BuilDroid: A Self-Correcting LLM Agent for Automated Android Builds

Jaehyeon Kim
NYU Abu Dhabi
jk7404@nyu.edu

Rui Rua
NYU Abu Dhabi
rui.rua@nyu.edu

Karim Ali
NYU Abu Dhabi
karim.ali@nyu.edu

***Abstract*—The continuous evolution of the Android ecosystem has led to a highly dynamic and fragmented development environment. This constant churn makes building Android projects, especially from open-source repositories, a notoriously difficult task. Developers and researchers encounter a daunting *build barrier* due to the rapid configuration drift, which results in a cascade of errors. These errors include version incompatibilities, missing dependencies, and inconsistent project configurations, hindering reproducibility and maintainability.**

To address these issues, we present BuilDroid, an LLM-based agent that automates the build process of Android projects. Operating within a self-contained, isolated environment, BuilDroid runs an iterative, self-correcting loop. Through this operation, BuilDroid captures errors and autonomously resolves them, either through predefined heuristics or by leveraging the reasoning capabilities of its underlying LLM.

Across 245 open-source Android projects, BuilDroid effectively resolves complex and evolving build errors, achieving a build success rate of 90.2%, surpassing existing solutions by a margin of over 30.2 percentage points. Consequently, BuilDroid reduces the barrier for researchers and developers, fostering greater software reproducibility and enabling more extensive and reliable empirical research within this rapidly evolving ecosystem.

Video demo: https://youtu.be/YAFLu7NSl5E

***Index Terms*—Software Maintenance, AI for Software Engineering, Android, Empirical Software Engineering**

I. Introduction

As the dominant mobile operating system, Android's rapid evolution poses a significant challenge. While developers gain access to modern Application Programming Interfaces (APIs) and security enhancements, constantly updating Android's Standard Development Kit (SDK), Android Gradle plugin (AGP), and core libraries creates a volatile and fragmented environment. Consequently, a project's build configuration quickly becomes obsolete, posing a continuous maintenance burden for developers. This relentless pace of development environment changes creates a substantial *build barrier* for anyone working with open-source and third-party projects in the wild. Developers aiming to inspect or modify third-party projects often fail to compile them from source code using popular build systems [1]. Similarly, researchers aiming to conduct large-scale empirical studies on Android app repositories [2] face this barrier as well.

A substantial portion of Android projects fail to build out-of-the-box [3], [4], not due to fundamental code errors, but due to configuration drift (e.g., a missing SDK component required by an older API level or a dependency that no longer resolves). Manually diagnosing and fixing these build issues, especially for large corpora of projects, is tedious, time-consuming, and often infeasible, severely limiting the scope and reproducibility of research. For developers, this barrier complicates the process of onboarding to new projects and reusing and contributing to open-source projects. Current approaches that automate this process are insufficient [4], even though most common build issues can be automatically fixed [5]. This is because these approaches typically rely on rigid heuristics [4], [6] that attempt a series of hard-coded fixes. While useful for simple cases, this mechanism is brittle and fails when faced with the long tail of complex, context-specific, or novel error messages that emerge with each new version of the Android toolchain.

To address these limitations, we present BuilDroid, an autonomous agent that automatically diagnoses and repairs build failures in Android projects. BuilDroid adopts a hybrid strategy that combines the flexible reasoning capabilities of Large Language Models (LLMs) [7] with the reliability of rule-based repair heuristics [4]. Operating in an iterative, tool-augmented feedback loop, BuilDroid resolves a wide range of environment, process, and configuration issues to successfully build projects from source. To ground our approach in a practical research context, we have implemented BuilDroid as an open-source tool and have also integrated it into PyAnadroid [4]. Our evaluation on a dataset of 245 open-source Android projects demonstrates that BuilDroid successfully builds 90.2% of the projects, surpassing existing automated solutions by more than 30%.

