CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Intent-Driven Code Generation for Android Application Testing Using Large Language Models

Master's Thesis in Computer Science and Engineering

Ali Gholamhosseinpour
Xiaoran Zhang

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Intent-Driven Code Generation for Android Application Testing Using Large Language Models

Ali Gholamhosseinpour
Xiaoran Zhang

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Intent-Driven Code Generation for Android Application Testing Using Large Language Models

Ali Gholamhosseinpour
Xiaoran Zhang

© Ali Gholamhosseinpour & Xiaoran Zhang, 2025.

Supervisor: Yinan Yu, Department of Computer Science and Engineering
Advisor: Dhasarathy Parthasarathy, Volvo Group Truck Technology
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Ali Gholamhosseinpour
Xiaoran Zhang
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Modern Android interfaces evolve rapidly, and conventional UI test automation struggles to keep pace with this change. This thesis presents an *intent–driven* framework that leverages large language models (LLMs) in combination with multi-modal UI representations to translate natural-language testing goals into executable Android tests. While inspired by crawler-based exploration, the framework adopts a modular architecture that separates *planning*, *selection*, *execution*, and *observation* stages. It incorporates memory for state tracking and includes an evaluator–optimizer loop to refine LLM outputs dynamically during execution. A hybrid screen representation—combining XML hierarchies and screenshots—enables the system to reason over both structural and visual elements of the UI, while a Python-based control layer drives actions on physical devices.

The framework is evaluated on three production-grade Volvo Group applications (Alarm Clock, System Settings, and Load Indicator). Across 45 reference scenarios, the generated tests achieve a **60%** aggregate pass rate – compared to manual tests at **87%**, reach up to **88%** functional correctness, and reduce the amount of written code by as much as **70 %** compared to manually implemented baselines. Ablation studies show that visual input in addition to XML consistently supports task success and rarely confuses the model, contributing to improved reasoning across a wide range of UI challenges. XML remains valuable for precise element localization, especially where structural anchors are critical. A reasoning analysis over 42 planner steps yields an average score of **4.3 out of 5** for correctness, indicating strong semantic alignment between global testing goals and selected local actions. The framework exhibits weaknesses in dynamic screens, complex seekbar interactions, and backend-dependent states, where test reliability remains limited.

This work contributes a modular LLM-based system for intent-driven UI testing, empirical evidence of its effectiveness and conciseness on industrial applications without model fine-tuning, and practical design guidelines for future intelligent testing tools, including prompt structures, tool invocation patterns, and memory-based tracking heuristics.

Overall, the study shows that combining multi-modal LLM reasoning with structured UI representations advances automated mobile testing toward more adaptive, maintainable, and goal-aligned workflows.

Keywords: Android UI Testing, Large Language Models, Intent-Driven Code Generation, Automated Software Testing, Multi-Modal Models, Test Script Generation, Semantic Reasoning

Acknowledgements

We would like to thank our academic supervisor Yinan Yu for their guidance and support throughout the thesis. We would also like to thank our advisor Dhasarathy Parthasarathy and colleagues at Volvo Group Truck Technology for their input and collaboration. Finally, we would like to thank our examiner Christian Berger for their constructive feedback.

Ali Gholamhosseinpour & Xiaoran Zhang, Gothenburg, June 2025

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Motivation 1
1.2 Objectives 2
1.3 Contributions 2
1.4 Scope 2
1.5 Structure of the Thesis 3

2 Background 5
2.1 Automated Android UI Testing 5
2.2 Android User Interface Structure 5
2.2.1 View Hierarchy 5
2.2.2 Dynamic Behavior of Android Applications 5
2.3 Controlling Android Applications 6
2.3.1 UI Automator: Accessibility-Based Interaction 6
2.3.2 Espresso and Instrumentation-Based Interaction 6
2.3.3 Black-Box vs. White-Box Control Methods 7
2.4 Fundamentals of Black-Box GUI Testing 7
2.4.1 What is Black-Box Testing in Mobile Context 7
2.4.2 Limitations of Accessibility-Based Control 8
2.5 State Space Exploration 8
2.5.1 Difficulties in Defining and Detecting App States 8
2.5.2 State Explosion 9
2.5.3 Model-Based Approaches 11
2.6 Tools and Frameworks Overview 11
2.6.1 Python-Based Control Layers 12
2.6.2 ATX Server and Remote Control 13

3 Related Work 15
3.1 Automated UI Testing 15
3.2 Practical Exploration Techniques 16
3.3 Language Model Integration in Testing 16
3.4 Multi-Modal LLMs for Vision-Enhanced UI Understanding 16
3.5 Prompting Strategies and Evaluation for Multi-Modal Models 17

3.6 Recent Advances in GUI Agents for Automated UI Testing 18
3.6.1 Gap in Intent-Based Android Test Generation 19

4 Methods **21**
4.1 Overview of Methodological Approach 21
4.2 Justification for Design Science 21
4.3 DSR and Research Questions 22
4.4 APP Selection for the Evaluation 24
4.5 Exploratory Work: Crawler-Based Initial Implementation 27
4.5.1 Overview 27
4.5.2 System Architecture 28
4.5.3 UI State Representation and Abstraction 29
4.5.4 State Identification and Hashing 29
4.5.5 Interaction and Exploration Strategies 31
4.5.6 Navigation Graph Construction 32
4.5.7 Intent Resolution and Semantic Annotation 33
4.5.8 Crawling Algorithm 34
4.6 LLM-Based Crawler 36
4.6.1 Motivation 36
4.6.2 Screen Representation 36
4.6.3 State Tracking 36
4.6.4 Exploration Procedure 36
4.7 Artifact Design and Implementation 37
4.8 System Architecture and Implementation Details 38
4.8.1 Modules using LLMs 38
4.8.2 Planning Module 38
4.8.3 Selection Module 39
4.8.4 Execution Module 40
4.8.5 Observation Module 40
4.8.6 Memory 41
4.8.7 Evaluator–Optimizer Workflow 42
4.8.8 Assertion Module 42
4.8.9 Network Sniffer Integration 42
4.9 RQ1 Evaluation: Code-Level Comparison Between Generated and Manual Tests 44
4.9.1 Line-Level Correctness 44
4.9.2 Unnecessary Steps 45
4.9.3 Flakiness 45
4.9.4 Robustness 45
4.9.5 Readability 46
4.10 RQ2 Evaluation: Semantic Understanding and Planner Evaluation . . 47
4.11 RQ3 Evaluation: Prompting Strategies and Multi-Modal Effectiveness 49
4.11.1 Experimental Design 49
4.11.2 Evaluation Metrics 50
4.11.3 Limitations 51

5 Results **53**

5.1 Test Selection and Comparison Overview 53
5.1.1 Example Manual vs. Automated Test Cases 53
5.2 RQ1 Results: Code-Level Comparison Between Generated and Manual Tests 54
5.3 RQ2 Results: Semantic Understanding and Planning Module Evaluation 56
5.4 RQ3 Results: Prompting Strategies and Multi-Modal Effectiveness . . 61
5.4.1 `click_xy`: Spatial Localization Accuracy 62
5.4.2 `click_id`: Semantic Targeting via ID Retrieval 66
5.4.3 `get_count`: Object Enumeration 67
5.4.4 `instance`: UI Component Classification 68
5.4.5 `get_text`: UI Text Retrieval 68
5.4.6 `seekbar`: Continuous Value Estimation 69

6 Discussion **73**
6.1 Discussion of RQ1: Code-Level Comparison Insights 73
6.1.1 Correctness 73
6.1.2 Unnecessary Steps 73
6.1.3 Flakiness 74
6.1.4 Robustness 74
6.1.5 Readability 75
6.2 Discussion of RQ2: Semantic Understanding and Planning Module Interpretation 75
6.2.1 Reasoning Rubric 75
6.2.2 Correctness and Semantic Alignment 76
6.2.3 App-Specific Reasoning Quality 77
6.2.4 Common Error Patterns and Limitations 77
6.3 Discussion of RQ3: Prompting Strategies and Multi-Modal Effectiveness 78
6.3.1 `click_xy`: Spatial Localization Accuracy 78
6.3.2 `click_id`: Semantic Targeting via ID Retrieval 80
6.3.3 `get_count`: Object Enumeration 81
6.3.4 `instance`: UI Component Classification 83
6.3.5 `get_text`: UI Text Retrieval 83
6.3.6 `seekbar`: Continuous Value Estimation 85
6.4 Threats to Validity 86
6.4.1 Internal Validity 87
6.4.2 External Validity 87
6.5 Overall Framework Performance 87

7 Conclusions and Future Work **89**

Bibliography **93**

A Appendix **I**
A.1 Estimating the number of distinct states I
A.1.1 Notation I

A.1.2 Chao1 Lower Bound Estimator II
A.1.3 Lincoln-Petersen Capture-Recapture II
A.1.4 Good-Turing Missing Mass Adjustment II
A.1.5 Branching-Aware Bayesian Augmentation II
A.1.6 Combined Point Estimate and Bounds III
A.1.7 Information-Theoretic Sample Complexity III
A.1.8 Practical Limitations III
A.2 Future Validation Strategy III

List of Figures

2.1 Illustration of state space complexity across applications. As the number of UI states and transitions increases, filtering techniques begin to break down. In highly dynamic apps, the state space becomes too dense and entangled to resolve effectively. 10

4.1 Main page of the Alarm application. Additional views can be found in Appendix A.1. 25
4.2 System main menu view. Additional images are included in Appendix A.2. 26
4.3 Main page of the Load Indicator app. Additional examples are available in Appendix A.3. 27
4.4 Navigation graph from a crawling session of an Android Application. 32
4.5 An example of semantically augmented nodes and edges from 4.4 where a settings button – highlighted by a transparent circle – is clicked in `Node 0` that gets connected to `Node 1` via `Edge 0`. 33
4.6 Overview of the system's workflow for intent-driven test generation. . 38

5.1 Number of `click_xy` predictions that fall inside the annotated bounding boxes. 62
5.2 L2 distance distributions for `click_xy` predictions, split by bounding box inclusion. 63
5.3 Manhattan distance distributions for `click_xy` predictions, stratified by bounding box inclusion. 64
5.4 Predicted click points for the task "set the fourth alarm to 2:13 AM." Pink corresponds to `XML`, orange to `IMG`, and green to `XML` $\oplus$ `IMG` modality. The X symbol marks the ground truth center of the target button, while the dashed red border indicates the bounding box of that button. 65
5.5 Predicted click points for the task "calibrate the third axle to 5 tons." Pink corresponds to `XML`, orange to `IMG`, and green to `XML` $\oplus$ `IMG` modality. The X symbol marks the ground truth center of the target button, while the dashed red border indicates the bounding box of that button. 66
5.6 Exact match counts for the `click_id` task. Applicable to XML modes only. 67
5.7 Edit distance histogram for `click_id` predictions in XML-capable modes. 67

5.8 Exact match counts for `get_count`, where the model must return the number of queried items. 68
5.9 Exact match counts for `instance`, where the model identifies the correct occurrence of duplicated UI elements. 68
5.10 Exact match counts for the `get_text` task. 69
5.11 Edit distance distributions for the `get_text` task. 69
5.12 L2 distance histogram for `seekbar` predictions. 70
5.13 Manhattan distance histogram for `seekbar` predictions. 70
5.14 Predicted click points for the task “set the 8kHz band to -9dB.” Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth target position on the 8kHz `seekbar`. 71
5.15 Predicted click points for the task “set media volume to 60%.” Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth target location on the media `seekbar`. 72

A.1 Alarm Clock App Screenshots V
A.2 System Settings App Screenshots VI
A.3 Load Indicator App Screenshots VII

List of Tables

3.1 Comparison of GUI agents on test code generation, intent input, multi-modal grounding, and assertion synthesis. 19

4.1 Robustness Rating Scale 46
4.2 Readability Rating Scale 46
4.3 Reasoning Quality Rubric 48
4.4 Modality configurations used for evaluation 50
4.5 Evaluation metrics by field, showing input types, metric used, and the output type of each metric. 50

5.1 Test Result Comparison: Manual (M) vs Automated (A), the manual test pass rate is 87% and automated test pass rate is 60%. 53
5.2 Code-Level Comparison Between Manual (M) and LLM-Generated (A) Tests 55
5.3 RQ2 Quantitative step Correctness and Qualitative Reasoning Assessment for three different Volvo apps 56
5.4 Average Reasoning Score by Test ID and Equivalence Case 57
5.5 Step-by-step evaluation of local intents in an equivalence case. Scores range from 1 (poor) to 5 (excellent). 58
5.6 Step-by-step evaluation of local intents in an unequivalence case. Scores range from 1 (poor) to 5 (excellent). 58
5.7 Correlation coefficients between equivalence label and average reasoning score. 58
5.8 Reasoning Examples by Global Intent, Local Intent, Local Intent Correctness (LIC), Reasoning, and Reasoning Score 61

1 Introduction

Modern Android applications are both feature-rich and highly dynamic, making reliable UI testing indispensable yet increasingly difficult. Conventional automated testing methods including random event generators and pre-scripted workflows often fall short in two major ways:

1. They fail to adapt when app interfaces evolve.
2. They lack the ability to align test execution with the developer's high-level intentions.

At the same time, recent advances in artificial intelligence, particularly large language models (LLMs), open a promising path forward. LLMs can interpret natural-language instructions and emit executable code, suggesting that they might bridge the gap between *what developers mean* and *what automated tests actually do*. This thesis explores that possibility by proposing and evaluating an *intent-driven framework* for Android UI test generation. The framework couples LLM-based reasoning with visual and structural exploration of the app, then plans and executes actions iteratively, adapting to observed outcomes instead of producing static scripts.

1.1 Motivation

The motivation for this work stems from a persistent gap in automated testing: while existing tools such as Espresso or UI Automator offer reliable and efficient execution, they operate at a low level and require developers to manually script view-specific interactions, leaving the actual testing intent implicit [5]. More recent LLM-based agents like DroidAgent attempt to generate high-level scenarios automatically, but rely solely on structural metadata and omit visual information entirely, which limits their applicability in modern, dynamic interfaces [40]. In this work, we explore whether combining LLMs with image augmented and structured, context-aware crawling can support a more adaptive testing strategy–one that engages with semantic intent, handles interface variability, and generates test cases aligned with realistic usage goals.

1.2 Objectives

This research is guided by three main objectives:

- **Design.** Create an LLM-based framework that can interpret natural language testing intents and translate them into Android UI test scripts.
- **Implementation.** Build a modular system that integrates planning, execution, and evaluation components, enabling adaptive test generation and exploration.
- **Evaluation.** Assess the system's performance across real-world industrial apps, focusing on its effectiveness, semantic alignment, robustness, and reasoning capabilities.

1.3 Contributions

This thesis makes the following contributions to the field of automated software testing:

- It introduces a novel architecture that unites intent understanding, dynamic app exploration, and automated test synthesis.
- It provides an empirical assessment on production-grade apps, reporting code-level correctness, reasoning scores, and prompt-engineering insights.
- It offers practical insights and design principles that can inform future research and development of automated UI testing tools.

By closing the gap between human intent and machine verification, the work pushes Android UI testing toward more scalable, adaptable, and semantically meaningful solutions.

1.4 Scope

This thesis focuses on industrial Android applications deployed on an embedded in-vehicle infotainment platform used in heavy-duty trucks. The platform runs a landscape-oriented display, and restricts third-party services for safety. All experiments are carried out on a physical Android rig controlled via uiautomator2 and Android Debug Bridge (ADB); the reasoning modules invoke GPT-4o without any fine-tuning. Consequently, the empirical findings are most valid for:

- Android apps whose UI elements expose accessibility metadata (XML hierarchy) and maintain relatively stable visual layouts.
- Test scenarios defined by high-level user intents rather than pixel-perfect regression checks.

To capture a realistic spread of UI patterns, we evaluate three proprietary truck apps—Alarm Clock (largely static), System Settings (scroll-intensive, many nested elements), and Load Indicator (dynamic, list-driven with pop-ups). Together, they cover static, scroll-heavy, and highly dynamic interaction styles. Findings should not be extrapolated to (i) apps dominated by custom canvas rendering or ViewGroups lacking accessibility hooks, (ii) platforms that require fully offline or on-premise inference, (iii) test suites that hinge on strict backend data or timing constraints.

1.5 Structure of the Thesis

The remainder of this thesis is organized as follows.

- **Chapter 2 – Background** introduces the technical foundations of Android UI testing, accessibility hooks, and multi-modal LLM reasoning.
- **Chapter 3 – Related Work** surveys prior research on automated GUI exploration, intent-driven agents, and vision-language models to position our contribution.
- **Chapter 4 – Methods** details the design-science methodology, overall architecture, and experimental setup.
- **Chapter 5 – Results** presents empirical findings from three industrial truck applications, contrasting LLM-generated tests with manual baselines.
- **Chapter 6 – Discussion** interprets the results, analyses limitations, and outlines practical implications.
- **Chapter 7 – Conclusion and Future Work** summaries the contributions and proposes future research directions.
- The **Appendices** supply supplementary figures, tables, and implementation details.

2

Background

2.1 Automated Android UI Testing

Modern mobile apps need to be fully tested to ensure correct behavior and user experience. Given the fast growth of the mobile app market and the wide variety of Android devices, automating the testing of apps' graphical user interfaces (GUIs) has become essential. Researchers and practitioners have developed numerous techniques to generate inputs and explore app behaviors automatically [13]. Automated Android GUI testing aims to simulate user interactions such as taps, gestures, or text entry and verify app responses without manual effort, which improves test efficiency and consistency.

2.2 Android User Interface Structure

2.2.1 View Hierarchy

Android UIs are structured as a hierarchical tree of UI components. Each visual widget is a `View`, and container elements, which are subclasses of `ViewGroup`, can hold child views or other containers, forming a nested layout structure [6]. Developers typically declare the UI in XML layout files, where a single root element, which can be a `View` or `ViewGroup`, contains nested elements defining the interface [6]. At runtime, the Android framework inflates this XML into the corresponding `View` objects, preserving the parent-child relationships. This view hierarchy is the basis for rendering the UI and is accessible to testing frameworks via instrumentation or accessibility APIs.

2.2.2 Dynamic Behavior of Android Applications

Android applications are event-driven and dynamic in nature. The UI presented to the user can change over time in response to user inputs or background events. For example, an app may start with a login screen and, upon successful login, dynamically load a new screen or update portions of the current screen. Many modern Android apps construct or modify their UI at runtime, for example by

adding list items from a web service or switching fragments within an Activity[1], so the set of UI states is not fixed at compile time [10]. In essence, an Android app can be modeled as a series of GUI states and transitions. Each screen state is defined by the current view hierarchy and content, and user or system events trigger transitions to other states [29]. This dynamic, state-dependent behavior poses challenges for automated testing, as tools must recognize when the app has reached a new state and handle the potentially large space of possible states.

2.3 Controlling Android Applications

2.3.1 UI Automator: Accessibility-Based Interaction

UI Automator is an Android testing framework that allows automated control of apps from outside the app's process, relying on Android's accessibility interface. It can interact with visible UI elements across different applications or system UI, using properties like the displayed text or content description to locate widgets [7]. UI Automator tests run as a separate instrumentation process and can simulate user actions such as clicks, swipes, or text entry on target apps without needing internal knowledge of the app's code. Notably, UI Automator is well-suited for scenarios such as navigating the device UI or testing flows that span multiple apps. For example, it can open the Settings app from within a test [7]. However, because it operates via the accessibility service, it may require the UI elements to be accessible and include identifiable text or resource IDs to reliably find them.

2.3.2 Espresso and Instrumentation-Based Interaction

Espresso is a popular Android UI testing framework written in Java and Kotlin. It operates within the app under test using instrumentation and runs in the same process as the app, allowing it to directly call UI framework methods and inspect the UI hierarchy from the inside. Espresso uses a declarative, concise API where testers specify view matchers to find UI elements by ID, text, or other properties, perform actions such as click or scroll, and make assertions on view state [5].

A key feature of Espresso is its built-in synchronization. It automatically waits for the app's main UI thread to be idle before performing actions, which greatly reduces flaky tests caused by timing issues [4]. Because Espresso has access to the app's internal structure through the instrumentation API, tests can use stable identifiers such as resource IDs to locate widgets and can even utilize knowledge of the app's internals. For example, tests can use custom view matchers or access model data if exposed, which blurs the line between black-box and white-box testing [5]. The framework is designed for single-app testing and cannot directly interact with other apps on the device. It focuses on fast and reliable testing of the target app's UI.

[1] An *Activity* in Android represents a single screen with a user interface. There is no universal standard for how activities are used; developers decide their structure. Many apps use just one activity, relying on fragments or navigation components for UI transitions.

Espresso is primarily a Java or Kotlin framework and is tightly coupled with the Android SDK and tooling.

2.3.3 Black-Box vs. White-Box Control Methods

Android supports both black-box and white-box approaches to UI testing, each with distinct advantages. UI Automator exemplifies a black-box method: the test treats the app as an opaque entity, interacting only through the UI exposed to the user and not relying on any internal implementation details [7]. This has the benefit of not requiring access to the app's code and enabling cross-app interactions, but it can be limited by what information is available through the accessibility API. In contrast, Espresso represents a white-box (or at least gray-box) approach: it leverages instrumentation to run within the app process, giving tests insight into the app's structure and life-cycle.

White-box methods allow more precise and efficient operations – for instance, they can avoid waiting arbitrarily by knowing exactly when the app is idle, and they often produce more stable tests due to synchronization and direct access to UI components [4]. The trade-off is that white-box tests typically require the app under test to be instrumented or built with test support, and they cannot easily exercise functionality outside the target app's scope. In practice, developers often combine these approaches: using Espresso for in-app UI flows and falling back to UI Automator for scenarios that require system UI or multiple apps.

2.4 Fundamentals of Black-Box GUI Testing

2.4.1 What is Black-Box Testing in Mobile Context

Black-box testing is a software testing methodology in which the tester evaluates an application solely through its external interface, with no knowledge of the internal code or implementation [23]. In the mobile app context, black-box testing typically means interacting with the app as an end-user would: sending touch events, entering text, and observing the resulting screen outputs or behaviors. The tester does not rely on internal variables or methods, instead verifying that given an input (e.g., a button press), the app produces an expected output (e.g., a new screen or a message), according to requirements. Mobile GUI testing under this paradigm treats the app as a closed box where only the GUI is available for interaction and verification. This approach aligns well with functional testing from a user's perspective and is often used when source code access is limited or when testing the app in a production-like environment.

2.4.2 Limitations of Accessibility-Based Control

Although black-box GUI testing using accessibility frameworks such as UI Automator or Appium is powerful, it carries inherent limitations.[2] One issue is that not all UI elements may be easily accessible or uniquely identifiable through the accessibility API. If developers have not set content descriptions, or if multiple elements share the same text, a test might struggle to distinguish them. The information retrieved via accessibility can sometimes be incomplete or inconsistent. For example, custom UI components might appear with generic class names or missing attributes, making reliable identification difficult [4].

Furthermore, because the test is external to the app, it lacks built-in synchronization cues [7]. The testing tool might need to poll or wait for UI elements to appear, which can lead to fragile tests if timing is off. This lack of direct insight can cause flakiness, where tests fail intermittently. For instance, waiting for a loading spinner to disappear may require an arbitrary sleep, as there is no direct event signal.

Performance is another consideration. Interacting via the accessibility layer can be slower than using direct method calls, since each action involves cross-process communication. In addition, pure black-box control cannot easily invoke certain app behaviors that are not exposed through the UI. For example, it may be impossible to trigger an internal function if there is no UI element linked to it. These limitations mean that although accessibility-based black-box testing is broadly applicable, it may require careful design of test logic and is sometimes complemented by instrumentation to achieve more complex validation.

2.5 State Space Exploration

2.5.1 Difficulties in Defining and Detecting App States

When performing automated exploration of a GUI – for example, in model-based testing or crawling an app, one fundamental question is how to define a "state" of the app. Intuitively, a state can be thought of as a distinct screen configuration of the app: a unique arrangement of views and content that the user sees at a given moment. However, in practice, deciding what constitutes a new state is non-trivial.

Many Android apps have content that updates or changes within the same screen or activity. For instance, a news feed may load new items over time. Should two instances of the feed screen with different data be treated as the same state or different states? If the criterion is too strict, treating any difference in content as a new state, the number of states explodes. If it is too lax, overlooking meaningful differences, the testing process might treat distinct scenarios as one state and miss coverage.

[2] `https://appium.io/docs/en/2.0/` – Appium is an open-source automation framework that supports multiple platforms and backends. It uses different automation engines depending on the target environment, such as `UIAutomator` for Android and `XCUITest` for iOS.

Detecting states automatically adds to the challenge. The testing tool must infer from the UI hierarchy and its properties whether the app has transitioned into a state not seen before. This involves comparing the current UI structure to previously observed ones and deciding if it is equivalent to any prior state. The presence of dynamically generated identifiers, animations, or nondeterministic content such as timestamps can make such comparisons difficult, as the UI may never be exactly identical between two runs.

Prior research highlights these difficulties. For example, defining each Android Activity as one state is often too coarse to capture dynamic UI changes. On the other hand, a very fine-grained view would flag even minor UI updates as separate states. For instance, if a screen displays a timestamp, the same screen revisited one second later could be treated as a different state solely due to the updated time [10]. As a result, automated testing tools need robust heuristics or definitions for GUI states to effectively navigate an app's UI without redundancy or omission.

To manage this trade-off, testing tools can apply filtering strategies during state comparison. A filtered state is one where specific UI elements or properties are deliberately ignored to reduce sensitivity to irrelevant changes. For example, testers may exclude dynamic elements such as timestamps, ads, or notification badges from the comparison process. This allows the tool to focus on structural or semantically meaningful differences, reducing redundant states while still preserving important transitions in the app's behavior.

2.5.2 State Explosion

The term "state explosion" refers to the rapid growth of the number of possible states as the system under test becomes more complex. In the context of Android GUI testing, state explosion can occur if the testing framework distinguishes states based on very granular UI differences. For example, imagine a calculator app that updates its display for each new digit entered, if the tester treats each distinct number on the screen as a separate state, the state space would be practically infinite. This extreme sensitivity leads to an explosion of states, overwhelming the testing process with redundant or trivial variations [10].

This happens when states that are meaningfully different get conflated into one because the chosen abstraction cannot tell them apart. For instance, two screens in a shopping app showing different products might both be represented by the same high-level "ProductPage" state if the criterion only considers the screen's view class, thereby losing information about which product is shown.

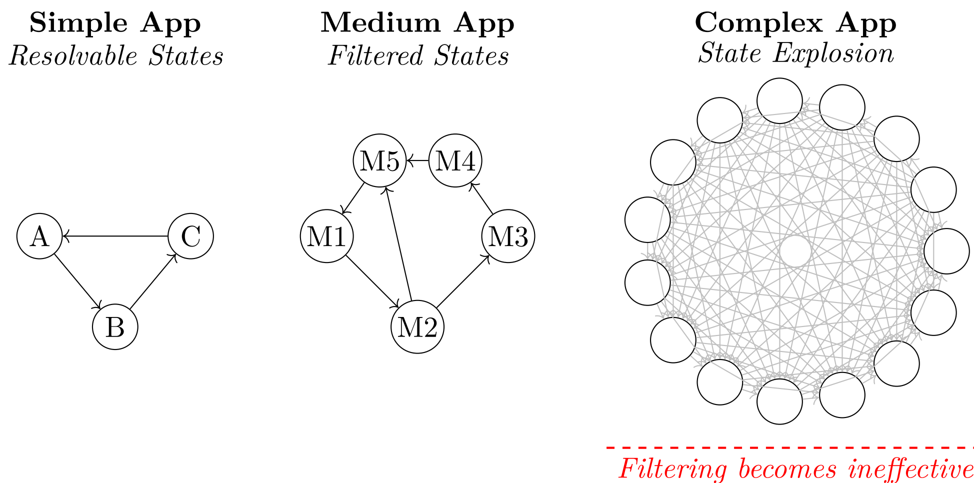


Figure 2.1: Illustration of state space complexity across applications. As the number of UI states and transitions increases, filtering techniques begin to break down. In highly dynamic apps, the state space becomes too dense and entangled to resolve effectively.

