POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Automated Remediation of Mobile Accessibility Barriers for Blind Users via LLM-Based Multi-Agent Systems

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Irene Lo Presti**

Student ID: 258445
Advisor: Prof. Luciano Baresi
Co-advisors:
Academic Year: 2025-26

For most people, technology makes things easier.
For people with disabilities, technology makes things possible.

— Mary Pat Radabaugh

Abstract

Mobile applications have become essential to everyday life, yet they still present significant accessibility barriers for blind users, who rely on screen readers to interact with digital interfaces. Although automated tools have considerably improved the detection of accessibility issues, their remediation remains a major challenge because it requires contextual understanding, semantic reasoning, and accessibility expertise. Large Language Models (LLMs) offer promising capabilities for automated software repair thanks to their ability to understand code and natural language. However, single-model approaches frequently suffer from hallucinations, reasoning inconsistencies, and role drift, limiting their reliability in real-world remediation tasks. This thesis presents an LLM-based Multi-Agent System (MAS) designed to autonomously resolve accessibility barriers in Android applications. Following a *Locate-Suggest-Fix* paradigm, the remediation workflow is distributed among a collaborative team of specialized autonomous agents. The system combines input validation, accessibility analysis, code repair, and iterative self-validation through dedicated agents. To support consistent decision-making, it also introduces an Accessibility Rule Book that formalizes remediation strategies for static and dynamic accessibility barriers affecting blind users while explicitly addressing the risk of auditory clutter. The proposed MAS was evaluated across diverse Android development frameworks, including XML/Java, Jetpack Compose, and Flutter, using state-of-the-art models on artificial, semi-artificial, and real-world testing scenarios. Experimental results show that the architecture consistently generated syntactically valid code patches, maintained a low output rejection rate, and outperformed single-LLM baselines in terms of reliability and remediation quality. Overall, the findings suggest that combining specialized agents with structured validation mechanisms can provide an effective and scalable approach to automated accessibility remediation in mobile applications.

Keywords: Mobile Accessibility, Large Language Models, Multi-Agent Systems, Automated Software Repair, Screen Readers, Android Accessibility.

Sommario

Le applicazioni mobili sono diventate strumenti essenziali nella vita quotidiana, ma continuano a presentare importanti barriere di accessibilità per gli utenti non vedenti, che interagiscono con le interfacce digitali attraverso screen reader. Sebbene gli strumenti automatici abbiano fatto enormi passi avanti nel rilevare queste problematiche, la loro risoluzione rappresenta ancora una sfida significativa, poiché richiede comprensione del contesto, ragionamento semantico e competenze specifiche di accessibilità. I Large Language Models (LLMs) offrono interessanti opportunità per la riparazione automatica del software grazie alla loro capacità di comprendere codice e linguaggio naturale. Tuttavia gli approcci basati su un singolo modello soffrono frequentemente di allucinazioni, incoerenze nel ragionamento e perdita del ruolo assegnato, limitandone l'affidabilità in scenari reali. Questa tesi presenta un Sistema Multi-Agente (MAS) basato su LLM per la risoluzione automatica di problemi di accessibilità in applicazioni Android. Adottando il paradigma *Locate-Suggest-Fix*, il processo di remediation viene suddiviso tra agenti specializzati, responsabili della validazione dell'input, dell'analisi delle problematiche, della modifica del codice e della verifica dei risultati. Viene inoltre introdotto un Accessibility Rule Book, che formalizza le strategie di risoluzione di problematiche statiche e dinamiche relative agli utenti non vedenti, includendo linee guida per prevenire fenomeni di inquinamento acustico. Il sistema è stato valutato su diversi framework di sviluppo Android utilizzando modelli LLM all'avanguardia su scenari di test artificiali, semi-artificiali e reali. I risultati mostrano che l'architettura genera costantemente patch sintatticamente corrette, mantiene un basso tasso di rigetto degli output e supera le baseline basate su singoli LLM in termini di affidabilità e qualità delle correzioni proposte. Nel complesso, i risultati suggeriscono che la combinazione di agenti specializzati e meccanismi strutturati di autovalidazione rappresenti un approccio efficace e scalabile per l'automazione della riparazione dell'accessibilità nelle applicazioni mobili.

Parole chiave: Accessibilità Mobile, Large Language Models, Sistemi Multi-Agente, Riparazione Automatica del Software, Screen Reader, Accessibilità Android.

Contents

1 | Introduction

The reliance on **mobile applications** is growing every day. However, 72% of users with disabilities report accessibility barriers when using them [2]. Failing to meet accessibility requirements not only hinders equal opportunities but also restricts the full digital participation of individuals with disabilities, ultimately limiting their independence.

1.1. Problem Statement

Digital barriers affect users with motor, cognitive, auditory, and visual impairments in fundamentally different ways. Therefore, each category must be addressed with specific strategies.

In particular, barriers faced by **blind users** have unique challenges: interacting with an application exclusively through audio guidance fundamentally alters the user experience (UX). Clear and precise content descriptions become essential, as well as the communication of dynamic changes.

Research efforts have mainly focused on the detection of accessibility issues, but this remains insufficient. In fact, a recent investigation conducted on a large number of Android applications found that even after multiple version updates, 81% of these applications do not address the detected accessibility issues [4]. This happens because the **remediation of accessibility barriers** presents profound technical complexities: it requires specialized knowledge, time, and resources. Furthermore, rule-based approaches have significant limitations for blind users' barriers because they often require semantic understanding of the context.

Large Language Models (LLMs) emerge as a promising alternative for addressing the remediation challenge, as they can understand semantic context. However, they present some limits, such as hallucinations, reasoning inconsistencies in long contexts, and role drifting.

1.2. Proposed Solution

LLM-based **Multi-Agent Systems** (MAS) are a favorable solution to the automated accessibility remediation problem. They allow the Separation of Concerns and offer techniques to reduce LLMs' inherent limitations, such as hallucinations.

We propose an LLM-based MAS where each agent is responsible for a specific phase of the remediation process, ranging from input validation to code modification. The interaction among agents is orchestrated through **iterative feedback loops**, enabling progressive refinement of the output and ensuring robustness against hallucinations. It takes as input the source code of an Android application and an accessibility log, and it outputs **code patches** correctly inserted into a copy of the original files, along with a clear and comprehensive report. The agents are designed as a professional development team:

- **Interpreter**: it performs input validation and translates the accessibility log.
- **Analyzer**: it estimates the severity level of the problem, and it suggests a strategy to resolve it.
- **Critic**: it reviews the Analyzer's output, checking if it is correct and without hallucinations.
- **Repairer**: it implements a Python script that saves the fix suggested by the Analyzer Agent in a new file, and creates a validation routine to check its syntax.
- **Reviewer**: it checks the Repairer's work by ensuring that the suggestion was addressed correctly.
- **Admin**: it manages the other agents and the feedback loops iterations.

Note that if the Critic or the Reviewer rejects the Analyzer or Repairer output, respectively, then the latter agents have to redo their work. They have three attempts each managed by the Admin Agent. The flowchart in Fig. 4.2 represents the system's execution flow. The architecture was developed with a Python script that employs the Microsoft AutoGen framework.

The work focuses **solely** on issues arising for **blind users**. To address the issues optimally, we designed an Accessibility Rule Book that categorizes the issues into Static and Dynamic, highlighting the Auditory Clutter problem.

1.3. Obtained Results

The evaluation of the system follows three Research Questions (RQs):

RQ1 How **effective** is the proposed MAS in correctly interpreting accessibility logs, formulating valid remediation strategies, and generating syntactically correct code patches across different Android frameworks?

RQ2 To what extent does the multi-agent architecture, specifically, through the iterative validation of the Critic and the Reviewer Agents, **mitigate LLM hallucinations** and improve patch reliability compared to a single-LLM baseline?

RQ3 How do different foundational **models** (e.g., Gemini 2.5 PRO vs. GPT-5.2) impact the reasoning capabilities, code quality, and overall stability of the automated remediation workflow?

To answer these questions, the evaluation systematically assesses the performance of each agent composing the MAS throughout the entire remediation workflow.

Empirical results demonstrate the **high effectiveness** and **reliability** of the proposed multi-agent architecture. By delegating the remediation workflow to specialized agents, the system successfully generated **syntactically correct code** patches across multiple Android frameworks (XML, Jetpack Compose, and Flutter). Crucially, the introduction of the Critic and the Reviewer validation loops drastically **reduced** the occurrence of **hallucinations** typically associated with single-LLM baselines, maintaining an output rejection rate below 3%. While the overall performance is partially bound by the reasoning capabilities of the underlying foundation models (Gemini 2.5 PRO and GPT-5.2), the findings confirm that a self-regulating, multi-agent approach provides a robust and scalable solution for automated software remediation.

1.4. Contributions

This thesis contributes to the field of automated mobile accessibility remediation through the following key advancements:

- **A novel LLM-based Multi-Agent System for Android accessibility remediation**: the proposed architecture automates the complete remediation workflow, from accessibility log interpretation to source-code modification, following a *Locate-Suggest-Fix* paradigm specifically tailored to mobile accessibility barriers affecting blind users.

- **An Accessibility Rule Book for blind-user accessibility issues**: a structured knowledge base was designed to categorize both static and dynamic accessibility barriers, define remediation strategies, and explicitly address the risk of auditory clutter during the repair process.
- **A self-validating multi-agent architecture for hallucination mitigation**: the system introduces dedicated agents that continuously inspect intermediate outputs through iterative feedback loops, reducing hallucinations and reasoning inconsistencies commonly observed in single-LLM solutions.
- **Support for multiple Android development frameworks**: the proposed approach is designed to operate across XML/Java, Jetpack Compose, and Flutter applications, enabling remediation in heterogeneous Android development environments.
- **An extensive empirical evaluation**: the architecture was evaluated on artificial, semi-artificial, and real-world accessibility scenarios using state-of-the-art foundation models, demonstrating high patch correctness and significant improvements over single-LLM baselines.

1.5. Thesis Outline

This thesis is organized as follows:

2 State of the Art: provides the theoretical background and reviews the current literature necessary to contextualize this work. It explores the application of Large Language Models in Software Engineering, the paradigms of Multi-Agent Systems, and the specific principles and tools currently available for mobile accessibility detection and remediation.

3 Problem: formalizes the accessibility remediation bottleneck. It provides a quantitative analysis of the current barriers affecting blind users and details the structural limitations of both manual interventions and single-LLM approaches when addressing context-dependent semantic repairs.

4 Solution: details the proposed multi-agent architecture. It describes the internal Accessibility Rule Book, the specific responsibilities of the six specialized agents (Interpreter, Analyzer, Critic, Repairer, Reviewer, Admin), the implementation of the iterative validation loops, and the technical framework choices.

5 Evaluation: presents the empirical evaluation of the system. It answers the defined

RQs by analyzing the system's effectiveness, reliability, and hallucination mitigation capabilities across artificial, semi-artificial, and real-world Android applications, ultimately comparing the multi-agent approach against single-LLM baselines.

6 Conclusion: summarizes the contributions of this work, highlights the structural advancements made over the current state of the art, discusses the architectural limitations, and outlines potential directions for future research.

2 | State of the Art

This chapter provides the theoretical background and the review of the state of the art necessary to contextualize the proposed work. It addresses Large Language Models, the principles of mobile accessibility, and methodologies to detect and fix accessibility barriers.

2.1. Software Engineering Design Principles

Let's begin this excursus with some of the most relevant Software Engineering (SE) design principles.

The principle of **Separation of Concerns** (SoC), originally introduced by Dijkstra [5], is a fundamental software engineering concept aimed at managing the complexity of large systems. According to this principle, a software system should be decomposed into distinct components, each responsible for addressing a specific concern or functionality. By isolating different concerns, developers can reduce interdependencies between components, improve maintainability, facilitate testing, and simplify system evolution.

The **Locate-Suggest-Fix** repair paradigm is a common example of SoC. It is adopted in automated software maintenance and accessibility remediation systems. In this paradigm, the problem is divided into three distinct phases: identifying the location of the issue within the source code (Locate), proposing a remediation strategy (Suggest), and generating the corresponding code modification (Fix). Each phase requires different reasoning capabilities and produces outputs that serve as inputs for the subsequent stage. [38]

2.2. Large Language Models for SE

Language is a fundamental human cognitive ability that enables complex expression and communication. Equipping **machines** with powerful Artificial Intelligence (**AI**) algorithms allows them to understand and **communicate** in the form of human language.

Large Language Models (LLMs) represent one of the most successful approaches toward this objective. LLMs are neural networks trained on massive collections of textual

data to **predict** the next token (i.e., the fundamental data unit the model processes) in a sequence. Through this training process, they learn grammatical structures, semantic relationships, and contextual patterns, acquiring the ability to understand and generate natural language. [39]

As parameter, data, and computational scales increase, LLMs exhibit *emergent abilities*, i.e., capabilities that are not explicitly programmed but arise from scale. These abilities include reasoning, instruction following, information extraction, and code generation.[39]

Software Engineering (SE) is one of those areas reaping the benefits of LLMs' emergent capabilities. Since many software engineering activities involve understanding, generating, or transforming textual artifacts such as source code, documentation, requirements, and bug reports, they can be effectively approached using LLMs.[15]

2.2.1. Code-LLMs

Among various applications of LLMs in SE, **code generation** and code modification are particularly relevant. Early studies showed that general-purpose language models (such as GPT-3) could already generate simple programs from natural language descriptions, even without being specifically trained for programming tasks.[3]

This observation led to the development of ***Code-LLMs***, i.e., language models trained or fine-tuned on large code repositories. By learning the structure and patterns of programming languages, these models significantly improve their performance in tasks such as code completion, code summarization, code translation, code repair, and source-code generation from natural language specifications. One of the first prominent examples is *Codex*.[3][15]

Modern Code-LLMs, including systems integrated into tools such as *GitHub Copilot*, can assist developers throughout the **software development lifecycle**. Beyond generating source code, they are also capable of understanding existing implementations, explaining their behavior, identifying defects, and proposing modifications.[18]

2.2.2. Techniques

In this section, we explore various techniques to improve the performance of LLMs in SE tasks.

Prompt Engineering is a method of enhancing model performance by using task-specific instructions, known as prompts, without modifying the core model parameters. This

approach enables LLMs to seamlessly integrate into downstream tasks solely based on the given prompts, guiding model behavior without updating model parameters. When using this technique, there are two possible approaches:

1. **Few-Shot Prompting**: the model is provided with a limited number of examples to perform a specific task. The model learns from these examples and generalizes to similar tasks with minimal training data.
2. **Zero-Shot Prompting**: the model is expected to perform a task without any explicit training on that task. Instead, it relies on the prompt provided during inference to generate the desired output.[15]

Semi-Formal Reasoning is a structured prompting methodology that requires the LLM to construct explicit premises, trace execution paths, and derive formal conclusions. Unlike the unstructured Chain-of-Thought technique, where each prompt is built upon the preceding one, Semi-Formal Reasoning acts as a certificate: no case can be skipped, and claims must always be supported. It is proven that in patch equivalence verification, fault localization, and code answering questions, this technique consistently improves accuracy.[30]

2.2.3. LLMs Limitations

LLMs have demonstrated remarkable capabilities across a wide range of natural language processing tasks. However, they present the following limitations.

Sycophantic Behavior, in the LLM context, is the tendency to agree with or flatter users excessively. This poses significant risks to its reliability and ethical deployment. This behavior can manifest in various ways, from providing inaccurate information to align with user expectations to offering unethical advice when prompted, or failing to challenge false premises in user queries.[22]

Hallucinations happen when the LLMs' generated output is nonsensical, incoherent (with the provided input or the precedent responses), or gets stuck in repetitive loops. This hinders performance and raises safety concerns for real-world applications. [17]

Long Context Struggles occur when LLMs take long contexts as input. Research shows that performance can degrade significantly when changing the position of relevant information, indicating that current LLMs do not robustly make use of information in long input contexts. They observed that performance is often highest when relevant information occurs at the beginning or end of the input context, and significantly degrades when models must access relevant information in the middle of long contexts, even for

explicitly long-context models. [20]

2.3. Multi-Agent Systems

LLMs' ability to reason and plan as humans aligns with the expectation of autonomous **agents** that can perceive surroundings, make decisions, and take actions. Hence, LLM-based agents have been studied and rapidly developed to understand and generate human-like instructions, facilitating sophisticated interactions and decision-making in a wide range of contexts. [14]

A Multi-Agent System (MAS) is a set of agents interacting to achieve a common purpose. [21] **LLM-based MAS** have been proposed to leverage the **collective intelligence** and **specialized profiles** and skills of multiple agents. Compared to systems using a single LLM-powered agent, multi-agent systems offer advanced capabilities by specializing LLMs into various distinct agents, each with different capabilities, and by enabling interactions among these diverse agents to simulate complex real-world environments effectively. In this context, multiple autonomous agents collaboratively engage in planning, discussions, and decision-making, mirroring the co-operative nature of human group work in problem-solving tasks. This approach capitalizes on the communicative capabilities of LLMs, leveraging their ability to generate text for communication and respond to textual inputs. Furthermore, it exploits LLMs' extensive knowledge across various domains and their latent potential to specialize in specific tasks. Research has demonstrated **promising results** in utilizing LLM-based multi-agents for solving various tasks, such as **software development**. [14]

2.3.1. Agents Profiling and Communication

In LLM-MAS, **agents** are defined by their traits, actions, and skills, which are tailored to meet **specific goals**. Across various systems, agents assume **distinct roles**, each with comprehensive descriptions encompassing characteristics, capabilities, behaviors, and constraints. For instance, in software development, agents could take on the roles of product managers and engineers, each with responsibilities and expertise that guide the development process. These profiles are crucial for defining the agents' interactions and effectiveness within their respective environments. [14]

The agents' communication is the critical infrastructure supporting collective intelligence. Three **communication paradigms** can be followed:

1. **Cooperative**: the agents work together towards a shared goal or objectives, typi-

cally exchanging information to enhance a collective solution.

2. **Debate**: the agents engage in argumentative interactions, presenting and defending their own viewpoints or solutions, and critiquing those of others. This paradigm is ideal for reaching a consensus or a more refined solution.

3. **Competitive**: the agents work towards their own goals that might conflict with the goals of other agents.

The **communication structure** can be chosen from four options:

1. **Layered** communication: hierarchically structured, with agents at each level having distinct roles and primarily interacting within their layer or with adjacent layers.

2. **Decentralized** communication: peer-to-peer network where agents communicate with each other.

3. **Centralized** communication: a central agent coordinates the system's communication, with other agents primarily communicating through this central node.

4. **Shared Message Pool**: shared message pool where agents publish messages and subscribe to relevant messages based on their profile, thereby boosting communication efficiency.

Agents' capabilities can learn and evolve **dynamically**. This can be done through feedback, i.e., the critical information that agents receive about the outcome of their actions, helping them **learn** the potential impact of their actions and **adapt** to complex and dynamic problems. There are four kinds of feedback:

1. **Feedback from Environment**: in software development scenarios, the agents obtain feedback from the code interpreter.

2. **Feedback from Agents Interactions**: the feedback comes from the judgment of other agents or from agents' communications.

3. **Human Feedback**: crucial for aligning the MAS with human values and preferences.

4. **None**: in some cases, there is no feedback provided to the agents. [14]

2.3.2. LLM-MAS Limitations and Some Remedies

Given that this kind of Multi-Agent Systems is based on LLMs, the LLMs' limitations persist.

In particular, **Sycophantic Behavior** [22], that also manifests in the interactions between agents, and **Long Context Struggle** [20] remain difficult to overcome.