II. Implementation

To enable reuse by the community, we made BuilDroid available as a Python package [8] that orchestrates an LLM-powered agent that builds Android projects within an isolated and reproducible environment. BuilDroid is model-agnostic, making it easy to use with any cloud-based or local model. Our tool was inspired by the agent-based approach of ExecutionAgent [7]. However, BuilDroid distinguishes itself through three key design decisions:

1) BuilDroid is highly specialized for the Android build domain, a task with a unique and complex set of failures (e.g., AGP, SDK incompatibilities) that general-purpose agents struggle with.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00370

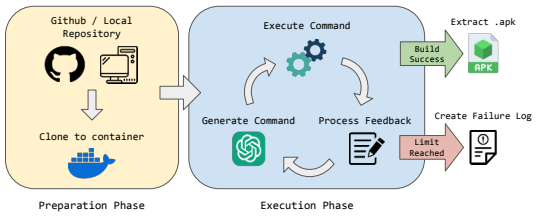

Fig. 1. The high-level workflow of BuilDroid.

2) We designed BuilDroid to be LLM-agnostic, allowing users to easily swap between models to balance effectiveness and costs.
3) BuilDroid's primary goal is producing build artifacts, addressing a different stage of the software lifecycle.

Figure 1 shows the main workflow of BuilDroid, which consists of three phases: a *Preparation Phase* for environment setup, an iterative *Execution Phase* for the build-and-repair cycle, and a *Post-Process Phase* to extract the final artifact.

A. Sandboxed Build Environment

To ensure build consistency and provide environment isolation, BuilDroid operates exclusively within a sandboxed Docker container [9]. Unlike prior solutions [4], [6], this approach guarantees a clean, reproducible environment, which is critical for the validity of large-scale empirical studies and verifiable research. This deep isolation also provides several additional benefits for system integrity such as avoiding polluting the host system with temporary artifacts, cached dependencies, or leftover files. Additionally, using BuilDroid eliminates the risk of making system-level configuration changes that may inadvertently cause conflicts on the host machine.

The default execution environment for BuilDroid is a `Ubuntu 22.04` container, pre-configured with OpenJDK 17, Android SDK Platform 35, and NDK 26.1. A key feature in BuilDroid is its ability to dynamically modify this environment at runtime, for instance by installing different JDK versions or downloading missing SDK components to meet a project's specific requirements.

B. LLM Repair Agent

The core innovation of BuilDroid lies in its autonomous build repair agent, which drives the build process through an iterative feedback loop.

1) Prompt Engineering as the Core Logic: The agent's logic is driven by a dynamic prompt constructed for the LLM in each repair cycle. This prompt provides the full context, including:

- Persona and Goal: an instruction to act as an expert developer with a clear objective.
- Constrained Action Space: a list of available commands, from general file operations to specialized, heuristic-based build fixes.
- Internal Knowledge Base: a set of guidelines for resolving common Gradle errors (e.g., Java version mismatches, dependency conflicts).
- Conversation History: previous commands and verbatim outputs for, providing memory for contextual reasoning.

2) A Hybrid Action Space: A key design choice is our agent's hybrid action space, which combines tool types to improve build success. Low-level, general-purpose tools (e.g., `linux_terminal`) provide the flexibility to address unforeseen errors. These are complemented by high-level, heuristic-based tools such as `dowload_sdk_platform`, derived from prior work [4]. These tools encapsulate proven solutions for frequent build problems, thereby constraining the model to a reliable fix, reducing computational effort and the chance of hallucinating incorrect solutions and monetary costs. For instance, while a heuristic such as `add_google_repo` resolves simple dependency issues, it fails on complex cases (e.g., migrating `net.opacapp:multiline-collapsingtoolbar` to JitPack). For this scenario, BuilDroid leverages the model to reason about the context and apply a context-specific patch, which for this app includes adding the JitPack repository and updating the dependency declaration.

3) Response Parsing and Self-Correction: To translate LLM reasoning into action, BuilDroid instructs its agent to respond in a strict JSON schema containing its *thoughts* and the *command* to execute. BuilDroid's Python backend parses this JSON and dispatches the command to the Docker container. The system includes basic self-correction capabilities. For instance, if the LLM generates malformed JSON or calls a non-existent command, the system returns a `missing_command` error, prompting the agent to re-evaluate and subsequently try a different approach. To prevent infinite loops and manage resource consumption, BuilDroid terminates the execution loop if a build is not successful after a configurable number of iterations (40 by default), a limit determined empirically to balance repair capability and efficiency.