Both extremes are problematic: state explosion wastes resources and time, while merging distinct states risks missing bugs that only occur in specific contexts. Achieving the right balance is a key challenge. Researchers have proposed solutions such as defining multi-level GUI state abstractions to tune the granularity. Baek and Bae (2016) introduced GUI Comparison Criteria (GUICC) at multiple levels of abstraction to decide state equivalence, finding that intermediate levels of abstraction can significantly reduce state explosion while still distinguishing important differences between states [10]. A conceptual overview of this trade-off is shown in Figure 2.1, where state space growth ranges from manageable to unresolvable, illustrating the need for careful abstraction strategies.

Nevertheless, some inherent trade-off remains, and no single criterion works best for all apps; testers often must choose or adjust the state definition based on the app's characteristics. For example, in apps with highly dynamic content or complex interactive components, it may be beneficial to ignore certain regions of the UI or apply targeted filtering strategies to treat volatile areas differently. In contrast, for simpler or more static apps, a stricter comparison policy may be feasible without leading to state explosion. One possible heuristic is to identify widgets that commonly host dynamic content, such as scrolling lists or repeated items. These include components like `RecyclerView`[3], which can be flagged during exploration and handled using relaxed comparison rules to avoid over-fragmenting the state space.

[3] `https://developer.android.com/guide/topics/ui/layout/recyclerview` – `RecyclerView` is a flexible Android widget designed for displaying dynamic content efficiently by reusing views. It is commonly used to manage large or frequently changing data sets.

2.5.3 Model-Based Approaches

Model-based GUI testing techniques construct an abstract model, often a state machine or graph, of the app's possible states and transitions, then generate and execute test cases to cover that model. One notable model-based approach for Android is the use of GUI Comparison Criteria, or GUICC, as proposed by Baek and Bae [10].

In their framework, the app is represented as a GUI graph where nodes correspond to distinct GUI states and edges correspond to events such as user actions or system events that trigger transitions. GUICC defines what information is used to determine if two GUI states are considered equivalent. For example, a simple criterion might be the Activity name, treating all screens in the same Activity as one state, while a more detailed criterion might include the presence and properties of certain key UI elements.

GUICC includes a multi-level design that allows the tester to toggle between different abstraction levels for state comparison, ranging from coarse-grained to fine-grained [10]. Their empirical results showed that using a multi-level approach can improve exploration effectiveness. Compared to a single fixed criterion, it achieved higher code coverage and reduced the state explosion by merging redundant states that differ only in incidental details.

However, model-based approaches like this also have limitations. Defining the right abstraction level often requires insight into the app's behavior, and an inappropriate criterion could still either miss behaviors or cause overload. Moreover, building and maintaining a GUI model can be computationally expensive for very complex apps. There are also diminishing returns if the model grows too large to be fully explored.

GUICC, while mitigating some issues, does not entirely solve the challenge that some app behaviors, especially those depending on unseen data or timing, might not be captured purely by the GUI state abstraction. Additionally, implementing such an approach in practice might require instrumenting the app or using custom tooling to extract the UI structure at runtime, which can be complex.

In summary, GUICC represents a significant step toward controlling state explosion through smarter state equivalence criteria. However, it also illustrates that modeling dynamic GUIs remains a hard problem, often requiring careful tuning and still subject to the fundamental trade-offs of abstraction.

2.6 Tools and Frameworks Overview

A wide range of frameworks have been developed for Android UI automation. Some, like **Espresso** and **UI Automator**, are officially supported by Android and operate via instrumentation and accessibility APIs, respectively. Others are external, language-agnostic frameworks that wrap underlying automation engines.

Appium is a widely-used cross-platform automation framework that supports Android and iOS through the WebDriver protocol[4]. On Android, Appium supports backends such as `UIAutomator` and `Espresso`. It allows tests to be written in multiple languages (e.g., Python, Java, JavaScript), making it popular in multi-platform environments. However, we do not use Appium in this work due to the complexity of its infrastructure: setting up Appium involves launching a Node.js[5] server, installing automation backends, and routing commands through several layers of abstraction.

Earlier Android-specific frameworks include **Robotium**[6], which extends Android's instrumentation capabilities to simplify test writing in Java, and **MonkeyRunner**[7], which enables basic automation via Python scripts. While historically important, these tools are now largely obsolete, offering less flexibility and lower compatibility with modern Android UIs.

2.6.1 Python-Based Control Layers

In addition to official frameworks provided by Android, the testing community has developed external tools to facilitate automated UI testing. One such innovation is Python-based control layers for Android, exemplified by the *uiautomator2*[8] library.

These tools wrap the Android UI Automator functionality and expose it through a network API, allowing tests to be written in high-level languages like Python instead of Java. The typical architecture involves running a lightweight server component on the Android device that can receive commands such as HTTP or JSON-RPC requests, and execute them using the UI Automator API [33].

In the case of *uiautomator2*, a background service, sometimes called the ATX agent, is installed on the device. It leverages Android's accessibility bridges to perform actions and query UI state, and communicates with a Python client running on the tester's PC.

This setup brings several benefits. Test scripts can be developed and iterated quickly in Python, taking advantage of its rich ecosystem for generating inputs, logging, or integrating with test frameworks, without going through the Android build process for each change. It also simplifies cross-platform integration. For example, a Python-based test system could coordinate multiple devices or interact with backend services as part of end-to-end testing.

Essentially, Python control layers act as an intermediary that translates high-level test logic into low-level UI Automator operations, making Android UI testing more accessible and flexible for testers comfortable with scripting languages.

In the implementation described in this thesis, we focus on a minimal yet effective

[4] `https://www.w3.org/TR/webdriver2/`
[5] `https://nodejs.org`
[6] `https://github.com/RobotiumTech/robotium`
[7] `https://developer.android.com/studio/test/monkeyrunner`
[8] `https://uiautomator2.readthedocs.io/en/latest/api.html`

set of tools. The primary execution interface is `uiautomator2`, which provides programmatic access to Android's accessibility API. For communication with the device and issuing shell-level commands, we use the Android Debug Bridge (ADB)[9]. Additionally, to support manual inspection and interaction during development, we use a Java-based screen mirroring utility, `scrcpy`[10].

2.6.2 ATX Server and Remote Control

The Android Testing XML (ATX) server, as used in frameworks like *uiautomator2*, refers to the on-device service enabling remote control of the UI. It is usually packaged as an Android application or instrumentation test that, when launched, starts a server listening on a certain port of the device, often forwarded to the host machine via ADB.

Through this server, a remote client can send commands such as "find UI element with text X" or "click button with resource ID Y," which the server executes using the device's UI Automator APIs [33]. The approach is similar to how Appium works, where an Appium server controls devices via the WebDriver protocol. However, ATX is a more lightweight and direct mechanism tailored to UI Automator.

Remote interaction via such a server allows complex test scenarios. For example, a Python test script can invoke device UI actions, then verify some conditions by pulling data from a remote API or database, and then continue on the device—all in one flow. The ATX server handles the execution of each action and returns the result or any exception back to the client.

One challenge with remote interaction is maintaining synchronization and state. Since the control commands go over a network interface, the client must sometimes wait for confirmation that an action is complete or that a UI state has changed. The ATX framework often provides helper methods, such as waiting for a selector to match an element, to support this.

Overall, the presence of a remote control server on the device turns the device into a web service for UI actions. This enables a form of black-box testing that can be orchestrated from virtually any environment, not just from inside the device or emulator.

[9] `https://developer.android.com/tools/adb`
[10] `https://github.com/Genymobile/scrcpy`

3

Related Work

3.1 Automated UI Testing

Automated UI testing has long been a focus of software engineering research due to its potential for reducing manual effort and improving reliability. However, traditional approaches face persistent challenges.

Randomized input generation tools like Google's **UI/Application Exerciser Monkey** (commonly just *Monkey*) [31] use stochastic or heuristic-based methods to simulate user interactions. Monkey is a utility that sends a stream of random UI events to an app; it's often used for stress-testing and is fully black-box, though not systematic. This tool differs from heuristic-based generators like Humanoid by focusing purely on random input streams without any intelligent guidance. However, such approaches often fail to achieve comprehensive coverage, particularly in complex applications with dynamic UI elements.

Model-based testing frameworks, such as proposed by Amalfitano et al. [2], address this by constructing finite state machine (FSM) models (e.g., GUI graphs) to guide systematic exploration. Tools like *GUICC* use these models to generate test cases, but scalability remains a critical limitation [10]. Modern Android applications, with their dynamically loaded content and context-dependent behaviors, strain the ability of such methods to adapt in real time [2].

Another framework, **Calabash**[1], provided Cucumber-style tests in natural language for Android and iOS, though its development has slowed in favor of newer tools. There are also emerging tools focusing on specific aspects, such as **Sapienz**[2] (a search-based test input generator for Android developed at Facebook) and **Stoat** [35], which use stochastic or evolutionary algorithms to automatically explore app states. Cloud-based testing services, including **Firebase Test Lab**[3] and **BrowserStack**[4], often incorporate one or more of these frameworks to allow running automated GUI tests on many device models.

---

[1] `https://github.com/calabash/calabash-android`

[2] `https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/`

[3] `https://firebase.google.com/products/test-lab`

[4] `https://www.browserstack.com`

In summary, beyond the primary Android Testing Support Library frameworks, there is a broad ecosystem: some tools prioritize ease of scripting (Appium, Python wrappers), others target intelligent exploration (automated crawlers and model-based tools), and each comes with its own set of trade-offs in terms of setup complexity, supported languages, and level of control. Test engineers choose among these based on project needs, sometimes combining them to leverage the strengths of each [15].

3.2 Practical Exploration Techniques

To bridge the gap between random exploration and intent-driven testing, lightweight tools like *DroidBot* [24] leverage UI metadata to guide input generation. DroidBot constructs a state-transition model by simulating user actions (clicks, swipes) without modifying the application under test. While effective for basic scenarios, its lack of semantic understanding—such as interpreting user intent or decoding visual elements—limits its ability to navigate complex UIs or adapt to evolving app states. Additionally, DroidBot's focus on structural exploration overlooks opportunities to incorporate natural language specifications or developer-provided goals.

3.3 Language Model Integration in Testing

Recent advancements in large language models (LLMs) have introduced novel approaches to test automation. For example, *DroidAgent* [40] integrates LLMs to generate targeted test scenarios, such as creating user accounts or executing multi-step workflows. While this demonstrates the feasibility of using natural language to guide testing, DroidAgent's reliance on textual app metadata (e.g., accessibility labels) restricts its ability to interpret rich visual UI contexts, leading to misaligned actions in visually dense interfaces. Similarly, *CAT* (Cost-effective UI Automation Testing) [16] combines retrieval-augmented generation (RAG) with LLMs to map high-level tasks (e.g., "book a flight") to UI elements. However, CAT's dependency on predefined datasets and its specialization in single-app workflows (e.g., WeChat) limit generalizability across diverse applications [16].

3.4 Multi-Modal LLMs for Vision-Enhanced UI Understanding

Recent work on **multi-modal large language models (MLLMs)** shows promising ways to improve how systems understand user interfaces by combining visual input with language reasoning. One example is *ScreenAI* [9], a vision-language model from Google Research that builds on the PaLI architecture and uses a flexible patching method from pix2struct. ScreenAI is trained on a mix of datasets, including a special Screen Annotation task, allowing it to recognize UI element types, positions, and descriptions from screenshots. It reaches top results on tasks

like WebSRC and MoTIF, showing how vision-language models can help with UI navigation, question answering, and summarization.

Another important example is *Ferret-UI* [41], a multi-modal LLM designed for better understanding of mobile UIs. Ferret-UI deals with the challenge of long or narrow screens and small objects by splitting screens into sub-images before sending them through the model. It is trained on both basic tasks (like icon or widget recognition) and more advanced tasks (like describing screen content or guessing function), and it outperforms many open-source UI MLLMs, even doing better than GPT-4V on some basic tests.

Finally, *MobileVLM* [39] focuses on both within-screen and across-screen understanding by adding extra pre-training steps meant just for mobile UIs. Using their Mobile3M dataset, which has 3 million UI pages and real user transitions, MobileVLM beats general vision-language models on both their own and public mobile benchmarks.

Overall, these systems show how combining vision and language can improve UI understanding and interaction. However, their use in intent-driven Android UI testing, especially for generating test code based on developer instructions, is still not well explored. This thesis works to fill that gap by building a framework that connects visual UI understanding with semantic intent planning to create useful automated tests.

3.5 Prompting Strategies and Evaluation for Multi-Modal Models

Recent work has exposed the difficulty of instruction grounding in mobile and web interfaces. WinClick [21] introduces *WinSpot*, a Windows-GUI benchmark where element selection is evaluated by matching the predicted click point against human-annotated bounding boxes; their study shows that screenshot-only agents degrade on cluttered or visually repetitive layouts, which suggests that vision alone is brittle.

SoEval [26] formalizes evaluation for structured outputs, arguing that schema-aware exact-match and field-level F1 are more informative than free-form text metrics; their benchmark targets JSON/XML answers and it reflects the strict output required in our study. This is especially relevant in UI interaction settings where a model must resolve a single valid JSON answer.

3DAxisPrompt [25] investigates visual prompt engineering for 3-D grounding in GPT-4o and demonstrates that injecting explicit geometric priors (axes, SAM masks) markedly improves localization across four 3-D datasets. While the paper does not quantify token overhead, the prompt variants it explores differ substantially in length, which motivates us to log token usage as a practical cost metric.

3.6 Recent Advances in GUI Agents for Automated UI Testing

The rise of foundation models and multi-modal learning has accelerated innovation in GUI agents, which leads to systems that can autonomously interact with digital environments in ways previously limited to humans. This technology could provide new methods for automated UI testing.

One notable example is Operator by OpenAI (2025), a computer-using agent (CUA) that leverages GPT-4o's multi-modal capabilities and reinforcement learning to execute tasks directly on live websites, interpreting visual screenshots and performing human-like interactions without relying on APIs [30]. Complementing this, the Browser-Use framework (2025) offers an open-source platform that simplifies web interactions for AI agents through structured interfaces and lightweight automation hooks, supporting tasks from data scraping to complex workflows [14].

Further expanding this landscape, recent research has proposed advanced training strategies for GUI agents. ARPO (Agentic Replay Policy Optimization) [27] introduces an end-to-end reinforcement learning approach with task selection and replay buffer mechanisms to address sparse rewards and delayed feedback in GUI environments. SpiritSight [20], on the other hand, integrates a Universal Block Parsing method with a large-scale GUI dataset (GUI-Lasagne) to improve precision in dynamic, high-resolution visual inputs, strengthening the agent's grounding and decision-making.

Beyond reinforcement learning, hybrid planning architectures like Agent S [1] combine hierarchical planning and experience augmentation to improve generalizability across operating systems, while smartphone-focused frameworks like CoCo-Agent [28] use multi-modal inputs to achieve state-of-the-art performance on mobile automation benchmarks.

Collectively, these systems represent a new generation of UI testing that overcomes key limitations in previous work:

- They operate across diverse environments without requiring predefined APIs or static metadata.
- They incorporate multi-modal (text + vision) reasoning to handle rich visual layouts.
- They apply advanced learning strategies (e.g., reinforcement learning, replay optimization) to improve long-horizon task execution.

Compared to prior LLM-integrated tools like DroidAgent or CAT, these recent approaches demonstrate significantly higher flexibility, generalization, and robustness, opening new opportunities for research on adaptive, vision-grounded, and intent-aligned UI testing.

3.6.1 Gap in Intent-Based Android Test Generation

Recent GUI agent research can be grouped into general-purpose systems for desktop and web, and mobile-specific agents. While these frameworks demonstrate interactive capabilities, none are designed to generate reusable Android test code with structured assertions derived from external intent.

General-purpose agents such as Operator [30], Browser-Use [14], Agent S [1], ARPO [27], and SpiritSight [20] focus on online interaction or visual understanding. These systems do not emit test code or assertion logic, and intent is either inferred implicitly or not modeled at all. Their outputs are transient and cannot be reused in validation or CI pipelines.

Mobile-specific agents include CoCo-Agent [28] and DroidAgent [40]. CoCo executes smartphone tasks using multi-modal inputs but is designed for task completion rather than test generation. DroidAgent generates test code from natural language but does not support visual inputs and lacks structured assertion synthesis.

Although *DroidAgent* already generates Android test code from natural-language intents, it relies *exclusively* on accessibility metadata (resource-ID, content description) and ignores screenshots, making it unsuitable for the visually driven widgets that dominate our industrial apps. Moreover, the publicly released prototype is tightly coupled to a patched version of Android Instrumentation and hard-coded UI heuristics, so adding multi-modal input or evaluator–optimizer feedback would have required a near rewrite of the core planner and executor modules. Given these architectural constraints, designing a clean, modular framework from scratch proved both lower-risk and more extensible than adapting *DroidAgent* to our use case.

System	Code Output	Intent Input	Multi-modal	Assertions
Operator [30]	No	No	Yes	No
Browser-Use [14]	No	Partial	Partial	No
Agent S [1]	No	No	Yes	No
ARPO [27]	No	No	No	No
SpiritSight [20]	No	N/A	Yes	No
CoCo-Agent [28]	No	Fixed	Yes	No
DroidAgent [40]	Yes	Yes	No	No
Our Framework	**Yes**	**Yes**	**Yes**	**Yes**

Table 3.1: Comparison of GUI agents on test code generation, intent input, multi-modal grounding, and assertion synthesis.

As shown in Table 3.1, to the best of our knowledge, no existing system supports all four key requirements: accepting external intent, using multi-modal input, generating test code, and inserting structured assertions. Our framework attempts to integrate these capabilities. Its assertion module uses multi-hop reasoning over the GUI hierarchy and screenshots to synthesize context-sensitive checks and allow for end-to-end generation of executable Android tests that are aligned with developer intent.

4

Methods

4.1 Overview of Methodological Approach

This thesis applies Design Science Research (DSR) as the methodological foundation for both the construction and evaluation of the artifact. DSR is defined by its focus on creating purposeful artifacts and generating knowledge through their iterative refinement and testing. The method is particularly appropriate when the objective is to solve practical problems through design, while also contributing generalizable insights to the knowledge base. This dual purpose is core to DSR, and it distinguishes it from explanatory or interpretive methods that aim primarily to describe or understand existing phenomena.

We follow the process model articulated by Peffers et al.[32] and the conceptual framework introduced by Hevner et al.[19], in which the design process is organized around six core activities: problem identification, definition of solution objectives, artifact design and development, demonstration in context, evaluation against objectives, and communication of the resulting knowledge. The artifact constructed in this project is a test generation system that uses large language models to interpret user intent and produce executable Android UI tests. This artifact is iteratively developed, tested, and refined through evaluations on real industrial apps. Design choices are informed by the needs of professional Android developers, as well as by technical limitations uncovered during empirical use. Each iteration contributes both to improved system behavior and to the accumulation of design knowledge related to semantic UI planning.

4.2 Justification for Design Science

DSR is the appropriate methodological choice for this work because the research has two simultaneous aims. The first is to construct a functional artifact: a software system that generates Android tests based on user intent. The second is to derive reusable insights into how large language models can be structured, prompted, and evaluated in the context of interactive software testing. DSR supports both aims by integrating structured artifact design with systematic evaluation and by requiring that research outputs include both implemented systems and formalized knowledge contributions.[19]

The project begins from a real-world need: enabling software testers and developers to write test cases without manually specifying each UI interaction. This need defines the problem space and anchors the relevance cycle. The artifact is developed through iterative implementation cycles, where each version is evaluated against its ability to interpret intent and successfully operate on Android UI structures. These iterations are part of the design cycle. Throughout the process, prior work on automated testing, multi-modal models, and prompt engineering is used to inform design decisions and complete the rigor cycle described by Hevner et al.

Alternative methodologies were considered but rejected. A purely experimental approach would allow performance measurement, but not artifact construction. A grounded theory approach would allow theoretical development but would not produce a working system. DSR uniquely supports the creation of novel artifacts and the rigorous analysis of their utility and behavior, making it the only suitable choice for a project whose contribution is both practical and theoretical.

4.3 DSR and Research Questions

This research is guided by three core questions:

- RQ1: How effective is the proposed framework in generating valid and executable test code for developer-defined intents?
- RQ2: How accurately does the framework understand developer intents and translate them into meaningful and correct actions?
- RQ3: What are the most effective prompting strategies and modalities for enabling accurate and semantically grounded responses from multi-modal models given visual and textual input?

These questions are designed to span all major activities in the Design Science Research process model as defined by Peffers et al.[32], and to connect explicitly to the three cycles in Hevner's DSR framework: the relevance cycle (connection to practical problem context), the rigor cycle (connection to existing knowledge), and the design cycle (artifact creation and evaluation).[19] Each question isolates a distinct axis of design concern: artifact utility in deployment, internal semantic performance, and external model control via prompt design.

RQ1 concerns the artifact's practical effectiveness when deployed in real-world development workflows. It corresponds to the Demonstration and Evaluation phases of the DSR process. The question asks whether the generated output meets the expectations of professional developers in terms of correctness, coverage, and executability. Here, the evaluation focuses on system-level behavior using empirical comparison against manually written tests for three industrial Android applications. These tests are treated as application-specific ground truth, and include embedded requirements documentation that defines intended behavior.

The system's output is evaluated based on pass/fail outcomes, execution trace validity, and behavioral alignment with those expectations. This question also supports the relevance cycle by grounding the artifact in a real industrial need: reducing the manual burden of writing UI tests by enabling developers to express them at a semantic level. In DSR terms, RQ1 evaluates whether the artifact, as instantiated, fulfills its practical design objectives in context and demonstrates fitness for purpose.

RQ2 addresses the internal reasoning quality of the artifact and corresponds to the Design and Evaluation activities of DSR. While RQ1 tests whether the system performs well at the output level, RQ2 investigates whether that output is produced through semantically coherent and logically consistent internal steps. Specifically, it asks how well the system interprets global intent and translates it into meaningful actions using the current GUI context. This includes the performance of the planning module, which decomposes intent into sub-steps, and the evaluation module, which classifies whether the intermediate state transitions match the task objective.

Failures in intent understanding may manifest as action misalignment, missed preconditions, or irrelevant navigation paths. The correctness of these decisions depends not only on LLM outputs but on how those outputs are conditioned by prompt structure, state abstraction, and context memory. Within DSR, RQ2 maps to the rigor cycle by linking design decisions—such as modular LLM role separation, screen representation, or state tracking—with observed behavior. The goal is to assess whether the artifact exhibits robust semantic behavior across variable UI states and task types.

RQ3 treats prompt design as a primary design variable and investigates its effect on the quality of outputs from a multi-modal model. It aligns with the Design and Evaluation phases of the DSR process, but contributes primarily to the rigor cycle by extracting knowledge that is not specific to this artifact alone. The question is motivated by the observation that prompting strategies for multi-modal models are still poorly understood, particularly in domains that involve vision-plus-action tasks such as GUI reasoning[38].

We selected GPT-4o because, at the time of this work, it was the only commercially available multi-modal LLM that simultaneously offered:

- State-of-the-art multi-modal model with vision support.
- A dependable public API with a large context window and consistent sub-second latency.
- Native support for interleaving screenshots and structured text in a single prompt without extra fine-tuning.

These characteristics let us run zero-shot experiments and keep infrastructure simple.

Open-source vision-language models such as Qwen-VL[11], Ferret-UI[41], or Mo-

bileVLM[39] would have required on-premise GPU clusters plus domain-specific fine-tuning to match GPT-4o's accuracy, which will add an impractical cost for an industrial test rig; research models (ScreenAI[9], SpiritSight[20]) aren't publicly usable; and other commercial MLLMs (Gemini 1.5 Pro[1], Claude 3[2]) are not accessible in our corporate environment. GPT-4o therefore remained the only option combining high accuracy, low latency, and a stable vision-text API.

The artifact uses GPT-4o to consume both screenshots and XML UI hierarchies as input, and the design of prompts determines how this mixed-modal information is presented. RQ3 explores how the structure, phrasing, and sequencing of prompts affect the interpretive performance of the model. Examples include whether to use chain-of-thought decomposition over visual input, whether to annotate screenshots with natural language descriptions, or how to order context (e.g. state summaries before action instructions). These design decisions are not incidental—they determine the system's ability to generalize across new apps and tasks. In DSR terms, it evaluates how a configurable design artifact (the prompt itself) affects artifact behavior, and produces transferable design knowledge that applies to any task involving multi-modal interface reasoning.

4.4 APP Selection for the Evaluation

We selected three applications from the Volvo Android environment for evaluation: *Alarm Clock*, *System Settings*, and *Load Indicator*. These were chosen not only for practical reasons of scope but also because they expose a broad range of interaction patterns, UI complexities, and architectural behaviors that are representative of Volvo's application ecosystem.

Alarm Clock appears simple in structure but presents significant challenges for automated testing. Several UI elements in this app—such as toggles and icons—are only visible as changes in the screenshot and are not reflected in the XML hierarchy at all. For instance, toggling an alarm does not result in any change to the underlying view tree, and the toggle itself is not recognized as an interactive element. This means that neither the action nor the resulting state change is detectable through structural inspection alone. Instead, image analysis is necessary to identify and reason about these interactions. Additionally, some icons within the app are purely visual with no descriptive attributes or resource IDs, requiring visual–structural correlation to infer their role. While the navigation structure is shallow and predictable, the app's dependence on visual context over structural metadata makes it uniquely challenging in the context of black-box exploration. A representative screenshot of the main page is shown in Figure 4.1.

[1]Google DeepMind, "Our next-generation model: Gemini 1.5," Google AI Blog (Feb 15 2024).
[2]Anthropic, "The Claude 3 Model Family: Opus, Sonnet, Haiku," Model Card v1.0 (Mar 2025).

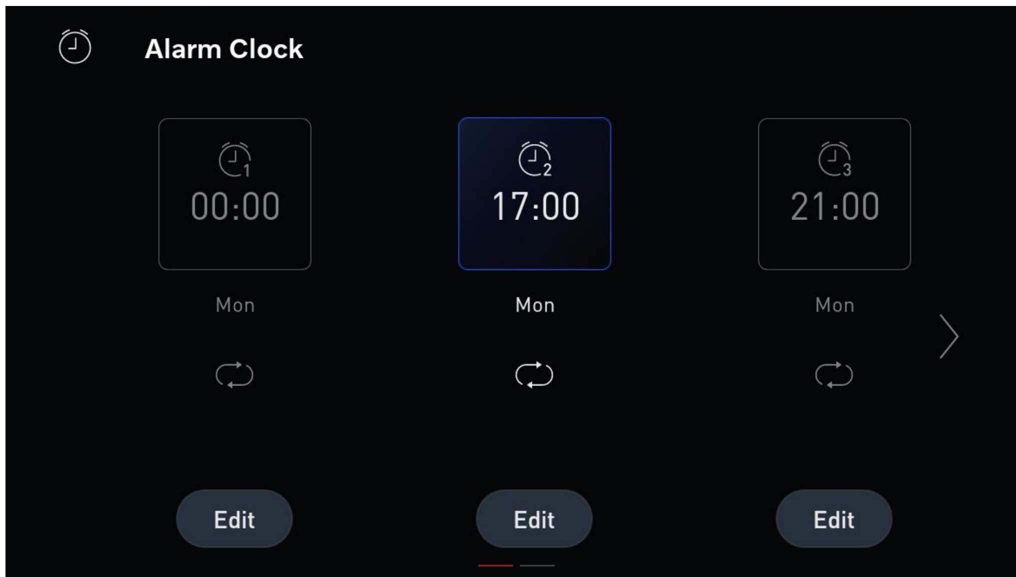


Figure 4.1: Main page of the Alarm application. Additional views can be found in Appendix A.1.

