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 introduces a multimodal AI-driven GUI framework for dynamic user interface adaptation, enabling real-time, personalized accessibility enhancements across platforms such as Flutter, SwiftUI, and more. With a particular focus on the health domain, the framework is designed to support 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), powered by a multi-agent architecture consisting of large language models (LLMs) using the Gemini API, they fuse 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 system includes a developer-friendly, cross-platform SDK that standardizes inputs into a JSON contract and communicates with a FastAPI backend using both rule-based logic and LLM-driven reasoning. The feasibility of the framework is demonstrated through system demonstrations in varied configurations and simulated metrics (adaptation accuracy, task completion time, latency/performance), showing significant accessibility improvements. By addressing the gap in intelligent, multimodal UI adaptation, this work advances human-computer interaction and lays the foundation for future AI models capable of autonomously rewriting UI code, offering a scalable and extensible solution for personalized user experiences.

Acknowledgments

Contents

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

This multimodal AI-driven framework for dynamic user interface (UI) adaptation is designed to deliver near real-time, personalized enhancements for accessbility-focused applications. The framework specifically addresses the needs of motor-impaired, visually impaired and hands-free users through cross-platform compatibility and extensibility for future modalities, such as eye tracking or virtual reality interfaces. Unlike traditional static UIs, this architecture continuously adapts and refines its behavior based on contextual awareness and feedback. This system is implemented as an Adaptive Smart Home Controller, with the main implementation written in Flutter, serving as a proof-of-concept in chapter 5.

The system architecture consists of 3 core layers:

- Frontend Layer: Responsible for rendering responsive, adaptive UI components across platforms (e.g., mobile, desktop, VR). It reacts to adaptation instructions and can reconfigure layout, elements, or content in real time.
- Input Adapter Layer: which standerdizes multimodal inputs from the user (e.g., touch, voice, gaze, keyboard/mouse, gestures) into a uniform format. It also serves as a modular entry point for integrating future modalities like eye tracking or BCI (brain-computer interfaces).
- SIF Backend layer, which employs **Smart Intent Fusion (SIF)** to generate personalized adaptation actions. It uses reasoning mechanisms (LLM's) to detect user goals, predict optimal adaptations, and trigger UI updates. It continuously learns from user profiles, UI context signals and user feedback.

A feedback loop ensures continues refinement of adaptations, by using past events and actions from the user on the UI. This allows for progressive personalization and learning from user interactions over time. This chapter explores the design principles, component interactions, and architectural decisions while outlining the framework's foundational concepts.

Throughout this chapter, we elaborate on:

- the different core layers making up the framework,
- the design principles and decisions guiding the framework (modularity, scalability, generalizability),
- the communication protocols between layers,
- the lifecycle of an adaptation decision from input to effect,
- and the flexibility of the system to be deployed in various domains and platforms.

3.2 Overview of the System Architecture

3.2.1 High-Level Architecture

The multimodal AI-driven framework for dynamic UI adaptation is high-level structured as a three-layer architecture designed to ensure modularity, scalability, and accessibility-focused personalization. Each layer serves a distinct purpose, with clear separation of concerns to facilitate cross-platform compatibility and extensibility for future modalities. The layers interact through standardized communication protocols, enabling real-time adaptation and continuous learning from user interactions.
The architecture is as follows:

1. **Frontend Layer**: This layer is responsible for rendering the user interface and capturing user interactions across platforms such as Flutter, SwiftUI, Unity, etc. It dynamically applies adaptations (e.g., enlarging buttons, increasing contrast) based on instructions from the backend. For the thesis, a smart home controller UI is implemented, featuring scrollable device cards (e.g., Lamp, Thermostat, Door Lock), buttons (e.g., On/Off, Toggle), sliders (e.g., temperature adjustment), and text (e.g., device statuses). The frontend is designed to be responsive, ensuring accessibility for motor-impaired, visually impaired, and hands-free users.

2. **Input Adapter Layer**: Acting as a middleware, this layer standardizes multimodal inputs (e.g., touch, keyboard, voice, gestures) into a uniform JSON contract format. It abstracts the complexity of input processing, ensuring compatibility with diverse modalities and enabling extensibility for future inputs like eye tracking or brain-computer interfaces (BCI). The layer validates and enriches events with metadata as extra to provide data (e.g., timestamps, confidence scores) before forwarding them to the backend.

3. **SIF Backend Layer**: The core of the framework, this layer employs Smart Intent Fusion (SIF) to process standardized events, user profiles, and interaction history to generate personalized adaptation actions. It combines rule-based logic for deterministic cases (e.g., miss-tap triggers button enlargement) with multi-agent LLM-driven reasoning (MA-SIF) for creative, context-aware adaptations (e.g., switching to voice mode for motor-impaired users). The backend uses MongoDB for persistent storage of profiles and history, with WebSocket (/ws/adapt) for real-time communication and HTTP (/context, /profile) for batch operations.

The layers are interconnected via a feedback loop, where user interactions (events) are logged, analyzed, and used to refine future adaptations. This continuous learning mechanism ensures progressive personalization, adapting the UI to individual user needs over time. Communication between layers leverages WebSocket for low-latency, real-time updates and HTTP for profile management and batch processing, balancing performance and reliability.

3.2.2 Diagram: System Overview

DIAGRAM
Description: The diagram illustrates the data flow: Frontend Layer (UI rendering, event capture) → Input Adapter Layer (event standardization) → SIF Backend Layer (SIF, profile/history storage in MongoDB) → Frontend Layer (apply adaptations). Arrows indicate WebSocket/HTTP communication and feedback loops. The diagram highlights modularity, with each layer as a distinct box, and extensibility points (e.g., new modalities, platforms).

Key Design Principles:

- **Modularity**: Each layer is independent, allowing updates or replacements without affecting others (e.g., swapping Flutter for SwiftUI).
- **Scalability**: MongoDB and async processing handle high event volumes; WebSocket ensures low latency.
- **Generalizability**: JSON contract and platform-agnostic design support diverse domains (e.g., smart homes, healthcare) and future modalities.
- **Accessibility Focus**: Prioritizes adaptations for motor-impaired, visually impaired, and hands-free users, aligning with the thesis's health-oriented goals.

