Hasselt University

Master's thesis presented for the attainment of the degree of Master of Science in Computer Science

A Multimodal AI-Driven GUI Framework for Dynamic User Adaptation

Author:

Yarne Dirkx

Promotor:

Prof. Dr. Kris Luyten

Academic year 2024-2025

Abstract

This thesis presents a multimodal AI-driven GUI framework for dynamic user interface adaptation, enabling real-time, personalised accessibility enhancements across platforms such as Flutter, SwiftUI, and beyond. Designed with a focus on the health domain, the framework supports motor-impaired, visually impaired, and hands-free users by integrating diverse input modalities including touch, keyboard, voice, and gestures. At its core is Smart Intent Fusion (SIF), a multi-agent architecture powered by large language models (LLMs) via the Google Gemini API. SIF fuses user inputs, events, profiles, and interaction history to infer intent and propose targeted UI adaptations such as button enlargement, contrast enhancement, and navigation mode switching. The reasoning engine combines rule-based logic for predictable, low-latency responses with LLM-driven reasoning for complex or ambiguous cases. The framework provides developer-friendly integration through standardised JSON contracts and a FastAPI backend, making it portable across domains and platforms. Its feasibility is demonstrated through varied configurations and simulated metrics including adaptation accuracy, task completion time, and latency showing measurable improvements in accessibility. By addressing the gap in intelligent, multimodal UI adaptation, this work helps advance human–computer interaction and lays the foundation for future AI models capable of autonomously rewriting UI code, offering a scalable and extensible solution for personalised user experiences.

Acknowledgments

I would like to express my deepest appreciation to all those who have supported and guided me throughout the course of my master's thesis.

First and foremost, I am sincerely grateful to Prof. Dr. Kris Luyten, my thesis supervisor, for his invaluable guidance, constructive feedback, practical tips, and continuous support.

I would also like to thank my fellow students for their encouragement, collaboration, and support during this journey. Special thanks go to my two dogs, Tobias and Casper, whose company during countless late-night coding sessions, and gentle reminders to take breaks helped keep me grounded.

My heartfelt gratitude extends to my family, especially my sister Phaedra, my brother-in-law Dennis, and my parents for their unwavering support, patience, and help with proofreading.

Finally, I am thankful to my friends for their motivation, belief in my abilities, and for making this challenging journey a rewarding one.

This work would not have been possible without the contribution and support of all these people, to whom I am deeply grateful.

Contents

1 Introduction **9**
- 1.1 Background and Motivation . . . 9
- 1.2 Problem Statement . . . 10
- 1.3 Research Objectives . . . 11
- 1.4 Thesis Structure . . . 12

2 Related Work **13**
- 2.1 Multimodal AI in User Interfaces . . . 13
  - 2.1.1 Pointing Devices and Touch Interfaces . . . 13
  - 2.1.2 Voice-Driven Interfaces . . . 14
  - 2.1.3 Gesture and Gaze Integration . . . 14
- 2.2 Adaptive GUIs Across Modalities and Platforms . . . 14
  - 2.2.1 VR Health Applications . . . 15
  - 2.2.2 Challenges in VR UI Design . . . 15
- 2.3 Classical Adaptive UI Techniques . . . 15
  - 2.3.1 Responsive Layouts . . . 16
  - 2.3.2 Context-Aware Design . . . 16
- 2.4 Programmable UIs . . . 16
  - 2.4.1 Reflow . . . 16
  - 2.4.2 UICoder . . . 17
  - 2.4.3 User Interface Adaptation using Reinforcement Learning . . . 17
- 2.5 User Profile built UIs . . . 18
  - 2.5.1 XML-Based Runtime UI Systems . . . 18
- 2.6 Multimodal Fusion and Input Event Modeling . . . 19
  - 2.6.1 Fusion Architectures . . . 19
  - 2.6.2 Event Abstraction Models . . . 19
- 2.7 LLMs as UI Controllers . . . 20
  - 2.7.1 Turning UIs into APIs . . . 20
  - 2.7.2 Agents . . . 20
- 2.8 Health and Accessibility Applications . . . 21

3 System Design and Architecture **22**
- 3.1 Introduction to System Design . . . 22
- 3.2 Overview of the System Architecture . . . 22
- 3.3 Frontend Layer: UI Design and Interaction . . . 23
  - 3.3.1 Interface Elements . . . 23
  - 3.3.2 Adaptation Levels . . . 24
- 3.4 Input Adapter Layer: Multimodal Input Processing . . . 24
- 3.5 SIF Backend Layer: Smart Intent Fusion (SIF) . . . 25
- 3.6 User Profiles and Context Modeling . . . 25
- 3.7 Dynamic Adaptation Mechanisms . . . 26
  - 3.7.1 Adaptation Pipeline . . . 26
  - 3.7.2 Supported Adaptation Actions . . . 26
  - 3.7.3 Continuous Learning and Feedback Loop . . . 27
  - 3.7.4 Design Considerations . . . 27
- 3.8 Chapter Summary . . . 27

4 Smart Intent Fusion (SIF) 28
4.1 Introduction to Smart Intent Fusion . . . 28
4.2 Theoretical Foundations of Smart Intent Fusion . . . 29
4.2.1 Multimodal Fusion . . . 29
4.2.2 Intent Inference . . . 30
4.2.3 Why Hybrid Works Best . . . 30
4.2.4 Connection to Accessibility . . . 30
4.3 User Profile and Context Integration . . . 31
4.3.1 User Profiles . . . 31
4.3.2 User Profile Structure . . . 31
4.3.3 How Profiles Affect Decisions . . . 32
4.3.4 Continuous Learning from History . . . 32
4.3.5 Role in Accessibility . . . 33
4.4 Modeling Multimodal Input Fusion . . . 34
4.4.1 Event Standardisation . . . 34
4.4.2 Timing and Confidence . . . 35
4.4.3 LLM Reasoning in Fusion Decisions . . . 35
4.5 Rule-Based Logic and LLM-Driven Adaptation . . . 36
4.5.1 Rule-Based Logic . . . 36
4.5.2 LLM-Driven Reasoning . . . 36
4.5.3 Hybrid Approach in SIF . . . 37
4.6 Multi-Agent Smart Intent Fusion (MA-SIF) . . . 37
4.6.1 Why Multiple Agents? . . . 37
4.6.2 Agent Roles . . . 37
4.6.3 Adaptation Flow . . . 38
4.6.4 Dynamic Configuration . . . 38
4.6.5 Example in Action . . . 39
4.6.6 Benefits of the Multi-Agent Approach . . . 39
4.7 Prompt Engineering for LLMs in SIF . . . 39
4.7.1 LLM Prompt Design Principles . . . 40
4.7.2 Prompt Structure from `sif_config.json` . . . 40
4.7.3 Disjunction Ambiguity in LLM Interpretation . . . 40
4.7.4 Balancing Model Parameters . . . 41
4.7.5 Avoiding Hallucinations and Bad Values . . . 41
4.8 Performance and Evaluation Metrics for AI Logic . . . 42
4.9 Limitations and Challenges of LLM Integration . . . 42
4.9.1 LLM selection . . . 42
4.9.2 Reliability and Latency Constraints . . . 42
4.9.3 Hallucinations and Invalid Output . . . 42
4.9.4 Token Limits and Context Size . . . 43
4.9.5 Validator Complexity . . . 43
4.9.6 Dependency on External APIs . . . 43
4.10 Future Directions for AI-Driven Adaptation . . . 43
4.11 Chapter Summary . . . 44

5 An Adaptive Multimodal GUI Framework using LLMs 45
5.1 Introduction to an Adaptive Smart Home Controller . . . 45
5.2 Development Environment . . . 45
5.3 Frontend Implementation: Smart Home Controller . . . 46
5.4 Input Adapter Layer: Multimodal Input Handling . . . 47
5.5 SIF Backend Layer: Implementation of Adaptation Logic . . . 48
5.5.1 Webserver layout and endpoints . . . 48
5.5.2 Data persistence and history management . . . 48
5.5.3 Smart Intent Fusion and MA-SIF . . . 48
5.5.4 Structured outputs and guardrails . . . 48
5.5.5 Rule-based fallback and resilience . . . 49
5.5.6 Latency, partial results, and error handling . . . 49
5.5.7 Security and CORS considerations . . . 49

5.5.8 Summary . . . 49
5.6 User Profile and Context Implementation . . . 49
5.7 Dynamic Adaptation Mechanisms: Implementation Details . . . 50
5.8 Cross-Platform Implementation Considerations . . . 51
5.9 Design Decisions . . . 51
5.9.1 Modularity Over Monolithic Design . . . 51
5.9.2 WebSocket for Real-Time vs. HTTP for Batch Processing . . . 51
5.9.3 MongoDB for Persistent Storage . . . 51
5.9.4 Rule-Based Fallback with LLM Reasoning . . . 51
5.9.5 Multi-agent LLM reasoning (MA-SIF) vs single-agent LLM reasoning (normal SIF) 52
5.10 Implementation Challenges and Solutions . . . 52
5.10.1 LLM reliability and output consistency . . . 52
5.10.2 Performance under real-time constraints . . . 52
5.10.3 Safeguards against malicious or replay attacks . . . 53
5.10.4 Testing with incomplete modalities . . . 53
5.10.5 Security and trust boundaries . . . 53
5.11 Chapter Summary . . . 53

6 Evaluation **54**
6.1 Feasibility Study: Evaluation Approach . . . 54
6.2 System Demonstration and Experimental Setup . . . 54
6.2.1 Demonstration Scenarios and Configurations . . . 54
6.2.2 Hardware and Environment . . . 54
6.3 Results and Observations . . . 54
6.3.1 Adaptation Effectiveness Across Configurations . . . 54
6.3.2 Accessibility Impact Analysis . . . 54
6.3.3 Adaptation Performance . . . 54

7 Discussion and Future Work **55**
7.1 Implications for Accessibility and HCI . . . 55
7.2 Key Findings and Contributions . . . 55
7.3 Comparison with Related Work . . . 55
7.4 Limitations and Challenges . . . 55
7.5 Future Work . . . 55
7.5.1 Extending to Existing UIs . . . 55
7.5.2 UI Component Analyzer . . . 55
7.5.3 Enhancing Multimodal Inputs . . . 55
7.5.4 LLM Agents for Autonomous Adaptation . . . 55
7.5.5 Specialized AI Model for UI Adaptation . . . 55

8 Conclusion **56**
8.1 Summary of Contributions . . . 56
8.2 Final Remarks . . . 56

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Whether it's a smartphone app, a website, or a smartwatch, users primarily interact with technology through graphical user interfaces (GUIs). These interfaces rely on visual elements such as icons, buttons, and menus to enable intuitive and efficient interaction, replacing the need for complex text-based commands. Before GUIs became standard, users had to operate computers through command-line interfaces, requiring memorization of commands and technical know-how. This barrier limited computing to experts. GUIs removed that limitation by introducing visual metaphors, opening up digital technology to the general public. Since their introduction in the 1980s, GUIs have revolutionized how humans engage with computers, transforming once-specialized machines into everyday tools. As the critical interface layer between human and machine, GUIs translate complex computational tasks into accessible, visually manageable actions.

Over the decades, GUIs have evolved far beyond their early desktop roots. From the classic WIMP model windows, icons, menus, and pointers interfaces have expanded to include touch, voice, gesture, and even spatial interactions in augmented and virtual reality. Today's user interfaces are increasingly multimodal, allowing for richer and more natural interactions. Yet, despite this evolution in interaction modalities, the underlying structure of most GUIs remains largely static. Interfaces are typically designed with a one-size-fits-all approach, offering the same layout, behavior, and visual elements to all users, regardless of their needs or context. This static nature becomes a barrier in today's increasingly diverse and dynamic world, especially for users with accessibility challenges. People with visual impairments, motor difficulties, or those who rely on hands-free interaction methods often struggle with standard interfaces that do not adapt to their specific needs. For example, small buttons can be difficult to tap for users with tremors, while low contrast or lack of screen reader support limits usability for visually impaired users. Moreover, users interact with devices in various environments some noisy, some bright, some on the move, further complicating interaction for those needing alternative input or output modalities. Devices range widely in size and capability, from smartwatches and smartphones to desktop monitors and VR headsets, but accessibility features often remain limited or inconsistent across platforms. This gap highlights a key challenge in interface design: how can we create graphical user interfaces that dynamically adapt to individual users' accessibility needs and context, ensuring inclusive and efficient interaction for all?

Adaptive UIs are user interfaces that can adjust themselves dynamically to accommodate the user's accessibility needs and contextual factors. These interfaces are designed to intelligently modify their layout, behavior or visual appearance based on individual user preferences, abilities, and environmental conditions. By responding to multimodal inputs such as touch, voice, gestures, and gaze, adaptive UIs aim to provide a more personalized, inclusive, and efficient user experience, especially for users with diverse accessibility requirements. When we call an interface "intelligent" we mean it can perceive, interpret, and respond to complex user behaviors and contexts autonomously. This level of responsiveness goes beyond simple preset rules, necessitating systems that learn from user interactions and adapt close to or in real-time. Artificial Intelligence (AI), especially advances in machine learning and large language models, offers powerful tools to enable such intelligent adaptation. By processing multimodal inputs, UI context and user data, AI-driven interfaces can dynamically tailor themselves to meet diverse user needs, making accessibility more effective and seamless.

Large Language Models (LLMs) like GPT have revolutionized natural language understanding, processing and generation. Their ability to comprehend context, infer intent, and produce human-like responses makes them highly suitable for interpreting user intent, preferences, and even subtle cues from multimodal inputs such as speech, gestures, or eye movements. In the context of adaptive UIs, LLMs can act as intelligent controllers that translate diverse and complex user interactions into actionable interface adaptations. For example, an LLM can understand a voice command like "make buttons bigger" or interpret hesitation in gestures to trigger UI changes that enhance usability for motor-impaired users. Furthermore, LLMs can dynamically generate or modify UI elements by turning the interface itself into an API enabling on-the-fly customization that is both personalized and context-aware.

Despite growing research and technological capabilities, adaptive user interfaces remain rare in practical use, often due to complexity in implementation and lack of robust frameworks. This gap motivated this thesis, inspired in part by emerging ideas on treating UIs as APIs controlled by intelligent models, an approach highlighted in recent discussions in the AI and HCI communities. Large Language Models (LLMs), in particular, offer a promising interface for driving these intelligent UI adaptations due to their ability to reason over context and user intent. However, current LLMs also come with challenges: latency when used via APIs, potential for hallucinations, a lack of specialization for GUI reasoning tasks or coding like rewriting UIs at runtime. These limitations highlight the need for a structured framework that can integrate intelligent models into adaptive UI systems while managing their weaknesses effectively. Rather than seeing LLMs as perfect agents, this thesis treats them as powerful, context-aware assistants that when properly guided and constrained, can significantly improve how interfaces adapt to users' real-time accessibility needs.

1.2 Problem Statement

Despite increasing attention on accessibility and personalization in user interfaces, most applications remain static and poorly adapted to users' changing contexts or abilities. While current interfaces can adapt in high-level ways such as adjusting to screen size, orientation or theme, these adaptations are mostly cosmetic. They do not address the deeper challenge of understanding and responding to the user's intent, cognitive state, or physical limitations in real time.

Designing truly adaptive UIs remains uncommon, in part due to the technical difficulty of modeling dynamic behavior and the lack of robust, general-purpose frameworks. Furthermore, real-world interaction is often messy and unpredictable: small variations in input such as a mistap, delayed reaction, or environmental noise can result in significantly different needs or outcomes. This non-linearity mirrors the principles of chaos theory, where minor initial differences lead to divergent results, making rule-based UI design fragile.

Addressing this requires systems capable of interpreting nuanced, multimodal signals and adapting accordingly, a task well suited to modern AI techniques such as large language models (LLMs). Yet existing UI frameworks rarely integrate such models in a way that supports real-time, intelligent adaptation, leaving a significant gap between the potential of adaptive interfaces and their current practical application.

1.3 Research Objectives

The primary objective of this research is to design and validate a modular framework for AI-driven GUI adaptation that enables dynamic, context-aware UI behavior. This framework will integrate multimodal input interpretation, a structured UI representation layer, and an SIF backend that suggests or performs near real-time adaptations by leveraging multiple data sources, including historical user behavior, contextual input signals, and dynamically evolving user profiles. The work aims to bridge the gap between static UI designs and adaptive, intelligent user experiences, especially for accessibility and personalization use cases. Additionally, the framework is intended to support cross-platform compatibility and scalability, ensuring broad applicability. The effectiveness of the approach will be demonstrated through prototype implementation and user evaluation, focusing on enhancing usability and inclusivity for diverse user groups.

