ResearchEnvBench: Benchmarking Agents on Environment Synthesis for Research Code Execution

Yubang Wang[2,4,5,6*], **Chenxi Zhang**[4,5], **Bowen Chen**[4,5,7*], **Zezheng Huai**[2,5,8], **Zihao Dai**[2,4,5], **Xinchi Chen**[1,3,4,5†], **Yuxin Wang**[4,5], **Yining Zheng**[1,2,5], **Jingjing Gong**[2,5] and **Xipeng Qiu**[1,2,3,4,5†]
[1]Institute of Trustworthy Embodied AI, Fudan University
[2]Shanghai Innovation Institute
[3]Shanghai Key Laboratory of Multimodal Embodied AI
[4]College of Computer Science and Artificial Intelligence, Fudan University
[5]OpenMOSS
[6]Wuhan University
[7]Nanjing University
[8]Jilin University

Abstract

Autonomous agents are increasingly expected to support scientific research, and recent benchmarks report progress in code repair and autonomous experimentation. However, these evaluations typically assume a pre-configured execution environment, which requires resolving complex software dependencies, aligning hardware and framework versions, and configuring distributed execution, yet this capability remains largely unbenchmarked. We introduce ResearchEnvBench, a benchmark for environment synthesis in research code execution. Given a research repository, documentation, and a target execution setting, agents must construct an environment that successfully executes at runtime. Evaluations on diverse research repositories reveal a substantial gap in current SOTA agents, with failures dominated by incomplete dependency resolution and brittle version coupling. ResearchEnvBench provides a realistic testbed for advancing autonomous agents toward reproducible scientific research.

1 Introduction

Recent work has studied autonomous software engineering and scientific discovery. Agents have demonstrated remarkable proficiency in complex tasks, ranging from repository-level bug fixing [Jimenez *et al.*, 2024; Yang *et al.*, 2024] to autonomous machine learning experimentation [Chan *et al.*, 2025; Edwards *et al.*, 2025]. However, these evaluations predominantly operate within a crucial abstraction: they assume the existence of a functional, pre-configured execution environment. In practical research scenarios, particularly within Deep Learning (DL) and High-Performance Computing (HPC), this assumption rarely holds. Resolving the dependency problem of Python libraries, aligning CUDA drivers with framework versions, and configuring distributed communication primitives, remains a notorious bottleneck (Figure 1). Without the capability to autonomously bridge this pre-execution barrier, an agent's ability to modify code or propose experiments remains theoretically promising but practically unverifiable.

However, benchmarking this environment set up capability remains fundamentally unresolved, particularly for research-grade machine learning codebases. Configuring these environments requires resolving system-level dependencies, such as aligning CUDA drivers with PyTorch versions, compiling custom C++ extensions, and configuring distributed communication primitives. Existing benchmarks fail to capture this runtime complexity. For instance, EnvBench [Eliseeva *et al.*, 2025] evaluates setup success primarily through static analysis (checking for missing imports), a metric that cannot detect runtime failures caused by binary incompatibilities or hardware mismatches. Similarly, while Multi-Docker-Eval [Fu *et al.*, 2025] and Repo2Run [Hu *et al.*, 2025] assess container build success rates, they do not enforce the rigorous runtime verification required to ensure that the constructed environment can actually execute computations on hardware accelerators. This evaluation gap leaves us without a reliable measure of an agent's ability to truly reproduce scientific experiments "in the wild".

To bridge this gap, we introduce ResearchEnvBench, a benchmark designed to evaluate the runtime reproducibility of deep learning research environments. Unlike prior datasets that focus on general software packages, we curate a set of 44 high-quality research repositories created after January 1, 2024. These repositories are selected for their complexity, involving heavy hardware dependencies, custom CUDA kernels, and distributed training requirements. To evaluate agent performance, we propose the pyramid of runtime verification, a hierarchical evaluation pipeline that moves beyond simple installation checks. Our protocol strictly enforces a sequence

*Visiting students at Fudan University.
†Corresponding author.