3.3 Frontend Layer: UI Design and Rendering

The Frontend Layer is the user-facing component of the multimodal AI-driven framework, responsible for rendering a adaptive and personalized user interface (UI) that dynamically adjusts to user needs, particularly for accessibility-focused applications. Designed to support cross-platform deployment on Flutter, SwiftUI, and future platforms like Unity for VR/AR, the layer captures multimodal user interactions (e.g., touch, keyboard, voice, gestures) and applies real-time adaptations based on instructions from the SIF Backend Layer. For the thesis, the Frontend Layer implements an Adaptive Smart Home Controller, a practical and relatable UI that emphasizes accessibility for motor-impaired, visually impaired, and hands-free users in a smart home context.

3.3.1 Primary Components: Button, Sliders, Text

The Frontend Layer incorporates three core UI elements, selected to balance simplicity and demonstrative power for accessibility adaptations:

1. **Button**: Primary interactive elements for controlling smart home devices, such as On/Off toggles for the Living Room Lamp, +/- adjustments for the Thermostat, and Toggle for the Front Door Lock. Buttons support adaptations like scaling (e.g., `increase_size` for motor-impaired users) and contrast changes (e.g., `increase_contrast` for visually impaired users). For example, a miss-tap on the Lamp's On/Off button may trigger an enlargement to improve subsequent interactions.

2. **Sliders**: Used for fine-grained control, such as adjusting the Thermostat's temperature (range: 15–30°C). Sliders adapt via thumb size increases (`increase_slider_size`) for motor-impaired users, improving precision for those with dexterity challenges. For example, a shaky or miss drag interaction may prompt the backend to enlarge the slider thumb.

3. **Text**: Displays dynamic information such as device statuses (e.g., "On", "20°C", "Locked") and user greetings (e.g., "Welcome, User!"). Text elements adapt through font size increases (`increase_font_size`) and high-contrast modes (`increase_contrast`) to enhance readability for visually impaired users.

The selection of buttons, sliders, and text as primary UI components is deliberate, striking a balance between simplicity and the ability to showcase the framework's accessibility-driven adaptations. Buttons, as the main interaction points, are critical for users with motor impairments, where scaling (e.g., `increase_size` via Flutter's `AnimatedScale`) directly improves tap accuracy, reducing frustration in real-world scenarios like toggling a lamp. Sliders, used for precise control like thermostat adjustments, address dexterity challenges by enlarging thumb sizes (`increase_slider_size`), making fine-grained interactions easier for motor-impaired users. Text elements, carrying essential information like device statuses, are vital for visually impaired users, with adaptations like `increase_font_size` and `increase_contrast` ensuring readability aligns with WCAG 2.1 guidelines (e.g., 1.4.4 Resize Text). Their design supports cross-platform portability, ensuring the framework's applicability to SwiftUI or future VR/AR environments, while the JSON contract enables seamless integration with the SIF Backend Layer's **Smart Intent Fusion (SIF)** for dynamic, personalized adaptations. This focused component set demonstrates the framework's practical utility in accessibility-focused applications, with clear potential for expansion to domains like healthcare or education.

3.3.2 Additional Needed Components

To enhance the UI's functionality and accessibility, additional components are integrated into the Frontend Layer, each serving specific purposes:

- **List View**: A scrollable list of device cards organizes the Smart Home Controller, each representing a usable home device (e.g., Lamp, Thermostat, Lock).

- **Icons**: Visual indicators within cards (e.g., light bulb for Lamp, lock for Door) provide intuitive feedback on device states. Icons adapt through color changes tied to `increase_contrast` actions, ensuring visibility for visually impaired users.

- **Cards**: Device cards that represents a controllable device in the Smart Home Controller. These cards group the different adaptable UI components together.

3.3.3 Adaptation Mechanisms

Adaptations in the Frontend Layer occur at 3 different levels, driven by instructions from the SIF Backend Layer:

1. **UI-Level Adaptations:**
   - **Button Scaling**: Buttons are resized using animations (e.g., Flutter's `AnimatedScale`) in response to `increase_size` actions, aiding motor-impaired users. For instance, a miss-tap on the Lock's Toggle button may increase its scale by 1.5x.
   - **Text Adjustments**: Font sizes are modified dynamically (`increase_font_size`) to improve readability, with smooth transitions for user comfort.
   - **Contrast Enhancements**: High-contrast modes (`increase_contrast`) adjust button, text, app background and icon colors (e.g., black background for buttons) to support visually impaired users.
   - **Showing Helpful Tooltip**: Tooltips are displayed on the bottom, providing additional context (e.g., "Tap to toggle Lamp or try to say 'Unlock'") to assist users with cognitive impairments in navigating the UI.
2. **Geometry-Level Adaptations:**
   - **Element Spacing**: Adjustments to the spacing between UI elements or cards (e.g., increasing padding) to improve touch targets for motor-impaired users.
   - **Element Resizing**: Dynamic resizing of UI elements (e.g., buttons, sliders) based on user preferences or context (e.g., larger controls for shaky hands).
3. **Input-Level Adaptations:**
   - **Mode Switching**: The UI switches to voice-driven navigation (`switch_mode: voice`) for hands-free users, enabling commands like "Turn on lamp" to trigger actions without physical input or UI navigation.
   - **Layout Simplification**: `simplify_layout` reduces UI complexity by hiding non-essential cards or repositioning elements (`reposition_element`) based on interaction frequency.
   - **Trigger Actions**: `trigger_button` actions automatically activate buttons (e.g., toggling Lamp to "On") when intent is clear from multimodal inputs (e.g., voice + gesture).

These mechanisms form the ground basis of the MA-SIF (SIF extension) framework, by using a combination of agents (LLM's). The MA-SIF framework leverages these three adaptation levels input, user interface (UI), and geometry to enable adaptive and personalized user experiences by mapping them across agents. Input adaptation tailors data processing to user-specific inputs, UI adaptation customizes the interface for usability, and geometry adaptation optimizes spatial configurations. By integrating these mechanisms through a combination of large language model (LLM) agents, MA-SIF dynamically adjusts to user needs, enhancing interaction efficiency and personalization.

3.3.4 Figma Diagram: Expanded UI View

[Figma mockup of the Adaptive Smart Home Controller]

Description: The diagram depicts the smart home dashboard with:

- A scrollable `ListView` containing cards for Lamp, Thermostat, and Lock.
- Each card with buttons (e.g., On/Off, +/-), text (e.g., "Status: Locked"), and sliders (Thermostat).
- Annotations showing adaptations (e.g., enlarged button, high-contrast text).
- Mock voice/gesture inputs (e.g., "Voice: Turn On", "Gesture: Point") for demo purposes.
- Visual cues for accessibility (e.g., larger fonts, simplified layouts).

Purpose: Illustrates how UI elements adapt dynamically to user needs, emphasizing accessibility for motor-impaired, visually impaired, and hands-free users.