To achieve this overarching goal, the following specific research objectives are defined:

1. Develop a Cross-Platform UI Framework: Design and implement a modular, scalable user interface framework that supports consistent rendering and dynamic adaptations across Flutter (desktop/-mobile), React (web), and SwiftUI (Apple platforms), with extensibility for future platforms like Unity (VR/AR). The framework will include a test UI, the "Art Explorer" application, featuring a vertical scrollable card list with interactive elements (cards, buttons, text).
2. Enable Multimodal Input Processing: Integrate support for multimodal user inputs, including touch/tap (via GestureDetector, onClick, .onTapGesture), keyboard (Tab/Enter via RawKeyboard-Listener, onKeyDown, .onKeyPress), voice (commands like "play" via speech_to_text, Web Speech API, or mock events), and gestures (hand movements via MediaPipe or mock events), to capture rich interaction context for accessibility-focused adaptations.
3. Support Accessibility-Focused Adaptations: Enable dynamic UI adaptations to enhance accessibility for motor-impaired, visually impaired, and hands-free users, including but not limited to:
    - Increasing element sizes (cards: 1.2x, buttons: 1.5x, text: 20pt).
    - Repositioning elements (e.g., buttons move 20px right).
    - Enhancing contrast (e.g., black text on white background).
    - Adjusting scroll speed (e.g., higher friction at 0.015).
    - Switching to voice navigation mode on repeated miss-taps (simulated if needed).
4. Create a Developer-Friendly expandable SDK: Design a cross-platform SDK with pre-built UI templates and event hooks (e.g., AdaptiveUI.sendEvent) to minimize integration effort (5–10 lines per platform), ensuring compatibility with standard inputs (touch, keyboard) and advanced inputs (voice, gestures) via configuration.
5. Evaluate Framework Effectiveness: Assess the framework's performance using simulated results, focusing on:
    - Adaptation accuracy (precision/recall of LLM-suggested actions).
    - User performance (task completion time, error rate for simulated motor-impaired users).
    - System performance (end-to-end latency for cloud-based adaptations).

Together, these objectives form the foundation of a robust and extensible system for AI-driven, context-aware GUI adaptation. By combining multimodal input analysis, platform-agnostic UI rendering, and real-time adaptive intelligence, this research seeks to advance both accessibility and personalization in modern interfaces. The successful implementation and evaluation of this framework will demonstrate the potential for generalized adaptive systems across domains, setting the stage for future integration in more immersive environments such as VR/AR, and domains like Health Care.

1.4 Thesis Structure

This section outlines the organization of the thesis to guide the reader through its content and structure. The thesis is structured as follows:

- **Chapter 1: Introduction**
  This chapter introduces the research, providing the background and motivation for developing a multimodal AI-driven GUI framework (Section 1.1). It defines the problem of static, non-adaptive user interfaces and their limitations for accessibility (Section 1.2), outlines the research objectives, including the development of the Smart Intent Fusion feature and cross-platform UI support (Section 1.3), and presents the thesis structure (Section 1.4).
- **Chapter 2: Background and Related Work**
  This chapter reviews the state-of-the-art in human-computer interaction (HCI), focusing on adaptive user interfaces, multimodal input processing, and (AI-driven) personalization. It discusses existing frameworks for accessibility, user profile built UIs and limitations of static UIs, and the role of large language models (LLMs) in UI adaptation, positioning the proposed framework's novelty.
- **Chapter 3: System Design and Architecture**
  This chapter details the proposed framework's architecture, including the frontend layer (cross-platform UI rendering), input adapter layer (standardizing multimodal inputs), and SIF backend logic (rule-based and LLM-driven Smart Intent Fusion). It describes the "Art Explorer" test UI, user profiles, and adaptation behaviors (e.g., increasing element sizes, adjusting scroll speed) for accessibility.
- **Chapter 4: Implementation**
  This chapter presents the implementation of the framework, covering the Flutter-based "Art Explorer" UI with touch, keyboard, and simulated voice/gesture inputs, the input adapter layer's JSON contract, and the FastAPI backend with mock LLM responses. It discusses developer integration via the SDK and simulation strategies for feasibility.
- **Chapter 5: Evaluation**
  This chapter evaluates the framework using simulated results, focusing on adaptation accuracy (precision/recall of LLM-suggested actions), user performance (task completion time, error rate), and system performance (latency).
- **Chapter 6: Discussion and Future Work**
  This chapter proposes future work, such as specialized AI models for dynamic UI code generation, Unity VR/AR support, and additional modalities (e.g., eye tracking).
- **Chapter 7: Conclusion**
  This chapter summarizes the contributions, including the Smart Intent Fusion feature and accessibility improvements, and discusses limitations (e.g., reliance on simulated results)

Chapter 2

Related Work

2.1 Multimodal AI in User Interfaces

User interfaces have evolved far beyond simple point-and-click paradigms. Modern systems increasingly incorporate **multimodal inputs**, integrating different sensing and communication modalities such as voice or speech commands, pen and touch interaction, eye tracking, and full-body gesture tracking. These technologies aim to create more natural, flexible, and inclusive experiences [**10.1145/319382.319398**]. Multimodal interfaces aim to mirror human communication, which rarely relies on a single channel, thereby expanding the possibilities for interaction and enabling more robust and adaptive user experiences.

A fundamental advantage of multimodal interfaces is their potential to improve accessibility and inclusivity. For example, voice commands can be crucial for users with motor impairments, while gaze-based control may assist users who cannot use their hands for input. Meanwhile, gestural input like in VR settings can support contactless control, useful in medical environments, during physical rehabilitation exercises or even astronaut training. Furthermore, the combination of modalities can reduce error rates and cognitive load by providing redundant and complementary channels of communication, improving user satisfaction and performance [**10.1145/319382.319398**].

Historically, the field of multimodal interaction gained attention through early research prototypes combining speech and pen input, which demonstrated that parallel and redundant channels lead to more fluid interaction [**10.1145/319382.319398**, **Oviatt2004**]. In recent years, rapid advances in machine learning, especially deep learning, have further enabled real-time recognition of speech, gestures, and gaze on consumer-grade hardware [**lugaresi2019mediapipeframeworkbuildingperception**, **Choudhury2015**].

2.1.1 Pointing Devices and Touch Interfaces

Before the rise of multimodal and intelligent input modalities, most user interfaces primarily relied on pointing devices such as the mouse, trackball, and touchpad. Fitts' law [**Fitts1954**] and subsequent pointing models formed the foundation for optimizing target sizes and layouts to reduce pointing time and error rates. Touchscreens expanded on these paradigms, introducing direct manipulation, gesture-based scrolling and multi-touch interactions. Touch interfaces, now ubiquitous in smartphones and tablets, benefit from intuitive mappings between finger movements and on-screen actions but also face challenges related to occlusion, precision, and physical fatigue [**Wigdor2011BraveNUI**].

Recent research has introduced adaptive techniques, such as "sticky" or magnetic cursor effects, which are already in use today as normal or accessibility features on for example Apple devices. Which help guide the pointer toward interactive elements, reducing effort and error, particularly for users with motor impairments [**Cockburn2008Sticky**]. These principles guide contemporary accessibility enhancements and encourage advancements in multimodal environments. In this thesis, we draw inspiration from those adaptive cursor behaviors to inform dynamic UI element resizing and magnetic effects in multimodal contexts, bridging classical pointing interactions with "smart" AI-driven adaptations.

2.1.2 Voice-Driven Interfaces

Voice interaction has become mainstream through virtual assistants like Amazon Alexa, Google Assistant, and Apple's Siri. These systems utilize automatic speech recognition (ASR) and natural language processing (NLP) to interpret user commands and execute them on the device [**Hoy02012018**]. Voice input offers hands-free accessibility benefits, particularly useful for users with mobility impairments or in multitasking scenarios. However, it also presents challenges such as interpreting unclear or ambiguous speech, addressing privacy concerns, and maintaining reliability in noisy environments [**HCI-076**]. Combining voice with complementary modalities such as gestures or gaze tracking has been shown to improve disambiguation and user confidence [**s21237825**]. Hybrid systems support error correction, implicit confirmation, and increased interaction bandwidth, thereby enhancing overall usability.

2.1.3 Gesture and Gaze Integration

Gesture-based interactions enable users to control interfaces through hand and body movements, providing expressive and intuitive command options. Advances in computer vision techniques and frameworks such as MediaPipe Hands and OpenPose have made real-time gesture recognition feasible on consumer-grade hardware including desktops and mobile devices [**lugaresi2019mediapipeframeworkbuildingperception**]. Eye tracking is another important modality, capturing user attention and intention. It has been used to implement gaze-contingent interfaces where UI elements respond dynamically to where users are looking, reducing physical effort and increasing interaction speed [**Duchowski2017**]. The integration of gesture and gaze input creates a powerful multimodal system, enabling users to point, select, and confirm actions more naturally. Studies indicate that combining these modalities improves error tolerance and enhances overall user experience, especially in contexts where precise manual input is difficult or limited, such as virtual reality environments [**7893331**, **Gazeinteractioninthepost-WIMPworld**].

One of the most influential early systems that explored the practical use of gaze in everyday computing contexts is the GUIDe project by Kumar and Winograd (2007), which demonstrated how gaze could be used as a lightweight augmentation of traditional input methods rather than a full replacement. Their EyePoint technique introduced a refined two-step "look-press-look-release" interaction loop that allowed for highly accurate gaze-based pointing by combining magnification with gaze refinement. Unlike previous systems that relied solely on dwell-based activation, GUIDe's use of keyboard-assisted targeting reduced false activations (the "Midas Touch" problem) and allowed for precise user control.

They extended this principle to application switching (EyeExposé) and adaptive scrolling (EyeScroll), showing that gaze input could become a viable interaction modality for able-bodied users if the design balanced gaze with explicit control channels like hotkeys or contextual awareness. Importantly, EyeScroll demonstrated adaptive scrolling speeds based on live gaze-driven reading behavior, which is a strong precedent for personalization and multimodal input adaptation. These contributions illustrate how gaze can serve as both a passive signal and an active intent channel, especially when fused with complementary modalities like touch or speech.

2.2 Adaptive GUIs Across Modalities and Platforms

As user interfaces evolve to support an increasing variety of input modalities and devices, adaptive graphical user interfaces (GUIs) have gained significant attention. Adaptive GUIs are designed to modify their layout, behavior, and appearance dynamically in response to contextual information, user preferences, or real-time interaction patterns. This adaptivity aims to enhance usability, accessibility, and personalization across a wide range of platforms, from traditional desktop systems to mobile devices and immersive virtual reality (VR) environments.

Adaptation can take many forms. For example, a GUI might increase button sizes for users with motor impairments, change color schemes to improve contrast for users with visual impairments, or rearrange content based on the user's task context or dominant hand. On desktop and mobile devices for example, adaptivity has often focused on supporting accessibility and optimizing layouts for different screen sizes or resolutions. Responsive design frameworks, which automatically adjust content and element placement across devices, represent one form of early structural adaptivity. More advanced adaptive systems might learn user habits over time and present solutions based on the users context by for example integrating large language models (LLMs) and advanced AI reasoning [**Gajos2008SUPPLE**].

2.2.1 VR Health Applications

Virtual reality offers unique opportunities for healthcare applications by enabling immersive, engaging, and personalized experiences for therapy, rehabilitation, and training [**Rizzo2017VRHealthcare**]. Adaptive GUIs in VR can tailor the interface to the user's physical and cognitive abilities, dynamically adjusting element sizes, positions, or interaction techniques to accommodate motor impairments, fatigue, or sensory limitations [**Freeman2020AdaptiveVR**]. For example, VR systems have been developed to assist stroke patients in motor rehabilitation by adjusting the difficulty and interactivity of tasks in real time, providing immediate feedback and encouragement [**Laver2017VRStroke**]. Adaptive UI elements also support elderly users or individuals with limited VR experience by simplifying controls and enhancing legibility [**Slater2020Usability**]. Despite these benefits, the use of adaptive GUIs in VR healthcare remains a nascent field with ongoing research to optimize adaptation strategies and validate clinical efficacy.

2.2.2 Challenges in VR UI Design

Designing effective VR interfaces poses several challenges distinct from traditional 2D GUIs. First of all, the 3D environment introduces spatial complexity requiring new paradigms for navigation and interaction [**Bowman20023DUI**]. Secondly, user comfort is critical, as poorly designed VR UIs can induce motion sickness or problems with fatigue and cognitive load [**LaViola2000VRSimulatorSickness**]. Adaptive GUIs must consider these factors by adjusting visual density, interaction distances, and input modalities. Third, hardware variability and tracking inaccuracies complicate consistent input interpretation, especially for hand tracking and gaze estimation [**Jerald2015VRBook**]. Which heavily depends on the hardware precision and its algorithms. Finally, user diversity in terms of physical abilities, experience with VR, and cognitive load demands highly flexible and personalized UI adaptation mechanisms [**Vasiljevic2017AdaptiveVRAccessibility**]. Addressing these challenges requires integrating real-time sensor data, user profiling, and AI-driven adaptation logic, motivating the development of multimodal AI-driven frameworks like the one proposed in this thesis.

2.3 Classical Adaptive UI Techniques

Before the widespread use of artificial intelligence and machine learning in dynamic interfaces, adaptive graphical user interfaces (GUIs) primarily relied on deterministic, rule-based strategies. These classical approaches, while limited in personalization and flexibility, were instrumental in shaping the early landscape of user interface adaptation. They focused on fixed logic, device-specific configurations, and contextual parameters defined explicitly by designers or developers. Although static by today's standards, these techniques addressed fundamental issues of usability, accessibility, and device heterogeneity.

One of the most foundational classical techniques is responsive layout design, which allows a user interface to adjust its layout and elements based on screen size, orientation, and device type. This approach is especially prominent in web and mobile design, where frameworks like CSS media queries or constraint-based layouts dynamically reposition and resize UI components to preserve usability across different platforms. While responsive design improves accessibility and device compatibility, it does not adapt to individual user behavior or preferences the adaptations are based solely on environmental conditions like screen dimensions or device type.

Another key classical technique is context-aware UI adaptation, where the interface adjusts based on predefined environmental or situational variables such as location, time of day, network connectivity, or battery level. For instance, a mobile app might switch to a dark theme automatically at night or reduce update frequency when on a metered connection. These adaptations are typically triggered by fixed if-then rules rather than probabilistic models, and although they improve relevance and efficiency, they lack the ability to learn from user interaction patterns over time.

Additionally, user role-based adaptations were sometimes employed in enterprise applications or educational platforms. For example, an interface might present different levels of information depending on whether the user is an administrator, a student, or a guest. These adaptations were predetermined by role or permissions, not by user behavior or inferred intent.

2.3.1 Responsive Layouts

Responsive design emerged as a critical approach to address the rapid increase of different screen sizes and devices, particularly with the rise of smartphones and tablets. Instead of creating separate fixed interfaces for each device, responsive layouts allow a single design to automatically adjust its elements to fit various screen dimensions and orientations based on core techniques like flexible grids, media queries, and adaptive content strategies. Those enable dynamic resizing and rearrangement of UI elements [**Marcotte2010Responsive**]. In mobile development, declarative UI frameworks such as Flutter and SwiftUI adopt similar principles, using constraints and reactive layouts that adapt hierarchically to screen changes. While originally developed to support multiple screen sizes, responsive layouts also improve accessibility by allowing users to zoom or scale interface components without breaking layout integrity. Despite their flexibility, traditional responsive designs are typically static in behavior they do not react to real-time user behavior or contextual signals beyond the device dimensions.

2.3.2 Context-Aware Design

Context-aware interfaces extend the notion of adaptation by responding to more complex environmental and user-specific factors. Originating in ubiquitous and pervasive computing research, context-aware design involves sensing parameters such as location, time of day, user activity, nearby devices, or even physiological signals to modify interface behavior [**Schilit1994ContextAware**, **Dey2001Context**]. For example, a context-aware mobile application might switch to a dark theme automatically at night or suggest different UI shortcuts when the user is driving versus sitting at a desk. In smart home systems, interfaces may change based on proximity sensors or room occupancy data. Some adaptive systems also integrate user profiles and long-term behavior patterns. For example, the SUPPLE system automatically generates personalized interfaces optimized for a user's specific motor abilities and preferences by solving optimization problems over interface layouts with the effect of facilitating faster access and lower error [**Gajos2008SUPPLE**]. Such rule-based or optimization-based approaches set the stage for later, more sophisticated AI-driven adaptations by highlighting the importance of context and user modeling.

While classical techniques like responsive layouts and context-aware design lack the semantic understanding and reasoning capabilities of modern AI systems, they provide robustness and predictability. They form a crucial baseline against which the benefits of AI-enhanced adaptations can be evaluated. Furthermore, they remain relevant in many production systems due to their lower computational cost and easier predictability for developers and designers.

2.4 Programmable UIs

Traditionally, user interfaces have been treated as static artifacts, closely tied to fixed layouts and hard-coded interaction patterns. However, recent advances in computational understanding of user interfaces propose a shift towards making UIs "programmable" that is, representing them as dynamic, semantically rich structures that can be analyzed, modified, and generated automatically. This paradigm enables interfaces to become more flexible and adaptable, allowing developers and even AI systems to reason about and transform UI elements in response to changing contexts, user needs, or platform constraints. Such a programmable layer is essential for realizing fully adaptive, AI-driven interfaces that can personalize experiences and improve accessibility in real time.

2.4.1 Reflow

A notable example of pixel-based UI adaptation is Reflow, introduced in Chapter 7 of Wu's dissertation on computational understanding of user interfaces [**Wu2024**]. Reflow proposes a system that automatically refines and optimizes touch interactions in mobile applications using only pixel-level information, without requiring access to the underlying application code or view hierarchy like a widget tree. The system operates by first detecting UI elements from a screenshot using machine learning-based pixel analysis. It then refines the layout based on a user-specific spatial difficulty map, which identifies the difficult-to-access areas of the screen, derived from calibration tasks that capture individual motor abilities and preferences. Finally, Reflow re-renders a modified version of the UI with optimized element positions and sizes, ensuring that critical interactive components are easier to reach and select.

Reflow exemplifies how computational understanding of UIs at the pixel level can enable automated personalization and accessibility enhancements even in closed-source or inflexible applications. It illustrates a promising direction for future adaptive systems that aim to provide user-specific improvements without requiring cooperation from original app developers or extensive code modifications. This aligns directly with the goals of AI-driven UI frameworks focused on dynamic, user-centered adaptation. Furtermore, the user study showed an average increase of 9% in interaction speed as well as improved interaction speeds by up to 17% [**Wu2024**].

