BootstrapAgent: Distilling Repository Setup into Reusable Agent Knowledge

Sihan Fu[1,*] **Oucheng Liu**[2,*] **Shiyuan Wang**[3,*] **Jin Shi**[1] **Chengkun Wei**[1,†]
[1]Zhejiang University [2]The Australian National University
[3]Institute of Information Engineering, Chinese Academy of Sciences
{fusihan,shijin,weichengkun}@zju.edu.cn
oucheng.liu@anu.edu.au wangshiyuan@iie.ac.cn
[*]Equal contribution. [†]Corresponding author.

Abstract

Code agents increasingly help developers work with unfamiliar repositories. They must first set up a working development environment for each repository. This setup requires costly trial and error. Agents learn how to resolve dependencies and repair environment issues along the way. Yet this knowledge remains confined to a single conversation, leaving future agents unable to reuse it. To make this knowledge reusable, we introduce **BootstrapAgent**, a multi-agent framework for repository bootstrapping. It distills the heuristics discovered during exploration into a persistent, verifiable **.bootstrap** contract. Future agents can consume this contract to guide repository setup. BootstrapAgent extracts repository evidence to build a structured plan for setup, diagnostic checks, and minimal verification. It then generates the contract, verifies it in Docker, and uses execution traces to guide repairs. The contract also preserves the repair knowledge gained during this process. We further propose *warm repair with clean replay* to accelerate iterative debugging without sacrificing cold-start reproducibility, and a *delta repair with sanity check* to prevent reward hacking. Experiments on three benchmarks show that BootstrapAgent achieves a 90.7% overall mean success rate on 212 repositories, while reducing downstream agent token usage by 20.3% and build time by 28.0%. Our code is available at `https://github.com/Vossera/BootstrapAgent`.

1 Introduction

Real-world development rarely starts from scratch but builds on existing repositories [3, 9]. Before starting a development task, developers need to set up a development environment for the repository. However, the setup process for a freshly cloned repository is often unclear [20]. Existing repository artifacts provide useful clues, but these clues are typically incomplete [16, 22, 25]: README files may describe package usage rather than the development setup; CI (Continuous Integration) workflows may depend on secrets, caches, service containers, specialized hardware, or long-running matrix jobs that are difficult to reproduce locally; and package metadata rarely specifies the minimal command that can confirm a local checkout is actually usable.

Code agents increasingly help developers work with unfamiliar codebases. However, setting up development environments remains a costly bottleneck for these agents [24, 36, 38, 40, 42]. As shown in Figure 1, a typical agent must read the README, inspect package metadata and CI workflows, infer setup commands, execute them, observe failures, repair the environment, and retry [4, 12, 34]. This loop consumes substantial tokens and time. More critically, the knowledge gained during this process cannot be reused. Resolved dependencies, working-directory corrections, and successful repair strategies remain trapped in a single-round conversation, unavailable to the next

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

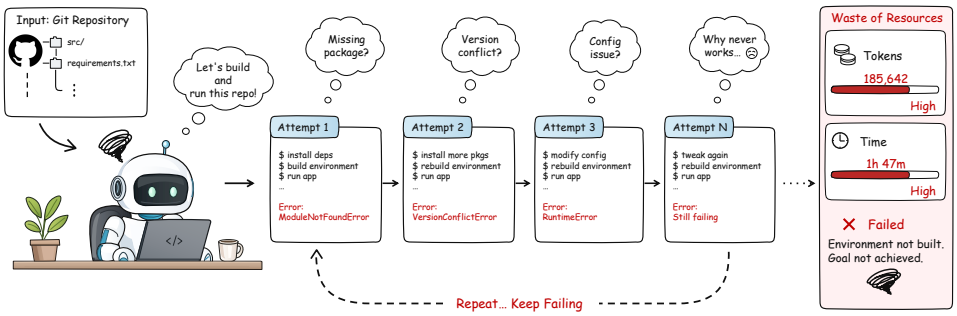


Figure 1: The process of deploying a repo by Code Agent.

agent or developer working on the same repository. Multiple agents may therefore repeatedly pay the same bootstrap cost. Recent agent-facing files such as `AGENTS.md` [10] and `CLAUDE.md` [2] reduce downstream development token cost by encoding repository-specific coding guidance, but they do not address the earlier question of how an agent should reliably setup a repository itself.

Our key observation is that the outcome of bootstrap exploration should not vanish with the conversation, but should instead be persisted as reusable startup knowledge. To this end, we propose **BootstrapAgent**, a multi-agent framework that distills bootstrap exploration into a reusable `.bootstrap` contract. BootstrapAgent begins with a discovery agent that scans repository structure, README files, CI workflows, lockfiles, package metadata, and build configuration files to extract bootstrap-relevant evidence. A planning agent then converts this evidence into a structured bootstrap plan containing setup commands, diagnostic checks, and verification commands. Based on this plan, a contract generator agent produces a `.bootstrap` contract. The contract is then replayed stage by stage by a deterministic verifier in a clean Docker [35] environment. If replay fails, the generator agent uses structured execution traces to apply feedback-driven minimal revisions to `.bootstrap`, and verification is re-run. This repair-and-verify loop continues until the `.bootstrap` contract passes clean replay or a stopping condition is reached. The pipeline is built around two key design choices: *warm repair with clean replay* allows fast iterative repair within the same container to reduce cost, but requires the final contract to pass cold-start verification in a fresh environment to ensure reproducibility; and a *delta repair with sanity checks* constrains each repair round to evidence-supported updates over the existing file, rather than unconstrained regeneration. This prevents agents from achieving superficial success by weakening the verification command.

We evaluate BootstrapAgent on 212 repositories drawn from three benchmarks and compare it against other environment setup approaches and agent-only bootstrap baselines. Across five independent runs on 212 repositories from three benchmarks, BootstrapAgent achieves an overall mean clean-replay success rate of 90.7%. It exceeds the strongest applicable baseline on each benchmark by 6.0–9.5 percentage points. When the generated contracts are reused by downstream Claude Code, token usage is reduced by 20.3% and build time by 28.0%. In summary, our contributions are as follows:

- We formulate repository bootstrapping as a reusable startup knowledge problem for code agents, and introduce `.bootstrap` as an agent-consumable contract.
- We design BootstrapAgent framework that combines evidence extraction, structured planning, and deterministic verification with trace-driven repair. Warm repair, clean replay, and delta repair with sanity checks make it efficient, reproducible, and robust to reward hacking.
- We evaluate on 212 repositories and demonstrate strong bootstrap success rates, showing that the generated contracts significantly reduce repeated setup cost for downstream agents.

2 Related Work

2.1 Automated Repository Bootstrapping and Environment Setup.

Prior work has studied repository execution from several angles, ranging from template- or rule-based Dockerfile generation [15, 43] to learning-based environment construction [30]. More recent systems move beyond one-time generation by using execution feedback, sandbox testing, and repository

context to infer dependencies and validation commands. RepoST [39] and Repo2Run [16] construct executable repository environments through context mining and sandboxed verification; Treefix [32] repairs execution through prefix-tree search; and CXXCrafter [44] targets the specialized build and dependency challenges of C/C++ repositories. Complementary benchmarks such as DI-BENCH [45], CSR-Bench [37], R2E [17], and R2E-Gym [18] further expose dependency inference, scientific software deployment, and executable-repository construction as important evaluation problems. These works show that repository setup is a difficult and measurable task. However, they primarily treat environment construction as a one-time deployment or benchmark outcome. BootstrapAgent instead treats setup exploration as reusable agent knowledge: it persists the setup process into a verifiable `.bootstrap` contract that can be cleanly replayed and reused by future agents.