3.3.5 Key Design Consideration

- **Accessibility Focus:** Components and adaptations prioritize WCAG-compliant features (e.g., high contrast, large hit areas) for inclusivity.
- **Cross-Platform Support:** Flutter ensures portability to mobile, desktop, and web; abstractions allow extension to Unity/SwiftUI and others.
- **User Feedback:** Visual feedback (e.g., glowing borders during adaptation) enhances demo impact and user understanding.
- **Extensibility:** Modular component design supports additional elements (e.g., progress bars) for future domains like healthcare.

3.4 Input Adapter Layer: Multimodal Input Processing

The Input Adapter Layer serves as a critical middleware in the multimodal AI-driven framework, responsible for standardizing diverse user inputs into a uniform JSON contract format for processing by the SIF Backend Layer. This layer ensures seamless integration of multimodal inputs such as touch, keyboard, voice, and gestures while maintaining extensibility for future modalities like eye tracking or brain-computer interfaces (BCI). By abstracting the complexity of input handling, the layer enables the framework to support accessibility-focused applications, such as the smart home controller, with minimal developer effort. It validates, enriches, and forwards events to the backend, ensuring compatibility across platforms (e.g., Flutter, SwiftUI) and facilitating Smart Intent Fusion (SIF) for personalized UI adaptations.

3.4.1 Command Interface and UI Manipulation

The Input Adapter Layer provides a standardized command interface to capture and process user interactions, ensuring consistent communication with the SIF Backend Layer. Implemented as a reusable `AdaptiveUIAdapter` class (e.g., in Flutter), the interface exposes methods like

`sendEvent(Event eventData)`

to handle inputs from various modalities. These methods translate raw inputs into structured JSON events compliant with the framework's JSON contract, which includes fields such as

`event_type, source, user_id, target_element, coordinates, confidence, and metadata.`

UI manipulation is indirectly supported by the layer through event generation. For example, a miss-tap on a button in the smart home controller generates a `miss_tap` event, which the backend processes to return adaptations like `increase_size`. The adapter ensures events are enriched with contextual metadata (e.g., tap coordinates, voice command text) to aid the backend's intent inference. The interface is designed to be platform-agnostic, allowing developers to integrate it with minimal code (e.g., 1-2 lines per input type), supporting both standard inputs (touch, keyboard) and advanced modalities (voice, gestures) via SDK configurations.

3.4.2 Event Handling

Event handling in the Input Adapter Layer is structured to process multimodal inputs efficiently and reliably:

1. **Input Capture**: The layer captures raw inputs from the Frontend Layer, such as:
   - **Touch**: Taps or miss-taps on buttons/sliders, detected via for example Flutter's `GestureDetector`. These can act as a form of bounding box collider to catch miss taps close to the UI element like a button for example.
   - **Keyboard**: Navigation (e.g., Tab) or selection (e.g., Enter) events via a `RawKeyboardListener`.
   - **Voice**: Commands like "Turn on lamp" or "Unlock door", mocked for this thesis but extensible to libraries like `speech_to_text` or Web Speech API.

- **Gestures**: Hand movements (e.g., point, swipe) via mock events, with future support for MediaPipe.
- **Future Modalities**: Eye tracking or BCI inputs, integrated via custom event handlers (e.g., `sendEyeTrackingEvent`).

2. **Event Standardization**: Each input is transformed into a JSON contract format:

```
{
  "event_type": "tap",
  "source": "touch",
  "timestamp": "2025-08-04T14:41:00Z",
  "user_id": "user_123",
  "target_element": "lamp",
  "coordinates": {"x": 100, "y": 200},
  "confidence": 0.9,
  "metadata": {"command": "turn_on"}
}
```

   The layer validates required fields `(event_type, source, timestamp, user_id)` and enriches events with optional metadata (e.g., gesture type, voice command).

3. **Event Forwarding**: Standardized events are sent to the SIF Backend Layer via WebSocket (`/ws/adapt`) for real-time processing or HTTP (`/context`) for batch operations. The adapter ensures profile existence by querying `GET /profile/{user_id}` before sending events, prompting a `POST /profile` if needed.

4. **Error Handling**: The layer detects and flags erroneous inputs (e.g., miss-tap if coordinates miss a target) and validates all inputs, as well as includes confidence scores to aid backend reasoning. It also handles network failures by queuing events locally for retry.

3.4.3 Example: UIAdapterAPI Implementation

The `AdaptiveUIAdapter` class serves as the core implementation of the Input Adapter Layer, designed as a modular template for extensibility. A typical implementation includes:

- **Initialization**: Configures the adapter with a `user_id` and a callback for handling adaptations (e.g., `onAdaptations` to apply backend responses directly to the frontend).
- **Profile Management**: Checks profile existence (`GET /profile/{user_id}`) and creates default profiles (`POST /profile`) if absent.
- **Event Sending**: Provides a `sendEvent` method to standardize and send events, taking an `Event eventData` parameter. The detailed implementation of the `Event` class and the `sendEvent` method is discussed in the next chapter.
- **Extensibility**: Supports new modalities via additional methods (e.g., `sendEyeTrackingEvent(target, gazeCoords)`) without altering any core logic.
- **Integration**: Embeds in the Frontend Layer with minimal setup (e.g., 1 line to initialize, 1 line per event).

This API minimizes developer effort while ensuring robust event handling, making it reusable across platforms and domains.

3.4.4 Figma Diagram: Expanded API View

[Figma diagram of Input Adapter Layer]

- **Description:** The diagram illustrates the flow of multimodal inputs through the Input Adapter Layer.
  - **Inputs:** Icons for touch (finger), keyboard (key), voice (mic), gesture (hand), and future modalities (eye).

  - **Processing:** A block for `AdaptiveUIAdapter`, showing input standardization into JSON contract format.
  - **Outputs:** Arrows to SIF Backend Layer via WebSocket/HTTP, labeled "Standardized JSON Events".
  - **Profile Check:** Connection to MongoDB via `GET/POST /profile`, labeled "Profile Validation".
- **Purpose:** Visualizes the layer's role as a modular bridge between the frontend and backend, emphasizing its extensibility and accessibility focus.

3.4.5 Key Design Considerations

- **Modularity:** The `AdaptiveUIAdapter` is platform-agnostic, reusable across Flutter, SwiftUI, Unity, ... with consistent event formatting.
- **Extensibility:** New modalities (e.g., eye tracking) can be added via new methods without changing existing code, leveraging the JSON contract's flexibility.
- **Accessibility:** Supports inputs tailored to:
  - **Motor-impaired:** keyboard inputs,
  - **Visually impaired:** voice inputs,
  - **Hands-free usage:** gesture-based or voice interactions,

  aligning with the thesis's accessibility focus.
- **Reliability:** Validates inputs and ensures profile existence, reducing errors in backend processing.