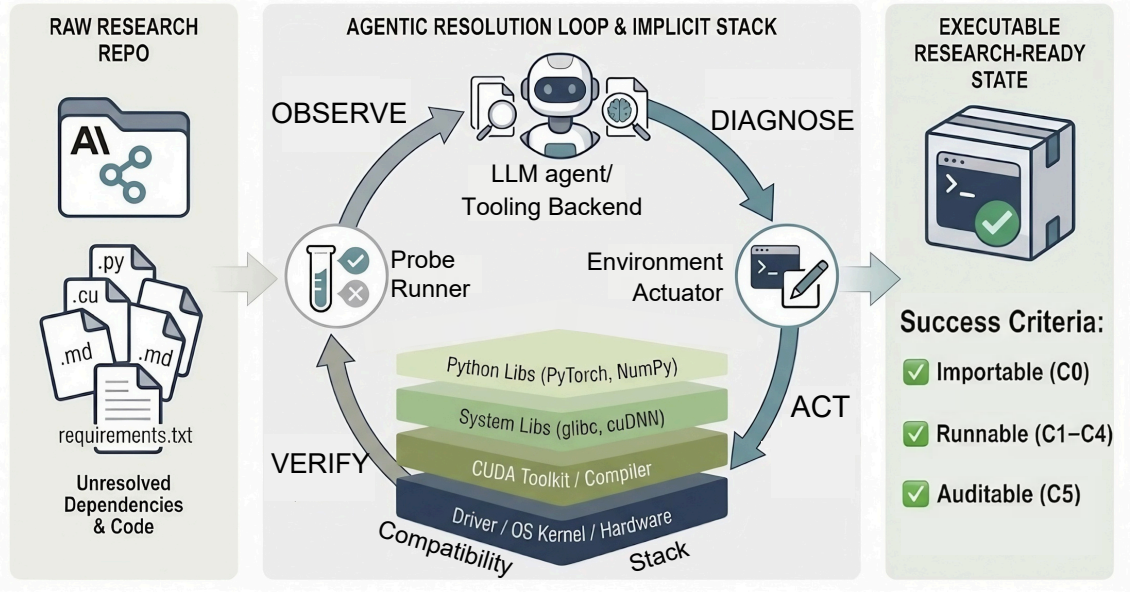

Figure 1: **Agentic closed-loop resolution of research environments under implicit constraints.** From a raw repository, an LLM-driven agent observes execution signals, diagnoses dependency and compatibility failures, edits the environment, and re-runs probes until it reaches a research-ready state. The dependency stack highlights downward compatibility constraints (Python → system → CUDA → driver/hardware) and upward failures in logs/tracebacks. Evaluation reports importability (C_0), runtime CPU/GPU checks (C_1–C_4), and auditable reports and logs (C_5).

of validation levels: from static dependency integrity, to hardware alignment, and finally to distributed readiness. Furthermore, we introduce metrics to quantify capability hallucination, measuring the discrepancy between an agent's self-reported success and the ground-truth runtime status.

In summary, our contributions are as follows:

1. **A Hardened Benchmark for Research Environment Synthesis:** We release **ResearchEnvBench**, a curated dataset of 44 high-complexity research repositories created after January 1, 2024. Unlike previous benchmarks that filter for general Python projects, our dataset specifically targets repositories with intricate hardware dependencies, custom CUDA kernels, and distributed training requirements, reflecting the challenges of real-world scientific research code.
2. **The Pyramid of Runtime Verification:** We propose a rigorous evaluation protocol that moves beyond static analysis. We formalize environment setup as a hierarchical capability ladder, requiring agents to pass a sequence of checks ranging from dependency integrity to multi-GPU distributed data parallel (DDP) execution. Additionally, we introduce a capability hallucination metric to quantify the discrepancy between an agent's self-reported success and ground-truth executability.
3. **Benchmarking SOTA Agents:** We conduct a comprehensive evaluation of five state-of-the-art agent settings on ResearchEnvBench and find a steep drop from hardware alignment to true entrypoint execution (best multi-GPU validation success rate is 37.5%), alongside large differences in self-report reliability (where conservative `null` reporting can reduce hallucination false positives compared to optimistic `ok` claims).

2 Related Work

Recent advances have propelled LLM-based agents from function-level code generation [Chen *et al.*, 2021] to repository-scale software engineering [Jimenez *et al.*, 2024; Yang *et al.*, 2024] and autonomous scientific discovery [Yamada *et al.*, 2025; Mitchener *et al.*, 2025; Ghareeb *et al.*, 2025]. While agents demonstrate growing proficiency in modifying code logic and iterating on experiments, a critical precursor remains under-evaluated: the ability to autonomously bootstrap the execution environment itself. ResearchEnvBench targets this gap, evaluating whether agents can establish reproducible runtime states for complex research repositories without human intervention.

2.1 Software Maintenance & Evolution

Benchmarks in this category evaluate an agent's ability to modify code logic to fix bugs or add features. SWE-bench [Jimenez *et al.*, 2024] established the standard for issue resolution, recently extended by SWE-Smith [Yang *et al.*, 2025] and R2E-Gym [Jain *et al.*, 2025] which use synthetic data to scale evaluation. Multi-SWE-bench [Zan *et al.*, 2025] and SWE-bench-multilingual [Yang *et al.*, 2025] further expand this to multilingual settings. However, these benchmarks typically operate within pre-built Docker containers where dependencies are already satisfied. They incentivize

modifying source code to pass unit tests. In contrast, ResearchEnvBench operates under a no-modification constraint on tracked files, but allows adding auxiliary files, assessing the agent's ability to construct a research-ready environment.

2.2 Agents for Scientific Discovery

Automating the machine learning research pipeline is a surging area. MLE-bench [Chan *et al.*, 2025] evaluates agents on 75 Kaggle competitions, focusing on model performance metrics. RExBench [Edwards *et al.*, 2025] and SciCode [Tian *et al.*, 2024] assess agents on implementing novel research extensions or solving scientific problems. While these benchmarks evaluate the research capability, they often bypass the "MLOps" Challenge by providing simplified environments. ResearchEnvBench complements these works by isolating the environment bootstrap phase. We verify whether agents can handle the specific dependency problem of modern AI research (e.g., FlashAttention compilation, NCCL linking) before any experiment can begin.