System Settings resembles standard Android system settings in layout and behavior. It includes a large number of scrollable views and deeply nested menus, making exploration time-consuming. The app also contains several types of interactive widgets that require special handling, including `seekbars` and `toggle` switches. These elements are often sensitive to small changes in gesture precision and timing. In many cases, setting a value cannot be done purely by invoking an action; it requires visual feedback to determine success. The presence of multiple layers of menus and the need for contextual interaction across those layers pose practical challenges for planning and execution. Figure 4.2 illustrates the top-level view of the system settings interface. Additional UI cases are shown in Appendix A.2.

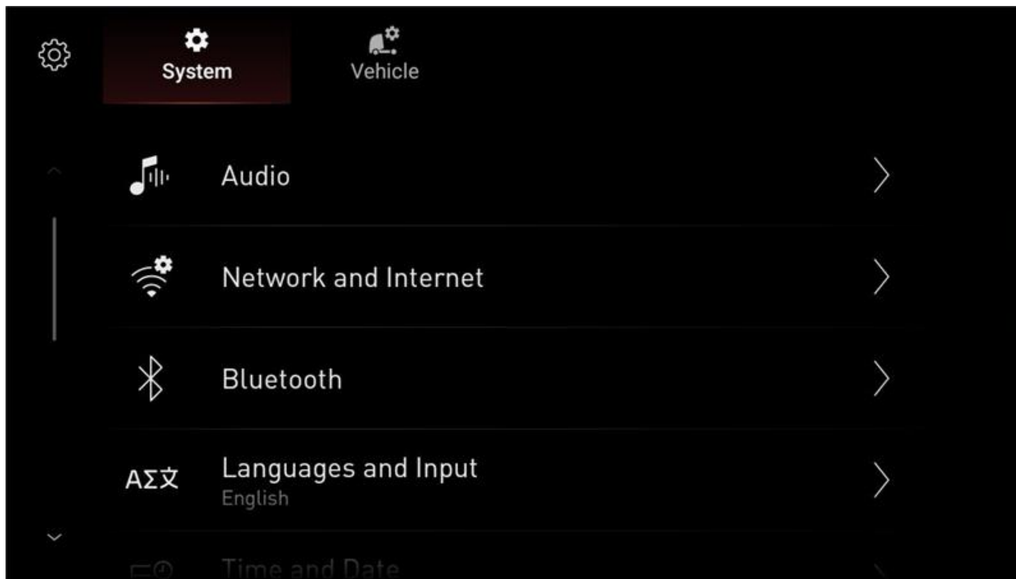


Figure 4.2: System main menu view. Additional images are included in Appendix A.2.

Load Indicator is the most complex of the three applications, particularly in terms of state space and navigation ambiguity. It contains a large number of dynamic UI components whose appearance and layout depend on the current backend mode of the app. These modes are often invoked through undocumented API calls, and their effects can completely reshape the UI layout. The app includes repeated, nearly identical screens, making state differentiation difficult. It also includes many visual-only components such as graphical representations of axles or trailers, which cannot be semantically interpreted through the XML hierarchy alone. Pop-up messages, mode changes, and polling-based updates further increase the difficulty of extracting stable state representations. In practice, understanding and interacting with Load Indicator requires tracking navigation paths, backend context, and visual structure in combination. A representative UI screen is shown in Figure 4.3.

Figure 4.3: Main page of the Load Indicator app. Additional examples are available in Appendix A.3.

These three applications were selected after evaluating other available apps in the same environment. Collectively, they encompass the most common design patterns and interaction challenges observed across Volvo's internal applications. Based on this coverage, we consider it reasonable to expect that methods developed and evaluated on this set will generalize to a broader portion of the application ecosystem.

4.5 Exploratory Work: Crawler-Based Initial Implementation

4.5.1 Overview

The crawler explores the graphical user interface (GUI) of an Android application without human guidance. Its output is a directed navigation graph whose vertices represent distinct interface states and whose edges represent user actions. That graph feeds a test generator that can replay paths matching a given intent.

Assumptions

The crawler operates under several structural and behavioral assumptions about the application under test and the Android platform. It assumes that the application is the only foreground `Activity` and that the Android accessibility layer exposes a complete and stable view hierarchy on demand. The crawler treats the screen as stable during the short interval following an action, meaning that the bounds and structure of UI elements do not change in response to background processing or transient animations within that window. This permits accurate coordinate-based interaction with screen elements, as their positions are assumed not to shift after initial observation.

For the purposes of this thesis, we define a *UI change* as any detectable modification in screen state based on either (i) a structural difference in the view hierarchy—such as node additions, deletions, or attribute changes—or (ii) a difference in visual appearance inferred through LLM-based reasoning over screenshots. Unlike low-level pixel hashing, our visual comparison is semantic in nature: the multi-modal model is used to judge whether a visible change has occurred that is meaningful in context, such as a toggle changing from "enabled" to "disabled" or a view element appearing or disappearing. This approach allows the crawler to capture both structural transitions and visually grounded changes, even when such changes are not reflected in the XML view hierarchy.

Every action performed by the crawler is expected to produce a deterministic and reproducible state, up to elements that are explicitly excluded from hashing (such as dynamic text). Transitions between screens are assumed to be reversible either through identifiable back-navigation elements or the Android system back button. This reversibility allows the crawler to return to prior states when exploration requires it.

The use of a language model is limited to generating semantic labels for states and UI elements. These labels are intended to assist with downstream test planning and reporting but are not used in any decision-making during exploration itself. The model is not queried continuously but only once per new element or state that lacks meaningful metadata.

4.5.2 System Architecture

The crawler is structured as a modular pipeline in which each stage transforms the application state or performs a decision based on it. At the base is the device interface, which executes low-level input events and retrieves the current screen content. On top of that, the state extraction module interprets the GUI hierarchy and screenshot to form an abstract representation of the screen that captures stable identifiers and actionable elements.

The extracted state is passed to the hashing module, which computes a content-based identifier to determine whether this state has already been visited. The exploration engine then selects an element with unexplored interactions and chooses the highest-priority action for that element. Once an action is executed, the crawler captures the resulting screen, computes its hash, and determines whether it represents a new state or a virtual variant of an existing one.

A central controller coordinates these modules and maintains a navigation graph as new states and transitions are discovered. This graph encodes the structure of the application's UI flow and accumulates metadata required for downstream test generation.

A single orchestrator object coordinates the pipeline: *capture* → *hash* → *decide* → *act* → *repeat*.

4.5.3 UI State Representation and Abstraction

Each visited UI node stores a structured *state descriptor*, which summarizes the current screen configuration and serves as the primary unit of exploration. This descriptor is defined as the tuple

$$S = \left(\text{pkg}, \text{activity}, I, \mathcal{E}\right),$$

where pkg is the package name, activity the foreground activity, I a base-64 screenshot thumbnail, and $\mathcal{E} = \{e_1, \ldots, e_n\}$ the set of UI elements extracted from the view hierarchy.

This representation captures all semantic and structural data necessary for planning and test synthesis, but it is **not** used as a state identifier. Equality between UI states—used, for example, in caching or cycle detection—is defined separately via a hashing function over selected components of S (see Section 4.5.4).

For every element $e_i \in \mathcal{E}$ the crawler stores:

- a stable view-tree path (XPath),
- screen center (x, y),
- widget class and resource identifier,
- interaction capabilities (clickable, toggle, scroll, editable),
- semantic labels $(\text{name}, \text{description}, \text{intent})$ generated once by an LLM.

Special element groups

Certain compound widgets demand bespoke handling:

- **Toggles** - binary widgets (check-boxes, switches).
- **Number pickers** - triplets {decrement, numeric display, increment}.
- **Replicators** - list/grid containers whose children share an identical structure.

These groups are detected heuristically (class names, sibling layout) and treated as *virtual action spaces* so that repetitive elements do not bloat the graph.

4.5.4 State Identification and Hashing

The crawler assigns a stable identifier to each observed UI state to detect revisits, control the exploration loop, and manage graph connectivity. Hashing is based on the structural and semantic content of the screen rather than its pixel-level appearance.

Each state is assigned a hash value derived from its activity name and a set of hashed element descriptors. An element descriptor includes the element's resource

ID, class name, XPath, and optionally an embedded image crop if no stable textual or structural identifier is available. Attributes such as live text, scroll positions, or dynamic counters are excluded to avoid false negatives due to incidental UI changes.

Formally, the state hash is defined as:

$$h(S) = \mathrm{MD5}\left(\texttt{activity} \parallel \operatorname*{concat}_{j=1}^{m} \mathrm{MD5}(p_j)\right)$$

where p_j is the serialized stable descriptor of the j-th UI element. The descriptors are sorted to make the hash invariant to view hierarchy ordering.

`MD5`[3] is a hashing algorithm used here to generate fixed-length fingerprints of both the full state and individual UI element descriptors. It provides a compact, deterministic way to compare structured input for equality.

The term *hierarchy ordering* refers to the structure and ordering of elements returned by the Android UI automation layer. In practice, the XML view hierarchy can vary across identical screens in several ways: (i) the order of attributes within a node may change; (ii) the order of child nodes may be unstable; and (iii) the absolute screen coordinates of elements may shift slightly due to layout rendering or animation effects. These variations do not necessarily reflect any meaningful change in the user interface. To ensure hash stability under such non-semantic differences, all descriptors p_j are normalized and sorted before hashing. This normalization allows the crawler to treat visually identical but structurally reordered screens as the same logical UI state.

To account for screens that are visually different but structurally equivalent, the crawler includes several layers of comparison. When a newly observed state produces a hash collision with an existing vertex, the corresponding UI trees are structurally compared using a recursive attribute-matching algorithm. If the number and types of elements are identical and their bounding boxes and identifiers match within a threshold, the states are treated as equal and no new vertex is added.

If the structural diff detects small changes that do not justify a new node (e.g. label text or image content variations), the new state is added as a *virtual variant* of the parent. These virtual nodes are flagged and tracked separately from primary graph vertices. In cases where the structure is ambiguous, or the element identifiers are absent or repetitive, the crawler captures a cropped screenshot region for each such element and includes a perceptual hash of the crop in the element descriptor. This provides a fallback path to distinguish visually unique components that lack stable metadata.

States that are similar but not identical are still stored independently if the structural diff suggests different possible interactions or navigational affordances. This conservative approach avoids mistakenly collapsing distinct user-visible configurations.

[3] `https://www.ietf.org/rfc/rfc1321.txt`, the official specification of MD5 by the Internet Engineering Task Force (IETF).

4.5.5 Interaction and Exploration Strategies

The crawler explores each GUI state by attempting interactions on every actionable UI element. An interaction is defined as the application of a concrete user-level action to a specific UI element in a specific state. To avoid redundant or semantically inert actions, the crawler follows a fixed priority scheme when selecting which action to attempt. Each element supports a subset of the actions `click`, `long_click`, `toggle`, and `set_text`, based on its declared properties and inferred type. The priority order is:

$$\texttt{set_text} \prec \texttt{toggle} \prec \texttt{long_click} \prec \texttt{click}$$

This order reflects the assumption that text input and toggling are more likely to discover sub-states rather than completely new states, which means they have a lower degree of disruptiveness.

Each action is executed only once per element and per state unless the element is later reclassified. After performing an action, the crawler captures the resulting state and compares its hash against known states. If the hash is unchanged, the action is marked as non-state-changing and not repeated. If the action produces a new state or a virtual variant, an edge is added to the navigation graph and the resulting state is further explored.

Toggle widgets require additional logic. If a click does not change the screen hash, the element is tentatively marked as a toggle. The crawler performs a second click to test reversibility. If the second click restores the original hash, the element is confirmed to be binary and both toggled states are modeled as virtual vertices. These transitions are labeled with synthetic intent strings (`toggle_on`, `toggle_off`) and the toggle behavior is not explored further.

For composite structures such as number pickers and replicator containers, the crawler applies group-level logic. In a number picker, increment and decrement buttons are treated as independent actions that change an internal numeric value but not necessarily the whole screen state. A sequence of increment actions may be simulated to discover value ranges or trigger dynamic behaviors. Replicators, such as lists or grids, are handled by sampling representative children and applying actions to a subset, rather than fully traversing every repeated instance. To avoid redundant transitions, only the first few distinct elements within a replicator are explored unless their content appears to change dynamically.

Each element-action pair is recorded with metadata about its outcome: whether it caused a state change, whether the transition was reversible, and whether it exposed any new interactive elements. This metadata is used by the planner to avoid revisiting inert interactions in future paths and to prioritize transitions likely to yield new application states.

4.5.6 Navigation Graph Construction

The crawler maintains a directed multi-graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a unique UI state and each edge $e \in E$ represents a concrete user interaction that causes a transition between states. This graph is updated incrementally during crawling and serves as the primary data structure for modeling application behavior.

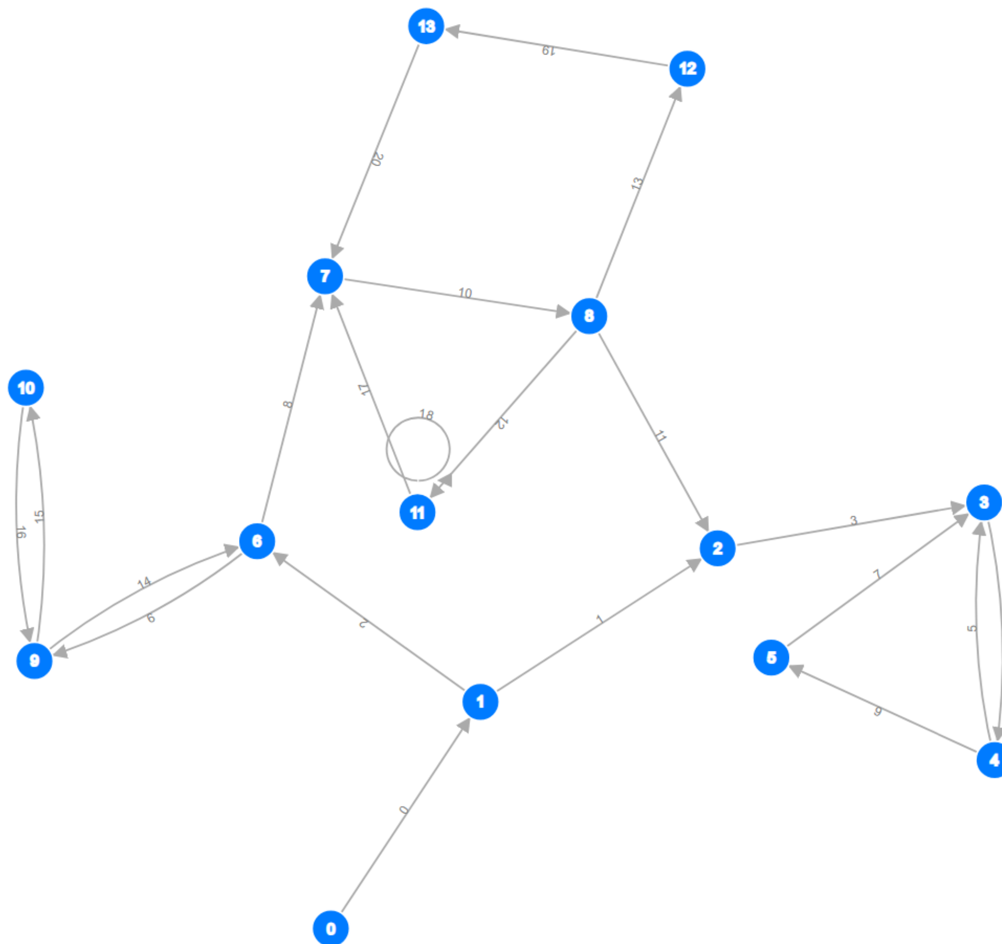


Figure 4.4: Navigation graph from a crawling session of an Android Application.

Each vertex is keyed by a state hash and stores the full abstract state S, including the UI elements, activity name, semantic annotations, and a screenshot thumbnail. Vertices are flagged as either *primary* or *virtual*, depending on whether they represent distinct application screens or intermediate configurations introduced by toggles or similar behaviors. Virtual vertices are not explored recursively but are connected by reversible edges to their parent nodes.

Edges encode the user interaction that triggered a transition from one state to another. Each edge is labeled with the action type (e.g., `click`, `toggle`), the element's stable XPath, its center coordinate on the screen, and an optional semantic intent string. These intent labels are derived from resource identifiers or inferred using the language model and are used to support intent-based test generation.

When the crawler executes an action on an element in state S and observes a resulting state S', it computes the hash $h' = h(S')$. If h' does not exist in the graph, a new vertex is added and linked from the current state. If h' matches an existing vertex, a new edge is added from the current state to that vertex. If the content differs only slightly, the new state may be added as a virtual child of the matched node, as explained in Section 4.5.4.

The graph also stores metadata used for navigation and analysis, including whether a vertex has been fully explored, whether an edge is reversible, and which elements have unexplored actions. Edges can also be annotated post hoc with additional metadata such as execution success or failure, timeouts, or abnormal terminations.

The graph structure supports multiple downstream queries: shortest path retrieval between arbitrary states, matching of screen states to intent descriptions, detection of navigation loops, and extraction of subgraphs rooted at semantically important states. All crawl-time decisions and post-processing analysis operate over this shared graph representation.

4.5.7 Intent Resolution and Semantic Annotation

The crawler utilizes an LLM to augment the navigation graph with semantic information. This is done in a constrained and strategic manner, without allowing the model to influence any control flow or decision-making within the exploration algorithm. The model is invoked only when the built-in metadata of a UI element or state is insufficient to describe its purpose or content.

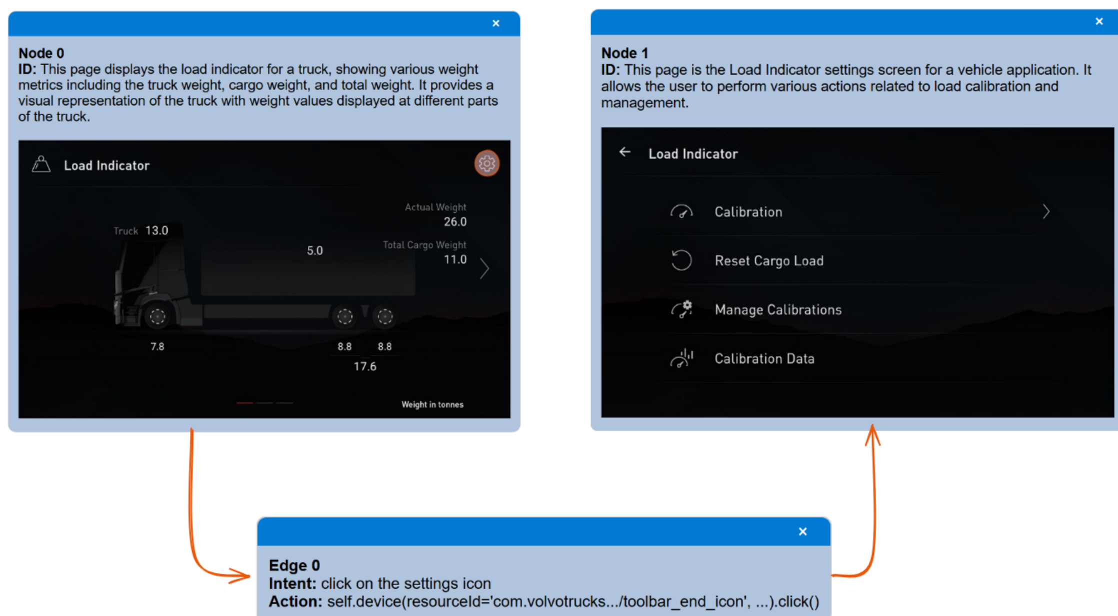


Figure 4.5: An example of semantically augmented nodes and edges from 4.4 where a settings button – highlighted by a transparent circle – is clicked in `Node 0` that gets connected to `Node 1` via `Edge 0`.

When a new UI element lacks a descriptive text label or resource identifier, the crawler extracts relevant context such as visible text, class name, and screen location, then queries the language model to generate a short name and a coarse intent string as can be seen in 4.5. These fields are stored alongside the element and associated with any outgoing edges that result from interacting with it. Examples of generated intent labels include descriptions like "navigate to profile settings" or "submit form."

Similarly, for each unique UI state, the crawler may request a short summary title and a one-line description. This is done only once per state and cached in the graph. These summaries improve the interpretability of the graph and assist downstream tools that generate tests from descriptions.

The semantic fields are optional and used only to annotate the graph. No interaction selection, path prioritization, or state comparison depends on the output of the language model. The crawler is designed to function fully without these annotations, and the presence or absence of a label does not influence its behavior.

4.5.8 Crawling Algorithm

Algorithm 1 summarizes the exploration loop; it proceeds until no vertex contains an unexplored interactive element.

Algorithm 1: Android GUI Crawling Loop

```
1  S ← CaptureState();
2  h ← Hash(S);
3  InsertVertex(h, S);
4  while true do
5      (h_target, i) ← NextUnexplored();
6      if h_target = nil then
7          break;
8      end
9      if h ≠ h_target then
10         NavigateTo(h_target);
11         h ← h_target;
12     end
13     a ← SelectAction(h, i);
14     Execute(a);
15     S′ ← CaptureState();
16     h′ ← Hash(S′);
17     ProcessTransition(h, a, h′, S′);
18     h ← h′;
19 end
```

Explanation of Algorithm 1. The crawler begins by capturing the initial GUI state and inserting it as a vertex in the exploration graph. On each iteration, it invokes NextUnexplored(), which returns a pair (h_{target}, i) representing the i-th

unexplored interactive element in a previously visited state with hash h_{target}. If the current state h is not equal to h_{target}, the crawler navigates back to it using NavigateTo(), which replays a known path through the interaction graph.

Once at the correct state, the crawler selects and executes the i-th action using SelectAction() and Execute(). The resulting UI state is captured, hashed, and added to the graph via ProcessTransition(). This loop continues until all reachable interactive elements have been explored.

The function NextUnexplored() selects the next unexplored action based on a priority ordering defined by interaction type and availability. These heuristics are described in Section 4.5.5, and guide the crawler toward interactions that are more likely to yield meaningful state transitions.

Termination Guarantee

In principle, the crawling loop halts once every vertex has no remaining unexplored element–action pair. This assumes that the set of reachable states is finite and that the hash function correctly identifies all structurally equivalent screens. However, in practice, this assumption does not always hold.

Android provides no reliable or stable mechanism for uniquely identifying GUI states across time. Neither the view hierarchy, nor `logcat`[4], nor low-level diagnostics such as `surfaceflinger`[5] or `dumpsys`[6] expose a persistent identifier or canonical structure for UI states. As a result, hash collisions and near-duplicates are both possible and unavoidable.

When the hash function under-approximates state identity, the crawler may treat visually different but functionally identical screens as distinct. This leads to state space explosion, particularly in apps that are highly dynamic. Conversely, when the hash function over-approximates and collapses states that differ in behaviorally relevant ways, some transitions may be lost or misclassified.

Therefore, the crawler does not have a formal termination guarantee in the general case. Its practical termination depends on the stability of the app under test, the reliability of state hashing, and the degree of structural noise in the UI. In well-behaved applications with static or semi-static screens, the crawl typically completes in finite time with manageable graph size. In highly dynamic applications, manual intervention or additional heuristics may be required to limit or bound the crawl.

[4] `https://developer.android.com/tools/logcat`
[5] `https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager`
[6] `https://developer.android.com/tools/dumpsys`

4.6 LLM-Based Crawler

4.6.1 Motivation

The original crawler implementation used algorithmic heuristics and tree-based XML comparison to explore the GUI application. This method was limited by its inability to reliably identify whether a new screen represented a unique application state. Small layout shifts or structural differences in XML often led to duplicated traversal or missed state transitions. To address this, a second crawler design was implemented. Unlike the first, this version delegates most decision-making to a LLM, using visual and semantic descriptions of the interface rather than structural comparisons.

4.6.2 Screen Representation

Each screen is represented using two primary sources: a screenshot and its corresponding UI XML hierarchy. The system collects both and passes them into a module called `DescribeScreen`, which returns a textual title, a semantic description of the interface, and a list of actions that can be taken. This information is stored as a representation of the current screen. Screens that involve scrollable containers use the `DescribeScreens` module, which aggregates descriptions from multiple vertically scrollable subviews.

4.6.3 State Tracking

The crawler attempts to determine whether the current screen has already been explored. This process uses a hybrid approach. First, image similarity is computed using the Structural Similarity Index Measure (SSIM[7]). If the SSIM between the current screen and any previous screen exceeds a fixed threshold (≥ 0.95), the system considers them visually similar. Second, the LLM module `IsUniqueState` is invoked to evaluate whether the semantic description of the current screen matches any stored descriptions. Only if both conditions suggest duplication is the screen treated as a previously seen state. Otherwise, the crawler assumes a new state and records it.

This approach does not fully eliminate the state detection problem. Visual similarity does not guarantee semantic equivalence, and LLM-based descriptions may introduce inconsistency or aliasing. As a result, the system's memory may contain duplicate or overly merged states. This design is functional for limited runs of the crawler where the memory is used only as an approximate record of visited states.

4.6.4 Exploration Procedure

The exploration loop consists of the following steps:

[7]SSIM is a method for measuring the similarity between two images. It evaluates perceived changes in structural information by comparing patterns of pixel intensities, and returns a score between -1 and 1, where 1 indicates perfect similarity.

1. Capture the current screen image and XML hierarchy.
2. Describe the screen using `DescribeScreen` or `DescribeScreens`.
3. Determine whether the state has already been seen.
4. Select an unexplored action from the list associated with the current screen.
5. Use the `Execute` module to generate code that performs the action.
6. Run the code, capture the post-execution screen, and classify the change using `GetChangingType`.
7. Update the memory:
   - If the action caused a layout change, add a new node and edge in the memory graph.
   - If the action only altered element states or failed, record the result accordingly.
8. If there are no more unexplored actions from the current state, resume from another state that still has unexplored transitions.

Each node in the memory graph stores a screen title, description, actions, and screenshot. Each edge represents an action taken between two screens, along with the code used.

4.7 Artifact Design and Implementation

This study develops an AI agent prototype system designed to support automated UI test generation for Android apps based on user intent. The system is implemented in Python and integrates several key components:

- An Evaluator–Optimizer workflow, which systematically evaluates LLM-generated outputs and fine-tunes prompt parameters to improve accuracy, alignment, and overall performance.
- GPT-4o, a multi-modal large language model used to interpret natural language intents and generate test logic;
- uiautomator2, a tool for interacting with and controlling Android devices at the UI level;

These parts work together so the AI agent system could take a user instruction, understand the current screen of the Android app, and choose the best actions to perform, similar to what a human tester would do.

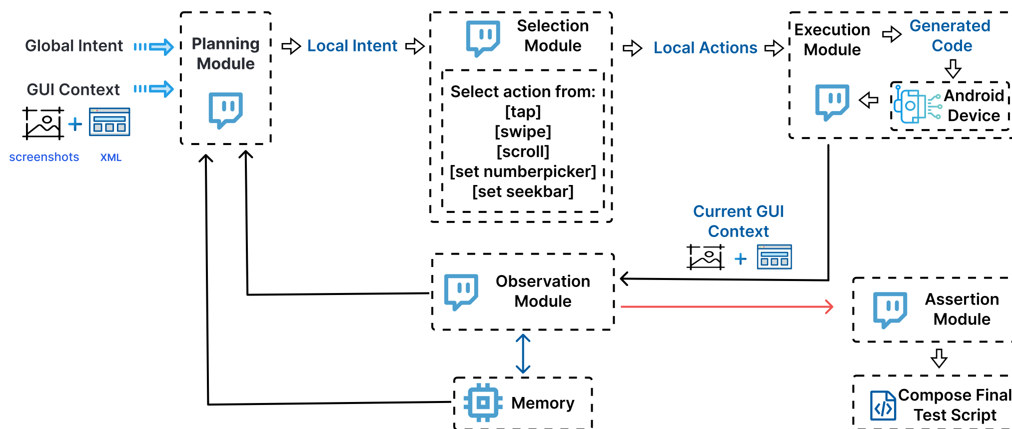


Figure 4.6: Overview of the system's workflow for intent-driven test generation.