2.2 Agent Context Files and Repository-Facing Documentation

Coding agents [1, 11, 28] have demonstrated strong abilities on coding evaluations [5, 7, 8, 14, 19, 23, 27, 29, 41]. Many studies have shown that context plays an important role in improving agent performance [6, 13, 21, 26, 31]. Files such as `AGENTS.md` [10] and `CLAUDE.md` [2] guide and support agents during development. They provide the repository-specific context agents need to perform development tasks. However, prior work has not studied how repository-facing context files can help environment setup, including dependency installation and recovery from setup failures.

3 BootstrapAgent

3.1 Problem Formulation

We study how to distill repository setup knowledge into a reusable startup contract. Let R denote a repository specified by a URL or a local path. Let P_R denote the process of setting up a working development environment for R. This process includes inspecting repository evidence, installing dependencies, diagnosing failures, applying repairs, and verifying the environment. The bootstrap configuration system must produce a contract C from P_R. C is a repository-local artifact:

$$C = (I, D, M, S, H),$$

where I is an ordered sequence of setup commands, D is a sequence of read-only diagnostic checks, M is a mandatory minimal verification command, S is an optional strongest locally reproducible verification command derived from CI or build evidence, and H is the compressed repair knowledge preserved for future agents. The contract exposes a simple execution protocol: run setup, run diagnostics, and then run verification.

To determine whether C can be reused, we evaluate it through *clean replay*. The setup commands, diagnostic checks, and minimal verification are executed sequentially in a fresh container with a fixed base configuration. Let Y_I, Y_D, and Y_M denote the outcomes of these three stages. We define contract validity as

$$\mathrm{Valid}(R, C) = \mathbf{1}[Y_I = \mathrm{PASS} \wedge Y_D = \mathrm{PASS} \wedge Y_M = \mathrm{PASS}]\,.$$

The optional command S provides an additional validation target but is not required to pass.

Once validated, C guides a fresh agent through environment setup. Let $\mathrm{Cost}_{\mathrm{reuse}}(R, C)$ denote the cost incurred by that agent when using C to reach the verified setup state. We measure this cost through token usage and wall-clock time. Our goal is to produce valid contracts for as many repositories as possible while reducing setup costs for downstream agents.

3.2 System Overview

Given the above formulation, BootstrapAgent produces a verified `.bootstrap` contract that specifies how to set up, inspect, and minimally verify a repository in a clean environment. As shown in Figure 2, this process proceeds through five stages: repository discovery, bootstrap plan generation, contract generation, containerized verification, and trace-driven repair. We describe each stage below.

3.3 Repository Discovery

The first stage inspects repository files to identify environment setup requirements and candidate verification commands. A *Discoverer Agent* explicitly scans files that are likely to affect bootstrapping,

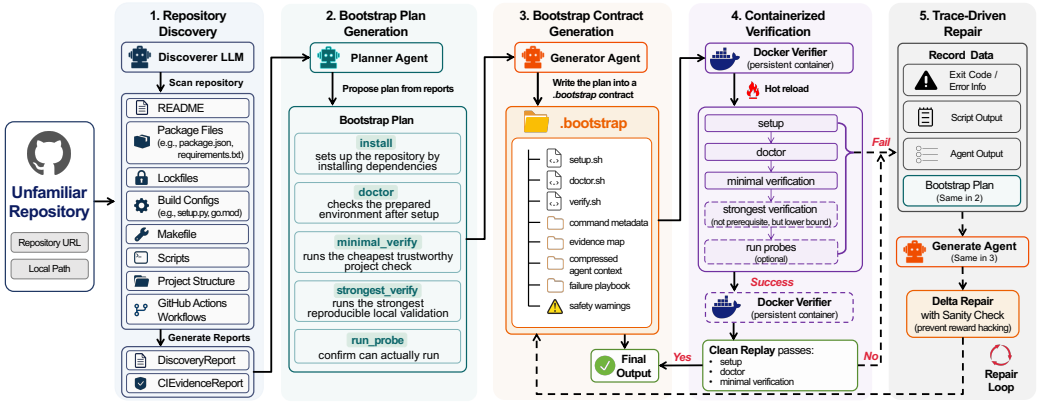


Figure 2: Overview of BootstrapAgent.

including `README` files, package metadata, lockfiles, build configuration files, Makefiles, scripts, project layout, and GitHub Actions workflows.

This stage produces two structured reports. `DiscoveryReport` records detected languages, package managers, important files, repository structure, and evidence snippets from relevant files. `CIEvidenceReport` records workflow files and local `run` commands that may serve as validation candidates. It also identifies non-local CI features, such as secrets, service containers, cloud services, or heavyweight external dependencies, and marks them as non-reproducible constraints that should not be selected for local verification.

3.4 Bootstrap Plan Generation

Given the discovery and CI-derived reports, the *Planner Agent* produces a structured `BootstrapPlan`. It is a schema-constrained description of the intended bootstrap strategy. It records the expected setup phases, the verification goals, the evidence used to justify each decision, and the constraints that later agents must preserve. The plan acts as a persistent task anchor during repair. Without an explicit plan, it can drift toward fixing the most recent error while forgetting the higher-level bootstrap objective.

3.5 Bootstrap Contract Generation

This stage generates the `.bootstrap` directory, which contains `setup.sh`, `doctor.sh`, `verify.sh`, command metadata, an evidence map, agent context, a failure playbook, and safety warnings.

Initially, no prior execution evidence exists, the *Generator Agent* constructs a complete `.bootstrap` contract only based on the plan. After the initial contract is synthesized, the system enters a repair-and-verify loop. The scripts are executed inside a Docker container, producing an execution trace. The trace is then provided to the agent together with the plan to generate `.bootstrap`. This process repeats until the verification succeeds or the maximum number of repair rounds is reached.

3.6 Containerized Verification

This stage executes the generated contract inside a Docker environment. It is necessary because repository bootstrap correctness cannot be reliably inferred from static evidence alone. The repository is mounted at a fixed path, and the verifier runs setup, doctor, minimal verification, strongest verification, and optional run probes with stage-specific timeouts and logs.

This staged execution has two competing requirements. During repair-and-verify loop, the system needs fast feedback so that small command-level changes can be tested without repeatedly paying the full setup cost. For final acceptance, however, the verifier must rule out accidental success caused by cached packages, generated files, or residual state from previous attempts. To satisfy both requirements, the *Docker Verifier* uses a two-level correctness mechanism.

