You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects

ISLEM BOUZENIA, University of Stuttgart, Germany
MICHAEL PRADEL, University of Stuttgart, Germany

The ability to execute the test suite of a project is essential in many scenarios, e.g., to assess code quality and code coverage, to validate code changes made by developers or automated tools, and to ensure compatibility with dependencies. Despite its importance, executing the test suite of a project can be challenging in practice because different projects use different programming languages, software ecosystems, build systems, testing frameworks, and other tools. These challenges make it difficult to create a reliable, universal test execution method that works across different projects. This paper presents ExecutionAgent, an automated technique that prepares scripts for building an arbitrary project from source code and running its test cases. Inspired by the way a human developer would address this task, our approach is a large language model (LLM)-based agent that autonomously executes commands and interacts with the host system. The agent uses meta-prompting to gather guidelines on the latest technologies related to the given project, and it iteratively refines its process based on feedback from the previous steps. Our evaluation applies ExecutionAgent to 50 open-source projects that use 14 different programming languages and many different build and testing tools. The approach successfully executes the test suites of 33/50 projects, while matching the test results of ground truth test suite executions with a deviation of only 7.5%. These results improve over the best previously available technique by 6.6x. The costs imposed by the approach are reasonable, with an execution time of 74 minutes and LLM costs of USD 0.16, on average per project. We envision ExecutionAgent to serve as a valuable tool for developers, automated programming tools, and researchers that need to execute tests across a wide variety of projects.

CCS Concepts: • **Software and its engineering** → *Development frameworks and environments*;

Additional Key Words and Phrases: large language models, LLM agents, autonomous software development, project setup automation, test suite execution, devops, software testing, artificial intelligence

ACM Reference Format:
Islem Bouzenia and Michael Pradel. 2025. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA047 (July 2025), 23 pages. https://doi.org/10.1145/3728922

1 Introduction

Executing the test suite of a software project is a critical step in various activities during software development and software engineering research. For human developers who are contributing to open-source projects, running the tests before submitting a pull request ensures that their changes do not introduce regressions. Likewise, the increasing popularity of large language model (LLM) agents that autonomously edit a project's code [3, 20, 33, 42, 45] creates a huge demand for a feedback mechanism to validate modifications [25, 32], and executing the test suite provides such a mechanism. Finally, researchers also depend on running tests, e.g., to evaluate the effectiveness of dynamic analyses [8] or to create benchmarks involving test execution, such as Defects4J [16], SWE-Bench [14], and DyPyBench [4].

Authors' Contact Information: Islem Bouzenia, University of Stuttgart, Germany, fibouzenia@esi.dz; Michael Pradel, University of Stuttgart, Germany, michael@binaervarianz.de.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTISSTA047
https://doi.org/10.1145/3728922

Unfortunately, executing the tests of an arbitrary project is far from straightforward in practice. Projects are developed in various programming languages and ecosystems, each with its own set of tools, dependencies, and conventions. Complex dependencies can pose significant challenges, especially when specific versions of libraries or tools are required. Documentation is often incomplete, inconsistent, or entirely missing, forcing developers to infer the necessary steps. Moreover, projects may have implicit assumptions about the environment, such as operating system specifics or required system configurations, that are not explicitly stated. The diversity of testing frameworks further complicates the process, as each framework has a unique setup and execution procedure.

Currently, there are three primary methods for executing the tests of a given project, each with notable limitations. The first method involves manually following the project's documentation and resolving any issues through trial-and-error. This approach is time-consuming and does not scale well with the number of projects. The second method is to reuse existing continuous integration/continuous deployment (CI/CD) workflows employed by project maintainers. However, not all projects have such workflows, and even when they exist, their execution often depends on a specific CI/CD platform, such as GitHub Actions, which is not fully accessible to the public. Directly using CI/CD workflows is further complicated by the fact that there are several popular platforms, each with its own configuration scripts and technology stack. The third method is to implement an automated, heuristic script designed to cover common cases. Such scripts typically focus on a single programming language and ecosystem, and they lack the flexibility to handle arbitrary projects: For example, Flapy [12] works only on Python projects that are available on PyPy, and even within the Python ecosystems fails to successfully run many test suites. Another example is the pipeline used to create GitBug-Java [31], which, starting from 66,042 repositories, eventually managed to execute the tests of only 228 repositories.

An effective solution would need to address multiple challenges. First, the approach should be aware of the latest technologies and tools for a range of popular programming languages. Second, the approach should be capable of understanding incomplete and partially outdated documentation. Third, the approach needs a way to interact with the system, such as executing commands, monitoring outputs, and handling errors. Finally, the approach must be able to assess whether the setup process has been successful, and if not, address any problems until the test suite runs successfully. To the best of our knowledge, there is currently no existing work that addresses these challenges. This gap presents a substantial obstacle for developers, automated coding techniques, and researchers who need a reliable and scalable solution for running tests across a wide variety of projects. Motivated by these challenges, the question addressed in this work is: *Given an arbitrary project, how can we automatically build the project and run its test suite?*

This paper presents ExecutionAgent, the first LLM-based agent for autonomously setting up arbitrary projects and executing their test suites. The approach addresses the four challenges described above as follows. For the first challenge, we present the novel concept of meta-prompting, which asks a recently trained LLM to automatically generate up-to-date, technology-specific, and programming language-specific guidelines instead of manually engineering and hard-coding a prompt. We address the second challenge by using the LLM to parse and understand the project's documentation and web resources related to the project. To handle the third challenge, we connect the agent to a set of tools, such as a terminal, to execute commands, monitor outputs, and interact with the system. Finally, we address the fourth challenge by enabling the agent to iteratively refine its process based on feedback from previous steps, similar to how a human developer would work.

To evaluate ExecutionAgent, we apply it to 50 open-source projects that use 14 different programming languages and many different build and testing tools. The approach successfully sets up and executes the test suites of 33/50 projects. Comparing to several baselines, such as manually designed and LLM-generated scripts targeting projects in a specific language, as well as a general-purpose

LLM-based agent, we find ExecutionAgent to outperform the best available technique by 6.6x. To validate the test executions, we compare them against a manually established ground truth and find that the test results (in terms of the number of passed, failed, and skipped tests) closely resemble the ground truth, with an average deviation of 7.5%. Studying the costs of the approach, we find that the average execution time is 74 minutes per project, and the average LLM costs are USD 0.16 per project. Overall, these results demonstrate that ExecutionAgent has the potential to serve as a valuable tool for developers, automated programming tools, and researchers that need to execute tests across a wide variety of projects.

In summary, this paper contributes the following:

(1) The first autonomous, LLM-based agent for automatically setting up arbitrary projects and executing their test suites.
(2) The novel concept of meta-prompting, which allows the agent to query an LLM for the latest guidelines and technologies related to building projects and running test suites.
(3) Several technical insights on design decisions that enable the agent to effectively interact with the system, execute commands, monitor outputs, and handle errors.
(4) Empirical evidence that the approach can successfully execute the test suites of a wide variety of projects, clearly outperforming existing techniques.

2 Approach

The following presents our approach for executing the test of arbitrary projects. We start by defining the problem we aim to address (Section 2.1), then provide an overview of our approach (Section 2.2), and finally describe the components of our approach in detail (Sections 2.3 and 2.4).

2.1 Problem Statement

The problem we aim to address is the following: Given a software project, identified, e.g., by a URL of a git repository or a file path, we want to automatically generate scripts to install the project and run its tests. Specifically, the desired output consists of two scripts: one to create an isolated environment, such as a container, and one to build the project and run its tests within the isolated environment. By running the tests, we mean executing the project's test suite and collecting the results, such as the number of tests that pass, fail, and are skipped.