2.3 Automated Environment Setup

The most relevant recent works focus explicitly on environment configuration. EnvBench [Eliseeva *et al.*, 2025] provides a large-scale evaluation for Python and JVM repositories but relies on static analysis (missing imports check) rather than execution. SetupBench [Arora *et al.*, 2025] evaluates system administration tasks like database setup, but focuses on general DevOps rather than Deep Learning stacks. Multi-Docker-Eval [Fu *et al.*, 2025] and Repo2Run [Hu *et al.*, 2025] assess Docker build success rates across multiple languages. Our work distinguishes itself through its Pyramid of Runtime Verification. Unlike Multi-Docker-Eval, which targets build success, or SetupBench, which targets general services, we explicitly verify GPU-executable runtime correctness. We require agents to prove CUDA Alignment and Distributed Readiness, capabilities that are critical for AI research but absent in general software engineering benchmarks.

3 ResearchEnvBench

3.1 Task Formulation

We formalize ResearchEnvBench as an interactive, multi-turn stage transition problem. The objective is to evaluate an agent's ability to autonomously transform a raw, unconfigured environment into a "Research-Readiness" stage capable of executing complex ML experiments.

Problem Definition

Let ϵ_0 denote the initial environment state (a bare-metal Docker container with CUDA drivers but no repository-specific dependencies) and R denote the target research repository. The task is a Markov Decision Process (MDP) where the agent interacts with the environment over T time steps to produce a final state ϵ_T.

- **The Agent Interface (Action Space A):** Unlike prior environment-focused benchmarks that restrict agents to static setup scripts such as Installamatic [Milliken *et al.*, 2024] or shell-only interaction such as SetupBench [Arora *et al.*, 2025], ResearchEnvBench provides a compound toolset that combines shell execution with precise file editing, consistent with modern software engineering agents such as SWE-agent [Yang *et al.*, 2024]. At each step t, the agent selects an action $a_t \in A$, which includes:
  - **Shell Interaction:** Executing shell commands to install dependencies and run build steps.
  - **File Navigation & Reading:** Navigating the repository and reading dependency manifests such as `requirements.txt`.
  - **Code Editing:** Editing files to create auxiliary setup scripts but not modifying tracked source code.
- **State Transition:** The environment transitions from state ϵ_t to ϵ_{t+1} based on the agent's action a_t. The agent receives an observation o_{t+1} consisting of command outputs and file contents, which serves as feedback for the next step.
- **Goal State:** The goal is to reach a converged state ϵ_{final} that satisfies the requirement of research implement. Specifically, ϵ_{final} must be a directly operable Docker image where the repository's training and inference entry points can be executed successfully once the required datasets are available (e.g., downloaded or mounted) without further manual dependency installation or environment configuration.

Verification Protocol

Upon task completion, we freeze the state ϵ_{final} and inject a suite of hidden verification probes. We evaluate each environment using the **ResearchEnvBench Pyramid of Validation**, where C_0 is an auxiliary static check and C_1–C_4 are runtime validations:

1. **Static Integrity (C_0):** Missing imports reported by `pyright` (`reportMissingImports`), reported as missing/total imports.
2. **Runtime Integrity (C_1):** The environment supports CPU execution of the model's entry point.
3. **Hardware Alignment (C_2):** The environment correctly maps framework binaries such as PyTorch to the underlying NVIDIA drivers.
4. **Single-GPU Computation (C_3):** The environment supports actual computation on a single GPU.
5. **Distributed Readiness (C_4):** (For supported repos) The environment is configured to support multi-GPU distributed data parallel (DDP) execution, a stringent requirement for modern AI research.
6. **Hallucination (C_5):** Quantifies discrepancies between the agent's self-report and the ground-truth outcomes from hidden probes (paths, versions, and claimed capabilities).

3.2 Dataset Construction

ResearchEnvBench targets the long-tail of complex, hardware-sensitive AI research code. We constructed our dataset through a rigorous pipeline that filters for research integrity, hardware awareness, and reproducibility feasibility (Table 1).

Sourcing and Automated Triage

We initiated our collection from GitHub, targeting Python repositories created after **January 1, 2024**. This temporal constraint ensures our benchmark evaluates agents against modern library ecosystems such as PyTorch 2.x, avoiding stale dependency trees common in older datasets.

From an initial pool of candidates, we employed a custom static analysis tool to programmatically filter repositories based on three dimensions of evidence:

1. **Research Integrity Signals:** We filtered for repositories containing explicit research artifacts in their `README.md`. The triage script searches for research-oriented keywords such as "arXiv" to verify the repository is intended for AI scientific contribution rather than general utility.
2. **Hardware & Distributed Semantics:** A unique feature of ResearchEnvBench is the mandatory requirement for hardware acceleration. The triage script scans the file tree for signatures of high-performance computing:
   - **GPU Dependency:** Presence of GPU-specific keywords such as `torch.cuda`.
   - **Distributed Training:** Detection of distributed training primitives.
   - **Compilation Complexity:** Identification of CUDA source files, indicating custom operator compilation.
3. **Hard Constraints:** To ensure quality, we enforced strict thresholds: at least 100 GitHub stars, no archived projects, and non-forked repositories.