4.8 System Architecture and Implementation Details

The system is written in Python 3.10 and follows a modular design, based on the structure shown in Figure 4.6. Each module has its own role and handles one part of the test generation and execution pipeline.

4.8.1 Modules using LLMs

The robot-like blue chat icon represents the LLMs are used inside the module, e.g. use LLMs to generate code, or use LLMs as a judge. In our system, the Planning Module and Selection Module use LLMs as decision maker, the Execution Module and Assertion Module use LLMs as code generator and the Observation Module use LLMs as judge and reasoner.

4.8.2 Planning Module

First, let us define the inputs and outputs to this module. The requirements for the test case, which can also be understood as the ultimate goal we want the system to achieve at the end of the loop, is called global intent[1].

Global Intent. A natural language string representing the user's input that serves as the overall guiding objective for the system. The global intent is sent to the Planning Module, where it is decomposed into a sequence of local intents for step-by-step execution. It is also be sent to the observation module, which continuously evaluates whether the system has successfully fulfilled the user's intended goal. This evaluation governs the control flow of the Evaluator–Optimizer Workflow: if the global intent is satisfied, the loop terminates.

Another input to the Planning Module is the GUI context[2], which provides visual

information about the current UI state and utilize the multi-modal ability of modern language models.

GUI Context. The GUI context represents the observable state of the application at a specific moment in time. It consists of a list of current screenshots capturing the visible UI and an accompanying XML hierarchy file that encodes metadata about the underlying UI elements—such as their types, positions, properties, and relationships.

The output of this module is the local intent[3], a single string contains the next immediate action to be perform, the action has certain types for the language model to select from, and will be generated with details indicating the UI elements to interact with or certain values to be input. for e.g. Tap on on the 'Edit' button under the first Alarm.

Local Intent. A decomposed intent from Global Intent to be performed on the current app screen, generated by the Planning Module. The generation of this local intent is subject to several constraints: (i) all actions must operate strictly within the current screen context (i.e., visible screenshot(s)); (ii) once an action may cause a page transition, no further actions should be chained after it; (iii) the output must focus solely on accomplishing the next immediate step toward fulfilling the global intent, without over-executing; (iv) for scrolling actions, the specific scrollable list must be identified; and (v) if a scroll is intended to locate a specific string, that string must be explicitly mentioned in the step.

Together, the global intent, GUI context, and the reasoning output from the Observation Module—captured after the last set of local actions—are passed as input to the language model. This enables comprehensive understanding and reasoning about the current task. Using chain-of-thought prompting, the language model generates the most appropriate next action to be performed, referred to as the local intent. In addition, the model outputs a textual reasoning trace, which reveals the intermediate thought process behind the decision and allows for human inspection and evaluation of its reasoning quality.

This module act as the decision maker of the entire system, it guides the system by outputting local intent as next step for Execution Module to run local actions, then the result of local actions[4] will be captured by Observation Module and send back to Planning Module for better reasoning.

4.8.3 Selection Module

Local Actions. An single action step generated by the Selection Module, translated from the local intent into actionable commands under the Volvo Android environment. The action represent one or multiple UI operations required to fulfill the specified local intent. These actions are executed on the real application using `uiautomator2`, enabling direct interaction with the device's user interface.

The Selection Module take the local intent as input, send it to LLMs for tool selection and generate Local Action[4] as output. The tool selection involves a list of pre-defined tools and limits the LLMs to pick the most appropriate one to use under current app screen. The local action must be one of the predefined primitive types: `tap`, `swipe`, `scroll`, `set_numberpicker`, `set_seekbar`. The tool list contains normal Android operations like Tap, Swipe and Scroll. There are also certain types of special UI actions like `set_numberpicker` or `set_seekbar`, which involve a complicated and correctly ordered series of actions like 'long click' to enter edit mode, 'set text' to set the digit, 'click' to save the value of this digit. We use a few-shot prompting to guide the language model to generate a correct series of local actions.

4.8.4 Execution Module

This module receives the local actions, send it to the language model, and language model translates it into a set of python code that can be executed on the Android device.

We use `uiautomator2` to perform actions such as clicking buttons, swiping, or entering text, enabling direct interaction with the device's user interface. For example, a local action such as "tap the 'Save' button" may be translated into `device(text="Save").click()`, while a local action like "scroll to the item labeled 'Volume'" could result in `device(scrollable=True).scroll.to(text="Volume")`. The execution result, along with any error messages, is then passed to the Observation Module as the outcome of the Execution Module.

4.8.5 Observation Module

After the python code translated from local actions have been executed on the app device, the application may transition to a new state (e.g., a different screen or modified UI elements) or remain in the original state, which may indicate that the actions were unsuccessful. To determine which path the app has taken and to obtain a visual understanding of the current state, we recapture the GUI context and pass it with the execution result of local actions and the global intent, to the language model.

The language model is first prompted to assess whether the performed local actions have successfully fulfilled the global intent. If so, the system exits the Evaluator–Optimizer loop. If not, the model is instructed to reason about why the global intent remains unfulfilled. This reasoning includes: (i) whether the app state has changed as expected; (ii) whether the local actions were successfully executed; (iii) the degree to which the global intent has been fulfilled; and (iv) the remaining steps required to achieve full completion of the intent.

The reasoning will be sent back to the Planning Module for better understanding the next move.

4.8.6 Memory

The memory component maintains a record of previously visited application screens and supports the identification of already-explored states. Each time a new screen is encountered, the framework captures a screenshot and its corresponding XML hierarchy. These inputs are passed to a summarization module that uses an LLM to produce a textual description of the screen. The prompt includes constraints and examples to constrain the model, increase consistency, and limit irrelevant output. This textual description becomes the key under which the screen's metadata is stored.

The metadata includes the screenshot, XML hierarchy, and information about the visible UI elements and their bounding boxes. Because both the image and bounds are available, individual elements can be isolated by cropping the relevant region from the screenshot. These stored elements and descriptions are used later to compare new screens against previously observed ones.

To determine whether a newly visited screen has already been seen, the framework generates a fresh description and searches for similar entries in memory. If candidates are found, it compares the new screenshot against stored screenshots using SSIM[8]. If the similarity score exceeds a threshold (e.g., 0.95), the screen is considered visually identical to a known state. This comparison is performed over the entire image.

Using SSIM alone is not always sufficient. In Android applications, screens may include dynamic content such as timestamps, small status messages, or slightly changed icons. These can cause minor visual differences without affecting the actual layout or meaning of the screen. A high SSIM score despite these small variations is usually a good indicator that the screens are functionally the same.

In addition to image similarity, the framework also considers how the screen was reached. If the current path through the app (for example, moving from screen A to B to C) matches a previously recorded path that also ended at a similar-looking screen, this provides further evidence that the two states are the same. The description, screenshot, XML hierarchy, and navigation path together form a more clear context.

When needed, the framework can provide the LLM with a shortlist of similar memory entries. The model is then asked whether the current screen should be treated as new or already visited. This additional step helps disambiguate borderline cases where image similarity is high but uncertainty remains. Combining SSIM, semantic description, execution path, and LLM judgment provides a more reliable basis for determining whether a screen is new or already known.

[8]SSIM (Structural Similarity Index) is a method for measuring the similarity between two images. It evaluates perceived changes in structural information by comparing patterns of pixel intensities, and returns a score between -1 and 1, where 1 indicates perfect similarity.

4.8.7 Evaluator–Optimizer Workflow

The Evaluator–Optimizer Workflow is a structured pattern commonly used in agentic systems and LLM-based pipelines to iteratively improve output quality. In this workflow, an *optimizer* generates an initial response or action, and an *evaluator* assesses its quality based on predefined criteria. The evaluation result is then used to refine the next output, enabling continuous feedback and learning [8].

In our system, the Planning Module, Execution Module, and Observation Module collectively implement this workflow. The process starts by evaluating how well the generated actions match the user's intent and how accurate those actions are when executed on the real UI. The evaluation outcomes are then used to update prompt parameters, adjust prompt structure, and guide the Planning Module toward more accurate decisions.

This feedback loop ensures that the system keeps improving as it executes more tests and observes more GUI contexts. The result is a dynamic, learning-capable AI agent that can behave like a human tester: understanding user intent, planning next steps, executing actions, and validating outcomes—all while refining its behavior in real time.

This AI agent system is a key part of the research. It shows how LLMs can be used in Android testing. It also lets us run tests to check how well it works by measuring things like test coverage, how well it matches the user's intent, and if the actions succeed on real Android apps used at Volvo Trucks.

4.8.8 Assertion Module

After the global intent has been fulfilled, the system exits the Evaluator–Optimizer Workflow and transitions to the Assertion Module. The Assertion Module take the complete code snippets as input to LLMs and let it generate reasonable assertions line by line to the input code snippets. At this point, assertions are added to each code snippet produced from the local actions. All these annotated code snippets are then assembled to compose a complete test case corresponding to the original global intent. Then the entire code snippet will be sent as the Compose Final Test Script, which is the output of the entire workflow.

4.8.9 Network Sniffer Integration

Certain UI states in Volvo Android applications are tightly coupled to backend behavior. For example, some pop-up dialogs or confirmation messages are triggered only after a specific API response is received. Similarly, application settings may be populated dynamically based on periodic polling of backend services. These interactions are not observable through the UI hierarchy alone and cannot be reliably inferred from screen transitions. This made it necessary to design a mechanism that could capture and analyze backend traffic associated with UI actions.

To address this, we implemented a custom packet sniffer and integrated it into the

framework. The goal was to observe HTTP traffic in response to UI events and extract information such as endpoint, method, status code, and request/response payloads. This information can be used for reverse engineering undocumented behavior and supporting test assertions that depend on backend responses. Additionally, identifying polling mechanisms or control requests helps in reconstructing payloads to restore application state in later tests.

Rather than relying on high-level packet capture libraries such as PyShark[9], the sniffer was implemented from scratch to allow fine-grained control over both the packet parsing and the capture strategy. This decision was motivated by two constraints: first, many execution environments had strict limitations on installing external dependencies; and second, we required full control over when and how packets were filtered, parsed, and assembled. The sniffer operates locally or remotely via SSH, with support for interface discovery and scoped, timed capture windows.

At the core of the sniffer is a TCP assembler. This component is responsible for grouping raw packets into bidirectional TCP connections. It tracks sequence numbers and connection metadata to avoid duplicate payloads and incomplete streams. It detects connection setup (SYN flag), teardown (FIN or RST)[10], and classifies each connection type as transient, persistent, WebSocket, or event-stream based on observed headers. All payloads for each direction are accumulated separately and stored once the connection is closed on both ends.

Once assembled, each TCP stream is parsed by a custom HTTP parser. This parser extracts fields from both the client and server side, including method, endpoint path, status code, headers, and content-type. If the content-type suggests a JSON payload, the parser attempts to deserialize it. The parser also includes logic to detect Volvo-specific media types and extract application name and version identifiers. Parsed results are stored as structured objects and logged to file for later analysis.

The sniffer supports both continuous capture and scoped capture. In scoped capture mode, the sniffer records only the transactions that occur during a defined window, typically tied to a UI action. This is implemented via a context manager that synchronizes packet sniffing with the start and end of the UI interaction. The mechanism is integrated at the HTTP request layer of the device control interface, so that any click or swipe can be wrapped with packet capture automatically. The duration of capture is configurable and defaults to a short window sufficient to include most API responses.

Captured transactions can be filtered post-capture using a flexible filtering system. Transactions can be included or excluded based on criteria such as HTTP method, status code, endpoint, or application name. These filters are used to suppress background noise or isolate specific types of traffic, such as polling endpoints

[9] PyShark is a Python wrapper for the Wireshark packet analyzer. It provides access to parsed packet fields but depends on tshark and is less suitable for constrained environments.

[10] SYN, FIN, and RST are TCP control flags. SYN initiates a connection, FIN signals an orderly shutdown, and RST indicates an immediate termination of the connection.

or user-triggered control messages.

The packet sniffer is not actively integrated into the core framework beyond basic logging. Its current role is limited to passive observation and payload capture, without influencing test generation or execution. However, it establishes the groundwork for more advanced functionality. Future iterations could support automated generation of backend assertions, classification of traffic types, or active replay of requests to emulate specific application states. The implementation was built to be modular and extensible for these purposes.

4.9 RQ1 Evaluation: Code-Level Comparison Between Generated and Manual Tests

To evaluate RQ1 — *"How effective is the proposed framework in generating valid and executable test code for developer-defined intents?"*, we define **effectiveness** as the ability of a test to correctly, reliably, and clearly express a developer's intent with minimal manual intervention or maintenance overhead. A generated test is effective if it (i) performs the intended interactions correctly, (ii) exhibits stable behavior across repeated executions, (iii) avoids redundant operations, and (iv) is written in a form that is understandable and maintainable by human engineers. Based on this definition, our evaluation includes only the generated tests that passed, along with their corresponding manually written counterparts that also passed, which serve as the ground truth. All tests are assessed across five structured metrics. Failed tests are excluded because the structured metrics—such as robustness and readability—are designed to evaluate the quality of valid and executable test cases.

4.9.1 Line-Level Correctness

Line-Level Correctness in software engineering refers to the accuracy and reliability of individual lines of code within a program. It assesses whether each line performs its intended function correctly and contributes appropriately to the overall behavior of the software. In our system, we measure the Line-level correctness by annotated each line as either **correct** or **incorrect**. A line is considered correct if it contributes directly to achieving the developer's specified intent through appropriate UI interaction or validation. Lines that target the wrong elements, perform actions not aligned with the test goal, or introduce unintended state changes are labeled incorrect.

This metric directly quantifies the functional precision of the generated code. An effective test must exhibit high correctness, meaning that nearly every line plays a valid and necessary role in fulfilling the intent. This is the most central metric in determining whether the generated output is actually valid and usable as a test.

4.9.2 Unnecessary Steps

Unnecessary steps refer to actions in a test script that do not contribute to fulfilling the intended goal or verifying the desired behavior of the system under test. We identify and count operations that do not contribute meaningfully to achieving the intended goal of our test. These are annotated manually and include:

- repeated navigations already covered by previous lines,
- sleep delays without a corresponding asynchronous dependency,
- hard–coded values or paths that do not influence test success.

Recent study by Amalfitano et al. also suggest that structured exploration leads to more efficient and meaningful test coverage[3], and study by Wang et al. discussed how such unnecessary navigations can significantly inflating test duration and diverting focus from intended test goals[37]. In our evaluation, we use this criterion to assess how well the generated test code remains focused and efficient in fulfilling its intended purpose.

4.9.3 Flakiness

Flakiness is measured as a binary attribute. A test is marked flaky if it produces inconsistent results (i.e., passes and fails non-deterministically) across three consecutive runs on a stable device configuration with unchanged application state.

Flaky tests are considered ineffective by definition, since their results cannot be trusted in regression analysis or continuous integration pipelines. Even a semantically correct test becomes unusable if it behaves inconsistently under identical conditions. Therefore, absence of flakiness is treated as a necessary condition for test effectiveness.

4.9.4 Robustness

Robustness is the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions[22].

To evaluate this property in the context of generated test code, we inspect the degree to which a test can tolerate minor UI, timing, or state changes without failing. We assign robustness on a fixed 5–point scale based on presence of defensive constructs.

In industrial settings where network delays or minor UI shifts are common, robustness ensures that tests remain valid and maintainable over time. High robustness supports effectiveness by reducing the cost and frequency of manual rework.

Score	Definition
1	No assertions or handling; test fails immediately on any unexpected condition.
2	Sparse assertions; no retries or timeouts.
3	Includes basic assertions and a minimal form of retry or timeout handling.
4	Assertions on critical steps and conditional logic for error recovery.
5	Fully defensive: structured assertions, retries, timeouts, and fallback paths.

Table 4.1: Robustness Rating Scale

4.9.5 Readability

Readability captures the clarity and structure of the test code. This includes use of helper abstractions, meaningful naming, logical structure, and appropriate commenting. Each test is scored on a 5–point scale via dual human review, supplemented with `pylint` output.

Score	Definition
1	Flat structure, poor naming, no modularization or comments.
2	Basic structure with limited abstraction or clarity.
3	Adequate structure; readable but lacks modularity.
4	Well-structured with helper methods and consistent naming.
5	Highly readable; modular, clearly named, and well documented.

Table 4.2: Readability Rating Scale

Effective tests are not only functionally correct but also maintainable by others. High readability reduces onboarding time and debugging effort in collaborative and long–term codebases.

4.10 RQ2 Evaluation: Semantic Understanding and Planner Evaluation

To evaluate RQ2 — *"How accurately does the framework understand developer intents and translate them into meaningful and correct actions?"* — we perform a semantic alignment analysis between the input developers intent and planner generated actions.

We evaluate the ability to translate a global intent[1] into accurate local intent[3] using a two-part process comprising (i) quantitative action-correctness scoring and (ii) qualitative reasoning assessment. We evaluate a total of 42 planner-generated steps across three representative Volvo applications: Alarm Clock(N=16), System Settings (N=11), and Load Indicator(N=15). For each step, both the action correctness and reasoning quality were annotated manually.

Quantitative step Correctness

To assess the planner's ability to generate valid and executable steps, we sample 42 planner logs under real app contexts. Each input includes a global intent and GUI context[2]. These inputs are fed to the planner, which queries the language model and returns a single string that contains the next immediate step, plus a reasoning field of the context and the thinking behind this action step. Human annotators assess whether the generated action is valid under the context of the current app screen and action sequence:

- **Correct**: The action is executable and appropriate in context. Performing it advances the app meaningfully toward fulfilling the intended task.
- **Incorrect**: The action is irrelevant, misaligned, or targets an incorrect UI element. Performing it does not bring the app closer to fulfilling the intent or may even deviate from the intended task path.

We report binary correctness using the following counts:

- N_{total}: Total number of planner-generated steps evaluated.
- N_{correct}, $N_{\text{incorrect}}$: Number of actions labeled as correct or incorrect, based on criteria showing above.

Qualitative Reasoning Assessment

To assess the planner's reasoning quality, we apply a 5-point scoring rubric to each explanation string, regardless of whether the corresponding action was correct. The three apps were selected to cover a range of UI complexities: Alarm Clock features relatively static layouts, System Settings involves nested toggles and configurations, and Load Indicator presents dynamic lists and pop-up dialogs. This score reflects the planner's understanding of the input user intent and its use under the current

app context:

Score	Description
1	**Poor**: Illogical or hallucinated reasoning, reflects a misunderstanding of current screen or recognized wrong UI elements.
2	**Weak**: Partial understanding of current context, but flawed logic reasoning such as correct understanding of the current app functionality but misunderstanding the elements in the current screen.
3	**Moderate**: Plausible reasoning with key gaps, such as correct understanding of app context but provides an incorrect chain of thought.
4	**Good**: Mostly sound reasoning with minor issues, such as correct understanding of entire context but missed one or two components on the screen.
5	**Excellent**: Clear, contextually sound reasoning aligned with intent, totally correct understanding of entire context, strong reasoning of the app logic and functionalities.

Table 4.3: Reasoning Quality Rubric

Each step's reasoning explanation was scored independently of correctness, using the 5-point rubric in Table 4.3. Reasoning was assessed based on clarity, contextual understanding, and logical soundness. Scores were recorded by human annotators with domain knowledge of the app under test. Table 5.8 presents example global intents, planner steps, reasoning texts, and their corresponding scores.

We report reasoning quality scores for each app as μ and σ: Mean and standard deviation of reasoning scores, computed across all, correct, and incorrect steps, respectively. Aggregate statistics across apps (shown in the "Total" row of Table 5.3) are computed by summing across all steps from all apps. Mean and standard deviation values are calculated over the full set of reasoning scores.

4.11 RQ3 Evaluation: Prompting Strategies and Multi-Modal Effectiveness

Research Question 3 (RQ3) investigates the question: *"What are the most effective prompting strategies and modalities for enabling accurate and semantically grounded responses from multi-modal models given visual and textual input?"*

This study evaluates multi-modal prompting strategies through synthetic and isolated tests derived from key patterns used in the full system. While the deployed framework often combines multiple inference modules in a single pipeline (e.g., ID and instance resolution, or center prediction followed by action validation), this study isolates atomic behaviors. This enables precise measurement of modality contributions and avoids confounds from task chaining, history, or fallback logic.

4.11.1 Experimental Design

We define six task types that cover common prediction demands in UI automation:

- `click_xy`: predict the (x, y) location of a target element
- `click_id`: retrieve the element ID corresponding to a prompt
- `get_count`: count the number of visual or logical items
- `get_text`: return the exact or best-matching visible string
- `instance`: identify the index of a repeated element
- `seekbar`: compute the (x, y) position for setting a slider to a given value

Each task is specified as a standalone prompt and has a known correct output in structured form. All tasks require returning a valid JSON-compatible value for a known field (e.g., `center`, `resource_id`, `count`, etc.).

The dataset consists of 44 high-level prompts, decomposed into 480 atomic tasks spanning six task types. Each task is stored in a structured CSV and linked to a JSON file containing:

- a screenshot image (Base64-encoded PNG)
- the raw Android XML layout hierarchy
- the labeled answer fields for each task

Each task is evaluated under three input modality configurations—XML only (`x`), image only (`i`), and both (`xi`)—as summarized in Table 4.4. For each configuration, three deterministic seeds (42, 88, 96) are used, resulting in 1440 total model invocations. All evaluations use GPT-4o via a stateless API with `temperature=0`.

Configuration	XML	Image
`XML`	✓	–
`IMG`	–	✓
`XML` ⊕ `IMG`	✓	✓

Table 4.4: Modality configurations used for evaluation

Prompts are dynamically generated per task to predict a single output field (e.g., `text`, `target_center`). Each prompt includes a natural language instruction and the relevant inputs (XML layout, screenshot, or both), passed via labeled fields such as `task`, `xml_hierarchy`, and `image`. Prompts enforce schema consistency and field naming across all tasks.

No few-shot examples, retrieval augmentation, chain-of-thought reasoning, fallback logic, or external schema validation are used. Only the model's final structured prediction is scored. All outputs are logged with the prompt, gold label, and computed metrics for analysis.

4.11.2 Evaluation Metrics

Each predicted field is evaluated with metrics appropriate to its type. The following table summarizes the metrics used:

Field	Type	Metric	Metric Type
`element_center`	(int, int)	CIB (binary) L2 Manhattan	binary float int
`target_center`	(int, int)	L2 Manhattan	float int
`resource_id`	string	Exact match Edit distance	binary int
`text`	string	Exact match Edit distance	binary int
`count`	int	Exact match	binary
`instance`	int	Exact match	binary

Table 4.5: Evaluation metrics by field, showing input types, metric used, and the output type of each metric.

For coordinate predictions such as `element_center` and `target_center`, we report

the L2 (Euclidean[11]) distance and Manhattan[12] distance to the ground truth in pixel units. Additionally, for `element_center`, we include a binary Center-in-Box (CIB) metric indicating whether the predicted center lies within the gold bounding box. Here, `element_center` refers to the center of a UI element with defined bounds (e.g., a button), while `target_center` denotes a specific point of interaction, typically on continuous controls like sliders or seekbars.

String fields like `resource_id` and `text` are evaluated using case-sensitive exact match as the primary metric. We also compute character-level edit distance (Levenshtein[13]) to support fine-grained analysis, though it does not affect correctness scoring.

Integer fields, including `count` and `instance`, are evaluated solely via exact match. Any deviation from the gold value is treated as incorrect.

All predictions are parsed deterministically. Missing, malformed, or improperly typed outputs are scored as incorrect. There is no learned evaluator and no tolerance via soft thresholds or fuzzy matching.

4.11.3 Limitations

This evaluation isolates each task to measure model performance on a single prompt with known inputs. It does not evaluate recovery, chaining, or interaction planning, which are part of the broader system. For instance, the main framework includes validation checks (e.g., verifying a predicted center lies within a bounding box or that an element exists before interaction), but these are excluded here to maintain test purity.

Likewise, prompt design is uniform and does not include demonstrations, retrieval augmentation, or few-shot conditioning. This maximizes internal consistency but may underestimate potential performance achievable with richer context or in-system integrations.

[11]Euclidean (L2) distance is the straight-line distance between two points in pixel space, computed as the square root of the sum of squared differences across axes.

[12]Manhattan distance is the sum of absolute differences across axes, reflecting axis-aligned movement between two points.

[13]Levenshtein distance measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into another.

5 Results

5.1 Test Selection and Comparison Overview

As shown in Table 5.1, from the broader pool of test cases created manually by Volvo developers, we selected a subset and compared our results against 11 manual tests for the Alarm Clock, 16 for System Settings, and 60 for the Load Indicator. We use the global intent[1] as input to our Planning Module. However, a portion of the manual tests includes verification of backend API data and external link checks, which are currently beyond the scope of our system. Therefore, we exclude these tests from our evaluation. As a result, we focus on 9 automated tests for the Alarm Clock, 14 automated tests for System Settings, and 22 automated tests for the Load Indicator.

The test scenarios were defined first by Volvo's team as manual test cases; we then used the exact same high-level intents when generating automated (A) tests. Consequently, every manual test (M) has a one-to-one counterpart in the automated set. The test cases that failed in both automated and manual testing are identical.

App	Method	Total	Passed	Failed	Expected Failures
Alarm Clock	M	11	7	4	0
	A	9	5	4	0
System Settings	M	16	16	0	0
	A	14	6	8	0
Load Indicator	M	60	53	4	3
	A	22	16	6	0
Total	M	87	76	8	3
	A	45	27	18	0

Table 5.1: Test Result Comparison: Manual (M) vs Automated (A), the manual test pass rate is 87% and automated test pass rate is 60%.

5.1.1 Example Manual vs. Automated Test Cases

To illustrate the differences captured by our code-level metrics, we present a representative pair of test scripts targeting the same user intent in the Alarm Clock app. Both test cases fulfill the following scenario:

Click on the edit button of Alarm1 and once it opens just click the save button. Navigate to the second page of the app. Click on the edit button of Alarm4 and once it opens just click the save button. Finally, return to the home screen.

Manual Test Case

The manually written test uses helper methods and is well-annotated with semantic markers such as preconditions, expected behavior, and system-level tags. It abstracts UI actions through reusable domain-specific methods for maintainability.

Automated Test Case

The generated test explicitly encodes every UI interaction using raw device commands. While this ensures transparency and functional correctness, it also leads to verbosity, repetition, and reliance on fixed delays. Unlike the manual tests, which use heavily abstracted layers that can sometimes obscure intent and make maintenance harder, the generated test remains straightforward.

5.2 RQ1 Results: Code-Level Comparison Between Generated and Manual Tests

As described in Section 4.9, we evaluate five dimensions of metrics including Line-level Correctness, Unneccesary steps, Flakiness, Robustness, Readability. The evaluation is performed on passed manual and generated test cases, including 5 for Alarm Clock, 6 for System Settings, and 16 for Load Indicator.

Table 5.2 presents the results. Each row corresponds to a test case, where the **ID** column indicates the identifier of the original manual test case used by Volvo developers, which serves as a traceable reference for aligning generated and manual test pairs. **M** denotes a manually written test and **A** denotes a generated (automated) test. The table enables side-by-side comparison between manual and automated tests targeting the same functionality.