2.4.2 UICoder

A further advancement in making user interfaces programmable is presented through UICoder, as described in Chapter 8 of Wu's dissertation on computational understanding of user interfaces [**Wu2024**]. While earlier systems such as Reflow focused on pixel-level adaptations of existing applications, UICoder tackles the challenge of generating high-quality UI code directly from textual descriptions using large language models (LLMs). UICoder addresses two major limitations of previous code-generation approaches for UIs: the scarcity of high-quality, self-contained UI code in existing datasets and the difficulty LLMs face in incorporating visual or spatial feedback into their training. Rather than relying on manually curated or external datasets, UICoder introduces an automated method to iteratively generate, filter, and refine synthetic UI code datasets.

The system begins by prompting an existing LLM to generate large collections of UI code samples (specifically SwiftUI, Apple's coding framework) from textual descriptions. These outputs are aggressively filtered and scored using compilers (to ensure syntactic correctness) and vision-language models (to assess visual relevance), producing a refined dataset for further finetuning. By iteratively repeating this cycle generation, filtering, and finetuning UICoder progressively learns to produce syntactically correct and visually coherent UI code. Derived from the StarCoder family (which lacked extensive Swift training data), UICoder ultimately generated around one million SwiftUI programs over five iterations. Despite these constraints, it significantly outperformed all open-source baselines and approached the performance of larger proprietary models in both automated and human evaluations. By leveraging automated feedback instead of costly human annotations, UICoder demonstrates a scalable approach to training LLMs for UI code generation. This method shows how generative models combined with programmatic refinement loops can enable on-demand creation of high-quality, personalized UIs directly supporting visions of adaptive and dynamically generated interfaces.

UICoder exemplifies the shift from static UI codebases toward dynamic, generatively defined interfaces. In the context of AI-driven adaptive GUIs, such a capability enables systems to not only adjust existing layouts but also generate entirely new interface code on demand, tailored to specific user preferences or device contexts. This directly supports the vision of self-adaptive and user-personalized UI frameworks proposed in this thesis.

2.4.3 User Interface Adaptation using Reinforcement Learning

Recent advances in programmable and adaptive user interfaces have explored the integration of machine learning techniques particularly Reinforcement Learning (RL) to support real-time personalization and dynamic behavior. A notable example is the doctoral work by Gaspar-Figueiredo (2023), which proposes a UI adaptation framework that leverages RL in conjunction with physiological data to enhance user experience (UX) [**gaspar2023learning**].

In this framework, RL is used to determine optimal UI adaptations (e.g., layout or content adjustments) by continuously interacting with the user and optimizing long-term reward signals. What sets this work apart is the use of physiological signals such as eye tracking, EEG, or other biosignals as objective measures of user response. This addresses a key limitation of traditional evaluation techniques that rely on subjective self-reporting, which can introduce bias or fail to capture moment-to-moment changes in experience.

The system learns from users' physiological reactions and interaction behaviors to determine which adaptations are effective in improving usability and engagement. Over time, the interface becomes more personalized and context-aware, reacting not only to explicit input but also to subtle cues from the user's cognitive or emotional state. This approach exemplifies a new direction in programmable UIs: systems that are not only defined by abstract models (e.g., UIML or USiXML) but are also capable of learning and evolving through interaction. By combining adaptive logic, ML-driven decision-making,

and, biosignal-based feedback, such frameworks could serve as the basis for next-generation interfaces especially in applications involving accessibility, cognitive load, or health and wellness.

2.5 User Profile built UIs

The increasing diversity of devices and user contexts has driven research into creating adaptable user interfaces that are both device-independent and user-centered. A significant challenge arises in balancing generalization so an interface works across multiple devices and personalization so it aligns with the unique preferences and capabilities of individual users.

To address this, Luyten et al. [**luyten2005profile**] proposed a framework that combines high-level XML-based user interface description languages (UIDLs), particularly UIML, with MPEG-21 Part 7 (Digital Item Adaptation) user profiles. UIML (User Interface Markup Language) allows designers to abstract the UI from concrete implementations, enabling interfaces to be rendered differently depending on device constraints. For example, a single abstract element like "choice from a range" can map to different widgets (slider, list, or text input) depending on the target platform. In this approach, MPEG-21 user profiles capture individual user preferences and requirements (e.g., accessibility needs, preferred interaction styles), which can then dynamically guide the adaptation of the UI described in UIML. This enables the generation of multi-device, personalized interfaces that are both broadly deployable and tailored to specific user needs.

An implemented prototype demonstrated how combining UIML and MPEG-21-based user profiles allows for seamless adaptation of UI layouts and interactions while minimizing design effort. By leveraging abstract UI definitions and structured user profiles, this method enhances both accessibility and usability across diverse platforms, making interfaces more "granny-proof" and inclusive. This user-centric adaptation model supports the vision of highly personalized digital experiences without sacrificing cross-device compatibility, contributing an important step toward universal, accessible, and adaptable user interfaces.

Early profile-based UI adaptation systems were often limited by the rigidity of XML-based markup languages and the static nature of their user models. Profiles were typically loaded once and assumed relatively stable user needs, which limited the ability to adapt interfaces dynamically over time or in real-time scenarios.

However, these systems laid foundational groundwork for modern AI-augmented adaptation layers. For instance, where MPEG-21 profiles might have been manually filled or gathered via forms, today's systems can infer similar parameters from behavioral data using machine learning. Similarly, abstraction principles from UIML still resonate in declarative UI frameworks like Flutter, SwiftUI, or React Native, which separate layout logic from presentation and enable device-responsive rendering.

These classical approaches remain valuable not only for historical understanding but also as a reminder of the importance of modularity, abstraction, and structured user modeling in UI design principles that continue to influence the development of multimodal, intelligent frontends today.

2.5.1 XML-Based Runtime UI Systems

One early precursor to programmable and adaptive UI systems was the use of XML-based runtime user interface description languages, especially for resource-constrained mobile and embedded systems. A notable example is the work by Luyten and Coninx (2001), who proposed a method that leverages XML for describing UI components and Java for rendering them on mobile devices such as PDAs. Their system allowed interfaces to be dynamically serialized, transmitted, and rendered on client devices based on contextual variables such as the user's role, device capabilities, and preferences. For example, the interface for controlling a projector would differ depending on whether the user is a professor or a technician, enabling personalized task-specific controls.

This work introduced several important ideas that prefigure modern programmable and adaptive UIs. First, it treated UIs as dynamic data rather than static views, enabling runtime generation and adaptation. Second, it used constraint-based filtering mechanisms to tailor UI components based on context. Third, it demonstrated that a separation between the UI's description (in XML) and execution (in Java) could support modular, reusable interface logic.

Although this approach predates current AI-powered frameworks, it exemplifies early efforts toward what is now called adaptive or context-aware UI. Its emphasis on platform-independence, modularity, and runtime flexibility directly anticipates features found in modern frameworks like Flutter, React, and Unity UI especially when extended with AI-driven mediation and multimodal inputs.

2.6 Multimodal Fusion and Input Event Modeling

As user interfaces become increasingly multimodal, systems must handle diverse streams of input signals in a coherent way. Multimodal fusion refers to the process of integrating these different input modalities such as voice, touch, gaze, and gestures into unified semantic commands. This enables interfaces to interpret combined user inputs more naturally and contextually, mirroring how humans often combine speech and gestures in daily communication.

Closely related, input event modeling abstracts low-level raw data (like finger coordinates, gaze points, or audio waveforms) into higher-level representations of user intent (such as “select”, “scroll”, or “confirm”). This abstraction layer serves as an essential bridge between noisy sensor data and actionable interface responses. By modeling input events at different levels of granularity from raw physical movements to semantic-level intents systems can reason about user behavior, handle ambiguities, and provide more robust feedback.

For example, in a health application, simultaneous voice and gaze inputs might be fused to allow a patient to say “yes” while looking at a confirm button, providing redundancy that improves reliability for users with motor or speech impairments. Similarly, in VR or AR, interpreting combined hand gestures and head movements enables more precise object manipulation and scene navigation, which would be challenging with single-modality input alone.

2.6.1 Fusion Architectures

Several architectures have been proposed to integrate multimodal inputs effectively. Early fusion approaches combine raw input data at a low level, for example merging voice and gesture signals before any individual interpretation occurs. While this can enable richer context awareness, it often requires precise synchronization and robust signal alignment, which can be technically challenging. On the other hand, late fusion architectures process each modality independently to obtain intermediate interpretations (such as recognized words or detected hand poses), and then merge these at a semantic level. This approach tends to be more modular and easier to maintain, as each input channel can be improved or swapped without affecting the others.

Hybrid fusion combines elements of both early and late strategies, allowing certain low-level signals to be shared while keeping higher-level interpretation pipelines separate [**Nielsen2021**, **Oviatt1999**]. Choosing an appropriate fusion architecture is critical for ensuring fluid and error-tolerant interactions. For example, in an adaptive health app, fusing gaze and speech can enable users with motor impairments to confirm commands more easily. Similarly, in AR/VR settings, combining hand tracking with eye gaze supports natural object selection and manipulation.

2.6.2 Event Abstraction Models

Once input signals are fused, systems must convert them into abstracted events that represent user intent rather than raw movements or spoken words. Event abstraction models define a hierarchy of events, from primitive input (e.g., ”finger tap at position x,y”) to higher-level semantic commands (e.g., ”confirm selection”, ”scroll left”, or ”open menu”). This abstraction not only simplifies the logic needed to respond to different input combinations but also improves the system’s adaptability across platforms and devices. By mapping diverse physical actions to a common set of abstract commands, a single system can support a broad range of user abilities and contexts.

Recent work in multimodal interaction design has emphasized the importance of flexible and extensible event models that can incorporate new modalities without extensive re-engineering [**Bolt1980**, **Turk2014**]. Moreover, integrating AI-driven models, such as large language models or gesture classifiers, can further enrich the abstraction process by inferring user intentions from subtle or ambiguous input cues. Together, fusion architectures and event abstraction models form a foundation for building robust, adaptive, and inclusive multimodal user interfaces. As systems continue to evolve, these

techniques will play a crucial role in creating personalized and accessible experiences across devices and environments.

A notable example of semantic event abstraction in real-world interfaces is the work of Dixon et al. [**Dixon**], who implemented a general-purpose, target-aware pointing enhancement based on the Bubble Cursor. Their system uses pixel-level reverse engineering (via Prefab) to identify interface components and overlay interaction semantics onto them, allowing the system to interpret raw pointer movement as intent to interact with semantically meaningful targets even when underlying applications do not expose accessibility metadata. This layered approach of separating visual identification from interaction intent closely mirrors the goals of input event modeling in multimodal systems, especially in contexts where event boundaries are ambiguous or where UI elements are rendered outside standard toolkits.

2.7 LLMs as UI Controllers

Recent advances in large language models (LLMs) have opened new opportunities for bridging natural language and user interfaces. Traditionally, UI control has relied on explicitly designed event handlers and rigid APIs, requiring precise user input or structured interaction patterns. LLMs, by contrast, enable more flexible, high-level interactions that resemble natural human communication, which is especially promising for non-expert users or contexts where accessibility is crucial.

By leveraging LLMs' powerful capabilities in understanding and generating natural language, it becomes possible to control user interfaces using text or voice instructions without needing predefined UI-specific commands. For example, a user could ask an application to "show me my upcoming appointments and move the next one to next week," and an LLM-based controller can parse this instruction, map it to the appropriate UI actions, and execute them seamlessly.

This paradigm allows user interfaces to function more like intelligent agents rather than static interaction surfaces, reducing cognitive load and lowering the technical barrier for interaction. Furthermore, LLM-driven UI controllers can adapt to user-specific phrasing and preferences over time, learning from previous interactions to provide more personalized and efficient support.

2.7.1 Turning UIs into APIs

One promising approach to LLM-driven interfaces is conceptualizing user interfaces as implicit APIs. Instead of interacting with the UI through direct manipulation (e.g., clicking buttons, dragging elements), the UI's functionalities are abstracted into callable actions through an adapter layer that can be invoked through language. This "UI-as-API" perspective treats every interactive element and functionality as an endpoint or command that can be described and triggered textually. As demonstrated in recent research on program synthesis and UI automation like the UICoder referenced earlier [**Wu2024**, **chen2023programmaticui**], LLMs can be trained to translate high-level instructions into structured API calls or internal code representations. This enables a two-layered interaction model: the LLM interprets free-form user instructions and maps them onto the UI's abstracted actions, which are then executed to update the visible interface.

A significant advantage of this model is that it can help unify interaction modalities. Whether a command is given via voice, text, or even gesture-based language input, it is ultimately funneled into the same set of abstracted API calls. This makes interfaces more robust to modality switching and enhances accessibility. Moreover, turning UIs into APIs facilitates automation and integration with external services, enabling systems to not only react to individual user commands but also orchestrate multi-step tasks and workflows automatically. While challenges remain such as handling ambiguous instructions or ensuring robust error handling and LLM hallucinations, this direction shows strong potential for creating more adaptive, intelligent, and user-centered interfaces.

2.7.2 Agents

The rise of LLM-based agents represents a powerful evolution of user interface control beyond simple command mapping. Unlike static UI controllers, agents leverage LLMs to autonomously interpret, plan, and execute sequences of actions on behalf of users, effectively turning the interface into an intelligent collaborator rather than a passive tool. These agents can reason about user goals, maintain conversational context, and dynamically choose the best sequence of interface actions to fulfill complex or vague

instructions. For example, an agent might respond to "Help me prepare for my upcoming trip" by checking the user's calendar, suggesting packing lists, booking transportation, and even setting reminders all without requiring the user to explicitly navigate each step.

Recent systems such as OpenAI's GPT-based function calling, AutoGPT, or tools like Microsoft's Copilot and Google's Duet illustrate how agents can operate over complex applications, combining high-level reasoning with programmatic UI actions. In research, approaches like ReAct (Reasoning and Acting) frameworks show how agents can chain thought processes ("chain-of-thought") and API-level actions in iterative loops, enabling them to verify outcomes and adapt their behavior [**yao2022react**]. Moreover, LLM-based agents can incorporate user feedback continuously to refine their behavior, creating highly personalized assistants that align closely with individual preferences and working styles.

To further enhance their adaptability, modern agents increasingly integrate environment modeling and multimodal perception, enabling them to not only parse language but also interpret visual interfaces, sensor data, and user gestures. Frameworks like SeeAct and ViperGPT have demonstrated how combining vision-language models with action planning allows agents to operate UIs from visual input alone clicking buttons, reading menus, or navigating unfamiliar applications much like a human would. This opens doors to agent-driven interfaces for accessibility scenarios (e.g., voice or gaze-controlled UIs), remote control of complex software, or even autonomous use of standard desktop/web applications.

2.8 Health and Accessibility Applications

Health and accessibility applications represent some of the most impactful and socially significant domains for adaptive and intelligent user interfaces. As populations age and chronic health conditions become more prevalent, digital health tools are increasingly critical for supporting independence, self-management, and personalized care. Similarly, accessibility-focused interfaces help reduce barriers for users with diverse abilities, ensuring equitable access to technology. Modern health applications leverage multimodal inputs and adaptive interfaces to create more engaging and supportive experiences. For example, apps for physical rehabilitation frequently combine motion tracking (via cameras or wearable sensors) with adaptive visual feedback to guide patients through exercises safely and effectively. Examples include tools like Kaia Health for musculoskeletal therapy and Reflexion for cognitive and physical rehabilitation, which adjust exercise difficulty and feedback based on real-time performance data.

In the mental health domain, conversational agents and virtual coaches are becoming increasingly popular. Applications like Woebot or Wysa utilize natural language processing to provide immediate, conversational mental health support. By continuously adapting their conversational style and recommendations to the user's emotional state and progress, these systems illustrate the power of dynamic, user-centered design. Research has shown that such adaptive, agent-based interactions can improve adherence and patient outcomes [**fitzpatrick2017delivering**]. Accessibility applications also demonstrate the importance of personalized, context-aware interfaces. For users with visual impairments, screen readers and AI-powered image descriptions enable rich content access. Gaze-based or switch-based interaction systems empower users with severe motor disabilities to control complex interfaces using minimal input. Projects like Apple's VoiceOver and Microsoft's Seeing AI show how integrating multimodal AI into accessibility solutions can drastically improve daily usability and independence.

Furthermore, combining health and accessibility perspectives opens opportunities for fully personalized assistive systems. For instance, an intelligent multimodal interface could dynamically adjust text sizes and contrast for a user with low vision while also simplifying navigation for cognitive accessibility and providing voice guidance tailored to the user's speech patterns or preferences. Looking toward future developments, advances in virtual reality (VR) and immersive technologies can further enhance these assistive systems. In VR-based rehabilitation, for example, adaptive multimodal interfaces can create engaging, safe, and highly individualized therapy environments, dynamically adjusting exercise difficulty, feedback modalities, and visual cues to match each patient's needs and progress. Such systems show promise for applications ranging from motor skill recovery after stroke to anxiety reduction and pain management.

Chapter 3

System Design and Architecture

3.1 Introduction to System Design

The adaptive multimodal GUI framework developed in this thesis is designed to deliver personalised, accessibility-focused interface adaptations in real time. It builds on the idea that different users have different needs, and that these needs can change depending on context, device, and input modality. By combining multiple input channels such as touch, voice, and gestures with AI-driven reasoning, the framework can adapt interfaces in a way that is both responsive and context-aware.

