POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

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

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Irene Lo Presti

Advisor: Prof. Luciano Baresi

Co-advisor:

Academic year: 2025-26

1. Introduction

As of 2026, the use of mobile applications continues to grow. Global mobile subscriptions are over 8.9 billion, and are expected to surpass 9.4 billion by 2030 [1]. This growth is driven by the increasing reliance on mobile applications in everyday life, from digital banking to healthcare access to social interaction. 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 full digital participation and limits independence. Digital accessibility is not a monolithic concept, as different disabilities require distinct strategies and technical interventions. Among them, blind users face unique challenges because interaction relies almost exclusively on audio feedback. While research has mainly focused on accessibility issue detection, remediation remains largely unexplored. Studies have shown that in 91% of Android applications with detected accessibility barriers, the problems remained unaddressed even after multiple version updates [4]. Recent efforts such as *FixAlly* [8] have demonstrated the potential of multi-agent LLM architectures for accessibility remediation. However, they focus on individual accessibility violations in iOS applications and do not explicitly address holistic reasoning across multiple issues, Android-specific remediation, or self-validating repair workflows. To bridge this gap, this thesis proposes an LLM-based Multi-Agent System (MAS) that automates the remediation of mobile accessibility barriers. The system takes as input Android source code and accessibility logs and produces validated code patches together with a remediation report. The architecture is grounded in the software-engineering principle of *Separation of Concerns*, decomposing the remediation process into specialized agents that collaboratively implement the *Locate-Suggest-Fix* paradigm. It advances the current state of the art by three structural advancements: holistic reasoning, iterative self-validation (to mitigate hallucinations), and Android-native remediation. Experimental results show that the architecture consistently generated syntactically valid code patches and outperformed single-LLM baselines in terms of reliability and remediation quality.

The remainder of this paper is organized as follows. Section 2 introduces the specific accessibility challenges for blind users. Section 3

provides background on LLMs and LLM-based Multi-Agent Systems. Section 4 details the proposed multi-agent architecture and the remediation workflow. Section 5 presents the evaluation of the system, including a comparison with a single-LLM baseline. Section 6 presents a review of the work related to this thesis. Finally, Section 7 discusses conclusions and related work.

2. Accessibility for Blind Users

The accessibility barriers faced by blind users present unique challenges. While sighted users rely on visual information such as icons, layouts, and animations, blind users access the same information through assistive technologies (AT). The primary assistive technology used by blind users is the screen reader (e.g., Android Talkback), a software application that renders text and images as speech. Consequently, 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, without a textual description, a screen reader can only announce it as an unlabeled element, preventing blind users from understanding its purpose. To support the remediation process, we organized the accessibility barriers a blind user can encounter into the *Accessibility Rule Book* (ARB), which was used as basic knowledge for some of the MAS agents. In particular, we divided the issues into Static and Dynamic. Static accessibility issues concern elements visible when the screen loads, such as missing labels or unclear relationships between interface components. Dynamic accessibility issues arise when information changes after the screen has already been loaded. They are categorized based on the Mehralian et al. research [9] into elements that appear, disappear, or both, components that move on the screen, and changes of attributes of static elements. It is also highlighted that providing excessive or repetitive information is equally detrimental. 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. In the ARB, for each identified category, we described how the issues can be addressed in each of the considered Android frameworks: XML/Java, Jetpack Compose, and Flutter.

3. LLMs and LLM-based MAS

As discussed in the previous section, addressing the accessibility barriers faced by blind users requires an understanding of the surrounding context. Large Language Models (LLMs) emerge as promising solutions for addressing these accessibility challenges. However, they remain affected by limitations such as sycophancy, hallucinations, difficulties handling long contexts, and role drifting. LLM-based MASs address these limitations by decomposing complex tasks into specialized agents that collaborate through structured interactions. They offer various techniques to overcome single-LLM limitations, and they have demonstrated promising results in solving various tasks, such as software development [6], as they allow *Separation of Concerns*.