Instead, **Hallucinations** [17] are still present in LLM-MAS, but some remedies have been found. One of the most successful ones is the use of a **LLM-as-a-Judge**, i.e., an LLM employed as an evaluator for complex tasks. More specifically, this kind of LLM is tasked with determining whether something falls within the scope of a given rule. When they are incorporated into multi-agent frameworks, their goal is to guide inter-agent interactions, thereby improving overall decision-making efficiency and quality. In particular, some studies have employed GPT-4 to identify logically structured yet nonsensical statements, while others have proposed critique-based approaches for hallucination evaluation, relying on evidence selection and detailed feedback generation. [13]

In addition to LLMs' limitations, a practical challenge emerges in extended collaboration: **Role Drift**, where agents deviate from their designated responsibilities. This phenomenon manifests as boundary violations (e.g., a planner writing code), redundant work, conflicting decisions, and futile debates, ultimately degrading system performance. An interesting approach is *RoleFix*, a lightweight framework for detecting and repairing role drift in multi-agent collaboration. The most compelling idea introduced by this framework is a structured protocol requiring agents to **declare their role**, commitments, and dependencies at each turn. Experiments on SE and research workflow tasks demonstrate that *RoleFix* reduces role drift incidents by 67.4% and improves task completion rates by 23.8% points compared to baseline multi-agent systems, while introducing only 8.3% latency overhead. [31]

2.3.3. MAS for Software Repair

Agent-based software repair approaches have received widespread attention, as repair agents can automatically analyze and localize bugs, generate patches, and achieve state-of-the-art performance on repository-level benchmarks. Zhang et al. propose an interesting approach: ***SGAgent***, a Suggestion-Guided multi-Agent framework for **repository-level software repair**. It follows the *Localize-Suggest-Fix*, introducing also a suggestion to strengthen the transition from localization to repair. The suggester starts from the buggy locations and incrementally retrieves relevant context until it fully understands the bug, and then provides actionable repair suggestions. Experimental results show that SGAgent with Claude-3.5 achieves 51.31% repair accuracy, 81.2% file-level, and 52.4% function-level localization accuracy.[38]

2.4. Accessibility in Mobile Applications

Mobile devices have become an indispensable part of daily life, offering users **unprecedented access** to digital services. However, they also introduce specific **usability challenges**, such as limited display size, unique interaction methods, diverse usage contexts, and a lack of platform consistency. These characteristics can significantly impact the user experience, particularly for **people with disabilities**. While accessibility evaluations have been thoroughly studied and standardized in the web domain, assessing **accessibility** in the context of **mobile devices** poses distinct technical and methodological **challenges**. Significant progress has been made globally through the adoption of standardized requirements, most notably the Web Content Accessibility Guidelines (**WCAG**). Recent iterations of these guidelines have explicitly expanded to address the mobile context, introducing specific success criteria to handle touch interactions, target sizes, and complex gestures. [27]

Despite the availability of these expanded guidelines, adherence by software developers and content authors remains highly **inconsistent** in practice. Ensuring mobile accessibility is further complicated by the diverse ecosystem of devices and operating systems, each presenting technical constraints and interaction models. Consequently, evaluating and repairing mobile applications requires a comprehensive understanding of **multiple factors**, including device characteristics, interface design, app functionality, and specific capabilities of the end users. [27]

2.4.1. Accessibility for Blind Users

Blind users interact with mobile applications differently from sighted users. While sighted users rely on visual information such as icons, layouts, and animations, blind users access the same information through assistive technologies (AT). As a consequence, design choices that appear obvious visually may become inaccessible when translated into audio.

For example, an icon representing a search function is immediately recognizable to a sighted user. However, if no textual description is provided, a blind user will only perceive an unlabeled element and will not understand its purpose.

The primary assistive technology used by blind users is the **screen reader**, i.e., an AT that renders text and image content as speech. It is a software application that attempts to convey what sighted users see on a display through non-visual means. In particular, Google's Android provides the **Talkback** screen reader. [32]

When blind users encounter an unfamiliar application, they navigate through the on-

screen elements sequentially to understand the app's layout. The gesture of **swiping** right and left allows the screen reader to move to the next or previous element, respectively. When an element is focused, the screen reader vocalizes its **textual description**, enabling users to gauge its functionality. The single-tap action used by sighted users corresponds to the double-tap action for screen reader users. [25]

The accessibility barriers a blind user can encounter fall into two categories: Static and Dynamic.

Static Accessibility Issues

Static Accessibility Issues concern elements that are visible when the screen loads. These problems typically prevent blind users from understanding the structure or purpose of interface components.

For instance, consider a screen containing a heart-shaped icon used to save a song as a favorite. A sighted user can easily infer its purpose from the icon itself, but a screen-reader user can only understand the button if an appropriate textual description is provided. Without such information, the button becomes unusable.

Therefore, in accordance with WCAG 1.1.1 [33], a proper label is mandatory when the component lacks visible text. Clearly, the description must be straightforward and **useful** as stated in WCAG 2.4.6 [36]. In addition, labels must not be redundant and must only be applied to non-decorative elements.

Moreover, visual relationships between elements must be clearly conveyed to screen-reader users. Consider a simple form containing the label "Name" followed by the value "Jane". A sighted user immediately perceives that the two elements are related because of their visual proximity. However, a blind user may navigate them separately and lose this contextual connection.

To preserve these relationships, related elements must be **grouped** so they are announced together, as recommended by WCAG 1.3.1. [34]. In the previous example, the screen-reader should present "Name: Jane" as a single piece of information rather than requiring two separate navigation actions. Furthermore, the information must follow a logical **presentation order**, with labels announced before their corresponding values, complying with WCAG 1.3.2 [35].

Dynamic Accessibility Issues

Dynamic Accessibility Issues arise when information changes after the screen has already been loaded.

For example, when a user adds a song to a playlist, an application may display a temporary notification stating "Song added successfully". A sighted user immediately notices the message because it appears on the screen. However, a blind user may never become aware of its existence unless the screen reader is explicitly instructed to announce it.

Dynamically changing the visual content of an application, e.g., through animations, is a commonly used technique to enhance visual aesthetics and, more importantly, to guide users' attention to specific parts of the screen. Therefore, specific techniques must be applied so that screen readers announce these dynamic changes. This includes:

- elements that **appear**, **disappear**, or both from the loaded screen, e.g., the notification that appears and disappears after a few seconds;
- components that **move** on the screen, e.g., when the app-related information is shifted to the bottom of the screen when the user presses the install button;
- **changes of attributes** of a static element, e.g., the progress of an app installation process. [25]

Auditory Clutter and Cognitive Load

It is important to point out that providing **excessive or repetitive** information is equally detrimental.

For example, the objective of decorative images is to enhance the mobile application's visual aesthetic, meaning that blind users do not need to learn about them. So, a verbose description such as "beautiful abstract blue waves background pattern" for a decorative image is only confusing.

This phenomenon, often referred to as **Auditory Clutter**, occurs when an interface **overwhelms** a screen-reader user with redundant, overly verbose, or poorly structured announcements. Blind users cannot quickly glance past irrelevant text as sighted users filter visual elements, because the auditory channel is strictly sequential. This can increase the user's cognitive load. [4][12][26]

Furthermore, the Auditory Clutter phenomenon concerns instances in which the AT presents information that is actually not intended to be available on the screen, such as **invisible information** that is covered or disabled for sighted users. For instance,

this can involve elements placed out of the screen boundaries, empty placeholders, or background layouts entirely covered by foreground panels. [23]

Balancing the right amount of information (ensuring it is descriptive yet concise) is delicate and essential.

2.5. Accessibility Testing and Detection

Testing techniques are essential for assessing accessibility. Over the years, a large number of methodologies and tools have been developed to assess mobile accessibility. The three main methods are:

1. **Manual Testing**: experts thoroughly examining applications to uncover issues.
2. **Automated Testing**: specialized tools are used to analyze code for identifying potential accessibility issues based on specific criteria.
3. **User Testing**: real individuals from a target audience interact with the application, observed by researchers. This provides valuable insights into user experiences, revealing accessibility barriers and deepening the researchers' understanding of users' challenges, leading to more effective improvements.

While user testing remains the standard for uncovering real usability issues, it is highly resource-intensive. Consequently, academia and industry have heavily invested in **automated testing tools** to detect accessibility violations at scale systematically.[27] A detailed analysis of their categories, together with an in-depth examination of selected tools, follows.

Static Analysis Tools evaluate the application without executing it, typically by inspecting the source code. The most prominent example in the Android ecosystem is Android Lint, a static code analyzer integrated into Android Studio that flags missing content descriptions or suboptimal touch target sizes during development.

Traditional Scanners analyze statically the rendered UI. Although effective for identifying deterministic static issues, e.g., missing labels, these scanners suffer severe limitations: intensive manual intervention (leading to low activity coverage) and the rule-based nature (inadequate to detect contextual or semantic errors). An example is the official Google Accessibility Scanner.

Dynamic Exploration dynamically navigates through different application states with a specific software, such as a crawler or a bot. It analyzes the state (sense phase) to extract the user interface hierarchy. Then it decides (think phase) which action to execute based

on the provided algorithms. Finally, it executes the event on the device or emulator (act phase), and the cycle restarts from the sense phase. It allows to analyze run-time generated content and complex navigation flows. The most used crawling tool in Android applications is *DroidBot*, a lightweight UI-guided test input generator. It can interact with an app on almost any device without instrumentation. It generates UI-guided test inputs based on a state transition model generated on-the-fly, and allows users to integrate their own strategies and algorithms.[19] Researchers have developed specialized dynamic tools to address specific interaction constraints, the following sections explore the most relevant ones.

2.5.1. Multimodal Detection

The ***A11yArgus*** framework integrates with *DroidBot* to analyze Android applications. It unifies structural and visual evidence, enabling the detection of **13 accessibility issue** types: from traditional contrast and labeling errors to gesture-only navigation and overlapping components. It achieves over 91.9% precision.[11]

To bridge the gap between sighted touch interactions and screen reader usage, the ***A11yPuppetry*** framework employs an innovative **Record-and-Replay** approach. This framework records standard touch gestures from a tester, translates them into equivalent TalkBack actions, and automatically replays them. This process uncovers critical issues, such as unfocusable elements or ineffective actions, that purely touch-based automated tools fail to identify.[29]

2.5.2. Temporal Analysis for Dynamic Changes

Despite these multimodal improvements, conventional dynamic crawlers typically wait for the user interface to reach a completely stable state before capturing a snapshot, thereby completely missing intermediate dynamic content changes. To address this temporal dimension, frameworks like ***TimeStump*** have been developed to continuously monitor applications in action rather than relying on stable-state snapshots. By comparing pre-action and post-action frames alongside real-time accessibility event logs, it successfully localizes **latent dynamic changes** that evade traditional analysis.[25]

2.5.3. LLM-Assisted Detection

In the sophistication of advanced dynamic and temporal frameworks, automated tools inherently struggle with contextual and **semantic understandings**. Recently, LLMs have

emerged as powerful instruments to bridge this semantic gap in accessibility detection.

For example, the ***ScreenAudit*** system utilizes LLMs to analyze captured TalkBack transcripts alongside UI metadata. By interpreting the actual speech output that a blind user would hear, it identifies nuanced screen reader accessibility errors, such as redundant labels, inappropriate grouping, and poor label quality.[40]

Furthermore, LLMs are increasingly being leveraged to streamline the manual user testing. Tools like ***Reca11*** automatically parse auto-generated transcripts from accessibility user testing sessions. By employing advanced prompt engineering and contextual guidelines, it can accurately extract, semantically match, and compile the actual accessibility and usability issues reported by disabled testers in real-world scenarios. [16]

2.6. Accessibility Remediation

The problem of automatically repairing accessibility issues has received significantly less attention than automated detection. In fact, detecting the accessibility barriers is only the first step: developers must understand the problem, locate it in the code, and determine and implement the most appropriate solution without introducing new defects.

This remediation process is particularly challenging because accessibility fixes are often context-dependent: they may depend on the application's structure, the user interaction flow, and the assistive technologies involved.

Recent advances in **LLMs** have opened **new opportunities** for automating accessibility remediation. Thanks to their ability to understand natural-language descriptions, reason about source code, and generate code modifications, LLMs can potentially support developers throughout the remediation process.

A 2025 **study** of Aljedaani et al. investigated whether LLMs can effectively **detect and resolve** accessibility violations in mobile applications. By analyzing common accessibility issues reported in the literature, this research aimed to determine whether LLMs can serve as reliable tools for improving mobile app accessibility. They used state-of-the-art LLMs (such as ChatGPT, Gemini, and Llama) to detect, categorize, and propose remediation for violations identified with an automated Accessibility Scanner. They found out that while LLMs show potential in automating accessibility analysis, challenges such as **inconsistent** detection accuracy, contextual **misinterpretation**, and **limitations** in generating effective remediation persist. Their findings highlight the need for further advancements to enhance LLMs' reliability in real-world applications. [1]

An interesting automated tool is ***FixAlly***. It is designed to suggest source code fixes for accessibility issues detected by automated accessibility scanners in iOS mobile applications. It employs a **multi-agent system** LLM architecture that follows the **Locate-Suggest-Fix** paradigm: it first localizes the reported accessibility violation within the source code, then generates a remediation strategy, and finally produces one or more candidate patches. The results demonstrate the potential of multi-agent LLM architectures to assist developers in accessibility remediation. By decomposing the task into specialized stages, FixAlly achieves better performance than single-agent approaches and shows that LLMs can effectively support the generation of accessibility-related code modifications. However, the authors explicitly define the objective of the system as the generation of *plausible* fixes rather than guaranteed correct solutions. Moreover, FixAlly analyzes accessibility violations individually, which **limits** its ability to reason about dependencies among related issues and broader application-level accessibility concerns. The approach also focuses primarily on single-view source files and does not address issues requiring coordinated modifications across multiple files or screens. Finally, the authors acknowledge that the applicability of the framework **beyond the iOS** ecosystem remains an **open research question**.[24]

These limitations suggest that, although **LLM**-based remediation is a **promising** direction, further research is needed to support more complex repair scenarios, improve the reliability of generated fixes, and provide stronger guarantees about the correctness of the proposed solutions.

2.7. Research Gap

Despite the significant advancements in automated accessibility detection and the emerging use of LLMs for software repair, the domain of automated accessibility remediation for mobile applications remains largely underexplored. This review of the state of the art reveals several **critical gaps** that hinder the adoption of fully autonomous repair systems in real-world mobile development.

To systematically illustrate these gaps, Table 2.1 summarizes the current landscape across eight key dimensions:

- AD: Automated Detection;
- AR: Automated Remediation;
- LB: LLM-Based;

- AO: Accessibility-Oriented;
- AS: Android Support;
- HA: Holistic Analysis, i.e., if the issues are managed in a correlated manner;
- DC: Dynamic Content;
- SV: Self-Validation, i.e., if it has an internal feedback to manage errors (such as hallucinations).

Approach/Tool	AD	AR	LB	AO	AS	HA	DC	SV
SGAgent[38]	✓	✓	✓	-	✓	✓	-	-
A11yArgus[11]	✓	-	-	✓	✓	-	-	-
A11yPuppetry[29]	✓	-	-	✓	✓	✓	✓	-
TimeStump[25]	✓	-	-	✓	✓	✓	✓	-
ScreenAudit [40]	✓	-	✓	✓	✓	✓	✓	-
Reca11[16]	✓	-	✓	✓	✓	✓	✓	-
Aljedaani et al. [1]	-	✓	✓	✓	✓	-	-	-
FixAlly[24]	-	✓	✓	✓	-	-	-	-

Table 2.1: Summary of existing mobile accessibility tools

As highlighted by the comparative summary, the current literature presents three major unfulfilled requirements.

First, while numerous autonomous tools are dedicated to detecting accessibility issues in mobile applications, the availability of corresponding **remediation frameworks** is disproportionately **low**. In particular, there is a distinct **lack** of robust remediation tools specifically designed to handle the complexities of the **Android** ecosystem and its native assistive technologies, such as TalkBack.

Second, existing approaches tend to analyze and address accessibility violations in strict isolation. Treating these barriers as standalone, localized defects severely limits the effectiveness of the generated repairs. Mobile accessibility often involves dynamic content and complex user interface hierarchies, making it essential to evaluate multiple interrelated issues simultaneously to coordinate cohesive, application-wide solutions rather than applying **disjointed patches**.

Lastly, current remediation frameworks **fail** to fully **exploit the potential of Multi-Agent Systems** to mitigate inherent LLM limitations, such as hallucinations and syco-

phantic behavior. Existing approaches consistently **lack** built-in **validation mechanisms**, leaving a critical need for rigorous processes capable of critiquing and verifying proposed modifications to ensure they are logically sound before deployment.

3 | Problem

As of 2026, the usage of **mobile applications** continues to grow: the number of global mobile subscriptions is over 8.9 billion, and it is expected to exceed 9.4 billion by 2030 [6]. This growth is driven by the increasing reliance on mobile applications in everyday life, from digital banking to healthcare access to social interaction. However, as the SOMAA (State of Mobile App Accessibility) Report [2] highlights, 72% of users with disabilities report **accessibility barriers** when using mobile applications, which results in a poor or ineffective experience.

Approximately 1.3 billion people (about 16% of the global population [37]) are currently living with a significant disability. Consequently, making applications accessible effectively means extending their usability to an additional 16% of users. More importantly, from an ethical standpoint, it is crucial to ensure that users with disabilities have equal access to digital applications: failing to meet accessibility requirements not only hinders **equal opportunities** but also restricts the full digital participation of individuals with disabilities, ultimately limiting their independence.

However, digital accessibility is not a monolithic concept. Digital barriers affect users with motor, cognitive, auditory, and visual impairments in fundamentally different ways. Consequently, a "one-size-fits-all" approach to remediation is structurally ineffective: each category of disability requires **ad-hoc strategies** and specific technical interventions. Among these, the barriers faced by **blind users** present uniquely intricate challenges: interacting with an application exclusively through a screen-reader fundamentally alters the user experience, as the two-dimensional visual layout is transformed into a sequential auditory stream. Therefore, a poorly calibrated solution, such as an overly verbose label, a redundant description, or a misaligned focus order, can generate severe **auditory clutter** and render the application entirely **unusable**. The necessity of **high precision** combined with the need for a **deep semantic understanding** of the interface makes resolving visual-to-auditory translation barriers exceptionally difficult, creating a critical bottleneck in the software development lifecycle.

3.1. Remediation Bottleneck

Research efforts in both academia and industry have mainly focused on the accessibility issues **detection**, while the problem of automated remediation remains largely unaddressed.

Recent empirical investigations highlight the severity of this gap. A large-scale analysis of Android applications revealed that nearly **89%** of them suffer from **accessibility issues**, with an average of 43 violations per application and 6.5 per screen. More alarmingly, the study demonstrates a critical **stagnation in issue resolution**: across version updates, the number of accessibility issues remained unchanged in over 81% of the applications analyzed. The development teams rarely fix these problems, and frequently, the release of new features introduces additional accessibility barriers rather than resolving existing ones. [4]

Detecting an accessibility issue is only the first step. Once a problem has been identified, developers must determine how to modify the source code without breaking the application.

Consider a simple example: an automated accessibility scanner reports that a button is missing a label. The developer must locate the corresponding component in the code, understand how it is used within the application, determine an appropriate description, and ensure that the modification does not introduce new accessibility problems. Note that this is a remarkably common scenario: empirical data shows that missing item labels are consistently among the top three most frequent accessibility violations in mobile applications. [4]

This remediation process often becomes the real bottleneck. The main **challenges** include:

1. time and resource consumption: the code must be manually investigated to implement the correct fixes;
2. specialized knowledge: to correctly fix accessibility issues, it is necessary to understand how assistive technologies work, how users with disabilities use them, and what the state of the art is for the specific programming language used;
3. human errors: manual intervention in a complex environment can cause new bugs or technical debt.

However, **rule-based solutions** have significant limitations. While this approach is effective for deterministic tasks, such as adjusting the contrast ratio or increasing the touch-

target size, it becomes more complex to manage when addressing issues that require contextual or **semantic understanding**. For instance, detecting an image without a label is relatively straightforward, whereas generating an appropriate and meaningful description requires contextual and semantic understanding. Moreover, accessibility remediation is highly **delicate**: incorrect fixes may worsen the user experience instead of improving it. For example, if an icon of a magnifying glass is grouped with the text "Search", adding a description to the icon will cause the screen-reader to announce "Search Icon, Search". This phenomenon is called **auditory clutter** and is caused by excessive accessibility announcements that overwhelm screen reader users with redundant or frequent feedback.

Finally, accessibility remediation is highly **context-dependent**: the same issue may require different fixes depending on the application structure and user interaction flow.

3.2. Challenges for LLM-based Remediation

When manual and rule-based remediation strategies prove insufficient, Large Language Models (LLMs) emerge as promising alternatives for addressing these accessibility challenges. However, the adoption of LLMs introduces several challenges:

- **Hallucinations**: LLMs often generate text that is illogical or unfaithful to the provided source input [17], e.g., an LLM may suggest modifying a component that does not actually exist;
- **Reasoning inconsistency in long context**: LLMs do not robustly utilize information in long input contexts [20], e.g., in large applications, the relevant information may be scattered across multiple files, so the model may correctly identify an issue in one section while overlooking important constraints described elsewhere;
- **Role drifting**: LLMs frequently shift from the given objective, e.g., when asked to identify accessibility issues in a source code file, an LLM may start proposing fixes. [31]

In addition, modifying existing source code is considerably more challenging than generating code from scratch. The generated patches must preserve the original intended application behavior. Despite these limitations, LLMs possess a crucial advantage: their ability to understand semantic context and adapt their solution to the given input. E.g., an LLM can choose an accurate description for an element.

3.3. Reasons for a Multi-Agent Approach

Accessibility remediation requires several distinct activities: understanding the reported issue, deciding how to solve it, modifying the source code, and verifying that the generated solution is correct. Asking one single LLM to perform all these tasks simultaneously can easily lead to degraded performance and reasoning errors.

An analogy can be drawn with a software development team. A developer writes the code, a reviewer checks its quality, and a tester verifies that the final result behaves as expected. Each role focuses on a specific responsibility.

A promising approach to address these challenges is the use of **Multi-Agent Systems**. They allow the application of the **Separation of Concerns** technique by defining a specific scope for each agent. This reduces the context and produces more precise and centered outputs. They are well-suited to **handle hallucinations** by using specific agents to review the work of others. [28]. Finally, their LLM-based nature allows them to understand context and give tailored answers.

4 | Solution

This chapter introduces the proposed solution, describing the overall system architecture, the rationale for each agent, and the technical framework choices.

