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, but every such task depends on a costly prerequisite: bootstrapping the repository into a usable development state. This process requires substantial trial-and-error exploration, yet the resulting knowledge—resolved dependencies, repair strategies—stays trapped in a single conversation, unavailable to future agents. We therefore formulate repository bootstrapping as a reusable startup knowledge problem and introduce **BootstrapAgent**, a multi-agent framework that distills the heuristics discovered during bootstrap exploration into a persistent, verifiable, agent-consumable `.bootstrap` contract. Through evidence extraction, structured planning, deterministic Docker-based verification, and trace-driven repair, BootstrapAgent generates a contract covering environment setup, diagnostic checks, minimal verification, and accumulated repair knowledge. 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 92.9% success rate, outperforming the baseline by over 10% while reducing downstream agent token usage by 25.9% and build time by 22.3%. 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], whether to extend features, reproduce experiments, migrate components, or build prototypes. Before any such task can begin, the target repository must first be brought into a usable development state. However, the path from a freshly cloned repository to a working environment 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.

Although code agents, with their emerging capabilities in repository understanding and code execution, are rapidly lowering the barrier for developers to work with unfamiliar codebases, repository bootstrapping remains a particularly costly bottleneck for these agents [24, 36, 39, 41, 43]. 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 produced during this

Preprint.

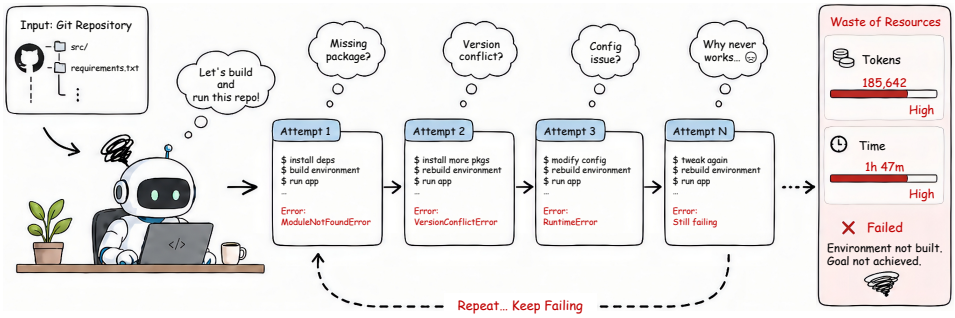


Figure 1: The process of deploying a repo by Code Agent.

process is ephemeral: resolved dependencies, working-directory corrections, and successful repair strategies remain trapped in a single-round conversation, unavailable to the next 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 prepare, diagnose, and verify 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 persistent, verifiable, agent-consumable `.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 heuristic repair-and-verify loop continues until the `.bootstrap` contract passes clean replay or a stopping condition is reached. Two key design choices underpin this pipeline: *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. BootstrapAgent achieves a 92.9% clean-replay bootstrap success rate, outperforming HerAgent [22] by over 10% (8.5 percentage points). When the generated contracts are reused by downstream Claude Code agents, token usage is reduced by 25.9% and build time by 22.3%. 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, 44] to learning-based environment construction [30]. More recent

systems move beyond static generation by using execution feedback, sandbox testing, and repository context to infer dependencies and validation commands. RepoST [40] and Repo2Run [16] construct executable repository environments through context mining and sandboxed verification; Treefix [32] repairs execution through prefix-tree search; and CXXCrafter [45] targets the specialized build and dependency challenges of C/C++ repositories. Complementary benchmarks such as DI-BENCH [46], CSR-Bench [38], R2E [17], and R2E-Gym [18] further expose dependency inference, scientific software deployment, and executable-repository construction as important evaluation problems. These works establish 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 discovered setup, diagnostic checks, verification commands, provenance, and repair knowledge 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 benchmarks such as SWE-bench [19, 42] and follow-on repository-level evaluations [5, 7, 8, 14, 23, 27, 29]. As coding agents have been adopted in unfamiliar repositories, agent developers and users increasingly rely on context files such as `AGENTS.md` [10] and `CLAUDE.md` [2]. Persistent artifacts such as configuration files, documentation, and contextual instructions continuously shape agent behavior [6, 13, 21, 26, 31]. However, prior work has not studied how repository-facing context files should specify and maintain the bootstrap process itself, including dependency installation, and recovery from setup failures