We aim to address this problem in a way that offers two important properties. First, the approach should be technology-agnostic, i.e., it should support projects written in different programming languages, using different build systems, and using different testing frameworks. This property is crucial as it allows the approach to be used across a wide range of projects. Second, the approach should be fully automated, i.e., it should not require any manual intervention or additional information beyond the project itself. This property is essential to ensure that the approach can be used at scale and without human intervention.

The formulated problem has, to the best of our knowledge, not been addressed by prior work. In particular, existing approaches either focus on specific programming languages or ecosystems [12, 31] or are based on significant manual intervention [4].

2.2 Overview of ExecutionAgent

We address the above problem by presenting an approach that leverages LLMs and the concept of LLM agents. By *LLM agent* we here mean a system that uses an LLM to autonomously interact with a set of tools in order to achieve a specific goal [25]. The tools we use are similar to those a human tasked with setting up a project might use, such as commands available via a terminal. The intuition behind this approach is that the LLM can "understand" various sources of information,

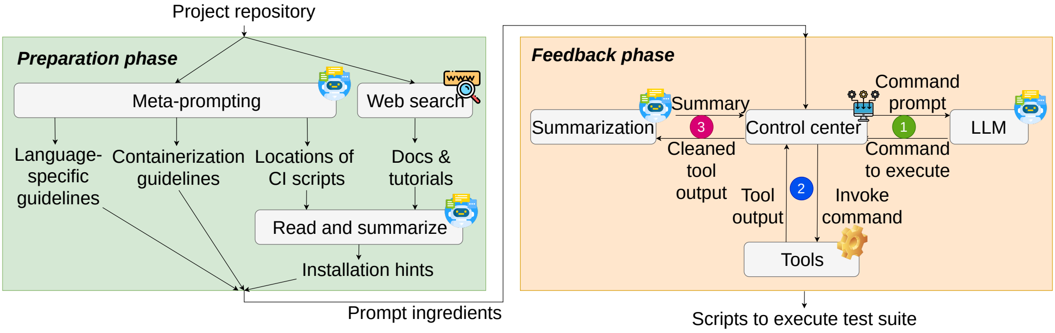

Fig. 1. Overview of ExecutionAgent.

such as project documentation, existing scripts, and the output of tools, and use this understanding to decide on the steps necessary to execute the tests of the given project. As shown in Figure 1, the approach encompasses two phases, which we briefly describe in the following.

Phase 1: Preparation. Given a project repository, this phase gathers information required to construct the initial prompt of the agent, called *prompt ingredients*. A key challenge is creating these prompt ingredients in a technology-agnostic way, as the project may be written in any programming language and use any testing framework. Our approach addresses this challenge through *meta-prompting*, a novel concept that allows the agent to query an LLM for the latest guidelines and technologies related to building projects and running test suites. Specifically, ExecutionAgent uses meta-prompting to generate language-specific guidelines, guidelines about the latest containerization technology, and possible locations of commonly used CI/CD scripts. In addition, the approach queries the web to gather hints on building the given project. Finally, our approach uses an LLM to generate general-purpose, language-independent guidelines for creating a build and test environment for an arbitrary project. These guidelines help the agent follow a high-level plan to reach its goals. The generated and gathered information is then passed as prompt ingredients to the second phase.

Phase 2: Feedback loop. This phase repeatedly invokes tools, as guided by the LLM, in order to build the project and execute its test suite. Specifically, ExecutionAgent repeatedly iterates through three steps. Step ① queries the LLM for the next command to execute, using a dynamically updated *command prompt* that contains the prompt ingredients from Phase 1 and a summary of the already invoked commands. Step ② executes the command suggested by the LLM by invoking one of the tools. Because the output of a tool may be verbose and contain irrelevant information, step ③ requests the LLM to summarize the output and extract the most relevant information. The summarized output is then used to update the command prompt, and the three steps repeat until the agent determines that the test suite has been successfully executed. This entire process is managed by a component we call the *control center*.

The steps taken by ExecutionAgent are described in more detail in Algorithm 1, which we will explain in the following sections.

2.3 Preparation Phase

The preparation phase is motivated by two goals: (1) to gather project-specific information that could be helpful for building the given project and running its tests, and (2) to obtain guidelines that will help the LLM agent invoke the right tools and commands during the feedback loop in

Algorithm 1 High-level algorithm of ExecutionAgent

Input: Repository URL u
Output: Scripts to set up an environment and run tests

```
 1: // Phase 1: Preparation
 2: lang_guidelines ← LLM("Give language-specific guidelines", get_languages(u))
 3: container_guidelines ← LLM("Give guidelines on recent containerization technology")
 4: ci_paths ← LLM("Give common locations of CI scripts within a repo")
 5: ci_hints ← read_ci_scripts_and_summarize(u, ci_paths)
 6: install_hints ← search_web(u)
 7:
 8: // Phase 2: Feedback loop
 9: attempts_left ← 3
10: p_attempt_lessons ← ""
11: while attempts_left do
12:     p_cmd ← create_command_prompt(lang_guidelines, container_guidelines, ci_hints, install_hints,
        p_attempts_lessons),
13:     budget_left ← 40
14:     while budget_left do
15:         // Step 1: Get next command
16:         thought, cmd ← LLM(p_cmd)
17:         // Step 2: Execute command
18:         cmd_result ← execute(cmd)
19:         if cmd == "task_done" and valid(cmd_result) then
20:             scripts ← read_target_files()
21:             return scripts
22:         end if
23:         // Step 3: Summarize command output
24:         p_summarize ← create_prompt(p_cmd, cmd_result)
25:         cmd_result_summary ← LLM(p_summarize)
26:         p_cmd ← update_prompt(p_cmd, thought, cmd, cmd_result_summary)
27:         budget_left ← budget_left − 1
28:     end while
29:     attempts_left ← attempts_left − 1
30:     p_attempt_lessons ← analyze_commands_and_thoughts(cmd_list, thoughts_list)
31: end while
```

phase 2. To reach these goals, we follow a set of steps that are detailed in Algorithm 1 between lines 1 and 6, and which we present in detail in the following. Since several of these steps use the idea of meta-prompting, we begin by explaining this concept.

2.3.1 Meta-Prompting. A key challenge faced by ExecutionAgent is that technologies and best practices for building software projects and running test suites are constantly evolving. One way to address this challenge would be to spend significant amounts of time in engineering suitable prompts for the LLM agent. Such prompts should provide guidelines on how to use common build tools, package managers, testing frameworks, etc. for all the programming languages, software ecosystems, and tools that the agent might encounter. However, this approach comes with several drawbacks: it is time-consuming, unlikely to cover all relevant technologies, and may become outdated quickly.

Guidelines Meta Prompt (for Java)

I want to install and run tests of a **Java** project. Give a list of 10 guidelines that I should follow to achieve my task (e.g., to figure out installation steps, dependencies, how to run tests) while considering latest practices and technologies.

LLM Response

1. Review Project Documentation and README
Why: Documentation typically includes installation steps, dependencies, and setup instructions.
What to do: Check the README.md or other documentation files for prerequisites and specific instructions for installation and testing.

2. Verify Java Version Compatibility
Why: Java projects may target specific versions (...)
What to do: Confirm the Java version required by checking the pom.xml (Maven) or build.gradle (Gradle) file (...)

3. Check for Dependency Management Tool
(...)

10. Generate Test Reports for Results and Debugging
Why: Test reports provide detailed logs and summaries, aiding in debugging.
What to do: Maven projects can use mvn surefire-report:report to generate reports. Gradle usually generates HTML reports by default (build/reports/tests/test/index.html).

Fig. 2. Meta-prompting to obtain guidelines for Java projects. Response shortened for brevity.

Instead of engineering such prompts manually, we leverage the impressive knowledge of LLMs to generate prompts with up-to-date guidelines targeted at the given project. Since LLMs are often trained on a vast range of documents, including documents describing the latest technological developments, they offer a built-in mechanism for adapting to evolving technology landscapes. We leverage this capability by using a high-level prompt, called a *meta-prompt*, to query the LLM for technology-specific guidelines and other information, which we then use to construct the command prompt for the agent. Specifically, ExecutionAgent uses meta-prompting to generate three kinds of prompt ingredients, which we describe in the following.