The architecture follows a three-layer design. The frontend layer renders the user interface, captures user interactions, and applies adaptations as instructed by the backend. The input adapter layer standardises events across modalities into a common JSON schema, ensuring consistent processing regardless of their origin. The backend layer processes these events using Smart Intent Fusion (SIF), which combines rule-based logic with multi-agent LLM reasoning to generate targeted adaptations. This layered approach was chosen to ensure modularity, scalability, and extensibility for future modalities. By separating concerns across layers, the framework can easily integrate new input methods or adapt to different platforms without significant rework. Each layer can evolve independently, allowing for targeted improvements and innovations.

This chapter presents the framework at a conceptual level, focusing on its architecture, data flow, goals and key design principles. Detailed implementation aspects, such as widget properties, SIF, API routes, or database queries, are deferred to Chapters 4 and 5.

3.2 Overview of the System Architecture

The framework follows a three-layer architecture that was chosen and designed to be modular, scalable, and adaptable to diverse accessibility needs. Each layer has a clearly defined role and communicates with the others through a common JSON-based event and adaptation format. This separation of responsibilities makes it possible to extend or replace individual components without disrupting the rest of the system, and ensures that adaptations can be applied consistently across platforms and modalities.

At the top, the frontend layer is responsible for rendering the user interface, capturing interactions, and applying adaptations received from the backend. The input adapter layer sits between the frontend and backend, converting raw interaction data from multiple modalities into the shared JSON schema so that all events are processed in the same way. The backend layer implements the Smart Intent Fusion (SIF) process, combining event data, user profiles, and recent interaction history to produce personalised adaptation actions. The backend can rely on both deterministic rules and multi-agent LLM reasoning, described in detail in Chapter 4. The system is built around a feedback loop. Each user interaction is captured, processed, and logged along with the resulting adaptation. This log contributes to the user's profile and informs future adaptation decisions, allowing the interface to become progressively more personalised over time.

To illustrate this flow in practice, consider a motor-impaired user attempting to press a "Lock" button. If the tap misses its target, the event is sent through the adapter, recognised as a miss-tap, and forwarded

to the backend. SIF responds by generating an adaptation that increases the size and spacing of the button. The frontend receives this instruction and animates the change in real time. For a hands-free user, the process might begin with a spoken command to "turn on the lamp." The voice input is captured, standardised, and sent to the backend, which returns instructions to switch the UI into voice mode and activate the lamp control.

[Diagram Placeholder]

The high-level diagram of this flow (Figure X) shows the event path from the frontend to the backend and back again, using WebSocket connections for low-latency adaptations and HTTP for profile management and batch operations. Extensibility points are built into the architecture so that new modalities, reasoning methods, or frontend platforms can be introduced without altering the core event–adaptation pipeline.

Key Design Principles:

- **Modularity**: Layers can be swapped or upgraded independently.
- **Scalability**: Async processing and MongoDB indexing handle high interaction volumes.
- **Generalizability**: Platform-agnostic design enables deployment in domains from smart homes to healthcare.
- **Accessibility Focus**: All adaptations are guided by WCAG 2.1 and target motor-impaired, visually impaired, and hands-free users.

The most challenging aspect of this design is ensuring seamless communication between layers while maintaining low latency for real-time adaptations, as well as carefully integrating LLMs as a central component for processing and understanding user intents. This will bring challenges on its own such as hallucinations or misinterpretations of user input. Making sure the LLM accurately captures user intent and context is crucial for effective adaptations. Prompt engineering and careful design like fallback options will be essential to mitigate these issues.

3.3 Frontend Layer: UI Design and Interaction

The frontend layer is the user-facing component of the framework, responsible for rendering an adaptive and personalised interface that adjusts dynamically to user needs. Designed for cross-platform deployment, it captures multimodal interactions such as touch, keyboard, voice, or gestures and applies adaptations returned from the backend in real time.

In this thesis, the frontend is demonstrated through the Adaptive Smart Home Controller, a deliberately chosen example that offers both familiarity and a variety of interaction types. The smart home context provides a set of clear, relatable tasks such as switching on a light or adjusting a thermostat that can be adapted for users with motor, visual, or input-related impairments. This use case also highlights the framework's potential for deployment in other domains where accessibility is a priority.

3.3.1 Interface Elements

The frontend incorporates a set of core UI elements chosen to balance simplicity with the ability to demonstrate a wide range of adaptations. Together, they form a complete interface that could plausibly be used in real-world scenarios, while remaining portable across platforms. Conceptually, these elements can be grouped into three categories:

1. **Action controls:** for example buttons or sliders, which allow users to manipulate device states or settings.
2. **Information displays:** such as text labels, which convey device status or contextual feedback.
3. **Organisational elements:** such as cards or list views, which group related controls for clarity and ease of navigation.

These categories were selected because they cover the most common interaction and accessibility challenges. Action controls benefit from size and spacing adjustments for motor-impaired users, information

displays can be adapted with text resizing or high-contrast themes for visually impaired users, and organisational elements help reduce cognitive load by presenting related controls together.

3.3.2 Adaptation Levels

For this thesis and its implementation, adaptations in the frontend can be understood at three conceptual levels:

1. **UI-Level Adaptations:** which modify the appearance of visible elements to improve clarity or ease of interaction.
2. **Geometry-Level Adaptations:** which adjust layout and spacing to reduce input errors or simplify navigation.
3. **Input-Level Adaptations:** which alter the way users interact with the interface, for example by switching to a voice-driven mode.

While the precise mechanics of how these adaptations are implemented, triggered and applied are described in Chapter 5, the design principle remains the same: each change should be both functional and clearly communicated to the user, reinforcing trust and improving interaction efficiency. These high-level principles guide the implementation of the SIF framework, ensuring that adaptations are user-centered and context-aware.

3.4 Input Adapter Layer: Multimodal Input Processing

The Input Adapter Layer acts as the middleware between the user-facing frontend and the reasoning backend, it takes care of bidirectional communication between both layers. Its main role is to take raw interaction data from any modality such as touch, keyboard, voice, or gestures and transform it into a standardised format that the backend can process consistently and vice versa for the frontend. By separating input capture from input interpretation, this layer allows the rest of the system to operate independently of how the input was generated, making it easier to add new modalities in the future.

Central to this layer is the **JSON Event Contract**, architecturally this is exposed through the `AdaptiveUI Adapter` class, which defines the public interface for sending interaction data to the backend. The primary entry point is the `sendEvent(Event eventData)` method. This method accepts an `Event` object the framework's internal representation of a user interaction, and is responsible for enriching it with contextual information before forwarding it to the backend.

The `Event` class acts as the architectural contract for interaction data. While its exact schema is defined in the implementation (Chapter 5), at the design level it contains:

- **Event type:** the category of interaction, such as a tap, miss-tap, slider miss, or voice command.
- **User ID:** a unique identifier for the user interacting with the system.
- **Timestamp:** the time when the event occurred.
- **Source modality:** the origin of the event (touch, keyboard, voice, gesture, etc.).
- **Target element:** the UI element or control associated with the event.
- **Coordinates:** spatial information where applicable, for example the location of a tap.
- **Confidence score:** a value indicating the certainty of the detected intent.
- **Metadata:** optional context such as UI element type (e.g., button, slider) and more.

By enforcing this structure, the adapter guarantees that all events passed to the backend follow the same rules, whether they come from Flutter, SwiftUI, or a future Unity-based frontend.

The adapter also handles profile verification and creation before sending events. This step ensures that the backend always has a corresponding user profile for contextual reasoning. If the profile does not exist, the adapter triggers its creation using default accessibility settings or a new user profile. These profiles are also bound by a JSON contract to ensure consistency, this design is described in chapter 4. Furthermore, for low-latency adaptation feedback, events are transmitted via a persistent WebSocket connection, while non-real-time operations such as profile updates, HTTP endpoints are used instead.

Lastly, the adapter manages adaptations returned by the SIF backend layer by listening to the WebSocket channel. When the backend sends adaptation instructions which is represented in its own JSON contract (in more detail described later), the adapter maps it to an internal representation based off of the JSON contract that can be easily processed by the frontend. This Adapter pattern design is deliberately chosen and platform-agnostic. Although the current implementation is in Dart for Flutter, the architectural pattern can be replicated in any language or platform by implementing the same `sendEvent()` contract and adhering to the same event object structure. This separation means that adding a new modality, such as eye tracking or brain-computer interfaces, only requires implementing the modality's input capture and mapping it to the existing `Event` format, no changes are needed to the backend or reasoning logic.

3.5 SIF Backend Layer: Smart Intent Fusion (SIF)

The SIF backend layer is the reasoning core of the framework, responsible for turning raw user interactions and contextual information into targeted UI adaptations. Operating behind the input adapter, it receives events in the standard JSON contract format and turns them into an internal structured representation (`Event` object), combines them with the user's profile and recent interaction history. This forms the basis for the LLM prompts, next it determines the most appropriate changes to apply to the interface using LLM reasoning.

From an architectural perspective, this layer has two main responsibilities:

- **Processing and interpretation:** validating incoming events, interpreting their intent, and prioritising them according to context.
- **Adaptation generation:** producing a set of structured adaptation actions in a JSON contract that the frontend can apply directly.

The backend is designed to support multiple reasoning strategies. In its current form, it combines deterministic rules with a multi-agent LLM process called Smart Intent Fusion. Rules handle straightforward accessibility needs; for example, increasing button size after a miss-tap, while the LLM process enables more context-aware adaptations that consider multiple factors simultaneously.

Although the underlying logic is covered in detail in Chapter 4, the architectural position of this layer is central in this framework: it serves as the decision-making hub, fed by standardised events from the input adapter, and returning validated adaptations to the frontend in near real time. This separation allows the reasoning configuration to be changed easily; for example, by swapping models, refining prompts, or integrating new agents without changing the structure of the rest of the framework.

3.6 User Profiles and Context Modeling

User profiles form the backbone of the framework's personalisation capability. They store accessibility preferences, interaction patterns, and contextual data that allow the system to adapt the interface to an individual's needs over time. The profile is not a static record, it evolves as the user interacts with the system, incorporating both explicit configuration and implicit observations from their behaviour.

Architecturally, the profile contains three types of information:

- **Static attributes:** such as preferred font size, contrast settings, or dominant input modality, which may be set during initial onboarding.
- **Learned preferences:** derived from patterns in the user's interactions; for example, frequent miss-taps on small controls may trigger a persistent increase in their size.
- **Contextual data:** including recent interaction history and device environment details, which help the backend reason about the most appropriate adaptations in a given moment.

When a new event is processed, the backend combines it with the relevant profile and context data before passing it to the reasoning logic. This ensures that adaptations are not just reactive to the most recent input, but also informed by longer-term patterns and situational factors.

The architecture treats profiles as a shared resource between all layers:

- The **input adapter** ensures that the correct user profile is referenced with every event, as well as creating or updating the profile when necessary.
- The **backend** updates profiles automatically as adaptations are applied or feedback is received.
- The **frontend** can query profile data to adjust default UI settings before any adaptations are applied.

By integrating profile and context information into every stage of the adaptation pipeline, the framework moves beyond static changes and supports continuous, data-driven personalisation. The specific data schema, storage mechanisms, and update strategies are described in detail in Chapters 4 and 5.

3.7 Dynamic Adaptation Mechanisms

The dynamic adaptation mechanisms are the part of the framework responsible for translating reasoning outputs from the backend into real-time, personalised changes for the user interface. Operating within the SIF backend layer, they take standardised events from the input adapter, combine them with profile and context data, and decide on the most suitable adaptation. These decisions are returned as structured actions to the input adapter, which maps the JSON contract to an internal representation that is then applied immediately by the frontend using callbacks.

At the architectural level, the adaptation process follows a continuous feedback loop. Every interaction is captured, processed, and logged, with the resulting adaptations influencing how the interface behaves in the future. This allows the system to progressively refine its responses, ensuring that changes are not only reactive but also shaped by longer-term usage patterns.

3.7.1 Adaptation Pipeline

The adaptation pipeline follows a structured lifecycle from input capture to UI update, and can be viewed as four main stages:

1. **Event reception and context gathering:** Inputs from any modality are captured, standardised, and sent to the backend. The backend retrieves the associated user profile and recent history to build a complete context for reasoning.
2. **Intent interpretation:** The backend analyses the combined event and context, using both deterministic rules and LLM-based reasoning to infer the most likely user intent.
3. **Adaptation Generation:** A set of structured actions is produced, describing changes to be applied to the interface. These may involve modifying the visual presentation, adjusting interaction geometry, or altering input modes. SIF employs:
    - **Rule-Based Logic:** Deterministic rules provide fast, reliable adaptations for common scenarios (mostly used for a reliable backup).
    - **LLM Reasoning:** The AI model infers complex intents and generates creative adaptations for the user.
    - **Heatmap Analysis:** Simulated via history counts, prioritizes problems with frequently interacted elements (e.g., repositioning elements after multiple taps).
4. **Application and logging:** The frontend applies the adaptations immediately, while the backend logs both the event and its outcome for future learning.

3.7.2 Supported Adaptation Actions

The framework supports a defined set of adaptation types that cover common accessibility needs. These include scaling or spacing adjustments for improved touch accuracy, font and contrast changes for better readability, and modality switches for hands-free interaction. The action set is deliberately kept broad enough to handle varied contexts, yet constrained enough to ensure reliable rendering across platforms.

3.7.3 Continuous Learning and Feedback Loop

A central design principle is that adaptations are not static changes. The system maintains a history of recent interactions for each user, enabling it to identify recurring patterns such as frequent miss-taps or repeated modality switches. When such patterns are detected, the backend can suggest persistent adjustments; for example, permanently enlarging a frequently used UI element, reducing the need for smaller repeated actions.

3.7.4 Design Considerations

- **Accessibility Focus:** All supported actions are chosen to address motor, visual, and input-related impairments, ensuring that adaptations enhance rather than complicate interaction.
- **Real-Time Performance:** Low-latency communication ensures that adaptations are applied quickly enough to feel seamless.
- **Reliability:** Rule-based logic ensures that adaptations continue to function even if LLM-based reasoning is temporarily unavailable.
- **Extensibility:** The action set and event format are flexible enough to incorporate new adaptation types and modalities in future deployments.

By combining immediate, context-aware changes with a persistent feedback loop, the dynamic adaptation mechanisms give the framework its ability to evolve alongside the user's needs, making it more effective over time.

3.8 Chapter Summary

This chapter has presented the system design and architecture of the multimodal AI-driven framework for dynamic UI adaptation, focusing on its ability to deliver personalised, accessibility-oriented solutions for motor-impaired, visually impaired, and hands-free users. The framework is built around a modular three-layer architecture: the Frontend Layer, Input Adapter Layer, and SIF Backend Layer that enables the seamless integration of multiple input modalities, including touch, voice, and gestures, and the delivery of real-time UI adaptations.

The architectural design emphasises modularity, scalability, generalisability, and accessibility, creating a solid foundation for extension into other domains such as healthcare and gaming. Communication between layers is handled through WebSocket for low-latency updates and HTTP for reliable profile and debugging operations, while MongoDB and a standardised JSON contract ensure scalability and flexibility in storing and processing interaction data. Together, these elements provide a framework capable of addressing real-world accessibility needs today.

Chapter 4

Smart Intent Fusion (SIF)

4.1 Introduction to Smart Intent Fusion

Smart Intent Fusion (SIF) is the central intelligence layer of the proposed multimodal AI-driven UI adaptation framework, responsible for transforming raw, heterogeneous user input signals into concrete, context-aware interface adaptations. Where the Frontend Layer renders the UI and the Input Adapter Layer standardises events, SIF performs the reasoning step. Architecturally, SIF occupies the central position in the adaptation loop, receiving standardised events from the Input Adapter Layer, combining them with user profile and context data, and returning validated adaptations back to the frontend for immediate application. This placement ensures that every adaptation decision is both context-aware and modality-agnostic, allowing the framework to maintain consistent behaviour regardless of how the interaction was initiated. it fuses current interaction data with user profiles, accessibility requirements, and recent interaction history to infer the user's underlying intent and translate this into actionable UI changes.

The motivation for SIF stems from a simple but critical challenge in accessibility-focused HCI: users rarely interact with a system through a single, perfectly clean input channel. Instead, interactions are often multimodal, noisy, and incomplete. A motor-impaired user may miss-tap a button but also issue a supporting voice command. A visually impaired user may attempt to activate a control by gesture but with low confidence, relying on high-contrast cues to complete the action. Traditional rule-based adaptive systems tend to process these signals independently, missing the opportunity to combine them into a unified, more reliable understanding of the user's goal.

SIF addresses this gap through a hybrid reasoning approach:

- Rule-based logic handles deterministic adaptations for example, "if miss_tap on target → increase size by 1.5×" ensuring baseline accessibility support and fast response times even without AI availability.
- LLM-driven reasoning can process richer multimodal context and can propose creative or proactive adaptations that go beyond fixed rules such as switching to voice mode after repeated input struggles, or combining voice + gesture input to trigger a button instantly.
- Multi-Agent SIF (MA-SIF) extends this further by distributing reasoning across specialised LLM agents (UI, Geometry, Input or more) and validating results through a dedicated Validator Agent to reduce hallucinations and conflicting actions as well ensuring the necessary validation.

The SIF layer sits behind well-defined APIs:

- WebSocket endpoint `/ws/adapt` for low-latency, real-time adaptation suggestions.
- HTTP endpoints like `/profile` for profile management, or `/full_history` for developer tooling and debugging.

These endpoints use a strict JSON contract, making it easy for any frontend platform to connect and work with the backend. This ensures that the system remains flexible and can handle different input modalities.