3 BootstrapAgent

3.1 Problem Formulation

We study repository bootstrapping as the problem of converting an unfamiliar repository into a startup contract that can be reused by future agents. Let R be a repository specified by a URL or a local path, and let E_R denote the observable bootstrap evidence extracted from R, including documentation, package metadata, lockfiles, build files, scripts, project layout, and CI workflows. The bootstrap configuration system must synthesize a contract C from E_R, without assuming access to maintainer knowledge or host-specific state. A `.bootstrap` contract 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. Each command also records its provenance in E_R. The contract exposes a simple execution protocol: run setup, run diagnostics, and then run verification.

Synthesis is mediated by a structured *bootstrap plan* P. The plan is generated from E_R, fixes the setup phases and verification goals, and links each decision to its supporting evidence, without committing to concrete shell commands. An initial contract C_0 is materialized from P, after which the system refines it through a bounded feedback loop. At iteration t, the contract C_t is executed by a verifier V in a fresh container with base environment B, producing an execution trace:

$$\tau_t = V(R, C_t;\, B),$$

which contains stage outcomes, exit codes, output, and failure locations. Conditioned on the fixed plan, the repair step proposes a trace-driven delta over the current contract:

$$C_{t+1} = \mathrm{Repair}(C_t,\, P,\, \tau_t),$$

while preserving commands and metadata that have already survived verification unless the trace provides direct evidence that they caused the failure. The plan P and evidence E_R constrain the feasible space of deltas: a sanity check rejects any revision that weakens an evidence-supported validation target. The loop runs under explicit budgets and stops once the contract is accepted or a budget is exhausted. The resulting contract is then frozen and consumed by downstream agents rather than further updated. In this sense, the iterative repair loop can be viewed as a heuristic system that operates under a bounded budget and terminates in a frozen, verified contract [37].

A contract is valid only if it passes *clean replay*: given B and V, the verifier runs C from a fresh container, with R mounted at a fixed path. Let Y_I, Y_D, and Y_M denote the stage outcomes of setup, diagnostics, and minimal verification, respectively. Then

$$\mathrm{Valid}(R, C;\, B, V) = \mathbb{I}\big[Y_I = \textsc{pass} \wedge Y_D = \textsc{pass} \wedge Y_M = \textsc{pass}\big],$$

subject to safety checks that reject non-local assumptions, swallowed failures, and degenerate verification commands. The strongest command S is not required to pass—real CI may depend on secrets, services, special hardware, or long-running jobs—but serves as an advisory guardrail: repair must not weaken an identified project-relevant target merely to clear the minimal gate.

The goal is not to make one container pass once, but to produce a contract that is reproducible, auditable, and cheap to reuse:

$$C^\star = \arg\max_{C \in \mathcal{C}(E_R)} \mathrm{Valid}(R, C;\, B, V) - \lambda_1 \mathrm{Cost}_{\mathrm{gen}}(R, C) - \lambda_2 \mathrm{Cost}_{\mathrm{reuse}}(R, C),$$

where $\mathcal{C}(E_R)$ is the space of contracts reachable from E_R through plan-mediated synthesis and repair, $\mathrm{Cost}_{\mathrm{gen}}$ is the cost of generating and repairing the contract, and $\mathrm{Cost}_{\mathrm{reuse}}$ is the downstream cost for a fresh agent to reach the same verified startup state using the contract. BootstrapAgent operationalizes this objective with bounded repair budgets, deterministic Docker-based verification, warm repair for efficient search, and a final clean-replay requirement for acceptance.

3.2 System Overview

Given the above formulation, BootstrapAgent synthesizes 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 prepares the target workspace and collects repository evidence. A *Discoverer Agent* explicitly scans files that are likely to affect bootstrapping, 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. A repair agent may perform multiple rounds of reasoning and tool calls; 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.

