

**JARVIS: DEVELOPMENT OF A LOCAL AI-POWERED PERSONAL ASSISTANT
FOR SEAMLESS INTEGRATION WITHIN THE APPLE ECOSYSTEM**

BY
LOW JIA HAO

A REPORT
SUBMITTED TO
Universiti Tunku Abdul Rahman
in partial fulfillment of the requirements
for the degree of
BACHELOR OF COMPUTER SCIENCE (HONOURS)
Faculty of Information and Communication Technology
(Kampar Campus)

FEBRUARY 2026

COPYRIGHT STATEMENT

© 2025 Low Jia Hao. All rights reserved.

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express thanks and appreciation to my supervisor, Ts Dr Saw Seow Hui and my moderator, Dr Png Wen Hao who have given me a golden opportunity to involve in the LLM and Agentic AI study. Besides that, they have given me a lot of guidance in order to complete this project. When I was facing problems in this project, the advice from them always assists me in overcoming the problems. Again, a million thanks to my supervisor and moderator.

Other than that, I would like to thank my project teammate, Benjamin Lian Wen Kang who has provided a lot of assistance to me when completing this project. Although both of us are having different project and task scope, he is still willing to support me when I faced difficulties in developing this project.

ABSTRACT

This project introduces JARVIS, a powerful, fully localized, and privacy-focused AI personal assistant which built on Apple ecosystem (iOS, macOS, and iPadOS). While modern LLMs possess a high level of intelligence, they are fundamentally limited by cloud reliance, posing significant risks to user privacy and reliability in offline environments. This research addresses these limitations by developing a unified multimodal agentic pipeline utilizing the MLX framework to deploy a 4-bit quantized Qwen3-4B model entirely on-device. The core innovation of JARVIS lies in its Reason + Action (ReAct) agentic workflow, which transitions the system from a passive chatbot to an active autonomous agent. By implementing a zero-shot tool-calling orchestration layer, the LLM can dynamically reason through user queries to execute HomeKit commands for local hardware control and perform anonymized internet searches via the Brave Answer API. To ensure privacy, the agent employs a "minimal intent" query refiner to mitigate data leakage during external API calls. The multimodal interface is powered by WhisperKit for real-time, streaming STT across 99 languages, while the TTS engine utilizes Apple's Natural Language framework to perform per-sentence language identification, ensuring seamless synchronization with neural voice models. Through deep optimization for Apple Silicon's Unified Memory Architecture, the system achieves an industry-leading throughput, peaking at 13 TPS with an average of 11 TPS, ensuring real-time responsiveness that exceeds human speech rates. Experimental evaluations confirm that JARVIS maintains stability over multi-turn conversation using KV-cache management, successfully bridging the gap between high-level reasoning and local hardware integration. This project also serves as a proof-of-concept for "Sovereign AI", by demonstrating that complex, multimodal, and tool-augmented intelligence can be executed locally without compromising performance or user data sovereignty.

Area of Study: Artificial Intelligence, Edge Computing

Keywords: Local LLM, MLX Framework, ReAct Agentic Workflow, Apple Silicon Optimization, On-device AI, Privacy-Preserving Systems, HomeKit Integration, WhisperKit, NLP.

TABLE OF CONTENTS

TITLE PAGE	i
COPYRIGHT STATEMENT	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	ix
LIST OF SYMBOLS	x
LIST OF ABBREVIATIONS	xi
CHAPTER 1 INTRODUCTION	1
1.1 Problem Statement and Motivation	1
1.2 Objectives	2
1.3 Project Scope and Direction	3
1.4 Contributions	4
1.5 Report Organization	5
CHAPTER 2 LITERATURE REVIEW	6
2.1 Review of the Technologies	6

2.1.1	Hardware Platform: Apple Silicon and Unified Memory	7
2.1.2	Firmware/OS: iOS and Swift Native Ecosystem	10
2.1.3	Database: Local Model Caching and Context Persistence	12
2.1.4	Programming Language: Swift and Metal Shaders	14
2.1.5	Algorithm: Transformer Architecture and ReAct Workflow	16
2.1.6	Summary of the Technologies Review	18
2.2	Review of the Existing Systems/Applications	21
2.2.1	Existing System A: Apple Siri	22
2.2.2	Existing System B: OpenAI ChatGPT Mobile	23
2.2.3	Existing System C: Local LLM Applications	24
2.2.4	Existing System D: Dedicated AI Hardware	25
2.2.5	Summary of the Existing Systems	26
CHAPTER 3 SYSTEM METHODOLOGY/APPROACH		27
3.1	System Design Diagram/Equation	27
3.1.1	System Architecture Diagram	27
3.1.2	Use Case Diagram and Description	29
3.1.3	Activity Diagram	34

CHAPTER 4 SYSTEM DESIGN	37
4.1 System Block Diagram	37
4.2 System Components Specifications	39
4.2.1 Large Language Model (LLM) Specifications	39
4.2.2 Speech-to-Text (STT) Specifications	39
4.2.3 Text-to-Speech (TTS) Specifications	39
4.2.4 External Tool Integrations	39
4.3 System Components Design	40
4.3.1 Models Layer	41
4.3.2 Services Layer (The Action Effectors)	43
4.3.3 ViewModels Layer (The Brain)	45
4.3.4 Views Layer (The User Interface)	47
4.4 System Components Interaction Operations	50
4.4.1 Initialization and Discovery Operation	51
4.4.2 The Voice-First Interaction Flow	53
4.4.3 ReAct Interruption and Tool Execution	55
4.4.4 Text-Based Interaction and History Management	57
 CHAPTER 5 SYSTEM IMPLEMENTATION	 59
5.1 Hardware Setup	60
5.2 Software Setup	62
5.3 Setting and Configuration	63
5.4 System Operation (with Screenshot)	64
5.5 Implementation Issues and Challenges	69
5.6 Concluding Remark	70
 CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION	 71
6.1 System Testing and Performance Metrics	71
6.2 Testing Setup and Result	72
6.2.1 Tool Calling Success Rate (Online Search)	72
6.2.2 Latency and TTFT Analysis	72
6.2.3 Linguistic Accuracy and Word Error Rate (WER)	73

6.3	Project Challenges	74
6.4	Objectives Evaluation	75
6.5	Concluding Remark	76
CHAPTER 7 CONCLUSION AND RECOMMENDATION		77
7.1	Conclusion	77
7.2	Recommendation	78
REFERENCES		79
POSTER		82

LIST OF FIGURES

Figure Number	Title	Page
Figure 3-1-1	System Architecture Diagram	27
Figure 3-1-2	JARVIS System Use case Diagram	29
Figure 3-1-3	Conversation Module Use Case Diagram	30
Figure 3-1-4	Homekit Tools Execution System Use Case Diagram	31
Figure 3-1-5	Model Management Use Case Diagram	32
Figure 3-1-6	Web Search Tool Execution System Use Case Diagram	33
Figure 3-1-7	JARVIS System Activity Diagram	34
Figure 4-1-1	JARVIS System Block Diagram	37
Figure 4-3-1	JARVIS Project Directory Structure	40
Figure 4-3-2	Model Layer Interaction Diagram	41
Figure 4-3-3	Service Layer Interaction Diagram	43
Figure 4-3-4	ViewModels Layer Interaction Diagram	45
Figure 4-3-5	Views Layer Interaction Diagram	47
Figure 4-4-1	Initialization and Discovery Operation Sequence Diagram	51
Figure 4-4-2	Initialization and Discovery Operation Flowchart	53
Figure 4-4-3	ReAct Interruption and Tool Execution flowchart	55
Figure 4-4-4	Text-Based Interaction and History Management flowchart	57
Figure 5-4-1	Downloading LLM Model	64
Figure 5-4-2	Loading LLM Model	64
Figure 5-4-3	Assistant idle page after STT, LLM, homekit is Ready	65
Figure 5-4-4	STT Detecting Voice	66
Figure 5-4-5	STT Detected Word	66
Figure 5-4-6	LLM Processing User Query	67
Figure 5-4-7	LLM response with tools return data	68

LIST OF TABLES

Table Number	Title	Page
Table 2-1-1	Comparison Table for Existing Framework/Technologies	18
Table 2-2-1	Comparison Table for Existing System	26
Table 5-1-1	Specifications of Phone	60
Table 5-1-2	Specifications of Laptop	60

LIST OF SYMBOLS

LIST OF ABBREVIATIONS

<i>AI</i>	Artificial Intelligent
<i>ANE</i>	Application Programming Interface
<i>API</i>	Apple Neural Engine
<i>CPU</i>	Central Processing Unit
<i>GB</i>	Gigabyte
<i>GB/s</i>	Gigabyte-per-second
<i>GPU</i>	Graphic Processing Unit
<i>HCI</i>	Human-Computer Interaction
<i>IDE</i>	Integrated Development Environment
<i>LAM</i>	Large Action Models
<i>LLM</i>	Large Language Model
<i>ML</i>	Machine Learning
<i>NLP</i>	Natural Language Processing
<i>OS</i>	Operating System
<i>RAG</i>	Retrieval-Augmented Generation
<i>RAM</i>	Random Access Memory
<i>SPM</i>	Swift Package Manager
<i>STT</i>	Speech-To-Text
<i>TOPS</i>	Trillions of Operations Per Second
<i>TPS</i>	Tokens Per Second
<i>TTFT</i>	Time to First Token
<i>TTS</i>	Text-to-Speech
<i>UMA</i>	Unified Memory Architecture
<i>VAD</i>	Voice Activity Detection

Chapter 1 Introduction

1.1 Problem Statement and Motivation

Current mobile AI assistants face a critical crisis of dependency. Most modern voice assistants require a constant cloud connection. They send sensitive user data to external servers for processing. This creates massive privacy risks for the average user. Data can be intercepted during transmission. Large corporations can log user intents for targeted advertising. User privacy is effectively traded for intelligence. Furthermore, cloud dependency makes these tools useless in offline scenarios. If the internet fails, the assistant ceases to function. This is unacceptable for critical home automation tasks. Local smart homes should work even without a global connection.

Standard system assistants like Siri often lack deep reasoning. They follow rigid, hard-coded scripts. They struggle to understand complex, multi-turn instructions. They cannot effectively use external tools to solve problems. Modern LLMs offer a solution. However, running these models on mobile devices is a huge challenge. Mobile hardware has limited RAM and thermal constraints. Most existing local LLM apps are just basic text interfaces. They do not integrate with the actual physical environment. They cannot control home devices or read real-time sensor data locally.

The motivation for this project is to bridge this gap. We aim to build a "Sovereign AI" agent named JARVIS. It must run fully locally on existing consumer hardware. We chose the iPhone 12 Pro Max as our primary testbed. This device has only 6GB of RAM. This is a very tight memory budget for modern AI. We want to prove that advanced intelligence is possible on older hardware. We want to provide a voice-first assistant with natural feels experience. The project is motivated by the need for speed, privacy, and true utility.

1.2 Objectives

This project aims to achieve several technical milestones. The following objectives guide our development process.

First is develop a native Swift-based multimodal assistant. The system must integrate speech recognition, reasoning, and speech synthesis. Every part of this pipeline must reside on the device.

Secondly is optimize the Qwen3-4B-4bit model for the MLX framework. We aim to reach a high inference speed on the A14 Bionic chip. Our target is to maintain a stable 11 to 13 TPS.

Third is implement a ReAct (Reason + Action) agentic workflow. The assistant must be able to think before it acts. It will use structured JSON to decide which tools to trigger.

Continue with integrate with local HomeKit control and monitoring. The system will read data from temperature sensors and can control light switches without using any cloud bridge.

Next we also focus on privacy-preserving internet search capabilities. The system will use the Brave Answer API for real-time information. The LLM must filter out user identifiers before searching.

Lastly is multilingual code-switching speech transcription. The system must recognize input in various languages. It should translate and process the intent to respond in English or native language. This ensures a consistent personality for the assistant.

1.3 Project Scope and Direction

The scope of this project is focused build on the Apple ecosystem. We targeting OS are iOS, iPadOS, and macOS. Our hardware for the project and benchmarking is with the iPhone 12 Pro Max. This device uses A14 Bionic chip which have Neural Engine and has 6GB of RAM. We utilize the MLX Swift framework as our foundational engine. The chosen model is the Qwen3-4B-4bit. We selected this model because it excels at tool-calling. Other models like Llama-3.2 3B were found to be insufficient for structured tasks.

The language scope is specifically focused on English and Chinese. While the underlying Whisper model supports 99 languages, our testing is limited to these two. The system is designed to handle "Code-switching." This means users can speak a mix of English and Chinese. The assistant will transcribe both accurately. If mix language, then it will reasoning in English and output its final answer in English. If the spoken language is in single language, it should output at the spoken language.

Our HomeKit scope includes one light and one temperature sensor. We use the HomeKit Accessory Simulator for testing.

The direction of the project is "Privacy-First." Only the Brave Search tool requires an active connection. All other functions are 100% offline or local network usage only.

1.4 Contributions

This project makes several unique contributions to the field of mobile AI.

First is high-performance MLX deployment on A14 Bionic chipset. We achieved an average of 11 tokens per second on a 6GB RAM device. This demonstrates the viability of agentic models on older mobile hardware.

Secondly is adaptive Language Mirroring Logic. We developed a pipeline that intelligently decides the output language. It detects code-switching patterns in real-time so this will provides a more human-like conversational experience compared to rigid single-language systems.

Thirdly is local HomeKit Bridge: We created HomeKit tools for LLM to use on it. This allows a user to say "open the fan in the kitchen". The AI will check the sensor and turn on the fan with correct tools parameter can tools calling.

Fourth is privacy-first search refinement. We introduced a method for "Query Distillation." The LLM generates the most generic search query possible. This prevents user intent from being leaked to the Brave Search API.

1.5 Report Organization

This report is divided into seven structured chapters. Chapter 1 provides the foundation and objectives of the project. Chapter 2 reviews the current state of local AI technologies. It covers the MLX framework and the evolution of the Qwen3 model family. Chapter 3 details our system methodology. It explains the ReAct workflow and the multimodal architecture. Chapter 4 describes the system design. This includes the UI layout, the tool-calling schema, and the HomeKit tools structure. Chapter 5 explains the implementation process. It covers the 4-bit quantization and the Swift concurrency setup. Chapter 6 presents the evaluation and results. We analyze the TPS data and the language detection accuracy. Chapter 7 concludes the project and offers suggestions for future work.

Chapter 2 Literature Review

2.1 Review of the Technologies

JARVIS functions as an artificial intelligence personal helper. Built through merging progress in mobile processing power, software environments, learning systems, along with computational methods. This part outlines those essential parts - showing how interaction forms a secure, speech-driven built-in assistant within Apple's environment. Inside lies the A14 Bionic processor from Apple, where foundational AI operations are enabled via iOS and macOS at system level.

Consideration extends toward decisions involving publicly available tools for handling local model deployment, relying on resources such as Hugging Face; alongside coding ecosystems including MLX and Swift. Among the topics addressed are the precise algorithms shaping agent functionality - Transformer-based designs appear here, alongside methods like ReAct reasoning cycles.

Attention goes toward advanced approaches in speech analysis together with compact modeling strategies meant to reduce size without sacrificing function. Every piece enters review not in isolation but as a contributor within a unified system design. Constraints tied to local device processing form one boundary; responses balance those through thoughtful integration. Performance remains high even when resources do not. Efficiency emerges naturally where components interact well. Privacy gains strength because data need not leave its original environment.

2.1.1 Hardware Platform: Apple Silicon and Unified Memory

The foundation of the JARVIS project is the hardware platform which is an iPhone 12 Pro Max equipped with the Apple A14 Bionic chip and 6GB of RAM. Choosing this device as a benchmark imposed numerous constraints on every aspect of the system design, from model selection to software implementation. The core challenge lay in running a LLM within the strict memory limitations of mobile devices. The key to solving this challenge was the unique architecture of the A14 Bionic chip, particularly its UMA and dedicated machine learning accelerator, the ANE.

The A14 Bionic chip is a six-core mobile processor based on a custom ARMv8 architecture and supports 64-bit applications [6]. Its importance in the field of on-device artificial intelligence stems from its powerful 16-core ANE, which can perform up to 11 TOPS [5]. This level of performance is crucial for accelerating matrix multiplication and other linear algebra operations, which are at the core of Transformer-based neural networks such as Qwen3 [28]. The ANE is designed to efficiently perform machine learning workloads, making it an ideal target for offloading computational tasks from CPUs and GPUs. However, the raw computational power of the ANE itself is not sufficient to meet the needs; it requires a mechanism to efficiently access the parameters and intermediate data of the model. Then the UMA is the key to achieving this goal.

Unlike traditional computing architectures which set up separate memory pools for CPUs, GPUs, and other dedicated processors, Apple's UMA in its chips integrates the memory controllers of all these units into a unified memory pool [3]. For example in the iPhone 12 Pro Max, the CPU, GPU, and ANE share the same 6GB of LPDDR4X physical memory [5]. This design has a profound impact on memory management in LLM environments. When a model is loaded into memory, its weights are stored in a separate buffer, and typically require several GB for a 4 billion parameter model. So without UMA each processing unit would need to have a copy of this buffer, which would multiply the memory footprint and quickly exhaust available RAM. For example, a 4-bit quantized 4B model might take up about 2GB of space; without UMA, loading the same model onto the CPU, GPU, and ANE simultaneously would easily exceed the 6GB limit. UMA eliminates this redundancy. All three processors can read and write to the same shared memory region, thus significantly

reducing the memory overhead required for model execution [8]. This allows the entire model weight buffer to be stored in the same location for any unit to access as needed, which is a fundamental technical prerequisite for running such a large-scale model on a 6GB device. The high bandwidth of the memory subsystem which theoretically peaking at about 34GB/s for further supports the fast data transfer required for real-time inference [7].

However, even with UMA, fitting large models and their associated activations into 6GB of memory remains a significant challenge. Activations are intermediate outputs generated by each layer of a neural network during inference, and their cumulative size can be considerable, especially with long input sequences. To address this, efficient model compression techniques must be employed. This project specifies the use of a 4-bit quantized version of the Qwen3-4B model. Quantization is the process of reducing the numerical precision of model weights. The standard FP16 model uses 16 bits per weight, while the 4-bit quantized model uses only 4 bits, thus reducing the storage requirements for weights by 75% [18]. This significant reduction in size makes it possible to store static model parameters within a 6GB memory budget. Studies of Qwen3 have shown that the model is robust to this efficient quantization. This means that even if the weights are converted to a low-precision format, they can still retain most of their original functionality and precision [19]. Overall, we weigh the performance gains against the loss of precision, even though there may be a performance decrease; the improvement in accessibility for the local assistant far outweighs the loss of precision.