4.1. LLM-Multi-Agent System

We developed an **LLM-MAS** where each agent is responsible for a specific phase of the **remediation process**, ranging from input validation to code modification. The interaction among agents is orchestrated through iterative feedback loops, enabling progressive refinement of the output and ensuring **robustness against hallucinations**. To do so, we adopted the paradigm of Feedback from Agents Interactions [14]. The MAS takes as input the source code of an Android mobile application and the accessibility log produced by either an autonomous tool or an expert, and outputs **code patches** correctly inserted into a copy of the original files, along with a clear and comprehensive **report** of the work done by the agents.

The MAS objectives are:

1. evaluate the **severity** of the identified issues;
2. suggest the **appropriate fix** for each issue without compromising the existing code;
3. accurately insert the generated code patches.

The team of agents is designed to mimic a **professional development team**.

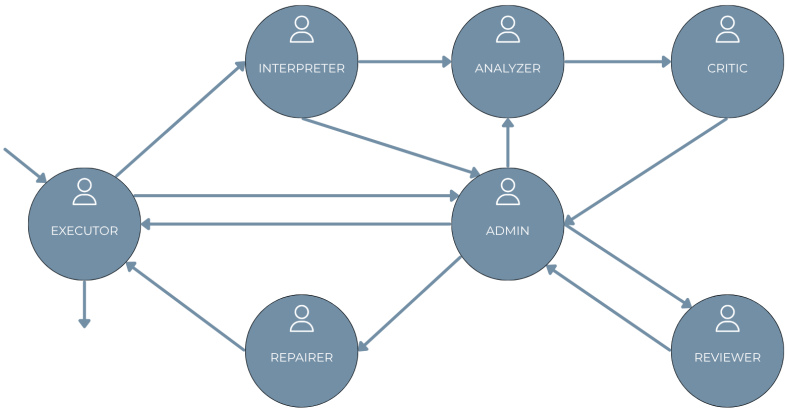


Figure 4.1: The MAS Architecture

Fig. 4.1 represents the MAS architecture as the interaction between agents.

4.1.1. Implementation Framework and Technology Choices

Microsoft **AutoGen** was selected for the MAS as it is well-suited for automating complex software activities. It is **open-source** and, most importantly, it is specialized in **code generation**, which is one of the system's primary goals. Another key feature of AutoGen is its ability to **execute** tools called by the agents as well as code generated by them. This is fundamental for our system. Lastly, since the agents are conversational LLMs, their chats are inherently **human-readable**, facilitating the analysis by the mobile development team.

Python is the programming language used for the MAS implementation. There are numerous reasons behind this choice:

1. AutoGen is a Python-**native framework** meaning it was primarly built as a Python library;
2. Python supports **dynamic code execution**: the agents can generate and execute code;
3. Python has a massive **ecosystem** of several libraries.

Using the AutoGen framework allows the MAS to be designed with a **model-agnostic** architecture, meaning that it works with different LLM models. In particular, the user can choose between:

- Gemini 2.5 PRO;
- GPT-4o;
- GPT-5.2.

4.2. Project Scope: Blind Users

The MAS focuses solely on issues arising for blind users. First of all, to make a sound and working system, each agent cannot rely on excessive information. This means that selecting a **specific demographic** helps in the design and development of the system. The *Automated Accessibility Analysis of Dynamic Content Changes on Mobile Apps* paper by Mehralian, He, and Malek [25] has inspired us to focus on blind users. Furthermore, the barriers they encounter have unique **technical complexities**: ensuring a seamless experience demands a deep semantic reconstruction of the application. By focusing on this demographic, the project addresses one of the most critical hidden flaws of mobile applications.

4.2.1. Accessibility Rule Book

To address the issues optimally, we designed an Accessibility Rule Book. It comprises a first categorization into static issues and dynamic issues.

The **static issues** are divided into:

1. **Element Grouping and Presentation Order**: adjacent and logically related UI elements must be grouped, e.g., a label and its corresponding value;
2. **Intent and Meaningful Labels**: uninformative items (except purely decorative ones) must have a meaningful label, e.g., an icon-only button.

The **dynamic issues** are categorized based on the Mehralian, He, and Malek research [25] into the following:

1. **Appearing Content**: elements that appear and remain on a loaded window must be announced, e.g., the close button that appears after a few seconds. This category can be divided into:
    - Minor window changes: the user must be informed about something, but the workflow was not interrupted;
    - Major window changes: the user focus must be moved because there is a new logical context;

2. **Disappearing Content**: elements that disappear after some time or a user interaction must be announced, e.g., the navigation bar that disappears when the user scrolls through the page;
3. **Short-Lived Content**: elements that appear for a brief duration must be announced both when they appear and disappear, e.g., when saving a song on Spotify, a notification appears to confirm the save and give a go back option, it disappears after a few seconds;
4. **Moving Content**: elements that are relocated on the screen must be announced, e.g., in the App Store, when the app information is shifted to the bottom of the screen after the user presses the install button;
5. **Content Modification**: the status change of elements on screen must be announced, e.g., a text that changes its value.

Auditory Clutter

To address the Auditory Clutter [4] challenge, we describe how to avoid it, depending on the identified issue. In particular:

- STATIC 2 - Intent and Meaningful Labels: risk of Static Clutter, i.e., **Redundant Labeling** [12]. It happens when an interactive element already has visible and meaningful text, e.g., a button displaying "Continue";
- DYNAMIC 5 - Content Modification: risk of Dynamic Clutter, i.e., **High-Frequency Updates** [26]. It happens when an element updates its state or text continuously and rapidly without any direct user interaction, e.g., a countdown timer;
- Dynamic Issues addressed with Live Regions: risk of Structural Clutter, i.e., **Nested Live Regions**. It occurs when the live region attribute is applied to both the parent container and its children.

Methods for resolution

For each identified category, we described how the issues can be addressed in each of the considered frameworks (XML/Java, Jetpack Compose, and Flutter/Dart). To avoid redundancy and excessive fragmentation, the following discussion focuses on general, language-independent solutions.

Static issues resolution:

1. Element Grouping and Presentation Order: enforcement of a coherent structural or-

ganization by encapsulating logically related components within containers properly recognized by screen readers;

2. Intent and Meaningful Labels: introduction of meaningful and context-aware labels guided by the surrounding interface context.

Dynamic issues resolution:

1. Appearing Content:
   - Minor window changes: apply appropriate accessibility attributes, such as "Live Region", to elements that appear on the screen;
   - Major window changes: shift the focus to the new element;
2. Disappearing Content: use appropriate accessibility attributes on the parent containers of the elements that disappear from the screen;
3. Short-Lived Content: prohibit placing clickable buttons inside the short-lived element, and use the appropriate accessibility attributes;
4. Moving Content: explicitly notify the users about the relocation of the element using the correct attributes;
5. Content Modification: add the accessibility attributes to the element that is changing its status.

The Accessibility Rule Book serves as the **shared knowledge base** across the MAS, guiding both the analysis and repair phases, while being selectively exposed to agents to prevent role-drifting [31].

4.3. Agents: a Team of Experts

As previously mentioned, the agents are designed as a professional development team that adopts the Cooperative Communication Paradigm [14]. Each agent has a specific role to allow **Separation of Concerns**. The role definition was based on the **Locate-Suggest-Fix** paradigm [38]: we designed the Interpreter Agent as the Locator, the Analyzer as the Suggester, and the Repairer as the Fixer. We then added the Critic and Reviewer Agents to **mitigate hallucinations**[17][14]. Finally, given the system's complexity, the Admin Agent was added as the project manager, and it coordinates the communication as a central node according to the Centralized Communication Structure [14].

To mitigate **role-drifting** [31], every agent has to announce its own role before addressing

its tasks.

This section describes each agent in detail, outlining its responsibilities, inputs, outputs, and interactions with other agents.

4.3.1. Interpreter

The Interpreter Agent is the primary gatekeeper. It has two main tasks:

1. **Input Validation**: it ensures that the source code is valid, the accessibility log is plausible, and that they are contextually related;
2. **Accessibility Log Translation**: if the input is valid, the Interpreter translates the accessibility log into a report understandable by the Analyzer Agent. In particular, the Interpreter categorizes every issue found into one of the Accessibility Rule Book classes and identifies the exact problematic UI component.

This agent has access to the Accessibility Rule Book but not to the resolution methods to mitigate **role-drifting**.[31] Furthermore, we employed the **Few-Shot Prompting** technique [15] by adding the following examples:

1. appearing and disappearing content: addressing multiple problems;
2. dynamic minor changes and content modification: handling the difference between content modification (dynamic 5) and minor change issues (dynamic 1b);
3. invalid input: spotting the inconsistency between source code and accessibility log.

4.3.2. Analyzer

The Analyzer Agent is an expert on mobile accessibility and inclusive design specifically for blind users. Its primary tasks are:

1. **Severity Level Estimation**: it evaluates the severity of each presented issue to decide whether it should be addressed and in which order. Furthermore, it provides the opportunity for developers to prioritize their efforts;
2. **Suggestions Production**: it provides suggestions to fix the issues to give the best possible experience for blind users, meaning that multiple suggestions must be well coordinated and issues must be addressed only if necessary and not cause any other problems, such as auditory clutter. The agent consults the Accessibility Rule Book to provide extremely granular suggestions, e.g., pointing to specific XML nodes detailing exactly what needs to be added.

To guide the agent, we employed the **Semi-Formal Reasoning** technique [30] by strictly following a precise output schema:

> **Explicit Premises**: extraction of the Interpreter's listed facts.
>
> **Code Trace**: detection of the execution path and data flow of the interested components in the source code. From definition to state changes.
>
> **Evidence**: mention of the exact line numbers and snippets to match the code trace and reference of the corresponding rule from the Accessibility Rule Book.
>
> **UX Assessment**: evaluation of the real impact on the blind user experience with the application. Verify if the suggested fix causes auditory clutter or a poor user experience.
>
> **Severity**: level of severity chosen between NONE, LOW, MEDIUM, HIGH.
>
> **Suggestion**: formal conclusion with the explicit modification to be done on the source code.

In this manner, the Analyzer builds its reasoning incrementally so that the suggestion is proposed as the conclusion of the latter.

In addition to the Accessibility Rule Book and resolution methods, the agent is provided with general information about accessibility for blind users [25] to assess issues confidently.

The **Few-Shot Prompting** technique [15] was also employed with this agent:

1. static grouping and content modifications: how to address multiple problems, both static and dynamic, in an XML and Java application;
2. dynamic content modification: how to address a medium severity issue in a Jetpack Compose application;
3. dynamic disappearing content and static intent: how to address multiple problems, both static and dynamic, in a Flutter application;
4. major window changes: how to address major window changes in a Jetpack Compose application;
5. resolving rule collision with high-frequency data: how to resolve dynamic accessibility issues that can create auditory clutter in a Jetpack Compose application.

Finally, **strict constraints** are listed for the agent to remember them during the elaboration of its response:

1. no conditional delegation: the Analyzer must make a definitive judgment and provide an absolute directive to the Repairer Agent;
2. never use vague terms in identifying the components, the name or ID must be explicitly stated.

4.3.3. Critic

The Critic Agent is designed to ensure the Analyzer's reliability. It is defined as an objective and impartial compliance inspector. Its primary focus is to provide assessments and feedback concerning the Analyzer's output. It is structured as an LLM-as-a-Judge [13]. Its tasks are:

1. **Analysis Verification**: the Analyzer's output must be grounded in the source code and the Interpreter's output;
2. **Severity Level Check**: it must be logically justified;
3. **Hallucination Detection and Suggestion Correctness Check**: confirm that the suggestion is based on the Accessibility Rule Book, is quoted correctly, and accurately selected. It must be in the correct programming language [28].

To examine the Analyzer's suggestion, the Critic Agent has access to both the Accessibility Rule Book and the resolution methods. In addition, we employed the **Semi-Formal Reasoning** technique [30] by defining the following rigid output schema:

Fact-Checking Trace: cross-reference the Analyzer's "Code Trace" and "Evidence" with the source code. The reasoning concludes with the outcome VALID or INVALID.

Rule Alignment: check if the Analyzer followed the Accessibility Rule Book correctly.

Auditory Clutter Check: verification of the Analyzer's causes of auditory clutter. The reasoning concludes with the outcome APPROVED or REJECTED.

Directive Precision: validation that the Analyzer made a definitive decision without conditional delegation. The reasoning concludes with the outcome PASS or FAIL.

Devil's Advocate: formulation of at least one strong reason why the Analyzer's suggestion might be wrong, hallucinated, incomplete, or may result in a poor user experience. This is a strategy to address the LLM agent's tendency to sycophancy.

[22]

Rebuttal: evaluation of the Devil's Advocate step.

Decision: APPROVED or REJECTED based on the reasoning above.

Actionable Feedback: if the Analyzer's work has been rejected, here the Critic explains the reason why.

Finally, **Few-Shot Prompting** [15] was leveraged by queuing the following examples:

1. rejecting a rule violation;
2. rejecting hallucinations;
3. approving a correct analysis.

4.3.4. Repairer

The code repair executor is the Repairer Agent, whose goal is to implement the exact fixes suggested by the Analyzer Agent. The Repairer's tasks are:

1. **Fix Implementation**: implementation of a fix that directly and exclusively addresses the suggestion provided by the Analyzer;
2. **File Saving**: write a Python script that saves the fixed code into a new file.
3. **Validation Routine Integration**: append a validation routine in the Python script to verify the syntax of the generated file. It raises an Exception if the validation fails.
4. **Diff Code Generation**: add the generation of a diff code (git-style) between the old source code and the new one.

To avoid **role-drifting** [31], the Accessibility Rule Book was not provided to this Agent, so it implements the Analyzer's suggestions without applying the rules on its own.

The **Semi-Formal Reasoning** technique [30] is employed by requiring the agent to explain its Repair Plan before outputting the Python code.

The following examples are appended to exploit **Few-Shot Prompting** [15]:

1. fixing the disappearing content in an XML file;
2. fixing a Java file through a programmatic announcement;
3. adding the correct semantics and import when fixing a Jetpack Compose application.

Finally, **strict constraints** are imposed to ensure that the agent understands its role and responsibilities. The Repairer is **not** allowed to:

1. modify, add, or delete any part of the source code that is not strictly requested by the Analyzer's suggestion;
2. refactor code for stylistic reasons, e.g., delete original comments, change variable names, alter indentation unnecessarily;
3. second-guess the Analyzer;
4. forget necessary imports required by the code patches.

4.3.5. Reviewer

Given the delicate nature of the Repairer's work, an agent that checks its output is mandatory: the Reviewer Agent. It is structured as an LLM-as-a-Judge [13]. Its tasks are:

1. **Ensure the Suggestion was Addressed**: the entire Analyzer's suggestion must be implemented without any hallucinations by the Repairer;
2. **Code Patches Correctness**: the new lines written by the Repairer must be correct and working. This means that if the validation routine failed due to pre-existing errors, this is not an issue;
3. **File Saving**: the new code files must be saved correctly;

To prevent **role-drifting** [31] and **hallucinations**, this agent receives the following input:

- the Analyzer's suggestion;
- the output of the Python code written by the Repairer and executed by the Executor Agent;
- the code diff (git-style) showing exactly which lines the Repairer added and removed compared to the original code.

Semi-Formal Reasoning was also used with this agent. Its output schema is the following:

> **Executor Trace**: explanation of the Python code's output.
>
> **Diff Trace and Integrity**: analysis of the code diff by listing exactly what was added and what was removed. The reasoning finishes with PASS or FAIL.

Code Trace: structured breakdown of each issue, including the Analyzer's request, the Repairer's implementation, and their alignment.

Flaw Hunt: active search for a flaw or potential discrepancy between the Repairer's output and the Analyzer's suggestion. This is a strategy to address the LLM agent's tendency to sycophancy. [22]

Rebuttal: evaluation of the traces and the Flaw Hunt.

Decision: APPROVED or REJECTED based on the reasoning above.

Finally, **Few-Shot Prompting** [15] was applied by attaching the following examples:

- reject missing imports in Jetpack Compose;
- reject destructive logic in Java;
- approving perfect execution in Flutter;
- handling pre-existing technical debt in XML.

4.3.6. Admin

At the center of the system, there is the Admin Agent, i.e., the workflow orchestrator. Its primary tasks are:

1. **Agents Managing**: based on the agents' outputs, it has to decide who is next;
2. **Feedback Loops Iteration**: it monitors the number of iterations of each feedback loop, i.e., the Analyzer-Critic dynamic and the Repairer-Reviewer one. After three failed attempts by the Analyzer or the Repairer, the Admin shuts down the entire process;
3. **Remediation Process Completion**: it decides when the process is complete, or must be interrupted.

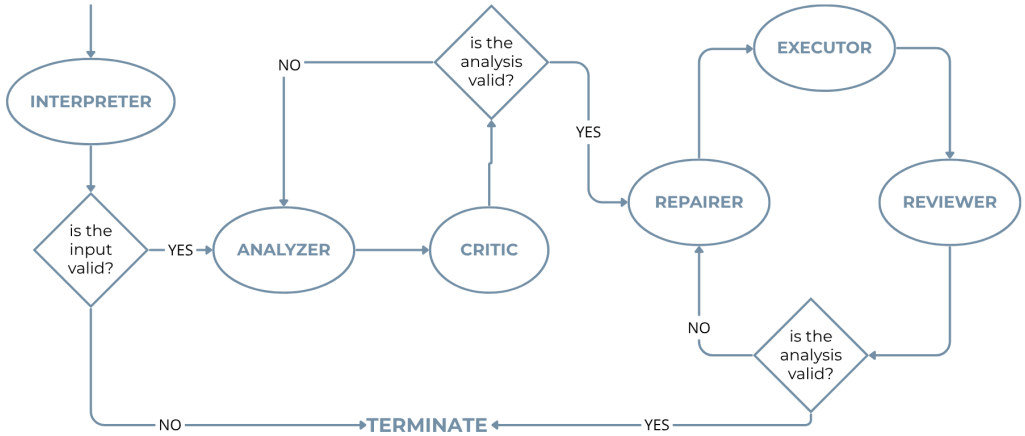


Figure 4.2: The Admin Agent's Workflow

The flowchart in Fig. 4.2 represents the decision process of the Admin Agent and, consequently, the system's execution order.

1. The remediation process starts with the Interpreter's **input validation**. If it is not valid, the Admin explains the problem and terminates immediately. Otherwise, the Interpreter creates the **translation** for the Analyzer Agent.
2. The Analyzer outputs its report, which is evaluated straight away by the Critic. If it is not valid, the Admin demands a redo. Otherwise, it passes the approved **suggestion** to the Repairer Agent.
3. The Executor Agent, i.e., the User Proxy Agent, executes the Python **code** produced by the Repairer. It outputs the results of the syntax check and the diff-code.
4. The Admin Agent passes everything to the Reviewer and waits for its output. If the Repairer's job is invalid, the Admin asks for a redo starting from step (3). Otherwise, the Admin summarizes everything and **terminates** the process.

4.3.7. Summary of Reliability Techniques

Table 4.1 summarizes the main techniques adopted throughout the MAS to address common LLM limitations.

Technique	Limitation	Application
Few-Shot Prompting	Imprecise outputs	Examples provided to most agents to guide reasoning and output structure
Semi-Formal Reasoning	Weak or inconsistent reasoning	Structured schemas requiring explicit premises, traces, evidence, and conclusions
Role Declaration	Role drift	Agents must explicitly state their role before executing tasks
LLM-as-a-Judge	Hallucinations	Critic and Reviewer validate the outputs produced by other agents
Separation of Concerns	Long context struggle	each agent is provided only with the information relevant to its specific task
Specific Constraints Positioning	Long context struggle	Critical constraints are placed at the end of prompts
Devil's Advocate Field	Sycophancy	Critic and Reviewer actively search for flaws before approving outputs.

Table 4.1: Techniques adopted to mitigate common LLM limitations.

4.4. Implementation Details

The MAS is managed with a simple Python script. It is modular and configurable, enabling flexibility in both model selection and execution behavior.

The **Executor Agent** is implemented as a User **Proxy** Agent. It is responsible for initiating interaction and handling code execution. It acts as the bridge between the conversational environment and the underlying system operations.

To help the Admin Agent orchestrate the interaction among agents, a **Finite State Automaton** (FSA) Fig. 4.1 defines the allowed communication flow. This formalization ensures that the system follows a controlled execution path, preventing invalid transitions.

The remediation process is initiated by providing the source code and the accessibility log to the system. Once the process is completed, the entire interaction is serialized and **stored in multiple formats**, including raw JSON for reproducibility, Markdown for

readability, and a refined report version designed for human analysis.

Overall, the implementation emphasizes **modularity**, **transparency**, and **reproducibility**, ensuring that the system can be reliably evaluated and extended.

4.5. Design Constraints and Trade-offs

Even though the system offers a structured and effective solution to accessibility issues in Android mobile applications, it presents some constraints and trade-offs. They represent relevant elements for a correct interpretation of the system potential.

The proposed MAS is not universal. It is specifically designed for **Android** applications developed in XML/Java, Jetpack Compose, and Flutter. Furthermore, it focuses exclusively on accessibility issues for **blind users**, leaving out cognitive and motor impairments as well as partial vision conditions.

Concerning **computational costs and latency**, the system relies on multiple interactive agents, each of which requires LLM API calls. This results in increased execution time and higher operational costs, particularly in the presence of iterative feedback loops.