Final Dataset Composition

From the candidates, 44 repositories passed all automated checks and manual verification for execution feasibility. While smaller in raw count compared to static analysis benchmarks, this scale is consistent with state-of-the-art execution-based benchmarks such as Multi-Docker-Eval [Fu *et al.*, 2025] (40 repos) and SetupBench [Arora *et al.*, 2025] (54 repos).

Crucially, ResearchEnvBench prioritizes validation depth over breadth. Nearly all selected repositories in our final set (43/44) support at least single-GPU execution, and a substantial subset additionally supports multi-GPU distributed data parallel (DDP) execution. The final dataset spans eight categories of modern ML research codebases (Figure 2), with proportions computed by repository count. Overall, it covers both model-centric workloads (e.g., generative vision, depth estimation, audio/speech) and system-centric stacks (e.g., LLM inference/acceleration and training infrastructure), which jointly stress dependency closure, CUDA alignment, and distributed readiness.

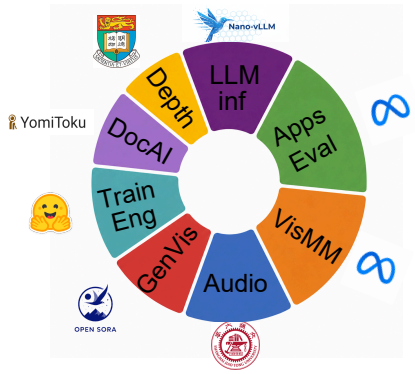

Figure 2: **Repository composition of ResearchEnvBench.** We curate 44 post-2024 ML research repositories spanning eight categories: GenVis (generative vision), Depth (depth estimation), Audio (audio & speech), LLM-Inf (LLM inference & acceleration), TrainEng (training & engineering frameworks), VisMM (vision/multimodal foundations), DocAI (document AI / OCR / translation), and AppsEval (applications & evaluation). Slice sizes denote the fraction of repositories per category by count, and each slice shows a representative flagship repository from that category.

This diversity ensures agents are tested against a wide spectrum of dependency graphs, ranging from standard PyTorch ecosystems to complex environments requiring the compilation of custom kernels and distributed communication primitives.

3.3 Evaluation Metrics & Pipeline

To rigorously assess the "Research Readiness" of the synthesized environments, we design a hierarchical runtime verification pipeline and a novel metric to quantify agent hallucinations.

The Pyramid of Runtime Verification

Unlike prior works that rely on static build success signals such as Docker exit codes [Fu *et al.*, 2025], ResearchEnvBench enforces a strict, multi-stage protocol visualized in Figure 3. The execution is orchestrated by a serial protocol (`run_all.sh`) that evaluates a hierarchy of environment capabilities. We verify correctness across C_0–C_4 (increasing runtime strictness) and additionally measure C_5 (capability hallucination), which captures false success claims.

- C_0**: Static Dependency Integrity.** Following EnvBench [Eliseeva *et al.*, 2025], we run `pyright` and count missing-import diagnostics (`reportMissingImports`), reported as missing imports over total imports. We treat this as an auxiliary proxy for dependency closure; overall readiness is determined by the runtime stages (C_1–C_4).
- C_1**: Minimal CPU Execution.** Following environment synthesis, we require the model's entry point (training or inference) to execute on CPU under the smallest feasible configuration. This validates that the resolved dependency graph is internally consistent, independent of hardware acceleration.
- C_2**: Hardware Alignment.** We verify that the installed deep learning framework matches the underlying driver

Benchmark	Repos	Langs	Domain	HW Aware	Evaluation
Installamatic [Milliken *et al.*, 2024]	40	Py	Gen. SWE	✗	Build success
EnvBench [Eliseeva *et al.*, 2025]	994	Py, JVM	Gen. SWE	✗	Missing imports
SUPER [Bogin *et al.*, 2024]	45	Py	ML	✗	CPU execution
Multi-Docker-Eval [Fu *et al.*, 2025]	40	9 langs	Gen. SWE	✗	Build + tests
SetupBench [Arora *et al.*, 2025]	54	7 langs	DevOps	✗	Functional checks
ResearchEnvBench (Ours)	**44**	**Py, C++**	**AI/HPC**	✓	**Runtime pyramid (C_0–C_4)**

Table 1: Comparison of ResearchEnvBench with state-of-the-art environment setup and execution benchmarks. Prior benchmarks often target general software engineering (Gen. SWE) or DevOps (development/operations) and evaluate via build success, missing-import checks, unit tests (e.g., `pytest`), or functional checks; ResearchEnvBench focuses on AI/HPC research and uses a hardware-aware runtime pyramid (C_0–C_4).