When in the repair loop, it uses **hot reload** in a Docker container to provide low-latency feedback for small structured plan deltas. This design is inspired by Spring Boot's hot-swapping workflow, where development tools monitor changes and support fast restarts or live reloads instead of repeatedly performing a full cold start [33]. The system then freezes the candidate contract and performs a **clean replay** from a fresh container. If the clean replay succeeds, the system outputs the final `.bootstrap` directory. If the clean replay fails, the system continues the repair loop. Thus, warm verification accelerates the search process, while clean replay preserves the semantic requirement that a bootstrap contract must work from an empty environment.

3.7 Trace-Driven Repair

The *Generator Agent* acts as the repair agent in this stage, but in a constrained repair mode. Instead of freely rewriting the entire contract, it proposes a structured delta over the existing `BootstrapPlan` and the current `.bootstrap` directory. A delta may insert a missing setup command, adjust command ordering, replace a failed command with an evidence-supported alternative, add diagnostic checks, or update timeout and fallback metadata. Commands and metadata that have already survived previous verifications are preserved unless the trace provides direct evidence that they caused the failure.

This delta-based design improves efficiency and stability across repair rounds. Because the agent only edits the failing or under-specified part of the contract, later rounds do not need to regenerate the entire `.bootstrap` directory. This reduces generation cost and limits the opportunity for hallucinated rewrites to degrade previously valid scripts, metadata, or evidence links. In other words, repair is treated as a localized update to an existing contract rather than as a new bootstrap attempt.

To further reduce **reward hacking** during multi-round repair, BootstrapAgent applies a sanity check after each proposed repair. The generator agent that repeatedly observes verifier failures may otherwise weaken the verification command to satisfy the acceptance criterion, for example by replacing a project-relevant build or test with a trivial version check. The sanity check compares the repaired contract against the original plan, the evidence map, and the strongest locally reproducible validation target. If the checks fail, the generator agent will regenerate a patch to .bootstrap.

4 Experiments

We use `DeepSeek-V4-Flash-2026-04-25` for all agentic planning and repair. The prompts used in our agent are shown in A.

4.1 Research Questions

Our evaluation is organized around the following research questions on three benchmarks. Detailed benchmark descriptions are given in Appendix B.

RQ1: Effectiveness. Can BootstrapAgent effectively generate a `.bootstrap` contract that passes clean replay on real repositories?

RQ2: Downstream bootstrap transfer. Does a generated `.bootstrap` contract reduce the cost for a fresh downstream agent to reach a verified startup state?

RQ3: Ablation. Which design choices contribute to BootstrapAgent's performance?

4.2 Effectiveness

Bootstrap success rate: We measure the bootstrap success rate in a fresh container, where success requires `setup`, `doctor`, and `minimal_verify` to pass. Detailed experimental settings are provided in Appendix C.1. Table 1 compares BootstrapAgent with three environment-construction baselines: ExecutionAgent [4], Repo2Run [16], and HerAgent [22], reporting the mean success rate and standard deviation over five independent runs on each benchmark. Repo2Run supports only Python and is therefore not applicable to the multi-language ExecutionAgent-50 benchmark. BootstrapAgent achieves the highest mean success rate on all three benchmarks: 81.2% on ExecutionAgent-50, 91.5% on Installamatic-40, and 94.3% on Repo2Run-122. These results exceed the strongest applicable baseline on each benchmark by 6.0, 9.5, and 7.1 percentage points, respectively, supporting the effectiveness of BootstrapAgent across both Python-only and multi-language repository collections.

Table 1: Effectiveness across benchmarks. Success rates (%) are reported as mean ± standard deviation over five independent runs.

Benchmark	ExecutionAgent	Repo2Run	HerAgent	BootstrapAgent
ExecutionAgent-50	66.8 ± 4.60	N/A	75.2 ± 4.38	$\mathbf{81.2 \pm 1.10}$
Installamatic-40	72.0 ± 3.71	82.0 ± 2.09	79.0 ± 3.79	$\mathbf{91.5 \pm 1.37}$
Repo2Run-122	83.1 ± 1.24	87.2 ± 1.49	81.8 ± 1.07	$\mathbf{94.3 \pm 1.30}$

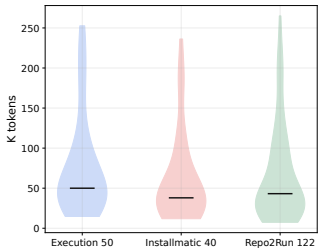


Figure 3: Token usage of BootstrapAgent.

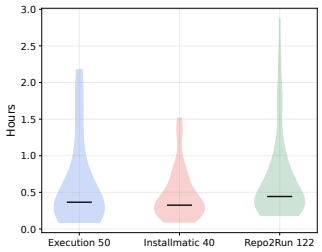


Figure 4: Time cost of BootstrapAgent.

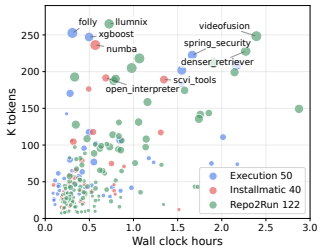


Figure 5: Relationship between token usage and time cost.

Token and time cost: We further analyze the token and time cost of generating '.bootstrap' files and validating them with minimal verification. Figure 3 shows the distribution of total token usage per project, Figure 4 reports the corresponding time cost and Figure 5 plots the relationship between token and time consumption.

Across all 212 benchmark runs, BootstrapAgent uses a median of 43.8K tokens and 24.6 minutes per repository; the 90th percentile reaches 169.2K tokens and 79.0 minutes. The reported token usage is acceptable. Although BootstrapAgent introduces additional generation and validation overhead, this cost is largely paid once. Downstream agents can then reuse this artifact. This benefit is further amortized as the same repository is reused by more agents or users. The time cost includes model calls, network communication, package downloads, and minimal verification. Given that this time is comparable to the effort required for manual repository setup while producing a reusable build specification, we consider the overhead acceptable.

4.3 Downstream bootstrap transfer

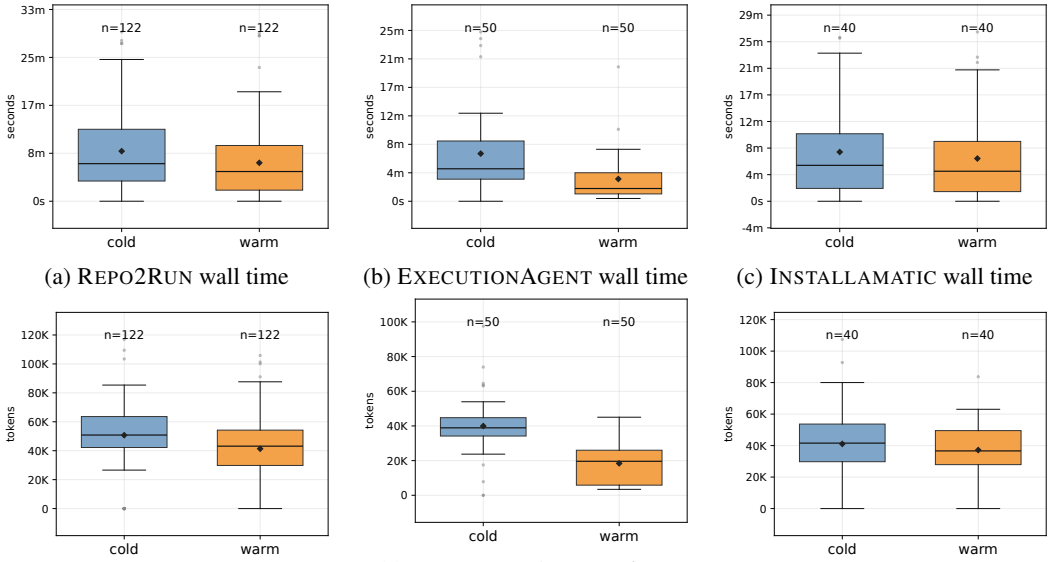
Overall result: We measure the impact of `.bootstrap` on downstream agents by comparing the cost of building projects with (warm) and without (cold) `.bootstrap`. We report two metrics: time and token usage. The detailed experiment setting is shown in Appendix C.2. As shown in Table 2, warm-starting consistently reduces both time cost and token usage across all three benchmarks.