4.6. Use Case 1: SmartProfile

To explain in detail how the system works, a complete example is described. *SmartProfile* (SM) simulates a common user setting screen with three distinct accessibility barriers. The XML and Java code were generated using Artificial Intelligence, as well as the following mock-up. The remediation process was done with the **Gemini 2.5 PRO** model.

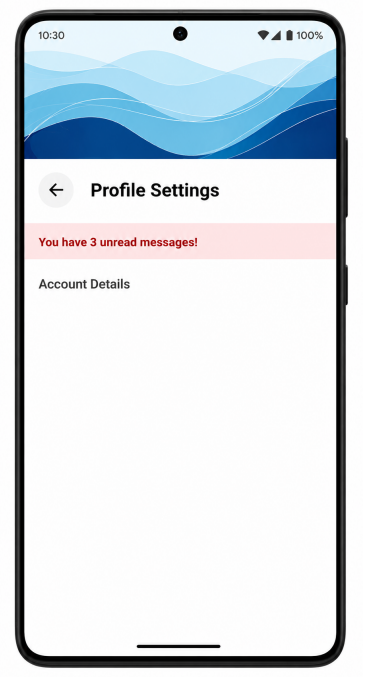


Figure 4.3: UC: SmartProfile Mock-up

- **The Problem**: the user profile screen presents three accessibility barriers, i.e., an unlabeled back button, a decorative image that could potentially create auditory clutter, and a dynamic alert banner that appears without notifying the screen reader.
- **MAS Action**: The system correctly identifies the issues, maps them to the Accessibility Rule Book, and generates targeted code patches without altering the application's visual layout or business logic.
- **The Result**: The back button is labeled, the decorative image is hidden from accessibility services, and the alert banner safely uses a "polite" live region to announce itself.

In particular, the application's code presents two accessibility barriers:

- Missing Label: the back button does not have a descriptor. It violates the Static Rule 2: **Intent and Meaningful Labels**. Therefore, a blind user cannot grab the utility of the component.
- Dynamic Notifications: when the notifications appear on the screen, they are not announced. This violates the Dynamic Rule 1a: **Appearing Content** - Minor Window Changes.

The accessibility log presents a third barrier: the decorative header is missing a descriptor. Note that this is correct according to Static Rule 2 - Intent and Meaningful Labels and

to anti **Auditory Clutter** suggestions. This catch was placed intentionally to test how the system handles potential auditory clutter.

Interpreter Phase

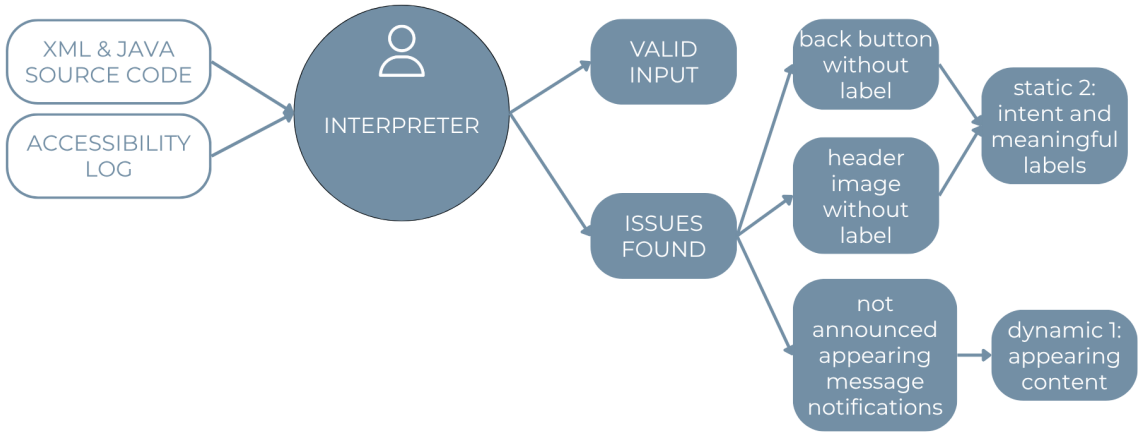


Figure 4.4: SM Interpreter Phase

The Executor Agent starts the conversation by asking the Interpreter to analyze the source code and the accessibility log.

The Interpreter outputs:

> **My role**: I am an objective log parser tasked with translating raw accessibility logs into structured, machine-readable data based on predefined rules.
>
> **Validation Trace**: The accessibility log mentions three UI elements by their resource IDs: `@+id/btn_back`, `@+id/decorative_header`, and `@+id/alert_banner`. A scan of the provided `ACTIVITY_SETTINGS.XML` file confirms that all three elements exist with the specified IDs and component types (ImageButton, ImageView, and TextView, respectively). The `SETTINGSACTIVITY.JAVA` file further confirms the dynamic behavior of `@+id/alert_banner`, which is set to VISIBLE in the onResume method. The inputs are correlated.
>
> **Validity**: VALID.

Accessibility Issues:

- **Element ID/Text:** `@+id/btn_back`
- Issue Category: STATIC
- Violated Rule: STATIC RULE 2 (Intent and meaningful labels)
- Raw Evidence: Missing Content Description android.widget.ImageButton `@+id/btn_back`; "The navigation control to return to the previous screen lacks descriptive text. TalkBack simply says 'Unlabeled button'"
- **Element ID/Text:** `@+id/decorative_header`
- Issue Category: STATIC
- Violated Rule: STATIC RULE 2 (Intent and meaningful labels)
- Raw Evidence: Missing Content Description android.widget.ImageView `@+id/decorative_header`; "The header image has no description."
- **Element ID/Text:** `@+id/alert_banner`
- Issue Category: DYNAMIC
- Violated Rule: DYNAMIC RULE 1b (Major Window Changes)
- Raw Evidence: Dynamic Content Focus Missing android.widget.TextView `@+id/alert_banner`; "When the activity loads and the user has pending notifications, a dynamic alert banner becomes visible at the top, but the screen reader focus remains at the beginning of the screen, leaving the blind user unaware of the alert."

Analyzer-Critic Phase

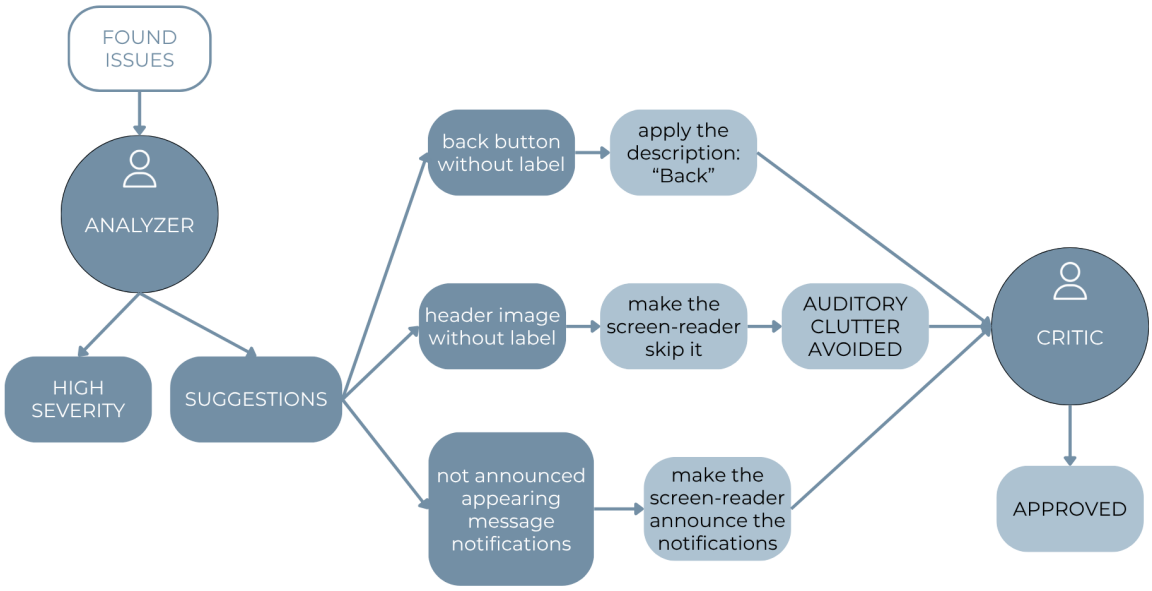


Figure 4.5: SM Analyzer-Critic Phase

The Analyzer Agent receives the interpretation and outputs:

> **My role**: I am Analyzer Agent, an expert on mobile accessibility and inclusive design specifically for blind users. My task is to analyze the accessibility issues of a specific mobile application, decide the severity of the problem, and determine how to fix it.
>
> **Explicit Premesis** The Interpreter Agent reported three issues on the User Profile screen:
>
> 1. An ImageButton with ID `@+id/btn_back` is missing a content description, violating STATIC RULE 2.
> 2. An ImageView with ID `@+id/decorative_header` is missing a content description, violating STATIC RULE 2.
> 3. A TextView with ID `@+id/alert_banner` appears dynamically but does not receive accessibility focus, violating DYNAMIC RULE 1b.
>
> **Code Trace**:
>
> 1. In `ACTIVITY_SETTINGS.XML`, the ImageButton with ID `@+id/btn_back` (lines

21-26) is an icon-only button with no `android:contentDescription` attribute. Its OnClickListener in `SETTINGSACTIVITY.JAVA` (lines 24-29) calls finish(), confirming its purpose is to navigate back.

2. In `ACTIVITY_SETTINGS.XML`, the ImageView with ID `@+id/decorative_header` (lines 8-13) uses a source image named `bg_abstract_waves`, indicating it is purely for decoration and conveys no information.

3. In `ACTIVITY_SETTINGS.XML`, the TextView with ID `@+id/alert_banner` (lines 41-48) is initially hidden with android:visibility="gone". In `SETTINGSACTIVITY.JAVA`, the onResume() lifecycle method (lines 32-35) changes its visibility to View.VISIBLE if there are pending notifications.

Evidence:

1. For `@+id/btn_back`: The code shows an unlabelled ImageButton. This violates STATIC ACCESSIBILITY FIXING RULES (2. Intent and meaningful labels): "If an item is uninformative (like an icon-only button), it must be labeled."

2. For `@+id/decorative_header`: The code shows an ImageView used for background decoration. This falls under STATIC ACCESSIBILITY FIXING RULES (2. Intent and meaningful labels): "Purely decorative elements must be hidden from the screen reader. SOLUTIONS: XML: ...For purely decorative views, use `android:importantForAccessibility="no"`."

3. For `@+id/alert_banner`: The code shows a TextView that appears after the screen loads. This is an Appearing Content issue. The Interpreter classified it as a Major Window Change (1b), but the alert does not change the entire page context or require immediate interaction. It is an important notification within the existing screen. Therefore, it better fits the definition of DYNAMIC ACCESSIBILITY FIXING RULES (1a. Minor Window Changes): "this happens when the user has to be informed about something, but the workflow was not interrupted."

UX Assessment: The unlabeled back button is a critical barrier to navigation. The decorative header image adds auditory clutter if announced and should be hidden. The alert banner contains important information that the user must be made aware of. However, forcing focus to it (a Major Change) would be disorienting, as the user has just landed on the "Profile Settings" screen. A polite announcement (a Minor Change) is the superior user experience, as it informs the user of the alert without disrupting their navigation flow. The three proposed fixes are independent and will

not create conflicts or auditory clutter.

Severity: HIGH.

Suggestion:

1. For the ImageButton with ID `@+id/btn_back`, apply the `android:contentDescription="Back"` attribute in the `ACTIVITY_SETTINGS.XML` file to provide a clear and concise label, as per STATIC ACCESSIBILITY FIXING RULES (2. Intent and meaningful labels).
2. For the ImageView with ID `@+id/decorative_header`, apply the `android:importantForAccessibility="no"` attribute in the `ACTIVITY_SETTINGS.XML` file to hide it from the screen reader, as it is purely decorative, following STATIC ACCESSIBILITY FIXING RULES (2. Intent and meaningful labels).
3. For the TextView with ID `@+id/alert_banner`, apply the `android:accessibilityLiveRegion="polite"` attribute in the `ACTIVITY_SETTINGS.XML` file. This will ensure the screen reader announces the alert's text when it becomes visible without abruptly shifting focus, in accordance with DYNAMIC ACCESSIBILITY FIXING RULES (1a. Minor Window Changes).

The Critic Agent responds:

My role: I am an impartial compliance inspector tasked with critically assessing the Analyzer's output, ensuring absolute compliance with the accessibility fixing rules and zero tolerance for hallucinations.

Fact-Checking Trace: The Analyzer's trace is accurate. The ImageButton `@+id/btn_back` (lines 21-26), ImageView `@+id/decorative_header` (lines 8-13), and TextView `@+id/alert_banner` (lines 41-48) all exist in `ACTIVITY_SETTINGS.XML`. The `SETTINGSACTIVITY.JAVA` file confirms the OnClickListener for btn_back (lines 24-29) and the logic to make alert_banner visible in onResume (lines 32-35). VALID.

Rule Alignement: Flawless.

1. For `@+id/btn_back`, the Analyzer correctly applies STATIC RULE 2, suggesting `android:contentDescription` for an unlabeled button.
2. For `@+id/decorative_header`, the Analyzer correctly applies STATIC RULE 2, suggesting `android:importantForAccessibility="no"` for a decorative el-

ement.

3. For `@+id/alert_banner`, the Analyzer correctly re-classifies the issue as a DYNAMIC RULE 1a (Minor Window Change) and suggests the preferred XML solution, `android:accessibilityLiveRegion="polite"`, for a passively appearing element. All rules are quoted correctly, and the solutions are the exact ones prescribed by the rulebook.

Auditory Clutter Check: The Analyzer's suggestions actively reduce auditory clutter by hiding the decorative image and providing essential information without being disruptive (using a polite live region). The fixes are independent and do not create any new clutter. APPROVED.

Directive Precision: The suggestions are definitive and target specific component IDs (`@+id/btn_back`, `@+id/decorative_header`, `@+id/alert_banner`). There is no conditional delegation or vague language. PASS.

Devil's Advocate: The Analyzer overrules the Interpreter's classification of the alert banner as a "Major Window Change". What if the alert, "You have 3 unread messages!", is considered critical enough in this app's context to warrant immediate attention? By treating it as a minor change, the Analyzer might be downplaying its importance, causing the user to miss a crucial notification if they navigate away too quickly.

Rebuttal: The Devil's Advocate argument is not compelling. The text "You have 3 unread messages!" is informational, not a blocker. It does not prevent the user from interacting with the rest of the "Profile Settings" screen. Forcing focus (a Major Change solution) would be disorienting and interrupt the user's primary task of viewing their settings. The Analyzer's judgment to use a polite announcement provides the information without hijacking the user's context, which is the superior and correct user experience for this type of notification. The Analyzer's reasoning is sound.

Decision: APPROVED.

Actionable Feedback: None.

The Analyzer's work is approved by the Critic Agent, so the Admin forwards everything to the Repairer Agent:

> Repairer, the Analyzer's work has been approved. Read the source code provided by the Executor at the beginning of this session and apply the following fixes.

Repairer-Reviewer Phase

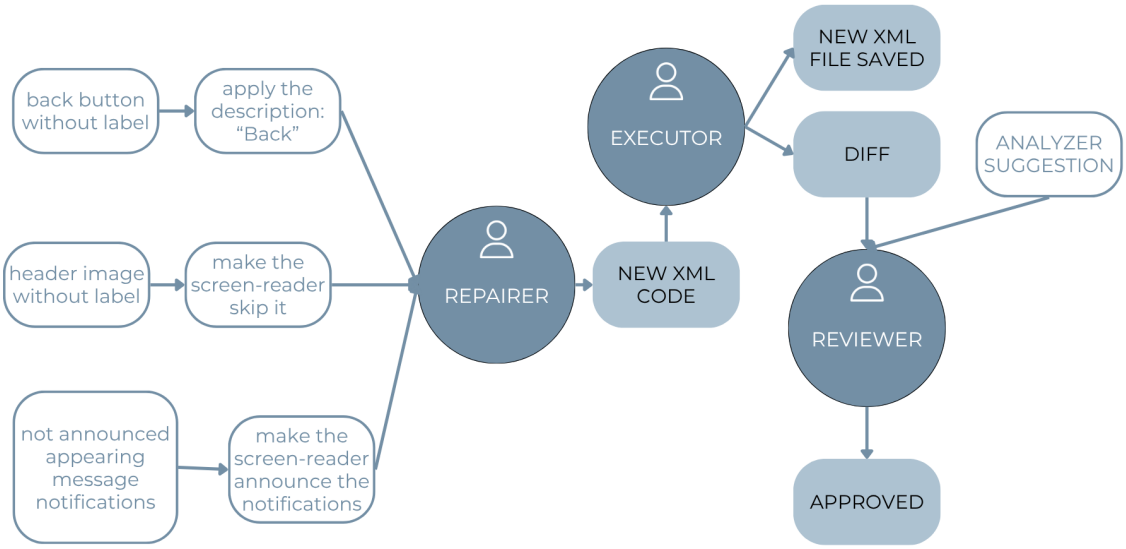


Figure 4.6: SM Repairer-Reviewer Phase

The Repairer Agent receives everything and outputs:

> **My role**: I am a mobile accessibility code repair specialist. My task is to act as a strict executor, implementing the exact fixes suggested by the Analyzer agent and outputting a Python script to save and validate the changes.
>
> **Repair Plan**: I will implement the three suggestions provided by the Analyzer Agent, all of which apply to the `ACTIVITY_SETTINGS.XML` file.
>
> 1. I will add `android:importantForAccessibility="no"` to the ImageView with the ID `@+id/decorative_header`.
> 2. I will add `android:contentDescription="Back"` to the ImageButton with the ID `@+id/btn_back`.
> 3. I will add `android:accessibilityLiveRegion="polite"` to the TextView with the ID `@+id/alert_banner`.
>
> I will not modify any other files or any other part of the `ACTIVITY_SETTINGS.XML` file. The final step will be to create a Python script to write, validate, and show the diff for the modified XML file.

Code:

```
import xml.etree.ElementTree as ET
import difflib

original_xml_content = """<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
↪  xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#FFFFFF">

    <ImageView
        android:id="@+id/decorative_header"
        android:layout_width="match_parent"
        android:layout_height="120dp"
        android:src="@drawable/bg_abstract_waves"
        android:scaleType="centerCrop"/>

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">

        <ImageButton
            android:id="@+id/btn_back"
            android:layout_width="48dp"
            android:layout_height="48dp"
            android:src="@drawable/ic_arrow_back"
            android:background="?attr/selectableItemBackgroundBor
            ↪  derless"
            android:layout_alignParentStart="true" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toEndOf="@id/btn_back"
            android:layout_centerVertical="true"
            android:layout_marginStart="16dp"
```

```
            android:text="Profile Settings"
            android:textSize="22sp"
            android:textStyle="bold"
            android:textColor="#000000" />
    </RelativeLayout>

    <TextView
        android:id="@+id/alert_banner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FFCDD2"
        android:textColor="#B71C1C"
        android:padding="12dp"
        android:text="You have 3 unread messages!"
        android:visibility="gone"
        android:textStyle="bold" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp"
        android:text="Account Details"
        android:textSize="18sp"
        android:textColor="#333333" />

</LinearLayout>
"""

fixed_xml_content = """<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
↪   xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#FFFFFF">

    <ImageView
        android:id="@+id/decorative_header"
        android:layout_width="match_parent"
        android:layout_height="120dp"
```

```
        android:src="@drawable/bg_abstract_waves"
        android:scaleType="centerCrop"
        android:importantForAccessibility="no"/>

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">

        <ImageButton
            android:id="@+id/btn_back"
            android:layout_width="48dp"
            android:layout_height="48dp"
            android:src="@drawable/ic_arrow_back"
            android:background="?attr/selectableItemBackgroundBor
            ↪  derless"
            android:layout_alignParentStart="true"
            android:contentDescription="Back" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toEndOf="@id/btn_back"
            android:layout_centerVertical="true"
            android:layout_marginStart="16dp"
            android:text="Profile Settings"
            android:textSize="22sp"
            android:textStyle="bold"
            android:textColor="#000000" />
    </RelativeLayout>

    <TextView
        android:id="@+id/alert_banner"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#FFCDD2"
        android:textColor="#B71C1C"
        android:padding="12dp"
        android:text="You have 3 unread messages!"
        android:visibility="gone"
```

```
            android:textStyle="bold"
            android:accessibilityLiveRegion="polite" />

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="16dp"
            android:text="Account Details"
            android:textSize="18sp"
            android:textColor="#333333" />

    </LinearLayout>
    """

    file_name = "ACTIVITY_SETTINGS_fixed.xml"
    with open(file_name, "w", encoding="utf-8") as f:
        f.write(fixed_xml_content)

    # Validation Routine
    try:
        ET.parse(file_name)
        print("XML Validation: SUCCESS.")
    except ET.ParseError as e:
        raise Exception(f"XML VALIDATION FAILED: {e}")

    # Diff Generation
    diff = difflib.unified_diff(
        original_xml_content.splitlines(keepends=True),
        fixed_xml_content.splitlines(keepends=True),
        fromfile='ACTIVITY_SETTINGS.XML',
        tofile='ACTIVITY_SETTINGS_fixed.xml',
    )
    print("\n--- DIFF ---")
    print("".join(diff))
```

Status: Completed

The Executor's output after the execution of the Python script is:

```
XML Validation: SUCCESS.
```

```
--- DIFF ---
--- ACTIVITY_SETTINGS.XML
+++ ACTIVITY_SETTINGS_fixed.xml
@@ -10,7 +10,8 @@
          android:layout_width="match_parent"
          android:layout_height="120dp"
          android:src="@drawable/bg_abstract_waves"
-         android:scaleType="centerCrop"/>
+         android:scaleType="centerCrop"
+         android:importantForAccessibility="no"/>

      <RelativeLayout
          android:layout_width="match_parent"
@@ -23,7 +24,8 @@
              android:layout_height="48dp"
              android:src="@drawable/ic_arrow_back"
              android:background="?attr/selectableItemBackgroundBorderless"
-             android:layout_alignParentStart="true" />
+             android:layout_alignParentStart="true"
+             android:contentDescription="Back" />

          <TextView
              android:layout_width="wrap_content"
@@ -46,7 +48,8 @@
          android:padding="12dp"
          android:text="You have 3 unread messages!"
          android:visibility="gone"
-         android:textStyle="bold" />
+         android:textStyle="bold"
+         android:accessibilityLiveRegion="polite" />

      <TextView
          android:layout_width="match_parent"
```

The Reviewer Agent's response is:

> **My role**: I am a strict code reviewer specialized in mobile accessibility. My task is to verify that the generated code is non-destructive, accurately targeted, and syntactically complete.
>
> **Executor Trace**: The Executor output states "XML Validation: SUCCESS.". The

Python script successfully wrote the new XML file and the syntax is valid.