2.3.2 Language-Specific and Language-Independent Guidelines. During initial experiments, we observed that the LLM agent benefits from guidelines that describe how to typically build projects written in a specific language and how to run their tests. Adopting the idea of meta-prompting, ExecutionAgent creates such guidelines by querying an up-to-date LLM to generate a list of language-specific guidelines. To this end, we heuristically infer the main programming language of the project during the preparation phase (using GitHub's Linguist library[1]) and use this information to generate the language-specific guidelines.

Figure 2 illustrates an example of a meta-prompt for a Java project and the response from the LLM. The prompt asks for a list of ten guidelines for a specific programming language. The reason for asking for a specific number of guidelines is that otherwise, the LLM sometimes generates only a few broad guidelines, which are lacking in detail, or may generate an extensively long list, which is hard to process. To avoid such variation, we explicitly request a specific number, and we found ten to be sufficiently comprehensive.

In addition to creating language-specific guidelines, we instruct the agent to collect further information about the projects and tests using a set of language-independent guidelines that we also generate with an LLM. These language-independent guidelines ensure that our approach

---

[1] https://github.com/github-linguist/linguist

Summarization response

```
{ 'summary': 'The karma.conf.js file configures the testing framework for Bootstrap,
     utilizing Jasmine for unit testing and Rollup for preprocessing. It supports headless
     testing with Chrome and Firefox, and includes configurations for coverage reporting and
      BrowserStack integration.',
  'extracted dependencies': ['jasmine', 'karma', 'karma-jasmine','karma-rollup-preprocessor'
       , 'karma-chrome-launcher', 'karma-firefox-launcher', 'karma-detect-browsers', 'karma-
       coverage-istanbul-reporter'],
  'important commands': ['npm test'],
  'important files/links/hyperlinks': [] }
```

Fig. 3. Example of an LLM-generated summary.

remains effective even when a project's setup does not strictly adhere to the common practices of its main language, for instance, due to significant influence from secondary languages.

2.3.3 Container Guidelines. To ensure that the project's tests are executed in an isolated environment, ExecutionAgent aims at building and testing the project inside a container. Similar to the language-specific guidelines, we use meta-prompting to query the LLM for guidelines on the latest and most used containerization technology. For example, at the moment of writing, the GPT-4o model responds to our meta-prompt by stating that Docker is the best option.

2.3.4 Existing CI/CD Scripts. Some projects include CI/CD scripts that provide hints about installation steps and project dependencies. To leverage such scripts, ExecutionAgent follows a three-step process. First, it uses meta-prompting to query the LLM for common folder names, file names, and extensions associated with CI/CD scripts. Second, based on the obtained list, it searches the repository for matching files and folders. Third, the identified files are analyzed by the LLM to determine which files are relevant to building and testing.

While CI/CD scripts can sometimes be outdated or incomplete, they may still provide useful insights into the overall setup process and any required non-standard steps. To extract the most relevant information, ExecutionAgent queries an LLM with the raw file contents, asking it to provide: (1) a concise natural language summary of the script, (2) a list of project dependencies, (3) important commands for setting up and testing, and (4) references to relevant files or links. Figure 3 shows an example of the LLM's response to the summarization prompt.

2.3.5 Web Search. Some projects maintain explicit and structured documentation on external platforms, such as *readthedocs.com*, which can provide clearer installation instructions than their respective GitHub repositories. Additionally, search engines efficiently retrieve the most relevant resources based on a few keywords, reducing the need for exhaustive exploration.

To leverage web resources, ExecutionAgent queries a search engine with "How to install the <LANGUAGE> project '<PROJECT>' from source code?", where <LANGUAGE> and <PROJECT> correspond to the project's primary programming language and name, respectively. Instead of parsing webpages sequentially—which would increase the number of queries and associated costs—the approach extracts text from the top five results and asks an LLM to summarize them. The summarization process extracts relevant build and test dependencies, as well as installation steps, including commands and configuration details. If a search result lacks useful installation hints, the model is instructed to return: "This page does not contain setup-related information."

2.4 Feedback Phase

Based on the prompt ingredients obtained in the preparation phase, the second phase of our approach is a feedback phase that iteratively invokes tools to build the project and execute its test suite. As shown in lines 8 to 31 of Algorithm 1, the feedback phase consists of two nested loops. The inner loop runs a series of commands, guided by the LLM, until either the test suite completes successfully or a configurable command limit (default: 40) is reached. The outer loop allows for multiple attempts to build and execute the test suite, with a configurable maximum number of attempts (default: 3). This design allows recovery from occasional failures where the build process encounters an error, preventing the tests from running. By allowing multiple attempts, the system can adjust its approach to overcome such errors. At the start of each outer loop iteration, the system incorporates lessons learned from the previous attempt (if any) into the command prompt. These lessons are generated by prompting the LLM to analyze the previous sequence of thoughts and commands, identify challenges encountered, and suggest adjustments for the next iteration (line 30). Each iteration of the inner loop consists of three steps, as outlined in Figure 1 and detailed in the following.

2.4.1 Step 1: LLM Selects the Next Command. The core of our approach is an LLM that selects the next command to execute based on the current state of the installation process. To guide the LLM in selecting the next command, the approach constructs a *command prompt* that contains the prompt ingredients obtained in the preparation phase, as well as a summary of the commands executed so far. The command prompt consists of predefined static sections, and dynamic sections, which are either created based on the preparation phase (line 12) or updated at the end of each iteration of the inner loop (line 26). Specifically, the prompt contains the following sections:

(1) Agent role (static): Defines the primary task of the agent and the success criteria. Concretely, the agent's role is to build the given project and ensure that the test cases execute properly.
(2) Goals (static): Outlines the specific objectives the agent must accomplish, namely:
  - Gather installation-related information and requirements.
  - Write scripts to build the project and execute its tests.
  - Run the produced scripts and refine them, if necessary.
  - Analyze the test execution results and summarize them by providing the number of passing, failing, and skipped tests.
(3) Tools (static): Lists the tools available to the agent. We describe the tools in detail in Section 2.4.2.
(4) Guidelines (dynamic): Contains the programming language-specific guidelines and the containerization guidelines obtained in the preparation phase.
(5) Installation hints (dynamic): Contains the installation hints obtained from any existing CI/CD scripts and the web search. It also contains lessons learned from previous iterations of the outer feedback loop.
(6) Tool invocation history (dynamic): Summarizes the tools invoked so far, as well as the output that was produced. Initially, this section is empty.
(7) Instruction for creating the next command prompt (static): Specifies that the LLM should provide the next command to execute based on the context available in the prompt. We define the expected format of the LLM's response in a TypeScript interface, as shown in Figure 4. The expected format includes the agent's thoughts, as well as the name of the tool to invoke and the arguments to pass to the tool. The rationale for asking the LLM to describe its thoughts is two-fold: First, it has been shown empirically to improve the quality of the LLM's responses [38]. Second, it allows for easier debugging and understanding of the LLM's decisions. If the LLM fails to respond in this format, the control center returns an error message to the LLM, mentioning the usage of the wrong format and reminding the LLM of the correct one.

Test results

```
interface Response {// Express your thoughts based on the information that you have
    collected so far, the possible steps that you could do next, and also your reasoning.
thoughts: string;
tool: {name: string; args: Record<string, any>; }; }

Here is an example of a command call that you can output:
{ "thoughts": "I need to check the files and folders available within this repository.",
"tool": {"name": "linux_terminal", "args": {"cmd": "ls"} } }
```

Fig. 4. TypeScript interface to specify the JSON format of the LLM's response to the command prompt.