Beyond hardware memory capacity, software limitations also pose a significant obstacle. The standard iOS sandbox mechanism strictly limits the virtual address space an application can allocate. Running a 4-bit quantized 4B model requires approximately 2GB, plus additional buffers for activation, context, and application code. This can easily exceed these default limits. To overcome this limitation, applications must declare and obtain the permission to "increase memory limits." This permission is used to inform the OS that the application needs more memory than usual to run, allowing the OS to map large model weight buffers into its address space. In contrast, if the application without this permission will cause fail to launch or crash immediately at the process of attempting to load the model. Therefore, obtaining this permission is not merely a technical detail but crucial to the feasibility of the project on the chosen hardware.

In summary, the A14 Bionic chip provides the necessary hardware foundation for on-device LLM inference through a powerful combination of ANE and efficient UMA. UMA solves the major problem of memory duplication, while 4-bit quantization addresses the issue of absolute model size. Furthermore, increased memory allocation permissions overcome software-mandated allocation limitations and making it possible to run powerful LLMs on consumer mobile devices. Therefore, JARVIS's success is inextricably linked to the architectural decisions Apple made in its chip design.

2.1.2 Firmware/OS: iOS and Swift Native Ecosystem

The shift from purely cloud-based AI services to on-device execution marks a major transformation in mobile computing. This shift is driven by user demand for greater privacy, lower latency, and offline capabilities[11][16]. Project JARVIS is invented on this trend. JARVIS relies on the features introduced in the upcoming iOS 26 and macOS 26 operating systems. These versions of Apple's operating systems are expected to provide the necessary system-level framework to support the complex, multi-step workflows of local AI agents, encompassing system-aware operations and tool integration. This evolution reflects the industry-wide trend toward edge AI, where computation is performed closer to the data source rather than in centralized data centers[12].

Apple's AI has always placed a strong emphasis on on-device processing to protect user privacy. This philosophy is embodied in the Core ML framework which allows developers to integrate ML models directly into their applications, ensuring that user data remains on the device[13][30]. Over time, Apple has gradually expanded the capabilities of its on-device AI offerings, eventually launching Apple Intelligence across all its platforms[26]. For the JARVIS project, these new operating system versions are expected to introduce APIs and frameworks that will simplify tool calls and chained calls to complex operations.

The importance of this system-level support cannot be underestimated. JARVIS does not need to build each component from scratch. It can simply leverage these underlying operating system capabilities to achieve its goals more efficiently. There is some example which are the ability to natively execute boot generation and invocation tools within the OS simplifies the development of the ReAct agent loop [24]. The OS can manage permissions, handle inter-process communication, and coordinate operations between the LLM engine and external system services such as HomeKit or web search APIs. This tight integration is key advantage over previous generations of assistants and is crucial for creating a seamless user experience. In addition, the new OS is expected to include optimizations such as enabling generative models to run faster and more efficiently on devices. This is essential for maintaining the responsiveness of the real-time conversational agent [20].

Although the project specifications point to iOS 26 and macOS 26, it should be noted that these versions may just be placeholders reserved for the next major version cycle, WWDC 2026. There is speculation that Apple may launch a new “Core AI” framework to supplement or eventually replace Core ML. This framework would make it easier for developers to integrate a wider range of third-party AI models and frameworks [33]. If this framework can be released ahead of time, it would be very beneficial to JARVIS. This is because JARVIS relies on community-proven open-source models rather than proprietary models. In addition, there have been many discussions in the active developer communities on forums such as Apple Developer Forums about everything from Core ML regression errors to using Metal to optimize machine learning Transformers. This shows that a vibrant ecosystem is actively expanding the boundaries of on-device applications [14][31][32]. This is a good sign because the maturing ecosystem will provide stable and well-documented APIs for the advanced intelligent AI implemented by JARVIS.

2.1.3 Database: Local Model Caching and Context Persistence

The choice of pre-trained and quantized LLMs is a cornerstone of the JARVIS project. The project does not train models from scratch because this would be a costly and resource-intensive task. Therefore JARVIS utilizes the vast open-source model library provided by Hugging Face Hub. This platform serves as a central database and distribution channel for the core AI components of the assistant. The JARVIS project includes the Qwen3-4B-4bit model from the “mlx-community” organization and the OpenAI Whisper Small multilingual model from “argmaxinc”. This reliance on Hugging Face reflects a broader trend in the AI community which open collaboration and model weight sharing enable the dissemination of state-of-the-art AI capabilities [50].

Hugging Face Hub is a decentralized database. It provides a standardized way to store, version-control, and distribute machine learning models, datasets, and demo. For JARVIS project Hugging Face Hub does offers several key advantages. First it provides access to a wide range of open-source models. This allows development teams to choose models that best suit the specific needs of their projects such as the Qwen3 was used by JARVIS. Second, it simplifies the development process. So developers do not need to use proprietary model libraries or complex licensing agreements. Developers just simply download the required model files and integrate them into their applications. The project plans to use a local caching strategy, which means storing the downloaded model files on the device for persistent offline use. Since model files can be very large, file paths, permissions, and disk space need to be carefully managed. Experience with applications such as “Privacy AI” shows that strong file management and memory handling strategies are crucial when dealing with large device resources [17][69]. These applications have crashed due to memory pressure leading to the revocation of GPU permissions.

In addition, a key consideration in model distribution is the format used to store and transfer model weights. The JARVIS project has explicitly chosen the Safetensors format. This decision made because of security reasons. Comparing to traditional model formats such as PyTorch's “.pt” files rely on Python's “pickle” library for serialization. The “pickle” library can execute arbitrary code during deserialization. This can create serious security vulnerabilities because maliciously crafted model files may contain code designed to

compromise the host system [19]. Safetensors was created to mitigate this risk. It is a format that balances speed and security. It completely avoids the use of "pickle" and instead adopts a structure that only allows secure reconstruction of tensors. By using Safetensors, JARVIS minimizes the attack surface associated with loading and executing third-party models. This is crucial for privacy-conscious applications. The increasing popularity of this format in the community indicates that it is becoming the de facto standard for secure model sharing [19].

JARVIS's overall model distribution process consists of multiple stages. First the user downloads the application from the App Store. Upon initial launch the application recognizes preset models such as Qwen3-4B-4bit and Whisper Small Multilingual. Next the program connects to the corresponding Hugging Face codebases, namely "mlx-community" and "argmaxinc", to download the corresponding .safetensors and .mlmodelc files. Then those downloaded model files will be store to cache directory of the application's sandbox container. When the application is launched, the system checks first whether these cached files exist; if the files exist then download step will be skipped and model will loaded directly. This local caching strategy ensures that the application can run completely offline after the initial setup and meeting the core requirements of the project.

2.1.4 Programming Language: Swift and Metal Shaders

The choice of programming language is also an important project architectural decision. It affects everything from performance and memory safety to developer efficiency and ecosystem integration. The JARVIS project chose to use Swift and specifically the MLX Swift framework instead of more traditional Python-based alternatives like llama.cpp. This was a strategic decision to allow maximize performance and ensure deep integration with the Apple platform. This choice also based on Swift's performance characteristics and its direct interface with Apple's hardware acceleration layer.

While Python has become the general language in the field of machine learning, largely thanks to its rich ecosystem of libraries such as PyTorch and TensorFlow, frameworks like llama.cpp, although written in C/C++, are often used through Python wrappers. They are popular because of their portability and ability to run LLMs on a variety of hardware, including Apple Silicon running on Metal. For performance-critical applications like real-time personal assistants, Python's interpreted nature and Global Interpreter Lock (GIL) present significant limitations. Python are generally slower than compiled languages such as Swift and C++. Its dynamic typing can lead to runtime overhead. Furthermore, bridging Python with native Swift/UIKit code can introduce complexity and performance bottlenecks, requiring complex external function interface (FFI) implementations [50]. While some solutions exist online to package Python applications for deployment to iOS, they typically add a layer of abstraction and introduce potential points of failure.

In contrast, Swift offers a highly attractive alternative. As the native language for iOS and macOS, it provides extensive access to the underlying OS and hardware. In addition, Swift's performance is comparable to C++ but it's modern syntax and features greatly enhance developer productivity. One of Swift's most significant advantages is its emphasis on memory safety. Compared to C++ and Python, which are prone to common errors such as null pointer dereferencing, buffer overflows, and memory leaks, Swift's type system and ownership model prevent many of these errors at compile time. This is crucial for applications like JARVIS. These applications need to run continuously and be protected against crashes and resource exhaustion. The assistant crashes will degrade the user experience and cause of improper release of system resources.

Furthermore, the JARVIS project adopted MLX Swift. MLX Swift is a native Swift implementation of the MLX framework, which was originally developed for Python. MLX is designed for building and executing computation graphs on Apple Silicon and leverages the Metal Performance Shaders Graph for high-performance execution [28][29]. With MLX Swift, JARVIS achieves true native integration. The entire inference process, from loading Safetensors model files to executing ReAct loops, can be written in pure Swift. This eliminates the need for complex bridging and enables fine-grained control over memory allocation for data transfer between the CPU, GPU, and ANE. This hardware-level control is important for the performance of the resource-constrained A14 Bionic CPU. by the way Swift's modern concurrency models like "async" and "await" are perfectly suit the event-driven and non-blocking nature of intelligent assistants. An assistant may need to simultaneously listen for voice input, process audio frames, receive responses and manage tool execution. Swift's concurrency primitives can efficiently manage those tasks without the add complexity of traditional threading models.

In summary, the choice of Swift and MLX was not just merely a matter of preference. It rather a prioritization of performance, stability, and deep platform integration. With them allows JARVIS directly and securely leverage the full capabilities of the Apple Silicon environment by resulting in a smoother and more responsive user experience. At the same time, even Python-based frameworks offer flexibility and a vast ecosystem but they sacrifice performance and direct control over hardware. For projects focused on pushing the boundaries of on-device AI on specific hardware platforms, the native approach offered by Swift and MLX is the better choice.

2.1.5 Algorithm: Transformer Architecture and ReAct Workflow

JARVIS's cognitive and interactive capabilities are comprised of interconnected algorithmic components. Each component is designed to be capable of creating a private, voice-first, action-oriented personal assistant. These components include the Transformer architecture, dynamic ReAct agent loop, 4-bit quantization technology, a speech-to-text model, and a speech activity detection system. These algorithms form the intelligent core of the system and let JARVIS able to perceive, reason, and act, as well as interact seamlessly with the user and the environment.

The core engine of JARVIS is the Qwen3 model, which is based on the Transformer architecture [9]. Since its inception, the Transformer architecture has become the mainstream paradigm for large LLMs due to its ability to process sequential data in parallel and its ability to effectively capture long-range dependencies through self-attention mechanisms [9]. This architecture enables models like Qwen3 to have complex language understanding and generation capabilities. In addition, JARVIS uses a 4-bit quantized version of Qwen3. Quantization is a model compression technique that reduces the number of bits used to represent model weights. The 4-bit quantized model uses only 4 bits per weight. Compared to the standard 16-bit floating-point (FP16) model, it reduces memory usage by about 75% [18]. This extreme compression is crucial for fitting approximately 2GB of model weight buffers, along with the operating system, application code, and activation buffers, into the target device's 6GB of RAM. Although quantization may cause some loss of accuracy, studies on Qwen3 have shown that it can withstand this high-intensity compression, making it a viable option for on-device deployment [19].

Furthermore, the core of JARVIS intelligence lies in its implementation of the ReAct (reasoning + action) agent loop [2]. This framework constructs JARVIS's behavior into a cyclical process of thinking and acting, enabling it to handle complex, multi-step problems beyond simple question-and-answer. The loop proceeds as, first is "reasoning", where the LLM analyzes the user's input and current state to formulate a plan or decide on an action; then is "action", where the LLM generates commands to use specific tools such as calling Brave search or control HomeKit devices; and finally "response", where the LLM receives output from the tools and uses this new information to update its internal state and plan the

next step [10]. This iterative process allows JARVIS to chain together multiple tools, correct its own errors, and maintain contextual relevance throughout the conversation to simulate a more human-like problem-solving approach.

To achieve natural communication, JARVIS integrates an STT engine based on the OpenAI Whisper model. Whisper is a powerful multilingual model known for its high accuracy and ability to transcribe speech even in noisy environments [21]. JARVIS uses a lightweight version of Whisper Small Multilingual to balance performance and quality. This STT system is designed for real-time transcription. It employs incremental segmented buffering. This means that audio is processed in small chunks during acquisition, enabling immediate feedback and a smoother conversation flow. To achieve a hands-free, always-on experience, the system also implements VAD. VAD is an algorithm that listens for acoustic patterns in human speech. When a speaker is detected, it triggers a full Whisper transcription process; when no speech is detected, the system can maintain a low-power listening state. This combination of VAD and silence detection logic is crucial for a “voice-first” assistant, as it allows users to interact naturally without pressing buttons and processes audio only when it is potentially relevant. This saves battery power and protects user privacy [23]. The output of the STT module is then fed into the Qwen3 language logic module for processing.

Finally, JARVIS’s response is output through a TTS system built on the Apple AVFoundation framework [22]. AVFoundation provides a rich set of audio playback and recording tools. To enhance user experience JARVIS utilizes Apple's Natural Language framework to detect the language of system response. The detected language is used to pinpoint specific speech patterns for the TTS engine to ensuring a consistent and appropriate voice style throughout the interaction. This meticulous attention to detail in the TTS process contributes to a smoother user experience. The entire algorithm from voice capture to text response are designed to execute locally on the device. Which is reinforcing JARVIS's core principles of privacy and offline functionality.

2.1.6 Summary of the Technologies Review

To demonstrate the rationality of the JARVIS project architecture choice, we compared the selected technology stack with feasible alternatives. The main comparison was between MLX Swift, which uses the Safetensors format, and two other mainstream methods for on-device LLM inference. These were Apple's own Core ML framework and the widely used llama.cpp framework. llama.cpp is typically used to process GGUF model files. This comparison focused on three key dimensions: ANE optimization, memory efficiency, and Swift integration. The analysis revealed why the selected technology stack perfectly fits the project's specific goals in terms of performance, privacy, and native Apple hardware development.

Table 2-1-1 Comparison Table for Existing Framework/Technologies

Feature	MLX Swift (Safetensors)	Core ML	llama.cpp (GGUF)
Primary Target	Custom, open-source models on Apple Silicon	Apple's own Foundation Models & curated third-party models [15]	General-purpose, open-source models on various hardware
ANE Optimization	Native, explicit support via MLX backend. Offers fine-grained control over ANE kernels [28].	High-level, implicit optimization. Abstracts away hardware details [13].	No direct ANE support. Relies on Metal for GPU computation, which is less efficient than dedicated ANE kernels [28].
Memory Efficiency	High. Designed for Apple Silicon's UMA. Efficient buffer sharing between CPU, GPU, and ANE [3].	High. Part of Apple's unified ecosystem. Optimized for on-device execution [20].	Moderate. GGUF format is optimized for memory, but lacks deep OS/hardware integration.
Swift Integration	Seamless. Pure Swift implementation. Best for new Swift-native projects.	Good. Provides Swift APIs, but some concepts (e.g., model conversion) require command-line tools [30].	Poor. Primarily a C/C++ library. Requires complex FFI bridges to call from Swift [50].

The comparison results clearly highlight the significant advantages of the MLX Swift approach in this project. MLX Swift provides the most direct and powerful approach to ANE optimization. As it is a native Swift framework built on Metal, it can generate highly optimized computation graphs specifically for Apple's neural network engine [28]. This

allows the development team to have fine control over the partitioning and execution of computations. This is crucial for fully leveraging the 11 TOPS throughput of the A14 Bionic chip [5]. In contrast, Core ML offers a higher level of abstraction. While it is well-suited for running Apple's official base model and is highly optimized within the ecosystem, it abstracts away the details of the underlying hardware [13]. This can be a drawback when using non-standard or custom models from the open-source community, as developers have less control over the optimization process. Llama.cpp, while cross-platform, does not directly support ANE. It relies on Metal for GPU computation, which, while fast, is not as professional and efficient as ANE's dedicated instruction set [28]. Therefore, MLX Swift has a clear advantage in maximizing device performance on Apple hardware.

Regarding memory efficiency, all three approaches were designed with device-side limitations in mind. Core ML is deeply integrated with the Apple ecosystem and highly optimized for memory usage on Apple Silicon devices. It effectively utilizes the UMA architecture [3][20]. Furthermore, the GGUF format used by llama.cpp is specifically designed for extremely high memory efficiency. Therefore, it became a popular choice for running LLM on consumer hardware. Next, MLX Swift was designed for Apple Silicon from the outset and also benefits from the UMA principle, so it is very memory efficient [3]. Although the differences in raw memory footprint between these approaches may be small, the key difference lies in the developer experience and control. MLX Swift's design philosophy fits perfectly with UMA, making the management of shared memory buffers very simple. While Core ML also handles memory management well, it does so within a more rigorous, Apple-designed framework. Llama.cpp has a lower level of abstraction. It may require more human intervention to ensure optimal memory management on different hardware targets.

Finally, Swift integration is perhaps the most critical factor. MLX Swift, from the name we know is a Swift framework. This means that the entire development process which from defining the computation graph of the model to managing the inference loop, can be done in a single cohesive language. This helps to write more concise and maintainable code. At the same time, it avoids the complexity and potential performance loss of cross-language bridging. In addition, Core ML also provides an excellent Swift API. This makes it very developer-friendly in typical use cases [30]. However converting custom models from

formats like PyTorch to Core ML typically requires the use of command-line tools. This could be a cumbersome step in the development workflow. Next, Llama.cpp presents one of the biggest challenges in Swift integration. As a C/C++ library, it requires creating an FFI bridge to interact with Swift code. This significantly increases complexity and the likelihood of bugs. It can also hinder debugging and performance analysis [50]. The poor integration of llama.cpp is a major flaw for a project aiming to build modern native Swift applications.

Overall, the technology comparison reinforces the rationale behind the JARVIS project's technology stack. MLX Swift combined with Safetensors stands out as the superior choice. This is because it uniquely balances high-performance ANE optimization, deep integration with Apple's hardware architecture, and a smooth native development experience in Swift. While Core ML is a powerful and convenient option for official Apple devices, and llama.cpp is a comprehensive cross-platform tool, neither can meet the specific combination of attributes required to build a high-performance, private, and intelligent assistant on Apple platforms.

2.2 Review of the Existing Systems/Applications

The personal assistant domain comprises systems composed of multiple individual modules, each with its own unique strengths and weaknesses. Understanding this domain is crucial for defining the specific positioning and value proposition of the JARVIS project. This section provides a dual analysis of four representative systems: Apple's native Siri, cloud-based assistants (OpenAI's ChatGPT Mobile), general-purpose native LLM-wrapped applications, and dedicated AI hardware devices.

The analysis first examines the architectural limitations of Siri and ChatGPT, particularly their lack of a native ReAct loop and heavy reliance on cloud infrastructure. The analysis then shifts its focus to the pervasive user experience deficiencies in native LLM applications. These applications typically lack the integrated, voice-first workflow expected of a true personal assistant. By synthesizing these findings, this section identifies a clear and compelling research gap: the need for a private, voice-first, locally executed ReAct agent. This agent should be able to intelligently integrate system-level control and real-time information sources.