Diff Trace & Integrity: PASS. Repairer added three new attributes to three different XML nodes as requested. No unrelated business logic, comments, or other nodes were touched. The changes are confined to the specified accessibility fixes.

Code Trace: Analyzer requested:

- Apply android:contentDescription="Back" to the ImageButton with ID `@+id/btn_back`.
- Apply android:importantForAccessibility="no" to the ImageView with ID `@+id/decorative_header`.
- Apply android:accessibilityLiveRegion="polite" to the TextView with ID `@+id/alert_banner`.

Repairer implemented:

- `@+id/decorative_header`: + android:importantForAccessibility="no"/>
- `@+id/btn_back`: + android:contentDescription="Back" />
- `@+id/alert_banner`: + android:accessibilityLiveRegion="polite" />

Match: YES. The Repairer has perfectly implemented all three requested changes in the correct XML file and on the correct nodes.

Flaw Hunt: I must find a potential discrepancy. The diff shows that for each change, a line was removed and two lines were added (e.g., - android:scaleType="centerCrop"/> was replaced by + android:scaleType="centerCrop" and + android:importantForAccessibility="no"/>). This could be interpreted as a more extensive change than just adding an attribute.

Rebuttal: The flaw I searched for is not a real issue. It is simply the standard representation of a diff when adding a new attribute to a self-closing XML tag. The original line containing the closing /> is removed, and two new lines are shown: one with the original attribute and a new one with the added attribute and the closing />. The logic and structure of the XML are perfectly preserved. The Repairer's work is clean, precise, and correct.

Decision: APPROVED.

Conclusion

The workflow is flawless, so the Admin Agent summarizes and terminates:

> The Reviewer has approved the Repairer's work. The accessibility issues have been successfully resolved.
>
> **Summary of the fix**: Three accessibility issues in the User Profile screen were addressed by modifying the `ACTIVITY_SETTINGS.XML` file.
>
> 1. The unlabeled back button (`@+id/btn_back`) was given a content description, "Back", making it clear to screen reader users.
> 2. The decorative header image (`@+id/decorative_header`) was hidden from accessibility services to eliminate auditory clutter.
> 3. The dynamic alert banner (`@+id/alert_banner`) was configured as a polite live region, ensuring that when it appears, its message is announced to the user without stealing focus.
>
> TERMINATE.

4.7. Use Case 2: Analyzer-Critic Feedback Loop

This use case shifts the focus to another use case to highlight the extended Analyzer-Critic Feedback loop.

The use case is denominated *DynamicTasks*, which involves a Flutter widget representing a simple task management interface. As shown in Fig. 4.7, the screen displays a status text (initially set to "Waiting") and a "Complete Task" button.

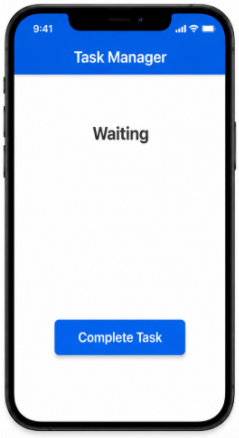


Figure 4.7: UC: DynamicTasks Mock-Up (Initial Screen)

Fig. 4.8 shows the dynamic UI changes after the user interacts with the button:

1. the status text dynamically changes to "Completed";
2. a new text "Task details revealed" appears on the screen;
3. a transient snackbar appears at the bottom with a "Confirmed" message and disappears after a few seconds.

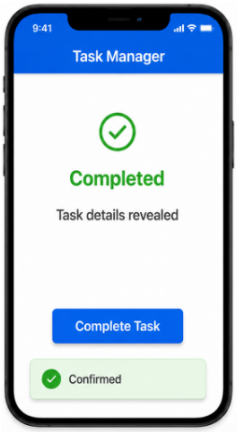


Figure 4.8: UC: DynamicTasks Mock-Up (Screen After Interaction)

- **The Problem**: a single user action triggers three distinct dynamic events. Announcing all of them would result in severe auditory clutter, creating a confusing and overwhelming experience for a screen reader user.
- **MAS Action**: the Analyzer and the Critic agents engage in a rigorous three-attempt iteration loop to negotiate a "Single-Announcement Strategy" and eliminate a persistent LLM hallucination (misquoting the Accessibility Rule Book).
- **The Result**: the system successfully formulates a strict, non-redundant accessibility plan that coordinates the three events, proving the MAS's ability to self-regulate, prioritize user experience, and strictly adhere to compliance guidelines.

Context: the Interpreter Phase

The Interpreter Agent accurately parses the accessibility log and maps the three events to the corresponding rules without proposing fixes:

- the status text modification is mapped to **Dynamic Rule 5** (Content Modification);
- the appearing details text is mapped to **Dynamic Rule 1b** (Major Window Changes);
- the transient `SnackBar` is mapped to **Dynamic Rule 3** (Short-Lived Content).

Iteration 1: Auditory Clutter and Misquotation

In its first iteration, the **Analyzer** attempts to address **all three issues independently**, proposing a live region for the status text, an explicit announcement for the `SnackBar`, and a programmatic focus shift for the details text.

The **Critic** intercepts this flawed logic and issues a **REJECTED** decision based on two critical failures:

- **Auditory Clutter Check (Rejected)**: the Critic's *Devil's Advocate* reasoning highlights that firing multiple announcements simultaneously will result in a "triple-read" sequence (e.g., "Completed", "Task details revealed", "Confirmed"). This violates the core UX requirement of preventing redundant speech.
- **Rule Alignment (Rejected)**: The Critic catches a direct hallucination. The Analyzer misquoted Dynamic Rule 3, using example text ("Item removed") that actually belongs to Rule 2 in the Rule Book.

The Critic demands a redo, instructing the Analyzer to formulate a definitive "single-announcement strategy" and to correct the quotation.

Iteration 2: Correct Strategy but Lingering Hallucinations

Prompted by the Admin, the **Analyzer** completely restructures its approach: it chooses to make the `SnackBar` the *only* spoken confirmation using `SemanticsService.announce`. It explicitly prohibits adding a live region to the status text to prevent double-reads, while maintaining the focus shift to the new details text to ensure they are discoverable.

The **Critic** evaluates this second attempt. The **Auditory Clutter Check** is now **APPROVED**, praising the definitive single-confirmation choice. However, the Critic issues another **REJECTED** decision. Despite the perfect architectural strategy, the Analyzer stubbornly retained the hallucinated "Item removed" quote. The Critic enforces strict compliance: the rule must be quoted exactly as written in the Rule Book.

Iteration 3: Flawless Alignment

In its final allowed attempt, the **Analyzer** maintains its robust **single-announcement strategy** and finally removes the incorrect quotation, replacing it with the exact wording from Dynamic Rule 3 for Flutter applications.

The **Critic** re-evaluates the output. With the strategy perfectly balancing the UX to avoid auditory clutter and the **Rule Alignment** completely cleared of hallucinations,

the Critic issues an **APPROVED** decision. The MAS successfully navigated a complex, multi-layered accessibility challenge, outputting a precise and coordinated repair plan ready for execution.

4.8. Use Case 3: Repairer-Reviewer Feedback Loop

In this section, we analyze another AI example created to address an auditory clutter issue. The primary objective is to illustrate the Repairer-Reviewer feedback loop, highlighting how the system ensures that accessibility patches are not only functionally correct but also syntactically sound and compliant with code quality standards.

Fig. 4.9 shows the mock-up of the application *Timer App* under consideration.

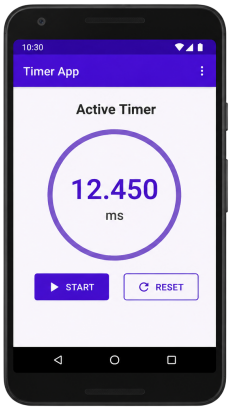


Figure 4.9: UC: Timer App Mock-Up

The screen involves an active timer whose internal state is updated every 10 milliseconds. This raw value is directly bound to a `Text` displayed on the screen.

- **The Problem**: the UI text updates 100 times per second. This high-frequency content modification overwhelms the TalkBack queue, creating severe dynamic auditory clutter and rendering the screen reader completely unusable.
- **MAS Action**: the Analyzer devises a strategy to decouple the internal timer from the displayed text. The Repairer implements this logic but introduces code-formatting technical debt. The Reviewer intercepts the validation failure and forces a correction.
- **The Result**: the Repairer successfully rewrites the patch, resolving the accessibility barrier while strictly adhering to Kotlin linting standards, proving the MAS's reliability as a safe automated developer.

Context: the Analyzer Phase

Before the Repairer intervenes, the Analyzer Agent establishes a strict remediation plan to address the auditory clutter:

- **State Decoupling**: keep the internal 10ms timer, but introduce a separate `displayMillisecon` state updated only once per second (1000ms). Bind the UI text exclusively to this new state.
- **Clutter Prevention**: explicitly prohibit the use of `LiveRegion` semantics or programmatic announcements to prevent continuous ticking sounds.

Iteration 1: Accessibility Success but Validation Failure

The Admin Agent forwards the approved plan to the **Repairer**, which **correctly** implements the **core logic**. It successfully introduces the 1-second cadence decoupling and refrains from adding any live regions.

However, the Executor Agent's pipeline (`ktlint`) flags two syntax formatting errors in the generated Kotlin file:

- a wildcard import (`import androidx.compose.runtime.*`), which violates Kotlin best practices;
- a function naming violation (the composable was named `TimerScreen` instead of the lint-required `timerScreen`).

The Reviewer Agent analyzes the Executor's trace and issues a **REJECTED** decision based on two critical factors:

- **Validation Trace (Failed)**: the Reviewer acts as a strict CI/CD pipeline gatekeeper. Even though the accessibility logic is flawless, the presence of syntax validation errors makes the code unfit for production.
- **Diff Integrity (Failed)**: the Reviewer notes that the Repairer failed to provide the mandatory git-style unified diff in its Python script. Without this diff, the Reviewer refuses to blindly approve the code, applying a strict "zero-trust" policy to ensure no unrelated business logic was altered.

Iteration 2: Code Quality Correction and Final Approval

Instructed by the Admin to resolve the code quality issues and supply the missing diff, the **Repairer** refines its output:

- **Syntax Fixes**: it replaces the wildcard import with explicit imports (`Composable`, `LaunchedEffect`, `mutableStateOf`, etc.) and renames the function to `timerScreen()` to satisfy the strict linter rules.
- **Verification Evidence**: it updates its Python script to correctly generate and print the unified diff alongside the newly formatted code.

The Executor runs the updated code and reports a successful validation (`Kotlin Validation: SUCCESS. No syntax errors.`).

The **Reviewer** performs its final evaluation on the new data:

- **Diff Trace Integrity (Pass)**. by analyzing the unified diff, the Reviewer mathematically confirms that the changes are surgically confined to the timer state, the imports, and the function name. No collateral damage or unrelated logic was introduced.
- **Code Trace (Pass)**: the Reviewer verifies that the decoupling logic exactly matches the Analyzer's blueprint (updating every 1000ms without `LiveRegion` semantics or programmatic announcements).

With code quality restored and the accessibility requirements perfectly met, the Reviewer issues an **APPROVED** decision. The MAS successfully patched a complex high-frequency dynamic issue while maintaining strict software engineering standards.

5 | Evaluation

This chapter focuses on the evaluation of the system for automated remediation of accessibility issues in Android mobile applications. It is necessary to analyze its effectiveness, reliability, and practical applicability. We aim to check the **correctness** of the generated code patches and the **quality** of the agents' reasoning and the robustness of the architecture.

5.1. Research Questions

This evaluation is guided by the following Research Questions (RQ):

Research Questions

RQ1 How **effective** is the proposed MAS in correctly interpreting accessibility logs, formulating valid remediation strategies, and generating syntactically correct code patches across different Android frameworks?

RQ2 To what extent does the multi-agent architecture, specifically, through the iterative validation of the Critic and the Reviewer Agents, **mitigate LLM hallucinations** and improve patch reliability compared to a single-LLM baseline?

RQ3 How do different foundational **models** (e.g., Gemini 2.5 PRO vs. GPT-5.2) impact the reasoning capabilities, code quality, and overall stability of the automated remediation workflow?

To answer these questions, the evaluation systematically assesses the performance of each agent composing the MAS throughout the entire remediation workflow. The analysis covers the Interpreter's input validation, the mapping of accessibility issues, the Analyzer's strategic suggestions, the Repairer's code patch generation, and the hallucination management performed by the validation agents.

5.2. Experimental Design

To thoroughly answer the established research questions, the evaluation was structured to test the system across progressively complex scenarios. This section details the test dataset, the language models and baselines employed, and the metrics used to evaluate the outcomes.

5.2.1. Dataset and Test Scenario

The evaluation was conducted using three **types of tests** designed to progressively increase the realism and complexity of the evaluation environment.

Artificial Tests

The artificial tests were generated ad hoc using AI. They consist of four accessibility remediation scenarios and three input-validation ones. The remediation scenarios are:

1. the ***SmartProfile*** use case illustrated in Sec. 4.6 written in XML and Java, with a human-readable log. It presents the following accessibility barriers: a button without a descriptor and a dynamic notification not announced. It also presents a **catch**: a decorative header with no description that is reported as an issue, even though it is, in fact, correct according to anti-auditory clutter suggestions;
2. the ***DynamicTask*** use case where the log is machine-generated, and the code is Dart/Flutter. It presents three different **dynamic issues**: appearing content (major window changes), short-lived content, and content modification;
3. the ***Timer*** use case that shows two types of **auditory clutter**: dynamic and structural. It is written in Jetpack Compose, and the log is machine-generated;
4. the ***NetworkBadge*** use case that exhibits a possible **rule collision**: a dynamically appearing text that does not have a specific live region, but its parent container does. The source code is in Jetpack Compose, and the accessibility log is machine-generated.

While the input-validation scenarios are:

- a Java **code** snippet **without** the corresponding accessibility **log**;
- a machine-generated **accessibility log without** the corresponding source **code**;
- a source code with a **random** accessibility log.

Semi-Artificial Tests

The semi-artificial tests consist of real accessibility logs, i.e., output of the tool ***TimeStump*** [25], and AI-generated source code (XML and Java), given that the tool was used on commercial applications. They comprise:

1. ***Spotify*** that presents two dynamic issues: a disappearing button and a short-lived snackbar;
2. ***Ticketmaster*** that contains a disappearing text button without a content description and marked as not important for accessibility. Note that adding a description would cause auditory clutter, but it is important to set it as important for accessibility;
3. ***Motivation*** that presents a major window change and a button without a label.

Real Use Cases

This kind of test takes as input **real applications** and the corresponding real accessibility log. In particular, the log is a comment written by Ana Ferreira as a GitHub issue [7–10] with the results of the tool developed by her and her team ***A11yArgus*** [11]. The source code is downloaded directly from the GitHub repository. In particular, we analyzed:

1. ***whoBIRD*** with missing images labels and redundant decorative descriptions [7];
2. ***Logline*** that contains redundant descriptions on decorative elements [8];
3. ***Suntime Widgets*** where several icons miss the descriptor [9];
4. ***Wispar*** some input labels are missing [10].

These applications were selected because they provide publicly available source code together with accessibility reports produced by accessibility experts.

5.2.2. LLM Models and Baseline Setup

To evaluate the impact of the underlying foundation models (RQ3), every test was executed twice with different state-of-the-art LLMs: **Gemini 2.5 PRO** and **GPT-5.2**.

Furthermore, to isolate the specific contribution of the multi-agent architecture and explicitly address RQ2, the proposed MAS is compared against a **Single-LLM Baseline**, inspired by the work of Aljedaani et al. [1]. This baseline was implemented as a Python script that sends the same inputs (source code and accessibility log) to a single model, prompting it to generate a code patch without any agent specialization, validation stages,

or inter-agent review. Both GPT-4o and GPT-5.2 were used for the baseline evaluation to ensure a comprehensive comparison.

5.2.3. Evaluation Metrics

The performance of the system and its individual agents (RQ1) was measured through a combination of quantitative and qualitative indicators tailored to each phase of the workflow:

1. **Input Validation**: measured through the number of true positives, true negatives, false positives, and false negatives produced by the Interpreter Agent.
2. **Accessibility Issue Interpretation**: measured through the number of correctly matched issues, hallucinated issues, incorrect matches, and ignored issues, comparing the provided accessibility log and the Interpreter's output.
3. **Suggestions Quality**: measured by evaluating the Analyzer's suggestion adherence to the Accessibility Rule Book, its auditory clutter management, and the correctness of the selected UI components.
4. **Code Patch Correctness**: measured through the compilation success of the Repairer's new code, its adherence to the Analyzer's suggestions, and the adoption of coding best practices.
5. **Hallucination Mitigation**: measured through the number of iterations required by the Critic and the Reviewer agents before approving an output.

Whenever quantitative metrics were insufficient, outputs were manually assessed by comparing them against the Accessibility Rule Book and Android accessibility guidelines.

5.3. Results

In this section, we analyze the test results. We focus on each evaluation goal by listing the data and, if useful, by presenting graphs.

5.3.1. Input Validation

Table 5.1 presents data about the input validation divided by test typology and LLM model. The acronyms used stand for:

- TP: True Positive, i.e., the output is VALID, and the decision is correct;

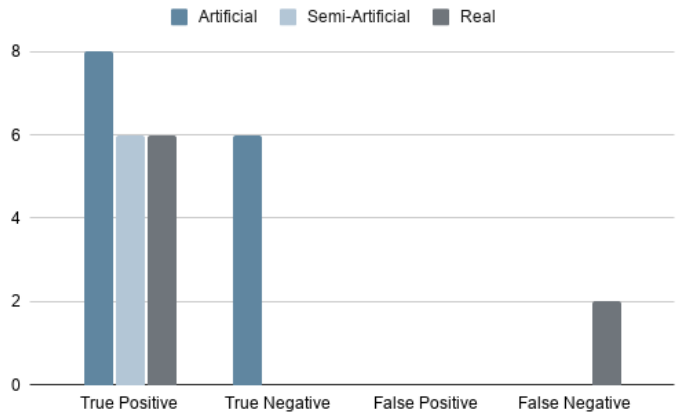
- TN: True Negative, i.e., the output is INVALID, and the decision is correct;
- FP: False Positive, i.e., the output is VALID, and the decision is incorrect;
- FN: False Negative, i.e., the output is INVALID, and the decision is incorrect;

Test Type	LLM Model	TP	TN	FP	FN
Artificial	Gemini 2.5 PRO	4	3	0	0
Artificial	GPT-5.2	4	3	0	0
Semi-Artificial	Gemini 2.5 PRO	3	0	0	0
Semi-Artificial	GPT-5.2	3	0	0	0
Real	Gemini 2.5 PRO	4	0	0	0
Real	GPT-5.2	2	0	0	2

Table 5.1: Input Validation Data

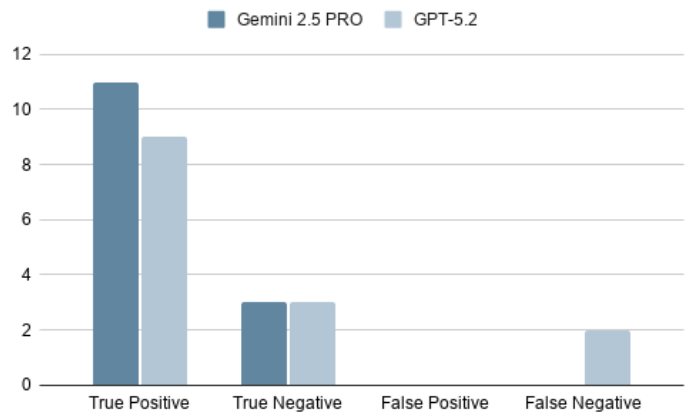
The results are overall good: the Interpreter Agent correctly classified the inputs **93%** of the time.