Quantitative analysis: Warm-starting is an effective resource optimization for repository setup agents. Figure 6 further quantifies these savings: median wall-time ratios range from 0.39× to 0.84×, and median total-token ratios range from 0.19× to 0.66×. The largest gain appears on EXECUTIONAGENT, where median wall time drops from 285.2s to 112.0s (0.39×), active tokens from 38.9K to 19.6K, and total tokens from 597.9K to 115.7K (0.19×). INSTALLAMATIC shows a smaller but consistent improvement, reducing median wall time from 337.8s to 282.3s (0.84×), active tokens from 41.6K to 36.6K, and total tokens from 594.4K to 312.6K. On REPO2RUN, median time cost decreases from 391.9s to 309.8s (0.79×), while median and total tokens drop from 50.8K to 43.1K and from 797.8K to 526.9K, respectively.

The reported warm-run token usage already includes the cost of reading .bootstrap and shows overall savings. However, aggregate savings do not rule out cost increases for repositories that are difficult to set up and have long .bootstrap contracts. To assess this possibility, we used cl100k_base to tokenize all nine top-level text files in each .bootstrap directory across all 212 repositories, excluding virtual environments and other environment directories. The median, mean, P95, and maximum contract lengths are 2,915, 3,234, 5,546, and 6,514 tokens, respectively. Among repositories in the

Table 2: Median execution time and token usage for cold and warm execution.

	Execution time (s)			Token usage (K)		
Benchmark	**Cold**	**Warm**	**Ratio**	**Cold**	**Warm**	**Ratio**
ExecutionAgent	285.2	112.0	0.39×	38.9	19.6	0.50×
Installamatic	337.8	282.3	0.84×	41.6	36.6	0.88×
Repo2Run	391.9	309.8	0.79×	50.8	43.1	0.85×
Overall	346.3	249.5	0.72×	46.3	36.9	0.80×



(a) REPO2RUN wall time (b) EXECUTIONAGENT wall time (c) INSTALLAMATIC wall time

(d) REPO2RUN token usage (e) EXECUTIONAGENT tokens usage (f) INSTALLAMATIC tokens usage

Figure 6: Warm-starting results across benchmarks. The first row reports wall-clock time, and the second row reports active token usage.

longest-contract quartile, the median net reduction in total token usage remains 28%, indicating that savings persist in repositories which have long .bootstrap contracts.

Why does these reduction happen? In cold runs, the agent often has to rediscover the project layout, infer programming language, install dependencies, locate verification commands, and diagnose environment-specific failures. These steps create long tool-use and reasoning trajectories, increasing both time cost and token usage. In warm runs, the .bootstrap contract provides repository-specific dependency requirements, setup commands, working directories, and verification targets. This guidance helps downstream agents avoid repeated exploration and trial and error.

4.4 Component-Targeted Ablation Study

We conduct a component-targeted ablation study to isolate the contribution of mechanisms in BootstrapAgent: repository discovery, verification-guided repair, warm repair, clean replay and sanity check. For repository discovery, we ask the agent to generate the bootstrap plan directly, while keeping the rest of the pipeline unchanged. For verification-guided repair, we disable iterative repair and evaluate only the first generated `.bootstrap` contract. For warm repair, each repair round is executed in a fresh Docker container instead of reusing the previous verified state. For clean replay, we accept a successful warm repair directly without rerunning the final contract from scratch. For sanity checks, we allow the repair agent to modify verification commands without checking the repaired contract against the original plan, evidence map, and strongest locally reproducible validation target. All variants are then evaluated independently in fresh containers using generated `.bootstrap` contract, without further repair. For BootstrapAgent, we use results from one of the five independent runs in the main evaluation as the full-system baseline.

Table 3: Component-targeted ablation of BootstrapAgent. Red numbers denote percentage-point drops relative to the full BootstrapAgent.

Variant	REPO2RUN	EXECUTIONAGENT	INSTALLAMATIC
w/o Repository Discovery	76.2% ↓ 19.7	52.0% ↓ 32.0	70.0% ↓ 25.0
w/o Trace-Driven Repair	26.2% ↓ 69.7	18.0% ↓ 66.0	40.0% ↓ 55.0
w/o Clean Replay	95.1% ↓ 0.8	82.0% ↓ 2.0	90.0% ↓ 5.0
w/o Sanity Check	91.8% ↓ 4.1	76.0% ↓ 8.0	90.0% ↓ 5.0
BootstrapAgent	95.9%	84.0%	95.0%

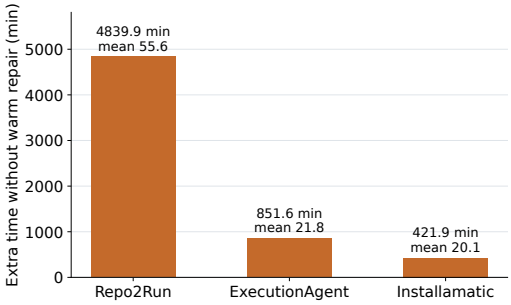


Figure 7: Estimated cost without warm repair.

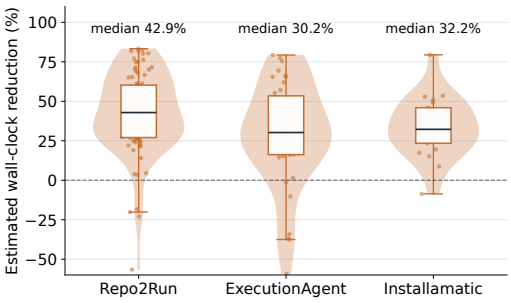


Figure 8: Distribution of warm-repair time reduction.

Bootstrap success ablation study: We analyze the impact of repository discovery and trace-driven repair on bootstrap success rate. As shown in Table 3, trace-driven repair has the largest impact: without iterative repair from execution traces, success drops by 69.7, 66.0, and 55.0 percentage points on the three benchmarks. This confirms that repository bootstrapping is an iterative trial-and-error process. Static repository evidence is often insufficient, and execution traces provide critical feedback for correcting missing dependencies, wrong working directories and incomplete verification scripts.

Repository discovery is also important. Removing the structured discovery report substantially reduces success, indicating that explicitly collecting bootstrap evidence before planning is more reliable than asking the agent to infer all setup information during generation.