App	ID	Type	Correct (%)	Unnec. Steps	Flaky	Robust	Readable
Alarm Clock							
	A01	M	100	0	F	4	3
		A	100	0	F	3	3
	A06	M	100	2	F	4	3
		A	100	0	F	3	2
	A08	M	100	2	F	5	3
		A	83	1	T	2	2
	A10	M	100	2	F	5	3
		A	82	0	T	3	3
	A14	M	100	1	F	5	3
		A	85	1	F	3	3
System Settings							
	S04	M	100	1	F	4	4
		A	65	1	T	2	3
	S05	M	100	1	F	4	4
		A	60	1	T	2	2
	S06	M	100	1	F	4	4
		A	57	4	T	1	2
	S08	M	100	1	F	4	4
		A	53	4	F	2	2
	S09	M	100	1	F	4	4
		A	53	3	T	2	2
	S10	M	100	1	F	4	4
		A	60	3	T	2	2
Load Indicator							
	L02	M	100	0	F	4	4
		A	44	1	F	2	2
	L08	M	100	1	F	5	4
		A	46	4	F	1	2
	L09	M	100	1	F	5	4
		A	54	4	F	2	2
	L12	M	100	1	F	5	4
		A	29	8	T	1	1
	L13	M	100	1	F	5	4
		A	29	7	T	1	1
	L16	M	100	1	F	4	4
		A	33	5	F	1	1
	L20	M	100	1	F	5	4
		A	50	3	F	2	2
	L21	M	100	1	F	4	4
		A	40	7	F	1	2
	L22	M	100	1	F	5	4
		A	33	5	F	1	2
	L23	M	100	1	F	4	4
		A	50	1	F	1	2
	L24	M	100	1	F	4	4
		A	57	0	F	1	3
	L25	M	100	1	F	3	4
		A	40	3	F	1	2
	L26	M	100	1	F	5	4
		A	33	2	F	1	2
	L27	M	100	0	F	5	5
		A	15	6	F	1	1
	L28	M	100	0	F	5	5
		A	20	2	F	2	1
	L29	M	100	0	F	5	5
		A	18	6	F	1	1

Table 5.2: Code-Level Comparison Between Manual (M) and LLM-Generated (A) Tests

5.3 RQ2 Results: Semantic Understanding and Planning Module Evaluation

As described in Section 4.10, we assess the semantic alignment between developer global intents[1] and the Planning Module generated local intents[3] using both quantitative and qualitative measures. Specifically, we evaluated 42 Planning Module logs across three applications–Alarm Clock (N=16), System Settings (N=11), and Load Indicator (N=15).

Each low-level intent is annotated for **action correctness** (correct or incorrect) and the reasoning field generated together with low-level intent is rated on **reasoning quality** using a 5-point rubric.

Table 5.3 summarizes the evaluation results. For each application, we report the total number of Planning Module generated low-level intent (N_{total}), the number of correct and incorrect intents (N_{correct}, $N_{\text{incorrect}}$), and the average reasoning score (μ) with standard deviation (σ) across all, correct-only, and incorrect-only low-level intents.

App	N_{total}	N_{correct}	$N_{\text{incorrect}}$	μ_{total}	σ_{total}	μ_{correct}	σ_{correct}	$\mu_{\text{incorrect}}$	$\sigma_{\text{incorrect}}$
Alarm Clock	16	14	2	4.00	1.32	4.29	1.07	2.00	1.41
System Settings	11	8	3	3.91	1.70	4.88	0.35	1.33	0.58
Load Indicator	15	8	7	3.13	1.06	3.88	0.83	2.29	0.49
Aggregate	42	30	12	3.67	1.37	4.33	0.92	2.00	0.74

Table 5.3: RQ2 Quantitative step Correctness and Qualitative Reasoning Assessment for three different Volvo apps

Table 5.4 presents the average reasoning scores and binary equivalence outcomes for all 35 automated tests paired with their corresponding manual tests.

Test ID	Equivalent	Average Reasoning Score
A01	T	5.00
A02	F	2.14
A03	F	2.65
A05	F	2.15
A06	T	4.83
A08	T	4.97
A10	T	5.00
A12	F	2.78
A14	T	5.00
S04	T	4.36
S05	T	4.85
S06	T	4.02
S07	T	4.44
S08	T	3.46
S09	T	4.00
S10	T	4.70
S11	T	4.92
S12	F	3.47
S13	F	1.28
S14	F	1.89
S15	F	2.33
S16	F	2.01
S17	F	3.08
L01	F	1.87
L02	F	1.67
L03	F	2.36
L04	F	1.33
L05	F	1.79
L08	T	3.52
L09	T	4.61
L10	F	2.25
L12	T	3.04
L13	F	2.49
L16	T	3.68
L19	F	2.24
L20	T	4.57
L21	T	3.08
L22	F	1.55
L23	F	1.17
L24	F	2.64
L25	F	2.01
L26	F	2.87
L27	F	1.83
L28	T	2.36
L29	F	1.98

Table 5.4: Average Reasoning Score by Test ID and Equivalence Case

Table 5.5 presents one example of an automated test case that **is equivalent** to its corresponding manual test. It lists the local intents generated by the Planning Module, their correctness, and the associated reasoning scores for each local intent.

Local Intent	Correct	Score
Tap on the 'Edit' button for Alarm1	T	5
Tap the 'Save' button in top-right	T	5
Tap on right arrow to go to next page	T	5
Tap on the 'Edit' button for Alarm4	T	5
Tap the 'Save' button in top-right	T	5
Tap the 'Home' button	T	5
Total Correct	*6/6*	*Average = 5.0*

Table 5.5: Step-by-step evaluation of local intents in an equivalence case. Scores range from 1 (poor) to 5 (excellent).

Table 5.6 presents one example of an automated test case that **is not equivalent** to its corresponding manual test. It lists the local intents generated by the Planning Module, their correctness, and the associated reasoning scores for each local intent.

Local Intent	Correct	Score
Tap on the 'Edit' button for Alarm1	T	5
Tap on the "Save" button to save changes to Alarm1	F	3
Tap on the right chevron icon to navigate to next alarms	T	2
Tap on the "Edit" button for second alarm	F	2
Tap on the toggle button to enable Repeat for fifth alarm	F	2
Tap on the toggle button to enable Repeat for fifth alarm	F	2
Total Correct	*2/6*	*Average = 2.67*

Table 5.6: Step-by-step evaluation of local intents in an unequivalence case. Scores range from 1 (poor) to 5 (excellent).

Table 5.7 presents the correlation results computed from the data in Table 5.4, quantifying the relationship between average reasoning score and test equivalence using Spearman Rank and Point-Biserial coefficients. Both metrics indicate a strong positive association, with the point-biserial coefficient showing particularly high alignment.

Correlation Type	Coefficient	p-value
Spearman Rank	0.820	5.706×10^{-12}
Point-Biserial	0.842	4.165×10^{-13}

Table 5.7: Correlation coefficients between equivalence label and average reasoning score.

Table 5.8 provides detailed examples of Planning Module generated Low-level Intent evaluated for semantic correctness and reasoning quality. Each row corresponds to a single Low-level Intent derived from a global intent, and is annotated with:

- the **Global Intent** provided as input,

- the corresponding **Local Intent** generated to fulfill the intent,
- a binary label indicating whether the Low-level Intent was **Correct**,
- the **Reasoning** justifying the Low-level Intent, and
- a **Score** from 1 to 5 based on reasoning quality.

Global Intent	Local Intent	LIC	Reasoning	Score
Turn ON the Repeat option for all the five alarms. Verify the recurring status of all alarms on UI.	Tap on the "Edit" button for the second alarm to modify its settings and enable the repeat option. After tapping, wait for 0.5s to allow the screen to update.	True	The current screen displays two alarms, with the first alarm having the repeat option enabled for specific days, while the second alarm does not have the repeat option enabled. The intent is to ensure that the repeat option is turned on for all five alarms. Since the second alarm does not have the repeat option enabled, the next step is to edit this alarm to enable the repeat option. The "Edit" button for the second alarm is visible and can be tapped to modify its settings.	1
Click 'Advanced Settings', verify options Legal information [System WebView licences, Third-party licences], Reset options [Reset network, Reset app preferences, Delete all data (factory reset)], Apps and notifications [Show all apps, Default apps, App permissions, Special app access], About [Build number, Android version].	Scroll down the list to reveal more options, including 'Advanced Settings'. Wait for 0.5s after the scroll action to allow the screen to update.	False	The intent is to access the 'Advanced Settings' screen and verify specific options. The previous steps included scrolling and tapping on 'Advanced Settings', but the current screen still shows the main settings menu. This indicates that the tap on 'Advanced Settings' did not successfully navigate to the intended screen. It is possible that the 'Advanced Settings' option was not visible or not tapped correctly, or there might be a misidentification of the 'Advanced Settings' option in the XML hierarchy.	1
Turn ON the Repeat option for all the five alarms. Verify the recurring status of all alarms on UI.	Tap on the right chevron icon to navigate to the next set of alarms. After tapping, wait for 0.5s to allow the screen to update.	True	The current screen displays three alarms, with the first alarm's repeat option not active, and the second and third alarms having the repeat option active only for Monday. The intent is to enable the repeat option for all five alarms, but only three alarms are visible on this screen. To proceed, we need to ensure that the repeat option is enabled for the visible alarms and then navigate to the next set of alarms to continue the process.	2
Ensures the spapi contains the correct data. Launches the app, ensure that we start at the main page, and wait for the main page to properly load. Asserts that the displayed weights are the same as the ones uploaded to spapi.	Tap on the back arrow icon at the top left corner to return to the previous page or main page where weight data might be displayed. After tapping, wait for 0.5 seconds to allow the page to load.	False	The current screen displays a menu with options related to calibration and data management, but it does not show the main page where the weight data is displayed. The intent is to verify that the displayed weights match those uploaded to the spapi. To achieve this, we need to navigate back to the main page where the weight data is visible. The back arrow icon at the top left corner of the screen is likely to take us back to the previous page, which might be the main page with the weight data.	2

Global Intent	Local Intent	LIC	Reasoning	Score
Click on 'Audio' option, then verify availability of settings: Volume Control, Soundstage, Equalizer, Balance/Fader, Loudness.	Scroll up the list in the 'Audio' settings menu to reveal the options that are currently not visible. After scrolling, wait for 0.5 seconds to allow the screen to update.	True	The intent is to verify the availability of specific audio settings, including Volume Control and Equalizer, which are not currently visible in the screenshot. The current screen shows the 'Audio' settings menu with options like Balance/Fader, Soundstage, and Loudness, but not all required options are visible. To fulfill the intent, we need to scroll up to check if the missing options are located above the currently visible items.	3
Turn ON the Repeat option for all the five alarms. Verify the recurring status of all alarms on UI.	Tap on the "Save" button to save the changes made to the first alarm. After tapping, wait for 0.5s to allow the screen to update and return to the list of alarms.	False	The current screen is for editing the settings of the first alarm. The intent is to enable the Repeat option for all five alarms. The screenshot shows that the repeat option is enabled for the first alarm, as indicated by the highlighted days of the week. However, the task is not complete because the repeat option needs to be enabled for the remaining four alarms. The next step should be to save the changes made to the first alarm and return to the list of alarms to select the next one for editing.	3
Click on 'Audio' and scroll to 'Speed Dependent Volume', verify if options Off, Low, Mid, High can be changed.	Scroll down the 'Audio' settings list to reveal more options. After scrolling, wait for 0.5 seconds to allow the screen to update.	True	The current screen displays the 'Audio' settings, but the 'Speed Dependent Volume' option is not visible. To fulfill the intent, we need to locate and access the 'Speed Dependent Volume' settings. Since the option is not visible, it is likely further down the list, requiring us to scroll through the list to find it.	4
Click on edit button of Alarm1 and once it opens just click save button. Navigate to second page of the app. Click on edit button of Alarm4 and once it opens just click save button. Click on Home button.	Tap on the "Edit" button for Alarm1. After tapping, wait for 0.5 seconds to allow the edit screen to load.	True	The intent is to edit Alarm1 and then navigate to the second page to edit Alarm4. Currently, the screen displays three alarms, with Alarm1 being the first one on the left. The immediate step is to tap the "Edit" button for Alarm1 to proceed with the intent.	5

Table 5.8: Reasoning Examples by Global Intent, Local Intent, Local Intent Correctness (LIC), Reasoning, and Reasoning Score

5.4 RQ3 Results: Prompting Strategies and Multi-Modal Effectiveness

Quantitative Results Across Interaction Modes

We report quantitative results from an ablation study comparing three interaction modes: XML-only (`XML`), Image-only (`IMG`), and XML & Image (`XML` $\oplus$ `IMG`) for a multi-modal large language model (MLLM) across multiple UI-focused tasks. Each mode was evaluated using consistent prompts and assessed using task-specific

metrics, covering structured classification, spatial localization, text prediction, and continuous regression.

5.4.1 click_xy: Spatial Localization Accuracy

The click_xy task requires the model to predict a screen coordinate corresponding to a target UI element. Figure 5.1 quantifies the number of predictions that fall within ground-truth bounding boxes. The IMG modality shows the lowest correct counts while both XML and XML $\oplus$ IMG modalities yield the similarly high counts.

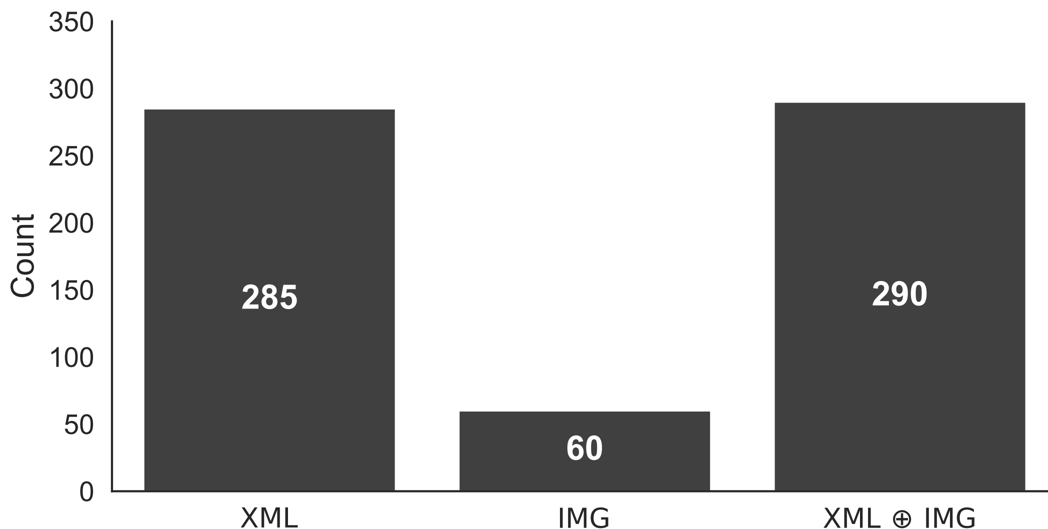


Figure 5.1: Number of click_xy predictions that fall inside the annotated bounding boxes.

In addition to inclusion counts, spatial precision is evaluated using continuous distance metrics. Figures 5.2 and 5.3 show histograms of Euclidean and Manhattan distances, respectively, stratified by whether the click fell inside or outside the target bounding box. In both metrics, XML and XML $\oplus$ IMG exhibit better distributions around low-error regions, particularly for in-box clicks, however the IMG has poor distribution across the board and most of the clicks are outside of the bounds.

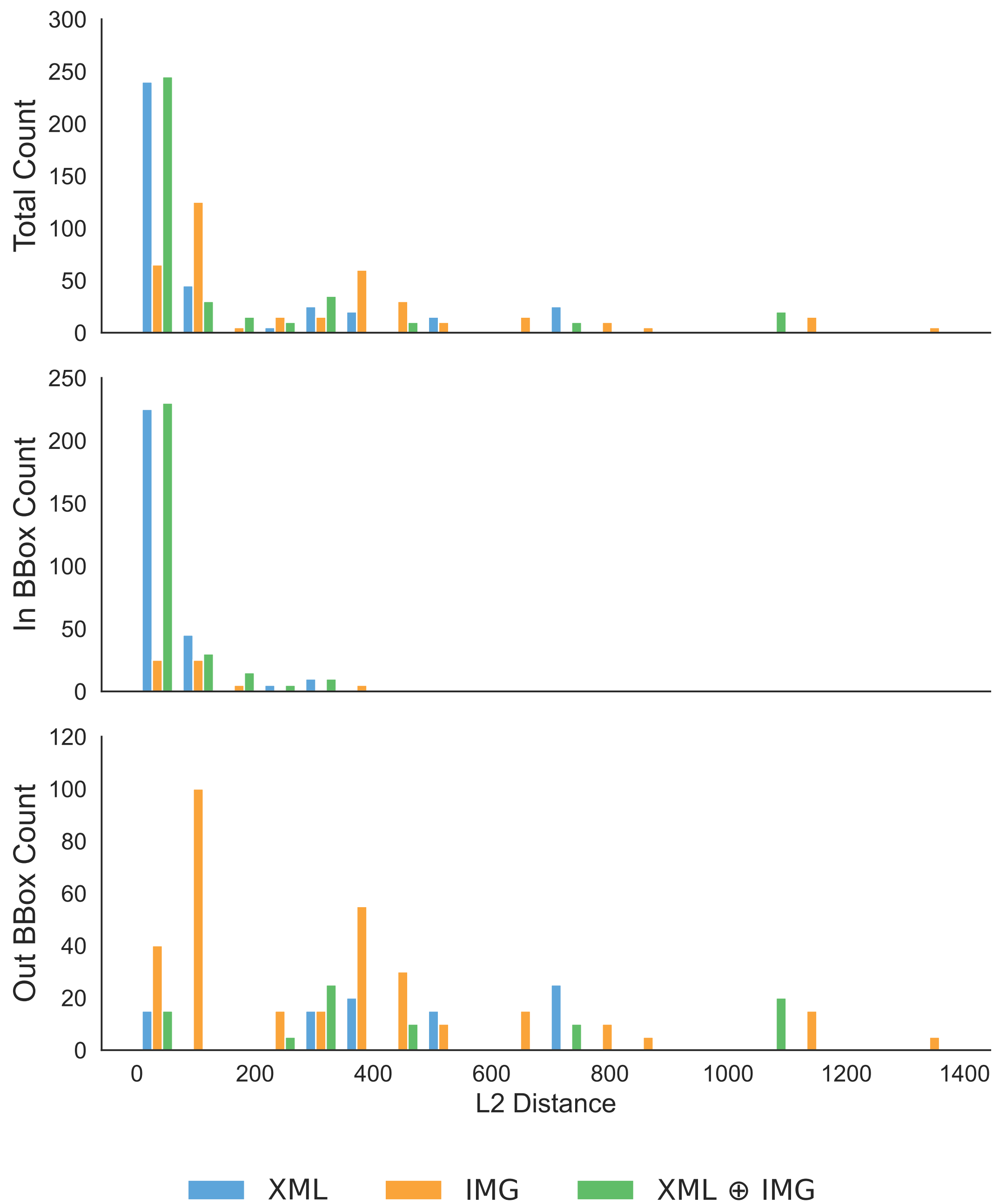


Figure 5.2: L2 distance distributions for `click_xy` predictions, split by bounding box inclusion.

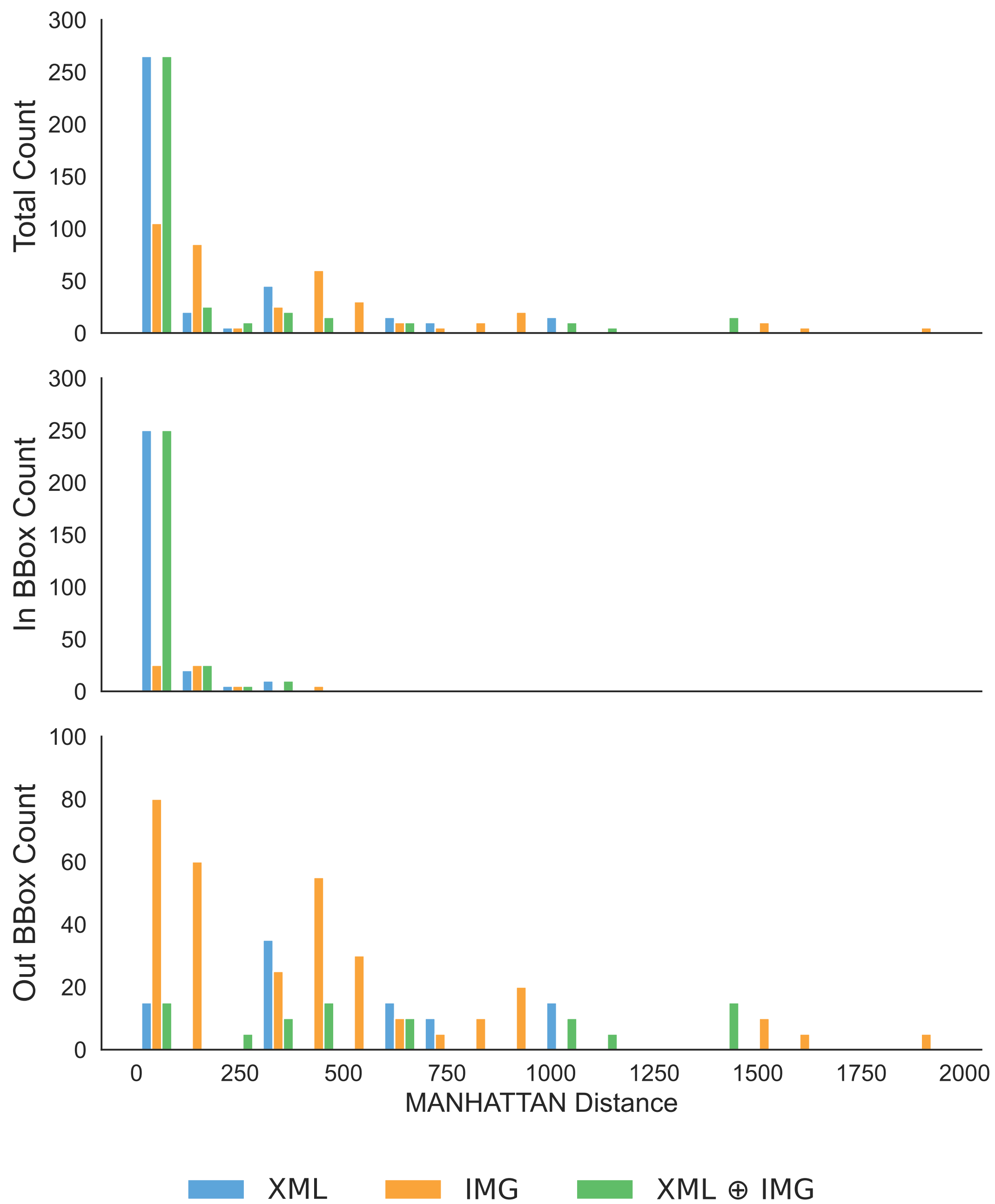


Figure 5.3: Manhattan distance distributions for `click_xy` predictions, stratified by bounding box inclusion.

Figure 5.4 illustrates the predicted click locations for the task *"set the fourth alarm to 2:13 AM"* using three different input modalities: `XML`, `IMG`, and `XML` ⊕ `IMG`. In this case, the interface initially displays only the first three alarms, and accessing the fourth requires interaction with a pagination control. Each modality's prediction reveals a distinct pattern of reasoning: the `XML` model appears to rely solely on the view hierarchy and confuses the third alarm's "Edit" button with the target; the `IMG` model predicts an approximate region for the navigation button without precise alignment; and the combined modality aligns its prediction with the interactive element expected to lead to the fourth alarm.

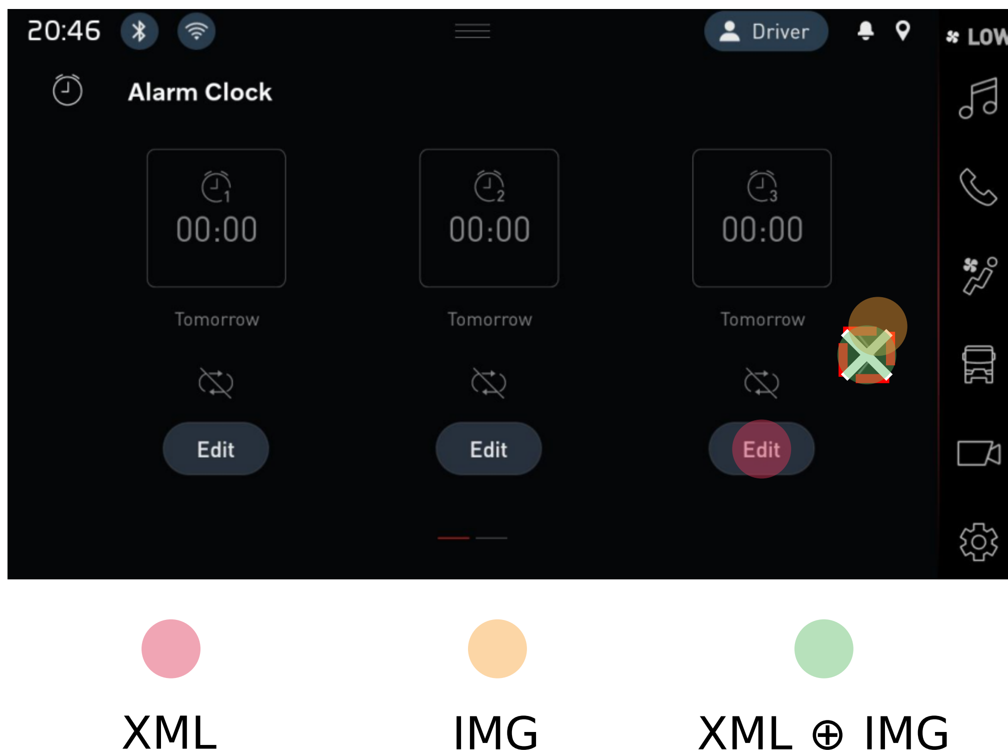


Figure 5.4: Predicted click points for the task "set the fourth alarm to 2:13 AM." Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth center of the target button, while the dashed red border indicates the bounding box of that button.

Figure 5.5 presents the predicted click locations for the task *"calibrate the third axle to 5 tons"* in the Load Indicator application. The screen displays a graphical truck representation with weight values aligned to each axle, and the user is expected to select the appropriate axle by clicking on the associated control. In this instance, both the `XML` and `XML` ⊕ `IMG` modalities predict clicks at the correct location, precisely at the center of the target button tied to the third axle. The `IMG` modality, by contrast, produces a click prediction that is misplaced toward the bottom right of the screen, far from any actionable UI element.

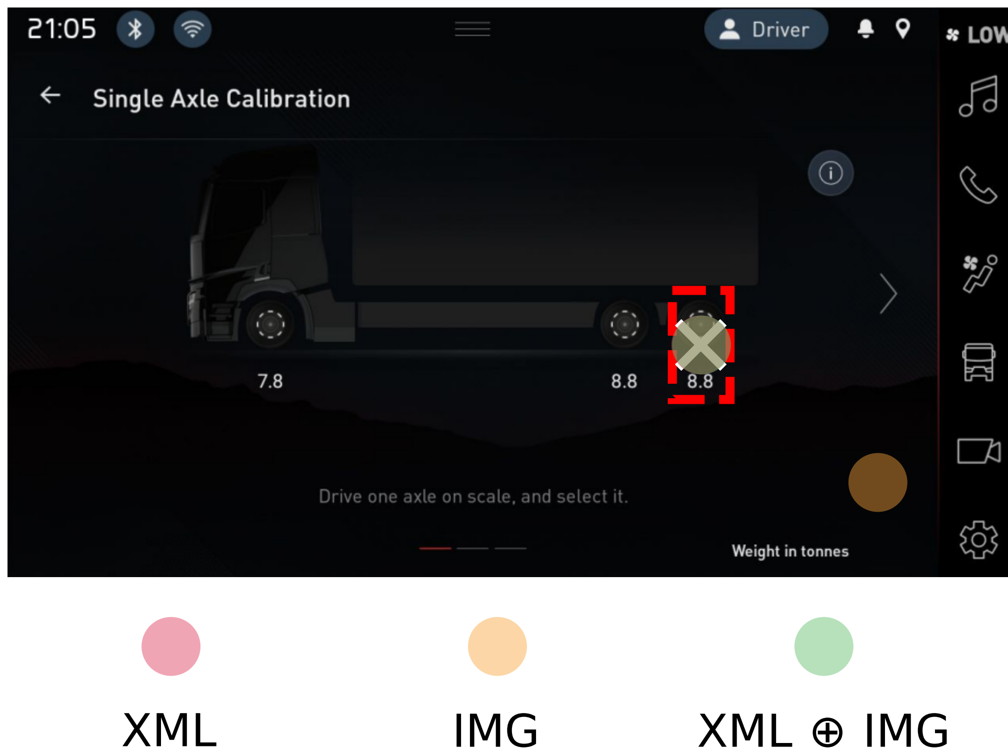


Figure 5.5: Predicted click points for the task "calibrate the third axle to 5 tons." Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth center of the target button, while the dashed red border indicates the bounding box of that button.