4. Proposed Solution

We developed 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. 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 agents are designed as a professional development team, each of them with a specific role to allow *Separation of Concerns* based on the *Locate-Suggest-Fix* paradigm, with the addition of specific agents to mitigate hallucinations. The designed agents are:

- *Interpreter* for input validation and accessibility log translation by categorizing the issues as explained in the ARB;
- *Analyzer* for assessing the severity of each issue on a four-level scale (from *None* to *High*) and producing remediation strategies, i.e., specific suggestions to fix the issues based on the ARB resolution methods;
- *Critic* to ensure the Analyzer's reliability by verifying that the analysis is grounded

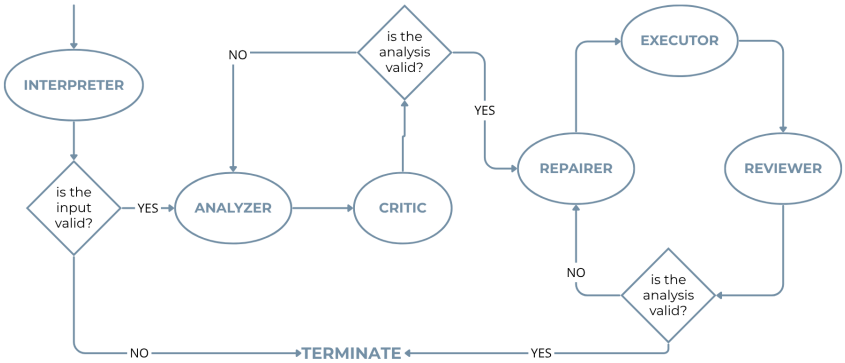

Figure 1: The Remediation Process Workflow

in the input and the suggestions correspond to the ones in the ARB;

- *Repairer* to implement the exact fixes suggested by the Analyzer by writing a Python script that saves the fixed code into a new file and then verifies the syntax correctness. It also has to create a Diff Code (git-style) between the original file and the new one.
- *Reviewer*, to ensure the Repairer's reliability by making sure that the Analyzer's suggestion was addressed and that the code patches were correct.
- *Admin* to manage the other agents and monitor the feedback loops iteration, 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.

4.1. Remediation Process Workflow

The flowchart in Fig. 1 represents the decision process of the Admin Agent and, consequently, the system's execution order. Note that, in addition to the designed agents, there is the Executor Agent, i.e., a *User Proxy Agent* responsible for initiating interaction and handling code execution.

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 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.

4.2. Used Technologies

The system was developed through a Python script that employs the open-source Microsoft AutoGen framework. Each agent was designed through a structured *System Prompt*. For the majority of the agents, a targeted Few-Shot Prompting technique was adopted to achieve more precise results. Along with the Semi-Formal Reasoning technique, a structured prompting methodology that requires the LLM to construct explicit premises, trace execution paths, and derive formal conclusions [10]. Furthermore, to address LLM's limitations, various state-of-the-art techniques were employed. To avoid role drifting, the agents were required to declare their role and commitments at each turn before performing their tasks. This was taken from the framework *RoleFix* by Wang et al. [11]. As previously mentioned, to reduce hallu-

cinations, we created the two specialized agents Critic and Reviewer. We employed the concept of *LLM-as-a-Judge*, where the LLM is tasked with determining whether something falls within the scope of a given rule. The challenges associated with long contexts are partially mitigated through the adoption of the Separation of Concerns principle, whereby each agent is provided only with the information relevant to its specific task. Furthermore, previous studies have shown that model performance is often higher when relevant information is placed at the beginning or the end of the input context [7]. Accordingly, key constraints were appended to the end of the prompts provided to the Analyzer and Repairer agents. Lastly, the Python script generated by the Repairer also produces a Git-style diff, allowing the Reviewer to focus its analysis solely on the modifications introduced. The last technique used regards sycophancy, which manifests not only with users' input but also when agents communicate with each other. In this case, it is important to ensure that the Critic and Reviewer agents do not automatically accept the Analyzer and Repairer outputs. Therefore, these two agents have the field *Devil's Advocate* in their structured output schema. There, they have to formulate at least one reason why the produced output might be wrong, hallucinated, incomplete, or may result in a poor user experience. This step is then evaluated, and if it is found true, the final decision is to reject the work of the other agents.

