×	×	×
hacs_waste_collection_schedule	✓	×	✓	✓	✓	✓	×	✓
memegen	✓	✓	✓	✓	✓	×	×	✓
piccolo	✓	✓	✓	✓	✓	✓	✓	✓
pyftpdlib	✓	✓	✓	✓	✓	✓	✓	×
python-holidays	✓	✓	×	✓	×	×	✓	×
twine	✓	✓	✓	✓	✓	✓	✓	✓
automlbenchmark	✓	✓	✓	✓	✓	×	✓	✓
hydra	✓	✓	✓	✓	✓	✓	×	✓
scvelo	✓	✓	✓	✓	✓	✓	✓	✓
pypose	✓	✓	✓	✓	✓	✓	✓	×
evox	✓	✓	×	×	✓	×	×	×
neurogym	✓	✓	✓	✓	✓	✓	×	✓
cellrank	✓	✓	✓	×	✓	✓	×	×
nevergrad	✓	✓	✓	✓	✓	✓	×	✓
datamol	✓	✓	✓	✓	✓	×	✓	✓
meerkat	✓	✓	×	✓	✓	×	✓	×
ajenti	×	✓	×	✓	×	×	×	×
cheroot	✓	✓	✓	✓	✓	✓	✓	×
fabric	✓	✓	✓	✓	✓	✓	✓	✓
pretalx	✓	✓	✓	✓	✓	✓	×	×
adaptix	✓	✓	✓	✓	✓	✓	✓	✓
wagtailmenus	✓	✓	✓	✓	✓	✓	✓	✓
django-lfs	×	×	×	×	×	✓	×	×
netbox	✓	✓	×	×	✓	×	×	✓
armi	✓	✓	✓	✓	✓	×	×	✓
flavio	✓	✓	✓	✓	✓	✓	✓	✓
monty	✓	✓	✓	✓	✓	×	×	✓
unyt	✓	✓	✓	✓	✓	✓	✓	×
kernel_tuner	✓	✓	✓	✓	✓	×	✓	×
openqasm	✓	×	✓	✓	✓	✓	×	✓
torchgeo	✓	✓	✓	×	✓	×	×	×
importlib_metadata	✓	✓	✓	×	✓	✓	✓	✓
pytest-xdist	✓	✓	×	✓	✓	✓	✓	✓
robotframework	✓	✓	✓	✓	✓	×	✓	×
yapf	✓	×	×	✓	✓	✓	✓	×
keyring	✓	✓	✓	✓	✓	✓	✓	✓
columnflow	✓	×	×	✓	×	✓	✓	✓
civet	×	✓	✓	×	✓	✓	✓	×

continued on next page

Table 12 – continued

Repository	SetupX +XPU	SetupX –XPU	CC	OC	QC	EA	R2R	EB
xknx	✓	✓	✓	✓	✓	✓	✓	×
aiogram_dialog	✓	✓	✓	✓	✓	✓	✓	✓
recipe-scrapers	✓	✓	✓	✓	✓	✓	✓	×
sphinx-gallery	✓	✓	✓	✓	✓	✓	✓	✓
fhempy	✓	×	×	×	×	×	×	×
kh2randomizer	✓	✓	×	×	✓	×	✓	×
modalities	✓	✓	×	×	×	×	×	×
molecule	✓	✓	×	✓	×	✓	✓	×
staged-recipes	×	×	×	×	×	×	×	×
powderday	✓	✓	×	×	×	×	×	×
fixator10-cogs	✓	×	✓	×	×	×	×	×
flask-security	✓	✓	×	×	×	×	×	×
miv-os	✓	×	×	✓	×	×	×	×
django-registration	✓	✓	×	×	×	✓	×	✓
iree-llvm-sandbox	×	×	✓	×	×	×	×	×
lobsterpy	✓	✓	✓	×	×	×	×	×
python-libjuju	✓	✓	✓	✓	×	✓	✓	✓
langgraph	✓	✓	✓	✓	×	✓	×	×
heltour	✓	✓	✓	✓	×	×	×	×
llm-random	✓	✓	✓	×	×	×	×	×
custodian	✓	✓	✓	✓	×	✓	✓	×
naomi	✓	✓	✓	×	×	✓	×	×
river	✓	✓	✓	✓	×	✓	×	×
oadoi	✓	×	×	×	×	✓	×	×
qmlcore	✓	✓	✓	✓	×	✓	×	×
poetry	✓	✓	×	✓	×	✓	✓	×
qiita	×	×	✓	×	×	×	×	×
sublimelinter	✓	×	×	×	×	×	×	×
pyuploadcare	✓	✓	✓	✓	×	×	×	×
django-autocomplete-light	✓	×	✓	✓	×	×	×	×
yubikey-manager	✓	✓	×	×	×	×	×	×

L LIMITATIONS

Scope and generalization. The main benchmark contains 100 Python repositories. The 22-family multi-repository track is a complementary stress test, not evidence of uniform performance across languages, platforms, or deployment ecosystems. Its manual ratings remain dependent on the stated rubric and evaluator judgment.

Verification coverage. Our definition specifies the target property of a functionality-correct environment; P–J operationalizes its assessment through executed tests and representative documented entry points. A pass indicates that no setup-induced failure was detected within this evaluation scope, rather than exhaustive verification of every declared feature. To assess alignment with the definition, we additionally validate P–J verdicts against independent expert audits of the configured environments (Appendix I), providing an empirical check on both setup correctness and failure attribution.

Cross-language experience transfer. The current ***XPU Database*** is distilled from Python repositories, which limits its coverage of language-specific dependency and build issues in other ecosystems. Future work will expand the experience-generation pool to additional programming languages and evaluate how well ***XPU*** representations and repair strategies transfer across package managers, toolchains, and build systems.