The chart in Graph 5.1 shows how the system struggles more with real applications as input.



Graph 5.1: Input Validation divided by Test Types Chart

However, it is useful to look at the results divided by the LLM model used, as shown in the chart in Graph 5.2.



Graph 5.2: Input Validation divided by LLM Models Chart

Gemini 2.5 PRO correctly classified the input in **100%** of cases, versus GPT-5.2, which achieved correctness only 86% of the time.

These results suggest that, when powered by Gemini 2.5 PRO, the Interpreter Agent can reliably distinguish valid and invalid inputs within the evaluated scenarios.

5.3.2. Accessibility Issues Interpretation

Table 5.2 shows how the Interpreter Agent classified the issues. Note that the Artificial Tests from 5 - 7 with an invalid input are not considered. This also applies to the Real Use Cases 2 and 4 that were considered invalid by GPT-5.2.

To better lay out the table, the following acronyms are used:

- CM: number of correct matches between the existing issue and the rule book;
- HM: number of hallucinated matches, i.e., the issue or the rule does not exist;
- IM: number of incorrect matches, i.e., both the issue and the rule exist, but the match is incorrect;
- IR: number of ignored issues.

The log types' acronyms are:

- HR: Human-Readable;
- MG: Machine-Generated.

Finally, the test types' acronyms are:

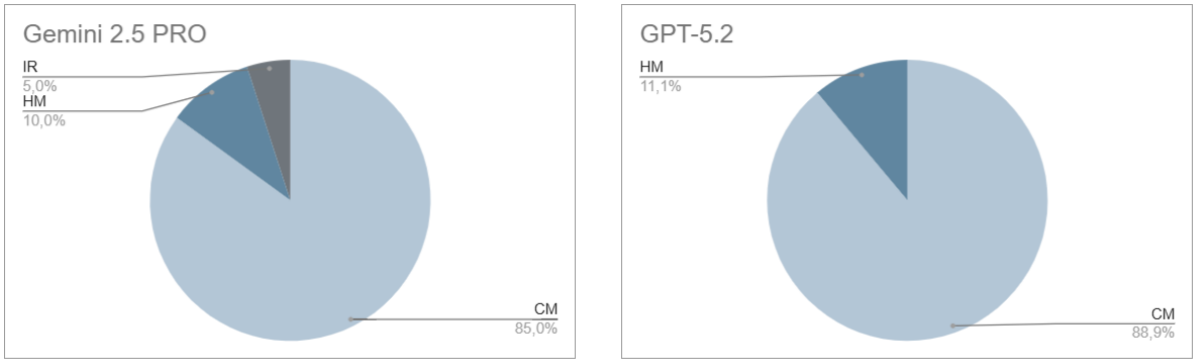
- A: Artificial test;
- SA: Semi-Artificial test;
- R: Real use case.

Log Type	Test Type	LLM Model	CM	HM	IM	IR
HR	A	Gemini 2.5 PRO	1	0	0	0
MG	A	Gemini 2.5 PRO	6	1	0	0
HR	A	GPT-5.2	1	0	0	0
MG	A	GPT-5.2	6	1	0	0
MG	SA	Gemini 2.5 PRO	5	1	0	1
MG	SA	GPT-5.2	6	1	0	0
HR	R	Gemini 2.5 PRO	5	0	0	0
HR	R	GPT-5.2	3	0	0	0

Table 5.2: Accessibility Issues Interpretation Data

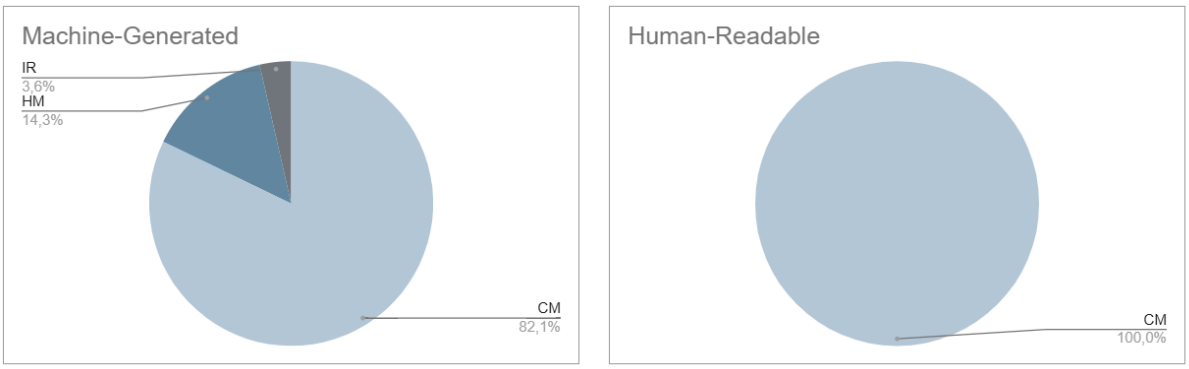
Across all evaluated issues, **97%** were **correctly identified and mapped** to the corresponding Accessibility Rule Book category. Four additional rule matches were hallucinated, corresponding to an **11% hallucination rate**, while only one issue was omitted, resulting in a **3% omission rate**.

To explain the one ignored issue, we can look at the pie charts in Graph 5.3, which display the difference between the results based on the **LLM model** used. The outcome is rather similar: the only difference in their approaches is in the semi-artificial test 2 (*Ticketmaster*): while GPT-5.2 identified correctly that the button is excluded from the screen reader access despite being a primary interactive control, Gemini 2.5 PRO directly omits the part regarding the button accessibility.



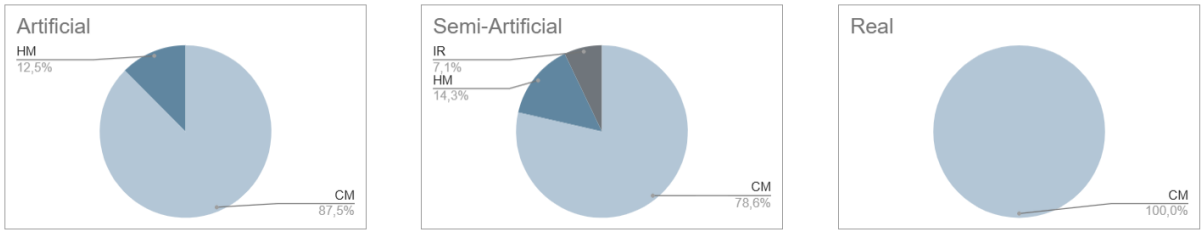
Graph 5.3: Pie charts comparing the accessibility log interpretation based on the LLM model

The pie charts in Graph 5.4 show the difference between **machine-generated** logs and **human-readable** ones. HR logs present a 100% rate of correct interpretation and matching without any hallucination, while the MG present a 14% rate of hallucinations.



Graph 5.4: Pie charts comparing the accessibility log interpretation based on the log type

Based on these findings, we can confidently state that the resulting data does not depend on the test typology.



Graph 5.5: Pie charts comparing the accessibility log interpretation based on the test type

By looking at the pie charts in Graph 5.5, one could assume that the real use cases had great results compared to artificial and semi-artificial tests. However, as we found out previously, the results depend mainly on the type of log, and the real use cases are all based on human-readable logs, which proved to be the most understandable. While the artificial tests are a mix of MG and HR logs, the semi-artificial tests are entirely composed of MG logs. Furthermore, analyzing the case of the omitted rule, we understood that it was caused by the model used and, consequently, not by the type of test.

5.3.3. Suggestions Quality

In Table 5.3, we focus on the quality of the Analyzer's suggestions. As in the previous section, the Artificial Tests from 5 - 7 with an invalid input and the Real Use Cases 2 and 4, which were considered invalid by GPT-5.2, are not considered. The analysis considers the **match** between the issue found and the Role Book **resolution method**, the **auditory clutter** management, and the **component** chosen. In particular:

- CM: number of correct matches with the Rule Book resolution methods;
- WM: number of incorrect matches with the Rule Book resolution methods;
- CNA: number of issues that were correctly not addressed;
- INA: number of issues that were incorrectly not addressed;
- ACA: number of correct suggestions to avoid auditory clutter;
- ACN: number of auditory clutter issues not resolved;
- ACP: number of auditory clutter produced;
- CN: number of correctly identified components;
- WN: number of incorrectly identified components.

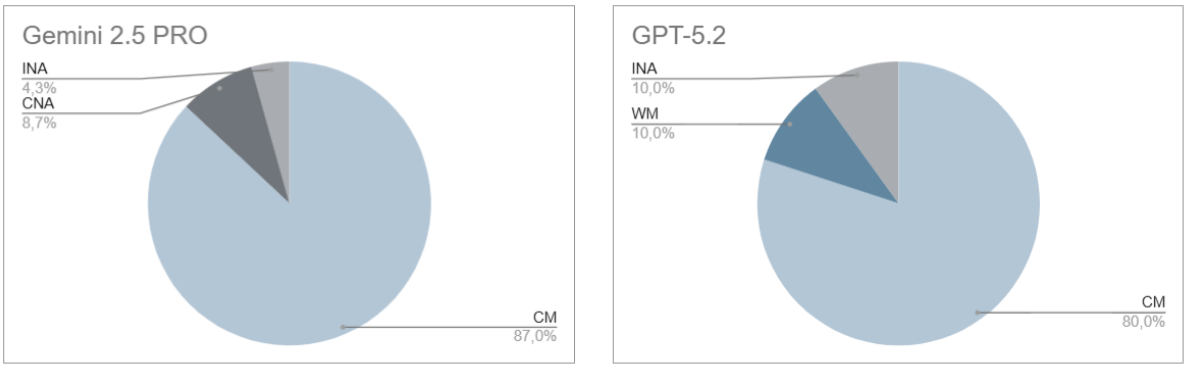
Test Type	LLM Model	CM	WM	CNA	INA	ACA	ACN	ACP	CN	WN
A	Gemini 2.5 PRO	8	0	1	0	3	0	0	7	0
A	GPT-5.2	6	2	0	0	2	1	0	6	1
SA	Gemini 2.5 PRO	6	0	1	1	1	0	0	6	0
SA	GPT-5.2	7	0	0	1	0	0	0	7	0
R	Gemini 2.5 PRO	6	0	0	0	3	0	0	6	0
R	GPT-5.2	3	0	0	1	2	0	0	3	0

Table 5.3: Suggestion Quality Data

The results show an **88% correct matching rate**, demonstrating that the Analyzer Agent can decide when it is useful to address the issues. Furthermore, it selects the **correct component 35 out of 36** times. It never creates **auditory clutter** on its own, and when it is already present, the agent addresses it **11 out of 12** times.

LLM Models Comparison

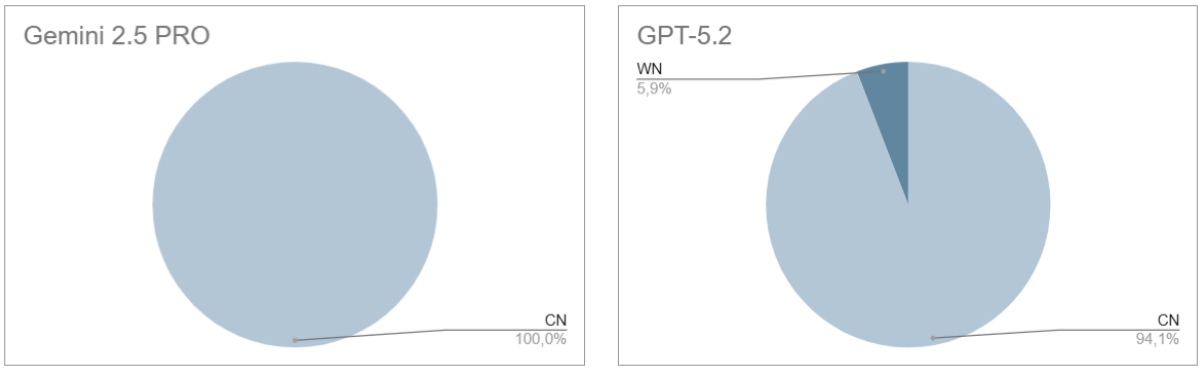
The pie charts in Graph 5.6 show differences in the LLM models' approaches to resolution model matches.



Graph 5.6: Pie charts comparing the resolution method match based on the LLM model

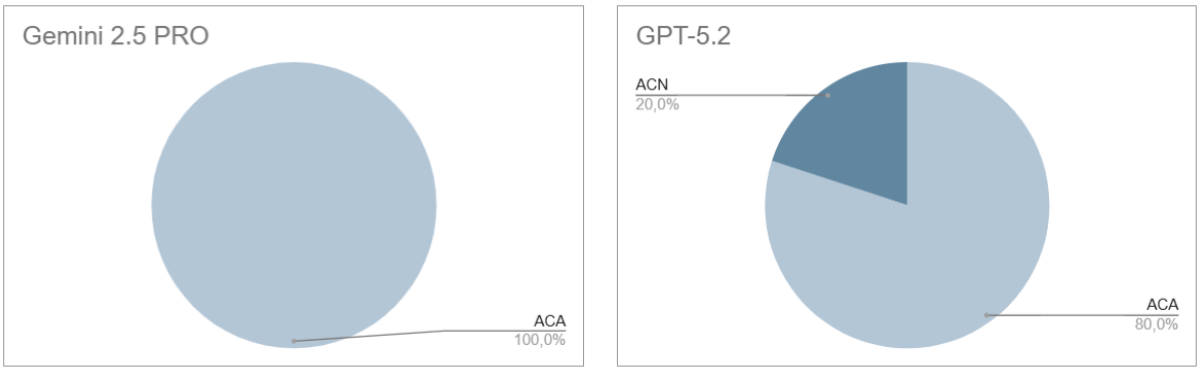
Gemini 2.5 PRO is definitely more capable of matching the issues to the correct resolution method with a **97%** correct matching rate (obtained with the sum of the correct matches and the correctly not addressed issues). GPT-5.2 not only has a lower matching rate (80%), but it has a higher percentage of incorrectly ignored issues (10%) and mismatched two issues.

Google's model is also clearly superior in choosing the correct component (100% correct component choice rate), as shown in the graphs in Graph 5.7.



Graph 5.7: Pie charts comparing the component choice based on the LLM model

In addition to the better results achieved, the measures adopted to avoid auditory clutter (100% auditory clutter addressing rate) are also presented in Graph 5.8.

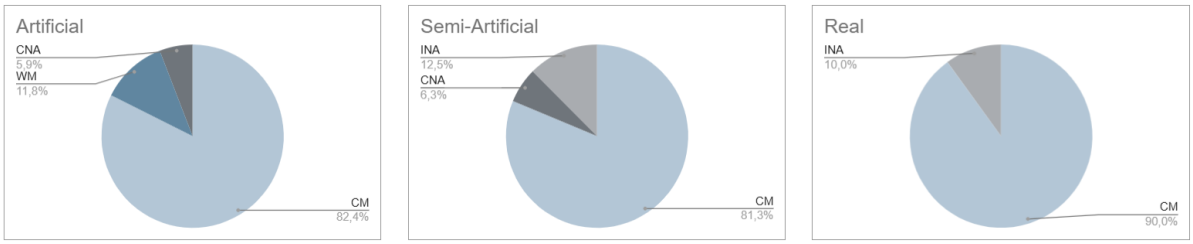


Graph 5.8: Pie charts comparing the auditory clutter management based on the LLM model

Test Typology Comparison

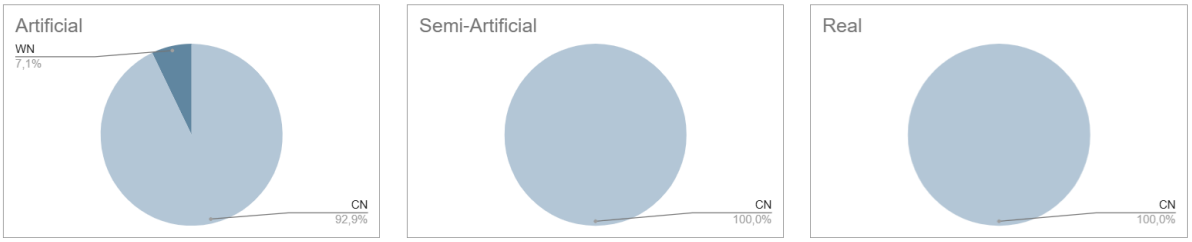
The results regarding the type of tests demonstrate that the Analyzer Agent makes better suggestions for real and semi-real situations. It has the most difficulties in the specifically arranged artificial tests, since we created the tests with specific challenging situations.

The pie charts in Graph 5.9 show that the correct matching rate is around 90% for every test typology. The artificial matched the rules incorrectly 2 times, while the semi-artificial and the real ones have a 10% of omitted issues.

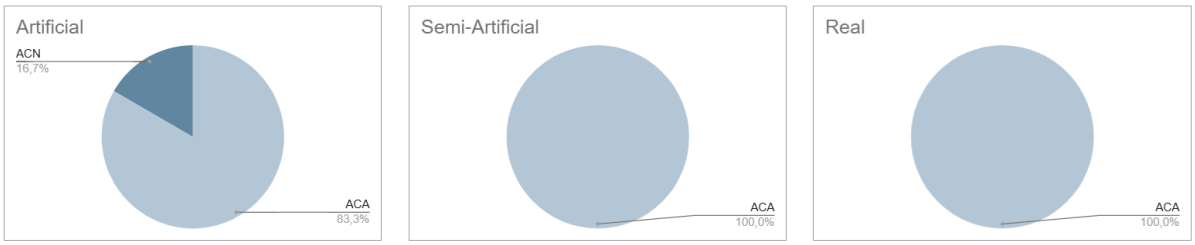


Graph 5.9: Pie charts comparing the resolution method match based on the test type

The component choice rate is shown in Graph 5.10 while the auditory clutter management is shown in Graph 5.11. The real and semi-real cases' suggestions were flawless.



Graph 5.10: Pie charts comparing the component choice based on the test type



Graph 5.11: Pie charts comparing the auditory clutter management based on the test type

Representative Case Studies

To better understand the quantitative results, we discuss three representative cases that highlight the Analyzer Agent's strengths and limitations.

In the ***Spotify*** case (semi-artificial test 1) executed with **Gemini 2.5 PRO**, the Interpreter Agent reported three dynamic accessibility issues: a disappearing button (dynamic rule 2), a short-lived snack bar (dynamic rule 3), and a minor window change (hallucinated issue). First of all, the Analyzer immediately classified the first two issues, i.e., the

real ones, as the primary ones, excluding the third one from the analysis. Furthermore, during the examination of the source code to find evidence of the issues, the Agent found out that the snack bar contains **another problem**: redundant labeling. It assesses:

> The ImageView and TextView within the snackbar convey the exact same information, which would create **auditory clutter** ('Added to Liked Songs, Added to Liked Songs') if announced.

The most interesting part is that it discovers **another accessibility issue** while listing the suggestions:

> To ensure the snackbar's contents ('Added to Liked Songs' and 'Change') are read as a single, logical unit if the user focuses on it, they must be grouped. According to STATIC ACCESSIBILITY FIXING RULES (1. Element grouping and presentation order), the solution is to "apply `android:focusable="true"` and `android:screenReaderFocusable="true"` to the parent layout". Therefore, add `android:focusable="true"` and `android:screenReaderFocusable="true"` to the `LinearLayout` with ID `custom_snackbar`.

The **only unmet direction** is keeping the "change" button inside the short-lived content. This violates the first directive of the resolution method. The model **GPT-5.2** addressed it by stating:

> Resolve the short-lived clickable action as a UX/product restructuring (no extra technical mechanism beyond the rulebook): Because DYNAMIC RULE 3 warns to "avoid including clickable elements in a short-lived manner," remove the time-limited interaction pattern by not placing the actionable "Change" control inside the transient `@+id/custom_snackbar`. Provide the "Change" action as a persistent control elsewhere in the UI so blind users can reliably find and activate it without a countdown.

Another interesting case is the ***Ticketmaster*** one (semi-artificial test 2). The accessibility issue is associated with a "Continue" button that not only disappears without notice when touched, but it also lacks a content description and, more importantly, is marked as not important for accessibility. When using the **Gemini** model, the Interpreter Agent only reports the disappearing issue, so the Analyzer Agent decides **not to address it**:

> Do not implement a fix for this issue. The disappearance of the "Continue" button is a direct result of a user-initiated action. Adding an announcement via `announceForAccessibility` or a live region would create redundant auditory clutter, as the user already expects a screen change after pressing "Continue". The

> best user experience is to allow the subsequent action or screen transition to occur without an explicit announcement about the button's removal.

In this case, the **GPT** approach is more complete. Given that the **Interpreter** Agent reported **all the actual issues**, the Analyzer Agent was able to address the matter entirely. Here follows its UX assessment:

- The primary blocker is that `@+id/btn_continue` is a **key interactive control** but is explicitly hidden from screen readers (`importantForAccessibility="no"`). This can prevent blind users from completing the flow.
- When the user activates "Continue" and it disappears, a single short confirmation announcement can prevent confusion about why the control vanished. This should be **one announcement only** (no live region on the disappearing node), to avoid auditory clutter.
- For the Rule Book's redundant labeling warning, adding a custom `contentDescription` to a button already labeled "Continue" would likely create an **echo effect**; it should be avoided.