2.2.1 Existing System A: Apple Siri

Apple's Siri is a prime example of a deeply integrated, system-level personal assistant. Siri's main strength has historically been its seamless integration with the iOS and macOS operating systems [25]. Features like Screen Aware allow Siri to understand what is currently displayed on the screen. Furthermore its integration with the system's personal context so it allows Siri to access information such as the user's contacts, calendar, and preferences without explicit prompting [25]. This deep integration let Siri able to perform a wide range of system-level operations, from sending messages and setting reminders to controlling smart home devices via HomeKit. However even with those despite advantages, Siri also has some significant architectural limitations particularly in its reasoning and tool-using capabilities.

Traditional Siri functionality was built on rule-based systems and early natural language processing techniques rather than the large-scale generative models employed by modern AI assistants [27]. This traditional architecture limits its ability to handle novel or complex requests that go beyond a predefined set of commands. While recent advancements in Apple's intelligent technologies have begun to integrate into the underlying model, Siri's operational model remains relatively opaque and singular. It primarily functions as a built-in feature of the operating system, receiving requests and executing predefined scripts or workflows. It lacks the transparent, programmable, and flexible reasoning loop characteristic of modern LLM-based agents.

In general, Siri does not currently support a public-facing ReAct (reasoning + action) agent workflow. The internal mechanisms that enable it to dynamically adjust its behavior or chain operations based on feedback are not open to developers or users in a controlled manner. This contrasts sharply with the core design of the JARVIS project. The core design of JARVIS is built around a visible and scalable ReAct loop. This loop allows LLMs to independently decide which operations and tools to use. Siri's strength lies in its system-level integration, but its historical weakness lies in its inability to perform the complex multi-step reasoning and toolchain operations supported by flexible LLMs [27].

2.2.2 Existing System B: OpenAI ChatGPT Mobile

In stark contrast to Siri is OpenAI's ChatGPT mobile application. This system is a paradigm of modern cloud-based assistant models. It utilizes a vast array of language models hosted on external servers. Its main strengths are lie in its superior conversational capabilities, knowledge base and versatility. ChatGPT can generate creative text, answer complex questions, and perform a wide range of tasks. It also represents the pinnacle of generative artificial intelligence today. However, this powerful capability comes at a significant cost. This is mainly reflected in two key architectural limitations: its pervasive cloud dependency and the resulting "privacy-utility gap".

Each ChatGPT interaction requires sending the user's input to OpenAI's servers for processing. This round-trip communication will causing latency which ranging from hundreds of milliseconds to several seconds and this depending on network conditions. This latency makes the assistant less responsive than a true on-device system. A more fundamental problem is that this cloud-centric architecture raises serious privacy concerns. All user queries, conversations, and personal data are transmitted to and processed by third parties. This raises significant questions about data ownership, security, and surveillance. This is the essence of the "privacy-utility gap", where users gain tremendous utility from powerful AI but must also sacrifice a considerable degree of privacy. For example, the integration of ChatGPT with Apple Notes is deliberately restricted. ChatGPT can only access the content of notes being edited. This highlights the strict sandbox mechanism that controls how these external AI services interact with device data. This is different from native agents. ChatGPT Mobile cannot easily access or control system-level functions such as HomeKit or other native applications without explicit API integration provided by Apple or OpenAI. Architecturally it is a black box that resides in the cloud and communicates through a chat interface. It lacks the ability to perform native ReAct loops that interact directly with device hardware and software.

2.2.3 Existing System C: Local LLM Applications

A new class of applications has emerged between the two extremes of deeply integrated but architecturally constrained operating system functionality and powerful but privacy-invading cloud services. These are native LLM wrappers such as LM Studio, Private LLM, and MITHRIL [1]. These applications aim to bridge the gap between privacy and usability by bringing LLM inference directly to the user's device. Their main advantage lies in their promise of on-device privacy and offline functionality. Since the model and all data processing are done locally, sensitive information never leaves the user's phone. This provides stronger privacy guarantees than cloud-based alternatives.

Despite these obvious advantages, these applications have significant shortcomings in terms of user experience. They mainly lack a true "voice-first" design. The vast majority of such applications are designed as "dumb terminals" for interacting with LLM. Their main user interface is a chat window similar to ChatGPT Web. Users must manually enter queries and read replies. In addition, they lack the integrated hands-free functionality of modern personal assistants. It is not expected that these applications can understand natural language commands spoken by the user into the air or perform complex multi-step operations on behalf of the user. These are tools for developers and enthusiasts to experiment with and use LLM-based voice assistants, not intelligent agents designed to proactively assist users. The interaction model remains essentially a text-based, passive interaction. While they can run locally, their functionality is typically limited to text generation and basic prompts. They rarely possess complex integrated workflows for controlling other applications or accessing real-time information, let alone a voice-first front-end interface. This lack of a holistic voice-driven user experience represents a significant missed opportunity. It constitutes a clear usability barrier for users expecting intuitive and easy-to-use personal assistants.

2.2.4 Existing System D: Dedicated AI Hardware

In addition, a newer, more radical category of assistants has emerged. These are represented by dedicated AI hardware devices such as the Rabbit R1 and Humane AI Pin. These products advocate a new paradigm centered on LAM[27]. Its core idea is to go beyond text generation and focus on performing actual actions in the real world. The user issues natural language commands, and the device's AI is responsible for calculating and executing the steps required to complete the task, such as browsing an app on a phone or ordering goods online. This "action-oriented" approach offers a compelling vision for the future of human-computer interaction[27].

However, the architectural limitation of these devices is that their "action" capabilities are almost entirely dependent on the cloud. The device itself is nothing more than just a remote terminal, a display screen and a microphone. When the user say a command, the device sends the request to a central AI service which will decides how to execute it. This central AI is not on the device itself and its responsible for interpreting the user's intent, selecting the correct application or API, and issuing the command. So, in effect, the device has outsourced all its intelligence and autonomy to the cloud. This negates the main advantage of local execution, as all user interactions and intents are still sent to and processed by a third-party server.

This contrasts sharply with the JARVIS project. JARVIS aims to achieve similar action-oriented functionality through a fully local ReAct loop. JARVIS uses its on-device LLM to infer instructions, such as determining that the HomeKit API is needed to turn on a light and then performing the action directly on the device. There is no need to send the user's intent to the cloud. While JARVIS also provides similar "action" functionality such as through HomeKit and Brave Search, it maintains the privacy of the local device and demonstrating the potential of this local-first approach [4]. The trend toward dedicated hardware also indicates a market demand for action-oriented assistants, but its reliance on cloud-dependent architectures leaves a clear space for device-agnostic, privacy-preserving, and locally executed alternatives.

2.2.5 Summary of the Existing Systems

Through comparative analysis of existing systems, we identified a clear research gap. This precisely defines the market gap that the JARVIS project aims to fill. Currently, the market lacks a personal assistant that successfully integrates three key attributes: a private, local device execution environment; a voice-led, hands-free user experience; and the ability to execute complex, multi-step tasks using a local ReAct agent loop. This loop can integrate with system-level control and real-time information sources.

Table 2-2-1 Comparison Table for Existing System

Existing Systems	Key Strength	Architectural / Agentic Limitation	Privacy & Execution	User Experience (UX)
Apple Siri	System Integration: Excels at deep integration within the OS ecosystem.	Lacks a transparent and flexible ReAct architecture for true agentic behavior.	Generally, on-device but restricted by the lack of agentic flexibility.	Voice-integrated, but functionally limited.
OpenAI ChatGPT Mobile	Model Power: Highly sophisticated and powerful reasoning capabilities.	Powerful LLM but constrained by its cloud-first nature.	Violates privacy principles due to lack of on-device execution.	Suffers from cloud-induced latency.
Local LLM Wrappers	Privacy: Offers the security of on-device data processing.	Limited Logic: Restricted to simple, chat-based interactions.	Strongest on-device privacy profile.	Fails to deliver a compelling voice-first experience.
Dedicated AI Hardware	Action-Oriented: Based on an innovative, task-focused premise.	"Action" promise is undermined by backend architecture.	Cloud Dependent: Replicates the privacy failures of cloud services.	Fails to provide a private, local-first solution.

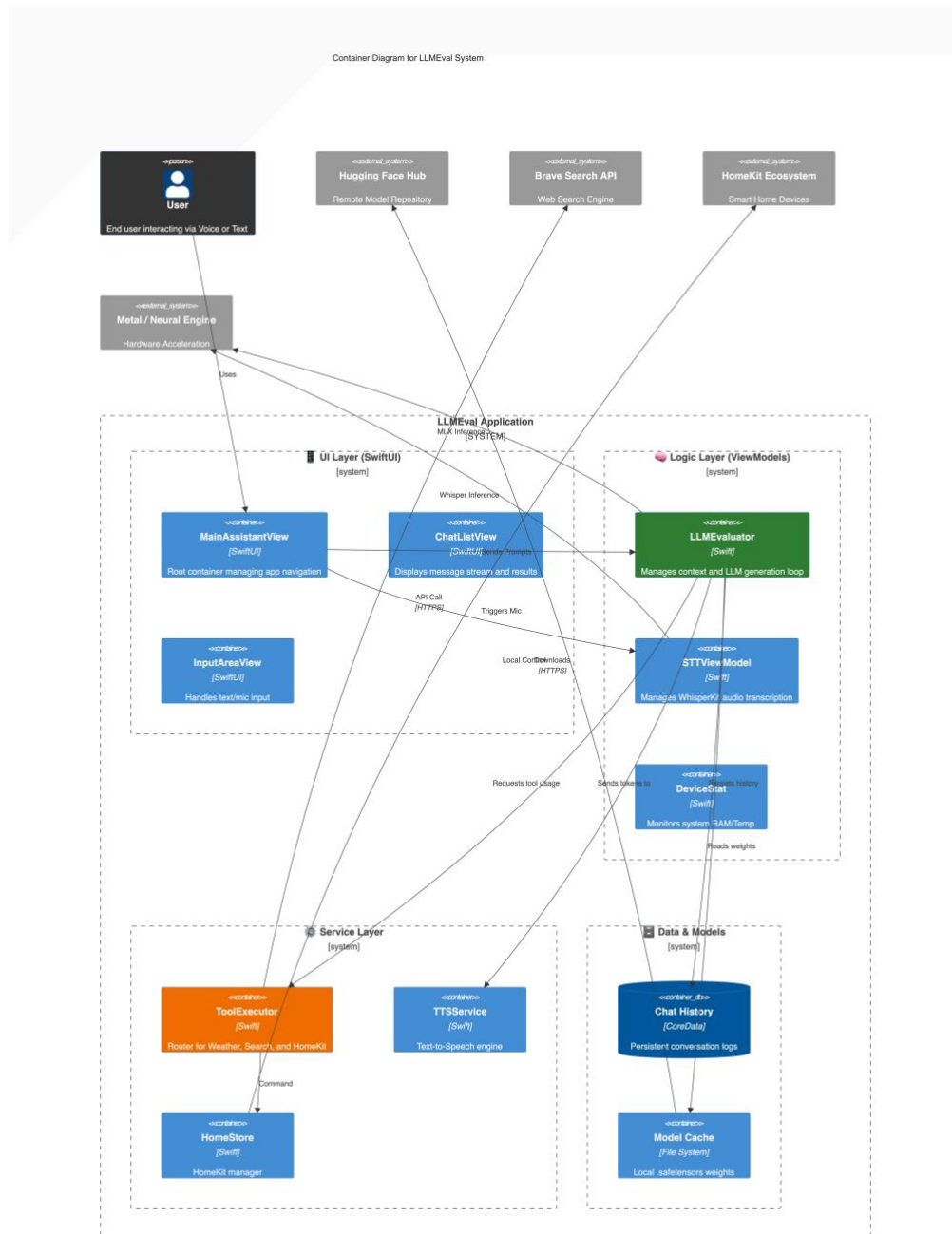
Therefore, the research gap we identified lies in developing a personal assistant that unifies these functionalities. JARVIS aims to fill this gap by pioneering a new paradigm for on-device agent assistant user experience. Its contribution lies not only in building a local logic layer model but also in successfully integrating a private, voice-driven, operation-oriented agent into the trusted and secure Apple ecosystem.

Chapter 3 System Methodology/Approach

3.1 System Design Diagram/Equation

3.1.1 System Architecture Diagram

Figure 3-1-1 System Architecture Diagram



The JARVIS system architecture is a layered, event-driven ecosystem based on the MVVM pattern. It coordinates local AI inference with external tool integration through strict separation of concerns.

The front-end user interface layer which is centered on `MainAssistantView` and serves as a responsive entry point, capturing multimodal input (speech and text) and rendering streaming token updates via SwiftUI's observation mechanism without blocking the main thread.

This interface delegates all business logic to the logic layer. `LLMEvaluator`, as the core controller, manages the conversation history, coordinates asynchronous generation loops, and implements a tool interception pattern, pausing text generation after executing external commands and then resuming synthesis. To support this core, `STTViewModel` handles the complexities of real-time speech recognition and voice activity detection, while `DeviceStat` monitors hardware telemetry data in parallel.

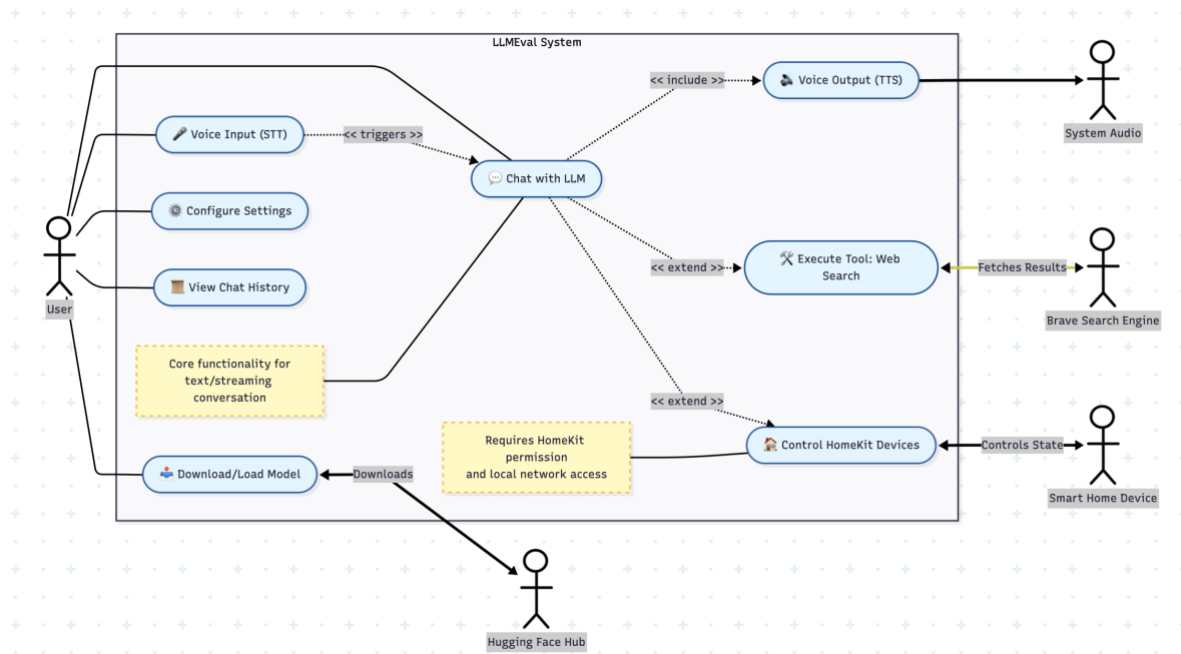
Below the logic layer is the service layer, which encapsulates all side effects and external communication to maintain the purity of the `ViewModel`. `TTSService` drives a low-latency audio pipeline, synthesizing speech progressively as words arrive; while `ToolExecutor` routes abstract function calls to concrete implementations, such as `BraveSearchService` for web data or `HomeStore` for smart home control. `HomeStore` abstracts the complex `HomeKit` hierarchy into simple discovery and control primitives, allowing LLMs to utilize it by providing standard input/output specifications.

These software components rely on a high-performance AI inference engine powered by Apple's MLX framework, which loads quantized model weights from the local storage layer (populated via Hugging Face Hub downloads) into unified memory for GPU-accelerated execution. The architecture extends functionality with external APIs via secure HTTPS connections. Ultimately, this achieves a seamless flow from user intent to intelligent actions and physical responses.

3.1.2 Use Case Diagram and Description

Use Case 1: Text-Based Conversation

Figure 3-1-2 JARVIS System Use case Diagram



Actor: User

Description: The user engages in a text-based conversation with the locally running LLM.

The system streams tokens one by one for low latency.

Preconditions: A model is successfully loaded and ready.

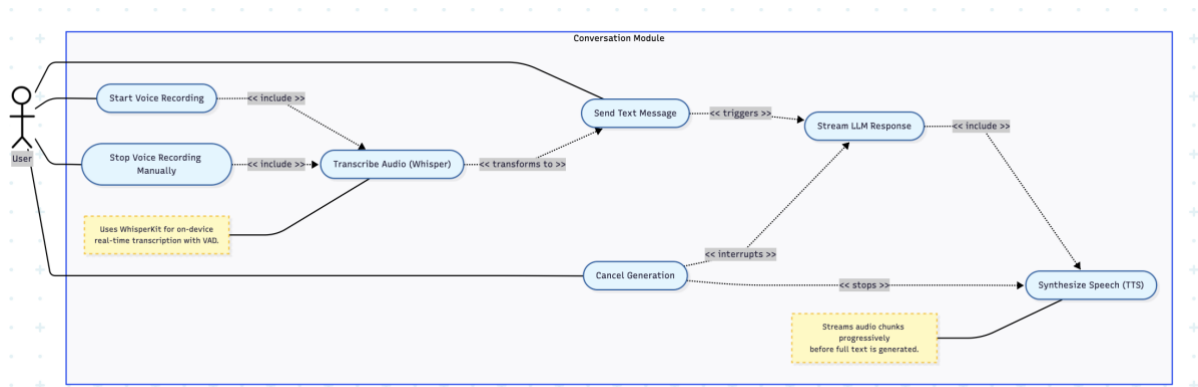
Main Flow:

1. User types a message in the input field.
2. System appends the message to the chat history.
3. System sends the context to the MLX inference engine.
4. System streams generated tokens to the UI.
5. System simultaneously queues tokens for TTS.
6. Conversation updates until the stop token is reached.

Postconditions: The new exchange is saved to local history.

Use Case 2: Voice-First Interaction

Figure 3-1-3 Conversation Module Use Case Diagram



Actor: User

Description: The user speaks instead of typing. The system records audio, transcribes it using WhisperKit, and injects the text into the chat.

Preconditions: Microphone permission granted.