Although clean replay and sanity checks have only a small effect on the success rate, they remain important for the reliability of accepted `.bootstrap` artifacts. Warm-only successes may depend on cached packages, generated files, or residual container state and therefore do not establish reproducibility in a clean environment. Sanity checks provide a complementary safeguard by rejecting repairs that weaken or bypass the intended build and test requirements. Manual inspection identified proposed repairs that deleted failing tests (`dynamiq`), replaced a CMake build with `ls/echo` (`imgui_test_engine`), or masked `pnpm test` failures with `|| echo` (`r2r`). The validation floor rejected all three, preventing apparent successes from being accepted as valid, reusable bootstrap contracts.

Bootstrap time ablation study: We further analyze the time-saving effect of *warm repair with clean replay* in Figures 7 and 8. Disabling it would increase total time by 4839.9 minutes on REPO2RUN, 851.6 minutes on EXECUTIONAGENT, and 421.9 minutes on INSTALLAMATIC. The per-repository distribution further shows median wall-clock reductions of 42.9%, 30.2%, and 32.2% on the three benchmarks, respectively. In conclusion, *warm repair with clean replay* makes trace-driven iteration computationally practical by avoiding repeated full setup costs.

Overall, trace-driven repair provides the main success-rate gain, repository discovery improves the quality of the initial plan, and warm repair reduces the cost of repeated verification. Clean replay and sanity checks contribute primarily to trustworthiness: clean replay prevents warm-state artifacts from being mistaken for reproducible contracts, while sanity checks prevent repair drift and verification weakening. These results show that BootstrapAgent relies on both adaptive repair for effectiveness and deterministic safeguards for reproducible, trustworthy bootstrap generation.

Table 4: Two-month retrospective replay of generated contracts.

Benchmark	N	Before	After	Drop
ExecutionAgent	50	42	36	6
Installamatic	40	38	34	4
Repo2Run	122	117	110	7
Overall	212	197	180	17

Table 5: Primary causes of 99 failed runs across five seeds.

Primary cause	Count	Share
Agent planning or repair failure	51	51.5%
Dependency reconstruction failure	39	39.4%
Unavailable external resources	9	9.1%
Total	99	100%

5 Discussion

5.1 Failure Analysis

We manually inspected all 99 failed runs across five seeds and assigned each a primary cause based on its trace (Table 5). Agent-related failures result from hallucinated and reward-hacking attempts. Dependency-reconstruction failures occur when the agent exhausts its repair budget. External-resource failures require unavailable services or downloads.

Agent-related errors account for 51.5% of all failures, indicating that agents are highly susceptible to reward hacking or hallucinations caused by catastrophic forgetting during long-horizon tasks. Dependency reconstruction failures account for 39.4%, and their repair histories show that incorrect early version choices consume a substantial amount of the repair budget. In these cases, the agent could have succeeded, but ambiguous version specifications in the repository led it to make incorrect choices. Therefore, a comprehensive, agent-readable .bootstrap artifact is crucial for reducing these failures. BootstrapAgent addresses this need.

5.2 Limitations

Outdate: Once a .bootstrap file is generated, it becomes part of the repository documentation. Like other documentation, it must be maintained as the project evolves; otherwise, it may become outdated and mislead future agent bootstrapping or manual setup. To assess its continued validity, we conducted a retrospective replay two months after the original experiments on 212 repositories(Table 4). Of the 197 contracts that originally passed verification, 180 (91.4%) still passed at the two-month follow-up, while 17 (8.6%) failed to replay successfully.

These results show substantial continued reusability, while also highlighting that successful initial verification does not guarantee long-term validity. Contracts should therefore be revalidated periodically and updated when repository changes, dependency updates, or runtime changes affect their setup and verification procedures.

Dangerous execution: The verifier establishes reproducibility but cannot establish that executed commands are benign. Since a repository may contain concealed malicious code, users should inspect the generated artifacts before production adoption.

Generality: In the current evaluation, contracts are generated and verified in a controlled Ubuntu 24.04/AMD64 container. Therefore, our reported success rate applies specifically to this target environment. We do not claim that an Ubuntu-generated contract can be executed unchanged on macOS, Alpine Linux, other Linux distributions, or different hardware architectures. Although some repository knowledge—such as dependency structure, command ordering, working directories, and verification targets—may remain useful on another platform, platform-dependent commands and system packages must be adapted and re-verified in the target environment.

5.3 Amortized token cost

Previous experiments show that reusing `.bootstrap` reduces token usage in downstream runs, but these savings do not account for the one-time cost of generating the contract. We further analyze amortized token costs across all 212 cases. Contract generation consumes 102,861 active tokens on average, while cold and warm downstream runs consume an average of 80,021 and 59,215 active tokens, respectively. Warm-run token usage already includes the cost of reading the contract.

When the same contract is reused by k downstream agents, the average token cost per agent is

$$\bar{C}(k) = C_{\text{warm}} + \frac{C_{\text{contract}}}{k}.$$

Based on these mean costs, reuse by two and four agents increases the average cost by 38.3% and 6.1% relative to cold runs, respectively, while reuse by eight agents reduces it by 9.9%. The break-even point is approximately 4.94 reuses, meaning that cumulative token savings cover the initial generation cost from the fifth reuse onward.

6 Conclusion

We introduced **BootstrapAgent**, a multi-agent framework that turns repository bootstrapping from a trial-and-error interaction into reusable startup knowledge. Instead of requiring each future agent to rediscover runtime versions, dependency constraints, working directories, diagnostic checks, and verification commands, BootstrapAgent distills these facts into a persistent and verifiable `.bootstrap` contract. The generated contract helps downstream agents save token and time cost when building environment. We use *warm repair with clean replay* to reduce setup time, and *delta repair with sanity checks* to mitigate hallucinations and reward hacking. Our evaluation on 212 repositories shows that this framework is effective in bootstrapping and reduce cost for downstream agents. Future work can extend this direction with stronger dependency solving and richer support for repositories requiring services, GPUs, or large-scale integration tests.

Acknowledgments and Disclosure of Funding

This research received support from the National Natural Science Foundation of China under Grant No. 62302441. The author gratefully acknowledges the support of Zhejiang University Education Foundation Qizhen Scholar Foundation. The authors declare no competing interests.

References

[1] Anthropic. Claude Code: AI-Powered Coding Assistant for Developers. https://claude.ai/code, 2026. Accessed: 2026-04-23.

[2] Anthropic. How Claude Remembers Your Project. https://code.claude.com/docs/en/memory, 2026.

[3] Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. Setupbench: Assessing software engineering agents' ability to bootstrap development environments, 2025. URL https://arxiv.org/abs/2507.09063.

[4] Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728922. URL https://doi.org/10.1145/3728922.

[5] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022. URL https://arxiv.org/abs/2208.08227.

[6] Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida. Agent readmes: An empirical study of context files for agentic coding, 2025. URL https://arxiv.org/abs/2511.12884.

[7] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. Securevibebench: Evaluating secure coding capabilities of code agents with realistic vulnerability scenarios, 2026. URL https://arxiv.org/abs/2509.22097.

[8] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development, 2026. URL https://arxiv.org/abs/2505.16975.

