Automating Android Build Repair: Bridging the Reasoning-Execution Gap in LLM Agents with Domain-Specific Tools

Ha Min Son[1,2] Huan Ren[2] Xin Liu[1] Zhe Zhao[1,*]
[1]Department of Computer Science, University of California, Davis
[2]CodeDroid LLC

Abstract

Android is the largest mobile platform, yet automatically building applications remains a practical challenge. While Large Language Models (LLMs) show promise for code repair, their use for fixing Android build errors remains underexplored. To address this gap, we first introduce AndroidBuildBench, a benchmark of 1,019 build failures curated from the commit histories of 43 open-source Android projects. Each problem is paired with a verified solution from a subsequent commit, ensuring that fixes are feasible. Second, we propose GradleFixer, an LLM agent with domain-specific tools for inspecting and manipulating the Gradle build environment. GradleFixer achieves a resolve rate of 81.4% (pass@1), significantly outperforming a state-of-the-art coding agent that relies on a general-purpose shell. GradleFixer's success suggests that while LLMs possess the high-level knowledge to solve these failures, they struggle to translate this knowledge into effective low-level actions using a general-purpose shell. We demonstrate the effectiveness of a strategy we term *Tool Bridging*, which replaces general-purpose shell commands with domain-aware abstractions. We hypothesize this approach works through two mechanisms: 1) it provides tools in an API-like format that LLMs use more reliably, and 2) it constrains the action space to relevant operations. This approach bridges the gap between the model's high-level reasoning and effective low-level execution.

1 Introduction

Android is the largest and most active mobile development platform in the world, with a market share of 73.9% of the global mobile OS market, over 2.2 million apps on the Google Play Store, and over 1.4 million questions tagged "android" on Stack Overflow (Statista, 2024; StatCounter Global Stats; Stack Overflow). Despite its popularity, a practical challenge remains: *applications cannot reliably be built*. A comprehensive study of 5,222 apps re-
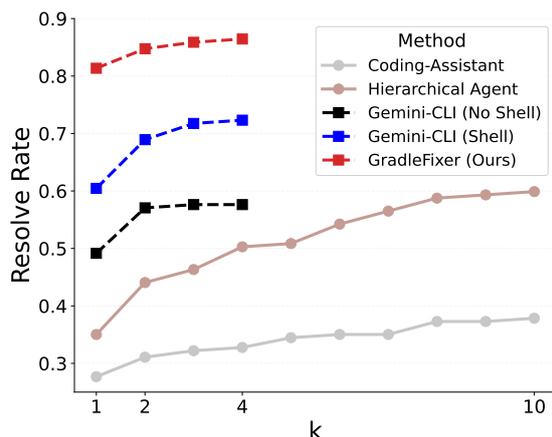

Figure 1: The pass@k resolve rates (percentage of problems solved within k independent sampling attempts) for different agent frameworks on our test set of 184 build errors. We find that replacing the general `shell` with domain-specific tools significantly improves performance.

vealed that only 31.3% could be built successfully out-of-the-box (Liu et al., 2024a). In our dataset, we find the primary source of issues is syntax errors (Table 3a). While Continuous Integration and Continuous Deployment (CI/CD) pipelines are a well-established practice for mitigating these errors (Rostami Mazrae et al., 2023; Baitha et al., 2024; Ghaleb et al., 2024; Liu et al., 2022), Liu et al. (2024a) reported only 21.8% out of the 5,222 apps had CI/CD scripts. Furthermore, even with CI/CD, maintaining a stable build requires significant effort. Hassan et al. (2017) found that over 74% of CI/CD configuration changes are dedicated to build fixing and environment updates. This maintenance is necessary to address a complex range of errors, including syntax mistakes, missing resource files, configuration errors, and library unavailability.

Large Language Models (LLMs) and their application in Automated Program Repair (APR) offer a promising alternative (Zhang et al., 2024a; Yin et al., 2024; Xia et al., 2023; Jin et al., 2023). However, the application of LLMs to Android

build errors remains an underexplored area. Existing LLM coding benchmarks, like SWE-bench (Jimenez et al., 2023), primarily evaluate models on resolving bugs from GitHub issues, not the transient, environment-specific build failures common in Android development. Furthermore, existing datasets of Android build failures (Liu et al., 2024a; Hassan et al., 2017) capture a repository at a single, broken point in time, making fixes difficult to verify as feasible without a corresponding working build to serve as ground truth.

To address these limitations, we make two primary contributions. First, we introduce AndroidBuildBench, a benchmark of real-world Android build errors. AndroidBuildBench contains 1,019 reproducible build failures curated from the commit histories of 43 popular open-source Android projects. Each failure is paired with a verified solution from a subsequent commit, ensuring all problems are feasible to fix. In addition to common human errors and dependency conflicts, our benchmark includes a category of LLM-generated errors to reflect the changing nature of software development.

Second, we propose **GradleFixer**, an agent-based approach for build error fixing using domain-specific tools. Inspired by prior work showing that specialized tools improve agent performance (Singh et al., 2025; Wang et al., 2023) (details in Sec. 5), GradleFixer equips an LLM agent with domain-specific tools, which are simple wrappers for shell commands, to inspect and manipulate the Gradle build environment. Our experiments show that GradleFixer significantly outperforms state-of-the-art baselines across all categories of build failures (Figure 1). We infer from GradleFixer's success that LLMs have the high-level knowledge needed to fix build errors. The poorer performance of shell-based agents, however, suggests this knowledge fails to transfer to effective low-level execution. This gap explains why other effective agents also rely on specialized toolsets (Wang et al., 2023; Singh et al., 2025), although these studies do not explore this choice as a research question. Our approach, which we term *Tool Bridging*, addresses this gap by replacing a general-purpose shell with domain-aware abstractions to bridge high-level reasoning to correct low-level execution. We hypothesize this strategy improves performance through two simultaneous mechanisms. First, by providing API-like tools that abstract away complex command syntax, it allows the model to focus on rea-

Table 1: Summary statistics of the 43 curated Android projects used in AndroidBuildBench.

Metric	Mean ± SD	Median
Stars	3,887 ± 4,702	1,500
Pull Requests	866 ± 1,429	251
Commits	4,258 ± 6,807	1,263
Files	1,041 ± 1,198	646
Lines of Code	117,971 ± 109,442	94,699
Last Commit (Days)	138 ± 332	14

soning rather than implementation. Second, it constrains the action space, reducing irrelevant operations. By reframing the task into a format the model uses more reliably (Liu et al., 2024b; Zhang et al., 2024b, 2025), our strategy connects the LLM's high-level reasoning to effective low-level execution. Our contributions are summarized as follows:

- We introduce **AndroidBuildBench**, a benchmark of 1,019 reproducible Android build errors curated from commit histories. By pairing failing commits with their subsequent fixes, we ensure problems are verifiable and solvable.
- We introduce **GradleFixer** and demonstrate that our strategy of *Tool Bridging*, which replaces general tools with domain-specific abstractions, is highly effective for Android build repair. We hypothesize this approach connects high-level reasoning to correct low-level execution by providing API-like abstractions and constraining the action space.

2 AndroidBuildBench: A Benchmark for Android Build Repair

2.1 Project Curation

AndroidBuildBench is curated from 43 active Android projects on GitHub. These were selected by filtering the top 100 most-starred applications for Java/Kotlin usage, >500 stars, >100 closed pull requests, recent commits, and a Gradle build script. This process yielded 43 open-source projects (Appendix F). A summary of the project statistics is shown in Table 1. We focus on the Gradle build system, as it is used by 95.1% of Android applications according to a recent study (Liu et al., 2024a).

Our approach focuses on individual commits. We are able to explore the specific changes that introduce build failures and the subsequent commits that fix them, which allows us to create problem-solution pairs. Unlike prior work that analyzes repositories at a single point in time without providing solutions, our approach creates verifiable problem-solution pairs.

Table 2: Distribution of errors in the test set compared to the entire dataset.

Error Category	Full Dataset	Test Set
Human-Committed	341	66
Augmented Dependency	486	43
LLM-Generated	192	75
Total Instances	**1019**	**184**
Contributing Projects	43	26

2.2 Build Error Curation

From the 43 curated projects, we created a dataset of 1,019 reproducible build failures. For fine-grained analysis, we created a test set of 184 instances randomly sampled from 26 projects, ensuring a diverse and representative subset suitable for manual inspection. An overview of the dataset distribution is provided in Table 2.

The benchmark contains errors from three curation methods designed to capture real-world build challenges, which are detailed below.

