

EnvAgent: Multi-Agent Approach to Building Science Environments in the Clouds

Yiming Cheng
The University of Chicago
Chicago, USA
eaminchan@uchicago.edu

Chris Low
The University of Chicago
Chicago, USA
chrislowzhengxi@uchicago.edu

Binrui Huang
The University of Chicago
Chicago, USA
binruih@uchicago.edu

Mengshi Zhang
TensorBlock, Inc.
USA
mengshiz@tensorblock.co

Zihao Chen
TensorBlock, Inc.
USA
zihao@tensorblock.co

Kate Keahey
Argonne National Laboratory /
The University of Chicago
Chicago, USA
keahey@mcs.anl.gov

Paul Marshall
The University of Chicago
Chicago, USA
marshallp@uchicago.edu

Kexin Pei
The University of Chicago
Chicago, USA
kpei@uchicago.edu

Abstract—AI agents are rapidly transforming scientific computing and broader software-intensive workflows, but their ability to generate, test, and execute code rests on a more fundamental prerequisite: a correctly configured execution environment. Automating environment construction across repositories remains difficult because projects encode heterogeneous and often incomplete requirements for operating systems, language toolchains, dependencies, and build procedures. Existing workflows thus depend on developer-maintained Dockerfiles and setup scripts, making environment preparation labor-intensive and difficult to scale, and potentially biasing scientific evaluations.

We present ENVAGENT, an agentic framework that automatically synthesizes, builds, and verifies repository-level Docker environments. Rather than relying exclusively on generic command-line interaction or narrowly specialized tools that encode fixed assumptions, ENVAGENT combines deterministic utilities for Docker execution and state management with coordinated agents for repository analysis, planning, Dockerfile synthesis, and iterative failure diagnosis and repair. We also introduce ENVBENCH, a benchmark of 54 scientific artifacts and real-world open-source repositories spanning seven programming languages drawn from Chameleon Trovi and Multi-SWE-bench. For each repository, we manually curate a declarative rubric specifying its required structure, configuration, and functionality without prescribing a particular implementation, and a shared evaluator applies 2,108 weighted checks consistently across generated environments. Across four matched language-model settings, ENVAGENT constructs an average of 43.5 executable environments out of 54, compared with 23.5 for the Codex and Claude Code baselines, and achieves 62.02 versus 24.19 on a composite score combining executability and rubric satisfaction. These results show that environment-specific agent orchestration substantially improves automated environment construction. Together, our findings highlight the value of environment-specific orchestration and rubric-based evaluation for developing more reliable and scalable environment-construction agents.

Index Terms—Large Language Models, Software Engineering, Environment Configuration, Multi-Agent Systems

I. INTRODUCTION

The emergence of large language model (LLM)-based agents has revolutionized software development practices, enabling automated code generation, bug fixing, and program synthesis at unprecedented scales [1–10]. These intelligent agents demonstrate remarkable capabilities in understanding and manipulating code at various levels of abstraction, providing

substantial automation benefits across diverse computational software engineering tasks including code modification, performance optimization, and security analysis. Yet turning these capabilities into reliable computational results rests on a more fundamental prerequisite: the target software must build and run in a correctly configured execution environment. For scientific workflows, this environment is part of the experimental artifact because it determines whether results can be reproduced and whether execution-based feedback can be trusted [11, 12].

Despite this critical role, environment construction is commonly treated as an input to agentic workflows rather than an automation target. Existing workflows and benchmarks often rely on repository-specific Dockerfiles and setup scripts maintained by developers or benchmark creators [3, 7]. These artifacts may be incomplete or outdated and can require substantial manual effort to repair and maintain. Moreover, many scientific artifacts and research prototypes are released as code repositories without complete, executable procedures for reconstructing their environments and reproducing the reported results. When their environments cannot be reconstructed reliably, these artifacts become difficult to validate or reuse, weakening the executable basis for verifying the corresponding scientific results. This limitation also narrows the pool of realistic artifacts available for developing and evaluating scientific AI agents, including agents that translate papers into implementations or attempt to reproduce reported experiments [13–15].

Automating this missing step requires substantially more than installing a declared list of packages. Environment requirements are distributed across README files, dependency manifests, build scripts, continuous-integration configurations, and source code, and these sources may be incomplete or mutually inconsistent. For example, a project may declare its language-level packages but omit a required system library, while a prebuilt dependency may require a newer version of `glibc` than the selected base image provides. Two libraries may also require incompatible versions of a shared native runtime, requiring a workaround that is documented by neither library. Moreover, a Docker image that builds and starts is not necessarily correct. Tests may be silently skipped because optional dependencies are missing, or a critical workflow may

fail only when a particular execution path is reached. Such latent configuration errors can create substantial debugging effort and, more seriously, compromise the reproducibility and validity of downstream results.

Challenges for directly applying AI agents. The diversity of repository requirements creates a tension between generality and specialization when automating environment construction. General-purpose code agents such as Codex and Claude Code preserve flexibility by letting LLMs directly operate through generic, command-line tools, e.g., `cat`, `mv`, etc. However, they leave the LLMs responsible for discovering the environment-construction workflow, composing these low-level operations, and maintaining state across repeated build failures. More sophisticated tools, e.g., template-based Dockerfile generators, can encode more of the workflow and alleviate the burden on LLMs to assemble the low-level tools, but are often rigid by assuming fixed build systems or repository structures. These assumptions make them difficult to generalize to diverse projects with native dependencies and nonstandard setup procedures. Our key observation is that environment construction combines a *stable operational structure* with *repository-specific decisions*. Specifically, operations such as initializing a workspace, building and running containers, capturing logs, and tracking execution state can be implemented deterministically, whereas dependency inference, configuration planning, and failure diagnosis require contextual reasoning.

Our approach. Based on this observation, we present ENVAGENT, a multi-agent workflow for automatically constructing repository-level execution environments. By delegating key operations to dedicated LLMs, ENVAGENT combines the reliability of deterministic tools and the flexibility of LLMs for diverse scenarios. Specifically, ENVAGENT assigns stable environment-management operations to deterministic utilities and uses coordinated LLM agents to inspect repository artifacts, formulate a setup plan, synthesize a Dockerfile, and diagnose and repair failures. After each construction attempt, ENVAGENT captures build and runtime feedback, summarizes the current state, and uses this information to guide the next attempt. This task-specific division of responsibilities structures the environment-construction process without hard-coding assumptions about a particular programming language, package manager, or build system.

Our benchmark. Environment construction also presents a distinct evaluation challenge. Build and launch success often only checks that the repository is executable after its environment is constructed, but not that it contains the configuration and functionality required by the repository. Traditional software tests generally presuppose a working environment and do not systematically characterize which environment requirements have been satisfied. We thus introduce ENVBENCH, a benchmark containing 54 scientific artifacts and real-world open-source repositories drawn from Chameleon Trovi and Multi-SWE-bench and spanning seven programming languages. For each repository, we construct a declarative rubric that specifies the required environment structure, configuration, and functionality without prescribing how these requirements must be implemented. A shared evaluation engine applies 2,108 weighted checks expressed through reusable checker primitives. While there are approaches leveraging LLMs as a judge for evaluation [16], we deliberately rely on manual process to

curate these rubrics to ensure reliability.