A defining feature of SIF is its integration with persistent user profiles stored in MongoDB. These profiles encapsulate three layers of information: static attributes such as preferred font size or contrast mode, learned preferences derived from recurring interaction patterns, and contextual data including recent history and environmental details. By merging these dimensions with each incoming event, SIF can make decisions that are not only responsive to the user's current action but also aligned with their long-term accessibility needs. By fusing this profile data with incoming events, SIF maintains a continuous personalisation loop progressively adapting the UI to match the user's abilities and context over time.

For example:

- A profile with `motor_impaired: true` will bias adaptations towards larger controls and simplified layouts.
- Repeated history of slider misses may lead to permanent slider thumb enlargement.
- A hands-free preference can automatically promote voice/gesture-driven navigation in relevant contexts.

In the context of this thesis, SIF is not only an internal backend feature, it is the core research contribution. The remainder of this chapter expands on the theoretical underpinnings of SIF, its integration with user profiles and multimodal fusion, the prompt engineering strategies for guiding LLM behaviour, the architecture of its multi-agent extension, and the performance metrics used to evaluate its effectiveness.

4.2 Theoretical Foundations of Smart Intent Fusion

The idea behind Smart Intent Fusion is simple:
users don't interact with a UI in one clean and perfect way. In real life, inputs are messy, mixed, and often incomplete. A person might tap the wrong spot on a button, then say a voice command to make sure it worked. Someone using gestures might point at something but not be perfectly accurate, so they rely on extra visual cues to finish the task. A lot of current adaptive systems still process these signals separately, and that means they miss the chance to combine them into a clearer picture of what the user actually wanted.

Smart Intent Fusion tries to fix that by taking all the inputs together: touch, keyboard, voice, gestures and mixing them with what the system already knows about the user (their profile, their history of interactions, and their accessibility needs). This way, it can guess the real intent and adapt the UI in the most helpful way.

4.2.1 Multimodal Fusion

In HCI, "multimodal fusion" just means combining different types of input to get a more reliable or richer understanding of the user's action.
This can be done in a few ways:

- **Early fusion**: merge the raw signals before interpreting them (e.g., combining the coordinates of a tap with the audio of a voice command immediately).
  → Useful when two inputs happen at the exact same moment.
- **Late fusion**: process each input type separately first, then merge the interpreted results (e.g., "voice command = unlock" + "miss-tap near unlock button" → final decision = trigger Unlock button).
  → This is where most SIF reasoning happens.
- **Hybrid fusion**: a mix of both, sharing some data early but also combining results later.
  → Used when one modality's data can make another modality's interpretation more accurate.

SIF uses the hybrid approach. The Input Adapter Layer already standardises each input into a clean JSON format, but the fusion step happens in the backend where the current event, past history, and profile are all processed and fused together. This makes it possible to combine patterns like "miss-tap + voice command" into one clear adaptation. In the overall architecture described in Chapter 3, this fusion step is the bridge between raw input handling and adaptation generation. By sitting in the backend, SIF can remain entirely modality-agnostic while still benefiting from the structured event format defined

by the Input Adapter Layer. This ensures that even when new input methods are introduced in the future, the fusion logic can remain unchanged, relying on the same JSON schema and profile-context pipeline.

4.2.2 Intent Inference

The main goal of SIF is not just to log inputs, but to figure out *why* the user did them. This is called intent inference. In other words, we want the system to answer the question: "What was the user trying to do?" If the system knows the intent, it can choose the best adaptation. The inference process is always profile-aware. Each decision is informed by a combination of static preferences (such as font size or contrast mode), learned patterns from past usage (such as repeated difficulty with sliders), and short-term contextual data (such as the most recent interactions). This means that two users producing the same input sequence could receive different adaptations if their profiles and histories differ, keeping the interaction completely personalised.
For example:

- If the intent is to press "Unlock" but the user misses, the adaptation could be to enlarge the button and trigger it right away.
- If the intent is to adjust a thermostat but the user struggles with the slider, the adaptation could be to switch to voice control and increase the slider size.

This is where LLMs can help. They are good at reasoning about context, combining clues, and filling in gaps when inputs are unclear. The downside is that they can be slow, be costly to run, and sometimes "hallucinate" an answer that doesn't make sense even if its not defined in the prompt. That's why SIF uses a **hybrid** approach:

- **Rules** handle clear, simple cases with instant feedback.
- **LLM reasoning** handles more complex or ambiguous cases.

4.2.3 Why Hybrid Works Best

A purely rule-based system is predictable but rigid. It can only respond to situations that were thought of in advance. A purely LLM-based system is flexible but not always reliable, especially when it needs to work in (close to) real time for accessibility. By combining both:

- The rules guarantee that basic accessibility changes (like increasing size after a miss-tap) always work. For example:

```
IF event_type == "miss_tap" AND profile.motor_impaired == true
THEN action = increase_button_size(target, 1.5)
```

- The LLM adds creativity and can adapt to situations the rules didn't cover, like combining unusual input patterns, proposing a mode switch for hands-free use or even something entirely new.

4.2.4 Connection to Accessibility

The whole point of this is to improve accessibility for different kinds of users:

- **Motor-impaired:** combine multiple input signals to avoid repeated failed attempts.
- **Visually impaired:** recognise when visual feedback is not enough and trigger higher-contrast or bigger fonts automatically.
- **Hands-free:** allow combinations like voice + gesture to instantly activate actions.

In short, the theoretical base for SIF comes from multimodal fusion, intent inference, and hybrid reasoning. These ideas are not new in HCI, but this framework applies them with a strong focus on accessibility and personalisation, and makes them work in real time with cross-platform UI code. For example, a motor-impaired user who misses a lock button twice and then issues a voice command might trigger a combined adaptation: the UI immediately enlarges the button for future taps, but also switches the interface to voice-first mode for the current session. This ability to layer short-term fixes on top of long-term

adjustments is what makes SIF more than just a reactive system, it is a continuously learning adaptation layer.

4.3 User Profile and Context Integration

For Smart Intent Fusion to be truly "smart," it needs more than just the current event it is processing. If SIF reacted to every tap, voice command, or gesture without knowing who the user is or how they usually interact, it would behave like a generic accessibility script rather than a personalised adaptation system. That's why the user profile and interaction context form the backbone of the reasoning process. They give SIF a sort of memory/personality, and the ability to adapt over time, not just in the moment. When a new event comes in like a tap, miss-tap, voice command, or gesture, SIF doesn't look at it in isolation. It combines that event with:

1. **The user profile** – a stored record in MongoDB with accessibility flags, preferred modalities, and UI settings.
2. **Interaction history** – the last 10 events that show patterns or repeated problems.
3. **Current UI context** – optional metadata about what's on screen, where buttons are placed, and their sizes.

This means SIF can make decisions that are personal and context-aware, not just reactive.

4.3.1 User Profiles

A user profile stores the information that makes one person's interaction style different from another's. This can include accessibility needs (motor-impaired, visually impaired, hands-free preferred), preferred input modalities, and baseline UI settings like font size, contrast mode, and button scale. It can be seen as the **memory** of the system. When combined with a short history of recent interactions, this profile turns SIF from a static decision engine into a continuous learning system.

Without profiles, SIF could still make adaptations, however they would always be reactive and temporary. For example:

- If a user with tremors keeps missing a button, the button might get enlarged for that session, but as soon as they restart the app, it would shrink back.
- If a user prefers voice input, SIF wouldn't know to automatically switch to voice mode when they struggle with touch, instead it would have to come up on its own that this switch is needed for this user.

Profiles ensure these adaptations stick and get better over time.

4.3.2 User Profile Structure

In the backend, each profile is a JSON document in the `profiles` collection of a MongoDB database. It's indexed by `user_id` so the system can look it up instantly whenever a new event arrives. A typical structure looks like this:

Listing 4.1: Simplified User Profile Example

```
{
  "user_id": "user_123",
  "accessibility_needs": { "motor_impaired": true },
  "input_preferences": { "preferred_modality": "voice" },
  "ui_preferences": { "font_size": 16 },
  "interaction_history": [
    { "event_type": "miss_tap", "target_element": "lamp" }
  ]
}
```

The full schema with all supported fields and metadata is given in Chapter 5.

This design keeps it simple but flexible. Architecturally, each profile contains:

- `accessibility_needs`: flags that tell SIF what kind of adaptations to prioritise.
- `input_preferences`: helps the system decide which modality to switch to when needed.
- `ui_preferences`: baseline visual parameters such as font size and contrast mode.
- `interaction_history`: capped log of recent events to support continuous learning.

MongoDB's indexing means the profile can be retrieved in milliseconds, even with a large user base, and capped histories ensure lookups are fast.

4.3.3 How Profiles Affect Decisions

When an event comes in, the backend follows a clear process:

1. Load the profile from MongoDB using the `user_id`. If it doesn't exist yet, create a new default profile.
2. Combine the event with the last few interactions from the history.
3. Pass the profile, history, and current event into the Smart Intent Fusion reasoning step.

This context completely changes how SIF decides on adaptations.

Some examples of influenced decisions:

Example 1: Motor-impaired user with repeated miss-taps
If the last three events in history are miss-taps on the same "Unlock/Lock" button, and `motor_impaired` is true in their profile, SIF might do two things at once:

Listing 4.2: Possible motor-impaired user adaptations

```
[
  {"action": "increase_button_size", "target": "button_unlock", "value": 1.5, "reason": "Multiple miss-taps detected, enlarging button for better accessibility"},
  {"action": "highlight_border", "target": "button_unlock", "reason": "Increase button visibility for the user"}
]
```

Without the profile, it might have just enlarged the button once and moved on.

Example 2: Hands-free preferred user A user with `"hands_free_preferred": true` points at a device card (gesture) and says "turn on the lights." SIF reasoning can fuse these into:

Listing 4.3: Hands-free user intent fusion

```
{
  "action": "trigger_button",
  "target": "button_light",
  "reason": "Gesture pointing + voice 'Turn on the lights' detected for hands-free user",
  "intent": "Activate Lights"
}
```

Because of the profile, SIF is confident enough to trigger the button immediately without asking for physical confirmation.

4.3.4 Continuous Learning from History

Profiles are not static. Every interaction is logged in the `interaction_history` and can influence future decisions. The interaction context can be seen as the system's **short-term awareness**. This makes SIF a learning system:

- If a user keeps manually enabling high-contrast mode, the system can update `contrast_mode` in their profile so it's always on by default.
- If increasing a button size significantly reduces miss-taps, that size can become the new permanent baseline in `ui_preferences`.
- If switching to voice mode solves repeated touch struggles, the profile can be updated to favour voice by default.

This feedback loop means the user doesn't need to "train" the system manually, it adapts naturally as they use it. If SIF only looked at the current event, it would miss important patterns and make short-sighted decisions. With profile + history + event combined:

- Adaptations can be proactive instead of reactive.
- The UI can stay consistent between sessions.
- The system can learn what really helps the user over time.

From an accessibility perspective, this is the difference between a generic interface that occasionally helps and a personalised tool that feels like it understands the user.

4.3.5 Role in Accessibility

User profiles are the backbone of accessibility-focused adaptations in this framework. They act as a persistent memory of the user's abilities, preferences, and interaction challenges, enabling the system to make targeted, proactive adjustments rather than relying solely on short-term reactive changes. By storing explicit accessibility flags alongside learned behavioural patterns, profiles allow SIF to tailor the interface to an individual user in ways that are both short-term and long-term.

From an accessibility perspective, profiles influence SIF's reasoning in three key user categories:

- **Motor-Impaired Users:** Profile flags such as `motor_impaired: true` prioritise adaptations that reduce fine motor precision requirements. This can include:
  - Enlarging touch targets (buttons, sliders) and increasing spacing to prevent accidental taps.
  - Offering alternative modalities such as voice commands or keyboard navigation to bypass touch interaction altogether.
  - Retaining enlarged target sizes across sessions once repeated miss-taps are detected.
- **Visually Impaired Users:** When `visual_impaired: true` is set, adaptations aim to maximise visual clarity. Examples include:
  - Switching to high-contrast themes and bold colour schemes to align with WCAG 2.1 contrast requirements.
  - Increasing font sizes and icon scales to meet text accessibility guidelines.
  - Highlighting the active element with a strong focus border or magnified overlay to improve navigation feedback.
- **Hands-Free Users:** Profiles with `hands_free_preferred: true` bias SIF towards non-touch input modes, reducing physical interaction demands. Adaptations may involve:
  - Automatically switching to voice or gesture navigation when interaction struggles are detected.
  - Providing clear, speech-friendly UI labels and tooltips to improve command recognition accuracy.
  - Simplifying layouts by reducing the number of visible controls at once, making it easier to select elements through voice or gesture.

By embedding these accessibility considerations directly into the profile structure, SIF can reason in a way that is both context-aware and user-specific. This allows the system to:

1. Anticipate needs before errors occur (e.g., pre-emptively enlarging critical controls for a motor-impaired user on a small screen).

2. Ensure adaptations persist across sessions, avoiding the frustration of having to reconfigure accessibility settings each time.
3. Combine profile knowledge with real-time interaction patterns, enabling nuanced decisions such as *"keep high contrast on by default, but also enable voice mode when the user is multitasking"*.

4.4 Modeling Multimodal Input Fusion

Smart Intent Fusion doesn't just take one input, it collects **multiple inputs from different modalities** like touch, voice, gestures, and more. Then fuses them into a single, well-reasoned adaptation. This process is called *multimodal input fusion*. Without fusion, the system would treat each event separately. A miss-tap would trigger one adaptation, and a voice command would trigger another, without realising both were aimed at the same action. With fusion, those two inputs can be combined into one confident and more helpful response which saves time and reduces frustrations. Furthermore, the user profile and interaction history also directly affect how this fusion works. A hands-free preferred user will have a lower threshold for fusing gesture + voice, but a visually impaired user's profile might cause SIF to always add a "highlight" adaptation when triggering elements via voice, even if not strictly necessary. If history shows repeated failures for a certain modality, its weight in fusion decisions can be reduced. For users with impairments, every extra action is extra effort. Because of how the fusion is designed, it cuts down on unnecessary steps so the UI adapts faster and smarter to the user's needs.

4.4.1 Event Standardisation

Before any fusion can happen, the raw input needs to be standardised. Every frontend in the framework Flutter, SwiftUI or a future VR client converts its local input data into the same JSON contract. This is handled by the Input Adapter Layer as described earlier.

A standardised event looks like this:

Listing 4.4: Standardised Event Example

```
{
  "event_type": "miss_tap",
  "source": "touch",
  "timestamp": "2025-08-04T14:41:00Z",
  "user_id": "user_123",
  "target_element": "button_unlock",
  "coordinates": {"x": 210, "y": 640},
  "confidence": 0.8,
  "metadata": {"UI_element": "button"}
}
```

This contract ensures:

- **Cross-platform compatibility**: all clients speak the same "language" to the backend.
- **Dynamic field names**: fields like `event_type` and `target_element` can be extended and modified without breaking existing logic.
- **Extensibility**: we can add new modalities (like gaze tracking) without breaking existing clients.

By the time the event reaches SIF, it doesn't matter whether it came from a phone, desktop, or VR headset, it always adheres to the same format for the backend. As said earlier SIF uses a hybrid approach for multimodal input fusion. The Input Adapter handles basic pre-processing (similar to early fusion), but the actual reasoning, deciding what adaptations to apply happens in the backend using late fusion.

Example: Touch + voice
Let's say a motor-impaired user taps just to the right of the "Lock/Unlock" button and, within a second, says "Unlock." Individually, the miss-tap could trigger an enlargement of the button and the voice command could trigger the unlocking action. However, with multimodal fusion, the system can understand that both inputs are related and prioritize the unlocking action while also enlarging the button for better accessibility.

With SIF fusion, the system sees both in context:

Listing 4.5: Example of Multimodal Fusion

```
[
  {"action": "increase_button_size", "target": "button_unlock", "value":
      1.5, "reason": "Miss-tap detected near Unlock button"},
  {"action": "trigger_button", "target": "button_unlock", "reason": "
     Voice command 'unlock' detected in combination with miss-tap"}
]
```

Now the button is both enlarged for future use and triggered immediately, reducing the number of actions the user needs to take, simplifying the interaction which is useful for motor-impaired users.

4.4.2 Timing and Confidence

SIF's LLM reasoning can also consider **when** and **how confidently** an input happened based on two fields from the standardised event:

- **Timing**: events close together in time (e.g., within 1–2 seconds) are more likely to be related.
- **Confidence**: each modality can provide a confidence score (e.g., gesture detection might be 0.7 certainty). Lower confidence might require a second modality before acting.

For example, a low-confidence gesture to point at a button might do nothing alone, but if followed by a high-confidence voice command naming that button, SIF can treat them as a combined intent.

4.4.3 LLM Reasoning in Fusion Decisions

While the fusion process benefits from deterministic rules for speed and reliability, one of the key innovations of SIF is its ability to leverage LLM reasoning to interpret and combine multimodal inputs in a context-aware way.

Once standardised events reach the backend, they are not processed in isolation. Each event is enriched with:

- **User profile data:** accessibility flags, preferred modalities, and baseline UI settings.
- **Interaction history:** the most recent events, revealing repeated struggles or patterns.
- **Contextual metadata:** details about the UI state (e.g., which elements are visible), UI element type, environment and more.

This enriched dataset is then included in the LLM prompt, allowing the model to reason comprehensively about the user's intent across modalities. For example, instead of treating:

- a low-confidence "point" gesture at a thermostat slider, and
- a voice command "set temperature to 22"

as separate actions, the LLM can infer that they describe the same goal and produce a single adaptation:

```
{
  "action": "adjust_slider",
  "target": "slider_thermostat",
  "value": 22,
  "reason": "Gesture and voice command combined to adjust temperature"
}
```

By bringing together different types of input modalities, user's preferences, and their recent actions, SIF can suggest changes that are not only accurate but also anticipate what will help the user most. This means the system can adapt quickly and make the interface easier to use based on what the user's most probable intent was.

