

MirAI

Technical Guide

Student Name	Student ID	Student Email
Georgijs Pitkevics	19355266	georgijs.pitkevics2@mail.dcu.ie
Chee Hin Choa	21100497	chee.choa2@mail.dcu.ie

Supervisor Name	Student ID
Gareth Jones	gareth.jones@dcu.ie

Date: 2 May 2025

MirAI is a local hosted smart assistant developed to provide an alternative option for privacy concerns associated with cloud-based assistants, such as Google Home, or Amazon Alexa, by running all data processing within the user's local environment. This backend can be run on any UNIX/Windows based hardware for stronger processing power, while the frontend can be run on any device that supports a modern browser. Giving as much customizability for users to personalize the AI assistant and integrate external API modules. This provides a great opportunity to reuse old device while not having to worry about processing power, this leaves a lot of flexibility for the user facilitating a multi-platform experience.

1. Motivation	2
2. Research	3
2.1 Key technologies considered and selected	3
STT Browser Choice	3
2.2 TTS Model	4
2.3 LLM Interfaces	5
2.4 Backend/API Considerations	5
3. System Design	6
3.1 Overall Architecture	6
3.2 Backend Components	6
3.2.1 Backend Structure	6
3.2.2 APIs	7
3.2.3 Database	8
3.2.4 Docker	8
3.3 Frontend Components	8
3.3.1 Frontend Pages	9
3.3.2 Components	10
3.3.3 Services	10
3.3.4 Config	11
3.4 Module Interactions	11
4. Implementation	11
4.1 Backend development Process	11
4.2 Backend Libraries Used	11
4.3 Backend Code Organization	12
4.4 Backend Key Implementation Features	13
4.4.1 NLP Service: Factual vs Opinion Analysis	13
4.4.2 Backend: API Module & Processing	14
4.4.3 Backend: Dynamic Prompt Building	15
4.4.5 Backend: WebSocket Broadcast System	16
4.4.6 Backend: SearXNG Web Search Integration	17
4.5 Docker Deployment	17
4.6 Frontend	18
4.6.1 Frontend WakeWord & STT	18
4.6.2 Frontend WebSocket	19
4.6.3 Config	20
4.7 Version Control & Workflow	21
5. Testing and CI/CD	21
5.1 Backend Testing	21

5.1.1 Coverage	21
5.1.3 Exclusions and Rationale	22
5.1.4 CI/CD	22
5.1.5 Manual Prompt Testing	22
5.1.6 TTS Testing	23
5.2 Frontend Testing	24
5.2.1 Testing Frameworks	24
5.2.2 Unit Testing	24
5.2.3 System Testing	25
5.2.4 Ad Hoc & Manual Testing	26
5.2.4 Testing Coverage	26
6. Problems Encountered and Solutions	27
6.1 Wake Word Training and Detection	27
6.2 Speech-to-Text Performance	27
7. Results	27
7.1 Feature	28
7.2 Use Case Execution Summaries	28
8. Future Work	30
8.1 Testing	30
8.2 Wake Word alternative	30
8.3 Smart Home Integration	31
8.3 Global multi-agent chat	31

1. Motivation

The goal of this project is to create a local hosted personal assistant, with the ability to personalize the AI assistants. The backend system is capable of managing different large language models, answering trivia questions, and generating a personalized voice back to the user. There is an increasing demand for personalized AI assistants in homes and businesses for privacy concerns, highlighting the need for flexible, self-hostable solutions that are not dependent on cloud providers.

2. Research

2.1 Key technologies considered and selected

- FastAPI: for building a fast, asynchronous, and scalable backend API service.
- Coqui TTS: for text-to-speech (TTS) generation.
- Ollama: for the ability to run open-source large language models (LLM) locally.
- WebSocket protocols: to enable real-time, bidirectional communication between the server and the clients.
- PicoVoice: for wake-word detection and speech recognition on browsers.

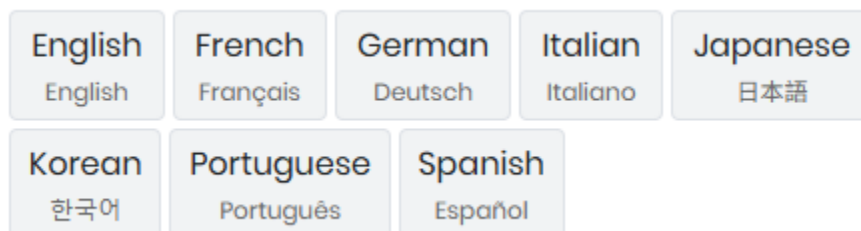
STT Browser Choice

Here are our findings about options available for STT on browser:

Library	Findings
Web Speech API	Sends data to external service for processing, doesn't align with our project focus.
Whisper(WASM)	A open-source project with the ability to run locally, however the file size is large and brings a dependency on the client's hardware.
Vosk-browser	A open-source project with ability to customize vocabulary, similar issue with whisper(WASM)
Picovoice(SDK)	Non open-source library, providing a fast and low-latency response with ability to run offline, however would have licensing limits.

We believe Picovoice would be the best suited for MirAI, in comparison with other available options, it doesn't not have dependency on client side hardware providing extreme flexibility, promoting reuse of older devices while still able to run offline, prioritizing user privacy.

Picovoice also provides various different models (e.g, speech recognition, wake word, and more) which is what MirAI needed. It also provides a quick, lightweight training for a custom wake word model, granting users the ability to create their own custom wake word for MirAI. Beside english model, Picovoice also supports multiple languages (e.g, french, spanish, japanese), allowing users to switch between languages but simply downloading and switching the model paths in MirAI settings.



2.2 TTS Model

When determining criteria for choosing a TTS model, we decided that the main criterion would be:

1. Ease of training with custom voices.
2. Ease of implementation without requiring external services hosted outside the user's LAN network.
3. Open-source software with no licensing fees.

We explored a number of different options for TTS models such as bark from Suno, ChatTTS, and XTTS-v2.