We evaluate ENVAGENT on ENVBENCH using Chameleon Cloud, an OpenStack-based research testbed [17]. Across the fixed model settings, ENVAGENT constructs an average of 43.5 executable environments out of 54, whereas the agentic baselines like Codex and Claude Code construct 23.5. ENVAGENT also achieves 62.02, compared with 24.19 for the baselines, on a composite score combining environment executability and rubric satisfaction. These gains require 2–10 $\times$ greater token consumption, exposing a tradeoff between computational cost and the iterative diagnosis needed to configure difficult repositories reliably. As a preliminary extension, we also explore whether requirements inferred from a repository can guide the provisioning of compatible cloud resources, one step before setting up the execution environment when the repositories need to be deployed on the cloud platform (Section V). This work makes the following contributions:

- We introduce ENVAGENT, an agentic workflow that combines deterministic environment-management utilities with coordinated agents to synthesize, build, diagnose, and verify repository-level Docker environments to support scientific evaluation and reproducibility.
- We develop ENVBENCH, a human-curated benchmark of 54 repositories and 2,108 weighted checks for evaluating both environment executability and satisfaction of repository-specific functionality requirements.
- We conduct various comparisons with frontier agent frameworks, such as Codex and Claude Code, using the same LLMs, and demonstrate promising preliminary improvements in environment construction, while sharing more lessons learned for future works on automated execution environment construction and configurations.

II. OVERVIEW AND MOTIVATION

Problem formulation. We formulate the repository-level environment construction as follows. Given a source repository and a target machine, construct a Dockerfile that provides the software stack required to build and execute the repository. Our primary task begins after the target machine becomes available, although the constructed environment must remain compatible with its operating system, processor architecture, and other hardware resources. We also include a preliminary study when these hardware resources need to be automatically configured, e.g., in a cloud environment (Section V-C). Cloud resource provisioning and multi-node orchestration are broader concerns that we consider only as preliminary extensions. Importantly, a successful environment should not just build and start, but also provide the dependencies, tools, configuration, and functionality expected by the repository.

Key challenges. The key challenge of environment construction is that its requirements rarely appear in a single well-defined authoritative specification. Instead, the steps and requirements are often distributed across README files, dependency configurations, build scripts, continuous-integration configurations, container specifications, and even the artifacts in source code, e.g., developer comments, docstrings, etc. These sources may also be incomplete or mutually inconsistent. For example, a package manifest may declare language-level dependencies while omitting a required compiler or system library, and a prebuilt dependency may require a newer version of `glibc`

than the selected base image provides. A repository may also combine multiple ecosystems, e.g., a Python package that builds a C/C++ extension or invokes an external system tool not managed by the Python package manager.

These constraints are interdependent and often non-monotonic. Installing or upgrading one package may resolve the current error while introducing a conflict with another dependency. The order of configuration operations can also matter, e.g., a compiler or system header must be installed before a native dependency can be built. As each construction attempt exposes only the failures reached under the current configuration, environment construction naturally requires an iterative process of execution, diagnosis, and revision [18, 19]. These challenges have made automated solutions extremely challenging that manual tuning has been the only way for this task. Unfortunately, a manually constructed environment does not by itself provide a repeatable specification from which the experiment can be reproduced. AI-based techniques, especially with recent advance in LLM agents, have thus become a promising direction given its capabilities from handling much larger space of uncertainties [11].

Selective specialization for environment construction. Existing off-the-shelf agents provide incomplete solutions to this problem. General-purpose code agents preserve flexibility by operating through generic command-line and file-editing tools, but leave the LLMs responsible for decomposing the construction workflow, composing low-level operations, interpreting build logs, and maintaining state across repeated failures. On the other hand, highly specialized agent tools, e.g., analyzing source code comments, can reduce this reasoning burden, but often encode assumptions about particular repositories, package managers, or build systems. Such assumptions make them difficult to apply to diverse repositories with mixed requirements, dependencies, or nonstandard setup procedures.

Environment construction thus benefits from selective rather than complete specialization. Stable operations, e.g., workspace initialization, Docker execution, log collection, state tracking, and termination, can be implemented as deterministic utilities. Repository-dependent decisions, e.g., inferring missing dependencies, selecting compatible versions, planning configuration steps, and repairing failures, require contextualized reasoning hard to be canonicalized. This division reduces the burden of repeatedly assembling low-level operations without constraining the agent to a fixed repository structure or software ecosystem.

Need for environment-level verification. Docker build and launch success are insufficient indicators of environment correctness. A container may build successfully while containing the wrong dependency version, omitting a required executable or environment variable, or silently failing tests because an optional dependency is absent. Traditional software testing also presuppose a usable environment and may not identify which environment requirements are satisfied or missing. Evaluating environment construction thus requires explicit, repo-specific criteria covering structure, configuration, and functionality. These criteria should specify what the environment must provide without prescribing how the agent constructs it, allowing different but functionally equivalent environments to be evaluated consistently.

These observations motivate our environment-construction-specific agentic workflow that combines deterministic tool sup-

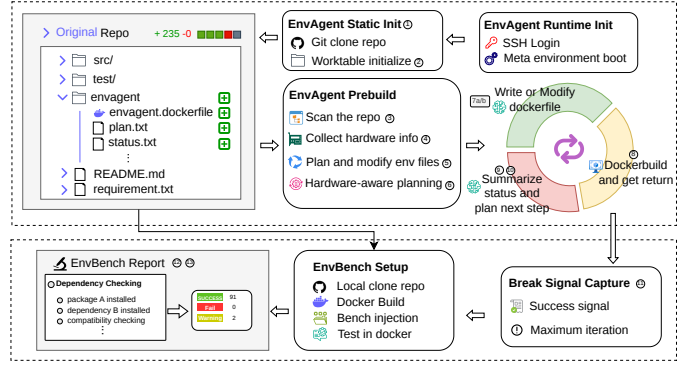


Fig. 1. Overview of ENVAGENT on ENVBENCH. Given a target repository, ENVAGENT coordinates specialized agents to analyze the repository, plan the environment, construct and revise a Dockerfile, and interpret execution feedback. ENVBENCH provides the repository for which ENVAGENT is tasked to construct the environment, and declarative evaluation rubrics to compute fine-grained evaluation signals. The numbered workflow components are detailed in Table I.

port with flexible repo-level reasoning and iterative repair. They also motivate a human-curated, rubric-based benchmark that evaluates both whether an environment executes and whether it satisfies the repository’s actual requirements. ENVAGENT and ENVBENCH instantiate these two requirements, respectively.

III. METHOD

This section overviews the workflow of ENVAGENT and its interaction with ENVBENCH. Figure 1 shows an overview.

A. Overview of ENVAGENT

ENVAGENT orchestrates multiple specialized agents through a three-phase pipeline comprising *preprocessing and planning*, *execution and feedback*, and *summarization and context management*. At each iteration, ENVAGENT employs the workflow following ReAct [20], where agents generate reasoning traces and execute actions, then incorporate environmental feedback to refine subsequent operations.

Phase 1: Preprocessing and planning. ENVAGENT initiates environment configuration through coordinated context collection by the following specialized agents. The *Scanning Agent* traverses the repository structure to construct a comprehensive file tree and identifies environment-relevant artifacts (e.g., configuration files, dependency manifests, build scripts). Concurrently, the *Hardware Checking Agent* performs system introspection to gather hardware specifications including CPU architecture, GPU models and memory capacity, system RAM, disk space, and operating system details. Subsequently, the *Planning Agent* synthesizes the collected repository context and hardware constraints to formulate a comprehensive environment configuration strategy, generating a step-by-step execution plan for containerized deployment.

Phase 2: Execution and feedback. Based on the preprocessing phase outputs, ENVAGENT enters an iterative refinement loop to progressively construct a functional Dockerfile capable of executing the target repository within the current hardware environment. This phase employs two complementary agents. The *WriteDocker Agent* generates or revises Dockerfile specifications based on the established plan and current execution state,

TABLE I
OVERVIEW OF **MAINAGENT**. IN TYPE COLUMN, **RULE-BASED** MEANS PURELY SCRIPT-DRIVEN PROCESSING OF THE WORKSPACE, **AGENT-BASED** MEANS CONTEXT-AWARE ANALYSIS AND MANIPULATION OF THE WORKSPACE BY AN AGENT.