3.5 SIF Backend Layer: Smart Intent Fusion (SIF)

The SIF Backend Layer is the core intelligence of this framework, responsible for interpreting standardized input events, fusing user context, and generating real-time adaptation actions for the UI. It acts as the decision-making engine, continuously analyzing incoming multimodal data such as touch, voice, and gesture events alongside user profiles and interaction history. By combining rule-based logic for deterministic adaptations, advanced large language model (LLM) reasoning for context-aware and creative solutions, and a feedback-driven learning loop, the backend ensures that every UI modification is both personalized and accessibility-focused. This dynamic approach enables the system to adapt to evolving user needs, proactively enhance usability, and maintain inclusivity for motor-impaired, visually impaired, and hands-free users.

3.5.1 LLM Design for Multimodal Input Processing

The SIF Backend Layer's core innovation lies in its use of large language models (LLMs) to interpret, reason about, and adapt to multimodal user inputs in real time. Unlike traditional rule-based systems, which rely on static mappings between input events and UI adaptations, the backend leverages LLMs to provide flexible, context-aware decision-making that can generalize across diverse scenarios and user needs.

Multimodal Input Fusion: The backend receives standardized events from the Input Adapter Layer, each representing a user interaction (e.g., tap, voice command, gesture) in a uniform JSON contract. These events are enriched with metadata such as input source, confidence scores, and contextual information (e.g., user profile, device state). The LLM is prompted with this structured data, allowing it to reason about the user's intent holistically—considering not just the immediate event, but also historical patterns, accessibility needs, and UI context.

Agent-Based Architecture: To maximize flexibility and maintainability, the backend employs a multi-agent extension architecture (MA-SIF), where specialized LLM agents are responsible for different adaptation domains:

- **UI Agent:** Focuses on visual and interactive adaptations, such as increasing font size, contrast, or button scale for accessibility.

- **Geometry Agent:** Handles spatial adaptations, including element resizing, spacing adjustments, and layout simplification.
- **Input Agent:** Manages input modality switching (e.g., from touch to voice), gesture interpretation, and error recovery for hands-free or motor-impaired users.
- **Validator Agent:** Always required. Ensures that suggested adaptations are valid, non-conflicting, and compliant with accessibility guidelines. It can change or add fields, but not remove nor add adaptations.

Each agent is configured via domain-specific prompts in `sif_config.json`, enabling rapid iteration and extensibility for new adaptation strategies or modalities. This design allows the backend to evolve without major architectural changes, simply by updating agent prompts or adding new agents as needed to accommodate the developer's needs. The validation agent plays a crucial role in ensuring that adaptations proposed by the other agents do not conflict or duplicate efforts, maintaining a coherent user experience and is thus required.

Basic example of a 2-agent configuration (validator excluded):

```
{
  "ui_agent": {
    "focus": "visual adaptations",
    "model": "gemini-1.5-flash",
    "allowed_actions": ["increase_size", "increase_contrast", "
     reposition_element"],
    "prompt": "Given the event data, suggest UI adaptations for
     accessibility. Consider user profile and interaction history."
  },
  "geometry_agent": {
    "focus": "spatial adaptations",
    "model": "gemini-1.5-flash",
    "allowed_actions": ["resize_element", "adjust_spacing", "
     simplify_layout"],
    "prompt": "Analyze the UI layout and suggest spatial adaptations to
     improve usability for motor-impaired users."
  },
  "validator_agent": {
    "model": "gemini-1.5-pro",
    "allowed_actions": ["switch_to_voice", "interpret_gesture", "
     recover_from_error", "increase_size", "increase_contrast", "
     reposition_element","resize_element", "adjust_spacing", "
     simplify_layout"],
    "prompt": "Validate proposed adaptations for conflicts, duplicates
     and inconsistencies based on user context, events and interaction
     history."
  }
}
```

Contextual Reasoning and Personalization: The LLM agents do not operate in isolation; they fuse real-time events with user profiles and interaction history stored in MongoDB. This enables the backend to personalize adaptation decisions based on individual preferences, accessibility flags, and behavioral trends. For example, if a user with motor impairments repeatedly misses a button, the UI agent may suggest enlarging the button, while the input agent may recommend switching to voice mode. The validator agent then ensures these adaptations are compatible and beneficial.

Handling Ambiguity and Edge Cases: LLMs excel at interpreting ambiguous or incomplete input, making them ideal for accessibility scenarios where user intent may not be clearly expressed. The backend can handle vague voice commands, noisy gesture data, or conflicting input events by leveraging the LLM's reasoning capabilities and fallback rules. In cases where the LLM is unable to generate a confident adaptation, deterministic rule-based logic ensures baseline accessibility is maintained.

Extensibility and Future-Proofing: The agent-based LLM design allows the backend to easily incorporate new modalities (e.g., eye tracking, bio-signals) and adaptation strategies without major architectural changes. By updating agent prompts and event schemas, the system can evolve to support emerging accessibility technologies and user needs.

Summary: Overall, the LLM-driven backend transforms static UI adaptation into a dynamic, context-aware process. By fusing multimodal input, user context, and agent-based reasoning, it delivers personalized, robust, and extensible accessibility solutions for a wide range of users and platforms.

3.5.2 Rule-Based Logic

For deterministic scenarios, the backend applies rule-based logic to ensure reliability and responsiveness. Simple rules such as enlarging a button after a miss-tap or increasing slider thumb size for shaky drags are encoded to provide immediate accessibility improvements. These rules act as a safety net, guaranteeing baseline adaptations even if LLM reasoning is inconclusive or unavailable. These deterministic rules are mostly used for the **Smart Intent Fusion (SIF)** to ensure that the system can handle basic adaptations without relying on LLMs, which may be computationally expensive, slow in certain scenarios or entirely not responding. For example, a rule may specify that if a user taps a button and the tap is detected as a miss-tap, the button size should be increased by 1.5x to improve future interactions. These are simple mock rules in case of LLM timeouts or failures, ensuring that at least a basic system remains responsive and accessible even in edge cases.