Feature/Criterion	Bark (Suno)	ChatTTS	XTTS-v2
<i>Custom Voice Integration</i>	complex/data-intensive	conversational style, fine-tuning possible	Excellent
<i>Implementation Effort</i>	Moderate to High (Resource intensive)	Moderate	Low: Pip installable, Hugging Face integration
<i>Local Deployment</i>	Yes	Yes	Yes
<i>External Service Dependency</i>	No	No	No
<i>Open Source & Licensing</i>	Yes (MIT License)	Yes (Research-focused license)	Yes (Coqui Public Model License - Permissive)
<i>Speech Quality & Naturalness</i>	High (can include non-speech sounds)	High (optimized for dialogue prosody)	High (Multilingual, good cloning fidelity)
<i>Maintenance & Support</i>	Active (by Suno & community)	Active (Community interest)	Active (Community, widely adopted)

We concluded that XTTS-v2 bu Coqui is the best option, for the following reasons:

- It doesn't require a lengthy training process, you simply provide it a 6-10 second sample speech audio file, which is trains on during execution.
- Ease of implementation, as it is readily available via hugging face and pip.
- It is a maintained open source project with up-to-date documentation and continued development support.

2.3 LLM Interfaces

We were considering and exploring a number of interfaces, we have previously used oobabooga/text-generation-webui, which is a programme that allows the user to interact with LLM's locally over a GUI and provides API endpoints.

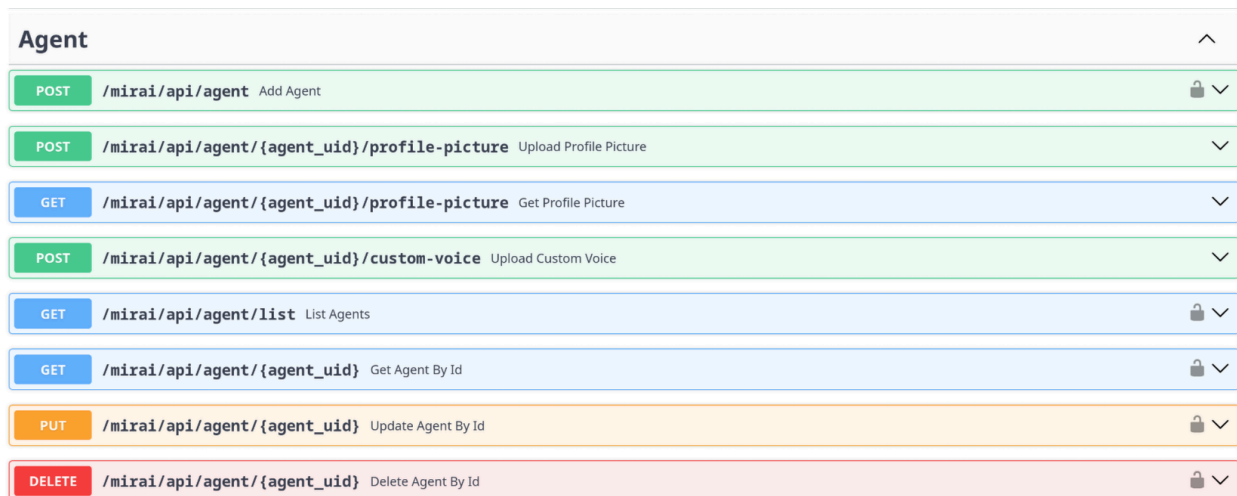
We later explored another popular option, Ollama, which we found far exceeded our expectations. It is simpler than text-generation-webui. This is because, if installed using docker, it does not provide a GUI, meaning it is a simpler API-based interface which provides the necessary functionality we need to interact with various models.

Additionally, Ollama provides an easy API-based interface to download, list and manage LLMs from Mistral and Llama.

2.4 Backend/API Considerations

Our original functional specifications outlined the use of Flask as the backend, while similar to Fast API, we found that it does not align with our goals due to it's lack of asynchronous execution.

Asynchronous execution is a key factor in allowing multiple classes & methods to function at the same time within the backend, allowing the user to be served information regularly while also allowing for background processing of tasks such as STT, NLP, and RAG.



Agent			
POST	/mirai/api/agent	Add Agent	🔒 ⌵
POST	/mirai/api/agent/{agent_uid}/profile-picture	Upload Profile Picture	⌵
GET	/mirai/api/agent/{agent_uid}/profile-picture	Get Profile Picture	⌵
POST	/mirai/api/agent/{agent_uid}/custom-voice	Upload Custom Voice	⌵
GET	/mirai/api/agent/list	List Agents	🔒 ⌵
GET	/mirai/api/agent/{agent_uid}	Get Agent By Id	🔒 ⌵
PUT	/mirai/api/agent/{agent_uid}	Update Agent By Id	🔒 ⌵
DELETE	/mirai/api/agent/{agent_uid}	Delete Agent By Id	🔒 ⌵

Figure 1 - FastAPI auto-generated documentation

Additionally, as seen in fig 1, FastAPI automatically generates a web-gui that showcases the API endpoints, and allows the developers to interact and call the various endpoints by filling in the information for the HTTP request. This was instrumental in early development stages.