Human-Committed Errors. This category captures the transient build failures that naturally occur during the software development lifecycle. We identify merged pull requests (PR) that successfully build, then analyze their commit history. If an intermediate commit within the PR fails to build, we classify this failing commit as a problem instance. The corresponding solution consists of the changes that lead to the final, successful PR.

Augmented Dependency Errors. To simulate configuration issues, we start with a successful commit and revert only its build-related file changes (e.g., build.gradle, Manifest.xml). A build failure resulting from this change simulates a common scenario where source code becomes out-of-sync with its build environment or dependency declarations. This process generates a targeted problem instance where the solution is known to be contained within the reverted configuration changes.

LLM-Generated Errors. To capture failures from AI-assisted development, we revert a successful commit and prompt an LLM with the commit message and the release notes to re-implement the functionality. If the code generated by the LLM fails to build, we include this failure in our benchmark. The successful human-written commit is the ground-truth solution, reflecting a realistic scenario where a developer might need to fix an imperfect, AI-generated patch. This data generation process is also highly scalable, allowing us to bootstrap a large number of diverse LLM-generated failures.

2.3 Error Categorization

We manually analyzed the 184 test set issues, adopting the five build failure categories proposed by Liu et al. (2024a).

- **Syntax Error:** errors within the Java or Kotlin source code. These are fundamental language mistakes, such as a missing semicolon or an incorrect keyword, that cause compiler errors.
- **Resource File Missing:** errors when the build process cannot find a required file. These files, such as keystore.properties or google-services.json, may be missing unintentionally or deliberately excluded from version control to protect confidential information.
- **Configuration Error:** errors from incorrect settings in project configuration files, most commonly build.gradle. A typical cause is a hard-coded, environment-specific path (e.g., to the Android SDK) that does not exist on the current build machine.
- **Library Not Available:** errors when the build system is unable to locate and download a specified dependency library. This can happen if a library has been removed from its repository, the repository URL is invalid, or the specified version is obsolete.
- **NDK Error:** errors in projects that use the Android Native Development Kit (NDK) to include C/C++ code. Errors often involve misconfigurations, such as a mismatch between the NDK version required by the project and the one installed, or build script issues related to changes in the NDK package structure over time.

Our analyses are presented in Table 3a. A notable finding is the absence of NDK errors. We posit this absence results from our curation method, which is anchored to successfully built pull requests. NDK errors often result from build machine environment incompatibilities. Since our method requires a successful build as an endpoint, it naturally filters out states with systemic environment issues. The errors captured are those transiently introduced and fixed within a pull request, a scope more representative of the typical software development life cycle. Furthermore, we find that syntax errors are the most common failure type. This is surprisingly high for human-committed errors, given that modern IDEs typically provide real-time syntax checking. For LLM-generated errors, this result is less surprising, as current models are known to be prone to syntactical mistakes (Tambon et al., 2025).

Table 3: Analysis of build failures and code changes by origin.

(a) Root Cause Analysis of Failures.

Root Cause	Human	Dep.	LLM	Total (%)
Syntax code	35	28	47	110 (59.8%)
Resource file missing	0	5	7	12 (6.5%)
Configuration error	27	5	14	46 (25.0%)
Library not available	4	5	7	16 (8.7%)
NDK error	0	0	0	0 (0.0%)
Total	**66**	**43**	**75**	**184 (100%)**

(b) Statistics of Code Changes.

Metric	Stat.	Human	Dep.	LLM
Files Changed	Mean	5.8	14.3	6.7
	Median	3.0	14.0	4.0
	Std Dev.	4.8	9.4	7.0
Lines Changed	Mean	258.6	436.9	274.1
	Median	66.0	229.0	81.0
	Std Dev.	467.0	519.1	457.9

Furthermore, we analyzed the statistical properties of the code changes, as shown in Table 3b. We find that Augmented Dependency Errors involve substantially larger code changes. Our curation method for these errors selects commits with build file modifications, which often correlate with more substantial source code changes. As we demonstrate in our results (Table 8), the magnitude of the code change (i.e., files and lines modified) that led to the error is a reasonable proxy for the difficulty of the repair task, rather than the category of the error. A full distribution is in Appendix D.

3 Experimental Setup and Approaches

3.1 Motivation: Providing the Right Tools

LLMs demonstrate strong proficiency with common shell commands and can often infer how to use novel tools from their descriptions (Patil et al., 2025; Zeng et al., 2025; Team, 2025). However, our preliminary analysis revealed a critical limitation when applying this general capability to a specialized domain like Android build repair. Although an LLM may possess the high-level knowledge to understand a build failure, it struggles to translate this knowledge into the correct sequence of low-level shell commands. For example, correctly synthesizing and sequencing a complex command like `./gradlew assembleDebug` along with necessary environment variables is highly error-prone compared to the high-level decision of when to run a build.

We find this gap between high-level reasoning and low-level execution (Tables 4 & 6) likely occurs because while common shell commands (e.g., `ls`, `grep`) are prevalent in training data (Liu et al., 2024b; Patil et al., 2024), the multi-step sequences required for domains like Android builds are less common. An agent with a general shell often knows the relevant commands individually but fails to apply them in the correct context, leading to inefficient trial-and-error loops. This phenomenon

is analogous to providing a specialist with a curated set of instruments rather than an exhaustive, unorganized toolkit.

Our approach addresses this through *Tool Bridging*, which replaces general-purpose tools with domain-aware abstractions. These specialized tools translate shell commands into high-level, API-like actions, presenting a format that LLMs handle more reliably and constraining the action space (Patil et al., 2025). This design allows the model to focus its reasoning on debugging rather than command synthesis, bridging the gap between conceptual understanding and correct execution.

3.2 Execution Environment

All experiments were conducted on a standardized Linux machine (256 CPU cores, 512 GB RAM) installed with the Android SDK. To accommodate the diverse requirements of different projects, we installed the major versions of the Java Development Kit (JDK), including versions 11, 17, 20, 21, 22, and 23. Following the methodology of prior work (Liu et al., 2024a), we build each application using the command `./gradlew assembleDebug --parallel`. Building Android applications is computationally intensive, with build times for some projects extending up to 15 minutes on a 256-core machine. We containerized each build process within an isolated Jupyter Notebook (Kluyver et al., 2016). This approach enabled us to run multiple tasks concurrently, significantly improving experimentation throughput. Additionally, to ensure a clean state between runs, we run `./gradlew clean --stop` before attempting to build, preventing cached states from affecting the current build.

3.3 Approaches

We evaluate the effectiveness of an agent-based approach to repairing Android build errors by comparing several configurations designed to isolate the impact of domain-specific tooling. All of our agent configurations use Gemini-2.5-Pro (Comanici et al.,

Table 4: Resolve rate of agent configurations on the AndroidBuildBench test set, measured by Pass@k success rates (%). Our method, **GradleFixer**, consistently outperforms all baselines across all failure categories. Best results in each column are in bold. (LLM calls are not budgeted.)

	Human Commit			Dependency			LLM-Generated		
Method	P@1	P@2	P@4	P@1	P@2	P@4	P@1	P@2	P@4
Coding-Assistant	30.2	32.6	32.6	19.4	19.4	20.9	36.0	42.7	44.0
Hierarchical Agent	27.9	41.9	51.2	37.9	40.9	47.0	36.0	49.3	53.3
Gemini-CLI (No Shell)	39.5	46.5	48.8	42.4	47.0	47.0	60.0	70.7	71.7
Gemini-CLI (Shell)	65.1	69.8	79.1	40.9	50.0	50.0	72.0	81.3	82.7
GradleFixer (Ours)	**84.1**	**90.9**	**90.9**	**77.8**	**79.4**	**82.5**	**82.3**	**84.8**	**85.9**

2025) as the core LLM.

As a baseline representing a state-of-the-art coding agent, we use the official, unmodified Gemini-CLI open-source implementation (Google Cloud). To allow for controlled experiments, we also developed a validated Python replica of the Gemini-CLI agent (Appendix A). This replica preserves the original agentic loop and prompting (Appendix B) but allows us to easily customize the available toolset. We use this replica for the **Gemini-CLI (Read/Write Only)** configuration and our proposed method, **GradleFixer**. We evaluate the following five distinct approaches:

- **Coding-Assistant:** A baseline that simulates a user-in-the-loop assistant. The agent is provided with file context and an error log but can only propose code modifications. It cannot execute commands, browse the file system, or run the build. We use the open-source Aider (Aider) framework for this configuration.

- **Hierarchical Agent:** A two-agent framework where a primary LLM agent reasons about the problem and then delegates code editing tasks to the **Coding-Assistant** by invoking it as a tool, similar to Ishibashi and Nishimura (2024); Liu et al. (2024c).