3.5.3 Heatmap Analysis

To further refine adaptation decisions, the backend can analyze interaction heatmaps derived from user event logs. By aggregating tap coordinates and gesture paths, the system identifies problematic UI regions (e.g., frequently missed buttons) and adapts layouts or element sizes accordingly. This data-driven approach supports continuous improvement and personalization, especially for users with evolving accessibility needs. By using tap frequency, the backend can suggest adaptations like repositioning elements or enlarging hit areas, enhancing usability.

3.5.4 Integration with Profiles and History

User profiles, stored in MongoDB, encapsulate accessibility needs, input preferences, UI settings, and recent interaction history. Upon receiving an event, the backend retrieves the relevant profile and fuses it with real-time input to contextualize adaptation decisions. Interaction history enables the backend to learn from past behavior, supporting progressive personalization (e.g., permanently switching to voice mode after repeated motor-impaired interactions).

3.5.5 Communication and Endpoints

The backend exposes WebSocket (`/ws/adapt`) and HTTP (`/profile`, `/context`) endpoints for real-time and batch processing. Events are validated, enriched, and processed asynchronously, with adaptation actions returned to the frontend for immediate UI updates. Profile management endpoints ensure user context is always available, supporting seamless onboarding and continuous learning.

3.5.6 Key Design Considerations

- **Accessibility Focus:** All adaptation logic prioritizes WCAG-compliant features and user-specific needs.
- **Extensibility:** Modular agent design and JSON contracts allow easy integration of new modalities and adaptation strategies.
- **Reliability:** Rule-based logic ensures robust baseline adaptations; LLMs provide creative, context-aware enhancements.
- **Continuous Learning:** Feedback loops and history-driven updates enable the system to evolve with user behavior.
- **Scalability:** Asynchronous processing and MongoDB storage support high event volumes and large user bases.

3.6 User Profiles and Context Modeling

The User Profiles and Context Modeling component is a foundational element of the multimodal AI-driven framework, enabling personalized UI adaptations through **Smart Intent Fusion (SIF)**. Stored in MongoDB, user profiles encapsulate individual preferences, accessibility needs, and interaction history, providing the SIF Backend Layer with rich context to infer user intent and generate tailored adaptations. This component ensures the framework's ability to deliver accessibility-focused solutions for motor-impaired, visually impaired, and hands-free users in the Adaptive Smart Home Controller implementation. By modeling user context and maintaining a continuous learning loop, the system adapts dynamically to evolving user behaviors, enhancing usability over time.

3.6.1 Profile Structure

User profiles are stored in MongoDB's `profiles` collection, indexed by `user_id` for efficient retrieval. Each profile is a JSON document with the following structure:

- **user_id:** A unique identifier (e.g., UUID or username) to associate events and adaptations with a specific user.
- **accessibility_needs:** Boolean flags indicating specific requirements, such as `motor_impaired: true`, `visual_impaired: false`, `hands_free_preferred: true`. These guide adaptation prioritization (e.g., larger buttons for motor-impaired users).
- **input_preferences:** Specifies the preferred input modality (e.g., `preferred_modality: "voice"`), influencing mode-switching decisions (e.g., enabling voice navigation for hands-free users).
- **ui_preferences:** Stores UI settings like `font_size` (e.g., 16), `contrast_mode` (e.g., "normal"), and `button_size` (e.g., 1.0), serving as defaults for rendering and adaptation.
- **interaction_history:** A capped array (10 events) of past interactions of the user with the UI. This stores all the `Event` data for those 10 events. This supports continuous learning by tracking user behavior.

Example of a user profile:

```
{
  "user_id": "user_123",
  "accessibility_needs": {
    "motor_impaired": true,
    "visual_impaired": false,
    "hands_free_preferred": true
  },
  "input_preferences": {
    "preferred_modality": "voice"
  },
  "ui_preferences": {
    "font_size": 16,
    "contrast_mode": "normal",
    "button_size": 1.0
  },
  "interaction_history": [
    {"event_type": "miss_tap", "target_element": "lamp",
    "timestamp": "2025-08-04T14:41:00Z"},
    {"event_type": "voice", "target_element": "lock",
    "metadata": {"UI_element": "button", "command": "unlock"}, "
     timestamp": "2025-08-04T14:42:00Z"}
  ]
}
```

3.6.2 Context Modeling and Usage

Context modeling integrates user profiles with real-time events to provide a comprehensive view for **Smart Intent Fusion**:

- **Profile Retrieval:** On receiving an event (via `/ws/adapt` or `/context`), the backend queries MongoDB for the user's profile using `user_id`. If absent, a default profile is created via `POST /profile`.
- **History Management:** Interaction history is updated asynchronously using MongoDB's `$push` and `$slice` operators to append new events while capping at 20 entries for efficiency. This ensures relevant context without excessive storage.
- **Intent Contextualization:** The SIF Backend Layer uses profile data (e.g., `motor_impaired: true`) and history (e.g., frequent miss-taps) to inform LLM reasoning.
  - For example, repeated miss-taps on the Thermostat slider for a motor-impaired user may trigger `increase_slider_size` or `switch_mode: voice`.
- **Continuous Learning:** Events are logged in the history, enabling the system to refine future suggestions.
  - For example, permanently switching to voice mode after increasing button size multiple times.

3.6.3 Storage and Access

- **MongoDB Storage:** Profiles are stored in the profiles collection, with `user_id` indexed for fast lookups. The logs collection separately stores event-adaptation pairs for evaluation and traceability, also mirrored in `adaptation_log.jsonl`.
- **Access Patterns:** The backend retrieves profiles on-demand for each event, caching frequently accessed profiles in memory (future optimization). Updates via `PUT /profile/user_id` use asynchronous `BackgroundTasks` for non-blocking writes, with MongoDB transactions ensuring atomicity for concurrent updates.
- **Edge Case Handling:** 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.

3.6.4 Role in Accessibility

User profiles are critical for accessibility-focused adaptations:

- **Motor-Impaired Users:** Profile flags trigger larger buttons/sliders or voice mode (e.g., `switch_mode: voice` after miss-taps).
- **Visually Impaired Users:** `visual_impaired: true` prompts high-contrast text/buttons and larger fonts.
- **Hands-Free Users:** `hands_free_preferred: true` prioritizes voice/gesture inputs, with adaptations like `simplify_layout` to reduce UI complexity.

3.6.5 Key Design Considerations

- **Scalability:** MongoDB's indexing and capped history ensure efficient storage and retrieval for large user bases.
- **Privacy:** Profiles are anonymized by `user_id`; sensitive data is encrypted in production (future).
- **Extensibility:** The JSON structure supports new fields (e.g., `bio_signals`) for future modalities.
- **Continuous Learning:** History-driven learning refines adaptations, aligning with the thesis's focus on dynamic personalization.