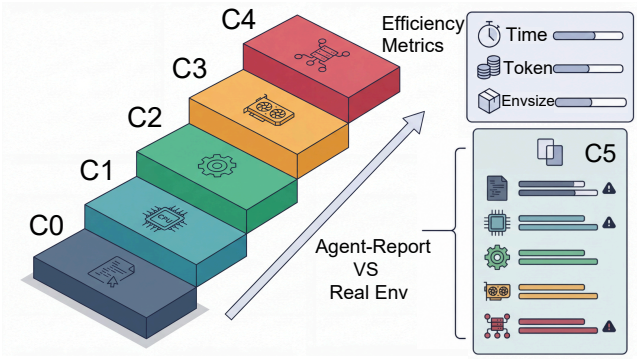

Figure 3: **The Pyramid of Runtime Verification.** We structure environment synthesis into a hierarchy of increasing complexity, ranging from static dependency resolution (C_0) to multi-GPU distributed training (C_4). The right panel illustrates our auxiliary metrics: *Efficiency* (Time, Token, Size) and *Capability Hallucination* (C_5), which quantifies the discrepancy between the agent's self-reported success (Report) and the ground-truth execution (Real Result).

version. The environment must pass a cuda-check probe, ensuring it can access the GPU device.

- C_3**: Single-GPU Computation.** The environment must support actual kernel execution on a GPU. This stage catches common version mismatches (e.g., `torch` compiled with an incompatible CUDA version) that only surface during tensor allocation.

- C_4**: Distributed Readiness.** For repositories supporting distributed training, we enforce a multi-GPU check using standard launchers. This verifies the correct configuration of distributed communication primitives like NCCL, representing the highest standard of research reproducibility.

Applicability and Scoring

Not all repositories admit every capability. For repositories without a runnable CPU-only entry point, we treat C_1 as skipped; for repositories without multi-GPU distributed support, we treat C_4 as skipped. Skipped capabilities are excluded from the denominator when computing aggregate success rates.

Quantifying Capability Hallucination (C_5)

Agents often exhibit overconfidence, claiming tasks are complete when underlying configurations are broken. We introduce **Capability Hallucination** to quantify the discrepancy between the agent's self-report and the observed reality.

The agent is required to generate a structured report summarizing the environment state after synthesis. We classify hallucination into three categories:

1. **Path Hallucination:** The reported python path does not exist or is not executable.
2. **Version Hallucination:** The reported library versions like torch_version differ from the actual versions detected by our hidden probes.
3. **Capability Hallucination:** A critical mismatch where the agent claims a capability is enabled but the corresponding runtime verification (e.g., C_2 or C_4) fails. We record a hallucination whenever the agent claims the check after environment setup passes but the probe fails. This metric directly measures how trustworthy an agent's completion claims are.

Efficiency Metrics

Beyond correctness, we measure the cost of environment synthesis:

- **Time-to-Ready:** Total wall-clock time from task initiation to agent execution end.
- **Token Cost:** Total number of tokens consumed during the interaction.
- **Environment Size:** Total size of the constructed environment, encouraging lightweight dependency management.

4 Experiment

We conduct a comprehensive evaluation of state-of-the-art LLM agents on ResearchEnvBench to assess their ability to bootstrap complex ML research environments from scratch.

4.1 Experimental Setup

Evaluation Environment

To ensure reproducibility and prevent environment contamination, each evaluation session is instantiated in a strictly isolated Docker container. The environment is initialized as

	C_0	C_1	C_2	C_3	C_4	C_5 Hallucinations ↓			
Agent	**missing/total ↓**	**CPU SR**	**CUDA SR**	**1-GPU SR**	**DDP SR**	**path**	**version**	**cap.**	**total**
Codex (GPT-5.1-Codex)	675/2858 (23.6%)	17/29 (58.6%)	35/44 (79.5%)	19/43 (44.2%)	11/32 (34.4%)	0	0	4	4
Codex (GPT-5.3-Codex)	641/2858 (22.4%)	16/29 (55.2%)	26/44 (59.1%)	14/43 (32.6%)	9/32 (28.1%)	0	2	18	20
Claude Code Agent (GLM-4.7)	761/2858 (26.6%)	16/29 (55.2%)	40/44 (90.9%)	21/43 (48.8%)	12/32 (37.5%)	0	1	17	18
Claude Code Agent (Sonnet 4.5)	728/2858 (25.5%)	15/29 (51.7%)	41/44 (93.2%)	18/43 (41.9%)	12/32 (37.5%)	0	0	20	20
NexAU (DeepSeek-V3.1-Nex-N1)	722/2858 (25.3%)	16/29 (55.2%)	41/44 (93.2%)	21/43 (48.8%)	11/32 (34.4%)	1	1	14	16

Table 2: Main results on ResearchEnvBench. C_0 reports missing imports over total imports after synthesis; C_1–C_4 report stage success rates (success/total on applicable repositories). C_5 counts hallucinations by type (Path/Version/Capability) and totals them (↓ is better).

a clean sandbox based on Ubuntu 22.04, equipped with 2× NVIDIA RTX 4090 GPUs (24GB VRAM each) and CUDA 12.4 drivers (Version 550.163.01). Crucially, the sandbox contains no pre-installed deep learning frameworks (e.g., PyTorch, TensorFlow) or build tools (e.g., CUDA Toolkit), testing the ability of agents to install these from scratch.

Agent Execution Loop

We utilize a standardized Host Orchestrator to manage the agent lifecycle. For each repository, the orchestrator initializes a clean Docker container and clones the target repository, then provides the agent with a standard interface supporting shell execution, file reading, and file editing. The agent is constrained by a maximum budget of time. Upon the execution completion signal (or upon timeout), the orchestrator immediately revokes access and executes the pyramid verification Pipeline (C0→C5) described in Section 3 via run_all.sh.