- **Gemini-CLI (Read/Write Only):** A baseline configuration (using our Python replica) where the agent has the ability to read and write files but is denied access to the shell. This configuration tests the agent's ability to solve build errors with a limited toolset.

- **Gemini-CLI (Read/Write and Shell):** The standard open-source Gemini-CLI agent (Google Cloud). *We use the unmodified open-source version for this baseline.* It is given file read/write capabilities and a general-purpose shell tool, allowing it to execute arbitrary commands.

- **GradleFixer (Our Method):** Our proposed method, which excludes the general-purpose shell but is given a set of domain-specific tools

designed for the Android build environment.

3.4 GradleFixer Domain-Specific Tools

The standard tools available to agents, as in Gemini-CLI, include the following file system commands: `ls`, `grep`, `glob`, `read_file`, and `replace_string`. Agents can also use a `search_google` tool, which queries Google and returns summarized findings. Our method adds three domain-specific tools, replacing the general `shell` tool:

- `run_build`: Executes the standard build (`./gradlew assembleDebug`) and returns the raw output.
- `run_gradle`: A more general tool that can execute any Gradle command (`./gradlew <task>`). This tool is a superset of `run_build`.
- `change_java_version`: A dedicated function that switches the active JDK in the environment (e.g., `export JAVA_HOME=...`).

All three tools are intentionally designed as wrappers for specialized shell commands. The outputs of these tools are not specially structured or processed. This design ensures the output is identical to that of a raw shell command, thereby isolating the impact of the tool invocation itself from any benefit of structured output. Full tool descriptions are available in Appendix C.

4 Results and Discussions

The performance of each agent configuration on the AndroidBuildBench test set is shown in Figure 1 and Table 4. Our proposed method, **GradleFixer**, significantly outperforms all baseline configurations across all categories of build errors.

The improvement of Gemini-CLI over *Coding-Assistant* and *Hierarchical Agent* confirms that Android build repair is not a localized code-fixing task. It is a holistic problem requiring an agent to explore the repository, inspect files, and interact with the build system. Additionally, we find that Gemini-CLI (Shell) outperforms *No Shell*. Without a shell,

Table 5: Performance of domain-specific tool combinations. The agent's available tools are grouped by category. *(LLMs are limited to **30 calls** for ablations resulting in slightly lower resolve rates.)*

Method / Tool Set	Pass@1
No shell (Baseline)	49.2%
Individual Tools (w/ Incr. Specificity)	
Only shell	54.3%
Only Gradle (run_gradle)	55.8%
Only build (run_build)	63.4%
Tool Combinations	
Build + Gradle	69.7%
Build + Java	69.8%
Shell + Build + Gradle + Java	70.7%
Build + Gradle + Java (Ours)	**74.0%**

Table 6: Tool usage frequency (%) across agent configurations. Agents are: **Base** (no shell), **Shell** (base with shell), **Gradle** (Only gradle), **Build** (Only Build), and **Ours** (Gradle, Build, and Java).

Tool	Base	Shell	Gradle	Build	Ours
LS	13.0	1.0	1.9	1.1	3.2
Grep	3.2	25.7	1.3	1.3	1.4
Glob	9.6	1.3	2.2	2.5	1.8
Read File	35.1	19.0	32.0	32.2	33.6
Replace	39.1	5.9	36.3	39.1	41.2
Google Search	0.0	0.1	1.7	3.6	1.6
Shell (Total)	-	**47.0**	-	-	-
Change Java	-	13.3*	-	-	0.6
Gradle	-	4.8*	24.7	-	1.8
Build	-	20.7*	-	20.2	14.8

* The agent calls these tools using the shell. Percentages for the tools are normalized to represent their fraction of **all** calls for direct comparability.

the agent cannot execute Gradle commands, including building the app to inspect the current state, or modify environmental variables like JAVA_HOME, which are important for resolving errors.

While the open-source Gemini-CLI already abstracts some shell commands (e.g., ls, grep) into dedicated tools, our method takes this direction further. We find that replacing the generic shell with the domain-specific tools run_build, run_gradle, and change_java_version, leads to a dramatic performance improvement. The most substantial gain was on Dependency issues, which is where Gemini-CLI (Shell) struggled the most.

4.1 Impact of domain-specific tools

We study the impact of the domain-specific tools, which are all abstractions of shell commands. We evaluated agents equipped with different tool combinations, from the most general (shell) to the most specific (run_build). The results are shown in Table 5. For ablations, LLMs are budgeted to 30 calls per task.

Our results show a clear trend of **improved performance as tools become more specific**. The agent with only the shell tool improves over the baseline. However, performance increases with the more abstract run_gradle tool and is highest with the most specific tool, run_build. This result is particularly notable because shell is a superset of run_gradle, which is a superset of run_build. The agent is most effective when given the most constrained, highest-abstraction tool, suggesting the performance gain comes from focus, not just capability. Performance further improved as tools were combined, with our full method (Build+Gradle+Java) achieving the highest resolve rate and demonstrating the value of well-designed

shell abstractions. Furthermore, our full method, which excludes the shell, outperforms the combination that includes it (74.0% vs 70.7%), suggesting that a constrained and relevant toolset is more effective than an unrestricted one. These observations align with findings in Wang et al. (2023), where specialized toolsets enable continuous performance improvement while more generic agents plateau.

These results validate our *Tool Bridging* strategy, which bridges the gap between an LLM's high-level knowledge and its low-level execution. The shell-based agent's behavior suggests such a gap exists. Table 6 shows it frequently attempts the correct raw commands for building (20.7%) and changing Java versions (13.3%), yet its low pass rate (Table 4) demonstrates it struggles to apply them effectively in a raw shell. This pattern suggests the agent possesses the high-level knowledge of *what* operations are needed but struggles to *correctly sequence and apply* those operations. We illustrate these patterns in our case studies where the shell-based agent becomes trapped by misleading logs or reverts correct fixes (Appendix E).

Tool Bridging addresses this gap by providing domain-aware abstractions. By replacing the general shell with specific tools like run_build, we reframe complex shell interactions as simple, API-like calls. This aligns with the format LLMs are trained to use reliably (Patil et al., 2025) and constrains the action space to relevant operations. This allows the model to leverage its high-level reasoning without failing on low-level command syntax and sequencing. Our results support this hypothesis. As shown in Table 5, performance improves as tools become more specific, suggesting LLMs

Table 7: The Pass@1 resolve rate comparison for Gemini-2.5-Pro and GPT-5-Mini (30 LLM calls per task).

Method	Gemini	GPT
Shell Agent (Gemini-CLI)	54.3%	41.5%
+ Tool Usage Guidance	58.9%	44.1%
GradleFixer (Ours)	74.0%	59.6%

know which commands are necessary but struggles to sequence them correctly as tools become less specific. GradleFixer achieves superior performance while calling `change_java_version` far less frequently (0.6%), suggesting effective targeted tool use rather than exploratory use.

In contrast to a general-purpose shell, a dedicated tool like `run_gradle` could be priming the LLM to Android knowledge not only through its API-like structure but also through its name and description. This explicit contextual signal appears to help the model more reliably apply its latent knowledge of Android builds, steering it away from the incorrect interactions with a generic shell, as demonstrated in our case studies (Appendix E).

4.2 Prompting vs. Dedicated Tools

Given that our domain-specific tools are specialized shell commands, we studied the impact of dedicated tools over prompting methods. We compared the standard shell-enabled Gemini-CLI agent to a version with a "Tool Usage Guidance" prompt that explicitly instructed the LLM on how to perform the build-related tasks using the shell. We added the following instructions to its system prompt:

> **Tool Usage Guidance:**
> - You can change the Java version by setting the environment variable: `export JAVA_HOME=.../[JAVA_VERSION]/lib/jvm`
> - You can run specific Gradle tasks using the shell command: `./gradlew [TASK_NAME]`
> - Once you apply a fix, you can test it using the shell command: `./gradlew assembleDebug`

We also include an evaluation using GPT-5-Mini to show that our finding is not limited to a single family of LLMs. While OpenAI provides its own coding agent, Codex (OpenAI, 2025), it is not open-source, so its exact toolset cannot be reproduced. We instead use the same tools as Gemini-CLI. The results are shown in Table 7.