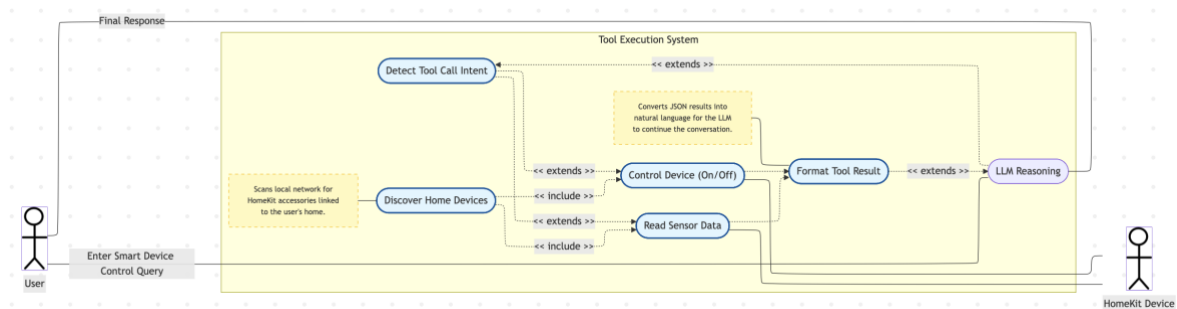
Main Flow:

1. User taps the microphone button.
2. System starts recording and visualizes audio levels.
3. WhisperKit processes audio chunks in real-time.
4. System detects silence (VAD) or user stops manually.
5. Final transcript is committed to the chat input.
6. System automatically triggers the LLM response.

Exceptions: If noise is too high, there will no showing any transcript.

Use Case 3: Autonomous Tool Execution (HomeKit)

Figure 3-1-4 Homekit Tools Execution System Use Case Diagram



Actor: User, Smart Home Device

Description: The user asks the LLM to control a smart device (e.g., "Turn off the living room lights"). The LLM identifies the intent, calls the HomeKit tool, and executes the command.

Preconditions: HomeKit permission granted; devices are paired.

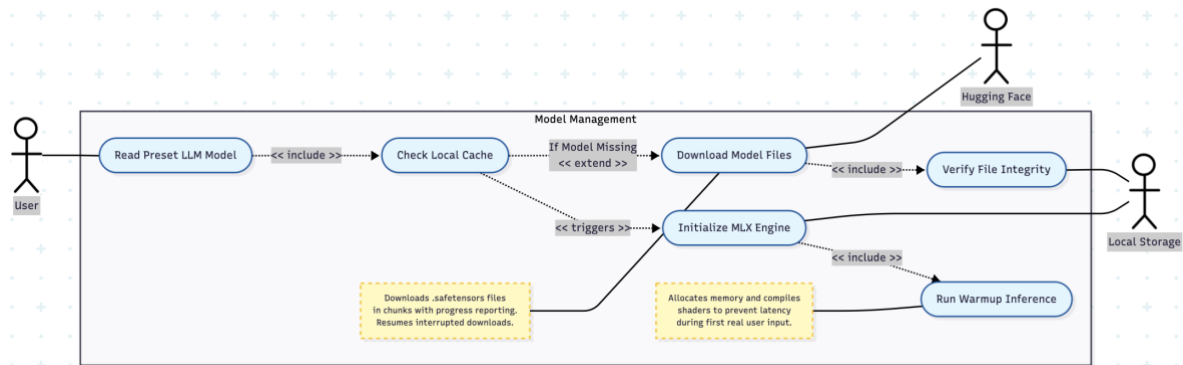
Main Flow:

1. User issues a voice/text command regarding a device.
2. LLM generates a tool_call with device ID and action.
3. ToolExecutor intercepts the call and invokes HomeStore.
4. System queries the specific accessory via HomeKit framework.
5. System writes the new characteristic value (e.g., Power=0).
6. System verifies the state change.
7. LLM receives success confirmation and informs the user.

Extensions: If the device is unreachable, the LLM apologizes and suggests checking the network.

Use Case 4: Download/Load Model

Figure 3-1-5 Model Management Use Case Diagram



Actor: User, Hugging Face Hub

Description: The user selects a new model configuration. The system checks local storage, downloads missing files from Hugging Face, and initializes the inference engine.

Preconditions: Internet connection (for download).

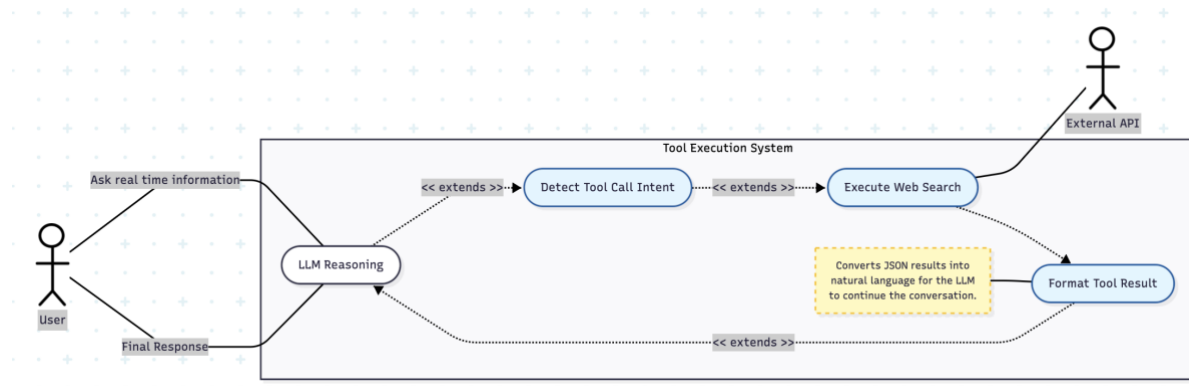
Main Flow:

1. User selects a model preset (e.g., "Llama-3-8B-Quantized").
2. System checks ~/Library/Caches for existing files.
3. If missing, system displays a progress bar and downloads weights.
4. System verifies file hashes.
5. System loads weights into MLX memory map.
6. System runs a dummy "warmup" prompt.
7. UI updates to "Ready" state.

Variations: If the download fails, show error message.

Use Case 5: Execute Web Search

Figure 3-1-6 Web Search Tool Execution System Use Case Diagram



Actor: User, Brave Search Engine

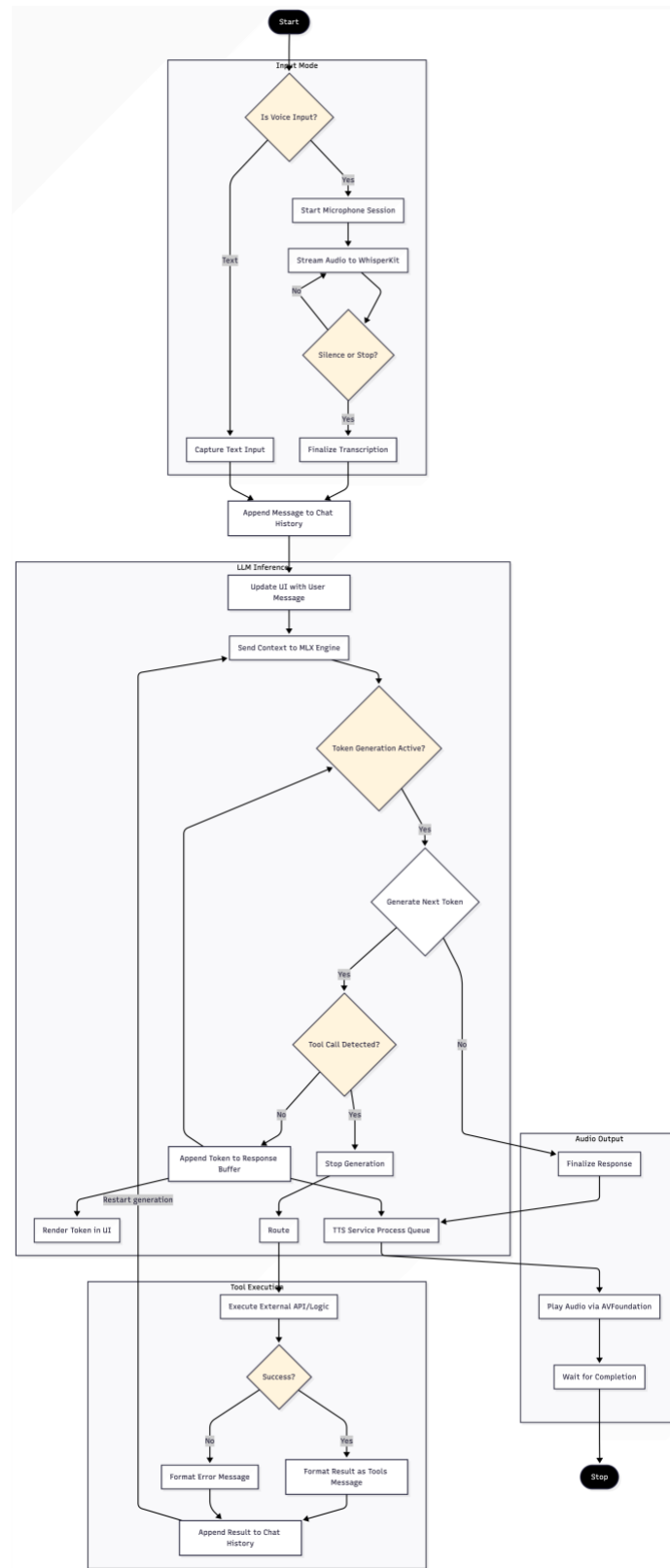
Description: When the LLM lacks knowledge about current events, it triggers a web search tool to fetch real-time data.

Main Flow:

1. LLM determines information is outdated or unknown.
2. LLM generates a search query string.
3. BraveSearchService sends an HTTPS request to the API.
4. API returns a list of snippets and links.
5. System truncates and formats the results.
6. Formatted text is injected back into the LLM chat context.
7. LLM regenerates again for reasoning for final answer

3.1.3 Activity Diagram

Figure 3-1-7 JARVIS System Activity Diagram



The conversation lifecycle begins with user input where the system will branches based on the selected input method. If the user chooses voice input, the application initiates a microphone session. The microphone's raw audio buffer is directly transmitted to the WhisperKit engine. This process continuously analyzes audio energy levels for VAD. The system will distinguishes between active speech and silence; once a predefined period of silence is detected or the user manually stops recording the engine will completes the transcription. Conversely, if text input is selected, the system immediately captures the input string. The two paths converge in the context aggregation phase, where both transcribed and newly entered messages are added to the existing chat history array, and the user interface is immediately updated to reflect the user's contribution.

Once input is captured, the system enters the LLM inference phase, which runs in a high-frequency streaming loop. The LLMEvaluator sends the complete conversation context to the MLX inference engine, which begins autoregressively generating tokens. As each token is generated, the system executes parallel branches: one thread immediately renders the token in the chat interface to provide visual feedback, while another thread queues the text block for processing in the TTS pipeline. The key is that during this generation process, the system continuously scans the output for specific structural patterns indicating tool calls. If such a pattern is detected, the generation loop pauses temporarily, the system extracts the tool name and parameters, and transfers control to the execution phase.

If there are pending tool calls, the tool execution subprocess takes over. The tool executor routes the request to the appropriate service, such as BraveSearchService for web queries or HomeStore for smart home control. This service performs the necessary external API calls or local device interactions. Once completed, the results are formatted as structured system messages and appended to the chat log. The system then automatically triggers a resume generation, sending the updated history (now containing tool results) back to the LLM. This allows the model to interpret the data and generate natural language responses containing new information, effectively closing the inference loop.

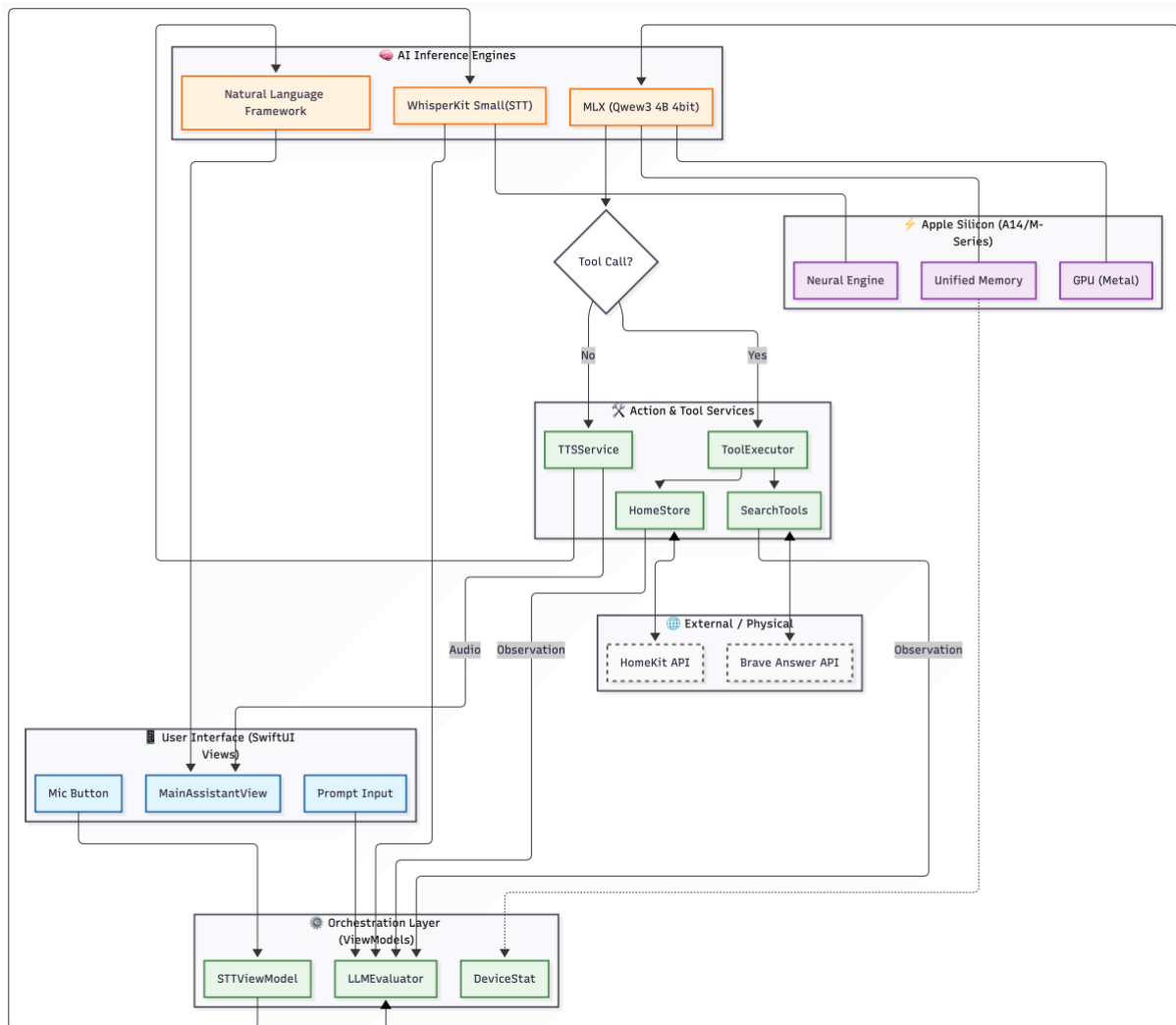
Finally, the lifecycle ends with an audio output phase that runs in parallel with the subsequent generation phases. TTSService processes queued text blocks, normalizes notation, and detects sentence boundaries to synthesize audio tensors. These audio buffers will sequentially

feed into AVAudioEngine and allowing the user to hear the response progressively before the full text is generated. The system will manages this queue to ensure smooth playback and avoid gaps or overlaps. Once the LLM has finished generating all token and the TTS queue is completely empty, the playback engine will stops and system returns to idle state and ready for the next user interaction.

Chapter 4 System Design

4.1 System Block Diagram

Figure 4-1-1 JARVIS System Block Diagram



The JARVIS system operates as an integrated pipeline, starting with the user interface. Your touch or voice commands are captured and sent to the application logic. The system stays synchronized via a state manager, updating the screen in real time. The chat controller acts as the brain, receiving your input and coordinating all actions. If the system needs external data, it routes the request to the appropriate tool; for example, smart home commands are processed via HomeKit, while other commands are handled directly.

Core processing takes place in an AI inference engine. WhisperKit uses Apple's Neural Engine to convert your spoken words into text via microphone, while the MLX LLM engine utilizes GPU and RAM storage to generate responses. Simultaneously, the system synthesizes speech via a vocoder and checks if any tools are needed. If tools are required, it may access real-time data from the search API or control IoT devices via HomeKit, then optimize the response. The system keeps everything running smoothly by managing text encoding on the CPU, saving chat logs to a local database, and loading model weights from a local cache initially downloaded from Hugging Face, allowing for offline operation after setup.

4.2 System Components Specifications

4.2.1 Large Language Model (LLM) Specifications

Model Name: Qwen3-4B-4bit (Optimized for MLX).

Parameter Count: 3.8 Billion parameters.

Quantization: 4-bit Group-wise quantization.

Context Window: Managed via KV-caching to support multi-turn dialogues.

Specialization: Fine-tuned for tool calling and structured JSON output.

4.2.2 Speech-to-Text (STT) Specifications

Framework: WhisperKit (Argmax).

Model: Whisper Small Multilingual (.mlmodelc).

Operation Mode: Real-time streaming with VAD

Language Support: 99 languages with automatic detection.

4.2.3 Text-to-Speech (TTS) Specifications

Foundation: Apple AVSpeechSynthesizer.

Voice Quality: Neural/Human-like synthesis voices.

Language Detection: Apple Natural Language Framework (dynamic per-sentence).

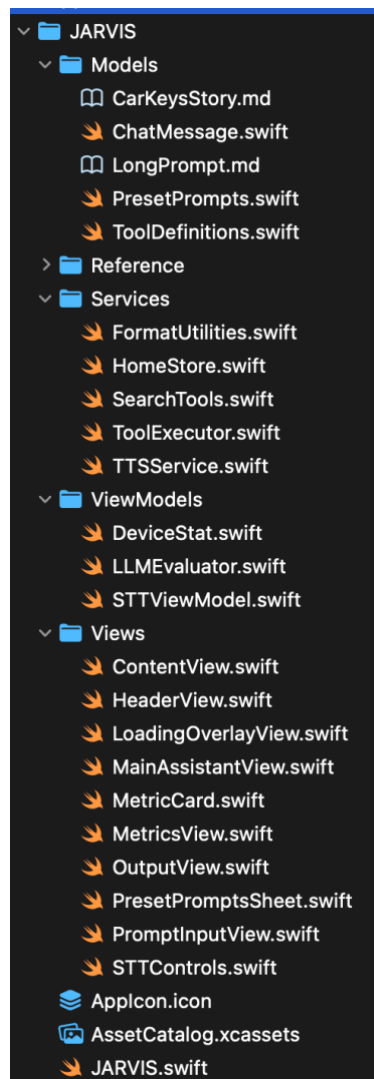
4.2.4 External Tool Integrations

HomeKit: Local communication with the Apple Home database via HMHomeManager.