Order	Component	Description	Type
Initialization Phase			
①	Stats	Records session start time and system metrics for performance tracking	Rule
②	Workspace Initial	Creates workspace structure with essential files (plan, status, etc.)	Rule
Analysis & Planning Phase			
③	Scanning	Scans repository structure, builds file tree, and index environment files	Agent
④	Hardware Checking	Detects hardware specifications using system introspection and LLM	Agent
⑤	Planning	Synthesizes repository context to generate build strategy	Agent
⑥	Hardware Adjustment	Refines the initial plan based on hardware limitations	Agent
Iterative Execution Phase			
⑦	Docker Initial	Generates initial Dockerfile based on planning phase outputs	Agent
⑧	Docker Revision	Revises Dockerfile based on execution feedback and error analysis	Agent
⑨	Run Dockerfile	Executes Dockerfile in containerized environment and captures logs	Rule
⑩	Summarize	Analyzes execution results, identifies errors, and generates actionable recommendations for next iteration	Agent
⑪	Logging	Aggregates current iteration status into disk and updates workspace	Rule
⑫	Status Checking	Evaluates success criteria and end construction	Rule
Finalization Phase			
⑬	Stats	Records session end time and computes metrics	Rule
⑭	Report	Displays formatted execution time and status	Rule

while the *RunDocker Agent* executes the generated Dockerfile within an isolated containerized environment and captures detailed execution logs and error diagnostics.

Phase 3: Summarization and context management. Following each execution cycle, the *Summarize Agent* processes the execution feedback to synthesize the current Dockerfile completion status and formulate actionable modification recommendations. This agent maintains contextual continuity across iterations by organizing error information and corrective suggestions into structured feedback for subsequent refinement cycles. Additionally, it monitors success metric to determine loop termination conditions, ensuring efficient convergence to a functional environment configuration.

Rule-based tool integration. To optimize computational costs and enhance operational efficiency, ENVAGENT strategically incorporates rule-based auxiliary tools alongside its intelligent agent framework. These deterministic components handle routine operational tasks that do not require sophisticated reasoning capabilities, thus reducing token consumption while maintaining system reliability.

The framework employs three primary rule-based utilities:

- *Workspace Initialization*, which systematically provisions the ENVAGENT working directory structure and creates essential configuration files (e.g., `plan.txt`, `history.txt`, `status.txt`, `EnvAgent.dockerfile`, and hardware specification files) through file system operations;
- *Directory Scaffolding*, which establishes standardized folder hierarchies and dependency tracking mechanisms without requiring agent intervention; and
- *Workspace State Management*, which maintains execution context through automated log aggregation, iteration tracking, and status synchronization across the multi-agent workflow.

The goal of this hybrid architecture is to leverage rule-based automation for predictable tasks while reserving LLM capabilities for complex reasoning and decision-making processes, making the overall agentic workflow token-efficient and remain generalizable to diverse scenarios. Table I presents a comprehensive overview of the EnvAgent framework’s components and execution orders.

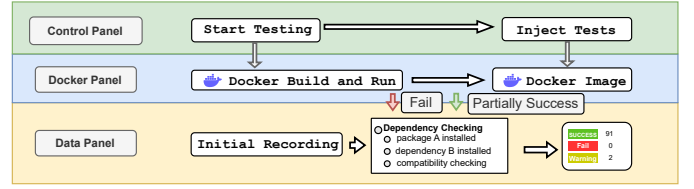


Fig. 2. Overview of ENVBENCH. Given a generated Dockerfile, ENVBENCH builds and launches the corresponding container, injects repository-specific declarative rubric checks, and executes them inside the constructed environment. It records each check as pass, warning, or fail and aggregates the outcomes into a report covering environment structure, dependency integrity, and functional correctness.

B. ENVBENCH Curation

To systematically evaluate automated environment configuration, we introduce ENVBENCH, a benchmark spanning 54 carefully curated repositories from two complementary sources: (1) reproducible research artifacts from the Chameleon Trovi service [21], and (2) multi-lingual open-source software from Multi-SWE-bench, covering more than seven programming languages [22]. ENVAGENT employs a three-dimensional framework tailored to environment construction assessment.

- *Basic Structure Verification* – validates the presence of essential files, directories, and environment variables required for proper software operation;
- *Dependency Integrity Assessment* – verifies not only dependency installation but also version compatibility and configuration correctness; and
- *Functional Validation* – executes targeted commands to confirm that the environment supports the intended software functionality without requiring full application execution.

Our evaluation design focuses on environment configuration rather than comprehensive software testing. Because the objective is to verify that an environment is correctly established, most checks do not require complete application execution. Instead, ENVBENCH evaluates the Docker containers produced from generated Dockerfiles by running environment-specific validation commands and checking their outputs. Functionally equivalent environments therefore receive consistent scores even when their implementations differ.

IV. EXPERIMENTAL SETUP

Datasets. Our benchmark comprises 54 heterogeneous repositories (Table II) covering a broad range of reproducibility requirements and application domains. Among them, 21 come from Trovi hosted on Chameleon [23] with the label “reproducible research” [17], so we can ensure the evaluation in the benchmark for environment setup is grounded. The remaining 33 artifacts come from Multi-SWE-bench [22] covering popular real-world software beyond research prototypes, which further expand the benchmark diversity in terms of programming languages (e.g., Java, TypeScript, JavaScript, Go, Rust, C, and C++) and software functionalities. Such a diverse dataset can evaluate the agent across different programming paradigms, build systems, and deployment requirements, providing a comprehensive testbed for automated environment configuration capabilities.

Baselines. We consider two widely used, representative agentic frameworks as our baselines: (1) *Claude Code* based on

Anthropic’s reasoning-optimized models claude-opus-4 and claude-haiku-3.5, and (2) *Codex* based on OpenAI’s code-specialized models gpt-4.1 and gpt-4.1-mini. We intentionally pick the cheap models to evaluate the delta brought by our agent harness, as powerful frontier models can often complement weak agentic architectures, making it unclear to evaluate the contribution of the agent framework. Similar to EnvAgent, the baseline methods are provisioned with unrestricted access to the target repository and complete hardware inspection capabilities to ensure a fair comparison. All baselines and ENVAGENT are initialized with an identical system prompt:

System Prompt

Please take a careful look at the current hardware and repo, and create a new Dockerfile named `codex.dockerfile` (or `claude.dockerfile`) at the root of this repo that, when built, puts me in a `/bin/bash` CLI setting at the root of the repository, with the repository installed.

This prompt instructs the model to inspect both the available hardware and the project repository, and to generate a new Dockerfile at the root of the repository. When built, the Dockerfile will include a working environment in which the repository is correctly installed and accessible from `/bin/bash` command-line interface at the repository root. This ensures that each system begins with the same context and a consistent environment setup procedure critical for reproducibility and fair evaluation.

A. Evaluation Rubrics

To systematically evaluate ENVAGENT’s environment construction capabilities, we develop ENVBENCH—a benchmark and, more importantly, a *general, rubric-driven evaluation framework* that targets the often-overlooked problem of verifying whether an execution environment has been correctly *constructed*, rather than whether software features are correct. Unlike ordinary unit tests, which presuppose a working environment, ENVBENCH checks the environment itself; and unlike prior efforts that rely on bespoke, per-repository validation scripts, it lets users specify only a declarative *rubric* while a shared engine performs the actual checking. The rubric defines a set of fine-grained requirements that a correctly constructed environment should satisfy, while the shared evaluation engine executes these checks and aggregates their outcomes into a quantitative environment-quality score. We detail the rubric design, its taxonomy of checks, and the evaluation procedure below.