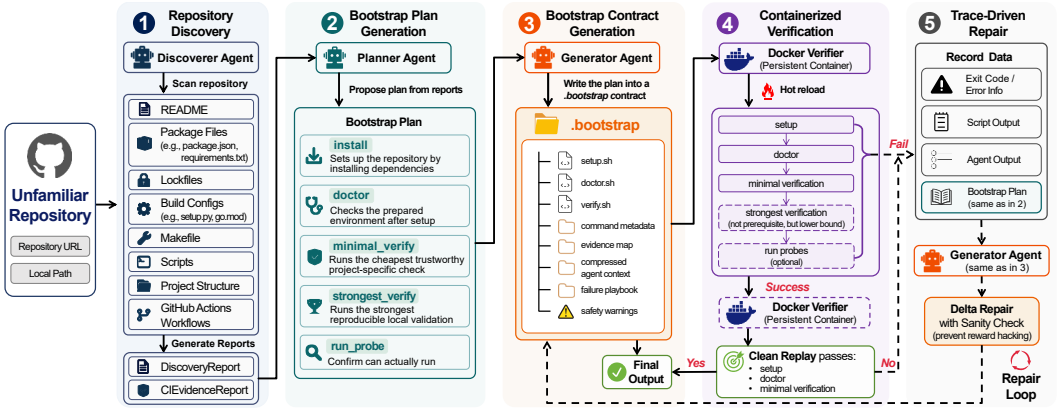


Figure 2: Overview of BootstrapAgent.

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 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

We measure the bootstrap success rate in a fresh container, where success requires `setup`, `doctor`, and `minimal_verify` to pass. The detailed experiment settings are shown in Appendix C.1. Table 1 reports the effectiveness of BootstrapAgent on the three public benchmarks. We compare against HerAgent, the strongest reported baseline across these benchmarks. Across 212 repositories, BootstrapAgent improves over HerAgent by 8.5 percentage points, corresponding to a 10.1% relative improvement. This indicates our multi-agent framework is effective for building most repositories.

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

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 and Table 3, warm-starting consistently reduces both time cost and token usage across all three benchmarks.

As shown in Figure 6, warm-starting consistently reduces downstream build-up cost across all three benchmarks. Warm-starting is an effective resource optimization for repository setup agents: 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.

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. Warm-starting reduces this repeated exploration by reusing prior setup state, cached observations, partially materialized environments, and existing build artifacts.

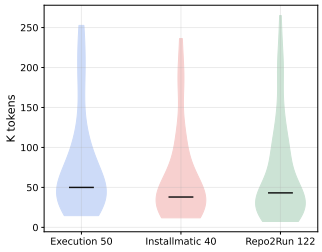


Figure 3: Token usage of BootstrapAgent.

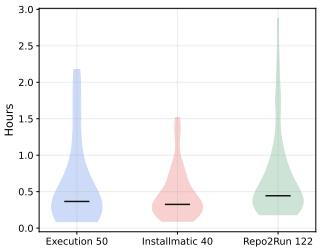


Figure 4: Time cost of BootstrapAgent.

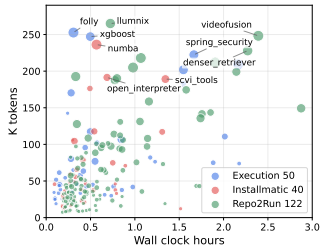


Figure 5: Relationship between token usage and time cost.

Table 1: Effectiveness across benchmarks.

Benchmark	HerAgent	Ours	Delta
ExecutionAgent	41/50	42/50	+2.0%
Installamatic	34/40	38/40	+10.0%
Repo2Run	104/122	117/122	+10.7%
Overall	179/212	197/212	+8.5%

Table 2: Median time of cold/warm execution.

Benchmark	Cold	Warm	Ratio
ExecutionAgent	285.2s	112.0s	0.39×
Installamatic	337.8s	282.3s	0.84×
Repo2Run	391.9s	309.8s	0.79×
Overall	346.3s	249.5s	0.72×

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, clean replay validation, warm-state repair, 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 clean replay validation, we accept a successful warm repair directly without rerunning the final contract from scratch. For warm-state repair, each repair round is executed in a fresh Docker container instead of reusing the previous verified state. 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.

We first analyze the impact of repository discovery, trace-driven repair, and clean replay on bootstrap success rate. As shown in Table 4, 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.