3.7 Dynamic Adaptation Mechanisms

The Dynamic Adaptation Mechanisms component of the multimodal AI-driven framework enables real-time, personalized UI modifications based on user interactions, profiles, and context. Integrated within the SIF Backend Layer, these mechanisms use Smart Intent Fusion (SIF) to process standardized events from the Input Adapter Layer, combining rule-based logic, LLM reasoning, and interaction history to generate and apply adaptations. The mechanisms operate through a feedback loop, continuously refining UI behavior to enhance usability for the user, as demonstrated in the Adaptive Smart Home Controller in the next chapter. This section outlines the adaptation process, supported actions, and the continuous learning loop that ensures progressive personalization.

3.7.1 Adaptation Process

The adaptation process follows a structured lifecycle from input capture to UI update, ensuring seamless and accessibility-focused modifications:

1. **Input Capture and Standardization:** As described earlier the Frontend Layer captures user interactions (e.g., tap, voice command, gesture) via UI elements (buttons, sliders, text) in the smart home controller. The Input Adapter Layer standardizes these into JSON contract events (e.g., `Event {"event_type": "miss_tap", "target_element": "lamp"}`), which are sent to the SIF Backend Layer via WebSocket (`/ws/adapt`) for real-time processing or HTTP (`/context`) for batch operations.
2. **Context Fusion:** The SIF Backend Layer retrieves the user's profile (e.g., `motor_impaired: true`) and recent interaction history (capped at 10 events) from MongoDB. Smart Intent Fusion combines the current event, profile, and history to infer user intent. For example, a miss-tap on the Lock button paired with a voice command ("Unlock") for a motor-impaired user suggests intent to toggle the lock.
3. **Adaptation Generation:** SIF employs:
   - **Rule-Based Logic:** Deterministic rules (e.g., `if miss_tap then increase_size`) provide fast, reliable adaptations for common scenarios (mostly used for mock or backup).
   - **LLM Reasoning:** The Gemini flash AI model infers complex intents, generating creative adaptations (e.g., `switch_mode: voice` for repeated miss-taps). The LLM prompt ensures JSON outputs align with predefined actions (e.g., `{"action": "increase_size", "target": "lamp", "value": 1.5}`).
   - **Heatmap Analysis:** Simulated via history counts, prioritizes frequently interacted elements (e.g., repositioning Thermostat card after multiple taps).
4. **Adaptation Application:** The Frontend Layer receives JSON adaptations and applies them dynamically to the target UI elements.
5. **Feedback Loop:** Events are logged in MongoDB's `logs` collection and `adaptation_log.jsonl`, as well as appended to the user's interaction history. This informs future adaptations and enabling continuous learning (e.g., permanent size increase after frequent miss-taps).

3.7.2 Supported Adaptation Actions

The framework supports a set of predefined actions designed for accessibility in the Adaptive Smart Home Controller, designed to ensure compatibility with the Frontend Layer's rendering capabilities and partial alignment with WCAG 2.1 guidelines (e.g., 1.4.3 Contrast Minimum, 2.5.5 Target Size). These actions are generated by Smart Intent Fusion (SIF) and its multi-agent extension (MA-SIF), leveraging rule-based logic, Gemini LLM reasoning, user profiles and interaction history to address the needs of motor-impaired, visually impaired, and hands-free users.

- **increase_button_size**: Scales buttons (e.g., Lamp's On/Off toggle) by a factor (e.g., 1.5x) to increase hit areas, aiding motor-impaired users with dexterity challenges. Applied via smooth animations (e.g., Flutter's `AnimatedScale`) for seamless user experience.
- **increase_font_size**: Increases text size (e.g., status "Locked" from 16pt to 18pt) to improve readability for visually impaired users, ensuring compliance with WCAG's text resizing guidelines.

- **increase_slider_size**: Enlarges slider thumbs (e.g., Thermostat's temperature control) to enhance precision for motor-impaired users, using custom rendering for visual feedback.
- **increase_contrast**: Switches to high-contrast mode (e.g., black backgrounds for buttons and text) to improve visibility for visually impaired users, aligning with WCAG contrast requirements.
- **highlight_border**: Applies a bolder border to buttons, aiding visually impaired and motor-impaired users by enhancing element visibility (WCAG 2.4.7 Focus Visible).
- **adjust_spacing**: Increases spacing between UI elements (e.g., device cards in the ListView) to reduce accidental taps for motor-impaired users, improving target accessibility (WCAG 2.5.5 Target Size).
- **show_tooltip**: Displays contextual tooltips (e.g., "Try saying 'Unlock'") near UI elements when modality struggles are detected (e.g., repeated miss-taps), guiding hands-free or motor-impaired users to alternative inputs like voice (WCAG 2.5.3 Label in Name).
- **switch_mode**: Shifts to keyboard, voice or gesture navigation mode for hands-free users (e.g., after repeated miss-taps), enabling more seamless interaction without physical input.
- **trigger_button**: Automatically activates buttons (e.g., toggles Lamp to "On") when intent is clear from multimodal inputs (e.g., voice "Turn on" + gesture), reducing interaction barriers.
- **simplify_layout**: Reduces UI complexity (e.g., fewer UI elements) for hands-free or cognitively impaired users, streamlining interaction for accessibility.

These adaptation actions form the cornerstone of the framework's accessibility-driven approach, addressing diverse user needs in the smart home controller. Each action is carefully designed to align with WCAG 2.1 guidelines, ensuring inclusivity for motor-impaired users (e.g., larger buttons/sliders, increased spacing), visually impaired users (e.g., high-contrast modes, highlighted borders), and hands-free users (e.g., tooltips, mode switching).

3.7.3 Continuous Learning and Feedback Loop

A core strength of the framework lies in its continuous learning and feedback loop, which enables dynamic refinement of adaptation strategies over time. Every user interaction is systematically logged in MongoDB (within the `logs` collection and mirrored in `adaptation_log.jsonl`), ensuring that contextual details—such as a miss-tap on the Thermostat—are persistently captured for analysis. These events are not only stored for traceability but are also appended to the user's profile history, maintained as a capped array to preserve the most recent interactions while optimizing storage and retrieval efficiency.

This historical data serves as a foundation for the learning mechanism. The large language model (LLM) regularly analyzes patterns within the interaction history, identifying trends such as frequent miss-taps or repeated modality switches. By recognizing these behavioral signals, the system can proactively suggest and implement permanent UI adaptations, such as simplifying the layout or switching to a more suitable input mode for the user.