3. System Design

3.1 Overall Architecture

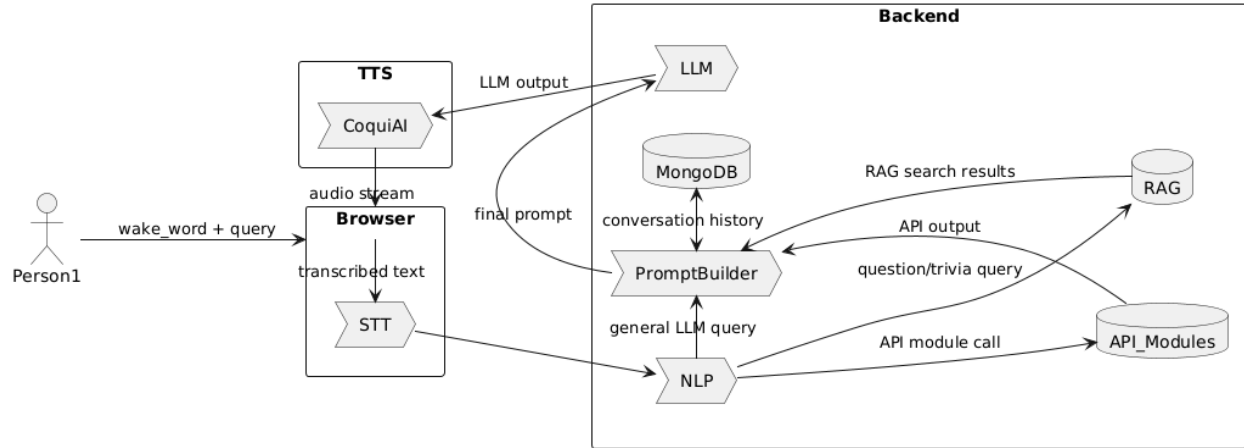


Figure 2 - Data Flow Diagram (DFD)

The DFD pictured in fig 2, showcases the overall architecture has been followed and used as the baseline for system architecture throughout the development, from start to finish.

The MirAI system follows a modernized microservices architecture approach, with divided frontend and backend components. The system is containerised using Docker for scalability and deployment purposes.

3.2 Backend Components

3.2.1 Backend Structure

The backend is built on FastAPI, which is well versed for high-performance frameworks for building APIs with Python. Python is an advantageous language for this project, as it has many libraries for working with AI & LLM related tasks.

The backend hosts a modular structure as follows:

- **API Layer:** handles HTTP requests/responses, input validation, and serving information to the frontend.
- **Service Layer:** Handles the business logic and operations.
- **Data Layer:** data handling, validation, and retrieval operations.

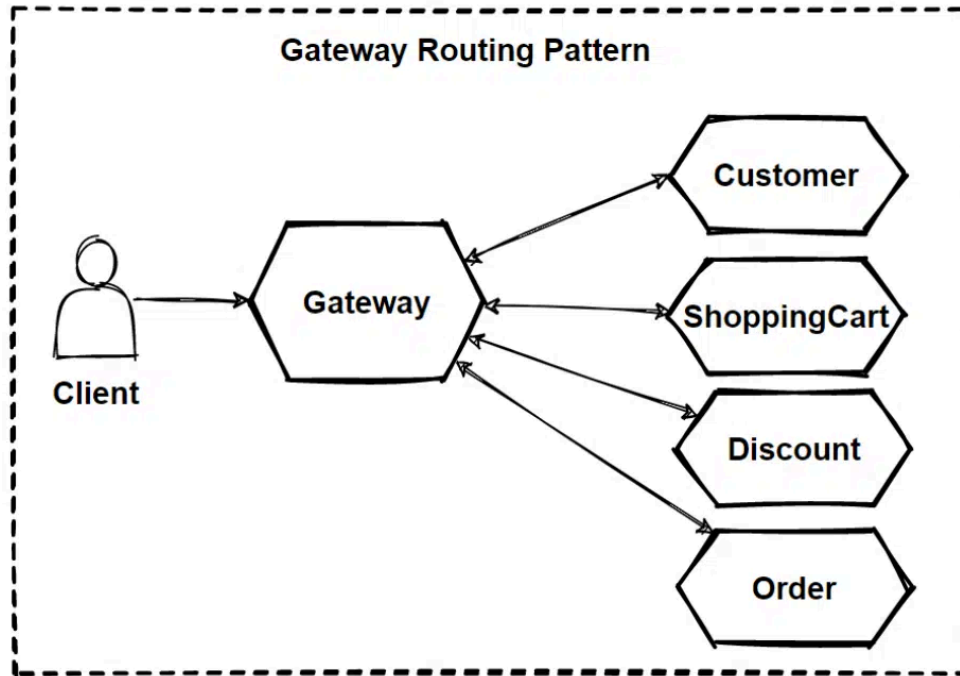


Figure 3 - Gateway Routing Pattern

The structure follows a Router → Service → Model pattern outlined in fig 3 [1] where:

1. The client send a request to the backend API layer.
2. Router validates the request, passes it to the corresponding process.
3. Service performs the logic, interacting (as necessary) with:
 - a. Database for data retrieval
 - b. LLM to generate a response
 - c. TTS
 - d. SearXNG search for RAG
4. Results are returned to the client with the REST API and/or Websocket.

3.2.2 APIs

The project follows a RESTful architecture approach with WebSocket support for RTC (real-time communication).

The following are the main API endpoint groups:

- **Authentication:** User registration, login, token management.
- **Conversation:** Managing multiple instances of conversations with Agents.
- **TTS:** Text-to-speech
- **LLM:** Manages LLM Access via Ollama.
- **Agent:** Manages AI agent profiles and configuration.
- **Global Conversation:** Multi-agent conversation.
- **API Module:** Custom user-defined API endpoints to be used as part of the Agent's prompt.
- **Websocket:** Managing RTC channels for audio streaming.

3.2.3 Database

The system uses MongoDB, which is a NoSQL document database. Due to the database frameworks NoSQL-nature, we were able to experiment with various data structures for data storage in early development stages. Without the need for rigid schema definitions, changes that are made in the backend with regards to pydantic model design, are automatically propagated to the database once a database operation is run.

MongoDB excels at Read-Heavy Operations, which are often run when fetching data like conversation history and Agent profiles.

The main collections within the database include:

- **Users:** Account information and permissions.
- **Conversations:** Multiple conversations and conversation metadata.
- **Agents:** Agent profiles and configurations.
- **Statistics:** Performance analytics of the performance of different systems.
- **Settings:** User and system settings.

3.2.4 Docker

The key components of the system are containerized in Docker containers.