We find that providing a tool usage guidance prompt moderately improves performance over the baseline. This shows that guiding the LLM toward important tool use is an effective strategy. However, the prompted agent still performs significantly

worse than our proposed agent, GradleFixer, which is equipped with domain-specific tools (Build, Gradle, and Java). This performance gap suggests that **providing domain knowledge through dedicated tools is more effective** than providing it through prompts for a general-purpose tool. We hypothesize this is because dedicated tools present operations in a more reliable, API-like format and constrain the action space, which more effectively connects the agent's high-level reasoning to the low-level execution required for builds.
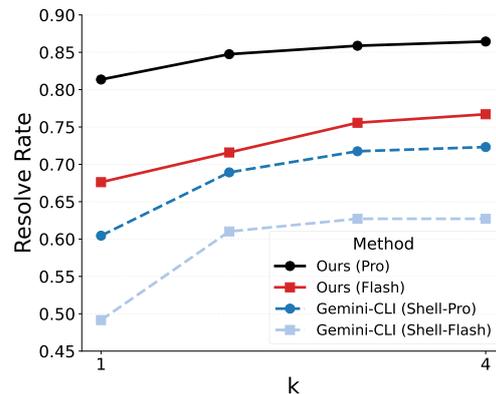

Figure 2: The pass@k resolve rates with a comparison for model sizes. We find that our method, using a smaller model outperforms Gemini-CLI using a larger model, supporting the importance of domain-specific tools on performance.

4.3 Model Size and Cost-Effectiveness

To study the effect of model size within the same family of LLMs, we evaluated both Gemini-2.5-Pro and the smaller Gemini-2.5-Flash model. We tested both LLMs using our GradleFixer method and the standard Gemini-CLI (Shell) agent. The results are shown in Figure 2. We find that GradleFixer with the smaller Flash model outperforms the standard Gemini-CLI agent using the larger Pro model.

This finding suggests that providing useful domain-specific tools can be more impactful than using a larger, more capable model, with significant implications for cost-effectiveness. The Gemini-2.5-Flash model costs less than a quarter of the price per token compared to the Pro model. The cost savings are further increased due to the nature of agent-based repair. When an agent fails, it continues attempting different strategies, consuming many more tokens. In our experiments, successful repairs with GradleFixer averaged 1.1 million input and 10,265 output tokens over 10.6 LLM calls. By contrast, failed attempts consumed nearly four times as many input tokens (4.1 million) and

Table 8: Breakdown of success and failure cases. For each metric, we show the performance of Gemini-CLI and our method, GradleFixer. Statistics of the change that led to the failure are reported as Median ± Std Dev.

Metric	Gemini-CLI	GradleFixer
Root Cause Category (Success Rate)		
Syntax (%)	78.4	91.0
Resource File (%)	92.3	100.0
Configuration (%)	57.8	82.2
Library (%)	88.2	88.2
Change that led to build error (Fixed Cases)		
Lines Changed	80.0 ± 332.6	75.0 ± 336.3
Files Changed	5.0 ± 7.8	4.0 ± 7.4
Change that led to build error (Failed Cases)		
Lines Changed	232.0 ± 677.0	714.5 ± 671.0
Files Changed	8.0 ± 7.4	15.0 ± 6.6

28,109 output tokens over 27.5 LLM calls. Because a more performant method fails less often, it is also more economical.

This principle also supports practical cost-saving strategies like a *cascading* agent design, where tasks are first attempted by a smaller, cheaper model and only escalated to a larger one upon failure. More broadly, this finding points toward a promising future where smaller, specialized models could be fine-tuned to achieve high performance on domain-specific tasks, potentially exceeding the capabilities of larger generalist models at a much lower cost (Belcak et al., 2025).

4.4 Analysis of Failure Cases

While our primary results (Table 4) suggests Dependency-Augmented errors are the most challenging category, a more detailed analysis reveals that the magnitude of the code change leading to the error is a stronger predictor of failure. Table 8 provides an overview of the performance by error category, and the statistics of code changes for both successful and failed attempts.

The substantially higher median lines and files changed for failed cases in both agents confirms that larger changes are inherently more difficult to fix. This trend is more pronounced for GradleFixer, where the median lines changed in failed cases is nearly ten times higher than successful cases (714.5 vs. 75.0). This shows that while GradleFixer mitigates challenges from different error types, its remaining failures are concentrated in tasks with extensive code modifications, regardless of cause.

This correlation is interesting because the agents do not have access to the original diff that intro-

duced the error. We hypothesize that larger changes are more likely to introduce compounding errors. An initial error can mask subsequent ones, forcing the agent to solve a series of problems iteratively, which significantly increases the task complexity. For example, one case study found that fixing a data binding error revealed a second build configuration issue (Appendix E.2). This finding suggests a clear best practice for developers using automated repair: *building frequently after small, incremental code changes creates an environment where these agents are most likely to succeed, maximizing both effectiveness and cost-efficiency.*

4.5 Analysis of Tool Semantics

To study how performance is impacted by semantic priming of tool names (e.g., run_build) and the functional abstraction of the tool itself, we conducted an ablation study masking tool definitions. We evaluated versions of GradleFixer where tool names were replaced with generic labels (e.g., tool_A) and descriptions had keywords similarly masked.

As shown in Table 9, masking tool names resulted in a negligible performance drop, whereas masking descriptions caused a larger drop. However, even with names and descriptions masked, the API-like tool structure achieves a pass@1 rate of 64.9%, outperforming the shell-based agent and the shell agent with prompt guidance. This indicates that while semantics (particularly descriptions) provide a benefit, the structural constraint of the action space and the reliability of the API-like format also provides significant performance gains.

Table 9: Ablation study disentangling tool semantics from structure. Even with masked names and descriptions, the specialized tool structure outperforms the generic shell.

Method	Pass@1
Shell Agent (Baseline)	54.3%
Tool Usage Guidance Prompt	58.9%
GradleFixer (Masked Names)	73.5%
GradleFixer (Masked Descriptions)	63.4%
GradleFixer (Masked Names & Desc.)	64.9%
GradleFixer (Full Semantics)	**74.0%**

5 Relation to Prior Work on Domain-specific Tools

Our primary contribution is the empirical validation of two agent design insights. First, replacing a general-purpose shell with a curated set of

domain-specific tools improves performance. Second, this performance gain increases as the specificity of the tools increases (Table 5). We term this strategy *Tool Bridging*.

While we were inspired by prior studies that successfully use domain-specific tools (Wang et al., 2023; Singh et al., 2025), their focus differs from ours. In prior work, domain-specific tools are a component for achieving a broader goal, such as lifelong learning or deep code analysis. In contrast, our work isolates the impact of domain-specific tools as the central research question and provides insights through empirical evidence.

Distinction from Voyager (Wang et al., 2023). The Voyager agent is an influential paper for lifelong learning within an embodied environment (Minecraft). It operates by building a growing skill library from executable JavaScript functions which call predefined, domain-specific APIs like `mineBlock(bot, name)`. These APIs form the agent's action space.

The key difference is that their contribution is not an analysis comparing domain-specific tools against general tools. The novelties are the learned skill library and iterative prompting. The specialized APIs are a prerequisite for the agent to function. However, the paper does not analyze settings where the agent uses a general Minecraft command-line shell, nor does it test if more specific tools would further improve performance. In contrast, we directly compare an agent with a general `shell` against one with domain-specific tools (GradleFixer), attributing the performance gain to the *Tool Bridging* strategy.

Distinction from Code Researcher (Singh et al., 2025). Code Researcher is a powerful agent designed to patch crashes in complex Linux systems code. It uses specialized tools for code analysis, such as `search_code(regex)` and `search_commits(regex)`, which are domain-specific for code and history investigation. Similar to Voyager, the paper's contribution is the deep research methodology. The authors do not investigate whether `search_commits` is superior to `git log`, but rather how an agent can use history-aware tools to solve complex bugs. Conversely, our work focuses directly on this comparison. We provide evidence that abstracting complex commands into a constrained, API-like action space is a key strategy for improving agent performance in the Android build domain.

This distinction also explains a key aspect of our experimental design. The goal of our agent is to repair a build error, not simply to identify the commit that introduced it. Because our dataset consists of build-breaking commits, a history-aware tool would allow the agent to solve the problem by reverting the change. We excluded this capability to keep the task focused on code repair.

6 Conclusion and Future Work

In conclusion, to address the high frequency of build failures in Android development, we introduced AndroidBuildBench, a benchmark of real-world build errors, and GradleFixer, an LLM agent with domain-specific tools. Our results demonstrate that providing specialized tools is more effective than relying on a general-purpose shell for Android build errors. GradleFixer significantly outperformed state-of-the-art baselines, and we found that a smaller, more cost-effective model with our toolset outperformed a larger model without it. The effectiveness of *Tool Bridging* also offer a potential explanation for the strong performance of agents in other domains that use special API-like tools (Wang et al., 2023; Singh et al., 2025). Our work provides the empirical evidence for this design pattern, suggesting it is a generalizable approach for building capable LLM agents.