5. Evaluation

The system's evaluation is guided by the following 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.

5.1. Experimental Design

The evaluation was conducted using three types of tests designed to increase the realism and complexity of the evaluation environment progressively. The evaluation included 7 artificial tests, 3 semi-artificial tests, and 4 real-world use cases. Artificial Tests, generated ad hoc, consist of four accessibility remediation scenarios and three input-validation ones. Semi-Artificial Tests are composed of real accessibility logs, i.e., output of the tool *TimeStump* [9], and AI-generated source code. Real Use Cases contain real applications downloaded from GitHub and the corresponding real accessibility log, i.e., a comment written by an expert based on the *A11yArgus* tool's output [5]. Furthermore, every test was executed twice with different state-of-the-art LLMs: Gemini 2.5 PRO and GPT-5.2 to evaluate the impact of the underlying foundation models. Lastly, to isolate the specific contribution of the multi-agent architecture, the proposed MAS is compared against a Single-LLM Baseline, inspired by the work of Aljedaani et al. [3].

5.2. Results

The evaluation demonstrated that the proposed solution is capable of addressing accessibility issues across multiple Android frameworks and testing scenarios. In particular, the Interpreter Agent correctly validated the inputs 93% of the time (100% with Gemini 2.5 PRO) and correctly identified and mapped the issues to the corresponding ARB category 97% of the time, with an 11% hallucination rate and a 3% omission rate. Furthermore, the Analyzer Agent achieved an 88% correct matching rate with the ARB resolution methods (97% with Gemini 2.5 PRO), selecting the correct component 35 out of 36 times. No instances of self-generated auditory clutter were observed, and when it is already present, the agent addresses it 11 out of 12 times. Note that Gemini has a 100% correct component choice rate and a 100% auditory clutter

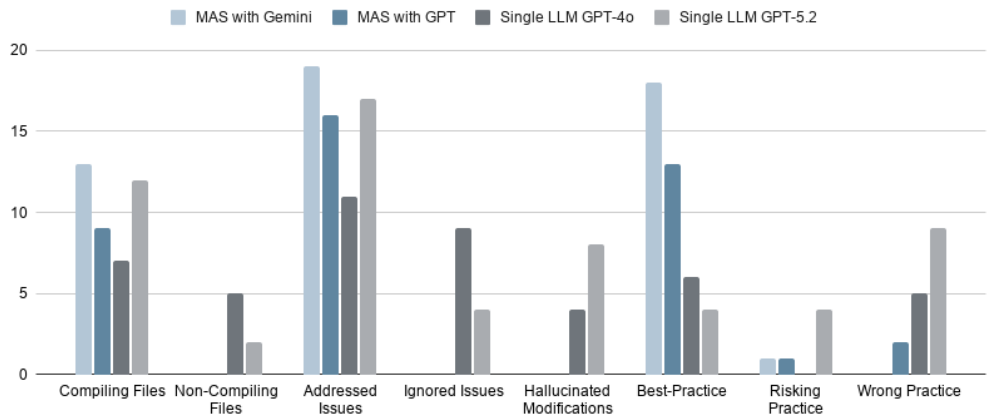

Figure 2: Charts comparing the produced code patches between the MAS and the two single LLMs

addressing rate. We also noticed 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. Moreover, the Repairer Agent produced 100% of files that compile correctly and addresses 100% of the Analyzer Agent's suggestions. The only minor flaws regard the adopted practices used to resolve the issues, 89% of the practices used can be considered the best ones. As emerged before, Gemini achieved 95% of best-practice solutions. Notice that the results are quite independent of the programming language used. Lastly, the Critic and the Reviewer maintain sufficiently rigorous standards to reject suboptimal work, ensuring that the remediation process successfully achieves a valid result in over 97% of cases. 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 (the GPT model is more prone to requesting a reiteration). We noticed that 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.