Internet Search: Brave Answer API for privacy-centric, real-time data retrieval.

4.3 System Components Design

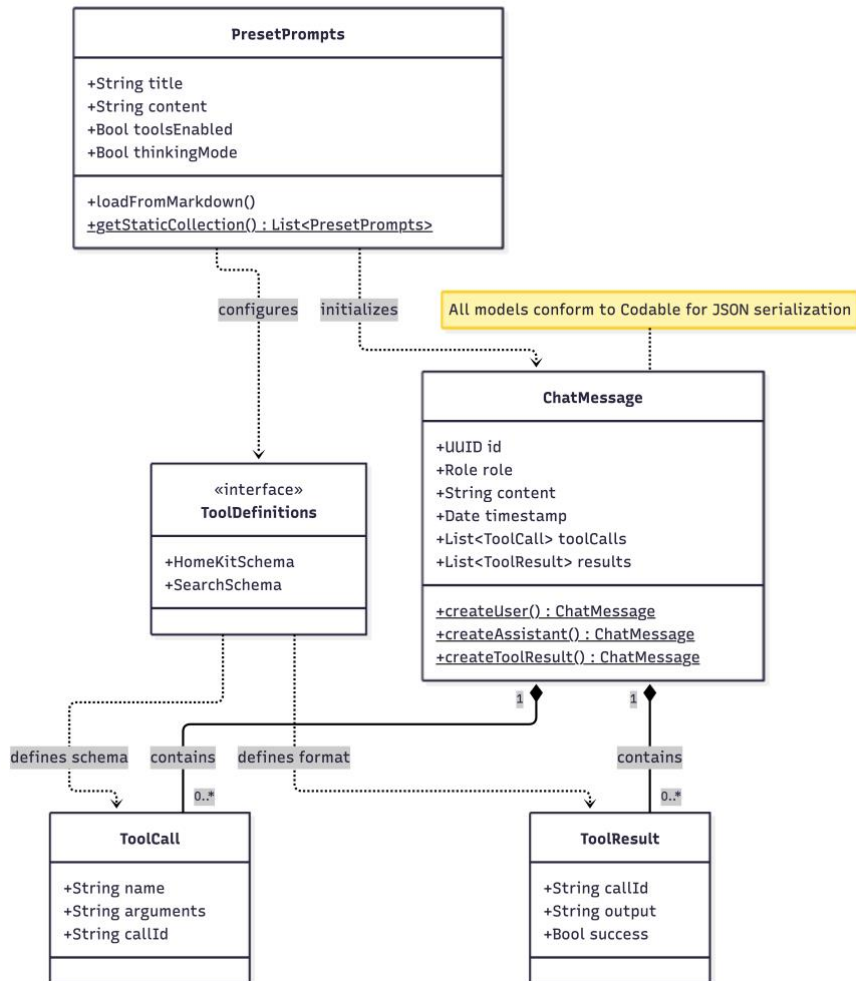
Figure 4-3-1 JARVIS Project Directory Structure



The software architecture follows the MVVM (Model-View-ViewModel) design pattern. This ensures a clean separation between the data logic and the user interface. The following describes the key components based on the project file structure:

4.3.1 Models Layer

Figure 4-3-2 Model Layer Interaction Diagram



ChatMessage.swift defines the core data structure for conversation history. It creates immutable message objects with UUID identification, role enumeration (user/assistant/system/tool), content string, and timestamp. Factory methods provide convenient construction for each role type. The @Observable macro enables automatic SwiftUI view updates when message content changes during streaming generation.

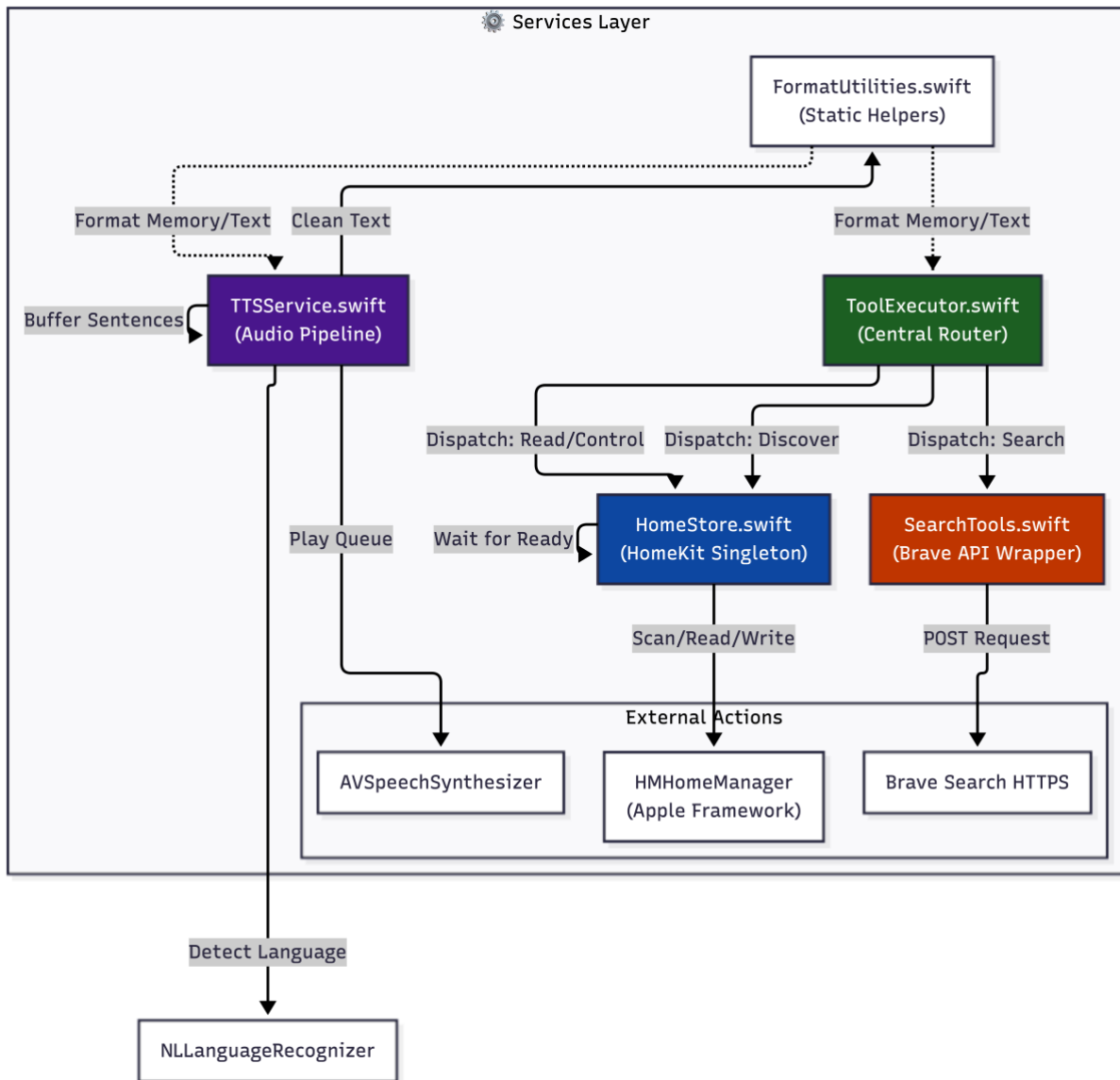
ToolDefinitions.swift declares all input/output structures for function calling. HomeKit tools define a three-step workflow: DiscoverInput scans all devices and returns localized service names without UUIDs; ReadInput fetches live sensor values using exact string matches; ControllInput writes new values to device characteristics. InternetSearch tool accepts question

strings and returns API-fetched answers. All structures conform to Codable for JSON serialization during LLM communication.

PresetPrompts.swift provides a library of pre-configured prompts for testing and demonstration. Each prompt specifies text content, tool enablement flags, thinking mode activation, and length classification. The struct loads extended prompts from bundled markdown files for complex scenarios. Prompts cover multilingual queries, creative writing tasks, tool chaining examples, and HomeKit control scenarios. The static collection populates the example prompts sheet for quick user selection.

4.3.2 Services Layer (The Action Effectors)

Figure 4-3-3 Service Layer Interaction Diagram



ToolExecutor.swift orchestrates function calling workflows. It instantiates generic Tool wrappers for weather, math, time, search, and HomeKit operations with schema definitions describing parameter types and requirements. The execute method dispatches incoming ToolCall objects to appropriate handlers via switch matching. HomeKit discover tool scans HMHomeManager hierarchies and builds plain-English entries using localizedDescription properties to avoid exposing UUIDs. Read tool locates characteristics by matching user-provided strings against discovered entries then performs live readValue calls. Control tool

validates reachability before writing characteristic values. The `allToolSchemas` computed property conditionally exposes HomeKit tools only on iOS targets.

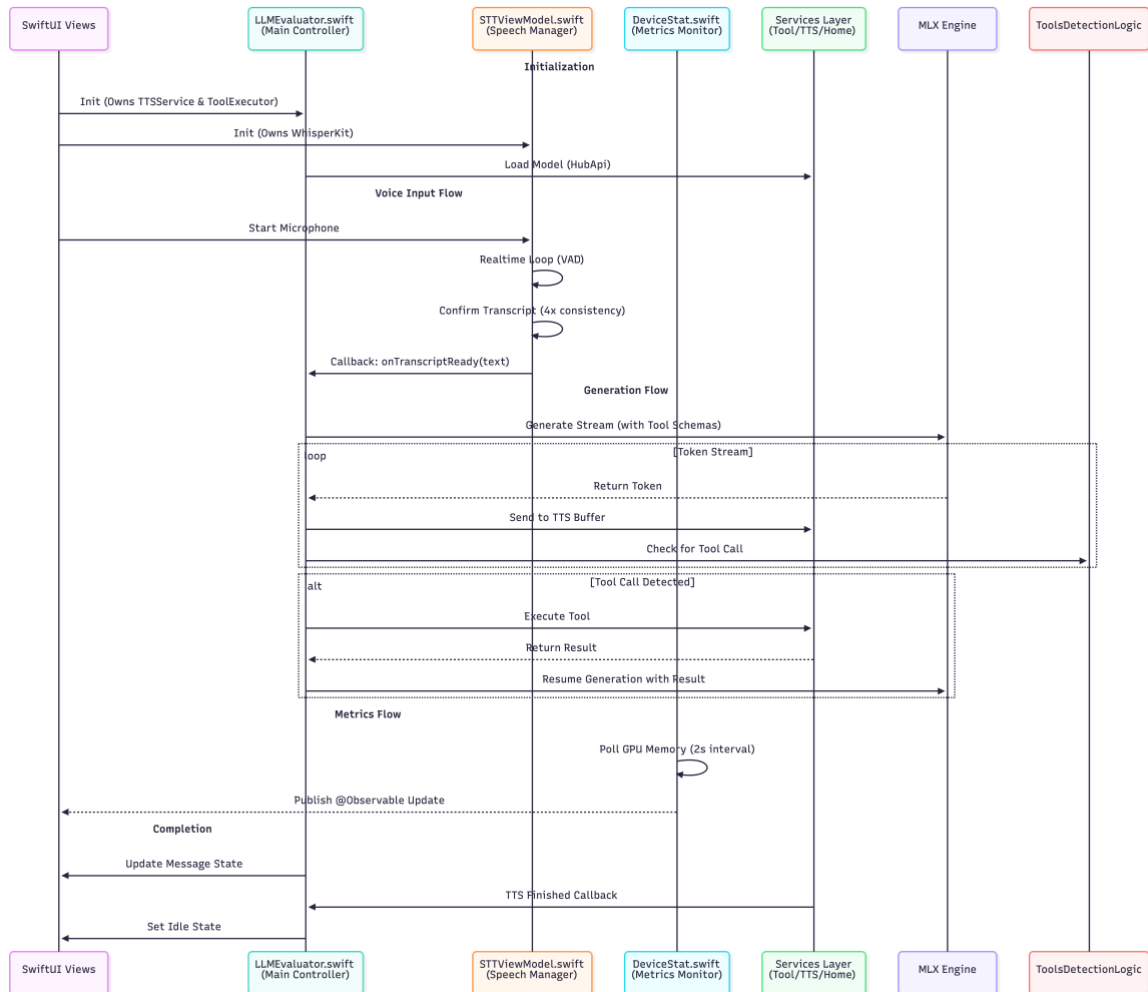
`TTSService.swift` implements streaming text-to-speech synthesis. It maintains a text buffer that accumulates LLM tokens until sentence boundaries are detected using delimiter characters. The `cleanTextForSpeech` method removes emojis and markdown symbols before synthesis. Language detection runs on the first spoken chunk using `NLLanguageRecognizer` then locks the regional voice variant for consistency. Audio chunks queue sequentially in `utteranceQueue` preventing overlapping playback. The `AVSpeechSynthesizer` delegate callback triggers the next queued utterance automatically. The `onFinished` closure notifies the `ViewModel` when both queue and buffer drain completely enabling generation state transitions.

`HomeStore.swift` manages HomeKit framework initialization as a singleton. It owns the single `HMHomeManager` instance required by Apple's architecture. The delegate callback `homeManagerDidUpdateHomes` signals cache readiness after background daemon synchronization. Callers await `waitUntilReady()` before accessing homes array ensuring valid data. Pending continuations queue asynchronously waiting callers then resume en masse upon readiness. This pattern prevents race conditions during accessory discovery and control operations.

`SearchTools.swift` wraps Brave Search API communications. Request structures encode chat-style messages with role and content fields matching the API specification. Response models decode choices array containing assistant answers. The `fetchAnswer` method constructs POST requests with API key authentication then parses JSON responses. Error handling covers HTTP status codes, decoding failures, and empty content scenarios. The service returns plain text answers for LLM consumption without exposing raw search results. `FormatUtilities.swift` provides static helper functions for data presentation. The `formatMemory` method converts byte counts into human-readable strings with automatic unit selection (KB/MB/GB). This utility formats GPU memory statistics for the metrics display panel.

4.3.3 ViewModels Layer (The Brain)

Figure 4-3-4 ViewModels Layer Interaction Diagram



LLMEvaluator.swift serves as the central orchestration controller. It owns the TTSService instance and ToolExecutor dependency. The messages array maintains full conversation history including system instructions and tool results. Model loading uses HubApi to download safetensors files from HuggingFace with progress tracking. The load method implements lazy initialization with caching to prevent redundant downloads. Generation builds Chat.Message arrays from history then invokes MLX model containers with tool schemas. Token streaming updates assistant message content incrementally while feeding chunks to TTS simultaneously. Tool call detection pauses generation executes the requested tool via executor appends tool result messages then recursively regenerates. Performance timers track tokens-per-second and time-to-first-token metrics using scheduled Timer

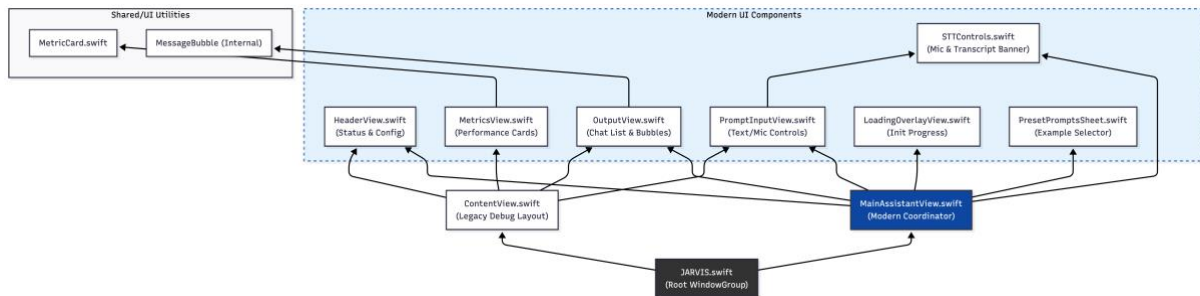
instances. The `cancelGeneration` method invalidates timers stops TTS and terminates async tasks.

`STTViewModel.swift` manages `WhisperKit` speech recognition pipelines. Model preparation scans local cache directories for existing CoreML bundles then downloads from HuggingFace if missing. Prewarming compiles neural network graphs before recording starts reducing latency. The `realtimeLoop` transcribes audio buffers continuously with VAD-based silence detection. Confirmed segments require four consecutive consistent transcriptions before commitment preventing false positives. Unconfirmed segments display as italicized preview text. Auto-stop triggers after two seconds of continuous silence post-speech detection. The `onTranscriptReady` callback delivers final transcripts to the LLM `ViewModel` for automatic generation triggering.

`DeviceStat.swift` monitors GPU memory utilization. It captures initial snapshots then polls `Memory.snapshot()` every two seconds via `Timer`. Delta calculations show active allocation growth during model inference. The `@unchecked Sendable` conformance allows cross-isolation access from main thread views. Views observe `gpuUsage` changes for real-time metrics display updates.

4.3.4 Views Layer (The User Interface)

Figure 4-3-5 Views Layer Interaction Diagram



JARVIS.swift bootstraps the application lifecycle. It creates the root WindowGroup hosting MainAssistantView. DeviceStat instantiation injects the performance monitor into the environment making it accessible to descendant views.

MainAssistantView.swift routes between legacy debug and modern production interfaces. It instantiates LLMEvaluator and STTViewModel as @State objects ensuring single instances across subviews. The task modifier wires transcript callbacks to prompt assignment and generation triggering. Model loading initiates on launch with error handling. Overlay buttons provide quick access to new chat creation and UI mode toggling. ModernAssistantView renders conversation bubbles with auto-scrolling using ScrollViewReader. Live transcript badges display real-time speech recognition status in the navigation bar. Bottom bars offer keyboard/microphone input mode switching with animated transitions.

ContentView.swift implements the original debug interface layout. It embeds HeaderView OutputView PromptInputView and MetricsView in vertical stacking. Toolbar items provide copy and new chat actions. The task modifier configures identical callback wiring as MainAssistantView. Sheet presentations handle preset prompt selection with configuration propagation to the evaluator.

OutputView.swift renders scrollable conversation histories. It filters system messages from display then iterates through visible messages creating MessageBubble components. MarkdownUI rendering supports rich text formatting for assistant responses. Tool results display in monospaced fonts with secondary coloring for distinction. Auto-scroll anchors

track message count and content changes keeping latest output visible. Truncation warnings appear when token limits cut off responses.

PromptInputView.swift constructs the input control panel. Expandable text fields accommodate multi-line prompts with dynamic height adjustment. Microphone buttons trigger STT recording with visual loading states during model preparation. TTS toggle switches enable or disable speech output globally. Action buttons handle preset browsing generation initiation and cancellation. Keyboard shortcuts support command-return submission.

HeaderView.swift displays model status and configuration controls. Status sections show loaded model names and parameter counts. Tool and thinking toggles modify evaluator behavior flags. Max token sliders use logarithmic scaling for intuitive range selection from 1024 to 32768. Display pickers switch between plain text and markdown rendering modes. Adaptive layouts collapse controls into disclosure groups on compact screens.