This work suggests several directions for future research. First, fine-tuning smaller, cost-effective language models on domain-specific datasets like AndroidBuildBench using specialized tools could exceed the performance of larger models. Second, the *Tool Bridging* strategy could be applied to other domains. A compelling research direction is developing agents that automatically generate and refine their own domain-specific tools from experience, allowing them to adapt to specialized tasks without manual tool engineering. Finally, by automating build fixing, our approach could lower the barrier for Android development. This may enable a more fluid and exploratory style of development, such as "vibe-coding," where non-developers and LLMs can experiment with less fear of breaking the build.

Limitations

Our study has several limitations. First, AndroidBuildBench is curated from 43 popular, open source Android projects. This selection may not fully represent the diversity of applications, particularly private or less actively maintained projects. Our curation method, which anchors failures to successfully resolved pull requests, also filters out

certain persistent environmental issues, such as the NDK errors noted in our analysis, and may therefore underrepresent some categories of build errors.

Second, the GradleFixer agent and its tools are designed for the Android Gradle environment. While we propose *Tool Bridging* as a general strategy, its effectiveness in other development ecosystems, like iOS or web development, has not been tested. Applying this strategy to new domains would require designing a new set of domain-aware tools.

Finally, our explanation for the success of *Tool Bridging* is a hypothesis supported by strong empirical results. Proving the exact internal reasoning of the language model remains a significant challenge in AI research, and our study does not address the deeper, mechanistic question behind our central finding. For instance, inspired by recent work on the internal mechanisms of LLMs (Lindsey et al., 2025), a deeper investigation could probe whether a dedicated tool like run_build activates a different, more specialized "computational circuit" within the model compared to a generic shell command. Such an analysis would be necessary to connect the behavioral success we observed with its underlying cognitive processes. Therefore, our claim is an inference based on observed performance rather than a direct measurement of the model's cognitive process. Furthermore, our experimental design deliberately excluded access to commit history to create a focused repair task. This differs from a real world scenario where a developer might use version history to diagnose a problem.

References

Aider. Aider - ai pair programming in your terminal. https://aider.chat/. (n.d.). Accessed: 2025-09-24.

Sanjay Baitha, Vijayakumar Soorya, Osho Kothari, Shinu M Rajagopal, and Niharika Panda. 2024. Streamlining software development: a comprehensive study on ci/cd automation. In *2024 4th International Conference on Sustainable Expert Systems (ICSES)*, pages 1299–1305. IEEE.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small language models are the future of agentic ai. *Preprint*, arXiv:2506.02153.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Taher Ghaleb, Osamah Abduljalil, and Safwat Hassan. 2024. Ci/cd configuration practices in open-source android apps: An empirical study. *ACM Transactions on Software Engineering and Methodology*.

Google Cloud. Gemini CLI. https://cloud.google.com/gemini/docs/codeassist/gemini-cli. Accessed: 2025-09-23.

Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 38–47. IEEE.

Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183*.

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 preprint arXiv:2310.06770*.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1646–1656.

Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, and 1 others. 2016. Jupyter

notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, and 8 others. 2025. On the biology of a large language model. *Transformer Circuits Thread*.

Pei Liu, Li Li, Kui Liu, Shane McIntosh, and John Grundy. 2024a. Understanding the quality and evolution of android app build systems. *Journal of Software: Evolution and Process*, 36(5):e2602.

Pei Liu, Xiaoyu Sun, Yanjie Zhao, Yonghui Liu, John Grundy, and Li Li. 2022. A first look at ci/cd adoptions in open-source android apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, and 1 others. 2024b. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Yuchi Liu, Jaskirat Singh, Gaowen Liu, Ali Payani, and Liang Zheng. 2024c. Towards hierarchical multi-agent workflows for zero-shot prompt optimization. *arXiv preprint arXiv:2405.20252*.

OpenAI. 2025. Introducing codex. https://openai.com/index/introducing-codex/. Accessed: 2025-09-25.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. 2023. On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. *Empirical Software Engineering*, 28(2):52.

Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2025. Code researcher: Deep research agent for large systems code and commit history. *arXiv preprint arXiv:2506.11060*.

Stack Overflow. Newest "Android" questions. https://stackoverflow.com/questions/tagged/android?sort=Newest&edited=true. Accessed: 2025-09-23.

StatCounter Global Stats. Mobile Operating system market share worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2025-09-23.

Statista. 2024. Google Play: number of available apps as of Q2 2024. https://www.statista.com/statistics/289418/number-of-available-apps-in-the-google-play-store-quarter/. Accessed: 2025-09-23.

Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65.

The Terminal-Bench Team. 2025. Terminal-bench: A benchmark for ai agents in terminal environments.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE.

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1274–1286.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024a. A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466*.

Shaokun Zhang, Yi Dong, Jieyu Zhang, Jan Kautz, Bryan Catanzaro, Andrew Tao, Qingyun Wu, Zhiding Yu, and Guilin Liu. 2025. Nemotron-research-tool-n1: Exploring tool-using language models with reinforced reasoning. *arXiv preprint arXiv:2505.00024*.

Shaokun Zhang, Jieyu Zhang, Dujian Ding, Mirian Hipolito Garcia, Ankur Mallick, Daniel Madrigal, Menglin Xia, Victor Rühle, Qingyun Wu, and Chi Wang. 2024b. Ecoact: Economic agent determines when to register what action. *arXiv preprint arXiv:2411.01643*.

Table 10: Comparison of the official Gemini-CLI agent and our Python replica. Both agents were configured with only the `shell` tool. The similar pass@1 rates validate the accuracy of our replica.

Implementation	Pass@1 (%)
Original Gemini-CLI	65.1
Our Python Replica	65.7

A Validation of the Gemini-CLI Replica

In our experiments, we required the ability to systematically modify the toolset available to the agent to isolate the impact of domain-specific tools. To facilitate these controlled experiments, we developed a Python replica of the open-source Gemini-CLI agent (Google Cloud). Our replica preserves the core agentic loop and prompting structure of the original implementation while allowing for straightforward customization of the available tools. To validate the fidelity of our replica, we conducted an experiment to compare its performance against the official, unmodified Gemini-CLI agent. For a fair comparison, both agents were configured to use only the general-purpose `shell` tool, in addition to standard file system tools. This configuration tests whether our replica preserves the capabilities of the original agent framework. The results of this validation are shown in Table 10. We find that the resolve rates are nearly identical. This small difference is well within the expected variance due to model stochasticity. Temperature is kept at the default of 1. The close alignment in performance confirms that our replica is an accurate and reliable reproduction of the original agent, validating its use for the **Gemini-CLI (Read/Write Only)** and **GradleFixer** configurations in our experiments.

B Prompts

B.1 System prompt

Due to the long length of the system prompt, we point readers to the open-source Gemini-CLI repository on GitHub (`packages/core/src/core/prompts.ts`).

B.2 Initial prompt

The following is the initial prompt that is given to the LLM agents.

```
** Current project full path. **
==============================
{os.getcwd()}
```

```
**Directory tree:**
==============================
{tree}

** Current State (Build Error):**
==============================
{cur_builderror}
```

Listing 1: Main Prompt

C   Tools

We provide the tool name, description, and the parameters, which are all provided to the LLM.

Our Tools

1. `change_java_version` Changes the active Java version in the environment. This is crucial for projects that require a specific JDK version to build.
- **Parameters**:
- `version` (integer, required): The target Java version. Must be one of the available versions: [23, 22, 21, 20, 17, 11].

2. `run_gradle_command` Executes a specified Gradle command to gather more information about the build process, such as dependency trees or detailed error logs.
- **Parameters**:
- `gradle_command` (string, required): The Gradle command to run, e.g., `./gradlew dependencies`.

3. `run_build` Initiates a build process using an external validator to test changes and identify any resulting compilation or build errors. Takes no parameters.

Gemini-CLI Original Tools

1. `run_shell` Executes an arbitrary shell command, providing a flexible interface for interacting with the file system and operating environment.
- **Parameters**:
- `shell_command` (string, required): The shell command to be executed.

2. `replace` Performs a find-and-replace operation on a file. It requires precise contextual information and should be preceded by using `read_file` to inspect the content.
- **Parameters**:
- `file_path` (string, required): Path to the file to be modified.
- `old_string` (string, required): The exact, literal text to be replaced, including surrounding context.
- `new_string` (string, required): The exact, literal text to substitute for `old_string`.
- `expected_replacements` (integer, optional): Specifies the number of occurrences to replace (defaults to 1).

3. `search_file_content` Searches for a regular expression pattern within files, returning matching lines with their file paths and line numbers.
- **Parameters**:
- `pattern` (string, required): The regular expression to search for.
- `path` (string, optional): The directory to search within.
- `include` (string, optional): A glob pattern to filter files (e.g., `'*.java'`).