From bespoke scripts to declarative rubrics. A key obstacle to environment-focused evaluation is that, unlike ordinary unit tests which presuppose a working environment, validating environment *construction* has traditionally required hand-written, repository-specific shell scripts—tedious to author and hard to compare across projects. To remove this burden, we design ENVBENCH as a *general, declarative evaluation framework*: the user specifies only a *rubric*—a machine-readable description of what a correctly configured environment must satisfy—while a shared execution engine performs the checking. This cleanly separates *what* to verify (the rubric, the only artifact the user authors) from *how* to verify it (reusable checker primitives supplied by the framework), so extending the benchmark to

TABLE II
ENVAGENT DATASET: REPOSITORY OVERVIEW

Repo	Lang	Tests	Repo	Lang	Tests	Repo	Lang	Tests
Chameleon Trovi (21 repos)								
Reltr	Python	71	Lottory	Python	97	Seed-gnn	Python	73
Tabpfn	Python	69	Rsn	Python	84	P4ctl	Python	110
Crossprefetch	C/C++	67	Symmc	Java	40	Fairify	Java	72
Exli	Java	49	Sixthsense	Python	64	Probfuzz	Python	104
Glutest	Python	82	Flex	Python	73	Acto	Python	63
Baleen	C/C++	63	Silhouette	C/C++	74	Anvil	Rust	70
Elect	C/C++	45	Rfuse	C/C++	48	Metis	C/C++	104
Multi-SWE-bench (33 repos)								
Zstd	C/C++	71	Jq	C/C++	95	Catch2	C/C++	39
Fmt	C/C++	105	Json	C/C++	96	Simdjson	C/C++	59
Cpp-httpilib	C/C++	41	Cli	Go	72	Grpc-go	Go	104
Go-zero	Go	34	Fastjson2	Java	52	Logstash	Java	42
Mockito	Java	101	Github-readme-stats	JS	59	Axios	JS	76
Express	JS	50	Daysjs	JS	101	Insomnia	JS	104
Svelte	JS	74	Ripgrep	Rust	82	Clap	Rust	59
Nushell	Rust	109	Serde	Rust	65	Bat	Rust	72
Fd	Rust	58	Rayon	Rust	81	Bytest	Rust	60
Tokio	Rust	60	Tracing	Rust	66	Darkreader	TS	26
Material-ui	TS	32	Core	TS	42	Ponyc	C/C++	74

a new repository reduces to writing one rubric rather than a validation program.

Rubric specification. A rubric is a list of fine-grained checks, where each check declares a reusable checker primitive (its *type*), the primitive’s parameters, a score weight, a semantic category, and optional prerequisite checks (*requires*) that impose execution ordering. Each check evaluates one concrete property of the constructed environment, such as whether a required file or directory exists, whether a required executable is available, whether an environment variable is correctly set, or whether a setup, build, or test command succeeds. Seven primitives suffice to express every check in our suite: `run_command` (a shell command must exit successfully), `commands_exist` (executables are on `PATH`), `files_exist/dirs_exist` (paths are present), `file_contains/output_contains` (a file, or a command’s output, contains required content), and `envvar_set` (an environment variable is set). Because these primitives are declarative and composable, authoring a rubric requires no scripting.

Two-level taxonomy. Each check is organized along a two-level taxonomy (Table III). The first level is a *validation category* capturing which aspect of environment readiness is probed: *Structure* (the repository layout and declared dependencies are in place), *Configuration* (tools, interpreters, versions, and environment variables are correctly installed), and *Functionality* (build, test, and run commands actually execute successfully), together with a minimal *Integration* probe for end-to-end container startup. The second level is the checker primitive that realizes the check. Table III reports the full cross-classification with per-cell counts over the rubric suite.

Execution and scoring. Evaluation proceeds in three automated stages. First, the rubric is injected into the ENVAGENT workspace. Second, the generated Dockerfile is built into a container. Third, the evaluation engine executes every rubric check inside the resulting container while honoring *requires* dependencies. Each check produces one of three outcomes: PASS, FAIL, or WARNING. These outcomes are converted into numerical scores and aggregated according to the check weights to obtain a repository-level rubric score. Specifically, we assign a normalized outcome score of 1 to PASS, 0 to FAIL, and α to

TABLE III
ENVBENCH RUBRIC TAXONOMY. EACH CHECK IS CLASSIFIED BY A FIRST-LEVEL VALIDATION *category* (WHICH ASPECT OF ENVIRONMENT READINESS IT PROBES) AND REALIZED BY A SECOND-LEVEL REUSABLE *checker primitive* (TYPE). COUNTS ARE AGGREGATED OVER THE RUBRIC SUITE (2,108 CHECKS).

Category (L1)	Checker primitive (L2)	Each check asserts	#
Structure (579)	files_exist	required files are present	227
	dirs_exist	required directories are present	145
	file_contains	a file pins required content	123
	run_command	structural probe command succeeds	84
Configuration (863)	run_command	install/setup command succeeds	440
	commands_exist	required tools are on PATH	227
	output_contains	command output matches (e.g., version)	118
	envvar_set	environment variable is set	62
	other	file/dir presence checks	16
Functionality (665)	run_command	build/test/run command exits 0	630
	output_contains	program output matches expected	17
	other	artifact presence checks	18
Integration (1)	run_command	end-to-end minimal container run	1
Total	7 primitives		2,108

WARNING, where $\alpha \in [0, 1]$ is a fixed partial-credit value. For repository r_i with M_i rubric checks, check k has weight w_{ik} and outcome score s_{ik} . The resulting rubric-based environment quality score is therefore $STSSR_i = \frac{\sum_{k=1}^{M_i} w_{ik} s_{ik}}{\sum_{k=1}^{M_i} w_{ik}}$. Thus, the rubric defines *what* is evaluated, while $STSSR$ quantitatively summarizes *how well* the constructed environment satisfies those requirements. In parallel, successful Docker build and runtime execution determines whether the environment is executable, which is used to compute $SECSR$. The two metrics capture complementary aspects of environment construction and are combined into the final composite score described in Section IV-B. This framework turns benchmark curation into rubric authoring: across the ENVBENCH suite, the rubrics encode 2,108 weighted checks spanning three primary validation categories, one integration probe, and seven checker primitives (Table III). Together, these checks provide a consistent and reusable evaluation interface for assessing ENVAGENT’s ability to construct environments across diverse software ecosystems and build complexities. Table II summarizes the repositories covered by ENVBENCH.

B. Evaluation Metrics

To comprehensively assess ENVAGENT’s performance in automated environment configuration, we employ complementary metrics that capture two distinct aspects of environment construction: (1) whether the agent can successfully construct an executable environment, and (2) how well the resulting environment satisfies the rubric-defined requirements. We further combine these two aspects into a single overall score.

Executable count. This metric measures how many functional Docker images ENVAGENT successfully generates that build and run without critical errors on the target hardware. For a given repository r_i and hardware configuration h_j , we define $executable_{i,j}$ as a binary indicator that equals 1 if the generated Dockerfile builds and runs successfully, and 0 otherwise. The total number of executable environments across the dataset is $executable = \sum_{i=1}^N executable_{i,j}$, where N represents the total number of repositories in the evaluation dataset.