5.4.2 `click_id`: Semantic Targeting via ID Retrieval

In the `click_id` task, the model must identify and click the element associated with a given semantic ID. Only XML-enabled modes (`XML`, `XML` ⊕ `IMG`) are evaluated here. Figure 5.6 reports exact match accuracy, where both `XML` and `XML` ⊕ `IMG` modalities have similar high accuracies. The reason `IMG` modality is excluded is because the information for `element_id` is not present within images.

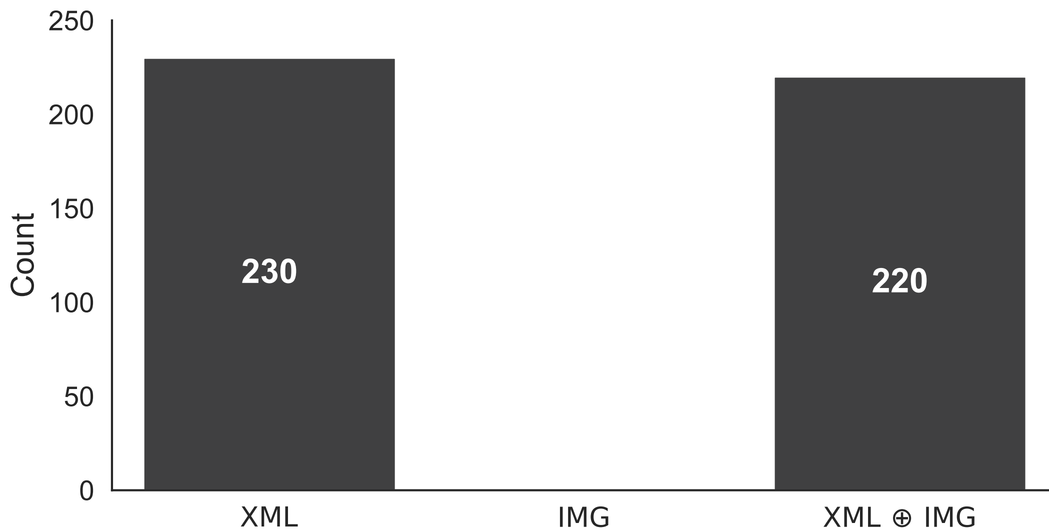


Figure 5.6: Exact match counts for the `click_id` task. Applicable to XML modes only.

Figure 5.7 shows edit distance distributions for the same task, providing a finer-grained view of output fidelity. Both `XML` and `XML` ⊕ `IMG` modalities have similar distributions.

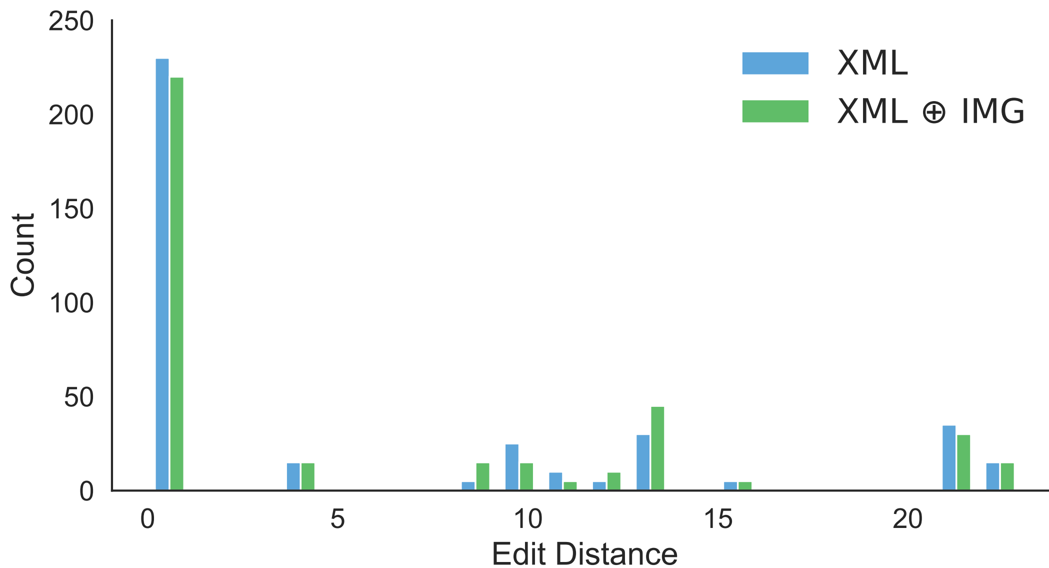


Figure 5.7: Edit distance histogram for `click_id` predictions in XML-capable modes.

5.4.3 `get_count`: Object Enumeration

The `get_count` task evaluates the model's ability to enumerate specific UI components. Figure 5.8 presents exact match counts. The `XML` modality performs worse, while both the `IMG` and `XML` ⊕ `IMG` modalities achieve similarly strong performance.

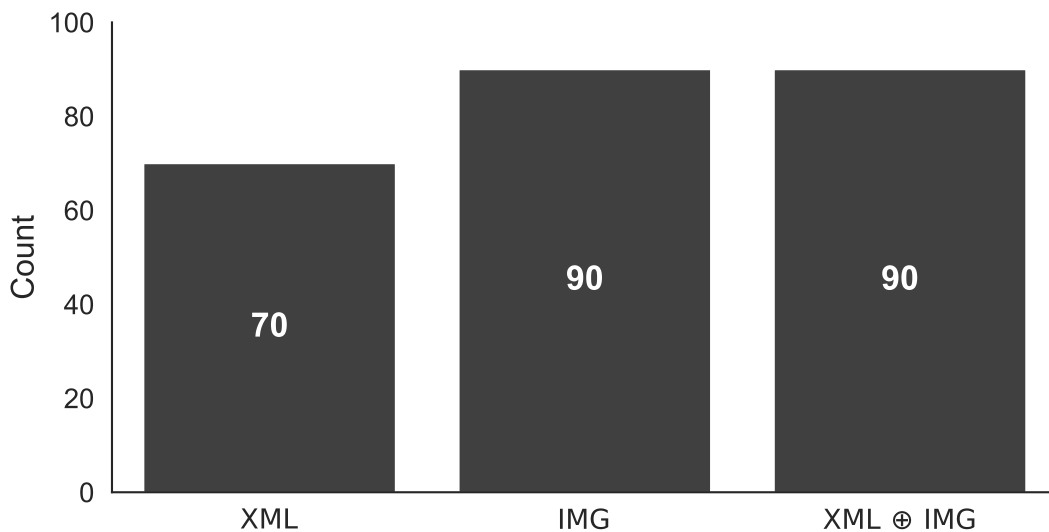


Figure 5.8: Exact match counts for `get_count`, where the model must return the number of queried items.

5.4.4 `instance`: UI Component Classification

The `instance` task evaluates the model's ability to correctly identify a specific occurrence of a duplicated UI element. As shown in Figure 5.9, all modalities performed similarly well, achieving high accuracy in instance selection across tasks.

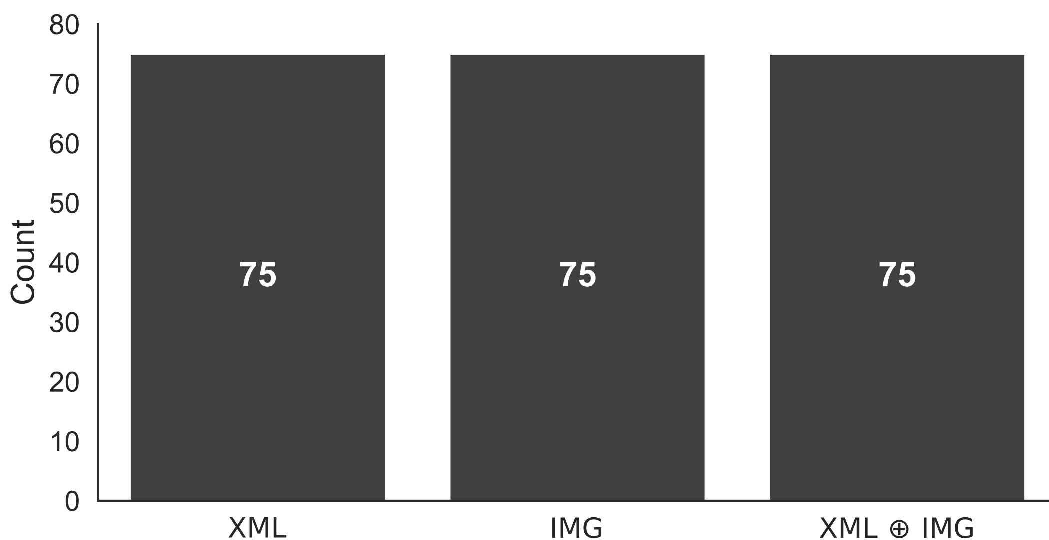


Figure 5.9: Exact match counts for `instance`, where the model identifies the correct occurrence of duplicated UI elements.

5.4.5 `get_text`: UI Text Retrieval

The `get_text` task requires the model to retrieve specific text from the UI. The `XML` and `IMG` modalities perform similarly. The `XML` ⊕ `IMG` modality performs best, with a small but measurable improvement.

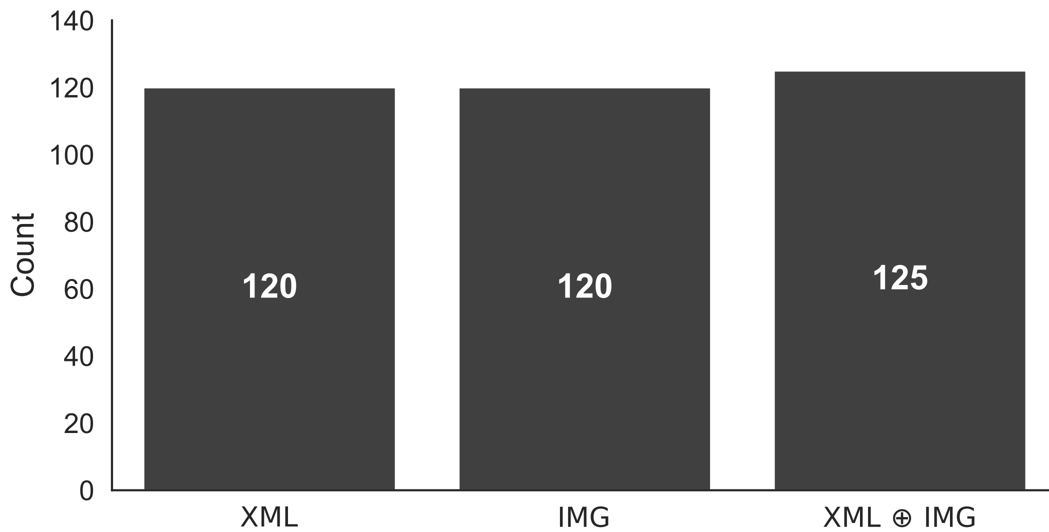


Figure 5.10: Exact match counts for the `get_text` task.

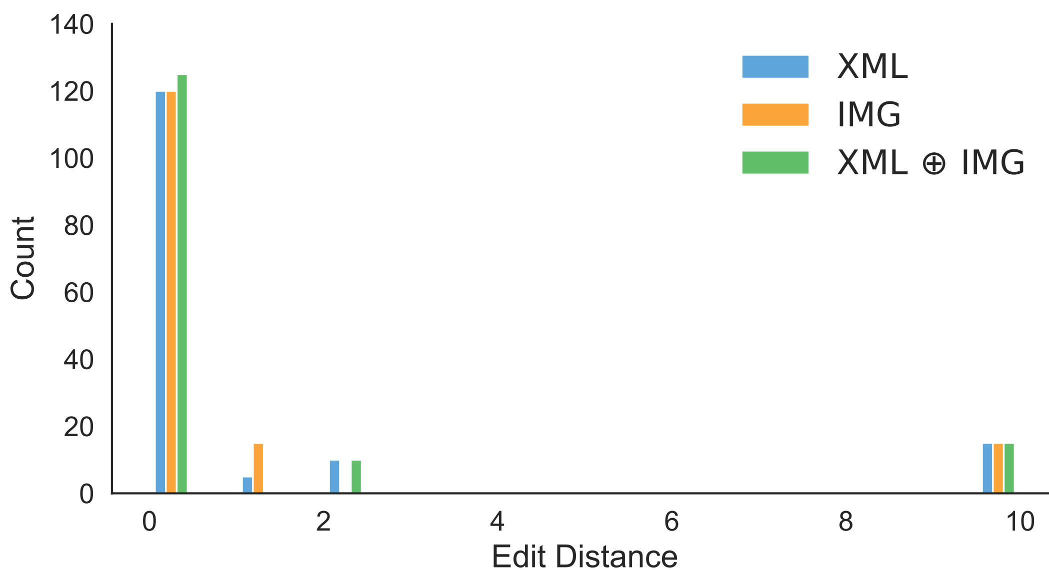


Figure 5.11: Edit distance distributions for the `get_text` task.

5.4.6 `seekbar`: Continuous Value Estimation

The `seekbar` task involves predicting XY coordinates corresponding to a specified seekbar position within the UI. Figures 5.12 and 5.13 show the L2 and Manhattan distance distributions between predicted and ground truth coordinates. The `XML` modality shows a tight distribution centered near zero, indicating many exact or near-exact predictions. The `IMG` modality displays a broad distribution with high error. The `XML` ⊕ `IMG` modality also shows a wide error distribution and performs worse than `XML`.

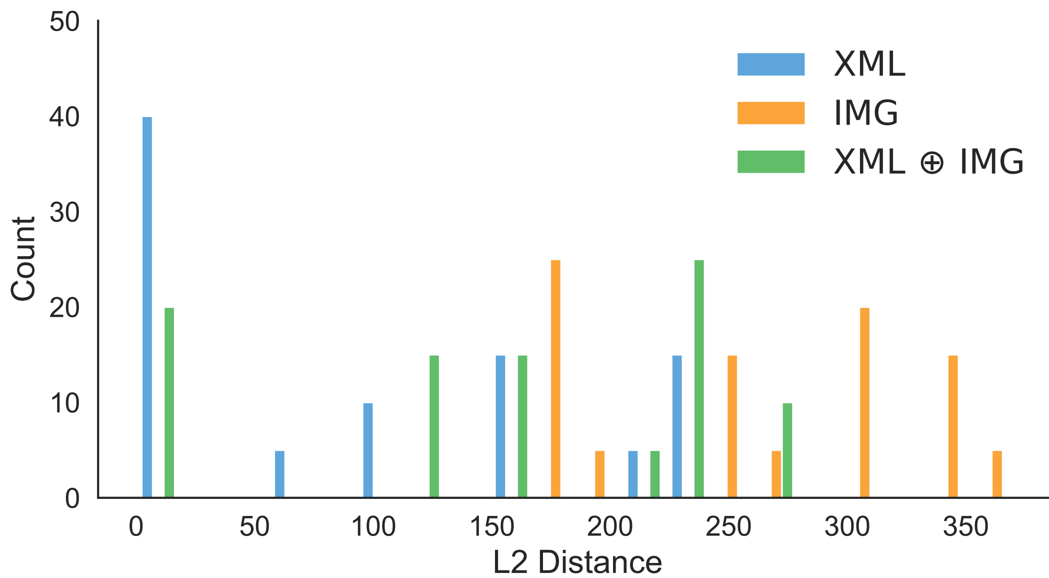


Figure 5.12: L2 distance histogram for `seekbar` predictions.

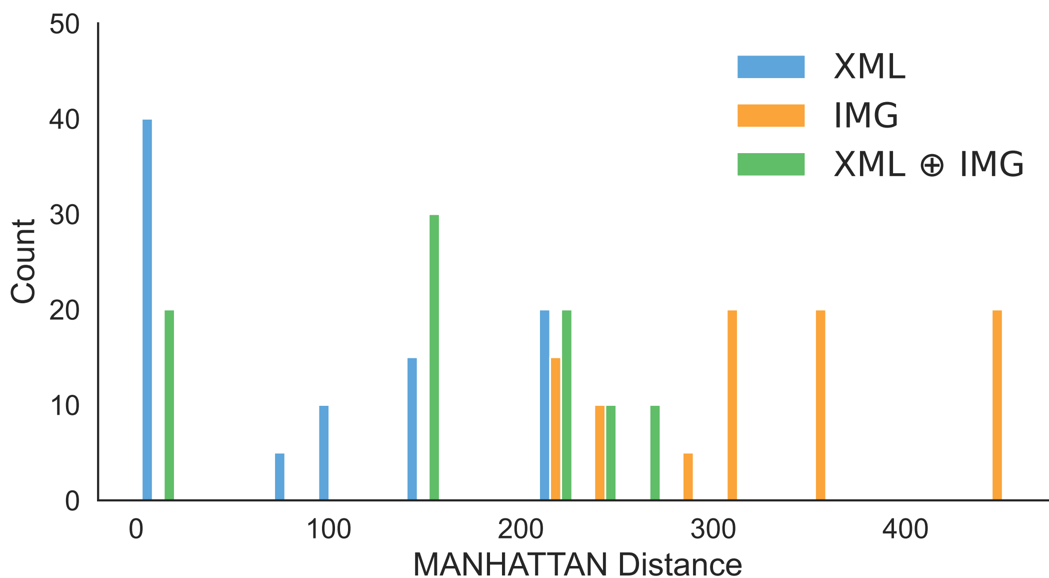


Figure 5.13: Manhattan distance histogram for `seekbar` predictions.

Figure 5.14 presents the predicted click locations for the task *"set the 8kHz band to -9dB"* in the Equalizer interface. The screen contains eight vertical seek bars, each corresponding to a different frequency band. The target in this case is the 8kHz band, and the intended click location lies near the lower end of its range, as indicated by the X mark. The `XML` and `XML` ⊕ `IMG` modalities both predict clicks at the 8kHz band but align with the upper part of the `seekbar`, closer to +9dB and overlooking the required negative value. The `IMG` modality prediction is farther off, registering a click near the top of the adjacent 16kHz band instead.

Figure 5.14: Predicted click points for the task "set the 8kHz band to -9dB." Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth target position on the 8kHz `seekbar`.

Figure 5.15 shows click predictions for the task *"set media volume to 60%"* in the Volume Control interface. Multiple horizontal seek bars are visible, each tied to a different audio category. The media volume `seekbar`, situated near the top of the screen, includes a circular handle that marks the target location, approximated by the X. Both the `XML` and `XML` ⊕ `IMG` modalities predict click points close to this handle which aligns with the intended 60% position. The `IMG` modality prediction diverges significantly, producing a click location in the upper-left corner of the screen, disconnected from any `seekbar` or handle.

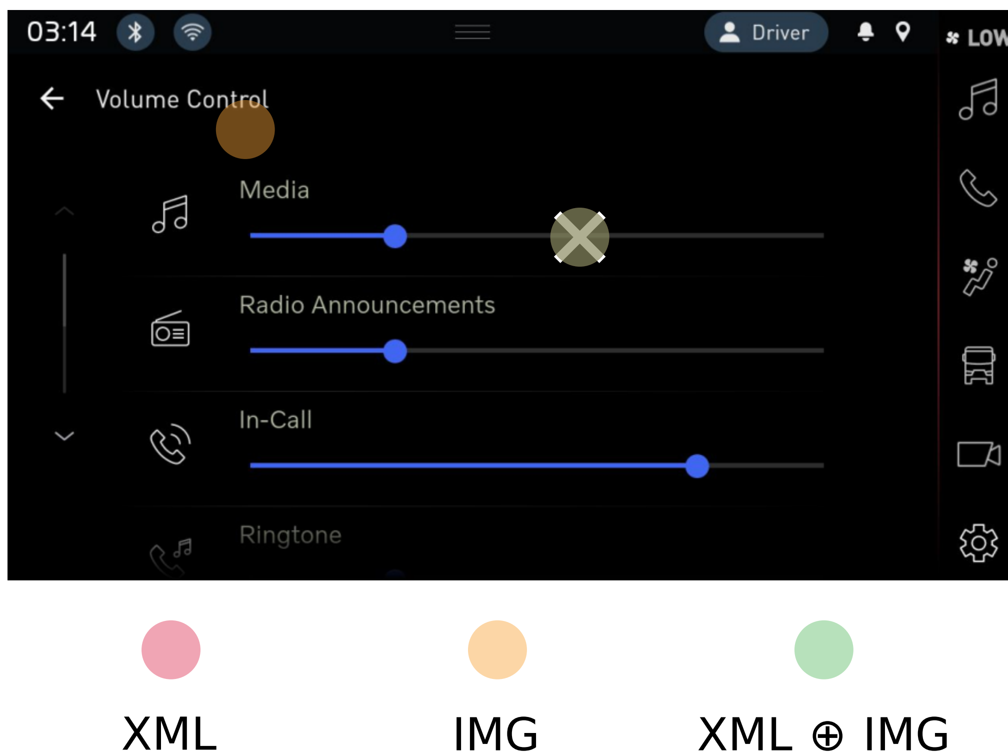


Figure 5.15: Predicted click points for the task “set media volume to 60%.” Pink corresponds to `XML`, orange to `IMG`, and green to `XML` ⊕ `IMG` modality. The X symbol marks the ground truth target location on the media `seekbar`.

6

Discussion

6.1 Discussion of RQ1: Code-Level Comparison Insights

To understand the practical implications of the RQ1 results, we analyzed 27 matched test pairs across three Volvo apps — Alarm Clock, System Settings, and Load Indicator. Each pair includes a manually written test (M) and an LLM-generated test (A), evaluated on five structured metrics. The key observations and implications are as follows:

6.1.1 Correctness

Manual tests consistently achieve 100% correctness, serving as the ground truth. In contrast, LLM-generated test correctness varies widely from 15% to 100%. Importantly, several automated test cases in Load Indicator have low correctness such as L27 has only 15% line-level correctness, which indicates it contains a large amount of incorrect lines and produce incorrect actions on the app. Correctness tends to drop significantly in the Load Indicator app, with most Automated tests scoring below 60%. This suggests that LLM performance degrades in complex scenarios, where coming up with correct local intents is harder.

6.1.2 Unnecessary Steps

Automated tests overall exhibit more unnecessary steps compared to manual tests. This trend is especially pronounced in the Load Indicator app, where the number of redundant or exploratory actions is significantly higher than in Alarm Clock or System Settings. The primary reason is the increased complexity and density of UI elements in Load Indicator, which makes it more challenging for the framework to semantically interpret the app structure and avoid redundant interactions during UI exploration.

An important nuance in this comparison is that manual tests often use abstraction through external packages (e.g., `assert app.select_edit_timer(0)`), which effectively eliminates unnecessary steps at the script level but introduces an additional abstraction layer that can increase maintenance overhead. In contrast, automated

tests—especially in the simpler Alarm Clock and System Settings cases—achieve comparable functionality without introducing new abstraction layers. Instead, they operate directly using elementary operations that are provided by UIAutomator, which can enhance maintainability and offer a more developer-friendly, transparent structure.

6.1.3 Flakiness

As expected, all manual tests are non-flaky, as they serve as the ground truth and have been carefully curated by developers. However, several automated tests are marked as flaky, particularly in the System Settings app. This flakiness primarily arises from the scrolling actions required to explore the settings menus. Even when performing the same scroll commands, slight differences in system delays, touch accuracy, or gesture precision can lead to divergent app states, causing inconsistent outcomes across repeated runs.

Additionally, the seekbar manipulation actions in System Settings further contribute to flakiness, as accurately setting the seekbar position can be sensitive to pixel-level precision. Similar issues appear in the Alarm Clock and Load Indicator apps, where two automated tests in each were marked flaky. In these cases, the root cause was often related to actions such as number picker interactions, which depend on accurate scrolling behavior.

Overall, these findings highlight a current weakness in the framework's handling of scroll-based and precision-sensitive actions. Addressing this limitation will require enhancing the system to select more reliable alternative actions or introducing corrective mechanisms to improve action accuracy and consistency.

6.1.4 Robustness

The manual tests, serving as ground truth, consistently achieve high robustness scores. This is because they include assertions and validations on nearly every critical line of code. However, it is important to note that many of these assertions are embedded within external abstraction layers (e.g., `assert app.save_timer()`), which encapsulate multiple low-level checks into a single high-level call.

In contrast, the automated tests exhibit relatively low robustness scores. This limitation stems from the current weaknesses of our system: at present, the system lacks the ability to check backend data or interact with API endpoints to validate deeper system states. As a result, automated tests are restricted to UI-level assertions, which limits the scope and reliability of validations. This leads to a noticeable absence of robust validation and comprehensive assertions in most automated test cases, reducing their defensive capacity compared to the manual baselines.

6.1.5 Readability

The manual tests generally demonstrate good readability as ground truth. However, their readability is somewhat reduced by the use of abstraction layers (e.g., `app.launch_by_navigation()`). While helper methods encapsulate complex logic, they can obscure the underlying actions, making it difficult for developers to understand the full behavior without inspecting the external package or diving into the function definitions.

In contrast, the automated tests provide full visibility into the test structure, as all actions are primitive UIAutomator calls explicitly listed within the test cases. While this flat and transparent listing enhances immediate understandability, it lacks meaningful abstraction, helper methods, logical structuring, and documentation — key factors in our readability criteria. As a result, the overall readability score of the automated tests is reduced. Moving forward, we plan to introduce enhanced prompting strategies to encourage the language model to generate modularized, well-commented test cases to improve readability.

6.2 Discussion of RQ2: Semantic Understanding and Planning Module Interpretation

6.2.1 Reasoning Rubric

As shown in Table 4.3, we evaluate reasoning quality using a 5-point rubric. The scoring is justified through an inductive comparison against our ground truth—manual test cases—based on the following equivalence criteria:

1. The automated test leads to the same resulting app state.
2. The automated test reproduces the same functionality as the manual test, step by step.

Given a Global intent as input, we inspect each local intent generated by the Planning Module within the automated test. Each reasoning step is scored individually according to the rubric, and the overall reasoning quality is determined by averaging these scores.

As shown in Table 5.5 and Table 5.6, a high average reasoning score across the full test case aligns with equivalence between the automated and manual tests in both outcome and functionality. Conversely, a low average score indicates that the automated test fails to faithfully replicate the manual test behavior. These case-level comparisons provide the basis for our justification of reasoning quality evaluation.

Table 5.4 presents the average reasoning scores and binary equivalence outcomes for all 35 automated tests paired with their corresponding manual tests. To validate the effectiveness of our Reasoning Rubric, we analyzed the relationship between these two variables—average reasoning score (continuous) and equivalence (binary).

We computed two correlation measures to quantify the strength of this association: Spearman[1] Rank correlation and Point-Biserial[2] correlation. As shown in Table 5.7, the Spearman Rank coefficient is **0.820** with a p-value of 5.706×10^{-12}, and the Point-Biserial coefficient is **0.842** with a p-value of 4.165×10^{-13}. Both coefficients indicate a strong positive correlation between reasoning quality and test equivalence. The relatively high coefficient values suggest that higher reasoning scores are strongly predictive of successful equivalence. The low p-values[3] for both tests confirm that these relationships are statistically significant. These findings support our claim that the Reasoning Rubric effectively captures meaningful aspects of agent behavior and closely aligns with the system's ability to produce functionally correct outcomes.