4.5 Rule-Based Logic and LLM-Driven Adaptation

Smart Intent Fusion uses two very different ways to decide what adaptation to apply:

- **Rule-based logic** handles clear, deterministic cases where the system can apply a known adaptation based on the event type and user profile. It is instant, and predictable.
- **LLM-driven reasoning** uses the Gemini API to process complex, multimodal contexts and propose creative adaptations that go beyond simple rules.

Both have strengths and weaknesses, which is why the framework combines them instead of choosing one.

4.5.1 Rule-Based Logic

Rule-based logic works by matching incoming events to predefined conditions and applying a fixed response.
For example:

```
if event.event_type == "miss_tap":
    return {"action": "increase_button_size", "target": event.
        target_element, "value": 1.5}
```

Advantages:

- **Simplicity**: Easy to implement and understand.
- **Speed**: Instantaneous responses to known events.
- **Predictability**: Consistent behavior for similar inputs and no risk of unintended consequences like hallucinations.
- **Baseline guarantee**: Acts as a safety net so that critical accessibility features still work if the LLM is unavailable, slow to respond, or returns unusable output.

Limitations:

- Can only handle cases explicitly programmed in advance.
- No ability to combine signals in creative ways.
- Doesn't learn new patterns unless a developer updates the rules.

This means that while rule-based logic is fast and reliable, it can also be rigid and unable to adapt to new situations without human intervention. In SIF, they are deliberately kept lightweight, mostly as a mock or backup layer for LLM-driven reasoning.

4.5.2 LLM-Driven Reasoning

The LLM can reason about the event, user profile, and history together to try and infer the user's intent more deeply. It can make connections that rules would miss, such as:

- Combining a miss-tap with a voice command into a single action.
- Switching to a different modality when it detects repeated failure in the current one.
- Proposing multiple coordinated adaptations for one intent.

Advantages:

- **Flexibility**: Can adapt to new situations without explicit programming by the developer.
- **Context-aware**: Takes profile and history into account naturally.
- **Learning**: Can improve over time by learning from user interactions and feedback.

Limitations:

- **Less predictable**: May generate unexpected or irrelevant responses.

- **Slower**: Network call + reasoning time.
- **Needs validation**: Output must be checked more thoroughly before applying.

4.5.3 Hybrid Approach in SIF

In practice, SIF doesn't fully choose between the two, it blends them:

1. **Rules first**: If the event matches a clear, high-confidence rule, apply it immediately, this is mostly done in the frontend by the user profile.
2. **LLM second**: Use the model for complex or ambiguous cases, or to suggest extra adaptations beyond the rules.
3. **Timeout Fallback**: If the LLM times out or fails, return rule-based or other LLM output only.

This ensures that basic accessibility features always work, while still allowing for creative, context-aware adaptations when needed. In other words, the rules form the "floor" of the system (the minimum guaranteed level of accessibility) while the LLM can raise the ceiling by adapting to more complex, ambiguous, or novel situations. Users are never left waiting for AI responses that might never come, and the system remains responsive even in worst-case scenarios.

4.6 Multi-Agent Smart Intent Fusion (MA-SIF)

While a single LLM can process events and suggest adaptations, it often tries to "do everything" at once. That makes it harder to constrain, more prone to hallucinations, less predictable and in some cases slower (depending on workload). Multi-Agent Smart Intent Fusion (MA-SIF) solves this by splitting the reasoning into specialised agents, each focused on one domain of UI adaptation, and then combining their outputs through a Validator Agent. This design brings the benefits of modularity, parallelism, and role-specific constraints, all of which improve reliability and make the system easier to maintain.

4.6.1 Why Multiple Agents?

When one model is asked to handle UI changes, geometry adjustments, input mode switching, and validation all at once, several problems appear:

- The prompt becomes long and vague.
- Allowed actions become harder to enforce and turn into hallucinations.
- Reasoning gets scattered between unrelated concerns.

By giving each agent a clear role, their prompts can be short, specific, and easy to maintain. For example, the UI agent only ever sees actions like `increase_font_size` or `increase_contrast`, while the Geometry agent only deals with spatial changes like `increase_button_size` or `adjust_spacing`. This separation means that each agent can focus on its specific task without being distracted by unrelated concerns. Parameters can be tuned independently for each agent, allowing for more precise control over their behavior (e.g., lower temperature for Geometry, slightly higher for Input). Finally, debugging becomes easier as each agent's logic is contained and easier to follow.

4.6.2 Agent Roles

At a high level, MA-SIF in this thesis's current configuration, consists of four specialised LLM agents:

1. a **UI Agent** for visual and interactive adaptations,
2. a **Geometry Agent** for spatial changes and layout simplification,
3. an **Input Agent** for modality switching and interaction simplification,
4. a **Validator Agent** for conflict resolution and accessibility compliance.

Each has a narrow focus, defined prompts, and a limited set of allowed actions, making their reasoning predictable and easier to validate. They have clearly defined scopes of responsibility, and their roles are deliberately narrow to keep prompts concise, outputs predictable, and debugging straightforward. **The**

UI Suggestion Agent is concerned purely with visual accessibility changes such as increasing font size, toggling high-contrast mode, or displaying contextual tooltips. For example, when a visually impaired user interacts with a dense text block, this agent might output an adaptation like increasing the global font scale by 1.2×.

The **Geometry Suggestion Agent** deals with spatial layout changes and the physical dimensions of interactive elements. It might recommend increasing the size of a button, expanding the hit area of a slider, or adding extra spacing between cards in a list. For example, a motor-impaired user who repeatedly misses a button may trigger an output to enlarge that specific button by 1.5× while keeping the rest of the interface unchanged.

The **Input Suggestion Agent** focuses on modality switching and simplifying interaction pathways. It is capable of suggesting transitions between touch, keyboard, voice, or gesture modes, as well as proposing layout simplifications to reduce cognitive load. For instance, if a user struggles with touch but succeeds with voice commands, this agent can suggest switching to voice-first navigation, potentially combined with a reduced interface complexity.

Finally, the **Validator Agent** operates after all others have completed their reasoning. This is the most computationally demanding role, as it must examine every proposed adaptation in detail, identify and remove duplicates, verify that all actions are allowed, and ensure that no output falls outside safe parameter ranges. It also resolves potential conflicts, such as two agents targeting the same element with incompatible values. Because of the breadth of this responsibility, the validator uses a larger Gemini model, a higher timeout of 30 seconds, and a dynamic thinking budget.

4.6.3 Adaptation Flow

When an event arrives at the backend, the fusion process begins by loading the user's profile and recent interaction history from MongoDB. This contextual data is then sent to each non-validator agent, alongside the event itself. The prompt for each agent is tailored to its domain, ensuring that the LLM only receives relevant instructions and the list of actions it is permitted to output.

Once each agent has processed the event, their suggestions are collected. Importantly, the system is tolerant of partial failures, if one or more agents fail to return a result due to a timeout or API error, the remaining agents' outputs are still retained. These partial results are then passed to the Validator Agent, which merges them into a coherent and conflict-free final adaptation set. Another possibility is fusing LLM adaptations with rule-based suggestions to create a hybrid output when one or more agents fail.

If the validator itself fails, either due to malformed output or a processing error, the system does not discard all results. Instead, it falls back to returning the raw, unvalidated suggestions from the agents since the output may still be more useful than rule-based alternatives. Only if all agents fail to produce output does the framework revert to the rule-based `mock_fusion` fallback. This layered approach ensures that useful adaptations are preserved whenever possible, rather than being lost due to a single point of failure.

4.6.4 Dynamic Configuration

A key advantage of MA-SIF is that it is configured almost entirely through the `sif_config.json`, making it very adaptable without requiring backend code changes. Each agent's behaviour can be fine-tuned individually including its allowed actions, model type, temperature, thinking budget, and timeout, simply by editing the configuration file. This flexibility extends to the ability to add new agents or duplicate existing ones with different focuses.

For instance, it is entirely possible to run two Geometry agents in parallel: one optimised for mobile devices with smaller screens and one tuned for desktop layouts. Both would analyse the same event and profile data but apply different heuristics in their prompts. The Validator Agent would then merge their suggestions, resolving overlaps and conflicts automatically. This makes it possible to create specialised agents for emerging modalities, such as gaze tracking in VR, without altering any core fusion logic.

This externalised configuration is particularly valuable in accessibility research, where rapid iteration is needed. For example, during a live user study, the prompt for the UI Agent can be adjusted to emphasise high-contrast themes over font scaling without redeploying the backend. Similarly, experimental agents

for new modalities such as gaze tracking or haptic feedback can be added in minutes, allowing researchers or developers to test new adaptation strategies with minimal downtime.

4.6.5 Example in Action

Consider a user with both `visual_impaired: true` and `hands_free_preferred: true` in their profile. They attempt to turn on the lights by tapping near the "Turn on" button but miss slightly, then say "Turn on the lights" almost immediately afterwards. The UI Suggestion Agent, informed by the user's visual impairment, recommends switching to high-contrast mode and displaying a tooltip to make it easier to know the app's workings next time. The Geometry Suggestion Agent identifies the repeated miss and proposes increasing the button size. The Input Suggestion Agent recognises the voice command and suggests both switching to voice mode and triggering the button directly.

These suggestions are passed to the Validator Agent, which removes any duplicate `increase_button_size` actions, ensures all parameters are sound, and merges the remaining actions into a single, ordered list. The final adaptation set includes the contrast adjustment, the button enlargement, the tooltip display, the voice mode switch, and the direct triggering of the button. All of these are applied in one update, making the interface immediately more accessible and easier to use in future interactions with the possibility of making them permanent for the user, by updating their profile.

This scenario also highlights MA-SIF's integration with persistent profiles and recent history: because the profile already records both visual impairment and a hands-free preference, the agents start from a position of context-awareness rather than guessing from scratch. The resulting adaptations are therefore both reactive to the immediate miss-tap and proactive in aligning with the user's long-term accessibility needs.

4.6.6 Benefits of the Multi-Agent Approach

The multi-agent architecture provides several practical advantages. Reliability is improved because a failure in one agent does not prevent others from giving valuable suggestions. The modularity of the design makes it straightforward to maintain, as each agent can be modified or tuned independently without risking regressions in unrelated areas. Running agents in parallel also improves responsiveness, particularly when some suggestions can be applied even before all agents have finished processing.

Specialisation further enhances the quality of the output, as each agent's prompt scope is narrow enough to minimise irrelevant reasoning. The dynamic configuration system makes it possible to scale the number of agents up or down, or to swap in different models, without any changes to the backend logic. This adaptability is especially important in research and prototyping contexts, where requirements may change quickly.

The practical implementation of MA-SIF including the `sif_config.json` schema, example prompts, and backend implementation is detailed in Chapter 5, where its real-time performance and resilience to partial failures are also evaluated.

4.7 Prompt Engineering for LLMs in SIF

One of the most important parts of Smart Intent Fusion is **how we talk to the LLM**. Unlike a traditional rule-based system, where logic is written explicitly in code, here the behaviour of the LLM depends on the instructions it receives, namely the prompt. If the prompt, by which we **query the LLM**, is unclear, missing context, or too open-ended, the output will either be wrong, inconsistent, or impossible to parse in code. The LLM could also be more prone to hallucinations, which means it generates answers that sound plausible but are actually incorrect or nonsensical. This means the design of the prompt, the model parameters, and the output format all directly affect how useful and reliable the adaptations are. Even though the prompts used in this framework are relatively simple compared to large fine-tuned systems, they still follow a consistent structure and design philosophy that make them work for this use case. In the architecture described in Chapter 3, each event passed to the SIF Backend Layer arrives already standardised, enriched with metadata, and paired with relevant user profile and history context. The following prompt engineering process simply embeds this structured data into the LLM request, ensuring that reasoning always starts from the same reliable, modality-agnostic representation.

4.7.1 LLM Prompt Design Principles

Prompt engineering in this context is not just about getting a correct answer, it directly impacts accessibility outcomes. A poorly constrained prompt could suggest adaptations that introduce visual clutter, require unnecessary actions, or even reduce usability for the intended audience. By embedding accessibility goals, WCAG criteria, and known user needs into the prompt, the LLM is guided toward changes that genuinely improve the interface rather than merely altering it. The main objectives that flow from this understanding when writing the SIF prompts were:

1. **Be unambiguous**: Avoid instructions that could be interpreted in multiple ways.
2. **Enforce a strict JSON schema**: The frontend and backend depend on predictable keys and types.
3. **Constrain actions**: Only allow adaptations that are valid for the given agent type.
4. **Tie reasoning to context**: The model should always consider the event, user profile, and recent history together.

The idea was to make the prompts as predictable as possible. In accessibility systems, consistency often matters more than creativity or flexibility.

4.7.2 Prompt Structure from `sif_config.json`

Each agent (UI, Geometry, Input) has its own prompt in `sif_config.json`.
Here's a shortened example from the UI agent:

Listing 4.6: Example UI Agent Prompt

```
{
    "prompt": "You're the UI suggestion Agent.
    Analyze user event: {event_json}
    User profile: {profile_json}
    Recent history (last 10 events): {history_json}
    Suggest UI adaptations as JSON in the strict format.
    Each suggestion must include 'intent', 'reason', and at least a '
       value' and 'mode' field. Value must be a reasonable number (e.g.,
        1.2) with at most one decimal place, and represents a scaling
       value unless stated otherwise in the metadata (e.g. font size).
       Target can be 'all' or specific elements."
}
```

The important elements here are:

- **Role definition**: explicitly stating the agent's focus ("UI suggestion Agent").
- **Context injection**: inserting the current event, profile, and history as JSON strings.
- **Output constraints**: telling the model exactly what keys and value types are allowed.
- **Value rules**: restricting numeric ranges so the model doesn't output absurd sizes.

The complete prompt set, agent-specific instructions, and their runtime configuration are detailed in Chapter 5.

4.7.3 Disjunction Ambiguity in LLM Interpretation

During testing, we observed that LLMs can misinterpret logical connectors such as **or** and **and**. For example, the instruction:

> "The adaptation must include a value or a mode field."

was intended to mean *at least one of these fields is required for this action type.* However, the model sometimes treated this as an exclusive choice (only one allowed) or as fully optional (neither required),

even in cases where one was necessary. This can result in incomplete adaptations. This behaviour aligns with the well-known inclusive–exclusive disjunction ambiguity described in requirements engineering and computational linguistics, where natural language "or" lacks explicit semantic constraints and is prone to misinterpretation.
For example:

- `{"action":"increase_button_size","target":"lamp","value":1.23}` — requires `value` but not `mode`.
- `{"action":"switch_mode","target":"all","mode":"voice"}` — requires `mode` but not `value`.
- `{"action":"highlight_button_border","target":"button"}` — valid with neither, as neither field is relevant for this adaptation.

Having both fields is rather uncommon, but not impossible. This should handled further by the frontend and depends on the developer's implementation choices. Even with this type of prompt engineering, it sometimes still returns a `mode` field when a `value` field would be more appropriate, and vice versa. To reduce misinterpretation, prompts should explicitly state inclusive meaning when a field is required, such as:

> "The adaptation must at least include a value and a mode field, depending on the action type."

This minimises the risk of missing required parameters due to inclusive-exclusive disjunction ambiguity.

4.7.4 Balancing Model Parameters

Even with a well-written prompt, model behaviour is strongly affected by:

- **Temperature**: how random the outputs are. Lower values (0.2–0.3) are better for consistent JSON.
- **Thinking budget**: how many reasoning steps the model takes before output. Too low, and it may skip checks; too high, and it can slow down or get stuck.
- **Timeout**: how long we wait before falling back to mock or rule-based logic.

The Validator Agent especially needs more time and budget because it has a heavier job. This includes more complex reasoning and validation tasks like checking for inconsistencies in the adaptation or checking every adaptation against allowed actions, as well as removing irrelevant adaptations or merging duplicates. In my testing, the validator with a low thinking budget and temperature often got stuck or took a very long time to respond, which is not ideal for real-time adaptations. The goal was to stay under a timeout of max. 30 seconds. As described earlier, the best settings for the validator are:

- **Model**: `gemini-2.5-flash` (larger than the lite models used in the other agents).
- **Temperature**: 0.3 (slightly higher to avoid "freezing" on unusual cases).
- **Thinking budget**: -1 (dynamic) to give it more room for complex validation.
- **Timeout**: 30 seconds instead of the default 15.

4.7.5 Avoiding Hallucinations and Bad Values

One common risk with LLM-driven adaptations is hallucination, where the model invents an action, target or value that doesn't exist within the scope of the app. To reduce this:

1. The allowed actions list is always clearly included in the prompt.
2. Targets are validated against the current UI state before applying them as well as by the Validator Agent.
3. A list of focus items is included to guide the model's attention and provide additional context to minimize irrelevant outputs.

4. Prompt clearly states the required fields and their expected values as well as adhering to the allowed actions and JSON contract.

Even so, the validator sometimes has to fix agent mistakes.
For example, if the geometry agent outputs:
{"action":"increase_button_size","target":"button_unlock"}
The validator can correct it to:
{"action":"increase_button_size","target":"button_unlock","value":1.5}

4.8 Performance and Evaluation Metrics for AI Logic

4.9 Limitations and Challenges of LLM Integration

Large Language Models make Smart Intent Fusion far more capable than any static rule set, but they also introduce new dependencies, performance constraints, and reliability issues. In this thesis, all reasoning was powered by the Gemini API, and while this enabled rapid development, it also shaped both the strengths and weaknesses of the final system.