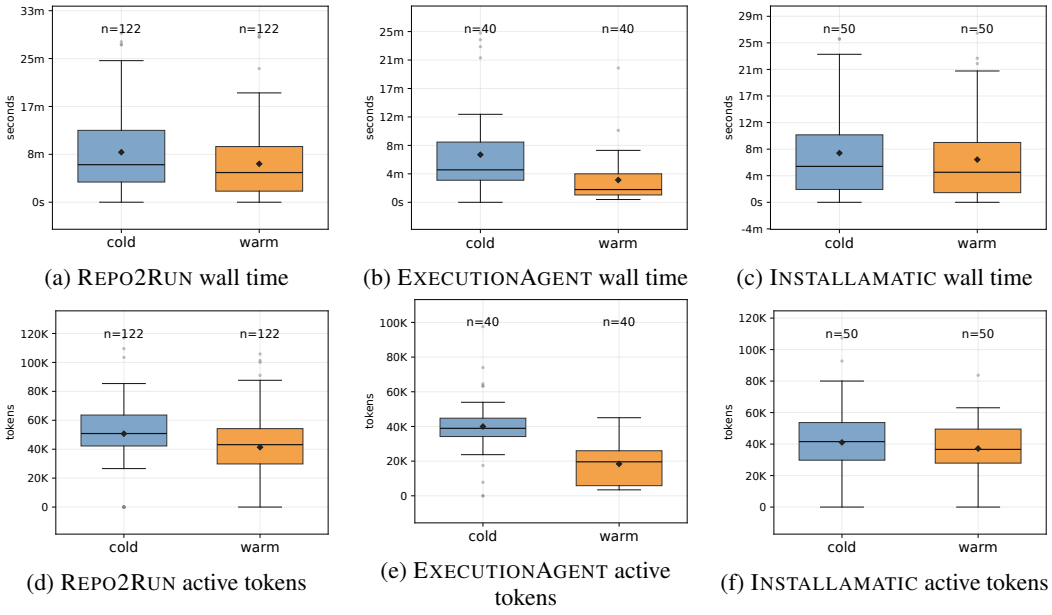
Repository discovery is also important. Removing the structured discovery report substantially reduces success, indicating that explicitly collecting bootstrap evidence before planning is more reliable than asking the agent to infer all setup information during generation. Clean replay has a smaller effect on the aggregate success rate, suggesting that warm-state repair usually preserves reproducibility. However, clean replay remains necessary because even a small number of warm-only successes may depend on cached packages, generated files, or residual container state, and should not be accepted as reusable bootstrap contracts.

We further analyze the time-saving effect of Warm-State Repair in Figures 7 and 8. Disabling Warm-State Repair 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. Overall, Warm-State Repair does not mainly improve success rate; instead, it makes trace-driven iteration computationally practical by avoiding repeated full setup costs.

Beyond success rate and efficiency, we examine whether the accepted `.bootstrap` artifacts remain trustworthy. Figure 9 shows that sanity checks catch 5/122 such cases on REPO2RUN, 4/50 on EXECUTIONAGENT, and 2/40 on INSTALLAMATIC. This mechanism improves reliability not by

Table 3: Token usage under cold and warm execution.

Benchmark	Cold Median	Warm Median	Ratio	Cold Total	Warm Total	Ratio
ExecutionAgent	38.9K	19.6K	0.50×	597.9K	115.7K	0.19×
Installamatic	41.6K	36.6K	0.88×	594.4K	312.6K	0.53×
Repo2Run	50.8K	43.1K	0.85×	797.8K	526.9K	0.66×
OVERALL	46.3K	36.9K	0.80×	710.7K	419.4K	0.59×



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

(d) REPO2RUN active tokens (e) EXECUTIONAGENT active tokens (f) INSTALLAMATIC active tokens

Figure 6: Warm-starting results across benchmarks. The first row reports wall-clock time, and the second row reports active token usage. Warm starts consistently reduce resource consumption, with the largest gains on EXECUTIONAGENT.

increasing apparent success, but by preventing reward-hacked or drifted repairs from being accepted as valid bootstrap contracts.