Furthermore, the feedback loop supports ongoing evaluation of the framework's effectiveness. By correlating logged events with adaptation outcomes, the system can compute metrics like adaptation accuracy (e.g., the proportion of correct button triggers) and user performance improvements (e.g., reduction in miss-taps over time). This data-driven approach not only validates the impact of adaptations but also guides future refinements, ensuring that the framework evolves in response to real user needs and behaviors.

3.7.4 Key Design Considerations

- **Accessibility Focus:** Adaptations prioritize WCAG compliance (e.g., high contrast, large hit areas) to support motor-impaired, visually impaired, and hands-free users.
- **Real-Time Performance:** WebSocket ensures low-latency adaptations, critical for seamless user experience in the smart home controller.
- **Reliability:** Rule-based fallback complements LLM reasoning, ensuring functionality during API failures.

- **Scalability:** Async MongoDB updates and indexed `user_id` queries handle high event volumes.
- **Extensibility:** The JSON contract and action set allow new adaptations (e.g., `adjust_brightness`) for future domains like healthcare.

3.8 Developer SDK and Integration

The Developer SDK and Integration component of the multimodal AI-driven framework is designed to simplify the adoption of Smart Intent Fusion (SIF) for developers building accessibility-focused applications across platforms such as Flutter, SwiftUI, and future environments like Unity for VR/AR. The SDK provides a modular, developer-friendly interface to integrate the framework's core functionalities capturing multimodal inputs, sending standardized events, managing user profiles, and applying dynamic UI adaptations all while minimizing integration effort. By abstracting the complexities of the Input Adapter Layer and SIF Backend Layer, the SDK ensures seamless interaction with the Adaptive Smart Home Controller and supports extensibility for new modalities and domains. This section outlines the SDK's structure, integration process, and its role in the possibility of enabling rapid development of adaptive UIs.

3.8.1 SDK Structure

The SDK is implemented as a lightweight, platform-agnostic library, with the primary implementation in Flutter as the AdaptiveUIAdapter class for this thesis. Its structure includes:

- **Initialization Module:** Configures the SDK with a `user_id` and backend connection details (e.g., WebSocket URL: `ws://localhost:8000/ws/adapt`, HTTP URL: `http://localhost:8000`). It establishes a persistent WebSocket connection for real-time event processing and provides callbacks for handling adaptation responses.
- **Event Handling API:** Exposes methods like `sendEvent(eventType, source, targetElement, metadata)` to capture and standardize multimodal inputs (e.g., touch, keyboard, voice, gestures) into the JSON contract format.
  - For example: a tap on the Lamp's On/Off button is sent as {`"event_type": "tap", "source": "touch", "target_element": "lamp"`}.
- **Profile Management API:** Includes methods for profile operations:
  - `checkProfile()` queries `GET /profile/{user_id}`
  - `createProfile(profileData)` uses `POST /profile`
  - `updateProfile(profileData)` uses `PUT /profile/{user_id}`

  This ensures user profiles are available before sending events.
- **Adaptation Application:** Processes backend responses (e.g., {`"action": "increase_size", "target": "lamp", "value": 1.5`}) and applies them to UI elements (e.g., scaling buttons, adjusting text size) via platform-specific rendering (e.g., Flutter's `AnimatedScale`).
- **Extensibility Hooks:** Supports new modalities (e.g., eye tracking via `sendEyeTrackingEvent`) by adding custom event handlers without modifying core logic.

The SDK is designed to require minimal setup typically 1-2 lines of code for initialization and 1 line per event, making it accessible to developers with varying expertise.

3.8.2 Integration Process

Integrating the SDK into an application involves a straightforward process, demonstrated in the next chapter of implementing the smart home controller:

1. **Setup:** Developers import the SDK and initialize it with a `user_id` and optional callback for adaptations.
   For example, in Flutter:
   `final adapter = AdaptiveUIAdapter('user_123', onAdaptations: applyAdaptations);`

2. **Profile Check:** On app startup, the SDK calls `checkProfile()` to verify profile existence via `GET /profile/{user_id}`. If absent, `createProfile()` sends a default profile (e.g., with `motor_impaired: true`) via `POST /profile`.
3. **Event Capture:** Developers attach SDK methods to UI interactions.
   For example:
   - A button tap: `adapter.sendEvent(Event('tap', 'touch', 'lamp'))`
   - A mock voice command:
     `adapter.sendEvent(Event('voice', 'voice', 'lock', metadata: {'command': 'unlock'}))`
4. **Adaptation Application:** The SDK receives JSON adaptations via WebSocket and invokes the callback to apply changes (e.g., scaling a button for `increase_size`, changing text color for `increase_contrast`).

3.8.3 Cross-Platform and Multiple domain Support

The SDK is designed to be platform-agnostic, ensuring compatibility across:

- **Flutter:** Primary implementation for the demo, leveraging `web_socket_channel` and `http` for communication, with `AnimatedScale` for smooth adaptations.
- **SwiftUI:** Uses `URLSession` for HTTP and WebSocket libraries, with native animations for UI changes.
- **Future Platforms (e.g., Unity):** Extensible to VR/AR via similar event-handling abstractions, supporting 3D manipulations (e.g., `transform.localScale`).

The JSON contract ensures consistent event formats across platforms, while the SDK's modular design allows platform-specific rendering without altering backend logic. Furthermore the framework's generalizability extends to diverse application domains beyond smart homes, such as healthcare, gaming, and education:

- **Healthcare:** The framework can adapt medical device interfaces by enlarging sliders for motor-impaired users or simplifying layouts for elderly patients. The same JSON contract and adaptation actions (e.g., `increase_size`, `switch_mode`) apply with domain-specific UI elements.
- **Gaming:** Adaptive hitboxes or controls (e.g., larger buttons for novice players) can be implemented using the same event adaptation pipeline, with history driven personalization (e.g., adjusting based on frequent misses).
- **Education:** E-learning interfaces can use `increase_font_size` or `simplify_layout` for learners with visual or cognitive impairments, leveraging voice/gesture inputs for hands-free navigation.

3.8.4 Key Design Considerations

- **Developer-Friendly:** Minimal integration effort reduces adoption barriers.
- **Scalability:** WebSocket and HTTP endpoints handle high event volumes, async profile updates ensure performance.
- **Reliability:** Local event queuing and profile checks mitigate network or backend failures.
- **Accessibility Focus:** Aligns with WCAG guidelines (e.g., larger hit areas, high contrast) for inclusive design.
- **Extensibility:** JSON contract and modular API support new platforms and modalities (e.g., BCI).