- **MongoDB:** Database.
- **Ollama:** Local LLM instance.
- **SearXNG:** Privacy-conscious search engine.
- **Redis:** In-memory storage for SearXNG.

Docker allows MirAI to be run on any system with minimal prior configuration. If it supports docker - it supports MirAI.

3.3 Frontend Components

The frontend of MirAI is built using ReactJS framework, with a strong focus on maintainability and component reusability. Following best practices for future continuous development, its file structure as figure 4:



Figure 4 - frontend file structure

To ensure a consistent design theming and pattern for best user experience, MirAI utilizes react library Material UI (MUI) alongside with TailwindCSS across the application. It enables a global dark theme across the entire application for accessibility and visual clarity. The user interface introduces a tabbed navigation structure, allowing dynamically changing layout between major pages like Conversations, Global Chat, API Module Configuration, Agent Management, Settings, and Statistics. The layout is fully responsive, with logic to adapt to mobile and desktop viewports using dynamic layout changes through JavaScript-based media checks and flexible layout containers. This provides an intuitive and clean user experience regardless of the device being used.

3.3.1 Frontend Pages

Under pages/ folder, it contains each page in MirAI as a React component, following a modular and compartmentalized architecture. These pages are navigated using tab selection managed in App.js together with sidebar.js. Most pages aim to offload logic to child components for maintainability (e.g., `<Conversations />`, `<AgentConfiguration />`).

AgentConfigurationPage Component Relationship

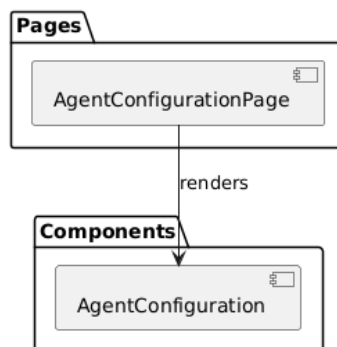


Figure 5 - AgentConfigurationPage Diagram

```
const AgentConfigurationPage = () => {  
  return <AgentConfiguration />;  
};  
  
export default AgentConfigurationPage;
```

Figure 6 - AgentConfigurationPage Code Snippet

Certain pages like SettingsPage act as a bridge between different components (e.g., Settings, LLMConfig, ModelManager).

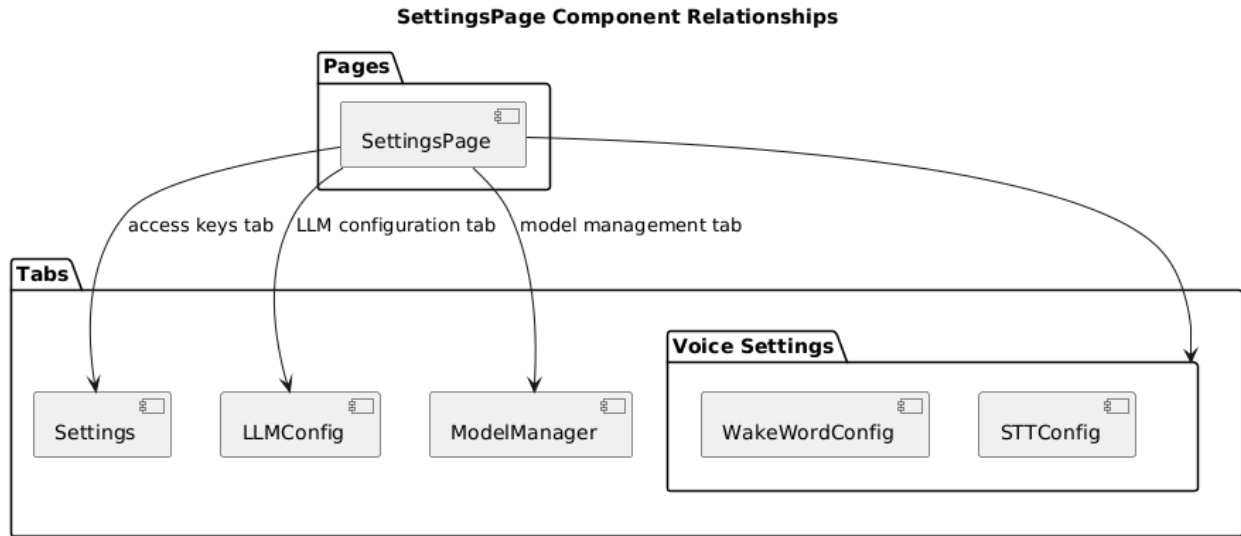


Figure 7 - SettingsPage File Structure Diagram

3.3.2 Components

In the component/ folder, files are split into logical domains such as agent, conversation, wakeword, stt, etc.

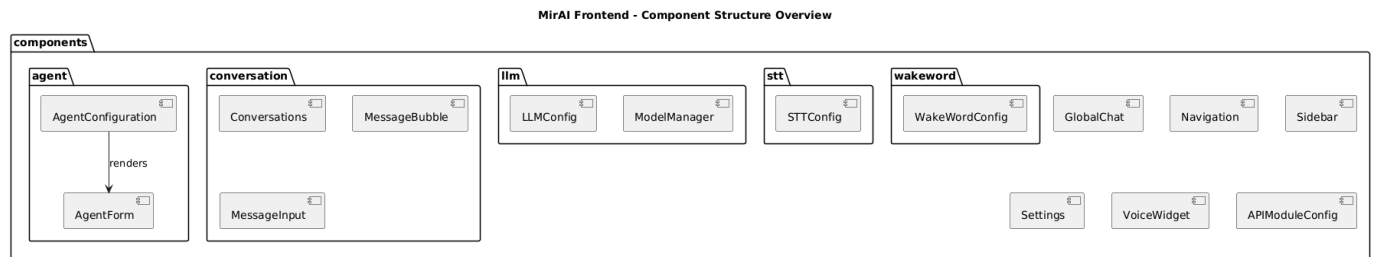


Figure 8 - Components Structure Diagram

3.3.3 Services

The services/ folder contains API abstraction layers needed for MirAI while maintaining a clean separation between UI components and backend logic. Every feature (e.g., agent management, LLM configs, TTS, WebSocket) has a corresponding service module to:

- Handle API endpoint calls.
- Format request bodies and parameters.
- Return promise-based results for React components.
- Handle error response.