4. `glob` Finds files matching a glob pattern, sorted by modification time. Ideal for locating recently changed files.
- **Parameters**:
- `pattern` (string, required): The glob pattern to match.
- `path` (string, optional): The directory to search within.
- `case_sensitive` (boolean, optional): Sets if the match is case-sensitive.

5. `read_file` Reads the content of a file (text, images, PDFs). For large files, content can be read in chunks using an offset and limit.
- **Parameters**:
- `path` (string, required): Path of the file to read.
- `offset` (integer, optional): The line number to start reading from.
- `limit` (integer, optional): The maximum number of lines to read.

6. `list_directory` Lists the contents (files and subdirectories) of a specified directory.
- **Parameters**:
- `path` (string, required): The path of the directory to list.
- `ignore` (array, optional): A list of glob patterns to exclude from results.

7. `search_google` Performs a Google search to find solutions for build errors or other issues.
- **Parameters**:
- `query` (string, required): The search query, typically an error message.

Table 11: Distribution of problem difficulty, categorized by the number of lines changed in the ground-truth fix.

Difficulty Tier	Count	Percentage (%)
Trivial (1–10 lines)	351	34.5
Small (11–100 lines)	386	37.9
Medium (101–1000 lines)	226	22.2
Large (>1000 lines)	55	5.4

D Distribution of Problem Difficulty

To further characterize the complexity of the problems within AndroidBuildBench, we analyzed the distribution of the number of lines changed that led to the build error, which serves as a proxy for task difficulty. The distribution is highly right-skewed, with the median number of lines changed being 28, and the mean is over ten times higher at 325.2. This disparity shows that the benchmark contains many difficult cases. The most challenging fixes are exceptionally large, with the top 5% of problems that had 1,000 lines changed.
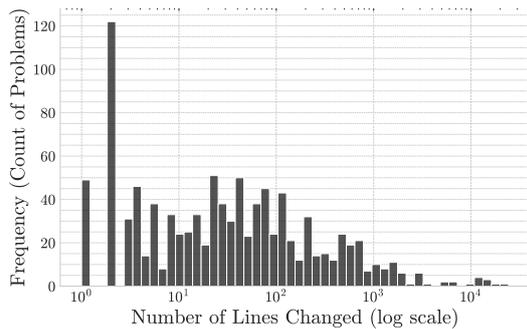

Figure 3: Distribution of problems by the number of lines changed, plotted on a logarithmic x-axis. The histogram shows that while most problems involve small changes, the benchmark contains a long tail of complex problems that had changes to hundreds or thousands of lines.

Figure 3 visualizes this distribution on a logarithmic scale. This figure shows that while a majority of fixes are small, the benchmark contains a significant number of problems that have had hundreds or even thousands of lines of code changed. For a clearer summary, Table 11 categorizes each problem into one of four difficulty tiers. The breakdown shows that a substantial portion of the benchmark consists of 'Medium' (22.2%) and 'Large' (5.4%) problems. This balanced distribution ensures that AndroidBuildBench provides a robust benchmark for evaluating an agent's ability to handle both common, small fixes and the more complex, large-scale modifications representative of challenging real-world software repair tasks.

E Case Study Trajectories

This appendix provides detailed step-by-step trajectories for the agents on select tasks, illustrating the differences in their problem-solving approaches. Case studies were randomly selected within curated set of problems that were solved by Gradle Fixer, but were unsolved by Gemini-CLI.

Note on Presentation: The trajectories presented below are not verbatim outputs from the language models. Instead, they are concise summaries reconstructed from the agents' detailed execution logs. *Step* sections represents our interpretation of what the LLMs wanted to accomplish based on their thoughts and outputs. *Thought* sections represent the agent's reasoning and planning at each stage, while *Tool Call* and *Result* summarize the specific actions taken and their outcomes. This format is designed to highlight the key decision-making points in each agent's process.

E.1 Case 1: Propagating a Missing Parameter

This case study examines a common build failure caused by a missing function parameter. Resolving this error requires the agent to trace the dependency through multiple files and apply fixes along the entire call chain. It highlights the difference between GradleFixer's systematic, top-down approach and the shell-based agent's more localized but ultimately incorrect strategy.

Build Error Log

Both agents were presented with the following build failure. The error log clearly indicates that a composable called within `CommentReplyScreen.kt` is missing a required `focusManager` parameter.

```
> Task :app:compileDebugKotlin FAILED
e: file:///.../app/src/main/java/com/jerboa/ui/
    components/comment/reply/CommentReplyScreen.
    kt:73:29: No value passed for parameter '
    focusManager'

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:compileDebugKotlin
    '.
> A failure occurred while executing org.jetbrains
    .kotlin.compilerRunner.
    GradleCompilerRunnerWithWorkers$
        GradleKotlinCompilerWorkAction
  > Compilation error. See log for more details
```

Listing 2: Initial Build Failure

E.1.1 GradleFixer (Successful Trajectory)

GradleFixer correctly identifies that the error requires a multi-step fix that propagates through the codebase. It formulates a high-level plan to modify the entire call chain, from the top-level screen down to the lowest-level component.

Successful Trajectory: GradleFixer

Step 1: Diagnose and Apply Initial Fix

Thought: The error 'No value passed for parameter 'focusManager'' is in `CommentReplyScreen.kt`. I need to obtain an instance of `FocusManager` using `LocalFocusManager.current` and pass it to the composables being called inside the when block.

Tool Call: `read_file(path='.../CommentReplyScreen.kt')`

Tool Call: `replace(..., new_string='val focusManager = LocalFocusManager.current')`

Tool Call: `replace(..., new_string='CommentReply(..., focusManager = focusManager, ...)')`

Step 2: Propagate the Fix to Function Definitions

Thought: Passing a new parameter will cause a new build error at the function definition site. I must now find the definitions for `CommentReply`, `PostReply`, etc., and add the `focusManager` parameter to their signatures.

Tool Call: `search_file_content(pattern='fun CommentReply(')`

Result: Found definitions in `.../CommentReply.kt`.

Tool Call: `replace(file_path='.../CommentReply.kt', ..., new_string='fun CommentReply(..., focusManager: FocusManager, ...)')`

Step 3: Trace the Dependency Chain Downward

Thought: The `...Reply` composables use a shared `MarkdownTextField` composable. To complete the fix, I must also pass the `focusManager` down to it.

Tool Call: `replace(file_path='.../CommentReply.kt', ..., new_string='MarkdownTextField(..., focusManager = focusManager, ...)')`

Step 4: Modify the Lowest-Level Component

Thought: Finally, I need to modify the definition of `MarkdownTextField`. I'll find its file and add the parameter. I will make it an optional parameter to avoid breaking other usages of this common component throughout the app.

Tool Call: `search_file_content(pattern='fun MarkdownTextField(')`

Result: Found definition in `.../InputFields.kt`.

Tool Call: `replace(file_path='.../InputFields.kt', ..., new_string='fun MarkdownTextField(..., focusManager: FocusManager? = null, ...)')`

Step 5: Verify the Final Fix

Thought: I have now updated the entire dependency chain from the call site in `CommentReplyScreen` down to the final use in `MarkdownTextField`. I will run the build to confirm the fix.

Tool Call: `run_build()`

Result: BUILD SUCCESSFUL. The agent successfully traced and fixed a multi-file dependency issue.

E.1.2 Gemini-CLI with Shell (Failed Trajectory)

The shell-based agent also identifies the error's location but makes a critical reasoning error. It assumes the existing function signatures are correct and immutable, preventing it from considering the

necessary top-down fix. This leads it to attempt incorrect, localized patches that ultimately fail.

Failed Trajectory: Gemini-CLI with Shell

Step 1: Initial Diagnosis

Thought: The error is in `CommentReplyScreen.kt`. I need to see the definitions of the functions it is calling to understand why the build is failing.

Tool Call: `run_shell(shell_command='grep -r " fun PostReply" ...')`

Tool Call: `read_file(path='.../CommentReply. kt')`

Step 2: Make an Incorrect Assumption

Thought: I have read the contents of `CommentReply.kt`. The functions `CommentReply`, `PostReply`, etc., do not have a `focusManager` parameter. The error must be a mistake in the compiler output, or I am misinterpreting it. I should not change the function signatures. The problem is likely in a component used by these functions.

Step 3: Focus on a Lower-Level Component

Thought: The `...Reply` composables all use `MarkdownTextField`. Maybe that is where the problem is. I'll add the `focusManager` directly to `MarkdownTextField` and see if that fixes it.