III. Results

To evaluate BuilDroid, we conducted two empirical experiments using a dataset of 245 open-source Android projects, randomly selected from F-Droid [2]. We first sampled 250 candidate repositories by selecting all projects that matched the following criteria: (1) native Android application (exclude libraries and non-Android projects) and (2) publicly buildable repository (no private submodules). After manual inspection we excluded 5 non-native projects and retained 245 projects used in experiments. The complete dataset, along with replication scripts, is available in our online appendix [10].

To evaluate the trade-offs with respect to build success, performance, and cost, we compare using 5 LLMs within BuilDroid: 4 larger popular cloud-based models and a smaller, local model (`Mistral-Nemo-Instruct-2407-4bit`). Table I shows that `GPT-4.1-mini` stands out as the top performer, achieving the highest success rate at 90.2%. This superior capability is coupled with the highest Time To Build (TTB, i.e., the entire time to build a project across all iterations) and Command Count (CC, i.e., number of commands dispatched by the agent during the execution loop), indicating that it successfully resolves more complex projects that require

TABLE I
EVALUATION METRICS FOR USING VARIOUS LLMS WITHIN BUILDROID. ALL REPORTED VALUES REPRESENT AVERAGES CALCULATED FROM SUCCESSFUL BUILDS ONLY

Model	Success Rate	CC	TTB (s)	CPP (US$)
`GPT-4.1-nano`	65.71%	6.90	207.58	0.019
`GPT-4.1-mini`	90.20%	9.04	350.52	0.100
`Gemini-2.0-flash`	76.73%	8.19	242.38	0.035
`Gemini-2.0-flash-lite`	76.32%	5.48	232.38	0.032
`Mistral-Nemo-Instruct-2407-4bit`	25.82%	12.59	667.27	N/A
Average	77.24%	7.40	258.22	0.047

TABLE II
COMPARING BUILDROID TO STATE-OF-THE-ART APPROACHES

			Issues		
Approach	**TTB (s)**	**Success Rate**	**Project**	**Environment**	**Process**
Naive JDK 21	61.0	37.96%	18	105	29
Naive Multi JDK	54.0	59.18%	45	163	29
PyAnaDroid	73.9	60.00%	45	163	29
BuilDroid	227.7	90.20%	67	118	32

additional, time-consuming repair steps. Consequently, it is the most expensive model at US$ 0.10 per project (CPP). Upon further investigation, we discovered that more than 73% of the `GPT-4.1-mini` costs are attributed to input tokens. This suggests that significant cost savings are possible through prompt optimization techniques (e.g., summarizing conversation history or truncating non-essential log outputs), without necessarily sacrificing performance.

For the cloud-based models, Gemini models offer a compelling balance between performance and cost. Both models deliver above-average success rates (>76%) while reducing the CPP by more than 30%. In contrast, `GPT-4.1-nano` is the cheapest cloud-based option but has the lowest success rate (65.71%). For the local model, we observe that it has a notably low success rate of only 25.82%. This evidence suggests that local models may still lack the capability to be effectively applied in the context of automated build repair. The main challenges we observed include frequent hallucinations (such as inventing directory names or command parameters) and a consistent inability to correctly parameterize Linux commands. For example, the model often omitted spaces between command parameters, leading to invalid executions.

We then compare BuilDroid to several alternatives: a baseline solution considering only one Java Development Kit (JDK) Version, a naive multi-JDK solution, and a PyAnaDroid-based solution. We do not consider ExecutionAgent [7], because it cannot build mobile apps without significant changes. The baseline approach using JDK 21 alone involves executing the standard Gradle command to generate the debug build variant. To maximize build success by proactively addressing Java compatibility issues, we also consider a multi-JDK baseline using four Long-Term Support JDK versions (8, 11, 17, and 21). These versions cover a wide range of projects, from legacy applications to modern builds requiring newer AGP versions. PyAnaDroid applies a similar approach, however, it lacks built-in support for dynamic JDK switching. To ensure consistent and comparable results, we executed PyAnaDroid separately under each JDK version. Table II shows the building time and success rate for each approach. We also present results regarding a categorization of the errors detected by each approach following according to the taxonomy proposed by Hassan et al. [5].