4.2 Baseline Agents

We evaluate five agent settings across four agent families. All agents receive the same "Senior MLOps Engineer" instruction focused on reproducibility and verifiability.

1. **Claude Code Agent (GLM-4.7):** Claude Code tooling [Anthropic, 2026] with the open-source GLM-4.7 model, which targets agentic coding and tool use [Zhipu AI, 2025].
2. **Claude Code Agent (Sonnet 4.5):** The same Claude Code tooling [Anthropic, 2026] with Claude Sonnet 4.5, evaluated under identical interfaces and budgets.
3. **Codex Agent (GPT-5.1-Codex and GPT-5.3-Codex):** Two Codex-style settings share the same agent interface and tool access. GPT-5.1-Codex serves as a code-specialized baseline [Chen *et al.*, 2021]; GPT-5.3-Codex tests whether a newer code-specialized model improves runtime readiness beyond static dependency closure.
4. **NexAU Agent (DeepSeek-V3.1-Nex-N1):** The NexAU framework [Team *et al.*, 2025] with the open-source DeepSeek-V3.1-Nex-N1 model, using iterative plan-execute-verify loops over build and runtime tools.

4.3 Main Results

Table 2 summarizes missing-import statistics (C_0), stage success rates (C_1–C_4), and self-report hallucinations (C_5).

Stage-wise executability. No single agent dominates the runtime pyramid. Codex (GPT-5.1-Codex) attains the best CPU execution rate (C_1, 58.6%), NexAU and Claude (Sonnet 4.5) tie on CUDA alignment (C_2, 93.2%), and the Claude variants lead DDP readiness (C_4, 37.5%). GPT-5.3-Codex has the lowest missing-import rate but the weakest runtime readiness (C_2 59.1%, C_3 32.6%, C_4 28.1%), showing that static closure does not guarantee accelerator builds, ABI alignment, or entrypoint execution.

Hallucination behavior. C_5 exposes self-report reliability gaps. GPT-5.1-Codex produces 4 hallucinations, while GPT-5.3-Codex, Claude Code (GLM-4.7), Claude Code (Sonnet 4.5), and NexAU produce 20, 18, 20, and 16, mostly capability hallucinations. GPT-5.3-Codex often reports capabilities such as `ddp_expected_ok=true` after generic CUDA/NCCL checks, while hidden repository-level multi-GPU probes fail.

4.4 Ablation Study: Reasoning Effort

To isolate inference-time reasoning budget, we rerun GPT-5.3-Codex with the same Codex agent interface under *low*, *medium*, and *xhigh* settings. Table 3 reports static closure, runtime readiness, hallucination, and resource metrics.

Higher reasoning effort improves runtime readiness: *xhigh* gives the best C_1–C_4 results, especially for CUDA and single-/multi-GPU execution, but also costs the most setup time, disk space, and C_5 hallucinations. *Medium* gives the best static closure and lowest C_5, while *low* is cheapest but loses CPU and GPU entrypoint readiness. Reasoning budget helps, but static closure, self-report calibration, runtime executability, and resource cost do not improve monotonically.

5 Analysis

5.1 Failure Mode Analysis: The Gap Between Existence and Readiness

We repeatedly observe an *existence–readiness gap*: agents produce plausible environments under shallow checks, yet native training/inference entrypoints still fail. The blockers are repository-specific assumptions—native extensions, auxiliary toolchains, and mixed-framework stacks—that manifests underspecify and real execution reveals.

The funnel of failure. Tab 2 shows the stage-wise drop across C_1–C_4. CPU execution (C_1) succeeds on only 51.7–58.6% of applicable repositories. CUDA alignment (C_2)

Reasoning	C_0 missing/total ↓	C_1 CPU SR	C_2 CUDA SR	C_3 1-GPU SR	C_4 DDP SR	C_5 total ↓	Avg. Setup Time (min)	Avg. Env Size (GB)
Low	651/2858 (22.8%)	9/29 (31.0%)	19/44 (43.2%)	7/43 (16.3%)	5/32 (15.6%)	10	13.9	4.73
Medium	529/2858 (18.5%)	14/29 (48.3%)	18/44 (40.9%)	8/43 (18.6%)	5/32 (15.6%)	9	17.1	5.24
Xhigh	641/2858 (22.4%)	16/29 (55.2%)	26/44 (59.1%)	14/43 (32.6%)	9/32 (28.1%)	20	19.1	6.29

Table 3: **GPT-5.3-Codex reasoning-effort ablation.** All rows use the same Codex agent interface and ResearchEnvBench protocol, varying only the model reasoning-effort setting. C_1–C_4 report success/total on applicable repositories.

reaches 93.2% for some settings, but success falls at single-GPU execution (C_3, 32.6–48.8%) and remains low for distributed execution (C_4, 28.1–37.5%). GPU visibility is therefore necessary but insufficient.