2.4.2 *Step 2: Invoking Tools.*

To enable our approach to take the steps necessary for building the project and running its tests, we provide four tools to the agent. ExecutionAgent invokes these tools based on the LLM's response to the command prompt (lines 17 to 22).

Terminal. Similar to human developers, access to a terminal is crucial to successfully set up a project and run its tests, e.g., to build dependencies or list available files. We provide the agent with the capability to execute any command available in a Linux terminal via the `linux_terminal` tool. The tool takes a command to execute as input and returns the output of the command.

File I/O Tools. While the terminal would, in principle, be sufficient to perform any kind of operation on the system, we provide two additional tools for interacting with the file system. These tools are meant to emulate the developer's interaction with text editors, where they open a file, read parts of it, and sometimes make changes by writing to it. The `read_file` tool takes a file path as input and returns the content of the file. The `write_file` tool takes a file path and content as input and writes the content to the file. The latter tool is particularly useful for writing the scripts expected as the output of ExecutionAgent, e.g., a `Dockerfile` that creates a container and a `install.sh` script with the sequence of commands to build dependencies, compile or build the project, and eventually run the test suite.

End of Task. In addition to the tools described above, we provide the agent with a special tool called `task_done`. This tool is used to signal that the agent has successfully completed its task, i.e., that the project has been built and its tests have been executed. The tool expects a natural language description that explains why the agent has finished the task.

2.4.3 *Step 3: Summarization and Extraction.*

The output of tools can be verbose and contain lots of irrelevant information. For example, suppose the agent executes a command to list all files in a directory or reads the content of a file, then simply returning the output of the tool to the LLM for the next command would quickly exceed the available prompt size. Even with LLMs that offer a large prompt size, it is beneficial to reduce the amount of information in the prompt to keep the agent focused on the most important information and to reduce the overall costs of the approach.

To reduce the amount of text that results from a tool invocation, ExecutionAgent shortens the output whenever the output exceeds a certain number of tokens (default: 200). The approach asks another LLM to summarize the output and extract the most relevant information (lines 24 to 25). The expected format for the summary is the same as illustrated in Figure 3. Once the output of the most recently invoked command has been summarized, the approach appends the command and its output summary to the tool invocation history section of the command prompt (line 26).

2.4.4 Control Center. The control center orchestrates the three steps described above, managing the interaction between the LLMs and the tools. Specifically, it performs the following tasks:

- Parse the LLM output and validate whether it conforms to the specified output format.
- Invoke the next step as specified in Algorithm 1, e.g., by executing a specific command.
- Monitor tool execution by tracking launched processes within the container and detecting when they terminate. Since some commands may take a long time before producing any output, merely watching stdout and stderr is insufficient. Instead, the control center ensures that process completion is explicitly detected. The complete stdout and stderr outputs are returned to the LLM once the process terminates.
- Enforce a configurable timeout (default: five minutes) for any command execution. If a command does not finish within this interval, the control center provides the LLM agent with the available output and process status. The LLM then decides among three options: (1) wait for another five minutes, (2) provide input to the running tool (useful for interactive prompts, such as confirming an installation with "y"), or (3) terminate the command.
- Clean the output of tools by removing terminal color codes and other special characters, such as those used for progress bars.

When the agent selects the `task_done` command, it indicates that the agent believes that the project has been successfully built and that its test suite has been executed. Before terminating ExecutionAgent, the control center validates that the agent has indeed met all goals. This involves checking whether three files exist: a file to create a container (e.g., a `Dockerfile`), an installation script to be executed within the container, and a file that contains the test results (i.e., the number of passed, failed, and skipped tests). If all goals are met, ExecutionAgent returns the installation and test scripts to the user (line 21). Otherwise, the control center provides feedback to the LLM agent, pointing out what exactly is missing, and asks for a new command. Figure 5 gives an example of the scripts produced for the OpenVPN project. The two scripts show the configuration chosen by ExecutionAgent, the installed dependencies, and the steps to build the project and execute its tests.

3 Evaluation

Our evaluation aims to answer to following research questions:

- RQ1 (effectiveness): How effective is ExecutionAgent at correctly setting up projects and running their test cases?
- RQ2 (costs): What are the costs of ExecutionAgent in terms of execution time and token usage when interacting with the LLM?
- RQ3 (ablation study): What is the impact of different components and configurations?
- RQ4 (trajectories): How does ExecutionAgent interact with the tools, and what trajectories does it take to reach the goal?

3.1 Experimental Setup

3.1.1 Implementation and Model. ExecutionAgent is implemented in Python and bash. To allow for an isolated execution of ExecutionAgent itself, we use a Docker container. The agent is based on OpenAI's GPT-4o-mini model, which we access through their Python API. The usage of other models is possible as long as they support JSON output format specified through an interface. At the moment of writing the paper, Claude, DeepSeek, and Llama models support this feature.

3.1.2 Metrics. The task performed by ExecutionAgent consists of two sub-tasks: (i) building the project, and (ii) running the test suite. We measure how effective the approach is at performing these tasks by measuring the *successful build rate*, i.e., the proportion of projects where the approach manages to correctly build the project, and the *successful testing rate*, i.e., the proportion of projects

Dockerfile

```dockerfile
FROM ubuntu:20.04
# Set the timezone to avoid interactive prompts
ENV TZ=XXX/YYY # omitted for anonymity
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
# Update package lists and install necessary build tools and dependencies
RUN apt-get update && \
apt-get install -y --no-install-recommends \
sudo \
build-essential \
liblzo2-dev \
libpam0g-dev \
liblz4-dev \
libcap-ng-dev \
libnl-genl-3-dev \
linux-libc-dev \
autoconf \
automake \
libtool \
cmocka-dev \
python3-docutils \
git \
&& apt-get clean && rm -rf /var/lib/apt/lists/* || exit 0
WORKDIR /app
# Clone the OpenVPN repository
RUN git clone https://github.com/OpenVPN/openvpn.git || exit 0
WORKDIR /app/openvpn
CMD ["bash"]
```

install_and_run.sh

```bash
#!/bin/bash
# Run autoconf to prepare the build system
autoreconf -fvi
# Configure the build system
./configure --enable-werror
# Compile the project
make -j$(nproc)
# Run tests
make check VERBOSE=1
echo "Setup␣and␣testing␣complete."
```

Fig. 5. Scripts produced by ExecutionAgent for the OpenVPN project.

where the approach manages to run the test suite. To determine these rates, we manually inspect the produced scripts and the test results.

To better understand the output of ExecutionAgent, we measure the *script size* of the produced scripts, which we define as the number of commands, excluding any comments, white spaces, and "echo" commands. We split composite commands and count their components individually. For example, the line `make . && make test` is counted as two commands `make .` and `make test`.

To validate the test suite executions, we compare the test execution results produced by ExecutionAgent with a manually established ground truth. The ground truth is obtained by searching for CI/CD logs of the targeted projects, and by extracting the number of passed, failed, and skipped tests. We compare the results of ExecutionAgent with the ground truth in terms of these three numbers. Specifically, we compute the *deviation from ground truth* as $deviation = |\frac{(nb_{EA} - nb_{GT})}{nb_{GT}}| * 100$, where nb_{EA} is the number of tests produced by ExecutionAgent and nb_{GT} is the number of tests in the ground truth. For example, if among the tests executed by ExecutionAgent, there are 95 passing tests, while the ground truth has 100 passing tests, then the deviation is 5%. We compute a separate deviation for the number of passing, failing, and skipped tests, and then average these deviations to obtain the overall deviation.

3.1.3 Dataset. To construct a dataset for evaluating ExecutionAgent, we gather 50 open-source projects from GitHub. The following describes the selection criteria and process.