5.3. Comparison with a Single-LLM Baseline

The bar charts in Fig. 2 compare two single-LLM baselines (one with GPT-4o as in the Aljedaani et al. [3] work, and one with GPT-5.2) with the MAS performances. The comparison highlights a clear advantage of the multi-agent approach. GPT-4o produced non-compiling files in approximately 42% of the evaluated scenarios, ignored 45% of the reported accessibility barriers, and generated four hallucinated modifications unrelated to the requested remediation. Although GPT-5.2 improved compilation success and addressed a larger number of issues, it still produced non-compiling patches, ignored 19% of the barriers, and generated eight hallucinated modifications. Beyond correctness, the quality of the applied fixes differed significantly. The MAS consistently adopted accessibility-aware remediation strategies, whereas the single-LLM approaches frequently selected suboptimal practices that could increase auditory clutter or negatively affect the user experience. In particular, GPT-4o and GPT-5.2 adopted non-recommended remediation strategies in 45% and 64% of the addressed issues, respectively. These results indicate that the observed improvements derive primarily from the multi-agent architecture rather than from the underlying foundation model.

5.4. Discussion

The obtained results allow us to answer the presented RQs:

RQ1 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.

RQ2 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.

RQ3 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. Consequently, improvements in foundation models are likely to benefit the performance of the remediation workflow.

6. Related Work

Existing research has primarily focused on accessibility detection tools, such as *TimeStump* [9] and *A11yArgus* [5], while automated accessibility remediation remains largely underexplored. Recent approaches, such as the single-LLM framework proposed by Aljedaani et al. [3] and the multi-agent system *FixAlly* [8], demonstrated the potential of LLMs for accessibility repair. However, these approaches either rely on isolated issues remediation or focus on the iOS ecosystem, and they do not incorporate explicit validation mechanisms to mitigate hallucinations and unreliable code modifications.

7. Conclusions

This thesis presented an LLM-based Multi-Agent System for the automated remediation of accessibility barriers affecting blind users in Android applications. It combines specialized agents, iterative validation mechanisms, and context-aware reasoning to generate reliable accessibility fixes across multiple Android frameworks. Compared with existing solutions, the proposed architecture introduces three key contributions. First, it enables holistic reasoning across multiple accessibility issues, allowing remediation decisions to consider the surrounding interface context and avoid introducing secondary barriers such as auditory clutter. Second, it incorporates iterative self-validation mechanisms that improve the reliability of generated remediation strategies and code patches. Third, it provides Android-native remediation support across multiple development frameworks. The evaluation demonstrated that the system consistently generated syntactically valid code patches with high remediation accuracy, significantly outperforming single-LLM baselines. These benefits come at the cost of increased execution time and API consumption, due to the multiple model interactions required throughout the workflow. Future work will investigate more cost-efficient architectures, including model distillation strategies. Overall, the results suggest that multi-agent architectures represent a promising direction for automated accessibility remediation in mobile applications.

References

[1] Ericsson mobility report, 2025.

[2] The state of mobile app accessibility report, 2025.

[3] Aljedaani et al. Llms for accessibility in mobile apps: Detection and repair. ACM, 2025.

[4] Chen et al. Accessible or not? an empirical investigation of android app accessibility. *IEEE Transactions on Software Engineering*, 2022.

[5] Ferreira et al. A11yargus: Automated detection and empirical analysis of accessibility issues in android apps. ACM, 2026.

[6] Guo et al. Large language model based multi-agents: A survey of progress and challenges. 2024.

[7] Liu et al. Lost in the middle: How language models use long contexts, 2023.

[8] Mehralian et al. Automated code fix suggestions for accessibility issues in mobile apps. *arXiv preprint arXiv:2408.03827*, 2024.

[9] Mehralian et al. Automated accessibility analysis of dynamic content changes on mobile apps. IEEE, 2025.

[10] Ugare et al. Agentic code reasoning, 2026.

[11] Wang et al. Detecting and repairing role drift in multi-agent collaboration with lightweight protocols. 2026.