[9] Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. Envbench: A benchmark for automated environment setup, 2025. URL https://arxiv.org/abs/2503.14443.

[10] Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating agents.md: Are repository-level context files helpful for coding agents?, 2026. URL https://arxiv.org/abs/2602.11988.

[11] Google. Gemini CLI: Build, Debug, and Deploy with AI. https://geminicli.com/, 2026. Accessed: 2026-04-23.

[12] Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks, 2026. URL https://arxiv.org/abs/2506.10954.

[13] Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. On the impacts of contexts on repository-level code generation, 2025. URL https://arxiv.org/abs/2406.11927.

[14] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. Swe-perf: Can language models optimize code performance on real-world repositories?, 2025. URL https://arxiv.org/abs/2507.12415.

[15] Eric Horton and Chris Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets, 2019. URL https://arxiv.org/abs/1905.11127.

[16] Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale, 2025. URL https://arxiv.org/abs/2502.13681.

[17] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2E: Turning any github repository into a programming agent environment. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 21196–21224. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/jain24c.html.

[18] 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, 2025. URL https://arxiv.org/abs/2504.07164.

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

[20] Alexander Kovrigin, Aleksandra Eliseeva, Konstantin Grotov, Egor Bogomolov, and Yaroslav Zharov. Piper: On-device environment setup via online reinforcement learning, 2025. URL https://arxiv.org/abs/2509.25455.

[21] Hao Li, Hicham Masri, Filipe R. Cogo, Abdul Ali Bangash, Bram Adams, and Ahmed E. Hassan. Understanding prompt management in github repositories: A call for best practices. *IEEE Software*, 43(2):85–93, March 2026. ISSN 1937-4194. doi: 10.1109/ms.2025.3644251. URL http://dx.doi.org/10.1109/MS.2025.3644251.

[22] Xiang Li, Siyu Lu, Federica Sarro, Claire Le Goues, and He Ye. Heragent: Rethinking the automated environment deployment via hierarchical test pyramid, 2026. URL https://arxiv.org/abs/2602.07871.

[23] Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

[24] Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories, 2025. URL https://arxiv.org/abs/2511.00197.

[25] Louis Milliken, Sungmin Kang, and Shin Yoo. Beyond pip install: Evaluating llm agents for the automated installation of python projects, 2024. URL https://arxiv.org/abs/2412.06294.

[26] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. Context engineering for ai agents in open-source software, 2026. URL https://arxiv.org/abs/2510.21413.

[27] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 81857–81887. Curran Associates, Inc., 2024. doi: 10.52202/079017-2601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/94f093b41fc2666376fb1f667fe282f3-Paper-Conference.pdf.

[28] OpenAI. Codex: AI Coding Partner from OpenAI. https://openai.com/codex/, 2026. Accessed: 2026-04-23.

[29] Alfin Wijaya Rahardja, Junwei Liu, Weitong Chen, Zhenpeng Chen, and Yiling Lou. Can agents fix agent issues?, 2025. URL https://arxiv.org/abs/2505.20749.

[30] Giovanni Rosa, Antonio Mastropaolo, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. Automatically generating dockerfiles via deep learning: Challenges and promises, 2023. URL https://arxiv.org/abs/2303.15990.

[31] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code, 2023. URL https://arxiv.org/abs/2206.12839.

[32] Beatriz Souza and Michael Pradel. Treefix: Enabling execution with a tree of prefixes, 2025. URL https://arxiv.org/abs/2501.12339.

[33] Spring Boot. Hot Swapping. https://docs.spring.io/spring-boot/how-to/hotswapping.html. Spring Boot Reference Documentation. Accessed: 2026-05-10.

[34] Matúš Sulír, Jaroslav Porubän, and Sergej Chodarev. Local software buildability across java versions (registered report), 2024. URL https://arxiv.org/abs/2408.11544.

[35] Sander Valstar, William G. Griswold, and Leo Porter. Using devcontainers to standardize student development environments: An experience report. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, page 377–383, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368742. doi: 10.1145/3341525.3387424. URL https://doi.org/10.1145/3341525.3387424.

[36] Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. Swe-bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories, 2025. URL https://arxiv.org/abs/2512.17419.

[37] Yijia Xiao, Runhui Wang, Luyang Kong, Davor Golac, and Wei Wang. CSR-bench: Benchmarking LLM agents in deployment of computer science research repositories. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 12705–12723, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.633. URL https://aclanthology.org/2025.naacl-long.633/.

[38] Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. Repost: Scalable repository-level coding environment construction with sandbox testing, 2025. URL https://arxiv.org/abs/2503.07358.

[39] Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. RepoST: Scalable repository-level coding environment construction with sandbox testing. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=2txrMBpw3q.

[40] 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, 2024. URL https://arxiv.org/abs/2405.15793.

[41] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

[42] He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1506–1518, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510222. URL https://doi.org/10.1145/3510003.3510222.

[43] Hongjie Ye, Jiahong Zhou, Wei Chen, Jiaxin Zhu, Guoquan Wu, and Jun Wei. Dockergen: A knowledge graph based approach for software containerization. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 986–991, 2021. doi: 10.1109/COMPSAC51774.2021.00133.

[44] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. Cxxcrafter: An llm-based agent for automated c/c++ open source software building. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025. doi: 10.1145/3729386. URL https://doi.org/10.1145/3729386.

[45] Linghao Zhang, Junhao Wang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Jiaheng Wen, Chengxing Xie, Maoquan Wang, Yufan Huang, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. Di-bench: Benchmarking large language models on dependency inference with testable repositories at scale, 2025. URL https://arxiv.org/abs/2501.13699.

A Key Prompts

This appendix lists the key prompts used by our agentic bootstrap-generation pipeline. The prompts are shown in template form: repository-specific discovery reports, CI evidence, current plans, and verifier traces are inserted at runtime. Together, these prompts cover repository evidence collection, CI-derived validation extraction, structured command planning, verifier-guided repair, two-level verification, and language-specific setup guardrails.

Main Bootstrap Agent System Prompt

```
You are the MainBootstrapAgent.

Coordinate specialized subagents to generate a verified .bootstrap package for
an unfamiliar repository. All machine-consumed outputs must be valid JSON or
YAML matching the provided schemas. Do not modify business source code. The
deterministic verifier is the only authority on success or failure.
```

Evidence Collection Prompts

```
Discovery agent:
Collect repository evidence only. Return DiscoveryReport JSON.

CI evidence agent:
Inspect CI files and return CIEvidenceReport YAML.

Planner agent:
Generate BootstrapPlan JSON from discovery and CI evidence.

Repair agent:
Generate RepairPlan JSON from a failed verifier trace.
```

Command Planner System Prompt

```
You are CommandPlannerAgent. Return only the structured BootstrapPlan.

{COMMAND_CONSTRAINTS}
```

Command Planner User Prompt

```
Generate a BootstrapPlan JSON for this repository. Use only commands that are
    plausible
from the provided evidence. Preserve provenance in source/reason fields. The plan
     must
describe bootstrap commands for the checked-out source. Commands that mutate the
    checkout
or use remote installers are allowed when necessary and will be logged as safety
    warnings.

{COMMAND_CONSTRAINTS}

DiscoveryReport:
{DISCOVERY_REPORT_JSON}

CIEvidenceReport:
{CI_EVIDENCE_REPORT_JSON}
```