4.9.1 LLM selection

Gemini was chosen as the sole LLM provider for this framework for a mix of practical and technical reasons. The generous free tier and large input/output token limits allowed for frequent iteration without cost becoming a limiting factor. The API provided access to both smaller, faster models (used for the UI, Geometry, and Input agents) and larger, reasoning-focused models such as `gemini-2.5-flash` or even `pro` (used for the Validator Agent). This made it possible to optimise speed where possible and allocate more resources to roles that needed deeper analysis.

Gemini's handling of structured JSON output was also an advantage, as Smart Intent Fusion depends on predictable schema-compliant responses. While this project did not directly benchmark other models like GPT-5 or Grok, partly to keep the system stable during development and partly because Gemini's free tier already covered the thesis's usage without cost. Stability was important for building and testing MA-SIF without constantly adjusting prompts and entire agents for different model behaviours. However, it also means that all testing and performance observations in this chapter are specific to Gemini's runtime behaviour, and the results may differ if another LLM were used.

4.9.2 Reliability and Latency Constraints

A constant challenge was balancing response quality with the speed needed for near real-time accessibility. Smaller "lite" models returned in fractions of a second and were ideal for the three suggestion agents. The Validator Agent, however, required the larger gemini-2.5-flash model to perform more complex and reliable checks across multiple agent outputs. The trade-off was that validation could sometimes become the slowest part of the pipeline, especially with its increased timeout (30 seconds) and dynamic thinking budget. Network latency or temporary API slowdowns in the agents could lead to fewer suggestions being returned for a given event. In those cases, partial results were still applied rather than waiting or retrying for a full set.

4.9.3 Hallucinations and Invalid Output

Even with strict prompts and explicit allowed-action lists, hallucinations still appeared in the output. These took the form of actions outside the approved set, targeting elements that did not exist, or producing unreasonable scaling values (for example, value: 10). The Validator Agent was able to remove or correct most of these before they reached the frontend, but this came at the cost of extra processing time and complexity. Without validation, such outputs could have caused visual glitches or broken layouts, especially in geometry-related adaptations. The Input Adapter Layer or the Frontend itself, should as a fallback also have extra validation checks.

4.9.4 Token Limits and Context Size

Gemini's token allowance was one of the main reasons for choosing it, but token limits were still a factor. Each agent's prompt included the current event, the user profile, and the last few events from history, all in JSON format. In cases where the history was long or metadata verbose, this could approach the model's input size limit. The solution was to truncate history in those cases, ensuring that the event and profile data always took priority, even if it meant losing some recent context. Furthermore the token limit per minute and per day also limited the number of requests that could be made, which is why the framework was designed to handle partial failures gracefully and still return useful adaptations even if some agents timed out or exceeded limits.

4.9.5 Validator Complexity

The Validator Agent is both the most important and the most resource-intensive part of MA-SIF. It merges outputs from multiple agents, removes duplicates, corrects invalid values, resolves conflicting suggestions, and ensures the final adaptations list passes schema validation. This workload made it prone to timeouts when handling a large number of adaptations at once. Increasing the thinking budget and timeout reduced these failures but also increased total response time, creating a constant balance between reliability and speed.

4.9.6 Dependency on External APIs

Finally, using an external LLM API means the framework is dependent on network availability and the stability of the provider. If the Gemini API is unavailable or returns errors, the system falls back to the rule-based logic. While this ensures baseline functionality, it also removes the more context-aware reasoning that makes Smart Intent Fusion valuable. In a production setting, this could be mitigated with on-device models or by integrating multiple LLM providers as backups.

4.10 Future Directions for AI-Driven Adaptation

While Smart Intent Fusion in its current form is functional and effective for the scenarios tested in this thesis, it is still a first iteration of what an AI-driven, multimodal UI adaptation system could be. The underlying architecture, especially the multi-agent design and the strict JSON-based API contract, was deliberately built with future extensions in mind. Several directions could significantly expand its capabilities and make it more adaptive and autonomous.

One natural evolution is the introduction of visual context through image, vision-based or even at runtime UI analyzer models. Currently, SIF relies on structured metadata from the frontend to understand the UI state. In future versions, a lightweight computer vision model could take periodic screenshots of the interface and produce a semantic map of UI elements, their sizes, positions, and visual contrasts. This map could then be passed to the LLM alongside the existing event, profile, and history data. The advantage of this approach is that the AI would be reasoning over actual UI layouts, rather than relying on the frontend to describe them accurately which can cause misinterpretations. This opens the door to truly context-aware adaptations. For example, increasing the size of the smallest actionable element on a crowded screen, even if it hasn't yet caused an interaction error or accurately reposition elements closer to the user's focus.

Another promising direction is integrating user feedback into the adaptation loop. At present, SIF updates the user profile implicitly, based on interaction patterns or the user itself. A future version could prompt the user after significant adaptations with a quick, accessible feedback mechanism ("Was this change helpful?"). This feedback could be stored alongside interaction history and used by the LLM to adjust its decision-making over time.

There is also scope for dynamic, UI-level code changes driven directly by the LLM. Currently, SIF works within a fixed set of allowed actions and values prone to some hallucinations. This could be expanded so that the LLM can modify layout constraints, create new UI elements, or reorganise screens entirely with safeguards in place to prevent breaking the interface. The LLM could use its own "hallucinations" to provide creative solutions for layout issues or user interactions. This would take SIF from an adaptation system to a full UI changing layer, capable of designing new interactions on demand.

Finally, the framework could explore multi-model, multi-provider reasoning and threading. At present, all reasoning is performed by Gemini and run sequentially, which simplifies development but limits the diversity and speed of outputs. Future versions could run agents across different LLM providers and threads or even mix LLMs with specialised non-language models (e.g., reinforcement learning agents for adaptation strategies), with the Validator Agent controlling and validating the final output. This would provide extra resilience against provider outages and allow different models to play to their strengths, as well as a strong speed improvement when asynchronous processing is implemented. Every agent could potentially run in its own thread or process and joined by the validator, allowing for true parallelism and faster overall response times.

4.11 Chapter Summary

This chapter presented Smart Intent Fusion (SIF) as the core reasoning layer of the adaptive framework. SIF combines multimodal inputs, user profiles, and recent interaction history to deliver personalised, context-aware UI adaptations. A hybrid approach of rule-based logic and LLM-driven reasoning ensures both reliability and flexibility, allowing essential accessibility features to work instantly while enabling more complex, context-sensitive changes.

The multi-agent architecture (MA-SIF) was introduced, with specialised agents for UI, geometry, and input adaptations, and a Validator Agent responsible for merging and cleaning outputs. This design improves reliability, supports partial fallbacks, and can be dynamically reconfigured via `sif_config.json`. Prompt engineering emerged as a critical factor in ensuring valid, schema-compliant LLM output, with careful wording required to avoid logical misinterpretations.

Limitations of LLM integration including latency, occasional hallucinations, and reliance on a single provider, were mitigated through strict validation and fallback mechanisms. Finally, the chapter outlined future directions for SIF, such as adding visual UI context, integrating user feedback, enabling deeper UI changes, and exploring on-device AI models.

Overall, Smart Intent Fusion was presented not just as an algorithm, but as a modular, extensible smart reasoning layer designed to work across platforms, adapt to different users, and remain robust in the face of LLM unpredictability. It is the component that turns multimodal input into meaningful, personalised adaptations for reshaping the interface to fit the user.

Chapter 5

An Adaptive Multimodal GUI Framework using LLMs

5.1 Introduction to an Adaptive Smart Home Controller

The Adaptive Smart Home Controller is the practical proof-of-concept used to implement and validate the multimodal AI-driven GUI framework presented in this thesis. It serves as a concrete example of how the framework's concepts, introduced in Chapter 3, can be applied in a real, interactive application. The Smart Home Controller simulates the control of typical household devices such as lights, thermostats, and door locks. While the devices themselves are virtual, the process of capturing inputs, interpreting them through the backend with user profiles and history, and applying adaptive changes to the interface is authentic and functionally representative of a real deployment.

The choice for a smart home context was deliberate. It offers a relatable and real-life use case structured set of interaction scenarios that cover a range of UI components, buttons, sliders, text elements which are central to accessibility-focused adaptations. Furthermore, it reflects real-world situations where users may have diverse abilities and input preferences. For example, a motor-impaired user might need larger buttons to avoid frequent miss-taps, while a visually impaired user may benefit from higher-contrast modes and enlarged text. By embedding these scenarios into a single, unified application, the Smart Home Controller provides a manageable but representative testbed for the framework's adaptive capabilities.

The system adheres to the three-layer architecture established in earlier chapters. The frontend layer, implemented in Flutter (`adaptive_ui_app.dart`), renders the interface, captures user interactions, and applies adaptation instructions as they are received from the backend. The Input Adapter Layer (`adaptive_ui_adapter.dart`) acts as a middleware component, converting raw inputs from multiple modalities into the framework's JSON-based event format and ensuring user profiles are retrieved or updated before events are transmitted. The backend layer (`backend.py`), built with FastAPI, Gemini API and MongoDB, implements the Smart Intent Fusion (SIF) process. This includes both deterministic, rule-based adaptations and multi-agent LLM-driven reasoning (MA-SIF), combining event data, user profiles, and interaction histories to produce targeted adaptation actions.

It is important to note that this first iteration does not yet incorporate every capability described in the framework's long-term vision. Certain modalities, such as gesture recognition, are currently simulated through mock events to keep the implementation focused on the adaptation pipeline rather than input hardware integration. Throughout this chapter, each component is discussed in detail, with a clear distinction made between fully implemented functionality, simulated elements, and features that remain as future work.

5.2 Development Environment

The development environment for the Adaptive Smart Home Controller was chosen to support rapid prototyping, cross-platform deployment, and straightforward integration with the multimodal adaptation framework described in earlier chapters. It needed to provide a fast feedback loop during implementation,

flexibility in UI design, and robust backend capabilities to support real-time Smart Intent Fusion. The final setup reflects a balance between practical constraints such as available hardware, time and technical requirements namely WebSocket support, database persistence, and LLM integration.

Flutter was selected for the frontend because of its ability to produce visually consistent applications across desktop, mobile, and web from a single codebase. The framework's reactive UI model and composable widget system made it straightforward to create adaptive components whose properties such as size, color, and layout can be updated dynamically in response to backend instructions. Flutter's hot reload feature also significantly reduced iteration time, which proved essential for testing the frequent, small adjustments needed when refining adaptation behaviors.

The backend is implemented in Python using FastAPI, chosen for its simplicity, asynchronous request handling, and native WebSocket support. FastAPI's lightweight structure allowed the project to keep the adaptation pipeline transparent and easily modifiable, while still offering the performance needed for real-time interactions. MongoDB was selected as the database because its document-based structure aligns directly with the JSON contracts used throughout the framework. It stores user profiles, interaction histories, and adaptation logs without the need for complex schema migrations, making it well-suited for iterative development.

Development and primary testing took place on macOS 15.6, with additional runs on Windows 11 to confirm cross-platform compatibility. Both environments used Visual Studio Code with the Flutter and Dart plugins for frontend work, and Python tooling for backend development. This combination made it possible to run and debug both layers side-by-side, with the frontend communicating directly to a locally hosted backend via WebSocket and HTTP.

For clarity, the main environment components were as follows:

- **Operating System:** macOS 15.6 (primary), Windows 11 (secondary testing)
- **Frontend Framework:** Flutter SDK 3.7.0 or higher
- **Programming Languages:** Dart 2.19.0+ (frontend), Python 3.9+ (backend)
- **Backend Framework:** FastAPI with Uvicorn ASGI server
- **Database:** MongoDB 6.0+ for persistent profile and interaction history storage
- **IDE:** Visual Studio Code with relevant Flutter/Dart and Python extensions
- **Communication:** WebSocket for real-time adaptation updates, HTTP for profile management and batch operations
- **Version control:** Git, with the repository hosted on GitHub for collaboration and version tracking.

While this configuration is sufficient for the current implementation, it is also designed to be portable. The backend can be deployed to cloud environments without modification, and the frontend can target iOS, Android, or desktop platforms simply by recompiling or adjusting the build configuration. This flexibility ensures that the same codebase can serve as both a research prototype and a potential foundation for future production-ready systems.

5.3 Frontend Implementation: Smart Home Controller

The frontend of the Adaptive Smart Home Controller is implemented in Flutter and serves as the primary user-facing component of the framework. Its role is to render the interface, capture user interactions across multiple modalities, and apply adaptation actions received from the backend in real time. Although the backend is responsible for reasoning about what adaptations to make, the frontend is where these adaptations become visible to the user and directly influence usability.

The application is structured around a scrollable list of device "cards," each representing a smart home device such as a lamp or thermostat. These cards contain core interactive elements, including buttons for on/off actions, sliders for settings such as temperature, and labels for contextual information. The layout is designed to be minimal and uncluttered, ensuring that accessibility adaptations, such as increased button sizes or higher contrast modes, have a clear and immediate effect on the interface.

When a user pressed on an mockup event in the test panel underneath each device, the frontend captures the event along with metadata such as the element's type for UI context, the interaction type, and any relevant parameters. This data is sent through the Input Adapter Layer, which standardises the event into the framework's JSON contract before forwarding it to the backend via WebSocket. The choice of WebSocket ensures that adaptations can be returned and applied in near real time, a critical requirement for maintaining a smooth user experience in accessibility scenarios. As a loading indicator, the frontend displays rotating linear gradient border around the card being interacted with, until the adaptations are received and applied.

Adaptations received from the backend are applied immediately using Flutter's reactive state management. For example, a `increase_button_size` action triggers an `AnimatedScale` widget update, enlarging the targeted button over a short animation to make the change more noticeable without disrupting the user's flow. Similarly, a `increase_contrast` action adjusts the application's color scheme by updating theme parameters, while text-related adaptations update font sizes dynamically. This direct mapping between adaptation actions and Flutter widget properties allows the frontend to respond flexibly to a wide range of changes without requiring hardcoded layouts.

The current implementation focuses on touch-based interaction as the primary live modality, with voice and gesture inputs simulated for demonstration purposes. These mock events are triggered through test buttons in the interface, allowing the adaptation pipeline to be validated end-to-end without requiring physical input hardware. This approach enables reliable testing of adaptation logic while keeping the system architecture ready for integration with real voice or gesture recognition modules in the future.

Overall, `adaptive_ui_app.dart` demonstrates how the frontend layer of the framework can be implemented in a way that is both platform-independent and responsive to dynamic adaptation instructions. By separating UI rendering from adaptation logic and using the Input Adapter Layer as a bridge, the application remains modular, making it easier to extend or replace individual components without affecting the overall system.

5.4 Input Adapter Layer: Multimodal Input Handling

The Input Adapter Layer is responsible for bridging the gap between raw user interactions in the frontend and the structured event format required by the backend. It ensures that all inputs, regardless of modality, are represented consistently, allowing the backend's adaptation logic to operate on a predictable and schema-compliant data structure. This layer also manages user profile checks before events are transmitted, ensuring that adaptation decisions are always made in the context of the most up-to-date profile information.

In the current implementation, the adapter intercepts mock events generated by the frontend, such as missed button presses, missed slider adjustments, or simulated voice commands. Each event is enriched with metadata, including the user's identifier, a timestamp, the type of interaction, and any target element references. The adapter then converts this information into the JSON event contract high-level defined in Chapter 3, which serves as the standard interface between the frontend and backend. This contract includes fields for event type, source modality, target element, coordinates if applicable, confidence level, time stamp, user_id and additional metadata such as the spoken command in the case of voice input.

Before forwarding an event, the adapter queries the backend via HTTP to check whether a profile exists for the given user. If no profile is found, it prompts the frontend to initiate a profile creation request, using default parameters or pre-filled accessibility preferences where available. This mechanism prevents situations where adaptation requests are processed without the necessary user context, which could lead to ineffective or even counterproductive UI changes.

Communication with the backend is handled primarily through WebSocket for low-latency adaptation feedback. HTTP requests are used for profile management, batch operations, and other non-real-time interactions. This division ensures that profile updates and administrative tasks do not interfere with the responsiveness of live adaptations.

While the adapter is fully implemented for touch-based events or voice and gesture events which can be triggered manually within the frontend's test interface, allowing the full adaptation pipeline to be exercised without requiring live hardware or third-party input recognition services. The adapter treats

these simulated events identically to real ones, ensuring that integration with actual input sources in the future will require minimal changes and solely constrained to the frontend.

The Input Adapter Layer plays a crucial role in keeping the framework modular. By isolating the event formatting and profile validation logic here, the frontend can focus purely on UI rendering and interaction capture, while the backend can rely on receiving consistent, validated input. This separation not only improves maintainability but also makes it possible to reuse the adapter design across different frontends, such as a SwiftUI application or a Unity-based VR interface, without rewriting the backend communication logic.

5.5 SIF Backend Layer: Implementation of Adaptation Logic

The backend implements the decision-making core of the framework. It receives standardised events from the adapter, fuses them with user profiles and recent interaction history, and returns concrete adaptation actions for the frontend to apply. The service is written in Python using FastAPI, with Uvicorn for serving requests and MongoDB for persistent storage of profiles and logs. WebSocket is used for low-latency, bidirectional communication during interaction, HTTP is used for profile management and auxiliary endpoints.