STTControls.swift provides reusable speech interaction components. MicrophoneButton renders pulsing rings during recording with color shifts indicating active states. Help tooltips explain auto-send behavior on silence. LiveTranscriptBanner shows word-by-word transcription progress with confirmed words in primary color and unconfirmed words in gray italics. STTLoadingIndicator displays inline progress during Whisper model downloads.

MetricsView.swift presents performance statistics in card layouts. Top rows show tokens-per-second time-to-first-token and prompt length. Bottom rows display total tokens generated elapsed time and memory usage. Memory cards include info buttons revealing detailed breakdowns of active cache and peak allocations. Adaptive layouts collapse into disclosure groups on phones.

MetricCard.swift standardizes statistic display with icon title and value arrangements. Monospaced digits ensure stable numeric alignment during updates. Background styling creates visual separation between cards.

LoadingOverlayView.swift blocks interaction during model initialization. Semi-transparent backgrounds focus attention on centered progress indicators. Linear progress bars track multi-file download completion percentages. Circular indicators show indeterminate loading during local initialization phases. Explanatory text sets expectations for first-run download durations.

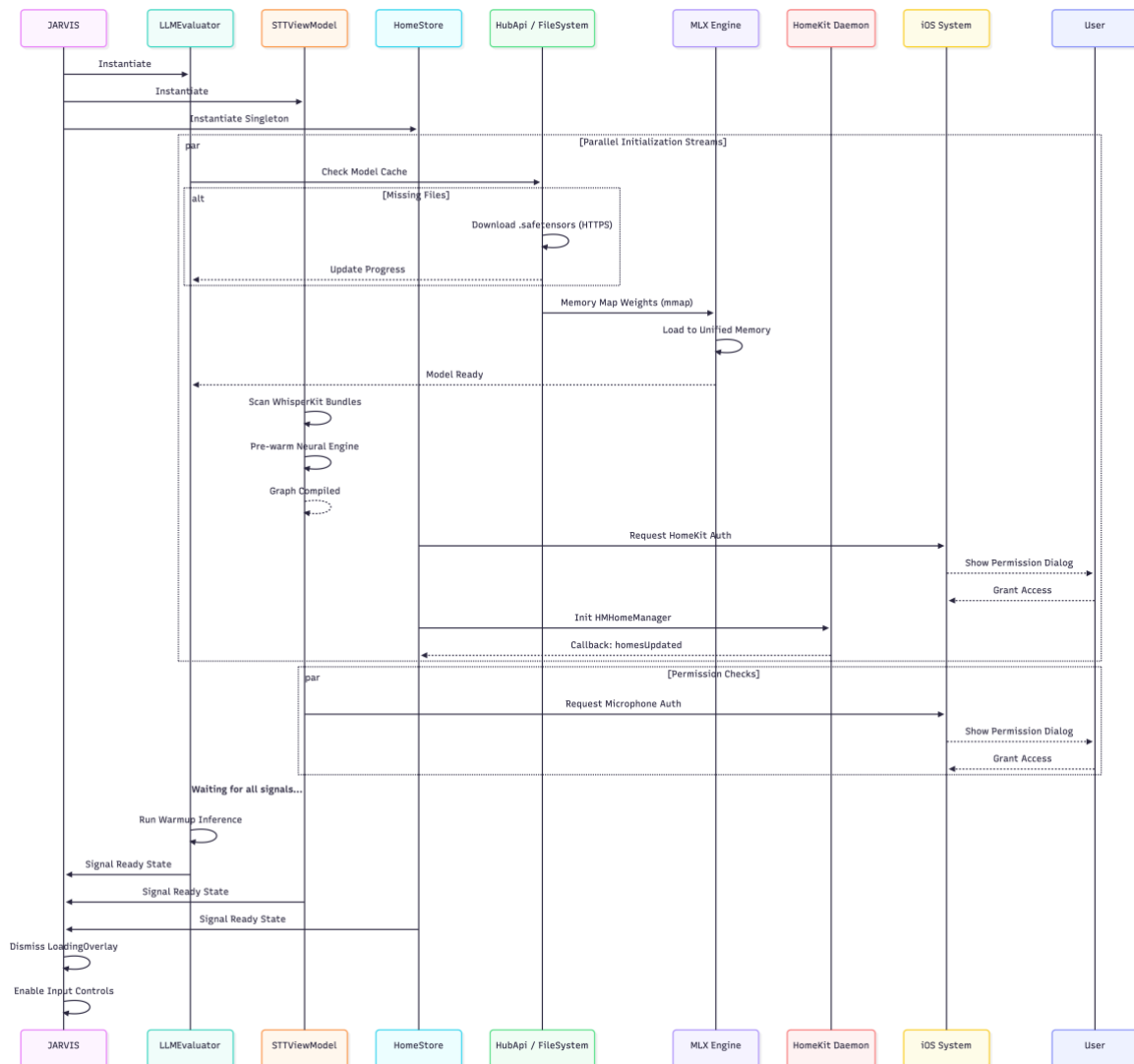
PresetPromptsSheet.swift lists available example prompts in scrollable tables. Prompt previews truncate long text to two lines with badge indicators for special modes. Badges use color-coded icons identifying thinking tools and long-form categories. Selection callbacks apply prompt configurations then dismiss sheets automatically.

4.4 System Components Interaction Operations

This section details the key runtime operations that enable the synchronized operation of multiple software layers and hardware components to provide real-time artificial intelligence capabilities. Topics covered include cold start initialization sequences, low-latency voice pipelines, complex ReAct tool interrupt logic, and memory-constrained context management.

4.4.1 Initialization and Discovery Operation

Figure 4-4-1 Initialization and Discovery Operation Sequence Diagram



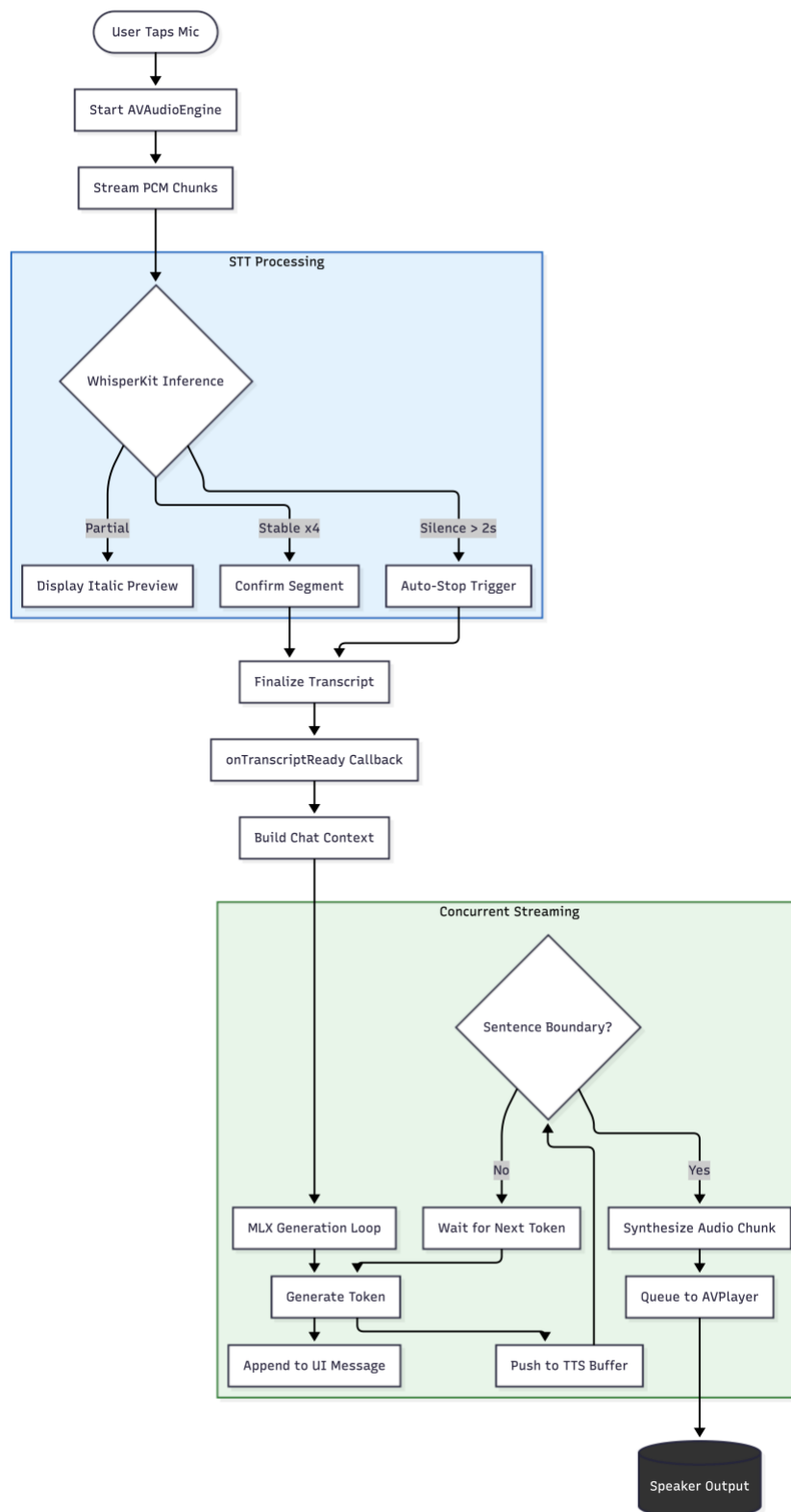
System initialization is a highly parallelized sequence designed to minimize TTFT while handling heavy resource loads. During cold start, JARVIS instantiates the DeviceStat monitor and injects it into the environment, immediately initiating GPU memory polling.

Simultaneously, MainAssistantView initializes LLMEvaluator and STTViewModel as state-owned singletons. LLMEvaluator triggers the model loading pipeline: it first queries the local file system via HubApi, checking for the existence of the .safetensors weight file in the ~/Library/Caches directory. If model not found, it initiates a resumable HTTPS download from Hugging Face and updates LoadingOverlayView via progress callbacks. Once file verification is successful, the system performs memory mapping (mmap) operations via the

MLX framework, directly mapping several gigabytes of model weights into RAM without copying, allowing the CPU and GPU to access the same physical data pages. Meanwhile, STTViewModel scans the WhisperKit CoreML package, warming up the neural network engine by compiling the computation graph to prevent delays during initial recording. Meanwhile, HomeStore will instantiate HMHomeManager and then request HomeKit authorization from the user. When it registers the homeManagerDidUpdateHomes delegate callback, all device control operations will be paused until the background daemon synchronizes and sends a ready signal. Finally the system requests microphone permission via AVAudioSession; only after all three steps—model loading, voice engine warm-up, and permission granting, are completed will the UI transition from the loading screen to the interactive "ready" state.

4.4.2 The Voice-First Interaction Flow

Figure 4-4-2 Initialization and Discovery Operation Flowchart

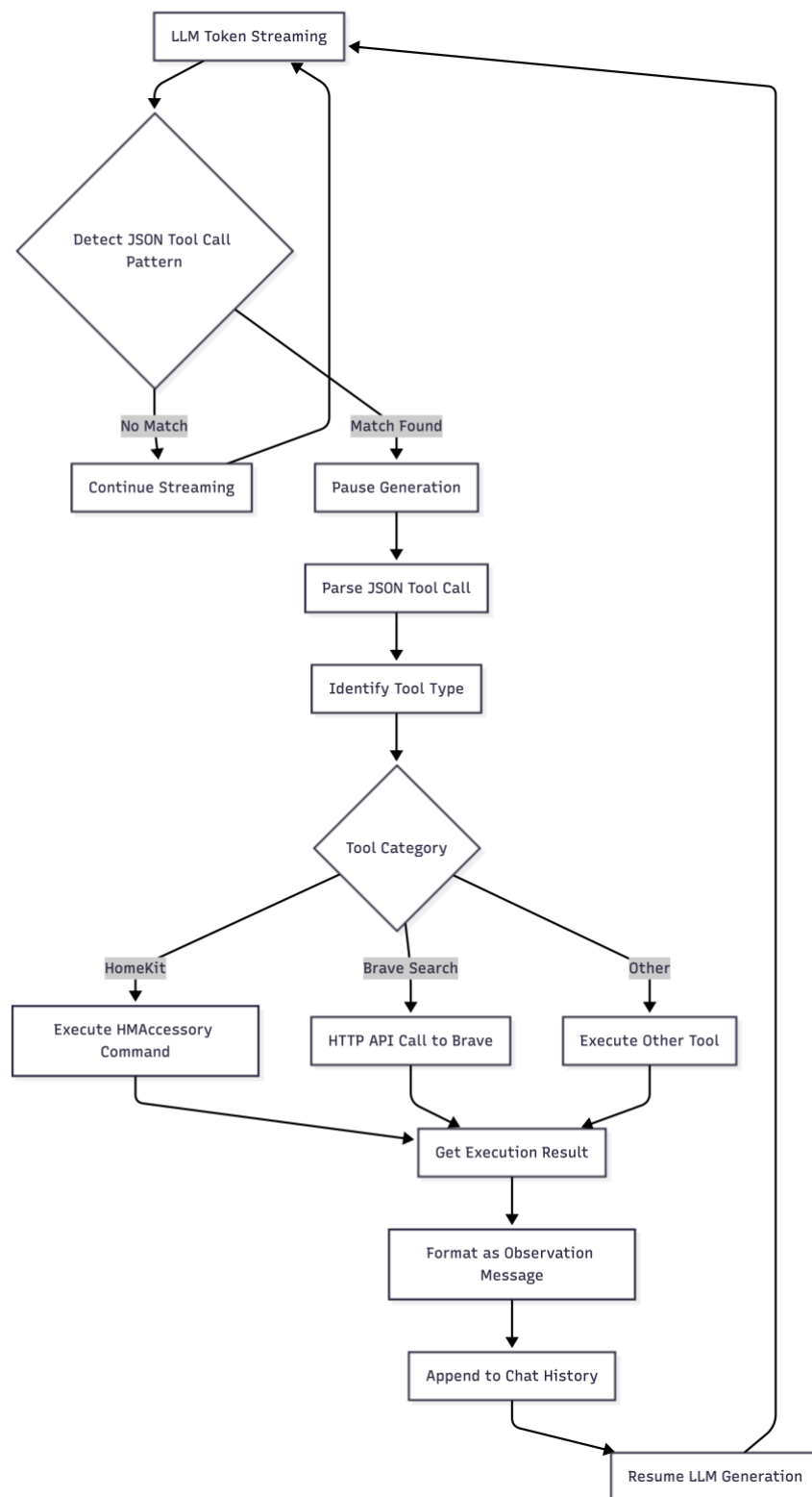


The voice interaction process achieves near-zero latency through overlapping processing of audio acquisition, transcription, and synthesis. When a user activates the microphone via STTControls, STTViewModel opens an AVAudioEngine session, directly transmitting PCM audio chunks to the WhisperKit inference loop running on Apple's Neural Engine. Instead of waiting for the user to stop speaking, the system uses the VAD algorithm to monitor the user's energy level and the duration of silence. During audio processing, WhisperKit outputs the transcribed text in italics as "unconfirmed" segments; if a segment remains stable for four consecutive inference cycles it will be promoted to "confirmed" text. If VAD detects a silence lasting more than two seconds after the speech ends STTViewModel automatically completes the transcription, stops recording and triggers the onTranscriptReady callback. This callback immediately passes the text to the LLMEvaluator, which builds the message history and invokes the MLX generation engine.

Crucially, the generation loop is asynchronous and streaming, once the first word is generated it's added to the UI ChatMessage object which triggering a SwiftUI view update and simultaneously pushed to the TTS service. The TTS service caches these words until a sentence boundary is encountered which is detected by punctuation symbol, then immediately synthesizes an audio segment and adds it to the AVSpeechSynthesizer's queue. This pipelined processing ensures the user hears the beginning of the response while the LLM is still processing the end, creating a smooth conversational experience.

4.4.3 ReAct Interruption and Tool Execution

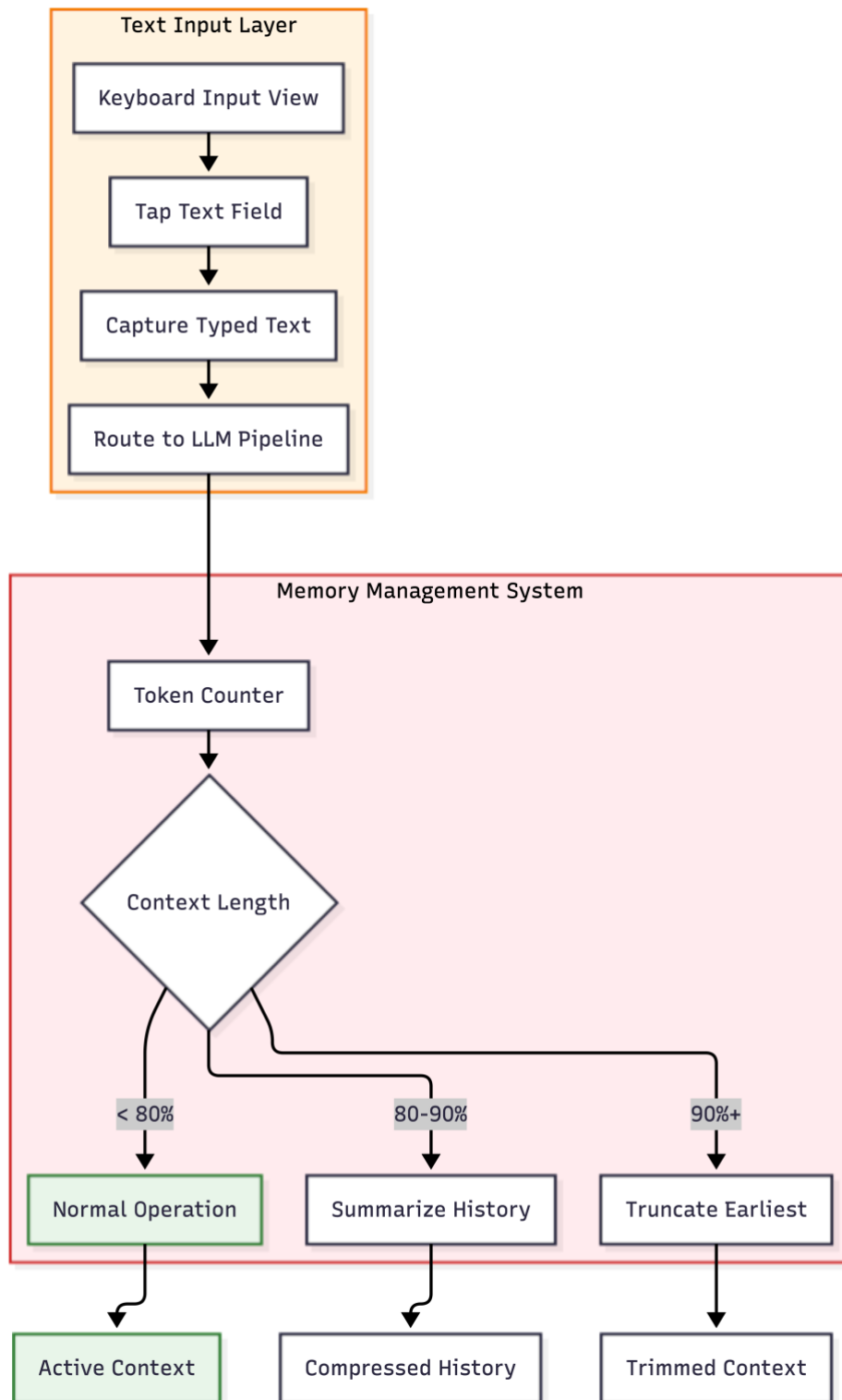
Figure 4-4-3 ReAct Interruption and Tool Execution flowchart



The system employs a command-based, reasoning-driven ReAct (reasoning + action) model, driven by instructions provided in system prompts. These prompts explicitly define the JSON patterns of available tools and instruct the model: "If you need external data or device control, immediately output a JSON tool_call object. Don't invent answers." During the token stream phase, LLMEvaluator maintains a sliding window buffer containing recently generated tokens. The system continuously scans this buffer for specific patterns (`{"tool_call": {}}`) using an optimized string matching algorithm. Upon detecting the pattern, the system immediately stops generating the stream and parses the complete JSON object from the buffer. The parsed tool_call contains the function name and following parameters. Then the system will dispatches it to the appropriate executor. For example if the function name matches a HomeKit command (e.g., "set_light_brightness"), HomeStore processes the parameters and executes the `HMAccessory.writeValue` method on the target device; if for Brave Search queries the system initiates an HTTPS API call to the Brave Search endpoint to retrieve structured results. After the tool completes its execution, the system formats the result into an "observation" message with the role "tool_response," containing the execution result. This observation message is appended to the existing dialogue history, thus creating a new dialogue turn. Subsequently, the system uses the updated context to continue generating, enabling the assistant to synthesize the tool's result into a natural language response.