A counterexample from GPT-5.3-Codex. GPT-5.3-Codex illustrates the limits of static setup metrics: $C_0 = 0/90$ on *LMCache/LMCache* still fails because `requests` is absent; *microsoft/renderformer* passes CPU/CUDA checks but misses `cuda.h` and `torchrun`; *yuanze-lin/Olympus* fails distributed execution through an invalid DeepSpeed launcher. Static closure still misses toolchain, launcher, and entrypoint requirements.

Dependency anatomy. Missing-module traces follow three recurring categories:

- **Native ops.** CUDA/C++ operators must be built or ABI-matched, e.g., `mmcv._ext` in *sapiens*, `flash_attn` in *nano-vllm*, and `lmcache.c_ops` in *LMCache*.
- **Auxiliary tooling.** Utilities such as `wandb` and `tensorboard` are hard dependencies on default training paths.
- **Framework mix.** Single-framework assumptions miss `jax`, `tensorflow`, and similar dependencies.

Overall, agents optimize for *generic setup* rather than *path-aware readiness*; this defines the gap between "it installs" and "it runs".

5.2 Capability Hallucination

Agents are also unreliable reporters. Comparing each `report.json` against hidden probes reveals recurring overconfidence (Table 2): agents infer readiness from clean logs rather than verified execution.

Path & version mismatches. Even static facts drift: *Path Hallucination* reports nonexistent executables (e.g., `/usr/bin/python`), and *Version Hallucination* reports assumed package versions (e.g., `torch 2.1.0+cu121`).

Runtime false positives. The most damaging case is *Capability Hallucination* (C_5): agents claim CUDA or DDP readiness without matching smoke tests. In *nano-vllm*, the agent reported inference readiness, but the first probe failed on a missing `flash_attn` kernel.

5.3 Efficiency and Resource Cost

Beyond success rates, agent benchmarks must account for efficiency. Table 4 reports per-repository averages.

Setup time and disk footprint are non-monotonic. GPT-5.3-Codex spends the most setup time (19.1 min) but obtains the lowest C_4 rate (28.1%), while smaller environments do not reliably imply better runtime readiness.

Agent	Setup	Env.	C_4 SR
Codex-5.1	6.8	6.06	34.4 (11/32)
Codex-5.3	19.1	6.29	28.1 (9/32)
GLM-4.7	9.8	6.65	37.5 (12/32)
Sonnet-4.5	13.3	6.88	37.5 (12/32)
DeepSeek-V3.1-Nex-N1	9.2	6.55	34.4 (11/32)

Table 4: Resource Consumption Comparison. Agent labels abbreviate the five settings in Table 2. Setup is wall time in minutes, Env. is environment size in GB, and C_4 SR excludes inapplicable repositories.

Disk footprint proxies how targeted an agent's dependency choices are: broad installs rarely fix ABI-sensitive operators and can increase version skew.

5.4 Case Study

We analyze *facebookresearch/sapiens*, where a CUDA-visible stack still fails on an implicit native-operator dependency. Both agents pass CUDA alignment (C_2) but fail when the training path imports `mmcv.ops` and cannot load `mmcv._ext` (`ModuleNotFoundError`), blocking C_1 and thus C_3/C_4. The root cause is build-bound: shallow checks can import `mmcv`, but the real path requires compiled CUDA/C++ operators matched to the target PyTorch/CUDA stack.

The failure is not due to insufficient effort. GPT-5.1-Codex spends 12.4 minutes and produces an 11.67 GB environment; NexAU spends 10.3 minutes and produces an 11.59 GB environment. Both fail on the same operator, showing that plausible DL stacks and broad installs are insufficient. Robust synthesis needs path-aware smoke tests and build-aware dependency resolution for native extensions.

6 Conclusion

ResearchEnvBench evaluates environment synthesis for AI/HPC repositories under hardware constraints. Its verification pyramid spans dependency checks, CPU/CUDA probes, single-GPU and distributed execution, and C_5 report audits, exposing the gap between "GPU-visible" stacks and repository-level executability. Across 44 repositories, failures stem from native operators, launcher assumptions, package/ABI coupling, and unsupported readiness claims. ResearchEnvBench shows that agents must tie setup actions to auditable evidence for runnable research code.

Acknowledgements

This work is in part supported by the New Generation Artificial Intelligence-National Science and Technology Major Project (2025ZD0123502) and the National Natural Science Foundation of China (No. 62521004).

Contribution Statement

Yubang Wang and Chenxi Zhang contributed equally to this work.

References

[Anthropic, 2026] Anthropic. Claude Code overview. Technical documentation. https://docs.anthropic.com/en/docs/claude-code/overview, 2026. Accessed: 2026-01-20.

[Arora *et al.*, 2025] Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. SetupBench: Assessing software engineering agents' ability to bootstrap development environments. arXiv preprint arXiv:2507.09063, 2025.

[Bogin *et al.*, 2024] Ben Bogin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. SUPER: Evaluating agents on setting up and executing tasks from research repositories. arXiv preprint arXiv:2409.07440, 2024.

[Chan *et al.*, 2025] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating machine learning agents on machine learning engineering. arXiv preprint arXiv:2410.07095, 2025.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[Edwards *et al.*, 2025] Nicholas Edwards, Yukyung Lee, Yujun Audrey Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. RExBench: Can coding agents autonomously implement AI research extensions? arXiv preprint arXiv:2506.22598, 2025.