Environment Construction Success Rate ($SECSR$). Besides the total count, we report the proportion of repositories for which ENVAGENT successfully constructs an executable environment: $SECSR = \frac{1}{N} \sum_{i=1}^N executable_{i,j}$. $SECSR$ measures the agent’s fundamental ability to produce a viable environment, independently of the fine-grained rubric score.

Test Suite Success Rate ($STSSR$). For each executable repository, $STSSR$ measures the quality of the constructed environment based on the weighted outcomes of its ENVBENCH rubric checks. For repository r_i , let M_i denote the number of rubric checks, w_{ik} the weight assigned to check k , and s_{ik} its normalized outcome score. We define the outcome score as $s_{ik} = 1$ for PASS, $s_{ik} = \alpha$ for WARNING, and $s_{ik} = 0$ for FAIL, where $\alpha \in [0, 1]$ is the fixed partial-credit value assigned to warnings. The repository-level $STSSR$ is then $STSSR_i = \frac{\sum_{k=1}^{M_i} w_{ik} s_{ik}}{\sum_{k=1}^{M_i} w_{ik}}$. This formulation ensures that more important environment requirements contribute proportionally more to the final score, while warnings receive partial credit rather than being treated as either complete success or complete failure. Let $\mathcal{R} = \{i \mid executable_{i,j} = 1\}$ denote the subset of repositories with executable environments. We aggregate $STSSR$ over this executable subset as the weighted mean of repository-level rubric scores: $STSSR = \frac{1}{|\mathcal{R}|} \sum_{i \in \mathcal{R}} STSSR_i$. Thus, $STSSR$ evaluates environment quality conditional on successful execution, while $SECSR$ accounts for repositories for which an executable environment could not be constructed.

Final Composite Score (FCS). To summarize both environment construction capability and environment quality in a single scalar, we compute $FCS = \left(\frac{SECSR}{100}\right) \times STSSR$. Here, $SECSR$ is expressed as a percentage, whereas $STSSR$ is normalized to $[0, 1]$. FCS therefore captures both the likelihood that a repository has an executable environment and the quality of that environment conditional on executability. If no repository is executable (i.e., $|\mathcal{R}| = 0$), we define $FCS = 0$. Together, these metrics provide complementary views of ENVAGENT’s effectiveness. Executable count reports the absolute number of viable environments, $SECSR$ normalizes construction success across datasets, $STSSR$ measures the quality of executable environments using the weighted rubric checks, and FCS integrates environment construction success and environment quality into a single overall measure. In this formulation, the rubric is the fine-grained evaluation specification, the individual check outcomes provide the underlying evaluation signals, and the metrics aggregate these signals into standardized measures for quantitative comparison across repositories and environments.

C. Implementation

We consider a diverse set of LLMs to ensure a comprehensive evaluation (Table IV). To enable systematic comparison and streamline the model management, we develop a unified API management framework using Forge [24], which provides centralized model orchestration and usage analytics.

Unified interface and configuration management. Our framework abstracts model-specific implementations through a standardized interface layer, enabling seamless model switching across all agent components without code modification. Model selection and parameters are managed through a declarative configuration system that supports both global and component-

specific model assignments. Researchers can easily configure experiments to compare model performance across the entire ENVAGENT pipeline or evaluate specific agents with different models while maintaining consistency in other components. This design ensures that complex multi-agent workflows can be executed with consistent parameters while maintaining model-agnostic operation logic, and provides the flexibility needed for systematic ablation studies and model comparison experiments with minimal configuration overhead.

Centralized usage tracking. The Forge-based management system provides comprehensive usage analytics, including token consumption, API call frequencies, and cost tracking across all models and agent components. This centralized monitoring enables precise resource accounting and facilitates fair comparison between different model configurations. The system automatically logs detailed metrics for each agent invocation, supporting both per-iteration and aggregate analysis of model utilization patterns. This unified model management approach ensures experimental reproducibility while providing the flexibility needed for comprehensive model evaluation across the complex multi-agent ENVAGENT workflow.

Infrastructure and implementation architecture. We conduct all experiments on the Chameleon Cloud testbed using single-node servers configured with x86_64 architecture running Ubuntu 20.04 LTS. For repositories requiring GPU acceleration, we provision NVIDIA A100 GPUs to ensure compatibility with compute-intensive workloads. We implement ENVAGENT from the ground up and evaluate every method on the same pre-provisioned hardware for a controlled comparison. A separate preliminary study of settings without pre-provisioned hardware is reported in Section V-C.

V. EVALUATION

We aim to answer the following research questions:

- **RQ1:** How effective is ENVAGENT at generating executable Docker environments compared to baseline approaches?
- **RQ2:** What is the overall quality and functional correctness of environments generated by ENVAGENT as measured by comprehensive benchmarks?

A. RQ1: Environment Generation Effectiveness

As shown in Table IV, ENVAGENT produces more executable environments than both baselines across matched model families. Its executable rates range from 66.7% to 90.7%, while the baselines achieved 37.0% to 51.9%, respectively. Note that executability is a prerequisite for this task, so if a Docker image cannot be built and run, downstream dependency and functional checks cannot be applied, and the agent receives a score of zero.

For the resource efficiency, e.g., tokens, time, dollar cost, etc., in Table IV, *All* reports the mean over all 54 repository runs for each agent-model configuration, including unsuccessful attempts, whereas *Success* reports the conditional mean over only the runs that produced an executable environment. Comparing these columns helps distinguish the overall evaluation cost from the resources consumed by successful constructions. In particular, failed runs may terminate early and lower the *All* average, while successful runs typically include the full build, execution, and validation workflow. The higher executable counts come with increased token consumption.

TABLE IV

ENVAGENT RESULTS ACROSS 54 REPOSITORIES PER AGENT-MODEL CONFIGURATION. FOR TOKENS, TIME, AND COST, *All* IS THE MEAN OVER ALL RUNS, WHEREAS *Success* IS THE MEAN OVER RUNS THAT PRODUCED AN EXECUTABLE ENVIRONMENT. INPUT/OUTPUT TOKENS ARE MEASURED IN THOUSANDS, COST IS IN DOLLARS, AND FCS (FINAL COMPOSITE SCORE) IS IN %.

Agent	Model	Executable		Input Tokens		Output Tokens		Time		Cost		FCS
		Count	Rate (%)	All	Success	All	Success	All	Success	All	Success	
codex	gpt-4.1	21	38.9	376.6	418.1	0.3	0.2	0:00:46	0:00:46	0.75	0.83	26.15
	gpt-4.1-mini	28	51.9	331.2	282.6	0.3	0.3	0:00:39	0:00:33	0.13	0.11	22.31
claude-code	claude-opus-4	25	46.3	26.6	24.7	0.3	0.2	0:01:24	0:01:15	0.58	0.58	30.10
	claude-haiku-3.5	20	37.0	18.6	17.9	0.2	0.2	0:00:23	0:00:23	0.02	0.02	18.21
EnvAgent	gpt-4.1	48	88.9	465.2	375.9	105.5	100.2	0:32:30	0:27:28	2.66	2.33	70.07
	gpt-4.1-mini	36	66.7	368.4	253.7	90.2	95.6	0:31:36	0:28:27	0.58	0.51	53.41
	claude-opus-4	49	90.7	322.8	294.1	75.5	79.9	0:42:55	0:45:01	10.5	10.1	74.01
	claude-haiku-3.5	41	75.9	272.3	184.3	15.2	14.7	0:11:39	0:10:43	0.27	0.21	50.59
	deepseek-r1	30	55.6	351.9	248.6	104.7	76.6	1:42:10	0:57:30	0.40	0.29	42.25
	deepseek-v3	39	72.2	319.1	278.4	42.2	44.4	0:27:21	0:27:33	0.13	0.13	49.54
	gemini-2.5-pro	35	64.8	556.2	612.1	198.9	239.6	0:50:39	0:54:07	2.68	3.36	66.81

ENVAGENT allocates additional reasoning budget to its plan-execute-feedback-revise loop (Section III-A), including multi-round error localization and Dockerfile repair. Baselines often terminate earlier after a failed attempt and therefore consume fewer tokens, but also construct fewer executable environments. Across model families, stronger reasoning models (e.g., GPT-4.1 and Claude Opus series) achieve more executable environments, whereas smaller models offer lower cost and latency. These observations motivate a possible curriculum policy that first uses a lower-cost model and escalates only after an unsuccessful attempt.