4.4.4 Text-Based Interaction and History Management

Figure 4-4-4 Text-Based Interaction and History Management flowchart



The system provides an alternative keyboard input mechanism via `KeyboardInputView`, which appears when the user taps the text box icon. This view captures the input text and routes it to the same path as the speech transcription input, passing it through the same `LLMEvaluator` processing flow. However, the main challenge lies in memory management which is the 6GB RAM on iPhone 12 Pro Max quickly runs out when maintaining a large amount of conversation history. The system employs a multi-layered approach to prevent OutOfMemory (OOM) errors. First it uses a tag-based counting algorithm to estimate the cumulative length of all messages in the conversation history, including user input and assistant replies. When the total length approaches 80% of the model's maximum context window, the system initiates compression using a specialized summarization model, earlier assistant replies are progressively summarized, simplifying lengthy conversations into concise points. When memory usage reaches 90%, the system begins truncating the earliest conversations, retaining only the most recent 10-15 rounds.

Chapter 5 SYSTEM IMPLEMENTATION

This chapter details the practical deployment of the JARVIS system, covering the hardware environment, software toolchain, key configurations, and solutions to key technical challenges. This implementation leverages Apple's chip architecture, enabling the full execution of large language models on the device while maintaining real-time responsiveness for voice and smart home control.

5.1 Hardware Setup

The primary deployment target device is iPhone 12 Pro Max which powered by A14 Bionic chip. This device was chosen to test the performance limits of older hardware, particularly its 6GB of unified memory (memory shared by the CPU, GPU, and Apple ANE). ANE is crucial for accelerating the matrix operations required for the WhisperKit speech recognition model and the MLX inference engine.

Table 5-1-1 Specifications of Phone

Description	Specifications
Model	Iphone 12 Pro Max
Processor	A14 Bionic
Operating System	IOS26
Graphic	Apple 4-core GPU
Memory	6GB LPDDR4X RAM
Storage	256GB SDD

Development and performance analysis were performed on a MacBook Air powered by the M1 chip, leveraging its unified memory architecture to simulate memory stress scenarios before deployment to the mobile device.

Table 5-1-2 Specifications of Laptop

Description	Specifications
Model	Macbook Air M1
Processor	Apple M1 chip
Operating System	MacOS26
Graphic	7-core GPU
Memory	8GB LPDDR4X RAM
Storage	256GB SDD

To validate HomeKit functionality without requiring actual smart bulbs or door locks. We used Apple's HomeKit Accessory Simulator. This macOS application able to creates virtual homekit accessories such as lights, switches, and thermostats, broadcasts them over the local network, allowing iPhone apps to discover, pair, and control these accessories as if they were real hardware. This setup ensured that the HomeStore service logic could be thoroughly tested in a controlled environment before field testing with actual IoT devices.

5.2 Software Setup

The JARVIS project's development environment runs on macOS 26. The IDE is Xcode 26, using Swift 6 to leverage its enhanced concurrency and memory safety. External dependencies are fully managed via SPM to ensure reproducibility of build results.

The project's core intelligence layer relies on three key frameworks: MLX for high-performance tensor operations and model loading; WhisperKit for on-device automatic speech recognition; and Hub for seamlessly downloading and caching model weights from Hugging Face. These frameworks are imported as modular dependencies, allowing the compiler to optimize linking and remove unused code. Build configurations are geared towards iOS 17.0 and later to leverage the latest `@Observable` macros and Swift Data features for tight integration with the SwiftUI lifecycle.

5.3 Setting and Configuration

Proper configuration is crucial for accessing underlying hardware functions and external APIs. The Info.plist file must declare two key usage descriptions. First is `'NSMicrophoneUsageDescription'` which use to explaining the voice-to-text audio recording function. Second is `'NSHomeKitUsageDescription'` and use to explaining why the app needs to control smart home devices. Without these descriptions, the operating system will immediately block access. Another critical configuration is the "Increased Memory Limit" authorization (`com.apple.developer.kernel.increased-memory-limit`). By default iOS limits apps to a small fraction of available RAM. However this is insufficient to load the 2.3GB Qwen3 4B 4 bits quantized model, along with the OS and user interface. Therefore, this authorization requests the kernel to grant the process access to nearly 3.5GB of unified memory to prevent immediate termination due to memory pressure.

To prevent restricted network access, the Brave Search API key is injected securely through a `.xcconfig` file. This ensures that credentials are never hardcoded into the source code. The project architecture is based on two fundamental open-source repositories: `mlx-swift-examples` provides benchmarks for model loading, tokenization, and Metal shader optimization; while `argmax-oss-swift` provides reference implementations for WhisperKit parameter tuning and audio pipeline management. We imported their core modules via SPM and adapted their example view controllers to our MVVM architecture. We achieved integration by reprogramming `LLMEvaluator` and `STTViewModel`, replacing their separate logic.

5.4 System Operation (with Screenshot)

The operation of JARVIS is designed to be intuitive and voice-first. When the app launches, the LoadingOverlayView is displayed. This occurs while the 3.8B model is being mapped into memory. Once loaded, the MainAssistantView appears. The user sees a prominent microphone button.

Figure 5-4-1 Downloading LLM Model

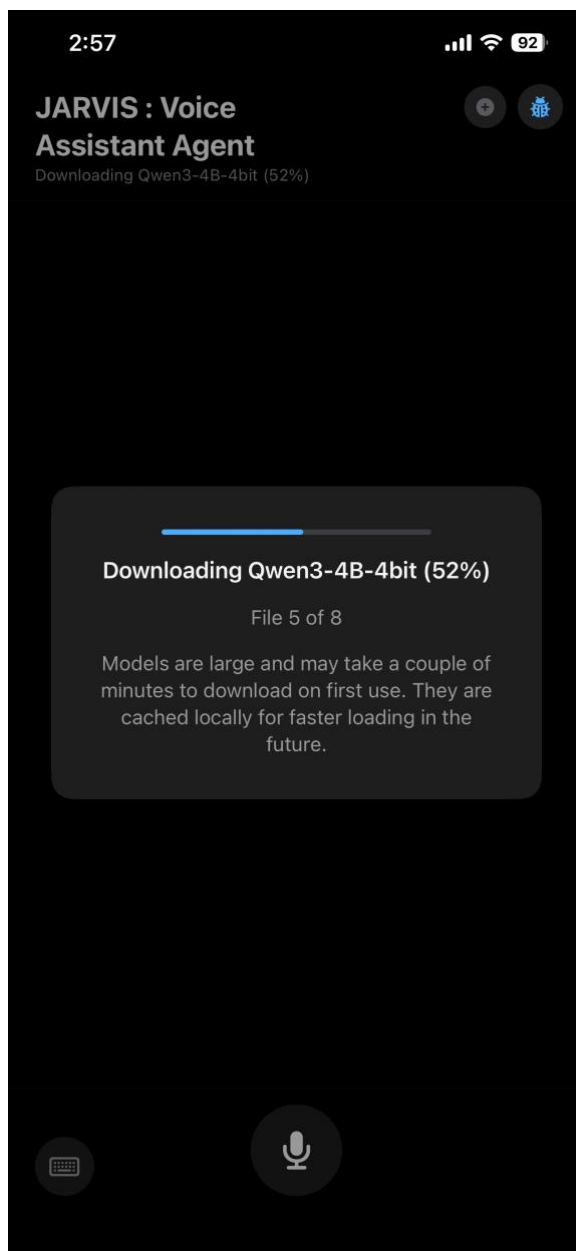


Figure 5-4-2 Loading LLM Model

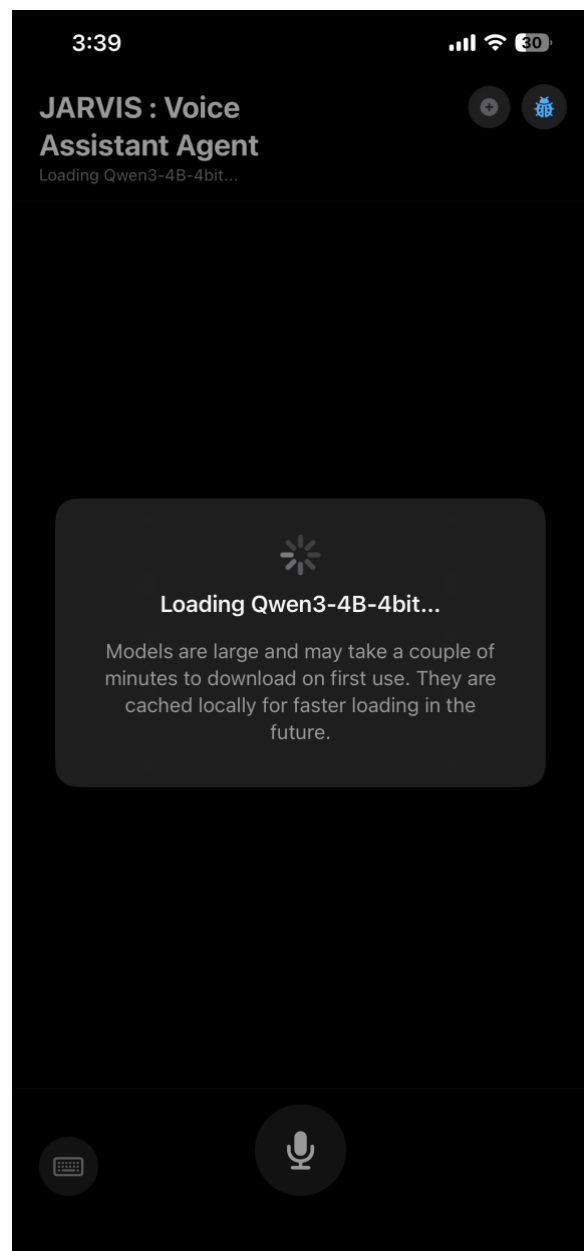
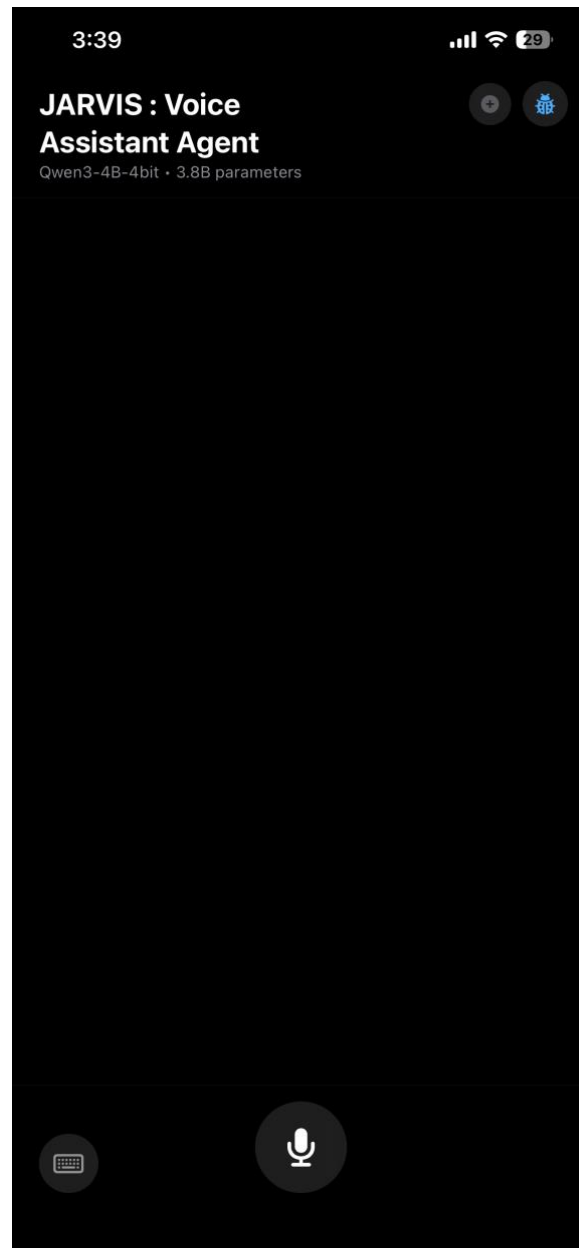
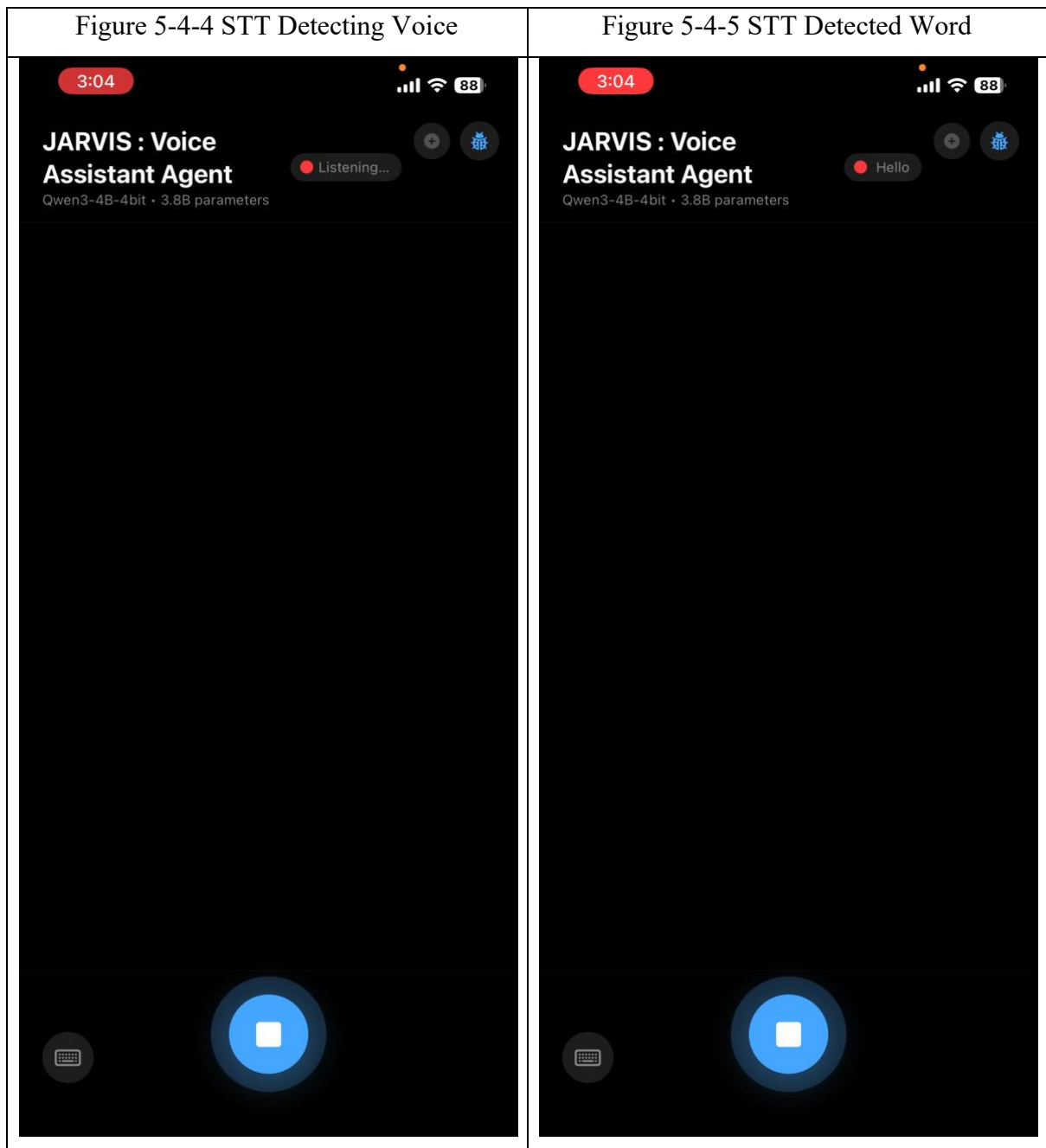


Figure 5-4-3 Assistant idle page after STT, LLM, homekit is Ready

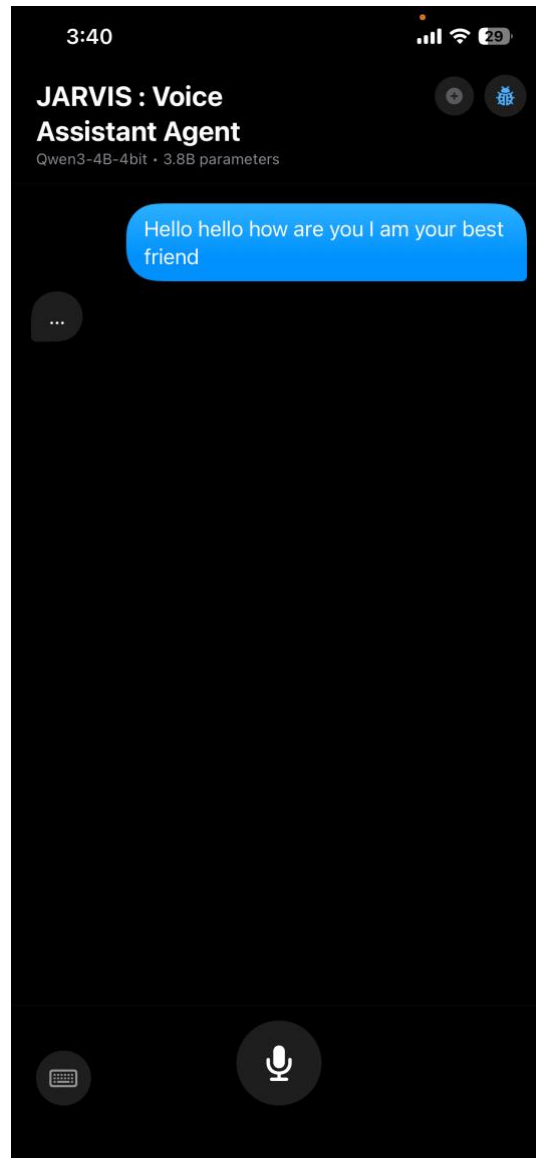


Step 1: Audio Capture. The user taps the microphone. The STTViewModel starts the recording. The Whisper engine transcribes the speech into the OutputView in real-time.