Table II shows that the Naive JDK 21 baseline has a poor success rate (37.96%), confirming that relying on a single JDK is insufficient. The Naive Multi-JDK and PyAnaDroid approaches have better success rates (59.18% and 60%, respectively), demonstrating that a multi-JDK strategy is a critical first step for any build repair tool. However, PyAnaDroid only builds two more projects compared to the naive baselines. This marginal improvement highlights the limitations of existing heuristic-based tools. In contrast, BuilDroid achieves a 90.2% success rate, outperforming PyAnaDroid by 30.2 percentage points. Overall, BuilDroid successfully resolves more issues than other tools. In particular, BuilDroid fixes 28% more environment-related issues (e.g., missing SDK components) than multi-JDK approaches. By fixing these initial errors, BuilDroid uncovers and addresses more complex errors. This capability is reflected in the 49% increase in project-level issues fixed (e.g., dependency conflicts). Furthermore, BuilDroid didn't cause more project issues and it was able to reach and resolve them, whereas the other tools failed earlier, evidencing that it possesses the ability to move beyond simple fixes and tackle complex errors that characterize real-world build failures. In our online appendix [10], we provide a detailed categorization of the build errors that we encountered in our experiments. This categorization consists of an extension and specialization of the taxonomy proposed by Hassan et al. [5], as well as a discussion comparing our results with the ones reported in their study.

IV. RELATED WORK

Our research on BuilDroid is situated at the intersection of three key areas: Automated Program Repair (APR), large-scale empirical analysis of Android apps, and the emerging field of AI/LLMs for software engineering.

1) Automated Build Repair (ABR): ABR is a specific subarea of APR focused on violation of system configuration issues, building language grammar, and dependency rules. Prior research on ABR has focused on creating heuristic/rule-based systems to address build failures [4], [11]. Hassan et al. [5] conducted an empirical study on popular build system failures for 200 top Java open-source repositories, concluding that even well-maintained projects trigger build errors (86 projects failed initial build attempts) and that least 57% of the build failures could be automatically resolved. Rua et al. [4] created an Android analysis pipeline with pattern-based rules to fix Gradle [12] build issues, improving success rates by about 20% over naive approaches. Kang et al. [11] proposed Gradle-AutoFix, a machine learning-based approach trained on error message feature vectors, able to identify build failure causes and also suggests fixes, achieving a 64.5% success rate.

State-of-the-art approaches rely on fixed, often outdated heuristics, making them brittle against new or complex errors.

BuilDroid overcomes this by combining rule-based methods with LLMs to interpret diverse error messages and generate context-specific solutions.

2) Large-Scale Empirical Software Analysis: BuilDroid is directly motivated by the needs of researchers conducting large-scale empirical studies on Android. Such studies aim to validate theories, identify trends and patterns, collect datasets, or evaluate tools/techniques. To achieve that, researchers typically rely on open-source repositories [2], [13]. Several frameworks have been developed to leverage such repositories for different types of analyses, including static taint analysis [14] and dynamic analysis through UI exploration [15].

A major challenge is the high failure rate of builds in open-source projects, with Sulír et al. [1] reporting failures in up to 59% of 10,000 Java projects, mainly due to dependency and compilation issues. This limits the effectiveness of many tools, which require buildable apps. BuilDroid tackles this by enabling large-scale automated recovery of broken builds.

3) LLMs for Software Engineering: The application of LLMs to software engineering tasks has recently gained tremendous momentum. Coding assistants (e.g., GitHub Copilot [16]) have demonstrated LLMs' proficiency in code generation. Recent LLM-based agents that interact with software environments to perform complex tasks such as Bouzenia et al. [7] introduced ExecutionAgent, a tool that autonomously executes tests for arbitrary projects by interpreting build instructions and resolving execution failures. Their preliminary evaluation has shown that ExecutionAgent successfully built and executed the test suites of 66% of 50 popular open-source projects. Similarly, SWE-agent [17] explored using LLMs for APR by combining agents and prompt engineering to operate command-line tools and resolve GitHub issues.