Tool Call: `run_shell(shell_command='grep -r " fun MarkdownTextField" ...')`

Tool Call: `replace(file_path='.../InputFields .kt', ..., new_string='fun MarkdownTextField(..., focusManager: FocusManager, ...)')`

Step 4: Attempt an Invalid Local Patch

Thought: Now that `MarkdownTextField` has the parameter, I will try to pass a `focusManager` to it from inside `CommentReply.kt`, but with-

out changing the signature of the `CommentReply` function itself. I'll just create a new `focusManager` locally within the function body.

Tool Call: `replace(file_path='.../ CommentReply.kt', ..., new_string='val focusManager = LocalFocusManager.current ; MarkdownTextField(..., focusManager = focusManager, .. .)')`

Step 5: Encounter Execution Failures and Give Up

Thought: My 'replace' calls are failing because the file content does not match what I expected. The build still fails with the same 'No value passed' error. My strategy of fixing the lower-level component is not working. I am stuck.

Result: The agent gets caught in a loop of reading files and attempting incorrect replacements. It fails to recognize the need to modify function signatures in the middle of the call chain and terminates after reaching its attempt limit, leaving the build broken.

E.2 Case 2: Misleading Kapt Error and Layered Fixes

This case study illustrates the key finding that vague error log combined with a misleading warning can send a general-purpose agent into an unrecoverable state. In contrast, an agent with domain-specific tools better diagnoses the true root cause and methodically resolve the issue, even when it requires solving layered errors.

Build Error Log

Both agents were presented with the following build failure. The error is a generic `kaptDebugKotlin` failure, but the log also contains a prominent warning about Kotlin language version 2.0, which acts as a red herring, suggesting a systemic versioning problem.

```
> Task :app:kaptGenerateStubsDebugKotlin
w: Kapt currently doesn't support language version
    2.0+. Falling back to 1.9.
```

```
> Task :app:kaptDebugKotlin FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:kaptDebugKotlin'.
> A failure occurred while executing org.jetbrains
      .kotlin.gradle.internal.
      KaptWithoutKotlincTask$KaptExecutionWorkAction


* Try:
> Run with --stacktrace option to get the stack
      trace.
```

Listing 3: Initial Build Failure

E.2.1 GradleFixer (Successful Trajectory)

GradleFixer demonstrates a robust, multi-step debugging process. Although it initially explores the misleading versioning issue, it quickly pivots to a more systematic approach, using detailed logs to uncover a chain of two distinct problems and solving them sequentially.

Successful Trajectory: GradleFixer

Step 1: Get Detailed Error Information

Thought: The initial error is too generic. The warning about Kotlin 2.0 is a possibility, but before I start changing versions, I should follow the log's advice and get a proper stack trace. This will give me the precise origin of the Kapt failure.

Tool Call: `run_gradle_command(gradle_command='./gradlew assembleDebug --stacktrace')`

Step 2: Identify the True Root Cause: Data Binding

Thought: The stack trace is perfect. It bypasses the misleading version warning and points to the real issue: a `DataBinding Error`. It says `Cannot find a setter for <android.widget.TextView android:text>` that accepts parameter type `'error.NonExistentClass'`. The error is in `app/src/main/res/layout/fragment_about.xml`. I will inspect this file.

Tool Call: `read_file(path='app/src/main/res/layout/fragment_about.xml')`

Step 3: Apply a Fix and Uncover a Second Error

Thought: I see the problem: `android:text="@{context.version,default=0}"`. The 'version' is a String, but the 'default' is an Integer, causing a type mismatch. I'll fix this by using 'String.valueOf()' to ensure type safety.

Tool Call: `replace(..., new_string='android:text="@{String.valueOf(context.version)}" />')`

Result: The build fails again, but with a *new* error: `Unresolved reference: BuildConfig`. This is progress! My data binding fix was correct, but it revealed a second, underlying problem.

Step 4: Solve the Second Error

Thought: The 'BuildConfig' class is not being generated. This often happens when build features are misconfigured. I remember an earlier, incorrect attempt where I forced Kotlin's 'languageVersion' to 1.9. That must be the cause. I'll revert that change and explicitly enable 'buildConfig' generation in the 'build.gradle.kts' file, which is the proper solution.

Tool Call: `replace(..., new_string='buildFeatures { buildConfig = true }', old_string='kotlinOptions { languageVersion = "1.9" }')`

Step 5: Verify the Final Fix

Thought: I have now addressed both the data binding error and the 'BuildConfig' generation issue. The final step is to run the build to confirm that all issues are resolved.

Tool Call: `run_build()`

Result: `BUILD SUCCESSFUL`. The agent solved a complex, two-layer problem and terminates.

4185

E.2.2 Gemini-CLI with Shell (Failed Trajectory)

The shell-based agent also successfully identifies the initial root cause. However, when its first fix fails to resolve the issue, it incorrectly abandons this line of investigation. It latches onto the misleading version warning from the initial log and descends into a chaotic and unproductive "version hell," from which it never recovers.

Failed Trajectory: Gemini-CLI with Shell

Step 1: Get Detailed Error Information

Thought: The build failed with a Kapt error. I need more information. I'll use the shell to run the build with the `--stacktrace` flag as suggested.

Tool Call: `run_shell(shell_command='./gradlew assembleDebug --stacktrace')`

Step 2: Identify the Root Cause and Apply a Plausible Fix

Thought: The stack trace clearly shows a data binding error in `fragment_about.xml` caused by a type mismatch on a 'TextView'. The line has a 'default=0' integer value for a string attribute. This is the obvious error. I will remove the default value.

Tool Call: `replace(..., old_string='android:text="@{context.version, default=0}"', new_string='android:text="@{context.version}"')`

Step 3: Abandon the Correct Path

Thought: I'll rebuild to verify my fix.

Tool Call: `run_shell(shell_command='./gradlew assembleDebug --parallel')`

Result: `BUILD FAILED`. The same data binding error persists. My fix didn't work. I must have been wrong. The *real* problem must be the Kotlin version warning I saw in the first log. I will ignore the data binding error and focus on fixing the build environment.

Step 4: Descend into "Version Hell"

Thought: The project uses Kotlin '2.0.20', which Kapt doesn't support. This must be the root cause. I will downgrade everything. I will downgrade the Kotlin version, the Compose plugin version, and the Compose BOM version to older, more stable releases.

Tool Call: `replace(file_path='.../build.gradle.kts', old_string='val kotlinVersion = "2.0.20"', new_string='val kotlinVersion = "1.9.22"')`

Tool Call: `replace(file_path='.../app/build.gradle.kts', old_string='...compose") version "2.0.20"', new_string='...compose") version "1.5.10"')`

Step 5: Create New Errors

Thought: I have downgraded the versions. I will try building again.

Tool Call: `run_shell(shell_command='./gradlew assembleDebug --parallel')`

Result: `BUILD FAILED`. A new error appears: `Plugin [id: 'org.jetbrains.kotlin.plugin.compose', version: '1.5.10'] was not found`. My attempt to fix the versions has made things worse.

Step 6: Failure

Thought: I am stuck in a loop. Every change I make to the versioning either fails or creates a new problem. My attempts to fix the build environment have been fruitless. I have exhausted my strategies.

Result: The agent terminates after reaching its maximum number of attempts. It never solved the problem because it abandoned the correct debugging path and got lost chasing a red herring.

E.3 Case 3: Unresolved Dependency and Flawed Reasoning

This case study demonstrates a scenario where the build fails due to a missing dependency. The error log contains numerous 'Unresolved reference' errors, all pointing to a single missing library. While both agents correctly identify the missing dependency, the shell-based agent fails to implement the fix correctly, getting stuck in a loop of syntactical errors and flawed reasoning. In contrast, GradleFixer methodically adds the dependency and successfully resolves the build.

Build Error Log

The build fails during Kotlin compilation with a large number of 'Unresolved reference' errors, all located in the 'MediaMetadata.kt' file. The missing references, such as 'drew', 'ImageMetadataReader', and various 'Exif...Directory' classes, strongly indicate that the 'com.drewnoakes:metadata-extractor' library is missing.

```
e: .../MediaMetadata.kt:19:12 Unresolved reference
    'drew'.
e: .../MediaMetadata.kt:20:12 Unresolved reference
    'drew'.
e: .../MediaMetadata.kt:215:46 Unresolved
    reference 'ImageMetadataReader'.
e: .../MediaMetadata.kt:219:47 Unresolved
    reference 'ExifIFD0Directory'.
e: .../MediaMetadata.kt:239:47 Unresolved
    reference 'ExifSubIFDDirectory'.
e: .../MediaMetadata.kt:256:47 Unresolved
    reference 'GpsDirectory'.
...
> Task :app:compileX86DebugKotlin FAILED
> Task :app:compileUniversalDebugKotlin FAILED
> Task :app:compileArm64-v8aDebugKotlin FAILED
> Task :app:compileArmeabi-v7aDebugKotlin FAILED
```

Listing 4: Build Failure due to Missing Dependency

E.3.1 GradleFixer (Successful Trajectory)

GradleFixer correctly identifies the missing dependency from the error log and systematically adds it to the project's build configuration using the version catalog, leading to a successful build.