[Eliseeva *et al.*, 2025] Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. EnvBench: A benchmark for automated environment setup. arXiv preprint arXiv:2503.14443, 2025.

[Fu *et al.*, 2025] Kelin Fu, Tianyu Liu, Zeyu Shang, Yingwei Ma, Jian Yang, Jiaheng Liu, and Kaigui Bian. Multi-Docker-Eval: A 'shovel of the gold rush' benchmark on automatic environment building for software engineering. arXiv preprint arXiv:2512.06915, 2025.

[Ghareeb *et al.*, 2025] Ali Essam Ghareeb, Benjamin Chang, Ludovico Mitchener, Angela Yiu, Caralyn J. Szostkiewicz, Jon M. Laurent, Muhammed T. Razzak, Andrew D. White, Michaela M. Hinks, and Samuel G. Rodriques. Robin: A multi-agent system for automating scientific discovery. arXiv preprint arXiv:2505.13400, 2025.

[Hu *et al.*, 2025] Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2Run: Automated building executable environment for code repository at scale. arXiv preprint arXiv:2502.13681, 2025.

[Jain *et al.*, 2025] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. arXiv preprint arXiv:2504.07164, 2025.

[Jimenez *et al.*, 2024] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? arXiv preprint arXiv:2310.06770, 2024.

[Milliken *et al.*, 2024] Louis Milliken, Sungmin Kang, and Shin Yoo. Beyond pip install: Evaluating LLM agents for the automated installation of Python projects. arXiv preprint arXiv:2412.06294, 2024.

[Mitchener *et al.*, 2025] Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C. Landsness, Daniel L. Barabasi, Siddharth Narayanan, Nicky Evans, Shriya Reddy, Martha Foiani, Aizad Kamal, Leah P. Shriver, Fang Cao, Asmamaw T. Wassie, Jon M. Laurent, Edwin Melville-Green, Mayk Caldas, Albert Bou, Kaleigh F. Roberts, Sladjana Zagorac, Timothy C. Orr, Miranda E. Orr, Kevin J. Zwezdaryk, Ali E. Ghareeb, Laurie McCoy, Bruna Gomes, Euan A. Ashley, Karen E. Duff, Tonio Buonassisi, Tom Rainforth, Randall J. Bateman, Michael Skarlinski, Samuel G. Rodriques, Michaela M. Hinks, and Andrew D. White. Kosmos: An AI Scientist for autonomous discovery. arXiv preprint arXiv:2511.02824, 2025.

[Team *et al.*, 2025] AGI Team, Yuxuan Cai, Lu Chen, Qiaoling Chen, Yuyang Ding, Liwen Fan, Wenjie Fu, Yufei Gao, Honglin Guo, Pinxue Guo, Zhenhua Han, Zhengfu He, Hanglei Hu, Kai Hu, Shengjia Hua, Tianyu Huai, Baodai Huang, Li Ji, Zhen Jiang, Zhikai Lei, Bufan Li, Jiahang Lin, Lizhi Lin, Jinxiu Liu, Shichun Liu, Ziming Liu, Yuchen Ni, Pengfang Qian, Yujiong Shen, Qingyun Shi, Wentao Shu, Peng Sun, Yiran Suo, Tian Tang, Boyu

Tian, Guoteng Wang, Junzhe Wang, Peixin Wang, Zhiheng Xi, Hang Yan, Jie Yang, Zhixiong Yang, Tianchu Yao, Guangze Ye, Qianxi Yu, Shuo Zhang, Xinyue Zhang, Yiqi Zhang, Jiarong Zhao, Miao Zheng, Rui Zheng, Enyu Zhou, Jiazheng Zhou, Maosen Zhou, Yuhao Zhou, Tao Gui, Yining Zheng, Xinchi Chen, Jie Zhou, Siyuan Feng, Qin Chen, Liang He, Qi Zhang, Xuanjing Huang, and Xipeng Qiu. Nex-N1: Agentic models trained via a unified ecosystem for large-scale environment construction. arXiv preprint arXiv:2512.04987, 2025.

[Tian *et al.*, 2024] Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. SciCode: A research coding benchmark curated by scientists. arXiv preprint arXiv:2407.13168, 2024.

[Yamada *et al.*, 2025] Yutaro Yamada, Robert Tjarko Lange, Cong Lu, Shengran Hu, Chris Lu, Jakob Foerster, Jeff Clune, and David Ha. The AI Scientist-v2: Workshop-level automated scientific discovery via agentic tree search. arXiv preprint arXiv:2504.08066, 2025.

[Yang *et al.*, 2024] 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. arXiv preprint arXiv:2405.15793, 2024.

[Yang *et al.*, 2025] John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. arXiv preprint arXiv:2504.21798, 2025.

[Zan *et al.*, 2025] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. arXiv preprint arXiv:2504.02605, 2025.

[Zhipu AI, 2025] Zhipu AI. GLM-4.7. Model documentation. https://docs.bigmodel.cn/cn/guide/models/text/glm-4.7, 2025. Accessed: 2026-01-20.