Overall, trace-driven repair provides the main success-rate gain, repository discovery improves the quality of the initial plan, and warm-state 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.

5 Discussion

5.1 Failure Analysis

Repositories may either omit essential build information, such as the required Python version, or provide inconsistent setup information across files. Both cases can lead the agent to choose the wrong environment at the start, and later repairs often fail because the resulting errors stem from version incompatibility rather than simple command mistakes. These failures explain why BootstrapAgent uses conservative acceptance criteria. On large or underspecified repositories, repeated repair can produce brittle fixes that silence the current error without yielding a transferable bootstrap, such as skipping a dependency, guessing an arbitrary pin, weakening verification, or relying on warm-container state. Clean replay and strongest-verification guardrails may lower the apparent solve rate, but they prevent overfitted `.bootstrap` files from being accepted as reusable startup knowledge.

Table 4: 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 Clean Replay	95.1% ↓ 0.8	82.0% ↓ 2.0	90.0% ↓ 5.0
w/o Trace-Driven Repair	26.2% ↓ 69.7	18.0% ↓ 66.0	40.0% ↓ 55.0
BootstrapAgent	95.9%	84.0%	95.0%

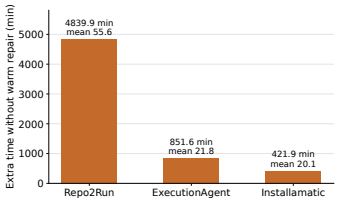


Figure 7: Estimated cost without warm repair.

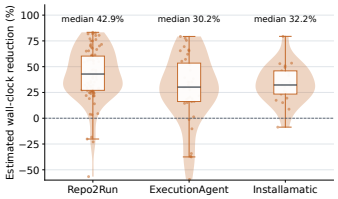


Figure 8: Distribution of warm-repair time reduction.

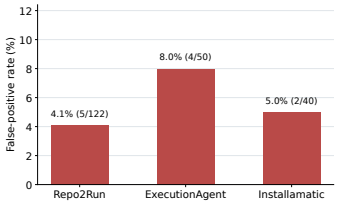


Figure 9: False positives caught by sanity check.

5.2 Limitations

BootstrapAgent establishes local bootstrap readiness rather than full CI reproduction or semantic correctness. This boundary is unavoidable for a broad benchmark: many real CI pipelines require secrets, external services or long-running jobs that cannot be made uniformly available without changing the task from repository bootstrapping to full system reproduction. Additionally, 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.

5.3 Broader Impacts

BootstrapAgent can reduce duplicated setup effort, failed commands, token use, and wasted compute when developers or coding agents work with unfamiliar repositories, especially public projects where users repeatedly rediscover the same setup and verification details. The main risk is over-trusting generated bootstrap contracts, which may encode transient workarounds, outdated dependencies, network assumptions, or weak validation. BootstrapAgent mitigates this through deterministic Docker verification, clean replay, strong-verification guardrails, structured traces, and warnings for fragile commands. Maintainer review remains necessary before adopting generated setup instructions in production documentation.

6 Conclusion

We introduced **BootstrapAgent**, a multi-agent framework that turns repository bootstrapping from an ephemeral 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. Through repository discovery, structured planning, deterministic Docker verification, trace-driven repair, warm repair with clean replay, and two-level verification, the generated contract helps downstream agents save token and time cost when building environment. Our evaluation on 212 repositories shows that this framework is effective in practice. These results suggest that repository setup should be treated not merely as a one-time environment construction task, but as an amortizable form of agent knowledge. At the same time, a successful `.bootstrap` contract does not replace full CI, maintainer judgment, or complete functional validation. Future work can extend this direction with stronger dependency solving, historical environment inference, external artifact recovery, and richer support for repositories requiring services, GPUs, or large-scale integration tests.

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] Jiayi Weng. Learning beyond gradients. https://trinkle23897.github.io/learning-beyond-gradients/, May 2026. Blog post.

[38] 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/.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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 5 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 recent work, 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.

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 6.

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.

Table 5: 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 6: 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	3600 s
Doctor command timeout	120 s
Setup command timeout	3600 s
Minimal verification timeout	300 s
timeout	1200 s

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 3600 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.