This modular design promotes a better practice at readability, code reusability and easier maintenance across all pages and components.

3.3.4 Config

A centralized control over different services, `api.config.js` are used to store all API and WebSocket base URLs, headers, and endpoint routes. This layer abstracts service-specific routes from the rest of the application and provides a base API fetch function for easier maintenance and scalability.

`API_BASE_URL` and `WS_BASE_URL` here are environment-aware, detecting existing environment variables, giving flexibility during deployment (local development or hosted).

3.4 Module Interactions

- **PromptBuilder:** Dynamically constructs prompts for the LLM based on user input and system instructions.
- **API Module:** Enables dynamic, user-defined, integration with external third-party services.
- **TTS Module:** Converts text to natural-sounding speech with ability to use a sample voice file as the Agent's voice.
- **Websocket:** Used for streaming responses in the form of text and audio from LLMs.
- **Ollama:** Ability to download and use various open-source models from Llama, Mistral, and others.
- **SearXNG:** provides privacy-focused web search results from multiple search engines, and provides the results to the PromptBuilder.

4. Implementation

4.1 Backend development Process

MirAI was built following a modular, service-based approach with the following key development steps:

1. **Backend Foundation:** FastAPI framework setup with routing and middleware.
2. **Data Handling/Storage:** Implementation of MongoDB for storage with Motor for async operations.
3. **Core Services:** Created modular services, 1-by-1 for NLP, TTS, and LLM.
4. **API Layer:** Created REST endpoints and WebSocket connections for RTC.
5. **Integration Components:** Added SearXNG for search capabilities and Ollama for LLM processing
6. **Docker Containerization:** Containerized services for easy set-up.
7. **Unit Testing:** Creating unit tests to validate proper and expected functionality.

4.2 Backend Libraries Used

The system relies on the following libraries for backend functionality:

```
# Web framework and API
fastapi
uvicorn[standard]
```

```

python-multipart
websockets>=10.4

# Authentication and Security
python-jose[cryptography]
passlib[bcrypt]
pydantic[email]

# Database
motor
pymongo

# NLP and Text Processing
spacy>=3.7.0
nltk>=3.8.1
en-core-web-sm

# Text-to-Speech
TTS==0.22.0
torch>=2.1.0
torchaudio>=2.1.0
pydub

# HTTP and Web
httpx
aiohttp>=3.9.0
beautifulsoup4>=4.12.0

```

4.3 Backend Code Organization

The backend follows this structured file system:

```

src/backend/
├── api/                # API layer
│   ├── routers/        # API endpoints
│   ├── services/        # Business logic
│   ├── models.py        # Data models/schemas
│   ├── database.py      # Database connections
│   └── security.py      # Authentication
├── promptBuilderModule/ # Prompt engineering
├── ttsModule/           # Text-to-speech processing
├── docker/              # Container configurations
├── tests/               # Test suite
└── main.py              # Application entry point

```

4.4 Backend Key Implementation Features

4.4.1 NLP Service: Factual vs Opinion Analysis

The system intelligently distinguishes between factual and opinion based queries, where RAG via SearXNG is required for factual queries:

```
async def analyze_query(query: str) -> Dict[str, Any]:
    """Analyzes a query to determine if it's a trivia or general query."""
    doc = nlp(query)

    # Extract linguistic features
    important_nouns = []
    important_verbs = []
    for token in doc:
        if token.pos_ == "NOUN" and not token.is_stop:
            important_nouns.append(token.text.lower())
        elif token.pos_ == "VERB" and not token.is_stop:
            important_verbs.append(token.text.lower())

    # Calculate factual score
    factual_score = 0
    for starter in FACTUAL_INDICATORS["question_starters"]:
        if clean_query.startswith(starter + " "):
            factual_score += 2
            break

    # Calculate opinion score
    opinion_score = 0
    for indicator in OPINION_INDICATORS:
        if indicator in clean_query:
            opinion_score += 2

    # Determine query type
    is_trivia = factual_score > opinion_score and factual_score >= 2

    return {
        "query_type": QueryType.TRIVIA if is_trivia else QueryType.GENERAL,
        "factual_score": factual_score,
        "opinion_score": opinion_score
    }
```

This critical feature utilizes a scoring system to allocate a score based on factual vs opinion based queries.

- **Factual Score:**

question words like "who", "what", "when", "where"	+2 points
verbs like "invented", "discovered", "built"	+1.5 points
factual nouns like "year", "date", "population"	+1 point

named entities (PERSON, ORGANIZATION, LOCATION)	+1.5 points
question mark ("??")	+1 point
Short, direct questions	+1 point

- **Opinion Score:**

opinion indicators like "favorite", "best", "worst", "recommend"	+2 points
personal references like "you" or "your"	+1 point
"why" or containing "explain" or "describe"	+1.5 points
opinion markers like "your opinion" or "what do you think"	Override scoring

The system determines a query type when the factual score is larger than the opinion score (and reaches a minimal threshold). We designed and iterated on the system to be biased towards factual score, as providing a false positive on factual query is not harmful to the resulting response when compared to a false positive opinion query in the case that the user queried for up-to-date factual information.