Criteria for selecting projects. We select projects that meet the following criteria: (i) *Diversity in programming languages:* We select ten projects each for Python, Java, C, C++, and JavaScript, ensuring that each project has one of these languages as either the dominant or second most-used language. (ii) *Availability of ground truth test execution results:* We ensure that test execution results are accessible through logs from CI/CD platforms. As different projects use different platforms, we collect ground truth data from GitHub Actions, CircleCI, Jenkins, CirrusCI, and occasionally, from a project's website (e.g., for TensorFlow). Our selection process prioritizes projects that maintain publicly accessible, detailed reports of test execution results, such as build/test logs on CI/CD platforms. One indicator of this is the presence of a CI/CD badge in the repository's README. (iii) *Project maturity and activity:* Each selected project has at least 100 stars and 100 commits, with at least one commit made within the six months prior to data collection. This criterion ensures that the repositories are active and maintained.

Dataset construction process. Our dataset construction follows a systematic process to ensure quality and relevance. We search for the most popular projects for each of the five primary languages and rank them based on their popularity (number of stars and forks). We then manually inspect projects, starting at the top of the ranked list, to check whether they meet the selection criteria described above. If a project does not meet the criteria, we move to the next project in the list. We continue this process until we have collected 50 projects that meet the criteria.

Dataset statistics. The projects in our dataset exhibit significant activity and complexity. The median project has approximately 10K commits, 45K stars, and 1.4K test cases. The code in the 50 selected projects is written in 14 programming languages. When considering only test code, the projects cover 8 languages, including Kotlin, Scala, and TypeScript, in addition to the five primary languages. On average, 87% of the tests are written in the main language of the project. Some projects write test cases in different languages than their main language, though. For example, the CPython project has many test cases written in Python that test code implemented in C.

From another perspective, 38 out of the 50 projects use automated build tools or provide configuration files for them.

All projects in the dataset are capable of running on a Linux system, as confirmed by logs retrieved from CI/CD results. The dataset also highlights variations in CI/CD usage: 33 out of 50 projects contain scripts for both building and testing, 10 projects have build scripts only, and 7 projects lack both build and test scripts. Additionally, 9 projects contain submodules, indicating a higher degree of structural complexity. Table 1 lists all 50 projects and their characteristics.

3.1.4 Baselines. To the best of our knowledge, there is no existing technique that addresses the same problem as ExecutionAgent. However, we compare our approach to three related baselines, which represent the state of the art in this domain.

LLM scripts. LLMs have seen large amounts of data, including documentation on how to install projects. We leverage this feature to ask an LLM to generate a script that prepares an environment to build an arbitrary project and run its tests. As our dataset is gathered by focusing on five main languages, we ask the LLM to create one specialized script for each of these five languages. The prompt we use is given in Figure 6. Given the script created in response to this prompt, we attempt to build a project using the script corresponding to the project's main language.

Table 1. Projects used for the evaluation. Languages: J=Java, P=Python, JS= JavaScript, TS= TypeScript, KT=Kotlin, ASM=Assembly, SH=Shell. Symbols: + means successfully built but tests not executed, - means failed to build.

Project	**Languages**	**Ground Truth**				**ExecutionAgent**			
		Tests	**Pass**	**Fail**	**Skip**	**Tests**	**Pass**	**Fail**	**Skip**
Activiti	J	3269	3266	0	3	2392	2389	1	2
ansible	P	1717	1703	0	14	+	+	+	+
axios	JS, TS	203	203	0	0	203	195	8	0
bootstrap	JS, Html/css	808	808	0	0	808	793	15	0
ccache	C, SH	47	36	0	11	47	36	0	11
Chart.js	JS, TS	3298	3298	0	0	+	+	+	+
commons-csv	J	856	845	0	11	856	845	0	11
cpython	P, C, C++	478	462	0	16	478	433	4	42
deno	JS, TS, Rust	1458	1446	0	12	511	474	36	1
distcc	C, P	57	57	0	0	57	53	4	0
django	P	17604	16151	5	1448	17605	16148	6	1451
dubbo	J	14020	13678	0	342	14020	13678	0	342
express	JS	1228	1228	0	0	1228	1228	0	0
flask	P	484	481	0	3	484	480	1	3
flink	J, Scala, P	105242	100644	0	4598	+	+	+	+
folly	C++, P	3089	3088	0	0	-	-	-	-
FreeRTOS	C, ASM	215	215	0	0	-	-	-	-
git	C, SH, Perl	31715	31715	0	0	30264	29203	265	796
guava	J	857221	857221	0	0	857221	857221	0	0
imgui	C++, C	348	348	0	0	-	-	-	-
json	C++	93	93	0	0	93	93	0	0
json-c	C	25	25	0	0	25	25	0	0
keras	P	12110	11860	0	250	12775	12445	10	320
langchain	P	1358	887	0	471	+	+	+	+
libevent	C	68	68	0	0	68	68	0	0
mermaid	JS, TS	3106	3104	0	2	3287	3276	0	11
mpv-player	C	14	14	0	0	-	-	-	-
msgpack-c	C	41	41	0	0	-	-	-	-
mybatis	J	1870	1851	0	19	1870	1851	0	19
nest	JS	1655	1655	0	0	1655	1655	0	0
node	JS, C++, P	4218	4218	0	0	-	-	-	-
numpy	P, C, C++	44861	43197	0	1629	-	-	-	-
opencv	C++, C, P	232	216	0	16	232	216	0	16
openvpn	C	96	95	0	1	91	85	2	4
pandas	P, C	199448	172973	1025	25486	183384	171565	2233	9586
pytest	P	3762	3640	11	111	3805	3775	11	119
react	JS, TS, Rust	5509	5509	0	0	9764	9738	1	25
react-native	C++, J, JS	158	158	0	0	158	158	0	0
rocketmq	J	2336	2324	0	12	2336	2324	0	12
RxJava	J	11863	9404	0	2459	11863	9395	9	2459
scikit-learn	P, C, C++	37064	31900	125	5039	37729	32546	128	5055
scipy	P, C, C++	52672	49745	0	2767	+	+	+	+
spring	J, KT	117	74	0	43	117	74	0	43
tensorflow	C++, P	3148	0	0	0	+	+	+	+
TypeScript	JS	96598	96598	0	0	96598	96598	0	0
vue	TS, JS	1449	1449	0	0	1449	1449	0	0
webpack	JS	28935	28815	0	120	-	-	-	-
webview	C++, C, P	12	12	0	0	-	-	-	-
xgboost	C++, P, R	208	207	0	1	+	+	+	+
xrdp	C, C++	206	206	0	0	+	+	+	+

Prompt for LLM scripts baseline

Create a script that automatically installs a <LANGUAGE> project (on an Ubuntu Linux machine) from source code and runs test cases. The script should account for differences between projects, test frameworks, and dependencies installation. The script should be as general as possible, but should also handle special cases that you are aware of.

Fig. 6. Prompt to generate general-purpose installation scripts ("LLM scripts" baseline).

Input to AutoGPT baseline

You are an AI assistant specialized in automatically setting up a given project and running its test cases. For your task, you must fulfill the following goals:
1. Gather installation-related information and requirements for the project <GITHUB URL>
2. Write a bash script (.sh) that allows to install dependencies, prepare the environment, and launch test case execution.
3. Refine the script if necessary: If an error happens or the output is not expected, refine the script.
4. Once the script launches the test suite successfully, analyze the results of running the test suite to further check whether there are any major problems (for example, some test cases would fail because the project or environment is not well configured, which would mean that the previous goals were not achieved).

Fig. 7. Input given to the AutoGPT baseline.

AutoGPT. Instead of creating a specialized agent for the task of automatically setting up arbitrary projects, one could also use a general-purpose LLM-based agent. We compare against such an agent, AutoGPT[2] [41], which, given a task description, autonomously reasons about a task, makes a plan, executes, and updates the plan over multiple iterations. Similar to ExecutionAgent, AutoGPT may call tools, such as web search, reading and writing files, and executing Python code. As a task description, we provide the input shown in Figure 7. We use the same model and provide the same budget (maximum number of iterations) to AutoGPT as for ExecutionAgent.