Longer runs are often dominated by compilation and image-building costs rather than agent inference alone. The standardized evaluation harness records the same inputs and measurements for every method. Build caching and artifact reuse could thus reduce both latency and cost in future implementations.

As shown in Table IV, ENVAGENT produces 36–49 executable environments across the four matched model settings, while the baselines produce 20–28. Averaged over these settings, the corresponding counts are 43.5 and 23.5, respectively.

B. RQ2: Environment Quality and Functional Correctness

We report ENVBENCH composite scores (defined in Section IV-B) across agents and models. As shown in Table IV, ENVAGENT achieves an average FCS of 62.02 across the four matched model settings, compared with 24.19 for the baselines—a margin of 37.83 percentage points. The improvement is consistent within each matched model family: FCS increases from 26.15 to 70.07 with GPT-4.1, from 22.31 to 53.41 with GPT-4.1-mini, from 30.10 to 74.01 with Claude Opus 4, and from 18.21 to 50.59 with Claude Haiku 3.5. Thus, the gains are not attributable to a single model provider or model scale.

The strongest ENVAGENT configuration reaches an FCS of 74.01, while even its weakest configuration among all seven evaluated models reaches 42.25, exceeding the strongest baseline at 30.10. This distinction is important because FCS jointly captures whether an environment is executable and how well an executable environment satisfies the weighted ENVBENCH rubrics. Together with the executable-count results in RQ1, these scores show that the gains persist under an end-to-end metric that evaluates rubric satisfaction rather than stopping at the initial build stage.

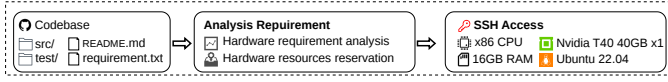


Fig. 3. Preliminary repository-based hardware provisioning for cloud experiments. The information in the repository is used to infer hardware requirements to provision, which are then matched against the provisioned cloud resources before SSH access is returned.

C. Moving to Cloud Environment

In this paper, the main evaluation assumes that a machine with fixed hardware is already available, so the effort is focused on constructing the environment on the given machine. As a preliminary effort, we also briefly explore the setting when an experiment must run on Cloud environment where the required hardware has not yet been provisioned [11, 21]. In this study, we deliberately obtained 20 additional repositories from Trovi with heterogeneous hardware requirements and is separate from the main 54-repository ENVBENCH evaluation. The goal of this study is to understand whether we can easily extend our

As Figure 3 shows, an agent infers CPU, memory, storage, operating-system, and accelerator requirements from repository evidence, matches them to available Chameleon resources, and returns SSH access after reservation [17, 25]. Representative cases are reported in Table V. As this study is preliminary, we do not treat the workflow as a another system or include it in the main ENVAGENT-ENVBENCH comparison.

VI. RELATED WORK

Agentic coding benchmarks. Recent benchmarks evaluate code agents on software engineering tasks including automated bug repair [1–4, 26], code-efficiency optimization [5, 6, 27], code migration and transformation [7, 28], and penetration testing [29–35]. Building these benchmarks often requires manually constructing executable Docker environments [36], which must be maintained as dependencies evolve, limiting scalability and potentially excluding repositories with complex or changing build requirements [18]. ENVAGENT addresses this infrastructure bottleneck by automating repository-level environment construction.

Repo-level execution environment construction. The reproducibility crisis in computational research has motivated efforts to automate environment setup and result reproduction [37]. Container-based solutions such as Docker improve portability, but creating reliable Dockerfiles still requires substantial effort from domain experts [38]. Existing LLM-based approaches provide partial support for dependency installation and configuration, but do not systematically combine iterative construction with rubric-driven environment evaluation [10]. Research reproduction benchmarks, such as PaperBench [15], focus primarily on reproducing computational experiments rather than systematic environment configuration assessment. Traditional reproducibility tools, such as ReproZip [39] and CDE [40], can capture computational requirements but require manual intervention for complex dependency chains [39, 40].

Code agents. LLM-based software engineering agents have rapidly evolved to automate a broad range of development tasks [1, 8, 9, 41–48]. Notably, Claude Code [49] and Codex [50] provide agentic coding capabilities that run directly in terminals, supporting end-to-end workflows including feature

implementation from natural language descriptions, debugging through error analysis, and codebase navigation with global project awareness.

However, most code agents rely on an already executable environment for test-driven development and execution-based evaluation [27, 51]. This assumption restricts their applicability to repositories whose environments can be prepared reliably [18]. ENVAGENT is complementary: it constructs the execution environment needed to extend agent training and evaluation to a broader set of repositories.

VII. THREATS TO VALIDITY

Limited evaluated models and baselines. Our evaluation covers seven model configurations, including GPT-4.1, Claude Opus 4, DeepSeek-R1, and Gemini-2.5-Pro (Table IV). These models may not represent the performance of future model families. Our baseline comparison is also limited to two general-purpose code agents, Codex and Claude Code, because we are not aware of a directly comparable agent specialized in end-to-end environment construction. Consequently, the measured improvements should not be generalized to every model or agent architecture.

Increased resource consumption. ENVAGENT’s plan–execute–feedback–revise loop consumes substantially more tokens and wall-clock time than the baselines in several configurations. This cost may limit practical adoption even when the resulting environment is more reliable. Build caching, artifact reuse, model cascading, and more efficient error localization could reduce this overhead.

Dockerfile-centric environment definition. Our framework defines environment construction success based on whether a single-container Dockerfile satisfies the declarative rubric checks in ENVBENCH. This definition may not fully capture real-world systems that require multi-node deployments, distributed architectures, external services, or orchestration platforms beyond a single container. While this choice is consistent with common practices in SWE-related benchmarks and the widespread use of Dockerfiles for environment specification, our preliminary cloud study only explores hardware provisioning and does not address these multi-container and orchestration requirements.

VIII. CONCLUSION

Execution environment construction remains a bottleneck in scaling the training and evaluation of code agents across real-world repositories. We present ENVAGENT, a multi-agent framework that analyzes a repository, plans its dependencies, and iteratively constructs and repairs a Dockerfile using execution feedback. We also present ENVBENCH, a rubric-driven benchmark covering 54 open-source repositories and reproducible research artifacts. ENVAGENT constructs an average of 43.5 executable environments across four matched model settings, compared with 23.5 for Codex and Claude Code, and achieves an average FCS of 62.02 versus 24.19. We additionally report a preliminary study of repository-driven hardware provisioning when a cloud machine is not yet available, separate from the core ENVAGENT-ENVBENCH contribution. We will release ENVAGENT, ENVBENCH, and the evaluation artifacts to support future research on reproducible code-agent infrastructure.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments and feedback, which helped us improve this paper. We also thank Maharani Irawan (University of Chicago) for early discussions on the definitions and boundaries of agents and tools. We are grateful to Professors Yiling Lou and Lingming Zhang (University of Illinois Urbana-Champaign) and Professor Claire Le Goues (Carnegie Mellon University) for valuable discussions on the design principles and perspectives of agents and software engineering. This work was supported by the National Science Foundation (Award No. 2431425), the Chameleon testbed [17], the TensorBlock Researcher Access Program, and the OpenAI Researcher Access Program. Generative models were used only to assist with language editing of this manuscript and the development of implementation code. They were not used to develop the research idea, formulate the problem, design the experiments, analyze the results, or make other substantive research decisions. All AI-assisted text and code were reviewed by the authors.