6.2.2 Correctness and Semantic Alignment

To further evaluate how accurately the framework translates developer intents into meaningful and correct actions, we conducted a quantitative analysis of local intents across three representative Volvo applications: Alarm Clock, System Settings, and Load Indicator. Each local intent was annotated for correctness and the associated reasoning score was computed using our rubric.

As shown in Table 5.3, the framework generated a total of 42 local intents, of which 30 were deemed correct. This corresponds to an overall correctness rate of **71.4%**, indicating that in the majority of cases, the Planning Module produced contextually valid and meaningful local intents.

We conduct a qualitative analysis on the reasoning that Planning Module has made to generate the local intents. More importantly, the reasoning scores provide additional insight into the quality of these local intents. Correct steps had a high average reasoning score of **4.33**, while incorrect steps had a significantly lower average score of **2.00**. This consistent gap demonstrates that the Reasoning Rubric is not only correlated with equivalence at the test level, but also sensitive enough to differentiate correct vs. incorrect actions at the local intent level.

These findings reinforce the conclusion that high reasoning quality strongly predicts successful local intent generation. Misunderstandings or reasoning breakdowns often coincide with functional failures, validating our semantic alignment methodology for evaluating intent understanding.

[1]Spearman rank correlation measures monotonic relationships between ordinal or continuous variables; suitable for comparing subjective scores with graded outputs.

[2]Point-biserial correlation measures the association between a continuous variable and a binary variable; appropriate for linking subjective scores to correctness labels.

[3]A p-value indicates the probability of observing the data (or more extreme) assuming the null hypothesis is true; lower values suggest stronger evidence against the null.

6.2.3 App-Specific Reasoning Quality

As shown in Table 5.3, the reasoning quality of local intents generated by Planning Module varied across the three evaluated apps, reflecting differences in UI complexity and structural predictability. Alarm Clock, with relatively static and repetitive layouts, achieved the highest correctness rate (87.5%) and a strong average reasoning score (μ= 4.00). In contrast, Load Indicator, which features dynamic lists and pop-up interactions, showed both a lower correctness rate (53.3%) and the lowest average reasoning score (μ= 3.13).

Interestingly, the System Settings app, which includes nested toggles and deep menu structures, exhibited higher reasoning scores for correct steps (μ= 4.88) than Alarm Clock or Load Indicator. This suggests that when the Planning Module navigated the structure successfully, its understanding was clearly conveyed.

These results show that the framework is more reliable in apps with predictable UI layouts and limited variability, such as System Settings, where there is no recursion or overlapping states — each action leads uniquely to the same destination, and the structure (scroll, click, etc.) is quite predictable. In contrast, dynamic or deeply nested UIs introduce challenges, as seen in the other two apps, where recursion and overlapping states allow multiple paths to share the same states, making the framework more prone to getting lost even if the state exists in memory. This variability affects both the Planning Module's decision-making accuracy and the clarity of its reasoning.

6.2.4 Common Error Patterns and Limitations

Analysis of incorrect steps revealed several recurring issues that contributed to test failures. One common pattern involved incorrect UI element targeting, such as confusing sibling list items or selecting the wrong instance(e.g. incorrect selection of an Alarm Clock instance). These errors typically coincided with reasoning scores of 2 or lower.

Another failure mode was reasoning repetition(e.g. attempting to rediscover or reselect UI elements that had already been accessed), especially prevalent in apps with scrollable or layered interfaces. These issues suggest limitations in incomplete memory or state tracking across local intents.

While our Reasoning Rubric effectively captures these reasoning flaws, the underlying challenge remains: enhancing the Planning module's awareness of app state transitions and screen-specific dynamics.

6.3 Discussion of RQ3: Prompting Strategies and Multi-Modal Effectiveness

6.3.1 `click_xy`: Spatial Localization Accuracy

Task Overview

The `click_xy` task evaluates the model's ability to predict the precise screen coordinates of a target UI element based on a natural language prompt. This requires the model to interpret the prompt, identify the correct UI component from the input representation(s), and output the center point of that component's bounding box in `(x, y)` format. Accurate completion of this task depends on both semantic understanding and spatial reasoning, and serves as a measure of how well the model can localize actionable elements within a mobile interface.

Performance Comparison Across Modalities

The results in Figure 5.1 reveal significant performance disparities across the three interaction modalities for the `click_xy` task. The XML-only (`XML`) modality achieves approximately 285 correct clicks out of 350, while the image-only (`IMG`) modality performs considerably worse, with only around 60 successful predictions. The combined modality (`XML` $\oplus$ `IMG`) slightly outperforms the XML-only modality, reaching nearly 290 correct predictions.

These results suggest that XML-based information provides highly reliable spatial localization cues. This is attributable to the structured nature of the XML data, which explicitly encodes bounding boxes for interactive elements. Given sufficient prompt guidance, the model can extract these coordinates, compute a center point, and return accurate `(x, y)` values. In contrast, the image-only modality lacks access to such structured spatial information. The model must infer clickable regions purely from visual features, which is inherently ambiguous and error-prone, particularly without specialized training on coordinate regression tasks in UI contexts.

The marginal improvement observed in the `XML` $\oplus$ `IMG` condition over the XML-only modality suggests that the inclusion of images can be beneficial when used in conjunction with XML. Although the images alone are insufficient for accurate localization, they may help disambiguate the target element when multiple XML nodes share similar attributes. However, this interpretation is speculative; a controlled analysis isolating such cases was not conducted and thus remains an open question.

Distance-Based Error Analysis

Figures 5.2 and 5.3 further support these conclusions by illustrating the distribution of localization errors in L2 and Manhattan distance metrics. The XML-only modality yields a sharply peaked distribution near zero, indicating that most predictions fall very close to the true center of the target bounding box. The shape is

reminiscent of a right-skewed unimodal distribution, concentrated in the low-error region.

By contrast, the image-only modality displays a wider and flatter distribution. Although some predictions cluster near the correct location, many are significantly off-target. This implies that the model, despite lacking explicit spatial representations, still attempts to localize the correct region heuristically – yet it often fails to refine its prediction to precise coordinates.

The `XML` $\oplus$ `IMG` modality demonstrates a modest leftward shift in the error distribution relative to the XML-only case. This shift suggests that when the model integrates visual context alongside structured XML, it can further reduce spatial error in some cases. This may occur when visual features reinforce or clarify the XML-based decision, leading to more confident and centered predictions. However, the gain is small, and the XML information remains the dominant factor in spatial localization performance.

Click Prediction Across Modalities

The results in Figure 5.4 reveal important modality-specific behaviors in response to the task of setting the fourth alarm. The `XML` modality incorrectly selects the "Edit" button associated with the third alarm. This behavior is expected given the limitations of the XML hierarchy, which lacks any semantic markers distinguishing the individual alarms beyond replicated structural patterns. The numerical labels (1, 2, 3) that visually denote the alarm indices in the screenshot are not present in the hierarchy. As such, the model encounters three visually identical alarm elements with incrementing instance indices but no semantic grounding. Without an explicit reasoning chain or iterative step-tracking—both of which were excluded in this one-shot ablation setup—it defaults to the last visible instance.

In contrast, the `IMG` modality appears to recognize from visual cues that the fourth alarm is not available on the current screen. It identifies an arrow-like UI component positioned on the right edge of the interface, likely indicating a pagination control, and attempts to click near it. While the click is not precisely aligned with the control, the spatial reasoning exhibited here is notable given the lack of structural information.

The combined `XML` $\oplus$ `IMG` modality performs the task most effectively. The visual input provides the contextual insight that a navigation step is required, while the structural XML allows the model to locate the actual interactive component responsible for this transition. As a result, the predicted click lands directly at the center of the pagination button, coinciding with the ground truth. This indicates that the multi-modal combination facilitates both high-level intent recognition and low-level target resolution, which enables a correct and contextually appropriate action.

Limitations and Future Directions

While the results establish the primacy of XML in spatial localization, they also raise questions about how to better leverage visual information. One unexplored avenue is image annotation. Augmenting images with visual cues such as bounding boxes or highlights may help the model correlate visual regions with semantic targets, potentially improving image-only and multi-modal performance. Further, training or fine-tuning with tasks explicitly targeting spatial coordinate extraction may enhance the model's sensitivity to such requirements.

Overall, the findings from the `click_xy` task affirm that current MLLM systems benefit substantially from structured XML input when tasked with coordinate prediction. Visual input, while insufficient on its own, can provide marginal gains when used in combination.

6.3.2 `click_id`: Semantic Targeting via ID Retrieval

Task Overview

The `click_id` task assesses the model's ability to identify and return the exact resource ID of a UI element based on a semantic prompt. The prompt typically describes the function, label, or visual role of the target element. To succeed, the model must parse the prompt, locate the matching element in the input representation, and extract its unique identifier. This task tests the model's capacity for semantic matching and symbolic precision, particularly in structured input formats like XML where element IDs are explicitly defined.

Modality-Based ID Retrieval

The `click_id` task evaluates the model's ability to identify and retrieve the exact element ID corresponding to a semantic prompt. As noted in the results (Figure 5.6), the XML-only (`XML`) modality achieves high accuracy, with approximately 230 out of 250 samples returning the correct ID. This is expected, as the XML provides direct access to structured metadata, including element IDs.

By design, this task excludes the image-only (`IMG`) modality. Images inherently lack embedded element identifiers, making them unsuitable for this task. The absence of an ID field in visual input renders any model operating solely on image data incapable of meaningful participation in this evaluation.

Impact of multi-modal Input

When combining XML with images (`XML` $\oplus$ `IMG`), performance remains high, but a small drop is observed – with accuracy falling slightly to around 220 out of 250. This minor degradation may indicate that the addition of image information introduces ambiguity into the model's reasoning process. Specifically, it is possible that the model attempts to correlate visual features with the structured XML representation. Since the image does not contain element IDs, but may include visible labels or text

resembling IDs, such correlation might lead to incorrect associations and distract the model from selecting the correct XML element. This hypothesis remains speculative, as no targeted ablation was performed to confirm this behavior.

Nonetheless, it is notable that the performance remains robust even with multi-modal input. The small decline does not compromise the overall efficacy of the system. This suggests that while there may be edge cases where the visual context introduces confusion, the model can generally prioritize the structured XML information effectively during ID resolution.

Edit Distance Analysis

To gain finer insight into the nature of incorrect predictions, we examine the edit distance distributions in Figure 5.7. These reveal that incorrect IDs typically diverge significantly from the ground truth. In both `XML` and `XML` $\oplus$ `IMG` modes, the majority of incorrect outputs have edit distances well above 10, with very few near misses (edit distance less than 4). This indicates that errors are not due to minor string variations or case mismatches; rather, they represent entirely incorrect selections – often pointing to unrelated elements.

This observation highlights an important caveat in the use of edit distance as a supplementary metric for ID prediction tasks. In Android development contexts, element IDs must match exactly, including case sensitivity. Therefore, any deviation from the correct string constitutes a failure, regardless of how minor it may seem in edit distance terms. The distributions, while informative, should be interpreted as descriptive rather than diagnostic in this setting.

Interpretation and Recommendations

Overall, the findings affirm the effectiveness of the XML modality for semantic ID retrieval tasks. Adding visual information yields no substantial improvement and may slightly degrade performance due to multi-modal interference. This warrants caution when designing multi-modal prompts for tasks dependent on exact symbolic matching. Further research could explore whether integrating visual input through more constrained or filtered mechanisms (e.g., masking irrelevant regions) might preserve or enhance precision without introducing confusion.

The current results suggest that XML remains the most reliable source for structured attribute extraction, and visual data should be integrated with care in tasks requiring precise identifier resolution.

6.3.3 `get_count`: Object Enumeration

Task Overview

The `get_count` task requires the model to enumerate elements of a specific type present on the screen. These include prompts such as “How many wheels are visible?”, “How many active alarms are currently shown?”, or “How many axles does the

truck have?". Solving this task requires the model to understand spatial repetitions and context-specific definitions of visual objects, often necessitating aggregation across either structural or visual cues.

Performance Trends Across Modalities

According to the results shown in Figure 5.8, the image-only (`IMG`) modality performs best, with around 90 correct predictions out of 100. The XML-only (`XML`) modality performs significantly worse, at around 70 out of 100. Notably, the combination of XML and image (`XML` $\oplus$ `IMG`) matches the performance of the image-only case, with no observable improvement or degradation.

This pattern suggests that enumeration in user interfaces is primarily a visually grounded task. The XML representation often lacks explicit counts or does not describe repeating visual patterns in a way that lends itself to direct aggregation. In some cases, the XML may not contain any useful structure for the objects being queried, or the relevant information may be encoded in a highly indirect form that is not straightforward to extract. Such cases can be challenging even for human readers, let alone for LLMs operating over structured data.

Interpretation and Modal Implications

The strong performance of the image-only modality indicates that vision-language models are well-suited to handle enumeration tasks when the target elements are visually distinct or spatially grouped. This includes cases where similar shapes, colors, or layout structures naturally imply a count. In contrast, structural representations like XML, while rich in node-level detail, may obscure the broader visual patterns necessary for such reasoning.

It is noteworthy that the addition of XML in the `XML` $\oplus$ `IMG` configuration does not impair the model's performance. This implies that the model can rely on visual input when it is sufficient, without being misled by uninformative or misleading XML structures. The ability to retain image-based performance while including XML data suggests a degree of robustness in the model's multi-modal integration strategy. This could be advantageous in settings where XML data must be included for pipeline uniformity or other task components.

This also offers a subtle insight: while XML contributes significantly to tasks requiring exact structural identifiers or coordinate-based localization, it appears to play a minimal role in visual aggregation tasks like counting. The model's ability to ignore extraneous or redundant input without degrading accuracy is a positive signal for broader multi-modal deployment, particularly in scenarios involving heterogeneous or variable-quality XML data.

6.3.4 `instance`: UI Component Classification

Task Overview

The `instance` task evaluates the model's ability to identify a specific occurrence of a duplicated UI component. This is particularly relevant in dynamic user interfaces where multiple elements of the same type and attributes exist—such as repeated list items or buttons—requiring disambiguation by instance index. In tools like UI Automator, selecting the correct instance is essential for reliable automation, as only the indexed form of the element can be targeted in the view hierarchy.

Performance Trends and Observations

In this experiment, 80 test samples were evaluated under three modality configurations: XML-only (`XML`), image-only (`IMG`), and combined XML and image (`XML` $\oplus$ `IMG`). All three modalities performed identically, achieving near-perfect accuracy in selecting the correct instance of the target element. This indicates that the task was solvable using any of the input types available, without any one modality providing a distinct advantage.

Interpretation and Modality Relevance

The uniform performance across all three conditions suggests that the information necessary to disambiguate instances was encoded consistently across modalities. In the case of XML, this likely comes from the explicit structural position of elements within the hierarchy, such as the index within a parent node. In images, this may be inferred visually through spatial order, layout alignment, or repeated visual patterns.

Importantly, the inclusion of both modalities in the `XML` $\oplus$ `IMG` modality did not introduce any interference or confusion, nor did it yield improvement. This result reinforces the idea that when both modalities redundantly encode the needed information, the model is able to select a consistent interpretation path regardless of how the information is presented.

From a practical standpoint, this suggests that for instance-level targeting tasks, either modality is sufficient, and multi-modal inputs can be used without risk of performance degradation. This flexibility may be useful in deployment scenarios where either visual or structural data is intermittently unavailable or varies in quality.

6.3.5 `get_text`: UI Text Retrieval

Task Overview

The `get_text` task evaluates the model's ability to extract a specific string of text from the user interface. The prompt specifies what text to retrieve, and the model must identify and return the exact string as it appears on screen. This requires both locating the appropriate element and correctly extracting its textual content.

Performance Across Modalities

As shown in Figure 5.10, both the XML-only (`XML`) and image-only (`IMG`) modalities performed similarly well, with approximately 120 correct responses out of 140. This suggests that either the structural representation or the visual appearance alone is often sufficient to identify and extract the requested text.

When both modalities are used together (`XML` $\oplus$ `IMG`), performance increases to around 130 out of 140, indicating a measurable benefit from multi-modal input. This improvement suggests that some instances are better handled when both XML and visual representations are available, likely due to complementary strengths in how the information is encoded across the two formats.

Interpretation and Modal Synergy

The observed improvement in the `XML` $\oplus$ `IMG` setting may reflect a resolving effect when the model is given access to both structural and visual inputs. In some cases, the XML may omit relevant text or present it in a non-obvious location within the hierarchy, while the image offers a direct visual representation. In other cases, the visual representation may be ambiguous—for example, due to overlapping UI elements or unconventional layout—where the XML provides the needed structure to disambiguate.

The model appears to benefit from this redundancy, aligning cues across modalities to more reliably identify the correct answer. However, this interpretation remains speculative, as no ablation analysis was conducted to isolate which specific samples were corrected by the addition of visual or structural context. Whether this performance gain is due to true complementarity or merely reinforcement of information remains an open question.

The distribution of edit distances for the `get_text` task as can be seen in 5.11 shows that when predictions are correct, they are usually *exact*, with an edit distance of zero—consistent with the exact match results. When the model is wrong, the deviations are often minor: many fall within an edit distance of 1 or 2, which may hint at superficial errors such as mismatched capitalization or minor omissions. We have not explicitly analyzed whether these small distances correspond to case sensitivity, and it remains unclear whether the Levenshtein metric we use treats case as significant. Notably, there are a few instances with much higher distances (e.g., 10+), which likely correspond to cases where the model predicted an entire phrase or string instead of a specific unit—something we've seen in examples where disambiguation was semantically demanding. Interestingly, `XML` $\oplus$ `IMG` (`XML` $\oplus$ `IMG`) avoids any edit distance of exactly 1, suggesting that combining modalities may resolve certain borderline failures. Conversely, at distance 2, `IMG` alone was correct while `XML` and `XML` $\oplus$ `IMG` both made errors, indicating that added structure is not universally beneficial. These patterns suggest nuanced modality interactions that warrant closer error-level analysis.

Implications

This result provides a useful demonstration that combining structured and visual representations can improve retrieval of exact text from UIs. Where accuracy is critical, using both modalities appears to offer an advantage with no apparent downside. This may be particularly valuable in testing contexts where string equality is strict and tolerances are low.

6.3.6 `seekbar`: Continuous Value Estimation

Task Overview

The `seekbar` task evaluates the model's ability to either determine or set the value of a seekbar widget within an Android UI. These widgets represent continuous values and require interaction through specific `(x, y)` coordinates, either by identifying the current position or selecting a new target location (e.g., "set seekbar to 80%"). This task is intentionally constructed to be difficult, as it requires precise spatial reasoning over both visual and structural inputs. It cannot be solved reliably by either modality alone; success depends on the model's ability to correlate visual layout with XML metadata and compute actionable coordinate values.

Performance Across Modalities

The results, presented in Figures 5.12 and 5.13, show that the XML-only (`XML`) modality performs moderately well. While not highly accurate, it produces a number of correct or near-correct predictions. The image-only (`IMG`) modality, by contrast, performs poorly, with a wide and dispersed error distribution. The combined modality (`XML` $\oplus$ `IMG`) also under-performs, only marginally improving upon image-only, and in some cases even trailing XML-only.

Figures 5.15 and 5.14 present modality-specific click behavior in response to interactions with `seekbar`s, specifically for the tasks of setting the media volume to 60% and adjusting the 8kHz equalizer band to -9dB, respectively. In Figure 5.14, a consistent pattern is observed in the `XML` and `XML` $\oplus$ `IMG` modalities, where the model predicts a click near the +9dB region of the 8kHz `seekbar`, despite the prompt requesting -9dB. While the model appears to interpret the magnitude of "9dB" correctly, the directionality indicated by the negative sign is ignored. This occurs despite the visual presence of a labeled vertical scale from +12dB to -12dB on the left side of the interface, suggesting that the negative sign was either visually overlooked or semantically dismissed.

The `IMG` modality performs worse in this task, placing its click on the 16kHz `seekbar` rather than the intended 8kHz target. This misalignment is likely due to the absence of structured layout information, leaving the model to infer positions solely from visual patterns. In contrast, the `XML` and combined modalities are able to correctly locate the 8kHz band—most plausibly by identifying the "8kHz" text node in the XML hierarchy, then resolving its sibling node corresponding to the interactive `seekbar` and computing its center based on bounding box coordinates.

In Figure 5.15, which addresses setting the media volume to 60%, both the `XML` and `XML` ⊕ `IMG` modalities again predict the correct click location on the "Media" `seekbar`. This alignment likely stems from the presence of the label "Media" in the XML, which allows the model to associate the prompt with the corresponding UI element. Notably, the model performs this alignment without the aid of any arithmetic tools or helper abstractions—inferring the target position from the four bounding box edges alone. The `IMG` modality, however, predicts a click in a visually unrelated region near the top of the screen, which does not correspond to any active `seekbar`.

Interpretation and Observations

The relative success of the XML-only modality can be attributed to the presence of bounding box metadata for the seekbar element. When prompted to set a value like "80%," the model can use this structural information to estimate a coordinate corresponding to that percentage along the widget's axis. In contrast, the image modality lacks such direct cues. The model must first visually identify the widget, interpret its scale and orientation, and then infer the appropriate position to interact with—an inherently difficult sequence without precise spatial anchors.

The fact that `XML` ⊕ `IMG` slightly outperforms image-only indicates that the XML still contributes useful information in the combined modality. However, the overall poor performance shows that the model struggles to reconcile spatial data across modalities in a precise, coordinated way. This is further evidenced by the error distribution: correct clicks tend to be exact, while incorrect ones are significantly off-target, reflecting an all-or-nothing pattern in prediction reliability.

Implications for Design and Evaluation

This task illustrates the limitations of current multi-modal prompting strategies in scenarios that require spatial alignment and fine-grained localization. It is one of the few examples where adding visual input degraded the performance relative to using XML alone, suggesting that naive multi-modal fusion can hinder reasoning in high-precision tasks.

The difficulty of this task underscores the importance of carefully designed prompts and alignment strategies when working with coordinate-dependent interactions. Further work is needed to improve grounding between visual and structural elements, particularly for UI widgets that span both modalities semantically but offer distinct kinds of spatial data.

6.4 Threats to Validity

We recognize several potential threats to the validity of our evaluation and results:

6.4.1 Internal Validity

Our evaluation relies on human annotations for key metrics such as correctness, robustness, readability, and reasoning quality. Although we applied a statistical validation process to mitigate subjectivity and calculated correlations (e.g., between equivalence and average reasoning score) to check consistency, some interpretative bias may still remain. Additionally, our system's current scope explicitly excludes tests involving backend or API validations. This limitation may lead to an underestimation of the true potential of the generated tests, as their robustness and effectiveness could be enhanced with backend integrations.

For the interaction tasks involving spatial or semantic understanding, we rely on task-specific automatic metrics (e.g., coordinate distance, exact ID match, edit distance). These may fail to capture near-miss predictions or semantically correct but formally invalid outputs. The absence of manual correction or qualitative inspection in such cases may under-report partial success. Furthermore, without controlled ablation studies or targeted error analysis, some claims—such as the benefits or interference of combining visual and structural inputs—remain speculative.

6.4.2 External Validity

The evaluation was conducted on three representative Volvo applications (Alarm Clock, System Settings, Load Indicator), which may limit generalizability. The performance of the system on other domains, app types, or non-Android platforms remains untested.

Additionally, the visual and structural properties of the selected applications may influence the observed effectiveness of different input modalities. For example, tasks involving visual counting or spatial localization may behave differently in UIs with complex layouts or sparse metadata. As such, the reported modality-specific trends should not be assumed to generalize across all UI environments without further validation.

6.5 Overall Framework Performance

The overall performance of the proposed framework shows promising results in generating executable and meaningful automated tests that par with the manual tests but still remains several critical areas for improvement.

Across both RQ1 and RQ2 evaluations, the framework successfully generates tests that are often concise (lower or comparable lines of code compared to manual tests) and functionally aligned with developer-defined global intents, particularly in simpler application contexts like Alarm Clock. The correctness rates, reasoning quality, and equivalence measures confirm that the system effectively captures developer-intended behaviors in many cases, demonstrating the potential of LLM-driven test generation using in real testing scenarios.

However, several limitations emerged in more complex applications such as System Settings and Load Indicator. The framework struggles to handle complex UI structures with dynamic or deeply nested elements, which leads to noticeably lower correctness and reasoning scores. It also falls short in maintaining robustness, as the automated tests often lack the defensive coding practices or validation layers that are present in the manual test baselines. Additionally, the system has difficulty avoiding flakiness, particularly in scenarios that require precise scrolling or seekbar adjustments, where even minor differences in execution can produce inconsistent app states. Finally, the readability of the generated code is limited, as the lack of modularization and absence of comments reduce the maintainability and clarity of the automated tests.

Despite these weaknesses, the framework's ability to generate functionally aligned automated tests, semantically interpret developer-defined global intents, and produce executable local actions represents an important advancement toward reducing manual testing effort. Future enhancements focused on improved prompt engineering, backend and API integration, memory-aware planning, and automated post-processing could further strengthen the framework's performance and move it closer to becoming a robust tool for industrial-scale automated testing.

7

Conclusions and Future Work

This thesis set out to examine whether large language models can translate high-level, natural-language testing intents into reliable Android UI test scripts. Grounded in Design Science Research, the work combined systematic crawler design, a modular LLM-driven agent, and empirical evaluation on three production-grade Volvo applications. The resulting framework demonstrates that intent-driven automation is not only feasible but also capable of producing concise and functionally accurate tests that reduce manual scripting effort.

By coupling structured exploration with multi-modal reasoning, the system achieved a 71.4% step-level correctness across 42 planner actions and generated tests that, in simpler interfaces, matched or outperformed manual baselines in code brevity without sacrificing clarity. The inclusion of visual context contributed positively to the overall performance of the system. In the majority of tasks, it had no measurable effect on accuracy, but crucially, it also did not degrade performance. In a notable subset of cases, however, visual input significantly improved the agent's ability to identify the correct UI element or action, leading to successful completions that were not achievable through structural information alone. These gains illustrate that visual and structural data can play complementary roles. While there was a specific interaction type where visual input led to a decline in performance due to coordinate ambiguity, such cases were rare. Overall, visual context proved beneficial more often than not and can serve as a valuable addition when applied with consideration for task characteristics.

The study also surfaced clear limitations. Test robustness lagged behind manual scripts due to sparse assertions and sensitivity to timing, and performance dropped in highly dynamic or deeply nested interfaces where state tracking and reasoning became less reliable. Nonetheless, the strong positive correlation between reasoning quality and functional equivalence confirms that qualitative assessment of LLM rationale is a meaningful indicator of test validity.

Overall, this research contributes a concrete architecture, a reproducible evaluation protocol, and evidence that LLM-based agents can narrow the gap between developer intent and executable verification. While challenges remain in scaling to complex state spaces and reinforcing defensive test practices, the thesis provides a foundation for future refinement of intent-aligned, vision-aware testing tools.

While the proposed framework demonstrates the feasibility of generating executable, meaningful, and semantically aligned test cases for Android applications, several areas remain underdeveloped and offer promising directions for future work.

Interaction Validity and Assertion Coverage

A critical limitation in the current pipeline is the absence of sufficient defensive assertions. The framework frequently attempts invalid interactions without verifying the existence or visibility of the targeted UI elements. Future implementations should incorporate preconditions—such as element existence, interactivity, and visibility checks—before executing any interaction. This would significantly reduce error propagation and improve overall test reliability.