3.9 Design Decisions

The design of the multimodal AI-driven framework for dynamic UI adaptation reflects a series of strategic decisions to balance accessibility, performance, scalability, extensibility, and developer-friendliness. These decisions prioritize delivering real-time, personalized adaptations for motor-impaired, visually impaired,

and hands-free users while ensuring the system is modular and adaptable to future platforms and domains. This section outlines the design decisions and their reasoning, highlighting how they align with the thesis's focus on Smart Intent Fusion (SIF) and accessibility-driven UI adaptation.

3.9.1 Modularity Over Monolithic Design

Decision: The framework adopts a modular, three-layered architecture (Frontend, Input Adapter, SIF Backend) with clear separation of concerns, using standardized JSON contracts for communication.

Reasoning:

- **Flexibility:** Each layer (e.g., Flutter-based Frontend, FastAPI Backend) can be updated or replaced independently, supporting diverse platforms like SwiftUI or Unity.
- **Extensibility:** The Input Adapter Layer's JSON contract allows new modalities (e.g., eye tracking) without altering backend logic.
- **Developer-Friendly:** Modularity simplifies integration, with the SDK requiring minimal code for event handling and adaptation application.

3.9.2 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 (`/context, /profile`) for batch operations 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`) and batch event processing, 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.

3.9.3 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:** Capped history supports SIF's learning loop, enabling personalized adaptations (e.g., permanent button size increase after frequent miss-taps).

3.9.4 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).

3.9.5 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.

3.10 Chapter Summary

This chapter has detailed the system design and architecture of the multimodal AI-driven framework for dynamic UI adaptation, emphasizing its role in delivering personalized, accessibility-focused solutions for motor-impaired, visually impaired, and hands-free users. The framework's modular, three-layer architecture (Frontend Layer, Input Adapter Layer and SIF Backend Layer) ensures seamless integration of multimodal inputs (e.g., touch, voice, gestures) and real-time UI adaptations, which will be demonstrated in the implementation of an Adaptive Smart Home Controller in the next chapter.

The architecture's design principles: modularity, scalability, generalizability and accessibility enable the laying of foundation for expansion to domains like healthcare and gaming. Communication via WebSocket and HTTP ensures low-latency, reliable adaptations, while MongoDB and the JSON contract provide scalability and extensibility. These elements collectively help addressing real-world accessibility needs with potential for future research into specialized UI-adapting models.

Chapter 4

Smart Intent Fusion (SIF)

Chapter 5

An Adaptive Multimodal GUI Framework using LLMs

5.1 Introduction to an Adaptive Smart Home Controller

The Adaptive Smart Home Controller serves as a practical implementation of the multimodal AI-driven framework, demonstrating its capabilities in delivering personalized, accessibility-focused UI adaptations. This chapter outlines the design and implementation of the frontend as well as the adapter, which integrates various input modalities (touch, voice, gestures) to control smart home devices (e.g. lights, thermostats, ...) while connecting with the backend for adapting the UI dynamically to user needs. The Smart Home Controller is built fully using Flutter and Dart.

5.1.1 Development Environment

As development environment, Flutter was chosen as frontend framework for its cross-platform capabilities, allowing the same codebase to run on Android, iOS devices and desktop environments. Without the need of platform-specific adaptations. Furthermore, Flutter's rich set of pre-built UI widgets and its reactive programming model facilitate the creation of responsive, adaptive interfaces that can dynamically adjust. These UI widgets can be easily modified from the ground up if needed, giving the developer full control over the UI design and accessibility. The hot reloading feature of Flutter also speeds up the development process by allowing developers to see changes in real-time without restarting and recompiling the entire application, which is particularly useful for this iterative design nature of the app as well as testing and evaluating different configurations of the adaptive UI components. Python is used for the backend, using FastAPI for building the RESTful API and WebSocket server, and MongoDB for storing user profiles and interaction history.

The development environment includes:

- **Operating System:** MacOS 15.6 for development, with the application also tested on Windows 11 to ensure cross-platform compatibility.
- **Flutter SDK:** Version 3.7.0 or higher, providing the necessary tools and libraries for building cross-platform applications.
- **Dart Language:** Version 2.19.0 or higher, the programming language used for Flutter development.
- **Python:** Version 3.9 or higher for running the FastAPI server and managing backend logic.
- **IDE:** Visual Studio Code with Flutter and Dart plugins installed for code editing, debugging, and testing.

5.1.2 Testing workflow

The main environment for coding and testing used was MacOS, with the application being tested on Windows as well to make sure it works seamlessly across desktop platforms. But given the code's

simplicity and inherent design to be platform-agnostic, it ensures that the application can be easily adapted to other platforms like iOS or Android with minimal changes. Version control is managed using Git, with the repository hosted on GitHub for collaboration and version tracking. The app's accessibility features are fully tested on device. For backend automation, a script is available that can be run to set up the backend environment, including the FastAPI server and MongoDB database, ensuring that the frontend can communicate with the backend seamlessly.

5.1.3 Tools and Libraries

The development of the Adaptive Smart Home Controller uses several tools and libraries to help with the implementation of the multimodal AI-driven GUI. The main tools and libraries include:

- **Flutter SDK:** The primary framework for building the cross-platform application, providing a rich set of UI components and tools for responsive design.
- **WebSocket Channel:** For real-time bidirectional communication with the SIF Backend Layer, enabling low-latency event processing and adaptation delivery.
- **HTTP Package:** For making RESTful API calls to manage user profiles and retrieve adaptation logs from the backend.
- **Provider Package:** For state management, allowing the application to reactively update UI components based on user interactions and backend responses.
- **Uvicorn:** Version 0.20.0 or higher for serving the FastAPI application, providing a high-performance ASGI server.
- **FastAPI:** Version 0.95.0 or higher for building the backend API, providing a fast and efficient way to handle HTTP requests and WebSocket connections.
- **MongoDB:** Version 6.0 or higher for storing user profiles and interaction history, ensuring efficient data retrieval and storage.

5.2 Frontend Implementation: Smart Home Controller

5.3 Input Adapter Layer: Multimodal Input Handling

5.4 SIF Backend Layer: Implementation of Adaptation Logic

5.5 User Profile and Context Implementation

5.6 Dynamic Adaptation Mechanisms: Implementation Details

5.7 Developer SDK: Implementation and Integration

5.8 Cross-Platform Implementation Considerations

5.9 Implementation Challenges and Solutions

5.10 Chapter Summary

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

Chapter 8

Conclusion

8.1 Summary of Contributions

8.2 Final Remarks