Flapy. This baseline is a human-written script written to automatically set up arbitrary Python projects and run their test cases.[3] The script was originally developed as part of a study of test flakiness [12], and by design, is limited to Python projects.

3.2 Effectiveness

ExecutionAgent. The results of applying ExecutionAgent to the 50 projects are reported in Table 1 and summarized in Table 2. ExecutionAgent successfully builds and tests 33 out of 50 projects. Out of the 33 successfully tested projects, ExecutionAgent achieves results identical to the ground truth in 17/33, while the rest (16 out 33) have an average deviation of 15.4%. In Table 2, we consider ExecutionAgent results to be *close to the ground truth* if the average deviation is less than 10%. Overall, these results show that ExecutionAgent is effective at executing the test suites of a large number of projects, and that the results are close to or equal to the ground truth in most cases.

To better understand the results, we analyze the projects by their main language (Figure 8). For each language, we show the number of projects that fail to build, are successfully built, and have their test suites executed. The results show that ExecutionAgent is most effective for Java, where it successfully builds and tests all projects. The two languages that are the most difficult to handle are C and C++, which we attribute to the less standardized build and test processes in these languages, which often requires recompiling packages to be compatible with current system dependencies and project requirements.

---

[2] https://github.com/Significant-Gravitas/AutoGPT

[3] https://github.com/se2p/FlaPy

Table 2. Effectiveness of ExecutionAgent and comparison to baselines.

	ExecutionAgent	**LLM scripts**	**AutoGPT**	**Flapy**
Built	41 / 50	29 / 50	9 / 50	6 / 10
Executed tests	33 / 50	5 / 50	4 / 50	0 / 10
Ground truth deviation < 10%	29 / 50	4 / 50	2 / 50	0 / 10

Another perspective that we analyze is the presence of configurations for automated build tools in the project repositories. Among the 50 projects, 38 use automated build tools, such as Make and Cmake (in 14 projects), Npm and Yarn (13 projects), and Maven/Gradle (9 projects). ExecutionAgent successfully runs the tests of 28 out of the 38 projects that use automated build tools, while the approach succeeds for 5 out of the 12 projects not using such tools. On the one hand, this result shows that the configuration and automation available within the project itself is a factor in the success of ExecutionAgent. On the other side, the result also shows that ExecutionAgent is capable of handling projects that lack such automation.

We also study the presence and impact of submodules. Submodules usually are standalone repositories that are included within the target project as a dependency and should be installed along with the target project. Our dataset has 9 projects that use submodules. We find that ExecutionAgent struggles with such projects, as only 2 out of 9 reach a successful test suite execution.

Finally, we compare the individual test case outcomes between ExecutionAgent and the ground truth. In 17 out of 33 tested projects, the same test sets were executed with matching outcomes. Among the remaining 16 projects, 99.9% of passing test cases aligned with the ground truth, but deviation arose primarily in failing test cases. A detailed analysis of five projects with deviations revealed three key factors contributing to these differences. First, some projects are structured into independent modules with separate builds and test suites. For example, in the *Activiti* project, a failure in compiling the *bpmn* converter module halted the remaining testing process, though an existing option could have allowed the continuation of tests in other modules, which ExecutionAgent did not invoke. Second, incomplete dependency installations may cause test failures or errors. In the *Axios* project, for instance, the absence of a headless Chromium binary resulted in an error, preventing 8 test cases from executing. Third, we attribute certain differences to test flakiness as errors occurred without clear indications of missing dependencies or setup issues.

Comparison with LLM scripts. In Table 2, we compare ExecutionAgent to the three baselines. The general-purpose, LLM-generated scripts successfully build many projects (29/50), but then often often fail to execute the test suites (only 5/29 succeed). Inspecting the results, we find that the LLM scripts often do not account for the differences between the setup required by a regular user and the development setup. The development setup often involves additional steps, such as installing additional software (e.g., a compiler, glibc-32, or a testing framework), recompiling some dependencies, or changing configuration files. In contrast, the usage setup is much simpler, often using an already complete requirement/setup file, such as setup.py for Python and pom.xml for Java. Because the LLM scripts do not account for these differences, they often fail to execute the test suite, whereas ExecutionAgent is able to iteratively fix unexpected errors.

Comparison with AutoGPT. Even though AutoGPT is given the same budget as ExecutionAgent, it builds only 9/50 projects and successfully runs the tests suites for only 4 of them. The results show that the task of running the tests of arbitrary projects is non-trivial, and that a specialized agent, such as ExecutionAgent is more effective at such tasks. Fundamentally, ExecutionAgent differs from AutoGPT by using meta-prompting, by managing the memory of already performed commands more effectively, and by using more sophisticated and robust tools.

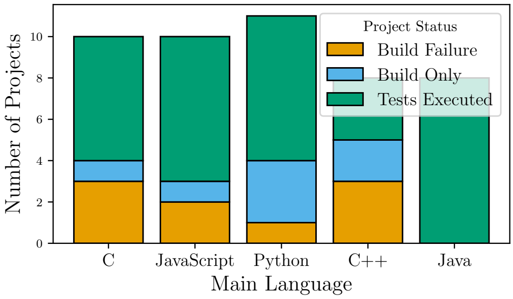

Fig. 8. Effectiveness of ExecutionAgent by main language.

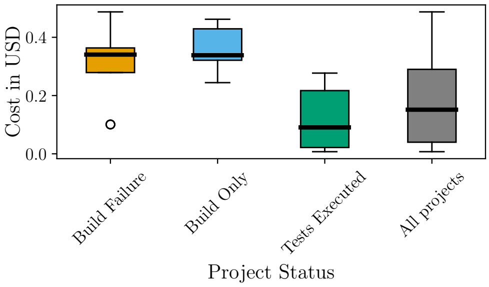

Fig. 9. Monetary costs due to LLM usage.

Comparison with Flapy. The right-most column of Table 2 compares with the Flapy baseline. Because this baseline only targets Python, we apply it only to the 10 projects in our dataset that have Python as their main language. The results show that Flapy reaches results similar to LLM-generated, general-purpose scripts in terms of building projects. However, like the LLM scripts, it also struggles to execute the test suites, and cannot successfully complete the tests of any of the 10 projects. Given the same 10 Python projects, ExecutionAgent successfully builds and tests 6 of them, all with testing results identical to the ground truth.

3.3 Costs

We evaluate the costs incurred by ExecutionAgent in terms of execution time and token usage. The average time required by ExecutionAgent to process a project is 74 minutes (on a 256GB RAM Xeon(R) Silver 4214 machine with five projects being processed in parallel). Given that the most reliable alternative to our approach is to manually set up and test the projects, we consider the time costs of ExecutionAgent to be acceptable.

ExecutionAgent relies on an LLM, which incurs costs computed in terms of token usage. Figure 9 shows the monetary costs due to the usage of the LLM. Based on current pricing, the average cost of processing a project is USD 0.16. The costs differs significantly depending on whether ExecutionAgent succeeds in building and testing the project. For projects that fail to build or test, the costs are higher because ExecutionAgent tries to fix any issues until exhausting the budget, leading to an average cost of USD 0.34. In contrast, for projects where the approach succeeds, the costs are lower, with an average of only USD 0.10. Overall, we consider the current costs to be acceptable given the benefits of the approach, and expect costs to decrease over time as LLMs become more efficient and cheaper.

3.4 Ablation Study

To evaluate the contributions of different components of ExecutionAgent, we conduct an ablation study with multiple variants. The studied variants are as follows:

- **No preparation phase:** The agent starts directly with the feedback phase, lacking the structured information from meta-prompting and retrieval.
- **No feedback phase:** Instead of iteratively refining its approach, the agent generates a single script using the information from the preparation phase.
- **No guidelines:** The agent does not use language-specific or general guidelines, reducing its ability to follow common installation and testing conventions.
- **No external retrieval:** No external information from web search or CI/CD scripts.