Repair Agent System Prompt

```
You are RepairAgent. Return only the structured RepairPlanDelta. Do not call
    tools.

{COMMAND_CONSTRAINTS}
```

Repair Agent User Prompt

```
Generate a RepairPlanDelta JSON from this failed verifier result. Return only the
     smallest
delta needed to repair the current BootstrapPlan; omitted or null fields are
    copied from the
current plan locally. Prefer replace_commands for one-command edits,
    move_commands for reordering,
and insert_commands or remove_commands for list changes. Use replace_doctor or
    replace_install only
when the whole list must change. For doctor/install list edits, use the zero-
    based index values shown
in the compact JSON and never remove an index that is not present. Only change
    setup, verify, doctor
commands, cwd, timeout, agent_context, evidence, or failure_playbook. Checkout
    mutations or remote
installers are allowed when needed for the repository bootstrap and will be
    logged as safety warnings.
Do not call tools or inspect files; use only the supplied JSON. Focus on the
    failed command and the
shortest repair.

Current BootstrapPlan compact JSON:
{CURRENT_BOOTSTRAP_PLAN_JSON}

VerifierResult compact JSON:
{VERIFIER_RESULT_JSON}
```

Core Command Constraints

```
Command constraints:
- Assume the verifier starts from a fresh, minimal Ubuntu container. Do not
    assume project users
  already have language runtimes, compilers, package managers, build tools, or
    test tools installed.
  If setup or verify needs a tool, install or enable it in install commands
    before first use.
- Doctor commands run after setup.sh and should normally be read-only health
    checks of the prepared
  environment. Put environment setup, package installation, dependency
    installation, and build steps
  only in install commands.
- Every command reason must match the actual command string.
- Put validation, import, and test commands only in minimal_verify,
    strongest_verify, or run_probe.
- minimal_verify should be the lowest-cost trustworthy project check and is a
    hard verification gate.
  strongest_verify is advisory and should be the strongest reproducible local CI-
    derived validation.
- Prefer the repository's native development workflow for a fresh source checkout.

- Do not install a published package with the same name as the repository as a
    substitute for checking
  the checked-out source.
```

```
- Do not use a runtime version check, such as python3 --version or node --version
    , as verification for
  a non-empty source repository.
- The verifier runs setup.sh, doctor.sh, and verify.sh sequentially in the same
    Docker container.
  Environment changes from setup.sh are available to doctor.sh and verify.sh.
- Prefer cwd "."; do not use host paths in commands.
- Commands that look risky are executed and logged in .bootstrap/safety_warnings.
    json rather than
  rejected by policy.
```

Language-Specific Guardrails

```
Additional project profiles are appended when repository evidence indicates Bazel
    /C/C++, Node.js,
Java, Rust, Go, or native C/C++ workflows. These profiles require the agent to
    use the checked-out
source workflow, install missing toolchains before use, prefer lockfile-indicated
     package managers,
and avoid replacing project validation with unrelated runtime version checks.
```

B Benchmark Collections

We evaluate BootstrapAgent on three public benchmarks that cover complementary repository setup scenarios. Table 6 summarizes their sizes and language distributions.

Repo2Run-Bench [16] is a Python benchmark centered on pytest-oriented environment construction. The original Repo2Run dataset contains 420 Python repositories; following the cross-benchmark setting used by HerAgent [22], we evaluate on the 122-repository subset used for comparison. This benchmark stresses Python dependency installation and test-command discovery.

ExecutionAgent-Bench [4] contains 50 repositories with diverse build and test ecosystems. In our benchmark metadata, the repositories are grouped into five main-language categories: JavaScript, Python, C, Java, and C++. The original benchmark further reports that these projects contain 14 programming languages when considering the full repository and test code. We use this benchmark to evaluate whether BootstrapAgent generalizes beyond Python-only setup.

Installamatic-Bench [25] contains 40 Python repositories with manually studied installation procedures and test suites. This benchmark is closely aligned with our target setting because it focuses on repository installation and validation rather than downstream patch generation.

We also examine the coverage of existing devcontainer configurations used by the existing-artifact baselines. Coverage denotes the percentage of repositories that provide such a configuration and is reported separately from bootstrap success. Devcontainer coverage is 24.0% on ExecutionAgent-Bench and 10.0% on Installamatic-Bench. For Repo2Run-122, our current audit identifies devcontainer configurations in 8 of the 122 repositories, corresponding to a confirmed coverage of 6.6%.

These results suggest that constructing development environments for repositories in these benchmarks is a nontrivial task that cannot generally be reduced to directly reusing existing devcontainer configurations.

C Experiment Setting

C.1 Effectiveness Evaluation

We evaluate the effectiveness of BootstrapAgent under a fixed execution budget and a controlled verifier environment. For each repository, the system is allowed to iteratively repair the generated bootstrap package until either verification succeeds or the configured budget is exhausted. The main budget parameters are shown in Table 7.

Table 6: Benchmark overview. ExecutionAgent-Bench is reported by main language in our benchmark metadata; the original paper notes that the 50 projects contain 14 programming languages in total.

Benchmark	Size	Language Distribution
Repo2Run-Bench	122	Python: 122 (100%)
ExecutionAgent-Bench	50	JavaScript: 12 (24%), Python: 11 (22%), C: 10 (20%), Java: 9 (18%), C++: 8 (16%)
Installamatic-Bench	40	Python: 40 (100%)

Table 7: Configuration used in the effectiveness evaluation.

Parameter	Value
Maximum repair loops	20
Maximum clean-replay repair loops	3
Maximum strongest-test repair attempts	5
Maximum structured LLM retries	5
Maximum repair structured LLM retries	5
Initial LLM timeout	300 s
Repair LLM timeout	180 s
Maximum shell commands	80
Maximum total wall-clock time	7200 s
Doctor command timeout	120 s
Setup command timeout	3600 s
Minimal verification timeout	300 s
Network timeout	1200 s

All verification runs are executed inside a deterministic Docker environment based on Ubuntu 24.04. The image contains only common system utilities required for repository inspection, dependency installation, and command execution, including `bash`, `curl`, `git`, `make`, `tar`, `unzip`, and related core Unix tools. The working directory inside the container is fixed to `/workspace/repo`. This minimal environment reduces interference from host-specific packages and ensures that successful verification depends on the generated bootstrap instructions rather than on pre-existing local state.

C.2 Downstream Agent Evaluation

To evaluate whether BootstrapAgent helps downstream coding agents, we conduct a paired controlled experiment using Claude Code as a representative code agent. Claude Code is executed in headless mode with session persistence disabled, so that previous conversations or historical context do not affect the results.

For each project, we compare two conditions. In the *warm* condition, the project directory contains the pre-generated `.bootstrap/` directory. The agent is instructed to read `.bootstrap/commands.json` and directly execute the provided `minimal_verify` and `strongest_verify` commands. In the *cold* condition, the original `.bootstrap/` directory is temporarily hidden. The agent must identify the programming language, build system, dependency requirements, and environment setup procedure from scratch. However, it is still given the same final verification commands from the original `commands.json`, ensuring that the warm and cold settings share the same target task.