4.4.2 Backend: API Module & Processing

The system identifies API Module's trigger phrases through pattern-matching and regex:

```

async def find_matching_api_module(query: str):
    """Find an API module that matches the user query."""
    modules = await get_all_api_modules()
    query_lower = query.lower()

    for module in modules:
        for trigger in module.trigger_phrases:
            # Extract variables from trigger templates
            placeholders = extract_variables_from_trigger(trigger)

            if placeholders:
                # Convert trigger to regex pattern for variable extraction
                pattern = re.escape(trigger)
                for placeholder in placeholders:
                    escaped_placeholder = "\\{" + placeholder + "\\}"
                    pattern = pattern.replace(
                        escaped_placeholder,
                        f"(?P<{placeholder}>[\\w\\s\\-\\.\\,]+)"
                    )

                # Match against query
                match = re.match(f"^{pattern}$", query, re.IGNORECASE)
                if match:
                    variables = match.groupdict()

```

```
variables = {k: v.strip() for k, v in variables.items()}
return module, trigger, variables
```

This pattern matching enabled dynamic API integration. The system detects if a trigger phrase pertaining to a user-defined API module matches the user's current message.

For example; a trigger phrase like "What's the weather in {city}" is transformed into a regex pattern that can match "What's the weather in London" and extract "London" as the city variable, which is then used to call the user-defined external API endpoint.

4.4.3 Backend: Dynamic Prompt Building

The PromptBuilder dynamically constructs prompts for the LLM, and includes necessary parameters like agent personality, user query, conversation history, and RAG/API Module results.

```
@staticmethod
def build_system_prompt(
    personality_prompt: str,
    tts_instructions: Optional[str] = None,
    rag_context: Optional[str] = None,
) -> str:
    """Build the system prompt with agent's personality and context."""
    system_prompt = """You are an AI assistant.

## CONVERSATION CONTEXT INSTRUCTIONS
1. If the user's query appears to lack specific context, assume it refers to the most recent
conversation points.
2. Do not introduce new topics when the query is context-less - continue the existing
discussion.
"""

    # Add search results or API information if available
    if rag_context and "API MODULE RESULT:" in rag_context:
        system_prompt += f"""

## API MODULE INFORMATION
{rag_context}"""
        elif rag_context:
            system_prompt += f"""

## FACTUAL INFORMATION
{rag_context}"""

    # Add agent personality
    system_prompt += f"""

## AGENT IDENTITY
{personality_prompt}
"""
```



```
return system_prompt
```

This PromptBuilder creates structured prompts that give instructions to the LLM in clear organised sections. The hierarchical approach helps the model perceive the relative importance of different instructions/context.

- For factual queries, it adds a “## *FACTUAL INFORMATION*” header.
- For API module queries, it adds a “## *API MODULE INFORMATION*” header.
- Every prompt incorporates the agent’s personality traits, as well as the TTS instructions from the LLM configuration.

4.4.5 Backend: WebSocket Broadcast System

The websocket broadcast system implements a pub-sub mechanism for real-time updates

```
# Store active connections
active_connections: Dict[str, Set[WebSocket]] = {"global": set()}
conversation_connections: Dict[str, Set[WebSocket]] = {}

async def broadcast_to_conversation(conversation_id: str, message: str):
    """Broadcast message to all connections for a conversation."""
    if conversation_id in conversation_connections:
        disconnected = set()
        for connection in conversation_connections[conversation_id]:
            try:
                await connection.send_text(message)
            except Exception as e:
                disconnected.add(connection)

        # Clean up disconnected clients
        for conn in disconnected:
            conversation_connections[conversation_id].remove(conn)

@router.websocket("/conversation/{conversation_id}")
async def conversation_websocket_endpoint(websocket: WebSocket, conversation_id: str):
    """WebSocket endpoint for conversation-specific updates."""
    await websocket.accept()

    # Add connection to conversation connections
    if conversation_id not in conversation_connections:
        conversation_connections[conversation_id] = set()
        conversation_connections[conversation_id].add(websocket)

    # Subscribe to conversation-specific messages
    channel = f"conversation:{conversation_id}:messages"
    pubsub_client.subscribe(channel, handle_message)
```

Each conversation has its own broadcast channel, which ensures that messages do not leak between user sessions, ensuring secure delivery to all connected clients.

4.4.6 Backend: SearXNG Web Search Integration

SearXNG is hosted in a docker container and leveraged with this python integration to provide we search capabilities:

```
async def search_web(query: str, max_results: int = MAX_RESULTS) -> List[Dict[str, Any]]:
    """Search the web using SearXNG for a given query."""
    # Check cache first
    cache_key = f"{query}_{max_results}"
    if cache_key in SEARCH_CACHE:
        cache_time, results = SEARCH_CACHE[cache_key]
        if time.time() - cache_time < CACHE_EXPIRY:
            return results

    # Try full engine set first
    results = await _search_with_searxng(query, max_results, DEFAULT_ENGINES)

    # If no results, try fallback engines
    if not results:
        results = await _search_with_searxng(query, max_results, FALLBACK_ENGINES)

    # Cache results
    if results:
        SEARCH_CACHE[cache_key] = (time.time(), results)

    return results
```

- The search engine utilizes multiple engines for redundancy such as Brave, Google and Bing.
- Redis provides an in-memory caching system to reduce latency and external API calls.
- It incorporates resilient retry logic to get around temporary search failures.
- Sorts results by relevance score.

4.5 Docker Deployment

The system utilises Docker Compose for easily orchestrating multiple services:

```
services:
  mongodb:
    image: mongo:latest
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: admin
      MONGO_INITDB_ROOT_PASSWORD: password
    volumes:
      - mongodb_data:/data/db

  ollama:
    image: ollama/ollama:latest
    ports:
      - "11434:11434"
```

```

volumes:
- ollama_data:/root/.ollama
runtime: nvidia
environment:
- NVIDIA_VISIBLE_DEVICES=all

searxng:
  image: searxng/searxng:latest
  ports:
  - "8080:8080"
  environment:
  - BASE_URL=http://localhost:8080/
  - INSTANCE_NAME=MirAI SearXNG
  - SEARXNG_SERVER_SECRET_KEY=MirAI-SearchEngine-Key
  volumes:
  - searxng_data:/etc/searxng
  - ./searxng_config/settings.yml:/etc/searxng/settings.yml
  depends_on:
  - redis

redis:
  image: redis:alpine
  volumes:
  - redis_data:/data

```

This container approach ensure consistent environments throughout development and production, with filesystems and components isolated from the base Operating System.