Table 3. Ablation study with four variants of ExecutionAgent.

	Built	**Executed tests**	**Deviation < 10%**	**Project cost (USD)**
ExecutionAgent	41	33	29	0.16
No preparation phase	12	8	7	0.19
No feedback phase	19	5	5	0.02
No guidelines	26	15	15	0.15
No CI/Web retrieval	31	24	23	0.18

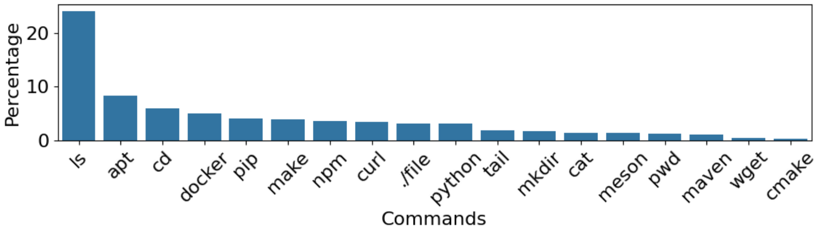

Fig. 10. Distribution of most frequently used terminal commands (./file represents calls to local scripts).

Table 3 presents the results of these variants compared to the full approach. The ablated versions demonstrate significantly lower performance. Removing the preparation phase leads to a notable drop in success rates (12 successful builds and only 8 test executions), highlighting the importance of structured planning and context retrieval. The absence of feedback further reduces effectiveness, as it eliminates the iterative refinement process, leading to only 5 successful test executions. We learn that generated guidelines play a crucial role, as without them, the effectiveness drops by 50%. While retrieving external information (e.g., CI/CD configurations and via web search) are important, removing it only drops the success rate by 21% and increases costs by 13%. Overall, these findings emphasize the importance of both the preparation phase and the feedback phase in ensuring successful project setup and testing.

3.5 Analysis and Discussion

3.5.1 Usage of Tools. To understand the behavior of ExecutionAgent, we start by a quantitative analysis of its tool usage. Overall, 74.8% of calls made by the approach invoke the tool `terminal`, while 16.0% and 8.2% invoke `write_file` and `read_file`, respectively. The "end of task" tool accounts for 1% of all tool invocations. This high-level analysis shows that ExecutionAgent uses all available tools, and that the terminal seems to be indispensible.

To further uncover the commands executed through the terminal, we count the frequency of all invoked commands. Figure 10 shows the results for the top-10 commands. We find that commands for exploring directories and files, such as `ls`, `cd`, and `pwd`, are common, summing up to 33%. Another subset of popular commands (21%) is installation and build commands, such as `apt install` for system packages, `pip` for python and `npm` for JavaScript. Finally, approximately 25% of executed commands fall outside the top 10, demonstrating that ExecutionAgent utilizes a diverse command set, making a hard-coded selection of commands ineffective.

3.5.2 Analysis of Trajectories. We also perform a qualitative analysis of the trajectories of the agent, where "trajectory" refers to the sequence of steps that ExecutionAgent takes when dealing with a specific project. The following describes three observations made during this analysis.

Recurring phases: Setup, dependencies, testing. The trajectories of many projects, particularly software libraries, such as react-native, commons-csv, and pytest, include three clearly identifiable phases: (1) *Setup*: The agent typically starts with preparation commands, such as `ls`, followed by reading necessary files (e.g., `README.md`, `pom.xml`, or `package.json`) to understand the project structure. (2) *Dependency management*: Commands like `write_to_file` for creating Dockerfiles or setup scripts are prevalent. For instance, when creating a Dockerfile, the approach indicates the images to use, e.g., `FROM node:20` or `FROM python:3.10-slim`, followed by invoking additional commands to install dependencies using `apt-get` and package managers like `npm` or `yarn`. (3) *Execution of tests*: Subsequently, the approach issues commands to run tests (e.g., `npm test`, `pytest`) and then confirms that the tests were executed as expected. Successful test execution is typically followed by logging of results, and by creating the scripts that ExecutionAgent is tasked to produce.

Consistent Docker usage. Many trajectories involve using Docker to create isolated environments for applications (e.g., `docker build`, `docker run`). We observe two specific uses of Docker: (1) *Building and running containers*: For projects that provide their own Dockerfile, the agent often uses this Dockerfile to create an environment and install dependencies. (2) *Debugging and validating the environment:* ExecutionAgent uses commands like `docker images` and `docker ps` to check that container images were created and to inspect running containers, showcasing an emphasis on ensuring the runtime environment is correctly configured.

Error handling and resilience. ExecutionAgent demonstrates an ability to handle errors and unexpected outcomes effectively. In particular, we observe the following strategies: (1) *Fallback actions:* Many commands in the produced scripts include fallback logic, such as using `||` to specify alternative actions when a command fails. (2) *Cleanup commands*: Commands are sometimes combined with cleanup actions to ensure that the environment is left in a consistent state, and that commands do not accumulate unnecessary files in case of errors.

3.5.3 Complexity of Scripts Generated by ExecutionAgent. We measure the script size (Section 3.1.2) to assess the complexity of the scripts generated by ExecutionAgent. Across the 33 successful projects, the average script size is 18, with a minimum of 12 and a maximum of 37. That is, the final setup requires running 16 commands, on average. While this number may not seem particularly high, the scripts are concise because the agent creates them once it has found a successful sequence of commands.

3.5.4 Limitations. Through manual trajectory analysis, we identify two behaviors that frequently contribute to not succeeding in executing the tests. First, the agent often repeats the same mistake, leading to wasted commands. A common pattern is attempting to use `sudo` when it is unavailable, resulting in the agent correcting itself after an error, yet repeating the mistake in subsequent commands. Second, ExecutionAgent often fails to follow up on certain commands. For instance, when installing a new version of Node or gcc, the old version frequently remains the default, unless explicitly changed. This oversight leads to repeated cycles of checking versions, installing new ones, and failing to set them as default, causing errors and wasting resources.

4 Threats to Validity

Our results are subject to several threats to validity. First, ExecutionAgent typically runs tests in a single configuration (e.g., one language version, browser, or operating system), which may not capture variations across different environments, such as multiple browsers or OS versions. This limitation can be addressed by allowing users to modify the prompt to specify the desired configurations. Second, the approach is designed around modern technologies (e.g., as asked for in the meta-prompt), which may limit the effectiveness for projects that require older dependencies.

However, this limitation is mitigated by the technique's iterative approach, allowing it to detect and adapt to legacy dependencies when necessary. Third, the approach has been primarily tested on popular projects with relatively good documentation. Results could differ on less well-known or poorly documented projects. In addition, a lot of projects in our dataset do not use submodules, which seem to challenge ExecutionAgent. Finally, our criteria for selecting projects, especially the availability of ground truth test execution results, may bias the dataset toward projects that use some form of CI/CD. Addressing this potential bias would require another way of establishing a ground truth, such as manually performing the task addressed by ExecutionAgent and comparing the results to those produced by the approach.

5 Related Work

Large language models in software engineering. The field of software engineering has seen a rapid increase in the use of LLMs in recent years. Generating code for a given function-level comment has become a standard task to evaluate the capabilities of LLMs [5, 19]. To make code generation practical also for real-world software development, researchers have proposed techniques to augment prompts with repository-level context [7, 9, 30, 44]. Beyond generating application code, LLMs have been used to generate unit tests [10, 17, 18, 23, 27, 28], to translate code from one language into another [26], and to fuzz test programs that accept code as their input, such as compilers [40]. These approaches demonstrate the potential of LLMs to automate software development tasks that require understanding and generating code. In addition to creating new code from scratch, LLMs have been used for modifying existing code. One line of work focuses on predicting code edits based on previously performed edits [13, 37]. Other work tries to automate common code changes, such as refactorings [6]. A team at Meta shows how to use LLMs for augmenting existing, human-written tests [1]. Finally, there is work on multi-step code editing [2], where an LLM first plans multiple edit steps and then performs them one after the other.