5.5.1 Webserver layout and endpoints

The application exposes a WebSocket endpoint, `/ws/adapt`, that accepts JSON events matching the contract introduced earlier. Each message is parsed into a Pydantic `Event` model, the user profile is loaded from MongoDB, and the event, profile, and short history window are passed to the Smart Intent Fusion routine. The resulting adaptation list is returned on the same socket, allowing the frontend to update the UI immediately. For non-interactive operations the backend offers `POST /profile` for profile creation, `GET /profile/user_id` for retrieval, and a small set of diagnostic endpoints such as `GET /full_history` and `GET /modalities`. Profile writes use FastAPI `BackgroundTasks`, which keeps the interaction path responsive while updates are persisted asynchronously.

5.5.2 Data persistence and history management

MongoDB stores user profiles in a `profiles` collection and event–adaptation pairs in a `logs` collection. Profiles are indexed by `user_id` to allow direct lookups during interaction. Each time an event is processed the backend appends a compact JSON representation to the profile's `interaction_history`, capped to a small sliding window. This keeps the prompt context focused on recent behaviour while avoiding unnecessary growth in the database. For redundancy, a JSONL file mirrors the log entries during development, which simplifies offline inspection or debugging purposes when the database is reset.

5.5.3 Smart Intent Fusion and MA-SIF

The fusion step supports two paths. A multi-agent configuration, MA-SIF, is the default. It calls a set of specialised LLM agents, UI, Geometry, and Input, each prompted with the current event, the user profile, and a short history. These agents propose structured adaptations in their domain, for example increasing button size, adjusting spacing, switching input mode, or triggering a button when intent is clear. Their outputs are then passed to a dedicated Validator agent that consolidates, filters, and normalises the suggestions into a final list. The validator removes duplicates, corrects out-of-range values, and ensures that every adaptation conforms to the allowed action set and includes a target and either a value or a mode and more.

A single-agent SIF path is also available. It produces a complete adaptation list in one call and is useful when quick iteration is preferable over agent specialisation. Both paths share the same I/O schema, which keeps the frontend indifferent to which reasoning strategy is currently set to active.

5.5.4 Structured outputs and guardrails

To reduce hallucinations and schema drift, the backend requests JSON-typed responses from the LLM with an explicit schema and allowed actions. The prompt defines required fields, expected types, and a one-of constraint that demands either a numeric `value` or a categorical `mode`. Agent prompts are

intentionally narrow, which improves determinism. The validator prompt is broader, since it must reconcile conflicting proposals and justify final choices. Despite these controls, invalid outputs still occur occasionally. It has a defensive layer that falls back to simple rules if validation fails.

5.5.5 Rule-based fallback and resilience

A lightweight rule engine acts as a safety net when LLM calls time out or the provider is unavailable. It covers essential accessibility behaviours, for example increasing button size after a miss-tap, switching to voice for users flagged as motor-impaired, or enabling high-contrast mode for visually impaired profiles. These rules are intentionally conservative, they guarantee progress without surprising the user, and they keep the implementation usable in environments with unstable connectivity.

5.5.6 Latency, partial results, and error handling

The WebSocket loop is designed to return something useful as quickly as possible. Agent calls run in sequence within short time budgets. If one agent fails to respond, the validator operates on the remaining suggestions rather than waiting indefinitely. The backend aims to keep per-event processing below the threshold where users notice a lag on interaction, which is important for accessibility, particularly when enlarging targets immediately after an error. In practice, most adaptations are returned quickly by the smaller suggestion agents, while validation can become the slowest step in complex scenes. When validation exceeds its budget the backend returns the best available subset, then continues to append the event to the user's history so future interactions benefit from the context.

5.5.7 Security and CORS considerations

During development the backend enables permissive CORS to simplify local testing across platforms. Profiles are keyed by `user_id` rather than personal data. For production deployment, stricter origins, authentication, and encryption would be required. These measures are outside the scope of this prototype, but the separation of concerns in the current design makes them straightforward to add.

5.5.8 Summary

In its current form the backend delivers a complete adaptation pipeline: events arrive over WebSocket, profiles and short history windows are loaded from MongoDB, MA-SIF produces structured suggestions, a validator consolidates them, and the result is returned to the frontend within a single interaction loop. When LLM reasoning is unavailable, conservative rules ensure the interface remains usable. This combination of multi-agent reasoning, strict schemas, and rule-based fallbacks gives the system both flexibility and reliability, which is essential for accessibility-focused adaptations.

5.6 User Profile and Context Implementation

The user profile and context subsystem was implemented as a dedicated data service in the SIF backend, designed to persist accessibility needs, interaction preferences, baseline UI configurations, and a capped history of recent events. MongoDB serves as the primary storage layer, with the `profiles` collection indexed on `user_id` as described earlier for constant-time retrieval during event processing.

When a new event is received via the WebSocket (`/ws/adapt`), the backend queries the profile store using the supplied `user_id`. If no profile exists, the profile will be created and added to the MongoDB using `insert_one` before continuing, otherwise the profile will be retrieved using `find_one` or updated using `update_one`, ensuring that all adaptation decisions are made in a contextualized environment. This retrieval step is synchronous, guaranteeing that the most recent committed profile is available to the reasoning pipeline before any LLM or rule-based evaluation occurs. Interaction history is maintained using MongoDB's `$push` with `$slice` operators to append the incoming event while capping the array length at 10 entries for efficiency. This rolling history provides the SIF agents with temporal context, enabling progressive personalization; for example, recognising a pattern of repeated miss-taps and proactively switching to voice mode. Updates to the history are performed asynchronously to avoid blocking real-time adaptation.

Profile documents are structured as JSON as described in chapter 3, containing four key sections: `accessibility_needs` (boolean capability flags), `input_preferences` (preferred modality and fallback

order), `ui_preferences` (default font size, button scale and more), and `interaction_history` (recent event log). This schema strikes a balance between simplicity and extensibility, allowing new fields to be added without migration overhead.

To optimise performance and safety, all profile mutations are atomic, relying on MongoDB transactions to prevent race conditions when simultaneous events and updates occur. Adaptation logs are stored separately in the `logs` collection and mirrored to a local `adaptation_log.jsonl` file, supporting offline analysis and reproducibility of evaluation results. Furthermore, if a profile update is in-flight during an event, the backend uses the latest committed profile, mitigated by client-side checks (waiting for `POST/PUT /profile` success) and server-side transactions.

This implementation ensures that every adaptation decision, whether produced by a static rule or the multi-agent LLM pipeline, is grounded in the user's persisted profile and immediate interaction context, enabling consistent, personalised, and stateful UI behaviour across sessions.

5.7 Dynamic Adaptation Mechanisms: Implementation Details

The dynamic adaptation mechanism is the final stage in the interaction pipeline, where decisions made by the backend are translated into immediate and visible changes in the user interface. In the current implementation, this process is tightly integrated with Flutter's reactive widget system, allowing adaptation actions to be applied without forcing full UI rebuilds or navigation resets.

When the backend sends an adaptation list over the WebSocket connection, the frontend parses each action and routes it to the relevant UI element. Actions are defined in the strict JSON schema described earlier, containing the `action` type, a `target` identifier, a `value` or `mode`, and a human-readable `reason` and `intent` it inferred from the adaptation. This standardisation allows the same adaptation handler to process diverse actions without requiring modality-specific logic.

The application of adaptations begins with a lookup to determine whether the `target` element exists in the current view. If it does, the corresponding widget state is updated directly. For example, an `increase_button_size` action adjusts the scale factor property of the button widget, and an `increase_font_size` action updates the text style parameter. Where appropriate, changes are animated using Flutter's `AnimatedScale` or `AnimatedContainer` to make the transition noticeable without distracting the user. This animation step is particularly important for accessibility, as it helps the user understand that the interface has been modified intentionally. The loading indicator displayed around the card being interacted is now also triggered to stop playing.

Adaptations that affect the entire interface, such as `increase_contrast` or `switch_mode`, are handled at the application theme level. Contrast adjustments update the color palette by replacing the primary and background colors with higher-contrast alternatives, while mode switches alter the active input modality, for example switching from touch to voice. These global changes are propagated across all widgets automatically through Flutter's state management, ensuring consistency without manually updating each element.

Conflicting adaptations such as multiple agents proposing different size adjustments for the same element are resolved in the backend before reaching the frontend, using the validator agent to select or merge the most appropriate action. This guarantees that the frontend always receives a coherent, non-overlapping set of changes, reducing the complexity of applying them in real time.

Not all adaptations are implemented with live modality inputs. For demonstration purposes, actions triggered by voice or gesture events are generated from simulated events in the frontend's test panel in each device card. However, these simulated actions follow the same processing path as real events, which means that integrating actual input sources in the future will require no changes to the adaptation mechanism itself.

5.8 Cross-Platform Implementation Considerations

5.9 Design Decisions

The design of the Adaptive Multimodal GUI Framework using LLMs reflects a series of deliberate choices aimed at balancing accessibility, performance, scalability, extensibility, and ease of integration. These decisions were made with the primary goal of delivering real-time, personalised adaptations for motor-impaired, visually impaired, and hands-free users, while ensuring that the framework remains modular and adaptable to future platforms and domains.

5.9.1 Modularity Over Monolithic Design

Decision: The framework adopts a modular three-layer architecture (Frontend, Input Adapter, SIF Backend) with clear separation of concerns, connected through a standardised JSON contract.

Reasoning:

- **Flexibility:** Each layer can be updated or replaced independently, enabling deployment across different platforms such as Flutter, SwiftUI, or Unity without reworking the entire system.
- **Extensibility:** The JSON contract in the Input Adapter Layer allows new modalities (e.g., eye tracking) to be integrated without modifying backend logic.
- **Developer accessibility:** Modularity simplifies integration, requiring minimal code for event handling and adaptation application.

5.9.2 WebSocket for Real-Time vs. HTTP for Batch Processing

Decision: The framework uses WebSocket (`/ws/adapt`) for real-time event processing and adaptation delivery, with HTTP (`/full_history, /profile`) for debugging, developer tooling and profile management.

Reasoning:

- **Low Latency:** WebSocket enables fast and bidirectional sending of data like adaptations (e.g., scaling a button after a miss-tap).
- **Reliability:** HTTP supports robust profile updates (`POST/PUT /profile`), ideal for non-real-time scenarios or debugging.
- **Accessibility:** Real-time feedback enhances usability for motor-impaired or hands-free users, where delays could disrupt interaction.

5.9.3 MongoDB for Persistent Storage

Decision: MongoDB is used for storing user profiles, interaction history, and adaptation logs, with `user_id` indexing and capped history arrays (10 events).

Reasoning:

- **Scalability:** MongoDB's NoSQL design and indexing ensure fast queries for large user bases, critical for real-world deployment.
- **Flexibility:** JSON-like documents align with the framework's JSON contract, simplifying profile/history storage.
- **Continuous Learning:** Retaining a limited history supports adaptive behaviour, such as making permanent size adjustments after repeated miss-taps.

5.9.4 Rule-Based Fallback with LLM Reasoning

Decision: SIF combines rule-based logic (e.g., `if miss_tap then increase_size`) with LLM reasoning for creative adaptations, with rules as a fallback for LLM failures or time-outs, since LLM's can have high respond latencies.

Reasoning:

- **Reliability:** Rules ensure deterministic adaptations (e.g., button enlargement for miss-taps) when LLM responses are unavailable or hallucinate.
- **Novelty:** LLM enables context-aware, proactive suggestions (e.g., `switch_mode: voice` for hands-free users), advancing beyond static rules.
- **Accessibility:** Hybrid approach ensures consistent support for motor-impaired, visually impaired users (e.g., high-contrast text).

5.9.5 Multi-agent LLM reasoning (MA-SIF) vs single-agent LLM reasoning (normal SIF)

A key architectural decision in the framework is the adoption of multi-agent LLM reasoning (MA-SIF) over a single agent LLM approach (normal SIF). In the single agent SIF model, one LLM is responsible for interpreting all input events and generating adaptation actions. While this simplifies integration and reduces system complexity, it can limit the granularity and specialization of adaptation logic, especially as the diversity of user needs and input modalities grows.

MA-SIF, by contrast, distributes reasoning across multiple specialized LLM agents, each focused on a distinct domain such as UI adaptations, geometry/layout changes, input modality management, and validation. This separation of concerns enables each agent to leverage tailored prompts, domain-specific knowledge, and focused reasoning strategies, resulting in more nuanced and context-aware adaptation suggestions. The validator agent further ensures that outputs from other agents are coherent, non-conflicting, and accessibility-compliant.

The multi-agent approach offers several advantages:

- **Scalability:** New agents can be added to address emerging modalities or adaptation domains without disrupting existing logic.
- **Extensibility:** Prompts and allowed actions can be updated independently for each agent, supporting rapid iteration and domain-specific improvements.
- **Robustness:** Specialized agents reduce the risk of LLM hallucinations or conflicting adaptations, as validation is enforced before application.
- **Personalization:** Agents can incorporate user profiles and history more effectively, enabling targeted adaptations for motor-impaired, visually impaired, or hands-free users.

5.10 Implementation Challenges and Solutions

Developing the Adaptive Smart Home Controller exposed several practical challenges, ranging from LLM-specific issues to general real-time system concerns. These were addressed through a mix of architectural decisions, fallback mechanisms, and compromises designed to keep the prototype functional and reliable under varying conditions.

5.10.1 LLM reliability and output consistency

One of the most persistent challenges was ensuring that the multi-agent LLM pipeline returned valid, schema-compliant output. Despite strict prompts and an explicit allowed-actions list, hallucinations still occurred, producing unsupported actions, targeting non-existent elements, or returning unreasonable values such as excessively large scaling factors. These errors risked breaking the layout or producing jarring visual changes. To mitigate this as described earlier, a validator agent was introduced to consolidate and correct suggestions before they reached the frontend. However, this validation step added extra latency and still could not guarantee complete protection against invalid values, making additional frontend-side checks a sensible next step for future versions.

5.10.2 Performance under real-time constraints

Adaptation latency was a critical factor for usability, particularly in accessibility scenarios where delayed feedback can reduce trust in the system. Smaller LLM models were assigned to the suggestion agents

to keep their execution time within fractions of a second, while the validator, which required more context and reasoning, used a larger model with a higher timeout budget. Even so, occasional slowdowns occurred due to network latency or provider-side delays. The backend was therefore designed to apply partial results if all agents did not respond in time, ensuring that at least some adaptations reached the user quickly.

5.10.3 Safeguards against malicious or replay attacks

In its current form, the system does not include strong safeguards against replay attacks or intentionally crafted events designed to trigger disruptive adaptations. This is acceptable in a controlled research environment but would require strict validation and authentication for production deployment. Measures such as signing WebSocket messages, verifying sequence numbers, and rejecting stale or malformed events would be necessary to prevent exploitation as well as extra protections in the adapter or frontend. Furthermore is user_id hijacking, where an attacker submits events under a different user's profile possible. This was intentional for the scope and development time constraints of this research.

5.10.4 Testing with incomplete modalities

Live gesture tracking and voice inputs were not integrated in this iteration, which meant that related adaptations had to be tested using simulated events. While this allowed the adaptation logic to be validated end-to-end, it did not account for the noise, recognition errors, or latency introduced by real input devices. Future work will need to focus on replacing these simulated events with actual input sources to fully evaluate performance in realistic conditions.

5.10.5 Security and trust boundaries

In its current implementation state, the backend accepts connections from any origin and does not require authentication for profile creation or event submission. Currently a pure localhost setup is implemented for the entire backend. This was a deliberate decision to speed up testing across platforms, but would need to be replaced with stricter CORS rules, token-based authentication, and access control in production. Without these measures, the system is vulnerable to unsolicited adaptation requests from external sources.

5.11 Chapter Summary

This chapter has detailed the implementation of the Adaptive Smart Home Controller as the primary proof-of-concept for the multimodal AI-driven GUI framework. The system was realised as a three-layer architecture, consisting of a Flutter-based frontend for rendering and applying adaptations, an Input Adapter Layer for standardising and transmitting events, and a FastAPI-based backend implementing multi-agent Smart Intent Fusion with MongoDB for persistent profile and history storage.

Key implementation aspects included the construction of a dynamic adaptation pipeline capable of applying size, contrast, modality, and content changes in real time, and the use of a strict JSON schema to maintain consistency between layers. While some modalities such as voice and gesture inputs were simulated for demonstration purposes, the system was designed so that integrating real devices will require minimal architectural changes.

The prototype also faced practical challenges, including LLM output validation, latency management, security considerations, and the absence of certain safeguards against malicious or replayed events. These were addressed through a combination of a validator agent, conservative rule-based fallbacks, partial-result handling, and a modular design that isolates critical components.

In its current state, the implementation demonstrates that the framework can deliver personalised, accessibility-focused adaptations in a responsive and modular manner, even under the constraints of a prototype environment. The next chapter evaluates this implementation through a feasibility study, assessing its responsiveness, adaptability, and practical potential in simulated real-world scenarios.

Chapter 6

Evaluation

6.1 Feasibility Study: Evaluation Approach

6.2 System Demonstration and Experimental Setup

6.2.1 Demonstration Scenarios and Configurations

6.2.2 Hardware and Environment

6.3 Results and Observations

6.3.1 Adaptation Effectiveness Across Configurations

6.3.2 Accessibility Impact Analysis

6.3.3 Adaptation Performance

Chapter 7

Discussion and Future Work

7.1 Implications for Accessibility and HCI

7.2 Key Findings and Contributions

7.3 Comparison with Related Work

7.4 Limitations and Challenges

7.5 Future Work

7.5.1 Extending to Existing UIs

7.5.2 UI Component Analyzer

7.5.3 Enhancing Multimodal Inputs

7.5.4 LLM Agents for Autonomous Adaptation

7.5.5 Specialized AI Model for UI Adaptation

Chapter 8

Conclusion

8.1 Summary of Contributions

8.2 Final Remarks