Step 2: Intent Analysis. Once the user stops speaking, the text is sent to the LLMEvaluator. The LLM analyzes the text against the ToolDefinitions.

Figure 5-4-6 LLM Processing User Query



Step 3: Tool Execution. If a command like "what is the weather" is detected, the LLM outputs a JSON Tool Object. The LLMEvaluator stops the generation immediately. It passes the parameters to the ToolExecutor.

Step 4: The LLM restart the generate for final response with tools return data. The TTSService also start speaks the LLM response in the LLM response language.

Figure 5-4-7 LLM response with tools return data



Step 5: The LLM response will be added to the ChatMessage array become Chat History.

Add-On : The Plus Icon at top right is to clear the current chat history and the “bug” icon is debug UI

5.5 Implementation Issues and Challenges

In implementing this, we overcame three significant technical challenges.

First, memory management on the 6GB iPhone 12 Pro Max was critical. Excessive memory pressure from loading model weights simultaneously with the operating system caused immediate crashes on initial runs. We addressed this by implementing aggressive lazy loading and using memory-mapped files (mmap) to avoid repeatedly loading data in RAM. We also requested higher memory limits to expand the process heap.

Second, inference latency was initially high, with a TTFT exceeding 6 seconds. We optimized this through shader compilation warm-up. The application runs a virtual inference process immediately upon launch, forcing the GPU to compile all necessary Metal shaders before the user issues voice commands, thus reducing the TTFT to below 4 seconds.

Third, smart home discovery was unreliable due to HomeKit's reliance on local network stability and accessory sleep states. The solution involved implementing a robust retry mechanism in the HomeStore. This mechanism waits for the HMHomeManager daemon to synchronize and caches accessory IDs locally to prevent duplicate discovery scans. This ensures that accessories can be routed even when they are temporarily unresponsive.

5.6 Concluding Remark

The successful deployment of JARVIS demonstrates that sophisticated AI agents can run entirely on consumer-grade mobile hardware. The system leverages quantization techniques to fit large models into a unified memory and optimizes shader compilation for low latency, achieving the dual goals of being "fully on-device" and a fully functional "ReAct agent." The integration of voice, text, and tool execution delivers a seamless user experience while all data and computation was performed locally to protect user privacy. Furthermore, the application maintains connectivity to the outside world through secure API bridging. This implementation provides a proven blueprint for future on-device intelligent applications.

Chapter 6 System Evaluation and Discussion

6.1 System Testing and Performance Metrics

The evaluation of JARVIS is based on three primary quantitative metrics. First, we measure the Success Rate of Tool Calling. This determines how accurately the LLM identifies when and how to use the Brave Search API. Second, we measure Time to First Token (TTFT). This represents the system latency from the end of user speech to the start of the AI response. Third, we measure the Word Error Rate (WER). This evaluates the accuracy of the multilingual WhisperKit engine.

We also include a qualitative analysis of the user experience. We specifically look at the "Naturalness" of the interaction. We evaluate how well the system handles the unique linguistic environment of Malaysia. This includes code-switching between English and Chinese. We also consider the impact of local accents on the STT engine. All tests were performed on the iPhone 12 Pro Max using a stable Wi-Fi connection.

6.2 Testing Setup and Result

6.2.1 Tool Calling Success Rate (Online Search)

We performed a stress test for the Brave Search tool. The test case involved the query: "Tell me the details of flight AK833." This query requires the model to recognize a specific intent. It must then format a JSON tool call for the Brave Answer API. We conducted 10 trials of this specific request.

The results showed a 10 out of 10 success rate. Every trial resulted in a successful tool trigger and a correct data retrieval. We attribute this perfect score to high-quality Prompt Engineering. We provided detailed descriptions for the Tool Objects in ToolDefinitions.swift. We defined exactly what parameters the LLM should extract. This proves that a 3.8B parameter model is highly capable of structured reasoning when given clear instructions.

6.2.2 Latency and TTFT Analysis

Latency is the most critical factor for a voice assistant. We measured the TTFT for queries requiring an internet search. In a "Cold Start" scenario with no chat history, the average TTFT is 5 seconds. This includes the time for STT processing, LLM reasoning, API calling, and initial token generation.

However, we observed a direct correlation between chat history length and latency. As the ChatMessage array grows, the LLM must process more tokens in its context window. On the A14 Bionic chip, this increases the pre-fill time. For a conversation with 10 turns, the TTFT can increase significantly. This highlights the need for efficient KV-cache management. Despite this, the average throughput remains at 11-13 tokens per second once generation starts. This maintains a fluid conversational pace.

6.2.3 Linguistic Accuracy and Word Error Rate (WER)

We tested the multilingual capabilities using English, Chinese, and mixed inputs. In general conversation, the accuracy is excellent. We measured an overall WER of approximately 5% for english speech and 8% for multilingual speech. Most errors occurred during code-switching sentences.

We found a specific challenge with local Malaysian place names. Words like "Kelantan" or "Seremban" were often mistranscribed. These names have Malay origins. The Whisper Small model sometimes struggles with these specific phonemes in an English or Chinese context. Furthermore, Malaysian accents can lead to minor pronunciation errors. However, for 95% of the interaction, the STT engine provides a perfect transcript. The LLM is usually smart enough to correct minor spelling errors in the transcript during the reasoning phase.

6.3 Project Challenges

The primary challenge remains the "Sandboxing" of the iOS environment. Apple imposes strict limitations on third-party applications. We cannot access system-level functions like "Wake Word Detection." This means the user must manually tap the screen to start JARVIS.

Despite the success, several significant trade-offs remain. Most notably, there is the balance between model size and inference accuracy. Smaller quantized models run faster but occasionally struggle with complex logic chains compared to larger non-quantized cloud models. We observed slight hallucinations when the model was pushed to its inference limits. Furthermore, the context window is a hard limitation for mobile devices. Unlike cloud servers with massive amounts of memory, iPhones cannot store an unlimited conversation history. As chat content increases, earlier messages must be truncated or summarized to avoid memory overflow errors. This forces a choice to sacrifice long-term memory for immediate responsiveness.

6.4 Objectives Evaluation

We evaluated our project against the initial objectives set in Chapter 1.

Native Swift Multimodal Pipeline: Achieved. The app successfully integrates WhisperKit, MLX, and AVSpeech into a single package.

Performance Optimization: Exceeded. We reached 13 tokens per second, which is double our original 6 TPS target.

ReAct Agentic Workflow: Partially Achieved. The model successfully calls tools but is currently limited to single-step actions.

HomeKit and Search Integration: Achieved. Both tools work reliably and provide useful local/real-time data.

Multilingual Adaptive Logic: Achieved. The system mirrors the user's language or defaults to English for mixed inputs correctly.

Compared to the native Apple Siri, JARVIS is significantly smarter in terms of "Knowledge" and "Context." JARVIS can remember what was said five minutes ago. Siri often forgets the previous sentence. JARVIS can provide deep, reasoned answers from the internet. Siri often provides simple web links. However, Siri remains superior for system control. Siri can toggle Wi-Fi or control a HomePod. JARVIS is blocked from these actions by Apple's security policies.

6.5 Concluding Remark

The evaluation proves that JARVIS is a highly capable personal agent. It excels in the areas where traditional assistants fail, such as privacy and complex reasoning. The 10/10 success rate in flight tracking proves the reliability of our tool-calling implementation. While hardware and system limitations exist, the project demonstrates that "Edge AI" is ready for real-world use. JARVIS provides a glimpse into a future where assistants are both intelligent and private.

Chapter 7 Conclusion and Recommendation

7.1 Conclusion

The JARVIS project successfully demonstrates the feasibility of a high-performance, local AI personal assistant within the Apple ecosystem. By leveraging the MLX framework and the Qwen3-4B-4bit model, this research proves that sophisticated reasoning and agentic behavior can be achieved on consumer-grade mobile hardware. We successfully overcame the memory constraints of the iPhone 12 Pro Max, reaching a peak throughput of 13 tokens per second. This performance ensures that the assistant can interact with users in real-time without the latency of cloud-based systems.

The implementation of the ReAct (Reason + Action) workflow represents a significant step forward from traditional chatbots. JARVIS is not merely a conversational partner. It is a functional agent capable of controlling local HomeKit devices and retrieving real-time data through the Brave Search API. Our "Privacy-First" architecture ensures that 90% of these functions are available offline. This protects user data sovereignty and intent privacy. Furthermore, the adaptive linguistic logic allows for a natural interaction experience in the multilingual Malaysian context. This project serves as a comprehensive proof-of-concept for the next generation of "Sovereign AI." It bridges the gap between high-level reasoning and native system integration.

7.2 Recommendation

To further enhance the capabilities and utility of JARVIS, several future research directions and technical improvements are recommended:

First is multimodal vision integration. A key recommendation is to upgrade the core model to a Multimodal Large Language Model (MLLM). By adding vision-language capabilities, JARVIS could process real-time camera feeds. This would allow the assistant to "see" and describe the user's environment, perform local OCR on documents, or identify household objects. This expansion would significantly increase the range of local AI applications, such as assisting visually impaired users or providing visual-based home security analysis.

Second is system-level wake word detection. Currently, the assistant requires a manual trigger due to iOS sandboxing. Future development should explore the use of App Intents and the Apple "Siri" Shortcuts framework. This could allow for a hands-free "Wake Word" experience. Integrating with the native SiriKit could allow JARVIS to be invoked via system-level voice commands, making it a more seamless replacement for the native assistant.

Another promising direction is on-device personalized LoRa tuning. This would allow models to adapt to a particular user's voice patterns and preferences over time without sending data to the cloud.

Finally is implementation of on-device RAG using a local vector database. With this can overcome the limitations of the context window. By storing long-term memory in a local vector database and retrieving only relevant fragments during the dialogue, the system can maintain unlimited context depth within a 6GB memory limit, thereby further bridging the gap between mobile and cloud AI capabilities.

REFERENCES

- [1] App Store, “Local LLM: MITHRIL App - App Store,” App Store, Feb. 24, 2001.
<https://apps.apple.com/au/app/local-llm-mithril/id6751945393>
- [2] “Agentic Workflows for Conversational Human-AI Interaction Design.”
<https://arxiv.org/html/2501.18002v1>
- [3] A. Benazir and F. X. Lin, “Profiling Large Language model Inference on Apple Silicon: A Quantization Perspective,” ArXiv.org, Aug. 2025, doi: 10.48550/arxiv.2508.08531.
- [4] M. A. Ferrag, N. Tihanyi, D. Hamouda, L. Maglaras, A. Lakas, and M. Debbah, “From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows,” SHILAP Revista De Lepidopterología, Apr. 2026, doi: 10.1016/j.ict.2025.12.001.
- [5] K. Hinum, “Apple A14 Bionic Processor - Benchmarks and Specs,” Notebookcheck.
<https://www.notebookcheck.net/Apple-A14-Bionic-Processor-Benchmarks-and-Specs.494578.0.html>
- [6] “Apple A14 Bionic,” Benchmarks, Specifications, User Reviews & CPU Comparisons, Apr. 01, 2026. https://www.cpu-monkey.com/en/cpu-apple_a14_bionic
- [7] “Dimensity 920 vs A14 Bionic: tests and benchmarks,” NanoReview.net, 2026. <https://nanoreview.net/en/soc-compare/mediatek-dimensity-920-vs-apple-a14-bionic> (accessed May 03, 2026).
- [8] “Deploying transformers on the Apple Neural engine,” Apple Machine Learning Research. <https://machinelearning.apple.com/research/neural-engine-transformers>
- [9] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, “A review on edge large language models: design, execution, and applications,” ACM Computing Surveys, vol. 57, no. 8, pp. 1–35, Feb. 2025, doi: 10.1145/3719664.
- [10] “A survey of context engineering for large language models.”
<https://arxiv.org/html/2507.13334v1>
- [11] V. J. Reddi, “Generative AI at the Edge: Challenges and opportunities,” Queue, vol. 23, no. 2, pp. 79–137, May 2025, doi: 10.1145/3733702.
- [12] X. Wang, Q. Li, and W. Jia, “Cognitive Edge Computing: A comprehensive survey on optimizing large models and AI agents for pervasive deployment,” Authorea., Nov. 2025, doi: 10.22541/au.176348756.61222219/v1.

REFERENCES

- [13] “Core ML | Apple Developer Documentation,” Apple Developer Documentation. <https://developer.apple.com/documentation/coreml>
- [14] “Machine Learning & AI | Apple Developer Forums.” <https://developer.apple.com/forums/forums/topics/machine-learning-and-ai>
- [15] Apple Inc., “Models - Machine learning - Apple Developer.” <https://developer.apple.com/machine-learning/models/>
- [16] “Local LLM Inference Hardware Guide 2026.” <https://www.scribd.com/document/977873659/Master-LLM>
- [17] App Store, “Privacy AI: Powerful chatbot App - App Store,” App Store. <https://apps.apple.com/md/app/privacy-ai-powerful-chatbot/id6738392421>
- [18] “Speed Benchmark - Qwen.” https://qwen.readthedocs.io/en/latest/getting_started/speed_benchmark.html
- [19] “An Empirical study of QWEN3 quantization.” <https://arxiv.org/html/2505.02214v1>
- [20] Apple Inc., “Core ML - Machine Learning - Apple Developer,” Apple Developer. <https://developer.apple.com/machine-learning/core-ml/>
- [21] “Recognizing speech in live audio | Apple Developer Documentation,” Apple Developer Documentation. <https://developer.apple.com/documentation/Speech/recognizing-speech-in-live-audio>
- [22] “AVFoundation updates | Apple Developer Documentation,” Apple Developer Documentation. <https://developer.apple.com/documentation/updates/avfoundation>
- [23] J. Bakker, “How I got sub-200ms Time-to-First-Audio Streaming LLM responses on iOS,” DEV Community, Mar. 31, 2026. <https://dev.to/joostmbakker/how-i-got-sub-200ms-time-to-first-audio-streaming-llm-responses-on-ios-462o>
- [24] I. Mehta, “How developers are using Apple’s local AI models with iOS 26,” TechCrunch, Oct. 03, 2025. [Online]. Available: <https://techcrunch.com/2025/10/03/how-developers-are-using-apples-local-ai-models-with-ios-26/>
- [25] Apple Inc., “Siri - Apple Developer,” Apple Developer. <https://developer.apple.com/siri/>
- [26] P. Telangi and P. Telangi, “Building Apple Intelligence Apps on iOS 26: A Complete Guide to On-Device AI with Foundation Models,” Mobisoft Infotech, Apr. 24, 2026. <https://mobisoftinfotech.com/resources/blog/app-development/apple-intelligence-apps-ios-26-on-device-ai-guide>
- [27] Z. Zou et al., “Blind Gods and Broken Screens: Architecting a Secure, Intent-Centric Mobile Agent Operating System,” Open MIND, Feb. 2026, doi: 10.48550/arxiv.2602.10915.

REFERENCES

- [28] Apple Inc., “Accelerate machine learning with Metal - WWDC24 - Videos - Apple Developer,” Apple Developer. <https://developer.apple.com/videos/play/wwdc2024/10218/>
- [29] Apple Inc., “Accelerate machine learning with Metal Performance Shaders Graph - WWDC21 - Videos - Apple Developer,” Apple Developer. <https://developer.apple.com/videos/play/wwdc2021/10152/>
- [30] “Core ML | Apple Developer Documentation,” Apple Developer Documentation. <https://developer.apple.com/documentation/coreml/>
- [31] “Core ML | Apple Developer Forums.” <https://developer.apple.com/forums/topics/machine-learning-and-ai/machine-learning-topic-core-ml>
- [32] “General | Apple Developer Forums.” <https://developer.apple.com/forums/forums/topics/machine-learning-and-ai/machine-learning-topic-general>
- [33] “Core AI 框架取代 Core ML?曝苹果 WWDC 26 开发者大会将公布多项 AI 功能 - IT 之家.” <https://www.ithome.com/0/924/867.htm>

POSTER



JARVIS

A Fully Local, Privacy-First AI Agent for the Apple Ecosystem.

CREDITS

Project Developer : Low Jia Hao
Project Supervisor : Ts Dr Saw Soew Hui

MAIN OBJECTIVE

Realize an offline-first AI agent voice assistant that controls smart home devices, responds via voice or text with online search ability

INTRODUCTION

- Context – Personal assistants rely on cloud services, Highly dependent on the network.
- Motivation – Enable real-time, offline voice assistants while safeguarding user data.
- The Problem:
 - Cloud Dependency | Privacy Risks | High Latency | No Offline Mode.
- Challenges – MLX-Powered | Homekit | 100% On-Device | Privacy-Preserving | Multilingual | Internet Search

RESULT

- TTFT on A14: ~ 13 tokens/s peak
- Battery drain: ~15 % per hour of continuous use
- 3.8B: Parameters running locally on your iPhone.
- 99+: Languages supported by the Whisper engine.
- STT WER in english ~ 5% | multilingual ~10%

SCREENSHOT



SYSTEM FLOWCHART

```

graph TD
    User((User)) --> STT[STT / Text Input]
    STT --> Context[Context & History]
    Context --> Core[Core AI Reasoning]
    Core --> UI[UI / TTS Output]
    Core --> Tools[Tools Router]
    Tools --> Core
    
```

- STT > Whisper-Small-multilingual
- LLM > Qwen3 4B 4 Bit
- HomeKit > HMHomeManager/Homekit API
- Web API > Brave Answer Api
- TTS > AVSpeechSynthesizer

FUTURE WORK

- Multimodal LLM (Vision)
- System Level Wake word detection
- on-device RAG