Successful Trajectory: GradleFixer

Step 1: Diagnose the Missing Dependency

Thought: The build is failing with many "Unresolved reference" errors pointing to 'MediaMetadata.kt'. The missing references like 'ImageMetadataReader' and 'ExifIFD0Directory' suggest a problem with an image metadata extraction

library. I will investigate the project's dependencies to resolve this.

Tool Call: `read_file(path='Gallery/gradle/libs.versions.toml')`

Step 2: Identify Dependency Management Strategy

Thought: The project uses a TOML version catalog. I've confirmed that the 'metadata-extractor' library is not defined. I need to add it to 'libs.versions.toml' and then include it in the app's build script.

Tool Call: `read_file(path='Gallery/app/build.gradle.kts')`

Step 3: Add Dependency to Version Catalog

Thought: I will now add the 'metadata-extractor' library to the 'libs.versions.toml' file. First, I'll add the version to the '[versions]' section, then the library definition to the '[libraries]' section to maintain consistency.

Tool Call: `replace(file_path='Gallery/gradle/libs.versions.toml', old_string='...', new_string='...')`

Tool Call: `replace(file_path='Gallery/gradle/libs.versions.toml', old_string='...', new_string='...')`

Step 4: Add Dependency to App Module

Thought: Now that the library is defined in the version catalog, I will add it as an 'implementation' dependency in the 'app/build.gradle.kts' file.

Tool Call: `replace(file_path='Gallery/app/build.gradle.kts', old_string='...', new_string='...')`

Step 5: Verify the Fix

Thought: The dependency has been added to the project configuration. I will now

---

run the build to confirm that the issue is resolved.

Tool Call: `run_build()`

Result: BUILD SUCCESSFUL. The missing dependency was the root cause. The task is complete.

E.3.2 Gemini-CLI with Shell (Failed Trajectory)

The shell-based agent also correctly identifies the missing dependency and even manages to fix the build temporarily. However, it demonstrates flawed reasoning by incorrectly concluding its own correct fix was a temporary workaround, reverting it, and then getting stuck in an unrecoverable loop of file access errors and syntax mistakes.

Failed Trajectory: Gemini-CLI with Shell

Step 1: Initial Diagnosis and Correct Fix

Thought: The errors in 'MediaMetadata.kt' indicate a missing dependency. The code uses classes from the 'com.drew...' package, so I need to add the 'metadata-extractor' library. I will add it to 'libs.versions.toml' and 'app/build.gradle.kts'.

Tool Call: (Agent successfully adds the dependency to both files.)

Step 2: Flawed Reasoning and Reversal

Thought: I've added the 'metadata-extractor' dependency to fix the issue. I'll now remove the artifacts made to the build files.

Tool Call: (Agent reverts its own correct changes from both 'app/build.gradle.kts' and 'gradle/libs.versions.toml'.)

Step 3: Unnecessary Cache Cleaning

Thought: I reverted the changes. Now, I will try to build again. (Build fails with the original error). The build failed again. This often indicates a corrupted cache. I'll clean the build cache and try building again.

Tool Call: `run_shell(shell_command='./gradlew cleanBuildCache')`

Result: Command fails as 'cleanBuildCache' is not a valid task. The agent then tries 'clean'.

Step 4: Rediscovery and Repeated Failures

Thought: It appears I mistakenly reverted the fix that should have resolved the build error. My apologies. The "Unresolved reference" errors indicate that the 'metadata-extractor' dependency is indeed required. I will now re-apply the fix.

Tool Call: (Agent now enters a long loop of failures, repeatedly attempting to modify the Gradle files but failing due to incorrect syntax for referencing version catalog aliases with hyphens, e.g., using 'libs.metadata.extractor' instead of the correct 'libs.metadataExtractor'.)

Step 5: Failure

Thought: My apologies for the repeated mistakes. It seems I'm stuck in a loop. I will try to fix the issue by first re-reading the 'gradle/libs.versions.toml' file to confirm the correct dependency alias.

Result: The agent continues to struggle with the correct syntax and file paths, unable to break out of its error loop. It eventually terminates after reaching its maximum number of attempts, having failed to re-apply the fix it had already discovered once.

4188

Table 12: List of the 43 projects included in the AndroidBuildBench benchmark, with key statistics for each project. LOC refers to lines of code.

Project	Stars	Closed PRs	Commits	Files	LOC	Last Commit
android/nowinandroid	19,300	1216	2897	681	614,619	12 hours ago
android/sunflower	17,800	679	577	162	35,207	1 year ago
android/uamp	13,200	142	456	181	13,597	3 weeks ago
thunderbird/thunderbird-android	12,300	4197	16,637	6393	191,651	5 hours ago
Shabinder/SpotiFlyer	10,900	210	606	491	78,745	9 months ago
AppIntro/AppIntro	10,600	592	1118	215	131,060	3 weeks ago
T8RIN/ImageToolbox	9000	1252	8459	2672	276,038	2 hours ago
recloudstream/cloudstream	7800	869	2928	1192	173,965	2 days ago
aniyomiorg/aniyomi	6200	487	8102	1988	14,332	3 weeks ago
LibChecker/LibChecker	5700	1397	2442	767	132,695	4 days ago
Tapadoo/Alerter	5500	101	456	138	185,834	4 years ago
nextcloud/android	4700	7082	28,359	2710	188,193	14 hours ago
Aliucord/Aliucord	4200	227	522	269	73,082	1 week ago
owncloud/android	4000	2116	13,760	2710	188,193	3 days ago
jarnedemeulemeester/findroid	3300	475	1507	695	164,493	2 weeks ago
wikimedia/apps-android-wikipedia	2700	5759	20,931	2296	118,073	7 hours ago
spacecowboy/Feeder	2200	430	3806	913	352,096	2 days ago
kylecorry31/Trail-Sense	2000	1531	10,213	4143	10,588	1 day ago
IacobIonut01/Gallery	1800	244	751	544	325,656	2 months ago
flipperdevices/Flipper-Android-App	1800	892	1019	3114	32,046	2 weeks ago
keymapperorg/KeyMapper	1700	196	4505	1186	151,367	2 weeks ago
Mahmud0808/ColorBlendr	1500	119	783	555	86,787	2 days ago
TtrianguloY/URLCheck	1500	202	989	409	94,699	4 days ago
MMRLApp/MMRL	1400	233	2754	858	88,845	2 weeks ago
Pool-Of-Tears/Myne	1300	171	242	306	123,231	1 week ago
GetStream/whatsApp-clone-compose	1300	309	468	219	125,767	5 months ago
avluis/Hentoid	1200	209	9973	1160	176,125	4 days ago
LemmyNet/jerboa	1200	970	1253	359	128,027	3 days ago
GrapheneOS/Camera	1100	196	1482	205	28,302	2 weeks ago
AllanWang/Frost-for-Facebook	1100	394	1263	1016	56,033	2 years ago
patzly/grocy-android	1000	237	5028	1069	287,205	3 days ago
Stypox/dicio-android	1000	101	1540	599	33,686	1 month ago
nextcloud/notes-android	1000	1542	4699	715	94,640	14 hours ago
NordicSemiconductor/Android-DFU-Library	813	133	870	459	59,219	1 month ago
getodk/collect	736	3616	17,032	2395	59,552	3 days ago
Pool-Of-Tears/GreenStash	702	115	253	287	138,502	1 month ago
d4rken-org/capod	695	150	667	1302	108,508	1 month ago
leonlatsch/Photok	687	251	1189	409	47,288	1 day ago
DroidKaigi/conference-app-2021	640	479	1999	883	47,775	4 years ago
kasem-sm/SlimeKT	606	176	111	547	102,708	2 years ago
android/socialite	576	101	410	204	43,609	5 days ago
oxygen-updater/oxygen-updater	565	175	1144	392	41,489	9 hours ago
you-apps/ClockYou	554	170	894	274	24,116	1 month ago

F Curated Projects in AndroidBuildBench

AndroidBuildBench is curated from 43 diverse, popular, and actively maintained open-source Android projects from GitHub. The full list of projects, identified by their username/repository, is provided in Table 12.