Our work differs from all the above by focusing not directly on code, but on the task of setting up and running test suites of software projects. We envision LLM-based techniques for code generation and code editing to benefit from our work by using the executable test suites as a feedback signal to evaluate the correctness of the generated code.

LLM-based agents. Recent work has started to explore the power of LLM-based agents in software engineering tasks. The most prominent agents focus on automated program repair, e.g., in RepairAgent [3] and on automatically addressing issues that describe bugs, missing features, and other improvements of a code base, e.g., in SWE-Agent [42], MarsCode Agent [20], Magis [33], and AutoCodeRover [45]. Another line of work explores an agent trying to describe the root cause of a software failure [24]. We refer to a recent survey for a more comprehensive overview of recent work [15]. To the best of our knowledge, our work is the first to explore the use of LLM-based agents for setting up and running test suites of software projects.

The concept of autonomous LLM agents is, of course, not limited to software engineering. In 2022, researchers proposed to let LLMs generate and execute code to answer a given question [11]. Building on this idea, others propose to augment LLMs with other tools invoked via APIs [22, 29]. The ReAct work [43] shows that LLM agents can outperform LLMs alone in a variety of tasks, e.g., question answering, fact verification, and interactive decision making tasks, such as webpage navigation. Copra is an agent-based approach for formal theorem proving [34]. Two recent surveys give a comprehensive overview of such "augmented LLMs" [21] and LLM agents [36]. Our evaluation compares to a general purpose LLM-based agent, AutoGPT, showing that ExecutionAgent outperforms it in the task of setting up and running test suites of software projects.

Benchmarks that rely on test suite executions. Beyond helping developers, the ability to execute test suites is also essential for researchers. In particular, several popular benchmarks rely on test suite executions, such as Defects4J [16], BugsInPy [39], SWE-bench [14], and DyPyBench [4]. Such benchmarks are used for various purposes, e.g., to evaluate fault localization, automated program repair, and dynamic analyses. ExecutionAgent could help to automate the creation of such benchmarks, by reducing the manual effort required to set up and run test suites.

Automated pipelines for test suite execution. Several research projects explore automated pipelines to execute test suites at a larger scale, either to create benchmarks , e.g., in BugSwarm [35] and GitBug-Java [31], or as part of an empirical study [12]. These approaches each target one programming language, e.g., Java [31, 35] and Python [12], and sometimes rely on projects using a specific CI/CD platform, e.g., TravisCI [35]. Our evaluation empirically compares against language-specific baselines, showing that ExecutionAgent is more effective while supporting multiple languages.

6 Conclusion

In this paper, we introduced ExecutionAgent, an automated technique designed to address the complexities of test suite execution across diverse software projects. By leveraging a large language model-based agent, ExecutionAgent autonomously builds an arbitrary project and runs its tests, producing tailored scripts to streamline future test executions. Our approach is modeled after the decision-making processes of a human developer, using meta-prompting and iterative refinement to adapt to project-specific dependencies and configurations. An evaluation on 50 open-source projects using 14 languages demonstrates that ExecutionAgent successfully executes most test suites and closely matches ground truth results, with clear improvements over existing methods. With reasonable computational and financial costs, ExecutionAgent holds strong potential as a useful technique for developers, autonomous coding systems, and researchers needing reliable test execution capabilities across varied software ecosystems.

7 Data Availability

ExecutionAgent is publicly available at https://github.com/sola-st/ExecutionAgent/ and https://doi.org/10.5281/zenodo.15202434.

8 Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832) and by the German Research Foundation within the DeMoCo and QPTest projects.

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *FSE*, Vol. abs/2402.09171. https://doi.org/10.48550/ARXIV.2402.09171 arXiv:2402.09171

[2] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2023. CodePlan: Repository-level Coding using LLMs and Planning. arXiv:cs.SE/2309.12499

[3] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. Preprint. arXiv:cs.SE/2403.17134

[4] Islem Bouzenia, Bajaj Piyush Krishan, and Michael Pradel. 2024. DyPyBench: A Benchmark of Executable Python Software. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N.

Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. In *FSE*. https://doi.org/10.48550/arXiv.2402.07138

[7] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. CoCoMIC: Code Completion By Jointly Modeling In-file and Cross-file Context. *arXiv preprint arXiv:2212.10007* (2022).

[8] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: A Dynamic Analysis Framework for Python. In *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[9] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *CoRR* abs/2401.01701 (2024). https://doi.org/10.48550/ARXIV.2401.01701 arXiv:2401.01701

[10] Sidong Feng and Chunyang Chen. 2024. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. In *ICSE*.

[11] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. PAL: Program-aided Language Models. *CoRR* abs/2211.10435 (2022). https://doi.org/10.48550/arXiv.2211.10435 arXiv:2211.10435

[12] Martin Gruber and Gordon Fraser. 2023. FlaPy: Mining Flaky Python Tests at Scale. In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 127–131. https://doi.org/10.1109/ICSE-COMPANION58688.2023.00039

[13] Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Language Models Meet Code Edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1483–1495. https://doi.org/10.1145/3611643.3616253

[14] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:cs.CL/2310.06770

[15] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future. *arXiv preprint arXiv:2408.02479* (2024).

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. 437–440.

[17] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*. 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[18] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th International Conference on Software Engineering, ser. ICSE*.

[19] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract-Conference.html

[20] Yizhou Liu, Pengfei Gao, Xinchen Wang, Chao Peng, and Zhao Zhang. 2024. MarsCode Agent: AI-native Automated Bug Fixing. *arXiv preprint arXiv:2409.00899* (2024).

[21] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented Language Models: a Survey. *CoRR* abs/2302.07842 (2023). https://doi.org/10.48550/arXiv.2302.07842 arXiv:2302.07842

[22] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. *CoRR* abs/2305.15334 (2023). https://doi.org/10.48550/ARXIV.2305.15334 arXiv:2305.15334

[23] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. arXiv:cs.SE/2403.16218

[24] Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Henrique B. Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring LLM-Based Agents for Root Cause Analysis. In *Companion Proceedings of the 32nd ACM*

International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024, Marcelo d'Amorim (Ed.). ACM, 208–219. https://doi.org/10.1145/3663529.3663841

[25] Abhik Roychoudhury, Corina Pasareanu, Michael Pradel, and Baishakhi Ray. 2025. Agentic AI Software Engineer: Programming with Trust. *arXiv preprint arXiv:2502.13767* (2025).

[26] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[27] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. In *FSE*.

[28] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Trans. Software Eng.* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[29] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *CoRR* abs/2302.04761 (2023). https://doi.org/10.48550/arXiv.2302.04761 arXiv:2302.04761

[30] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[31] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 118–122.

[32] Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Premkumar Devanbu, and Toufique Ahmed. 2025. Calibration and Correctness of Language Models for Code. In *International Conference on Software Engineering (ICSE)*.

[33] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. *arXiv preprint arXiv:2403.17927* (2024).

[34] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. arXiv:cs.LG/2310.04353 https://arxiv.org/abs/2310.04353

[35] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.

[36] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2023. A Survey on Large Language Model based Autonomous Agents. arXiv:cs.AI/2308.11432

[37] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Coeditor: Leveraging Contextual Changes for Multi-round Code Auto-editing. *arXiv preprint arXiv:2305.18584* (2023).

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[39] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[40] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 126:1–126:13. https://doi.org/10.1145/3597503.3639121

[41] Hui Yang, Sifu Yue, and Yunzhong He. 2023. Auto-gpt for online decision making: Benchmarks and additional opinions. *arXiv preprint arXiv:2306.02224* (2023).

[42] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[43] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations,*

ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[44] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. arXiv:cs.CL/2303.12570

[45] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *ISSTA*.

Received 2025-02-27; accepted 2025-03-31