4.6 Frontend

4.6.1 Frontend WakeWord & STT

In order to achieve wake word detection and speech recognition on the browser, MirAI implemented **leopard-react** and **porcupine-react** Library provided by **PicoVoice** following the documentation provided from its website.

The following pseudo code shows the logic in ConversationVoice.js.

The initialization logic for MirAI's voice input system is handled within React useEffect hook, which runs when the component is mounted or when dependencies (like agent configuration or access key) changes. This function first validates the requirement such as the existence of a Picovoice access key, a selected agent, and that the application is being served over a secure context (HTTPS is required for microphone access). Then it performs a cleanup of previously running voice processes to prevent duplication or conflicts. After validating the availability of required models (**leopard_params.pv** for speech-to-text and **porcupine_params.pv** for wake word detection), it initializes the speech recognition model and wake word model with a selected or default wake word. If the microphone was already enabled before initialization, the system will automatically starts listening for the wake word. This logic offers a state-aware setup process for voice input and also handles error gracefully.

```

useEffect(() => {
  const initializeVoice = async () => {
    if (!accessKey || !agent || !isSecureContext) return;

    try {
      await cleanup(); // Stop any existing audio processing

      // Check models exist
      const leopardModel = `${window.location.origin}/models/leopard_params.pv`;
      const porcupineModel = `${window.location.origin}/models/porcupine_params.pv`;

      // Initialize STT (Leopard)
      await initStt(accessKey, { publicPath: leopardModel });

      // Set default or user-defined wake word
      const keyword = { builtin: builtInKeywords.includes(agent.built_in_wakeword)
        ? agent.built_in_wakeword
        : "Computer"
      };

      // Initialize Wake Word detection (Porcupine)
      await initWakeWord(accessKey, [keyword], { publicPath: porcupineModel });

      // Start detection if mic is already enabled
      if (micEnabled) await startWakeWordDetection();

    } catch (err) {
      console.error("Voice initialization failed:", err);
      setError(`Voice setup error: ${err.message}`);
    }
  };

  initializeVoice();
  return () => cleanup(); // Teardown on unmount
}, [agent, accessKey]);

```

4.6.2 Frontend WebSocket

WebSocket is one of the core component for MirAI, as it allows the server and client to have real-time communication (e.g, live messages). **websocketManager.js** implements a **WebSocket abstraction class** to be used across the application. It manages the connection lifecycle with **connect** and **disconnect**, message parsing and dispatching to registered handlers, and automatically reconnect with exponential backoff when connection failed. This allow to support **multiple message handlers**, making it possible to plug different components into the same stream. It is a helper class for **websocketService.js** to manage per-endpoint socket channels (e.g., global chat or conversation threads).

```

class WebSocketManager {
  constructor(baseUrl) {
    this.baseUrl = baseUrl;
    this.socket = null;
  }
}

```

```

    this.handlers = [];
    this.endpoint = null;
    this.isConnected = false;
  }

  connect(endpoint) {
    if (alreadyConnectedTo(endpoint)) return;
    disconnectPrevious();
    openNewSocket(this.baseUrl + "/" + endpoint);
    socket.onmessage = (data) => callAllHandlers(data);
    socket.onclose = () => tryReconnect();
  }

  addMessageHandler(handler) {
    this.handlers.push(handler);
    return () => remove(handler);
  }

  disconnect() {
    socket.close();
  }
}

```

4.6.3 Config

Here is how MirAI implemented **api.config.js** as the centralized configuration constructing base URL (API & WebSocket), Header defaults (Content-type), and REST endpoint path (via ENDPOINT object). It can dynamically resolves API_BASE_URL and WS_BASE_URL based on if environment variables was provided or use defaults based on current window location. It also exports a **fetchAPI()** helper that wrap **fetch()** with error handling and support for **FormData** uploads.

```

const API_BASE_URL = getEnvOrFallback(); // from window.location
const WS_BASE_URL = inferWsUrlFrom(API_BASE_URL);
const ENDPOINTS = { ... }; // defines all backend paths

async function fetchAPI(endpoint, options) {
  const url = API_BASE_URL + endpoint;
  const headers = { "Content-Type": "application/json", ...options.headers };

  // If uploading files, remove content-type
  if (options.body instanceof FormData) removeContentType(headers);

  const response = await fetch(url, { ...options, headers });

  if (response.ok) return parseJson(response);
  else throw Error("Request failed");
}

```

4.7 Version Control & Workflow

A **Gitlab DCU** repository was utilized for hosting our code and ensuring version control using git.

Specifically the project followed the **Git Flow** branching model to ensure a well-maintained collaborative development environment, where all activities are tracked.

The Git Flow branching model utilises the following branches and sub-branches:

- **main** - stable production branch.
- **develop** - where new completed features are merged into before entering the main branch.
- **feature/*** - individual branches for new features, ie; “*feature/promptBuilderModule*”.
- **bugfix/*** - fixing non-critical bugs found in development ie; “*bugfix/conversation-error*”.

All source code and configuration files are version controlled through git allowing easy rollback when needed.

5. Testing and CI/CD

5.1 Backend Testing

The MirAI backend utilizes a testing approach that combines unit testing and mock-based testing. Mocks simulate the external services like SearXNG and Ollama. This ensures that the core functionality is reliable and with the implementation of GitLab pipelines, allows us to run the tests every time code is pushed.