Each run uses the same timeout and budget limits, namely 1800 seconds and a maximum API budget of 3 USD. After each cold run, the original `.bootstrap/` directory is restored to avoid contamination of subsequent experiments by files generated during the cold condition.

We record the structured JSON output produced by Claude Code, including total elapsed time, API time, input tokens, output tokens, cache-read tokens, number of interaction turns, and estimated cost. In addition, the agent is asked to report the final task status, the number of failed commands, and the number of search-related commands. We quantify the benefit of BootstrapAgent by comparing the paired cold and warm runs for the same project. For example, time savings are computed as

$$T_{\text{cold}} - T_{\text{warm}},$$

token savings as

$$C_{\text{cold}} - C_{\text{warm}},$$

and cost savings as

$$P_{\text{cold}} - P_{\text{warm}}.$$

Because every project is evaluated once under each condition, this forms a within-project paired comparison, reducing confounding effects from differences in project size, language ecosystem, dependency complexity, and test-suite cost.

D Additional Reproducibility and Ethics Details

D.1 Reproducibility Details

BootstrapAgent is evaluated as a repository bootstrapping system rather than as a model training method. The input to each run is a public GitHub repository from one of three external benchmark collections: Repo2Run-Bench, ExecutionAgent-Bench, and Installamatic-Bench. The output is a repository-local `.bootstrap` contract plus an evaluation log. The contract contains setup, diagnostic, and verification commands; the log records success or failure, stage outcomes, retry counts, token usage, time cost, and verifier traces.

All agentic planning and repair runs use `DeepSeek-V4-Flash-2026-04-25` with temperature 1.0. The implementation exposes the same setting through the `rethink bootstrap` and `rethink batch` commands, with `deepseek:deepseek-chat` used as the provider identifier in the released scripts. The default budgets are 20 repair rounds, 3 clean-replay repair rounds, 5 strongest-verification repair attempts, 5 structured-output retries for initial plans, 5 structured-output retries for repairs, 80 shell commands, and a 7200 second total time budget. LLM requests use a 300 second timeout for initial planning and a 180 second timeout for repair planning.

Verification is performed inside a Docker container built from an Ubuntu 24.04 base image. The base image includes only general-purpose development utilities such as `bash`, `git`, `curl`, `wget`, `make`, archive tools, and standard Unix file utilities. Language runtimes, compilers, package managers, project dependencies, and test tools are not assumed to be preinstalled; the generated `.bootstrap/setup.sh` must provision them when needed. Stage timeouts are 3600 seconds for setup, 120 seconds for doctor commands, 300 seconds for minimal verification, and 1200 seconds for strongest verification.

Success requires clean replay: `setup`, `doctor`, and `minimal_verify` must pass from a fresh container. `strongest_verify` is recorded when available and used as a guardrail against validation downgrades, but it is not required as the universal success gate because many real CI workflows depend on secrets, service containers, specialized hardware, or long-running jobs. External services, private credentials, GPU requirements, and expensive training workloads are outside the default success criterion.

D.2 External Asset Use and Redistribution

The benchmark composition is documented in Appendix B. The experiments use those existing public repository benchmarks rather than a newly constructed private benchmark.

The released research artifacts include the BootstrapAgent prototype, run scripts, benchmark URL lists, generated `.bootstrap` contracts, and summarized evaluation logs. We do not redistribute third-party repository source code as part of the paper artifact. Each upstream repository remains owned by its original maintainers and governed by its original license and terms of use. The URL lists are intended to let researchers recover the same public repositories through the upstream benchmark sources or GitHub, subject to those upstream terms.

D.3 Data, Ethics, and Asset Licenses

The research uses public GitHub software artifacts from existing benchmark collections. It does not involve private data, participant recruitment, crowdsourcing, surveys, or intervention with human subjects. BootstrapAgent may inspect repository files and public CI configuration to infer setup and verification commands, but the released paper artifact does not repackage third-party source code.

Existing repositories are credited by URL in the benchmark lists and remain governed by their original licenses and terms. The new assets introduced by this work are the BootstrapAgent prototype, the `.bootstrap` contract format, run scripts, generated contracts, and evaluation summaries. These artifacts are separate from the upstream repositories used as evaluation inputs.

D.4 LLM Usage

LLMs are a core component of the method and evaluation. BootstrapAgent uses LLM-based agents for repository evidence interpretation, command planning, and trace-driven repair. The downstream transfer experiments also use coding agents to compare cold repository setup against setup with a generated `.bootstrap` contract. LLMs are therefore part of the scientific method under evaluation, not merely a writing or formatting aid.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction summarize BootstrapAgent, its reusable `.bootstrap` contract, and its evaluation on three repository benchmarks. The claims about bootstrap success and downstream setup costs are evaluated in Section 4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Section 5.2 discusses contract staleness, risks of executing untrusted repository code, and limited portability beyond the evaluated Ubuntu 24.04/AMD64 environment.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A].

Justification: The paper contains a formal problem definition and optimization framing, but it does not introduce theorem-, lemma-, or proof-based theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes].

Justification: Section 4 describes the results. Appendices C and D.1 describe the evaluation protocols, success criteria, model settings, execution environment, and budgets for reproduction.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: Our code and benchmark data are publicly available at `https://github.com/Vossera/BootstrapAgent`. The `README.md` provides instructions for reproducing the experiments.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: Appendices C and D.1 specify the model configuration, Docker environment, repair and timeout budgets, and downstream cold/warm evaluation protocol.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes].

   Justification: Table 1 reports mean success rates and standard deviations over five independent runs for each method and benchmark.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: Section 4 reports token and time costs, and Appendix D.1 reports the Docker-isolated Ubuntu 24.04 base image, process controls, timeout budgets, repair budgets, and hosted LLM API usage. The method does not depend on a specific local GPU model or GPU memory capacity.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

   Justification: As described in Appendix D.3, the research uses public GitHub software artifacts from existing benchmark collections and does not involve private data, human-subject data collection, participant recruitment, surveys, or intervention.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes].

Justification: Section 4.3 reports reductions in downstream setup time and token usage. Section 5.2 discusses risks from outdated contracts and malicious repository code, and recommends periodic revalidation and manual inspection.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A].

Justification: The paper does not release high-risk pretrained models, image generators, or sensitive scraped datasets. The evaluation uses public software repositories from existing benchmark collections, and the released artifacts contain code, scripts, URL lists, generated bootstrap contracts, and summarized logs rather than redistributed third-party source code.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes].

Justification: We cite the benchmark papers in Appendix B and identify evaluated repositories by URL. The license of third-party repository source code is properly respected.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: The released code is accompanied by installation, usage, and evaluation instructions in `README.md`, and is distributed under the MIT license.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing, participant recruitment, surveys, or human-subject experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: No human-subject data collection or intervention is part of the benchmark, method, or evaluation.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes].

Justification: LLM-based agents are central to the method and evaluation. Sections 3 and 4 and Appendix D.4 describe their use for repository evidence interpretation, command planning, trace-driven repair, and downstream cold/warm setup evaluation.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.