This example demonstrates how the overall quality of the remediation process strongly depends on the Interpreter's ability to extract all relevant accessibility issues. Missing information at this stage propagates throughout the entire workflow and may prevent the system from addressing critical accessibility barriers.

Lastly, the ***whoBIRD*** test case (real use case 1) is worthy of a deep analysis. This app recognizes bird species based on their sounds. The accessibility barrier resides in the lack of bird **image descriptors**. Given the scope of this application, the description needs to be really useful: it must specify the specific kind of bird in the picture, so it must be set **dynamically**. The **GPT** model does **not realize it**, while Gemini does. Here is the Analyzer suggestion when executed with **Google**'s model:

> In the `BirdInfoActivity.kt` file, within the `onItemClick` listener method, programmatically set the `contentDescription` for the informational views. When a bird is selected, retrieve its name and set the `contentDescription` on `binding.webview` to "**Media of [Bird Name]**" and on `binding.icon` to "No media available for [Bird Name]". This ensures the description is **dynamic** and **accurate**.

This case is particularly challenging because the accessibility fix requires **semantic understanding** of the application domain. The correct description cannot be statically assigned and must be generated dynamically based on the selected bird species. This

highlights the advantage of LLM-based reasoning compared to purely rule-based remediation systems.

5.3.4. Code Patch Correctness

In Table 5.4, we analyze the Repairer Agent's output. The focus is on the correct **compilation** of the produced files, the number of **suggestions addressed**, and the **quality** of the patch. In particular:

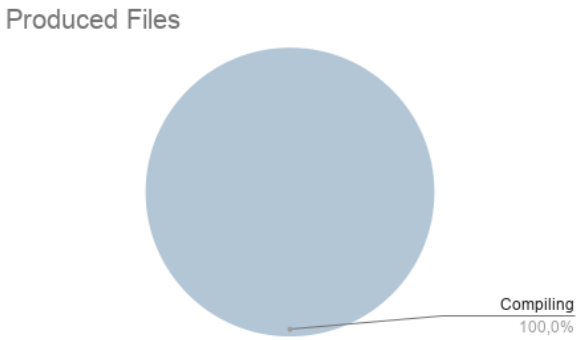
- CC: number of files that compile correctly;
- NC: number of files that do not compile;
- AA: number of Analyzer's suggestions addressed;
- AN: number of Analyzer's suggestions not addressed;
- HM: number of hallucinated modifications, i.e., not connected to the suggestion;
- BP: number of issues for which the best practice was adopted;
- RM: number of issues for which the best practice was not adopted, putting the app at risk of malfunctioning, but still addressed.
- WA: number of issues wrongly addressed, causing problems for users when using the application.

Test Type	P. Language	LLM Model	CC	NC	AA	AN	HM	BP	RM	WA
A	XML+Java	Gemini 2.5 PRO	1	0	3	0	0	3	0	0
A	Kotlin	Gemini 2.5 PRO	2	0	3	0	0	3	0	0
A	Flutter	Gemini 2.5 PRO	1	0	3	0	0	3	0	0
A	XML+Java	GPT-5.2	1	0	3	0	0	3	0	0
A	Kotlin	GPT-5.2	1	0	1	0	0	1	0	0
A	Flutter	GPT-5.2	1	0	3	0	0	3	0	0
SA	XML+Java	Gemini 2.5 PRO	3	0	4	0	0	3	1	0
SA	XML+Java	GPT-5.2	3	0	6	0	0	3	1	2
R	XML+Java	Gemini 2.5 PRO	4	0	4	0	0	4	0	0
R	Kotlin	Gemini 2.5 PRO	1	0	1	0	0	1	0	0
R	Flutter	Gemini 2.5 PRO	1	0	1	0	0	1	0	0
R	XML+Java	GPT-5.2	3	0	3	0	0	3	0	0

Table 5.4: Code Patch Correctness Data

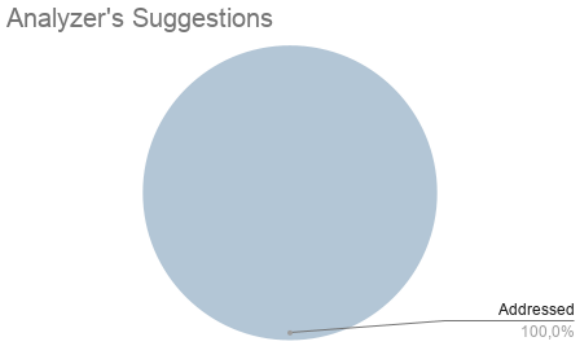
The Repairer Agent achieved perfect functional correctness across all evaluated scenarios. All generated patches compiled successfully and implemented every suggestion produced by the Analyzer Agent.

In particular, as shown in the pie chart in Graph 5.12, **100%** of the produced files **compile correctly**.



Graph 5.12: Pie chart representing the percentage of compiled produced files

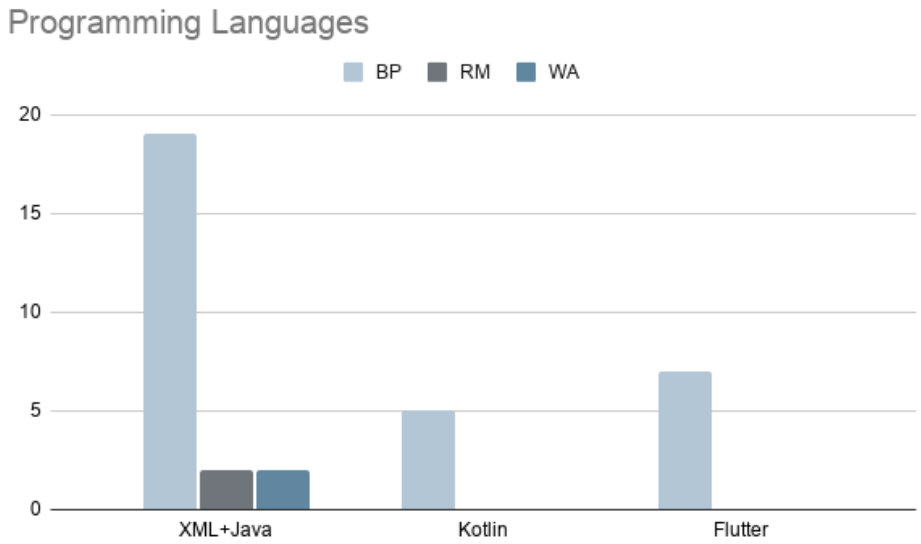
More importantly, as presented in Graph 5.13, **100%** of the Analyzer Agent's **suggestions** were addressed by the Repairer Agent. However, addressing a suggestion does not necessarily imply that the adopted implementation represents the optimal solution, as discussed in the following analysis.



Graph 5.13: Pie chart representing the percentage of addressed Analyzer's suggestions by the Repairer Agent

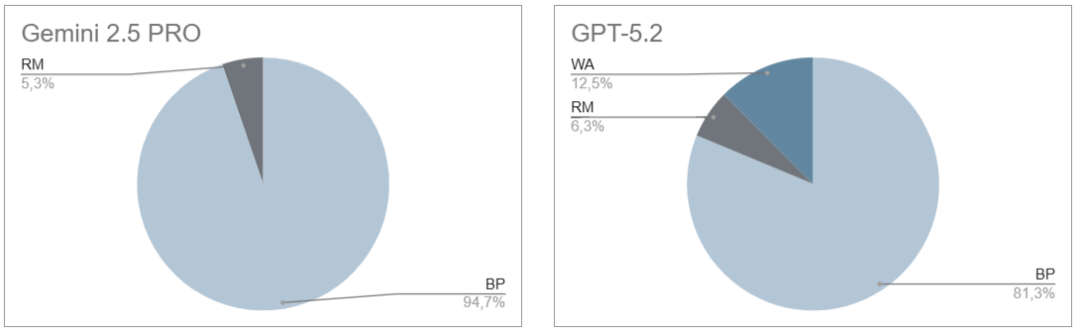
The only minor flaws regard the adopted practices used to resolve the issues. We look into this aspect, dividing the examination into the model used, the test typology, and the programming language. Note that still **89%** of the practices used can be considered the **best** ones.

The chart in Graph 5.14 shows the number divided by **programming language** used. Both the risky and the wrong practices appear only in the **XML** applications. However, the number of analyzed applications that reached this point of the remediation process is predominantly XML ones. In fact, the percentage of cases where the best practice was used is 83%.



Graph 5.14: Pie charts comparing the practices used based on the programming language

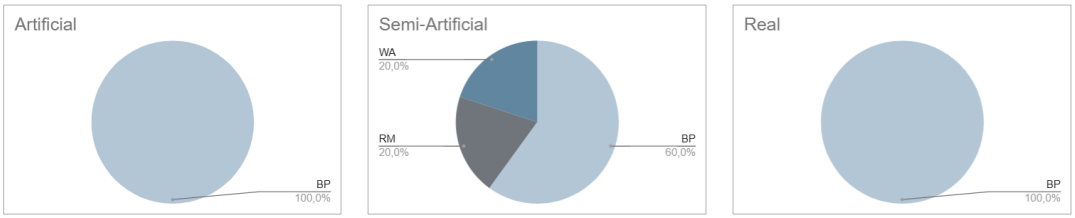
As emerged before, the **GPT** model gives slightly **worse** results. Also in this case, as shown in Graph 5.15, it achieved an 81% of **best-practice** solutions versus the **95%** accomplished by **Gemini**. Furthermore, the errors made in the approaches happened only with GPT-5.2.



Graph 5.15: Pie charts comparing the practices used based on the LLM model

Graph 5.16 compares the pie charts representing the different results obtained depending

on the test typology used.



Graph 5.16: Pie charts comparing the practices used based on the test type

It is interesting to note that all risky and incorrect practices were used in the **semi-artificial** tests. Given that they are a combination of a real machine-generated accessibility log and AI-generated code, it makes sense that addressing the issues in the best way possible becomes more difficult.

Comparison with a Single-LLM Baseline

Given the importance of the generated code patches, here follows a comparison between the proposed MAS and a single-LLM baseline inspired by the work of Aljedaani et al. [1].

We started by using the **GPT-4o** model to reproduce accurately the work of Aljedaani et al. [1]. The evaluation uses a similar metric to the one adopted for the MAS:

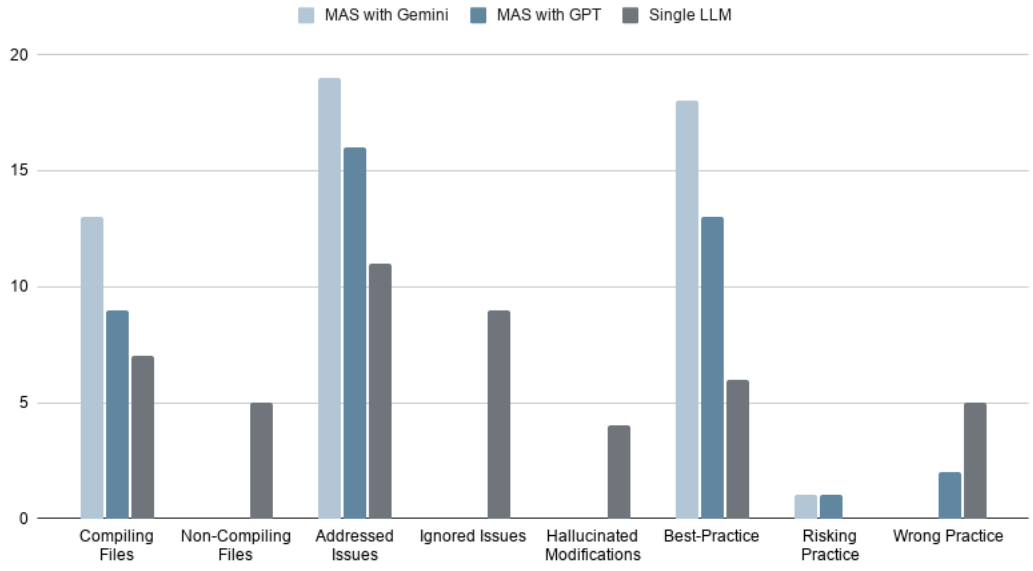
- CC: number of files that compile correctly;
- NC: number of files that do not compile;
- AA: number of issues correctly addressed;
- AN: number of ignored issues;
- HM: number of hallucinated modifications, i.e., not connected to the suggestion;
- BP: number of issues for which the best practice was adopted;
- RM: number of issues for which the best practice was not adopted, putting the app at risk of malfunctioning, but still addressed.
- WA: number of issues wrongly addressed, causing problems for users when using the application.

The resulting data is listed in Table 5.5.

Test Type	P. Language	CC	NC	AA	AN	HM	BP	RM	WA
A	XML+Java	1	1	3	0	0	2	0	1
A	Kotlin	1	1	2	1	0	1	0	1
A	Flutter	1	0	2	1	0	1	0	1
SA	XML+Java	3	0	1	4	3	0	0	1
R	XML+Java	0	2	2	2	1	1	0	1
R	Kotlin	1	0	1	0	0	1	0	0
R	Flutter	0	1	0	1	0	0	0	0

Table 5.5: Code Patch Correctness Data Single-LLM Baseline (GPT-4o)

The charts in Graph 5.17 compare the single LLM approach's data in Table 5.5 and the MAS approach in Table 5.4



Graph 5.17: Charts comparing the produced code patches between the MAS and the single LLM

The bar charts show that with the single-LLM GPT-4o remediation process:

- about **42%** of the produced files do **not compile**;
- **45%** of the presented accessibility **barriers** are **ignored**;
- there are 4 **hallucinated modifications**;

- about 45% of the addressed issues are modified in a wrong manner, leaving the total percentage of **correctly addressed issues to 30%**.

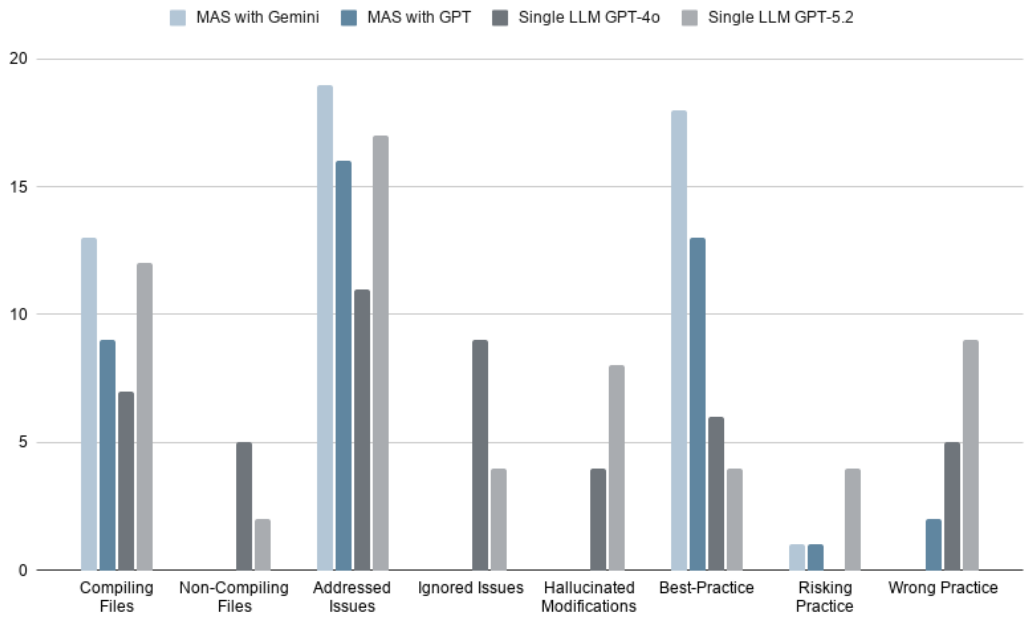
In particular, in one case, the model lists the modifications to do, but it applies them to a generic example rather than to the provided code.

For completeness, we replaced the GPT-4o model with the GPT-5.2 one to have a more meaningful comparison. The collected data is in Table 5.6.

Test Type	P. Language	CC	NC	AA	AN	HM	BP	RM	WA
A	XML+Java	2	0	3	0	0	1	1	1
A	Kotlin	1	1	3	0	2	0	2	1
A	Flutter	0	1	3	0	3	0	1	2
SA	XML+Java	4	0	3	3	3	2	0	1
R	XML+Java	3	0	3	1	0	0	0	3
R	Kotlin	1	0	1	0	0	1	0	0
R	Flutter	1	0	1	0	0	0	0	1

Table 5.6: Code Patch Correctness Data Baseline GPT-5.2

The results are quite interesting. To have better insights, the bar charts in Graph 5.18 compare the single-LLM baselines with the proposed MAS.



Graph 5.18: Charts comparing the produced code patches between the MAS and the two single LLMs

This model produces better results than the previous one, however, there is still a clear advantage of the multi-agent approach. In particular, the GPT-5.2 model:

- improves the compilation success, but it still produces **non-compiling files** (2);
- increases the number of addressed issues, however, it still presents **ignored issues** (19%), and it produces the **highest** number of **hallucinated** modifications (8);
- generates multiple fixes that do not follow the recommended accessibility best practices and often introduce unnecessary changes that may **increase auditory clutter**, specifically **64%** of the time, the **wrong practice** was adopted.

The goal of this comparison is to **isolate the contribution of the multi-agent architecture**. This is the reason why the baseline uses the same LLM technology while removing specialization, validation stages, and inter-agent review.

The comparison also highlights an important **trade-off**. A **single-LLM** solution is simpler to deploy, generally executes **faster**, and incurs **lower API costs** because it requires only one model invocation per remediation task.

5.3.5. Hallucination Mitigation

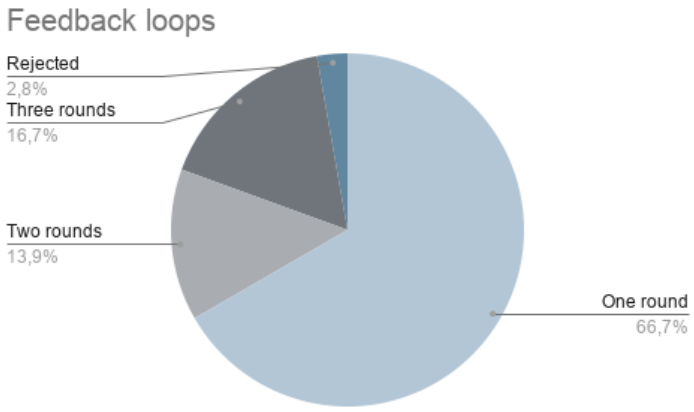
To analyze the hallucination mitigation, we examine the work of the **Critic** and the **Reviewer** Agents. Table 5.7 contains the data about the feedback loop of both agents. In particular:

- C1: number of analyses accepted by the Critic Agent on the first try;
- C2: number of analyses accepted by the Critic Agent on the second try;
- C3: number of analyses accepted by the Critic Agent on the third try;
- CR: number of analyses rejected by the Critic Agent;
- R1: number of repairs accepted by the Reviewer Agent on the first try;
- R2: number of repairs accepted by the Reviewer Agent on the second try;
- R3: number of repairs accepted by the Reviewer Agent on the third try;
- RR: number of repairs rejected by the Reviewer Agent.

Test Type	LLM Model	C1	C2	C3	CR	R1	R2	R3	RR
A	Gemini 2.5 PRO	3	1	0	0	4	0	0	0
A	GPT-5.2	0	2	1	1	1	1	0	1
SA	Gemini 2.5 PRO	3	0	0	0	2	0	0	0
SA	GPT-5.2	0	2	1	0	3	0	0	0
R	Gemini 2.5 PRO	4	0	0	0	3	0	0	0
R	GPT-5.2	0	0	2	0	1	0	1	0

Table 5.7: Feedback Loop Iteration Data

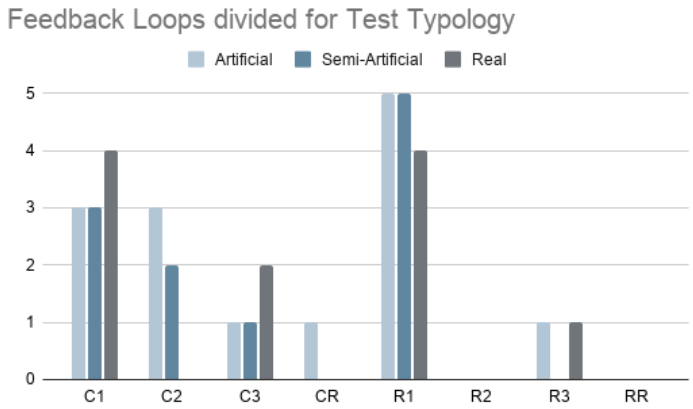
We start by reviewing the feedback loop management: data is represented in the pie chart in Graph 5.19.