Unlike these APR tools, BuilDroid applies LLMs to the build system, focusing on repairing Android configuration and environment issues (prerequisites for compilation and testing). This specialization enables a wide range of downstream tasks reliant on successful builds.

V. Conclusion

In this paper we present BuilDroid, an open-source tool that leverages LLM agents to automate building Android projects. By combining the power of rule-based systems with LLMs reasoning and problem-solving capabilities, BuilDroid successfully builds 90.2% of 245 open-source projects, surpassing existing solutions by more than 30%.

BuilDroid's promising results enable several key applications. For example, it can be integrated into CI/CD pipelines to automatically repair broken builds, accelerating software delivery. Using BuilDroid, researchers can now conduct large-scale empirical studies on Android more reliably, because BuilDroid overcomes the common barrier of build failures. Finally, BuilDroid helps simplify developer onboarding and boost productivity by handling complex build issues.

We plan to develop BuilDroid further by extending support to cross-platform mobile development frameworks. Additionally, integrating other platforms (e.g., iOS) would make BuilDroid more suitable for modern, multi-platform app development. To minimize financial and computational costs, future research could explore the use of smaller, fine-tuned local models as efficient alternatives to cloud-based LLMs, as well as investigate prompt and build error summarization techniques to optimize token usage.

References

[1] M. Sulír, M. Bačíková, M. Madeja, S. Chodarev, and J. Juhár, "Large-scale dataset of local java software build results," *Data*, vol. 5, no. 3, 2020. [Online]. Available: https://www.mdpi.com/2306-5729/5/3/86

[2] F-Droid - Free and Open Source Android App Repository. Accessed: October 3, 2025. [Online]. Available: https://f-droid.org/

[3] R. Rua and J. Saraiva, "A large-scale empirical study on mobile performance: energy, run-time and memory," *Empirical Software Engineering*, p. 67, 12 2023.

[4] ——, "Pyanadroid: A fully-customizable execution pipeline for benchmarking android applications," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2023, pp. 586–591.

[5] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang, "Automatic building of java projects in software repositories: a study on feasibility and challenges," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17. IEEE Press, 2017, p. 38–47. [Online]. Available: https://doi.org/10.1109/ESEM.2017.11

[6] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, "Greendroid: A tool for analysing power consumption in the android ecosystem," in *2015 IEEE 13th International Scientific Conference on Informatics*, 11 2015, pp. 73–78.

[7] I. Bouzenia and M. Pradel, "You name it, i run it: An llm agent to execute tests of arbitrary projects," 2025. [Online]. Available: https://arxiv.org/abs/2412.10133

[8] J. Kim. (2025) Buildroid python package. Last visit: 2025-07-22. [Online]. Available: https://pypi.org/project/buildroid/

[9] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," Linux Journal, mar 2014. [Online]. Available: https://bitly.cx/W44T

[10] J. Kim and R. Rua. (2025) Buildroid online appendix. Last visit: 2025-07-22. [Online]. Available: https://sites.google.com/view/buildroid

[11] M. Kang, K. Taeyoung, S. Kim, and D. Ryu, "Gradle-autofix: An automatic resolution generator for gradle build error," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, pp. 1–21, 05 2022.

[12] Gradle. (2023) Gradle. [Online]. Available: https://gradle.org

[13] P. Liu, L. Li, Y. Zhao, X. Sun, and J. Grundy, "Androzooopen: Collecting large-scale open source android apps for the research community," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020)*, ser. MSR '20. Association for Computing Machinery, 2020, p. 548–552. [Online]. Available: https://doi.org/10.1145/3379597.3387503

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594299

[15] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 23–26.

[16] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, "Practices and challenges of using github copilot: An empirical study," *arXiv preprint arXiv:2303.08733*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08733

[17] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024, pp. 50 528–50 652. [Online]. Available: https://bitly.cx/JpLKu