REFERENCES

- [1] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, “Marscode agent: Ai-native automated bug fixing,” *arXiv preprint arXiv:2409.00899*, 2024.
- [2] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” *arXiv preprint arXiv:2304.05128*, 2023.
- [3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” *arXiv preprint arXiv:2310.06770*, 2023.
- [4] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, “Explainable automated debugging via large language model-driven scientific debugging,” *Empirical Software Engineering*, vol. 30, no. 2, p. 45, 2025.
- [5] H. Lin, M. Maas, M. Roquemoire, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbükten, F. Papa *et al.*, “Eco: An llm-driven efficient code optimizer for warehouse scale computers,” *arXiv preprint arXiv:2503.15669*, 2025.
- [6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [7] K. Cheng, X. Shen, Y. Yang, T. Wang, Y. Cao, M. A. Ali, H. Wang, L. Hu, and D. Wang, “Codemenv: Benchmarking large language models on code migration,” *arXiv preprint arXiv:2506.00894*, 2025.
- [8] A. Grishina, V. Liventsev, A. Härmä, and L. Moonen, “Fully autonomous programming using iterative multi-agent debugging with large language models,” *ACM Transactions on Evolutionary Learning*, vol. 5, no. 1, pp. 1–37, 2025.
- [9] J. T. Liang, C. Yang, and B. A. Myers, “A large-scale survey on the usability of ai programming assistants: Successes and challenges,” in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.
- [10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.
- [11] K. Keahey, M. Richardson, R. Tolosana Calasanz, S. Hunold, J. Lofstead, T. Malik, and C. Perez, “Report on challenges of practical reproducibility for systems and hpc computer science,” Apr. 2025. [Online]. Available: <https://doi.org/10.5281/zenodo.15306610>
- [12] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi *et al.*, “Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning,” *arXiv preprint arXiv:2501.12948*, 2025.
- [13] A. Liu, A. Ho, A. M. Droste, D. Martin, E. Wong, E. Zhou, I. Zhou, J. Park, J. Jiao, K.-R. Skelly *et al.*, “Lifescibench: Evaluating language models on realistic, expert-level tasks in the life sciences,” *OpenAI preprint*, 2026.
- [14] M. Seo, J. Baek, S. Lee, and S. J. Hwang, “Paper2code: Automating code generation from scientific papers in machine learning,” in *International Conference on Learning Representations*, vol. 2026, 2026, pp. 38 867–38 932.
- [15] G. Starace, O. Jaffe, D. Sherburn, J. Aung, J. S. Chan, L. Maksin, R. Dias, E. Mays, B. Kinsella, W. Thompson *et al.*, “Paperbench: evaluating ai’s ability to replicate ai research,” *arXiv preprint arXiv:2504.01848*, 2025.
- [16] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu *et al.*, “A survey on llm-as-a-judge,” *The Innovation*, vol. 7, no. 6, 2026.
- [17] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Collieran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, “Lessons learned from the Chameleon testbed,” in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC ’20)*. USENIX Association, Jul. 2020.
- [18] M. Rahman, S. Khatoonabadi, A. Abdellatif, H. Samaana, and E. Shihab, “The impact of environment configurations on the stability of ai-enabled systems,” *arXiv preprint arXiv:2408.02825*, 2024.
- [19] M. Nazario, R. Bonifacio, and G. Pinto, “Mitigating configuration differences between development and production environments: A catalog of strategies,” *arXiv preprint arXiv:2505.09392*, 2025.
- [20] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” *arXiv preprint arXiv:2210.03629*, 2022.
- [21] K. Keahey, J. Anderson, M. Powers, and A. Cooper, “Three pillars of practical reproducibility,” in *Proceedings of the Reproducible Workflows, Data Management, and Security (ReWorDS 23) at eScience’23*, Oct. 2023.
- [22] D. Zan, Z. Huang, W. Liu, H. Chen, L. Zhang, S. Xin, L. Chen, Q. Liu, X. Zhong, A. Li *et al.*, “Multi-swe-bench: A multilingual benchmark for issue resolving,” *arXiv preprint arXiv:2504.02605*, 2025.
- [23] chameleon, “A collection of shared artifacts - an open platform for practical reproducibility.” 2021. [Online]. Available: <https://trovi.chameleoncloud.org/dashboard/>
- [24] TensorBlock, “Forge: Ai model provider middleware,” 2025, accessed: 2025-09-12. [Online]. Available: <https://github.com/TensorBlock/forg>

- [25] K. Mei, X. Zhu, W. Xu, W. Hua, M. Jin, Z. Li, S. Xu, R. Ye, Y. Ge, and Y. Zhang, "Aios: Llm agent operating system," *arXiv preprint arXiv:2403.16971*, 2024.
- [26] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–31, 2025.
- [27] M. Shetty, N. Jain, J. Liu, V. Kethanaboyina, K. Sen, and I. Stoica, "Gso: Challenging software optimization tasks for evaluating swe-agents," 2025. [Online]. Available: <https://arxiv.org/abs/2505.23671>
- [28] D. Pomian, A. Bellur, M. Dilhara, Z. Kurbatova, E. Bogomolov, A. Sokolov, T. Bryksin, and D. Dig, "Em-assist: Safe automated extractmethod refactoring with llms," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 582–586.
- [29] L. Gioacchini, M. Mellia, I. Drago, A. Delsanto, G. Siracusano, and R. Bifulco, "Autopenbench: Benchmarking generative agents for penetration testing," *arXiv preprint arXiv:2410.03225*, 2024.
- [30] I. David and A. Gervais, "Multi-agent penetration testing ai for the web," *arXiv preprint arXiv:2508.20816*, 2025.
- [31] P. Jing, M. Tang, X. Shi, X. Zheng, S. Nie, S. Wu, Y. Yang, and X. Luo, "Secbench: A comprehensive multi-dimensional benchmarking dataset for llms in cybersecurity," *arXiv preprint arXiv:2412.20787*, 2024.
- [32] H. Lee, Z. Zhang, H. Lu, and L. Zhang, "Sec-bench: Automated benchmarking of llm agents on real-world software security tasks," *arXiv preprint arXiv:2506.11791*, 2025.
- [33] A. K. Zhang, N. Perry, R. Dulepet, J. Ji, C. Menders, J. W. Lin, E. Jones, G. Hussein, S. Liu, D. Jasper *et al.*, "Cybench: A framework for evaluating cybersecurity capabilities and risks of language models," *arXiv preprint arXiv:2408.08926*, 2024.
- [34] J. Guo, C. Wang, X. Xu, Z. Su, and X. Zhang, "Repoaudit: An autonomous llm-agent for repository-level code auditing," *arXiv preprint arXiv:2501.18160*, 2025.
- [35] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "Cweval: Outcome-driven evaluation on functionality and security of llm code generation," in *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 33–40.
- [36] czguo cs, "Automatically install repos with swe-agent," 2025. [Online]. Available: <https://github.com/SWE-bench/SWE-smith/issues/115>
- [37] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.
- [38] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 323–333.
- [39] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "Reprozip: Computational reproducibility with ease," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 2085–2088.
- [40] P. J. Guo, "Cde: Automatically package and reproduce computational experiments," in *Implementing Reproducible Research*. Chapman and Hall/CRC, 2018, pp. 79–111.
- [41] A. Kumar, Y. Bajpai, S. Gulwani, G. Soares, and E. Murphy-Hill, "Sharp tools: How developers wield agentic ai in real software engineering tasks," *arXiv e-prints*, pp. arXiv–2506, 2025.
- [42] Z. Chen, W. Ma, and L. Jiang, "Unveiling pitfalls: Understanding why ai-driven code agents fail at github issue resolution," *arXiv preprint arXiv:2503.12374*, 2025.
- [43] M. Tufano, A. Agarwal, J. Jang, R. Z. Moghaddam, and N. Sundaresan, "Autodev: Automated ai-driven development," *arXiv preprint arXiv:2403.08299*, 2024.
- [44] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "Openhands: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.
- [45] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, "Metagtpt: Meta programming for a multi-agent collaborative framework." International Conference on Learning Representations, ICLR, 2024.
- [46] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong *et al.*, "Chatdev: Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023.
- [47] W. Li, A. Jan, B. Ray, J. Yang, C. Mao, and K. Pei, "Editlord: Learning code transformation rules for code editing," *arXiv preprint arXiv:2504.15284*, 2025.
- [48] S. Chen, Y. Wang, Y.-F. Wu, Q. Chen, Z. Xu, W. Luo, K. Zhang, and L. Zhang, "Advancing tool-augmented large language models: Integrating insights from errors in inference trees," *Advances in Neural Information Processing Systems*, vol. 37, pp. 106 555–106 581, 2024.
- [49] Anthropic, "Claude code," 2025. [Online]. Available: <https://github.com/anthropics/claude-code>
- [50] OpenAI, "Openai codex cli," May 2025. [Online]. Available: <https://github.com/openai/codex>
- [51] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