Graph 5.19: Pie chart representing the percentage of redone work during the remediation process

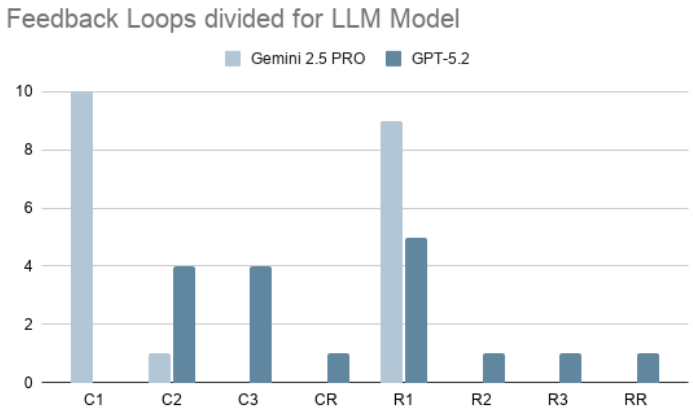
First and foremost, the **rejection rate** is minimal, remaining below **3%**. This indicates that the Critic and the Reviewer maintain sufficiently rigorous standards to reject sub-optimal work, ensuring that the remediation process successfully achieves a valid result in over 97% of cases. Furthermore, in the majority of cases (**67%**), the Analyzer and Repairer jobs are accepted on the first try, demonstrating that the system works well **without interruptions**. Moreover, approximately **30%** of the tasks required at least one additional **iteration** before approval, highlighting the practical contribution of the Critic and Reviewer agents in refining intermediate outputs.

We proceed by analyzing the data, dividing it by test typology and by the LLM model used. The charts in Graph 5.20 represent the distribution of data by test typology: the process is quite **similar in each type**. The interesting aspect concerns the differences between the work done by the two agents. The **Critic** is more likely to **reject** work: this is reasonable given that the Analyzer's job is much more delicate and complex than the Repairer's.



Graph 5.20: Chart representing the percentage of redone work during the remediation process, divided by test type

To examine the results based on the LLM model used, we created the graph in Graph 5.21, which shows that Gemini 2.5 PRO accepts analysis and repair on the first try in 95% of cases. While the Critic Agent with GPT-5.2 never accepts the Analyzer's work on the first try.



Graph 5.21: Chart representing the percentage of redone work during the remediation process, divided by the LLM model used

We analyzed the entire process in depth, and every Critic's rejection was well-grounded, while only once the Reviewer rejected the Repairer's work because of its own hallucination: it misunderstood the diff-file three times in a row (GPT). Anyway, the Reviewer's work is generally impeccable: it is always capable of differentiating between pre-existing technical debt and injected errors (by the Repairer).

Despite the role constraints defined in the system prompt, the Analyzer Agent attempted to perform part of the Reviewer's responsibilities in one instance with Gemini's model. This observation confirms that role drifting remains possible even in carefully designed multi-agent systems and motivates the inclusion of dedicated validation agents.

5.4. Discussion

The evaluation demonstrated that the proposed solution is capable of addressing accessibility issues across multiple Android frameworks and testing scenarios. However, the interpretation of the results extends beyond the reported metrics. This section conducts a deeper analysis to understand the implications of these findings and to identify the strengths and weaknesses of the proposed system.

5.4.1. Effectiveness of the Multi-Agent Approach

RQ1

*How **effective** is the proposed MAS in correctly interpreting accessibility logs, formulating valid remediation strategies, and generating syntactically correct code patches across different Android frameworks?*
The proposed multi-agent architecture is **highly effective** in automating accessibility remediation across **different Android frameworks**. Decomposing the workflow into specialized tasks allows the system to maintain reasoning **quality** and implementation **correctness** without being tied to a single framework.

In particular, the **Interpreter** Agent **successfully validated** and analyzed the provided inputs in the vast majority of cases, while the **Analyzer** Agent generated remediation **strategies** that were generally **consistent** with the Accessibility Rule Book and Android accessibility guidelines. One of the most relevant findings concerns the **Repairer** Agent. All generated **patches compiled** successfully and implemented every accepted Analyzer's suggestions.

Therefore, **decomposing** the remediation workflow into specialized tasks allows the system to maintain both **reasoning quality** and **implementation correctness**. Rather than relying on a single model to perform the entire remediation process, the separation of responsibilities appears to reduce the complexity of each task and improve the reliability of the overall workflow.

The evaluation also shows that the system can operate across **different Android devel-**

opment frameworks, including XML-based applications, Jetpack Compose, and Flutter. Although the number of evaluated applications remains limited, the results indicate that the proposed architecture is not tightly coupled to a specific framework and can generalize across different development environments.

5.4.2. Hallucination Mitigation and Architectural Value

RQ2

To what extent does the multi-agent architecture, specifically, through the iterative validation of the Critic and the Reviewer Agents, ***mitigate LLM hallucinations*** *and improve patch reliability compared to a single-LLM baseline?*
The **iterative validation stages** are crucial for the **reliability** of the process, significantly outperforming a single-LLM approach. Validation agents actively **reduce** the impact of **hallucinations** and reasoning inconsistencies before the final output generation.

The collected data highlights the importance of the Critic and Reviewer agents. Approximately one-third of the analyzed tasks required at least one additional iteration before reaching an approved state. This suggests that the **validation stages** are actively contributing to the **reliability** of the process. Several incorrect analyses and implementation flaws were detected before reaching the final output, **reducing** the impact of **hallucinations** and reasoning **inconsistencies**.

Interestingly, the **Critic** Agent **rejected** analyses **more frequently** than the Reviewer rejected code patches. This behavior is consistent with the nature of the two tasks. Accessibility analysis requires semantic reasoning, interpretation of accessibility guidelines, and user-experience considerations, making it inherently more susceptible to hallucinations and incomplete reasoning. In contrast, code patch generation is more constrained by the Analyzer's instructions and therefore presents fewer opportunities for significant deviations.

5.4.3. LLM Variability

RQ3 - LLM Variability

*How do different foundational **models** (e.g., Gemini 2.5 PRO vs. GPT-5.2) impact the reasoning capabilities, code quality, and overall stability of the automated remediation workflow?*

The effectiveness of the proposed architecture is partially **dependent** on the capabilities of the underlying **language model**. While both models successfully completed the remediation flow, their approaches yielded distinct stability profiles.

Gemini 2.5 PRO generally produced more **stable** outputs. It achieved a higher first-pass approval rate and generated fewer suboptimal solutions. On the other hand, **GPT-5.2** occasionally demonstrated a deeper **understanding** of specific accessibility scenarios, e.g., the *Ticketmaster* case.

Consequently, improvements in foundation models are likely to benefit the performance of the remediation workflow.

5.4.4. Remaining Challenges and Threats to Validity

Despite the encouraging results, the evaluation highlighted several challenges that limit the current generalization of the approach.

First, the quality of the remediation process is strongly dependent on the **quality of the accessibility log**. Missing or incomplete information can propagate through the workflow and prevent the system from addressing critical issues.

Furthermore, some instances of **role drifting** and **reasoning inconsistencies** were observed, confirming that multi-agent systems inherit several limitations of the underlying LLM.

5.4.5. Conclusion

Overall, the evaluation suggests that the proposed MAS constitutes a **promising approach** for automated accessibility remediation. The combination of specialized agents, structured reasoning, and iterative validation enables the generation of reliable accessibility fixes while reducing common LLM-related issues. Although several **limitations** remain, the obtained results indicate that multi-agent architectures can **effectively support developers** in addressing **accessibility barriers** and represent a valuable direction

for future research.

6 | Conclusion

6.1. Summary of Contributions and Addressed RQs

This thesis proposed and evaluated a novel **LLM-based Multi-Agent System** (MAS) designed to automate the **remediation** of **accessibility barriers** for blind users in Android **applications**. The primary objective was to investigate how specialized LLM agents could collaboratively resolve complex accessibility issues while mitigating the inherent limitations of foundation models.

By extensively evaluating the architecture across various scenarios, this research successfully addressed three core research questions:

RQ1 *How **effective** is the proposed MAS in correctly interpreting accessibility logs, formulating valid remediation strategies, and generating syntactically correct code patches across different Android technologies?*

The study demonstrated that the proposed MAS is **highly effective**. By decomposing the remediation workflow, the system **reliably** interpreted accessibility logs, matched them with the official Accessibility Rule Book [Sec. 4.2.1], and generated **100% syntactically correct code** patches across diverse Android frameworks (XML, Jetpack Compose, and Flutter).

RQ2 *To what extent does the multi-agent architecture, specifically, through the iterative validation of the Critic and the Reviewer Agents, **mitigate LLM hallucinations** and improve patch reliability compared to a single-LLM baseline?*

The evaluation proved that a multi-agent architecture significantly **outperforms** a single-LLM baseline. The introduction of dedicated validation agents (the Critic [Sec. 4.3.3] and the Reviewer [Sec. 4.3.5]) successfully intercepted reasoning inconsistencies and role-drifting, keeping the final rejection rate below 3% and ensuring structurally sound, **ready-to-deploy code**.

RQ3 *How do different foundational **models** (e.g., Gemini 2.5 PRO vs. GPT-5.2) impact the reasoning capabilities, code quality, and overall stability of the automated*

remediation workflow?

The comparative analysis between Gemini 2.5 PRO and GPT-5.2 highlighted that while the **architecture is model-agnostic**, the **stability** of the workflow **depends** on the underlying **foundation model**. Gemini demonstrated higher first-pass approval rates, whereas GPT showed slightly deeper contextual understanding in complex dynamic scenarios, proving the flexibility of the MAS to adapt to future LLM advancements.

6.2. Advancements over the State of the Art

To fully contextualize the value of the proposed MAS, it is crucial to outline how it advances the current state of the art in automated accessibility remediation. While previous works have laid the foundation of LLM-based software repair, this thesis introduces three major structural advancements:

1. **Holistic Reasoning**: prior LLM remediation attempts, such as single-LLM approaches [1] or *FixAlly* [24], treat accessibility barriers as isolated, single-component defects. Our MAS introduces a paradigm shift by implementing an **Analyzer Agent** [Sec. 4.3.2] capable of holistic reasoning. As demonstrated in the *Spotify* and *Ticketmaster* case studies [Sec. 5.3.3], the system evaluates the surrounding UI context to **avoid** introducing secondary issues such as **Auditory Clutter** [Sec. 2.4.1], a nuance that previous automated tools completely ignore.

2. **Self-Validating Architecture**: the most significant limitation of current LLM-based repair tools is their vulnerability to hallucinations, often resulting in syntactically incorrect or logically flawed code. Our approach advances the state of the art by embedding a multi-layered **validation loop** natively within the workflow. The introduction of the **Critic and Reviewer Agents** acts as a safety net, ensuring that every proposed strategy strictly adheres to the Accessibility Rule Book and every generated patch compiles correctly. This elevates the output from "plausible fixes" (as defined by previous literature [24][1]) to structurally sound, **ready-to-deploy code**.

3. **Android-Native Remediation**: while the limited existing remediation frameworks have primarily targeted the iOS ecosystem or general web guidelines, this work specifically tackles the fragmented and complex Android environment. The proposed system successfully navigates the intricacies of **different Android technologies** (XML, Jetpack Compose, Flutter) and native TalkBack behaviors, filling

a massive platform-specific void in the current literature.

Ultimately, this thesis demonstrates that the **future of automated accessibility remediation** does not rely on simply using a more powerful language model, but rather on orchestrating multiple models into a **structured**, **validated**, and **context-aware multi-agent workflow**.

6.3. Limitations and Future Work

While the proposed MAS demonstrates a robust and modular approach to automated accessibility remediation, it presents some limitations that pave the way for future research:

- **Dependency on the Input Quality**: the system's effectiveness is strictly tied to the accuracy of the provided accessibility log. Future iterations could integrate an autonomous multimodal crawler directly within the MAS, allowing the system to actively explore the application and verify accessibility barriers in real-time without relying entirely on external static logs.
- **Computational Costs and Latency**: the reliance on multiple interacting agents inherently increases execution time and API costs compared to single-LLM approaches. Future research should explore model distillation or the integration of smaller, locally hosted models for simpler tasks (e.g., the Interpreter Agent [Sec. 4.3.1]), reserving heavier models like Gemini 2.5 PRO exclusively for complex reasoning.
- **Scope Expansion**: currently, the system is purposely constrained to Android technologies and focuses exclusively on accessibility issues for blind users via TalkBack. A natural future extension would be adapting the MAS to cross-platform environments (e.g., iOS/VoiceOver) and expanding the Accessibility Rule Book to include motor and cognitive impairments.

Bibliography

[1] W. Aljedaani, A. Aljohani, M. M. Eler, A. Habib, and H. Do. Llms for accessibility in mobile apps: Detection and repair. In *Proceedings of the 22nd International Web for All Conference (W4A '25)*, pages 1–12, New York, NY, USA, 2025. ACM. ISBN 9798400718823. doi: 10.1145/3744257.3744270. URL `https://doi.org/10.1145/3744257.3744270`.

[2] ArcTouch and Fable. The state of mobile app accessibility report, 2025. URL `https://arctouch.com/state-of-mobile-app-accessibility`.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. doi: 10.48550/arXiv.2107.03374.

[4] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu. Accessible or not? an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 48(10):3954–3974, 2022. doi: 10.1109/TSE.2021.3108162.

[5] E. W. Dijkstra. On the role of scientific thought. 1974.

[6] Ericsson. Ericsson mobility report, 2025. URL `https://www.ericsson.com/en/reports-and-papers/mobility-report/key-figures#:~:text=8%2C510-,8%2C660,-9%2C430`.

[7] A. Ferreira. Accessibility: Missing labels for bird identification and redundant decorative description, 2026. URL `https://github.com/woheller69/whoBIRD/issues/125`. GitHub Issue #125.

[8] A. Ferreira. Accessibility: Redundant descriptions on decorative elements and critical ui fixes, 2026. URL `https://github.com/Patch4Code/Logline/issues/40`. GitHub issue #40.

[9] A. Ferreira. Accessibility: Missing descriptions on astronomical icons and focus order

fixes, 2026. URL https://github.com/forrestguice/SuntimesWidget/issues/915#event-22615488070. GitHub issue #915.

[10] A. Ferreira. Accessibility: Missing input labels and focus order navigation fixes, 2026. URL https://github.com/Scriptbash/Wispar/issues/358. GitHub issue #358.

[11] A. Ferreira, M. Ribeiro, B. Miranda, R. Gheyi, I. Machado, and B. Fonseca. A11yargus: Automated detection and empirical analysis of accessibility issues in android apps. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software (CHASE'26)*, Rio de Janeiro, Brazil, 2026. ACM. ISBN 979-8-4007-2484-8. doi: 10.1145/3794860.3794906.

[12] Google. Make apps more accessible, 2024. URL https://developer.android.com/guide/topics/ui/accessibility/apps?_gl=1*1jwrcd4*_ga*MjA1NDY4NzE0Mi4xNzc3NjMwODE0*_ga_QPQ2NRV856*czE3Nzc2MzA4MTMkbzEkZzEkdDE3Nzc2MzA4NDIkajMxJGwwJGgw.

[13] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, H. Liu, S. Wang, K. Zhang, Y. Wang, W. Gao, L. Ni, and J. Guo. A survey on llm-as-a-judge. *arXiv preprint arXiv:2411.15594*, 2025. URL https://arxiv.org/abs/2411.15594.

[14] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024. URL https://arxiv.org/abs/2402.01680.

[15] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2024. doi: 10.48550/arXiv.2308.10620.

[16] S. F. Huq, M. Tafreshipour, K. Kalcevich, and S. Malek. Automated generation of accessibility test reports from recorded user transcripts. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE 2025)*, pages 204–216, Ottawa, ON, Canada, 2025. IEEE. ISBN 979-8-3315-0569-1. doi: 10.1109/ICSE55347.2025.00043.

[17] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, D. Chen, W. Dai, H. S. Chan, A. Madotto, and P. Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 2022.

[18] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024. doi: 10.48550/arXiv.2406.00515.

[19] Y. Li, Z. Yang, Y. Guo, and X. Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017. ISBN 978-1-5386-1589-8. doi: 10.1109/ICSE-C.2017.8.

[20] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts, 2023.

[21] D. Maldonado, E. Cruz, J. A. Torres, P. J. Cruz, and S. del Pilar Gamboa Benitez. Multi-agent systems: A survey about its components, framework and workflow. *IEEE Access*, 12:80950–80979, 2024. doi: 10.1109/ACCESS.2024.3409051.

[22] L. Malmqvist. Sycophancy in large language models: Causes and mitigations. *arXiv preprint arXiv:2411.15287*, 2024. URL `https://arxiv.org/abs/2411.15287`.

[23] F. Mehralian, N. Salehnamadi, S. F. Huq, and S. Malek. Too much accessibility is harmful! automated detection and analysis of overly accessible elements in mobile apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*, Rochester, MI, USA, 2022. ACM. ISBN 978-1-4503-9475-8. doi: 10.1145/3551349.3560424.

[24] F. Mehralian, T. Barik, J. Nichols, and A. Swearngin. Automated code fix suggestions for accessibility issues in mobile apps. *arXiv preprint arXiv:2408.03827*, 2024.

[25] F. Mehralian, Z. He, and S. Malek. Automated accessibility analysis of dynamic content changes on mobile apps. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2689–2700. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00039.

[26] Microsoft. .net documentation, 2024. URL `https://learn.microsoft.com/en-us/dotnet/api/android.views.view.accessibilityliveregion?view=net-android-35.0`.

[27] L. S. Pereira, M. Matos, and C. Duarte. Exploring mobile device accessibility: Challenges, insights, and recommendations for evaluation methodologies. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*, Honolulu, HI, USA, 2024. ACM. ISBN 979-8-4007-0330-0. doi: 10.1145/3613904.3642526.

[28] M. W. U. Rahman, R. Nevarez, L. T. Mim, and S. Hariri. Multi-agent actor-critic generative ai for query resolution and analysis, 2025.

[29] N. Salehnamadi, Z. He, and S. Malek. Assistive-technology aided manual accessibility testing in mobile apps, powered by record-and-replay. In *Proceedings of the 2023 CHI*

Conference on Human Factors in Computing Systems (CHI '23), Hamburg, Germany, 2023. ACM. ISBN 978-1-4503-9421-5. doi: 10.1145/3544548.3580679.

[30] S. Ugare and S. Chandra. Agentic code reasoning, 2026.

[31] F. Wang, H. Cui, L. Yang, C. S. Lee, Z. Li, and C. Wen. Detecting and repairing role drift in multi-agent collaboration with lightweight protocols. *Preprints.org*, 2026. doi: 10.20944/preprints202603.0348.v1. URL `https://doi.org/10.20944/preprints202603.0348.v1`. Preprint, not peer reviewed.

[32] Wikipedia contributors. Screen reader, 2026. URL `https://en.wikipedia.org/wiki/Screen_reader`.

[33] World Wide Web Consortium (W3C). Success criterion 1.1.1: Non-text content, 2018. URL `https://www.w3.org/WAI/WCAG22/Understanding/non-text-content`.

[34] World Wide Web Consortium (W3C). Success criterion 1.3.1: Info and relationships, 2018. URL `https://www.w3.org/WAI/WCAG22/Understanding/info-and-relationships`.

[35] World Wide Web Consortium (W3C). Success criterion 1.3.2: Meaningful sequence, 2018. URL `https://www.w3.org/WAI/WCAG22/Understanding/meaningful-sequence`.

[36] World Wide Web Consortium (W3C). Success criterion 2.4.6: Headings and labels, 2018. URL `https://www.w3.org/WAI/WCAG22/Understanding/headings-and-labels`.

[37] WorldHealthOrganization. Disability, 2025. URL `https://www.who.int/health-topics/disability#tab=tab_1`.

[38] Q. Zhang, C. Gao, Y. Han, Y. Shang, C. Fang, Z. Chen, and L. Xiao. Sgagent: Suggestion-guided llm-based multi-agent framework for repository-level software repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2024. doi: 10.1145/xxxxxxx.

[39] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2026. doi: 10.48550/arXiv.2303.18223.

[40] M. Zhong, R. Chen, X. Chen, J. Fogarty, and J. O. Wobbrock. Screenaudit: Detecting

screen reader accessibility errors in mobile apps using large language models. *arXiv preprint arXiv:2504.02110*, 2025. doi: 10.48550/arXiv.2504.02110.

List of Figures

List of Tables

List of Graphs

Acknowledgments

Desidero innanzitutto ringraziare il mio relatore, il Professor Luciano Baresi, non solo per la costante disponibilità ma soprattutto per la fiducia riposta in me sin dal primo giorno. I suoi feedback diretti e puntuali sono stati fondamentali per indirizzare al meglio questa tesi. Lavorare sotto la sua supervisione è stato un piacere, oltre che un'incredibile occasione di crescita.

A special thanks goes to Ana Ferreira and Professor Marcio Ribeiro for their genuine interest in our work. I deeply appreciate the time they dedicated to our call and for sharing the insights on their work, which allowed me to better analyze and leverage the public data generated by their tool.

Furthermore, I would like to express my gratitude to Professor Sam Malek and his team, specifically Forough Mehralian and Ziyao He, for their crucial support. Sharing the data generated by their tool provided an essential contribution to the development and evaluation of this work.