Enhancing Screen Understanding Across Modules

The quality of the interaction, observation, and planning modules is directly tied to their understanding of the current screen context. Misinterpretations at any of these stages often lead to invalid actions, incorrect assertions, or flawed intent planning. Improving prompt engineering and enabling deeper correlation between XML structure and visual features may help the LLM generate more context-aware responses. Structured prompt templates, visual–structural alignment, and training on UI-specific reasoning tasks could serve as mechanisms to boost understanding.

Robust State Detection and Tracking

One of the most persistent challenges encountered during this project was reliable state detection. Determining whether a newly visited screen is truly distinct or a slight variation of a known state remains an unresolved issue. The current memory-based screen comparison methods are prone to both false positives and false negatives, resulting in either under-exploration or graph explosion during crawling. Developing a hybrid state detection mechanism that combines SSIM with more advanced semantic embeddings or learning-based screen representations may yield more reliable state equivalence judgments.

Revisiting Crawling Strategies

Although we transitioned away from full crawling in favor of intent-driven limited exploration, this decision was motivated by limitations in state detection, not by inherent flaws in crawling. With improved state tracking and de-duplication, it may be possible to revisit crawling-based approaches to achieve broader coverage. This could be particularly beneficial for applications with deeply nested menus or non-linear navigation paths, where full exploration remains useful for building interaction graphs or discovering unreachable states.

Tool Design and Protocol Abstraction

A promising direction for future development is the standardization of tool interfaces through a structured *Model Context Protocol* (`MCP`). `MCP`[1] is an abstraction protocol that defines how tools can be represented as structured function calls with typed inputs and outputs, allowing them to be invoked by a language model in a controlled and deterministic manner. Each tool (e.g., seekbar setter, toggle switcher, element selector) is described through a schema that specifies the function name, input arguments, output structure, and side effects.

In this framework, `MCP` would allow tools to be defined independently of the model and executed remotely. The LLM would be provided with the available tool definitions, and instead of generating raw code or free-form descriptions, it would select and invoke tools by emitting structured requests conforming to the `MCP` schema. This provides a standardized, inspectable interface between planning modules and execution backends.

Integrating `MCP` into the framework would allow for better modularization, remote execution on test rigs, and safer interaction handling. It would also enable systematic logging, testing, and extension of tool capabilities without altering the planner logic.

Backend Integration

The framework currently lacks a mechanism for interacting with backend APIs, even though many application behaviors—such as mode switching, settings updates, or state confirmations—are mediated by backend services. While we implemented a packet sniffer to observe and correlate UI actions with backend traffic, this pipeline stops at passive analysis. Future work should focus on designing a module that synthesizes API calls based on observed traffic and partial documentation (e.g., incomplete Swagger files or undocumented endpoints). Such a module could use LLMs to infer the appropriate request structure and generate valid payloads. These backend interactions could serve two purposes: (1) setting up complex application modes by issuing API calls directly, and (2) validating frontend operations by querying the backend to ensure consistency.

In particular, these assertions would allow the framework to verify that user-facing changes are correctly reflected in the vehicle's internal systems. Many automotive applications communicate with ECUs[2] over the backend. Confirming that a UI-based mode change is reflected in the corresponding ECU would significantly increase test reliability and enable better end-to-end validation.

[1] For more information on `MCP`, see `https://modelcontextprotocol.io/`, `https://www.anthropic.com/news/model-context-protocol`

[2] ECU (Electronic Control Unit) refers to embedded systems in vehicles that control specific hardware functions, such as engine control, brakes, or transmission.

Bibliography

[1] Saaket Agashe et al. *Agent S: An Open Agentic Framework that Uses Computers Like a Human*. 2024. arXiv: 2410.08164 [cs.AI]. URL: https://arxiv.org/abs/2410.08164.

[2] Domenico Amalfitano et al. "MobiGUITAR: Automated Model-Based Testing of Mobile Apps". In: *IEEE Softw.* 32.5 (Sept. 2015), pp. 53–59. ISSN: 0740-7459. DOI: 10.1109/MS.2014.55. URL: https://doi.org/10.1109/MS.2014.55.

[3] Domenico Amalfitano et al. "Using GUI ripping for automated testing of Android applications". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 258–261.

[4] Android Developers. *Espresso Idling Resources*. https://developer.android.com/training/testing/espresso/idling-resource. Official guidance on synchronization in Espresso tests; accessed 2025-04-29.

[5] Android Developers. *Espresso Testing Framework*. https://developer.android.com/training/testing/espresso. Accessed: 2025-04-29.

[6] Android Developers. *Layouts*. https://developer.android.com/guide/topics/ui/declaring-layout. Accessed: 2025-04-29.

[7] Android Developers. *UI Automator*. https://developer.android.com/training/testing/ui-automator. Accessed: 2025-04-29.

[8] Anthropic. *Building Effective AI Agents*. https://www.anthropic.com/engineering/building-effective-agents. Accessed: 2025-05-22. 2023.

[9] Gilles Baechler et al. *ScreenAI: A Vision-Language Model for UI and Infographics Understanding*. 2024. arXiv: 2402.04615 [cs.CV]. URL: https://arxiv.org/abs/2402.04615.

[10] Young-Min Baek and Doo-Hwan Bae. "Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 238–249. DOI: 10.1145/2970276.2970313.

[11] Jinze Bai et al. *Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond*. 2023. arXiv: 2308.12966 [cs.CV]. URL: https://arxiv.org/abs/2308.12966.

[12] Anne Chao. "Nonparametric estimation of the number of classes in a population". In: *Scandinavian Journal of Statistics* 11.4 (1984), pp. 265–270.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated test input generation for Android: Are we there yet?" In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2015), pp. 429–440.

[14] Browser-Use Developers. *Browser-Use: The AI Browser Agent*. https://browser-use.com. Accessed: 2025-05-25. 2025.

[15] Sergio Di Martino et al. "GUI Testing of Android Applications: Investigating the Impact of the Number of Testers on Different Exploratory Testing Strategies". In: *Journal of Software: Evolution and Process* 36.7 (2024), e2640. DOI: 10.1002/smr.2640.

[16] Sidong Feng et al. "Enabling Cost-Effective UI Automation Testing with Retrieval-Based LLMs: A Case Study in WeChat". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1973–1978. ISBN: 9798400712487. DOI: 10.1145/3691620.3695260. URL: https://doi.org/10.1145/3691620.3695260.

[17] Andrew Gelman et al. *Bayesian Data Analysis*. 3rd ed. CRC Press, 2014.

[18] I. J. Good. "The population frequencies of species and the estimation of population parameters". In: *Biometrika* 40.3–4 (1953), pp. 237–264.

[19] Alan R. Hevner et al. "Design science in information systems research". In: *MIS Q.* 28.1 (Mar. 2004), pp. 75–105. ISSN: 0276-7783.

[20] Zhiyuan Huang et al. *SpiritSight Agent: Advanced GUI Agent with One Look*. 2025. arXiv: 2503.03196 [cs.CV]. URL: https://arxiv.org/abs/2503.03196.

[21] Zheng Hui et al. "WinClick: GUI Grounding with Multimodal Large Language Models". In: *arXiv preprint* (2025). arXiv: 2503.04730.

[22] ISO/IEC/IEEE. *ISO/IEC/IEEE 24765:2017 – Systems and Software Engineering Vocabulary*. https://www.iso.org/standard/71952.html. International Standard. 2017.

[23] Richard Kissel. *Glossary of Key Information Security Terms*. NIST IR 7298 Revision 3. National Institute of Standards and Technology. 2019.

[24] Yuanchun Li et al. "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. DOI: 10.1109/ICSE-C.2017.8.

[25] Dingning Liu et al. *3DAxisPrompt: Promoting the 3D Grounding and Reasoning in GPT-4o*. 2025. arXiv: 2503.13185 [cs.CV]. URL: https://arxiv.org/abs/2503.13185.

[26] Yu Liu et al. "Are LLMs good at structured outputs? A benchmark for evaluating structured output capabilities in LLMs". In: *Information Processing Management* 61.5 (2024), p. 103809. ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2024.103809. URL: https://www.sciencedirect.com/science/article/pii/S0306457324001687.

[27] Fanbin Lu et al. *ARPO:End-to-End Policy Optimization for GUI Agents with Experience Replay*. 2025. arXiv: 2505.16282 [cs.CV]. URL: https://arxiv.org/abs/2505.16282.

[28] Xinbei Ma, Zhuosheng Zhang, and Hai Zhao. *CoCo-Agent: A Comprehensive Cognitive MLLM Agent for Smartphone GUI Automation*. 2024. arXiv: 2402.11941 [cs.CL]. URL: https://arxiv.org/abs/2402.11941.

[29] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for Android apps". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 224–234.

[30] OpenAI. *Introducing Operator*. https://openai.com/index/introducing-operator/. Accessed: 2025-05-25. 2025.

[31] Samad Paydar, Mahdi Houshmand, and Elham Hayeri. "Experimental study on the importance and effectiveness of monkey testing for android applications". In: *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)*. 2017, pp. 73–79. DOI: 10.1109/CSICSSE.2017.8364659.

[32] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *J. Manage. Inf. Syst.* 24.3 (Dec. 2007), pp. 45–77. ISSN: 0742-1222. DOI: 10.2753/MIS0742-1222240302. URL: https://doi.org/10.2753/MIS0742-1222240302.

[33] OpenATX Project. *openatx/uiautomator2*. https://github.com/openatx/uiautomator2. Accessed: 2025-04-29. 2024.

[34] G. A. F. Seber. *The Estimation of Animal Abundance and Related Parameters*. 2nd ed. Macmillan, 1982.

[35] Ting Su et al. "Stoat: Automated framework for efficient exploration of Android apps". In: *Proceedings of the 2017 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM. 2017, pp. 66–76.

[36] Gregory Valiant and Paul Valiant. "Estimating the unseen: An $n/\log n$-sample estimator for entropy and support size". In: *Proceedings of STOC 2011*. ACM, 2011, pp. 685–694.

[37] Wenyu Wang et al. "Vet: identifying and avoiding UI exploration tarpits". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '21. ACM, Aug. 2021, pp. 83–94. DOI: 10.1145/3468264.3468554. URL: http://dx.doi.org/10.1145/3468264.3468554.

[38] Junda Wu et al. *Visual Prompting in Multimodal Large Language Models: A Survey*. 2024. arXiv: 2409.15310 [cs.LG]. URL: https://arxiv.org/abs/2409.15310.

[39] Qinzhuo Wu et al. *MobileVLM: A Vision-Language Model for Better Intra- and Inter-UI Understanding*. 2024. arXiv: 2409.14818 [cs.CL]. URL: https://arxiv.org/abs/2409.14818.

[40] Juyeon Yoon, Robert Feldt, and Shin Yoo. "Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents". In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2024, pp. 129–139. DOI: 10.1109/ICST60714.2024.00020.

[41] Keen You et al. *Ferret-UI: Grounded Mobile UI Understanding with Multimodal LLMs*. 2024. arXiv: 2404.05719 [cs.CV]. URL: https://arxiv.org/abs/2404.05719.

A

Appendix

A.1 Estimating the number of distinct states

This section is exploratory and not part of the core contributions of the thesis.

Estimating the number of distinct UI states in a mobile application serves three key purposes. First, it provides a way to estimate coverage by comparing the number of observed states to the inferred total. This allows us to know how much of the app has been explored. Second, it helps determine how little crawling effort might be sufficient by indicating when additional exploration is unlikely to uncover many new states. Third, the total number of reachable UI states can act as a rough complexity metric for the app's user experience and interface structure.

Given an Android app, let S_{tot} denote the (finite) set of all reachable UI states, each being the full view hierarchy produced by some interaction sequence. Because an exhaustive crawl is infeasible, we explore ways to obtain a rough estimate of $|S_{\text{tot}}|$ based on a crawl log containing n total state visits. The techniques below are drawn from other domains such as Ecological Statistics and Information Theory, and should be seen as exploratory rather than definitive.

A.1.1 Notation

S_{tot} Total number of distinct reachable UI states.

S_{obs} Number of distinct states observed during the crawl.

f_k Number of states seen exactly k times.

f_1 States seen once (singletons).

f_2 States seen twice (doubletons).

n Total state visits (including repeats): $n = \sum_{k \geq 1} k f_k$.

b_i Interactive widgets unvisited on the first encounter of state s_i.

A.1.2 Chao1 Lower Bound Estimator

Chao's nonparametric estimator [12] may serve as a rough lower bound on the number of distinct items in a population:

$$\hat{S}_{\text{Chao1}} = S_{\text{obs}} + \begin{cases} \dfrac{f_1^2}{2f_2}, & f_2 > 0, \\ \dfrac{f_1(f_1-1)}{2(f_2+1)}, & f_2 = 0. \end{cases} \tag{A.1}$$

The estimator assumes random sampling and a closed population. Since these assumptions do not hold in deterministic app crawling, any result should be treated as a rough lower bound, not a precise measure.

A.1.3 Lincoln-Petersen Capture-Recapture

The Lincoln-Petersen estimator [34], originally used in ecological capture-recapture studies, could provide an alternate way to estimate total cardinality. If two independent crawls are run with comparable effort, the overlap O might reflect how saturated the exploration was:

$$\hat{S}_{\text{LP}} = \frac{L_1 L_2}{O}, \tag{A.2}$$

where L_1 and L_2 are the numbers of unique states found in each crawl. This method assumes independent sampling. In practice, deterministic traversal strategies likely break this assumption, so the resulting estimate may be unstable or misleading.

A.1.4 Good-Turing Missing Mass Adjustment

Good and Turing [18] proposed that the probability of encountering an unseen item on the next draw is roughly f_1/n. Applying this idea, one might adjust the Chao1 estimate to reflect that unseen states still carry mass:

$$\hat{S}_{\text{GT}} = \hat{S}_{\text{Chao1}} \left(1 + \frac{f_1}{n}\right). \tag{A.3}$$

This adjustment is only a rough approximation, and like Chao1, it inherits strong assumptions about the underlying sampling process.

A.1.5 Branching-Aware Bayesian Augmentation

To account for unvisited interactive elements that may lead to undiscovered states, we consider a simple Bayesian augmentation. We assume the number of such hypothetical children u_i follows a Poisson distribution with rate λ, and place a Gamma prior on λ [17]. This is a common approach for count modeling, though its suitability here is uncertain:

$$\mathbb{E}[u_i \mid b_i] = \frac{\alpha + b_i}{\beta + 1}. \tag{A.4}$$

With basic hyperparameters $\alpha = \beta = 1$, this yields a soft adjustment:

$$\hat{S}_{\text{Bayes}} = \hat{S}_{\text{GT}} + \sum_{i=1}^{S_{\text{obs}}} \mathbb{E}[u_i \mid b_i]. \tag{A.5}$$

This formulation is speculative. It implicitly assumes that unexplored widgets are equally likely to lead to new states, which may not be true in practice.

A.1.6 Combined Point Estimate and Bounds

To bound the estimate conservatively, we take:

$$\hat{S}_{\text{tot}} = \min(\hat{S}_{\text{Bayes}}, \hat{S}_{\text{LP}}), \tag{A.6}$$

and define the upper bound as $\max(\hat{S}_{\text{Bayes}}, \hat{S}_{\text{LP}})$. This pairing is entirely heuristic and not based on formal statistical justification. It is intended to provide a plausible range, not an accurate confidence interval.

A.1.7 Information-Theoretic Sample Complexity

Valiant and Valiant [36] showed that any estimator attempting constant relative error must observe:

$$n \gtrsim \frac{S_{\text{tot}}}{\log S_{\text{tot}}} \tag{A.7}$$

samples. Given that app crawls typically gather only $n \leq 100$ visits, this theoretical requirement is rarely met. As such, we do not expect the estimates here to converge or generalize reliably. These bounds serve more as qualitative indicators than quantitative estimations.

A.1.8 Practical Limitations

1. **Traversal bias;** Strategies like DFS or BFS produce non-random samples, which can distort all statistical assumptions.

2. **Sparse data;** Many real crawls yield $f_2 = 0$, which can destabilize Chao1.

3. **Skewed access;** Some UI states are deeply gated or rare, violating equal discovery probability assumptions.

4. **Widget overcount;** Many unexplored widgets do not actually lead to new states, which can inflate b_i.

5. **Deterministic overlap;** Capture-recapture fails if the two crawlers are too similar or too divergent.

A.2 Future Validation Strategy

The estimation techniques outlined in this section remain speculative without empirical validation. Due to time constraints, we were unable to perform a systematic

evaluation of these methods against known ground truths. However, we propose a feasible approach for future work to assess the reliability and utility of these estimators.

A promising strategy is to select one or two relatively simple Android applications whose state spaces can be manually enumerated with reasonable effort. For these apps, we could construct a high-confidence estimate—or in some cases, an exact count—of the number of distinct UI states by exhaustively exploring all reachable screens and transitions. This would provide a baseline ground truth against which estimated values can be compared.

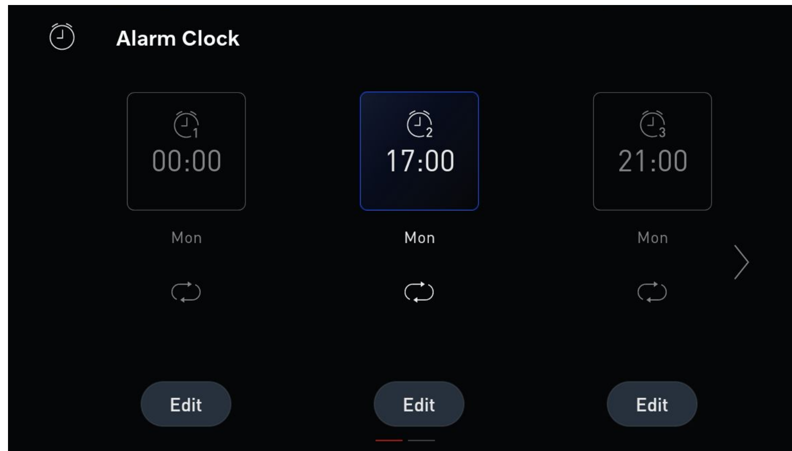
Each of the proposed estimation methods (Chao1, Lincoln-Petersen, Good-Turing adjustment, and Bayesian augmentation) could then be applied to the crawl logs of these applications. Their outputs can be assessed based on proximity to the manually obtained reference count, allowing us to evaluate the relative accuracy and robustness of the estimators. While perfect accuracy is not expected, estimators yielding results in the correct order of magnitude or exhibiting consistent under-/overestimation behavior may still prove useful in practice.

Additionally, this methodology can be extended to a broader set of applications with differing characteristics:

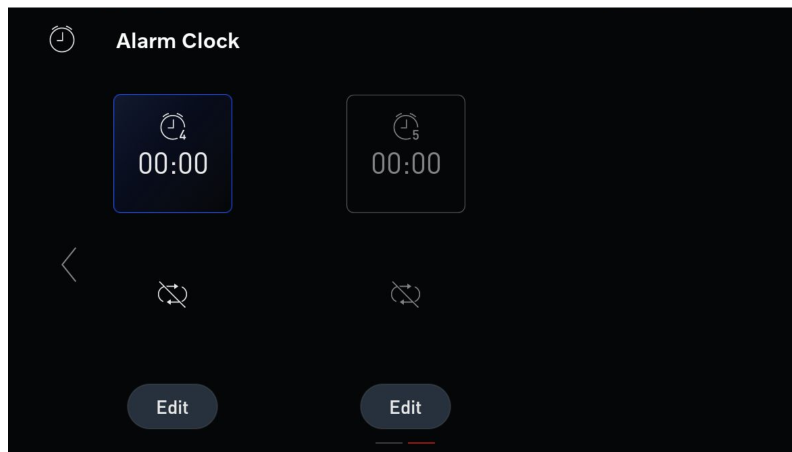
- Applications with near-infinite state spaces due to dynamic content loading or unbounded user input.
- Applications known to have very large but finite state spaces (e.g., on the order of thousands), where the exact count is unknown but approximate scale is understood.

In these scenarios, while ground truth is unavailable, qualitative validation can still be performed by comparing the estimators' outputs to known behavioral patterns of the app (e.g., size, structure, interaction density). Divergence between estimation outputs across applications may also indicate sensitivity to app complexity, which can inform method selection in future work.

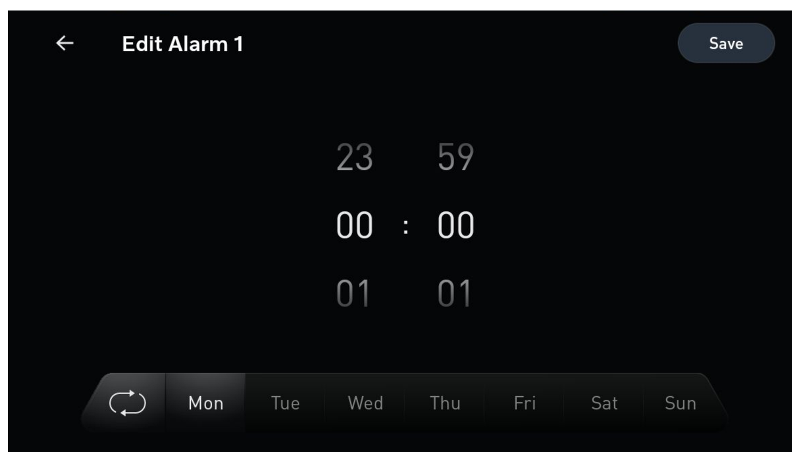
To collect the necessary input data for these estimators, we can employ the existing crawler described in Section 4.6. This crawler captures a sequence of UI states during exploration by hashing screen content, allowing a rough identification of state uniqueness. Alongside hash-based de-duplication, it records metadata such as the number and types of visible widgets, the count and class of interactive elements, and structural layout features. This information can be used to instantiate the variables required by the estimation formulas (f_1, f_2, n, b_i) and provide a useful dataset for empirical evaluation.



(a) Main Page of Alarm

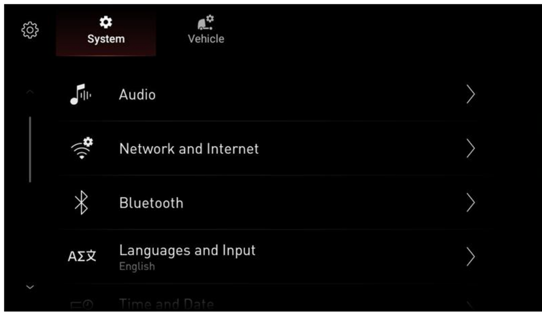


(b) Second Page of Alarm

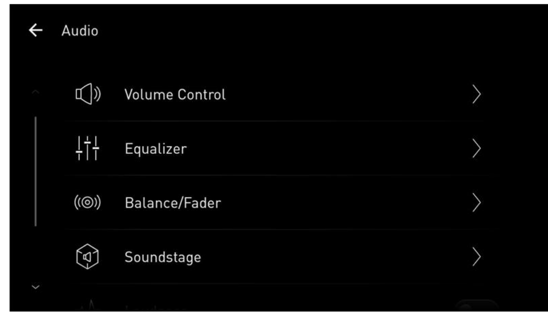


(c) Edit Alarm

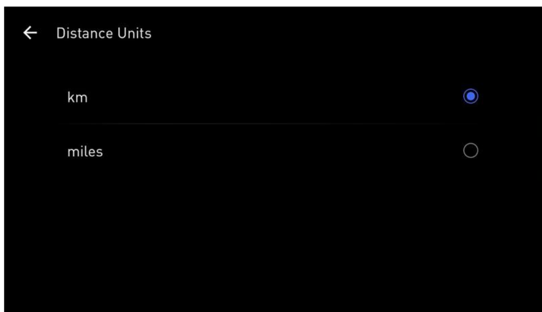
Figure A.1: Alarm Clock App Screenshots



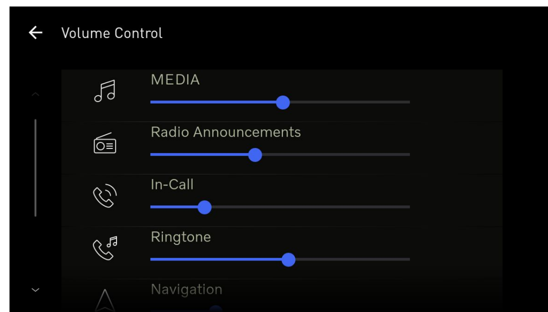
(a) System Main Menu



(b) Audio Settings Page



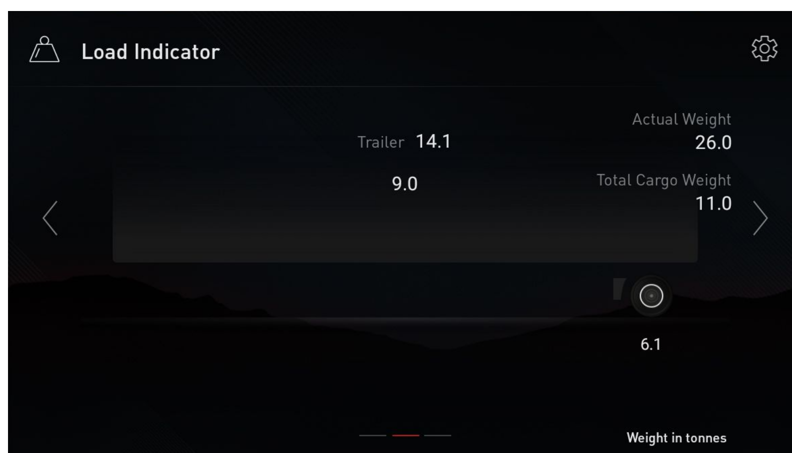
(c) Distance Unit Toggle



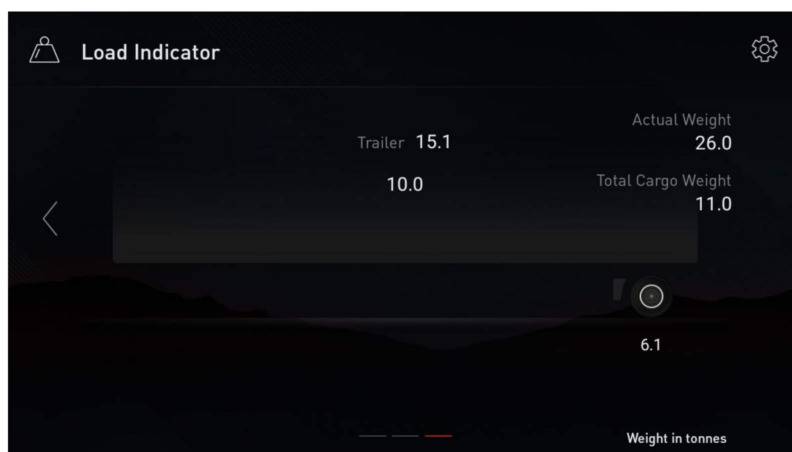
(d) Volume Control Seekbar

Figure A.2: System Settings App Screenshots

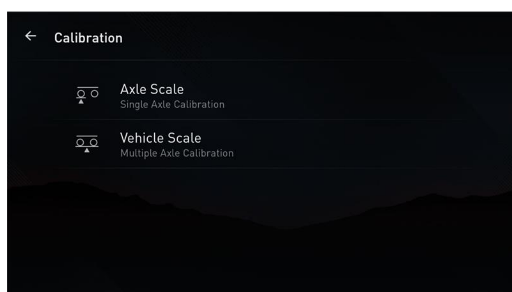
(a) Main Page



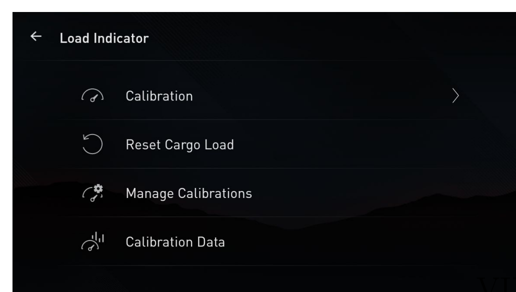
(b) Trailer View 1



(c) Trailer View 2



(d) Calibration Page



(e) Settings Page

Figure A.3: Load Indicator App Screenshots