APPENDIX A

PREDICTED VS. PROVISIONED HARDWARE RESOURCES

Table V reports, for a representative subset of repositories, the hardware requirements specified by repository analysis together with the corresponding requirements inferred by the baseline agents (Codex and Claude) and the resources ultimately reserved by EnvAgent+ on bare-metal (TACC) and KVM platforms. These per-repository results complement the aggregate provisioning results discussed in Section III-C.

TABLE V
SPECIFIED AND PROVISIONED HARDWARE RESOURCES BY REPOSITORY
AND RESOURCE CATEGORY.

Repository	Category	Specified	Codex	Claude	Bare metal (TACC)	KVM actual
tao_tutorials	CPU	8	4	8	24	8
	RAM (GB)	16	16	32	512	16
	Disk (GB)	100	50	100	6400	160
	GPU	Yes	Yes	Yes	4 x NVIDIA MI100	not provisioned
	GPU memory (GB)	16	8	16	4	not provisioned
Gym	CPU	unknown	4	16	48	4
	RAM (GB)	8	16	64	384	8
	Disk (GB)	5	20	100	6400	80
	GPU	No	Yes	Yes	N/A	N/A
	GPU memory (GB)	unknown	8	40	N/A	N/A
ai-avatar	CPU	unknown	4	4	48	2
	RAM (GB)	unknown	8	8	384	4
	Disk (GB)	unknown	10	20	6400	40
	GPU	No	Yes	Yes	N/A	N/A
	GPU memory (GB)	unknown	4	8	N/A	N/A
Kilosort4	CPU	4	4	8	16	4
	RAM (GB)	8	16	32	192	8
	Disk (GB)	unknown	10	50	4000	80
	GPU	Yes	Yes	Yes	2 x NVIDIA K80	not provisioned
	GPU memory (GB)	8	8	8	2	not provisioned
CLM-GS	CPU	8	4	16	20	2
	RAM (GB)	128	16	64	256	4
	Disk (GB)	500	50	100	4000	40
	GPU	Yes	Yes	Yes	2 x NVIDIA M40	not provisioned
	GPU memory (GB)	24	8	24	2	not provisioned
CellViT-plus-plus	CPU	16	4	16	20	16
	RAM (GB)	32	16	64	256	32
	Disk (GB)	30	20	100	4000	320
	GPU	Yes	Yes	Yes	2 x NVIDIA M40	not provisioned
	GPU memory (GB)	24	8	24	2	not provisioned
CellViT-Inference	CPU	16	4	8	28	16
	RAM (GB)	32	8	32	256	32
	Disk (GB)	30	10	50	4000	320
	GPU	Yes	Yes	Yes	2 x NVIDIA P100	not provisioned
	GPU memory (GB)	24	8	16	2	not provisioned
plonky2-gpu	CPU	8	4	16	24	8
	RAM (GB)	16	8	64	512	16
	Disk (GB)	unknown	10	50	6400	160
	GPU	Yes	Yes	Yes	4 x NVIDIA MI100	not provisioned
	GPU memory (GB)	12	4	16	4	not provisioned
svrtk-docker-gpu	CPU	6	4	8	48	8
	RAM (GB)	16	8	32	384	16
	Disk (GB)	30	20	50	6400	160
	GPU	Yes	Yes	Yes	N/A	not provisioned
	GPU memory (GB)	12	4	8	N/A	not provisioned
node0	CPU	unknown	4	4	48	16
	RAM (GB)	8	8	8	384	32
	Disk (GB)	80	20	50	6400	320
	GPU	Yes	No	No	N/A	not provisioned
	GPU memory (GB)	16	0	0	N/A	not provisioned
SciDOCX	CPU	8	2	4	48	8
	RAM (GB)	16	4	16	384	16
	Disk (GB)	10	10	20	6400	160
	GPU	Yes	No	No	N/A	not provisioned
	GPU memory (GB)	6	0	0	N/A	not provisioned
U-Time	CPU	4	4	8	48	4
	RAM (GB)	8	8	32	384	8
	Disk (GB)	4096	10	100	6400	80
	GPU	Yes	Yes	Yes	N/A	not provisioned
	GPU memory (GB)	unknown	4	8	N/A	not provisioned
bootstrap	CPU	unknown	2	8	48	2
	RAM (GB)	unknown	4	16	384	4
	Disk (GB)	unknown	10	50	6400	40
	GPU	No	No	Yes	N/A	not provisioned
	GPU memory (GB)	unknown	0	8	N/A	not provisioned
pdf-to-podcast	CPU	8	4	8	48	2
	RAM (GB)	64	16	32	384	4
	Disk (GB)	100	20	50	6400	40
	GPU	Yes	Yes	Yes	N/A	not provisioned
	GPU memory (GB)	unknown	8	24	N/A	not provisioned

Notes. *Specified* denotes the hardware requirements inferred by EnvAgent+ through repository analysis. *Unknown* indicates that no reliable value could be determined from the repository analysis. *Codex* and *Claude* report hardware requirements inferred independently by the baseline agents and do not perform resource discovery or provisioning. *Bare metal (TACC)* and *KVM actual* report the hardware reserved by EnvAgent+, which is selected based on inferred requirements and available platform resources. Because TACC and KVM allocate fixed hardware configurations, the reserved resources may exceed the specified requirements. *N/A* indicates that the resource is not applicable, whereas *Not provisioned* indicates that no compatible resource was provisioned during the evaluation. These cases do not affect the aggregate provisioning results reported in Section III-C.

DATA AVAILABILITY

The ENVAGENT framework, ENVBENCH evaluation suite, and our preprocessed dataset compilation are open source and publicly available at github.com/EaminC/EnvAgent. Our experiments use publicly available artifacts from the Chameleon Trovi Reproducible Research Collection and the Multi-SWE-bench Cross-Language Collection.