5.1.1 Coverage

The test coverage varies across the different backend components:

Component Area	Coverage	Notes
<i>Overall</i>	29%	Indicates areas for future improvement
<i>Services Package</i>	44%	Focus on core business logic
conversation_service	86%	High coverage for critical sections
agent_service	86%	High agent management coverage
nlp_service	70%	Good coverage for intent understanding
rag_service	71%	Good RAG logic coverage

search_service	41%	Limited by external dependency
api_module_service	20%	Needs improvement for dynamic API calls
tts_service	17%	Limited by GPU hardware dependencies
Other services	0.0%	Untested

5.1.3 Exclusions and Rationale

Some components have been strategically excluded from the metrics for coverage:

Excluded Component	Reason	Testing Strategy
llm_service.py	External Ollama API calls	Higher-level integration testing
ttsModule/	Hardware-dependant	Test configuration, not generation
database.py	Environment-specific connection required	Integration tests with test DB
file_storage_service.py	File systems vary by environment	Generate mock file system

Our focused approach prioritizes testing unique business logic.

5.1.4 CI/CD

All tests are automatically run via the GitLab CI on every push, which prevents regression of the codebase. Coverage reposts are generated and stored for a fixed time-period, to ensure consistent quality control.

5.1.5 Manual Prompt Testing

Prompt Testing requires manual testing by a human to determine subjectively whether one prompt is better than another, due to the nature of LLMs and their tendency to hallucinate and produce varying outputs, unit tests are not possible.

A list of static messages is given to the agents, which then respond, and the responses are rated with a like or dislike directly in the UI. This message metadata is then compiled by the statistics service and served to the frontend in the “Statistics Dashboard” as seen in Fig 9.

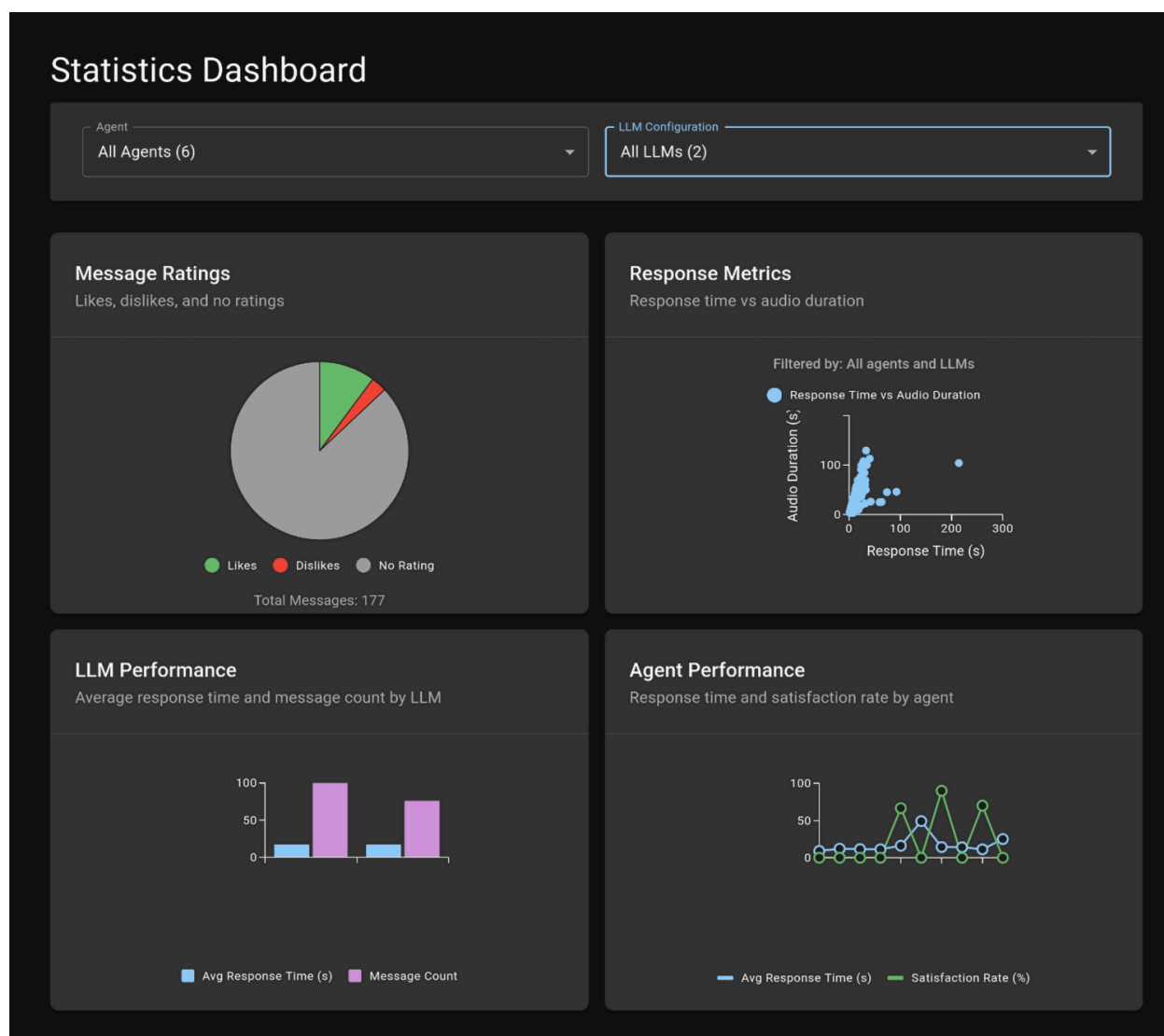


Figure 9 - Statistics Dashboard.

This message rating statistic allowed us, the developers, to perform manual tweaks to the prompt to perform A/B testing and compare prompts in an iterative manner by creating new LLM and Agent configurations for each iteration, ensuring that we don't regress in our prompt testing strategy.

5.1.6 TTS Testing

Throughout the manual prompt testing stages, data for response time vs audio duration is collected with each message. Meaning that the end result is independent of the prompt used, and purely associates the time it took to generate a TTS audio file with comparison to the duration of the file in seconds.

We came up with two solution for TTS implementation:

1. **Parallel:** Split the resulting Agent message into multiple sentences and process them asynchronously using the CPU.

2. **Serial:** Split the message into sentences and one-by-one as a whole using the GPU process each message sequentially.

As seen from our Response Metrics graph below in figs 5 & 6, the results show an overall faster response time for Serial execution.

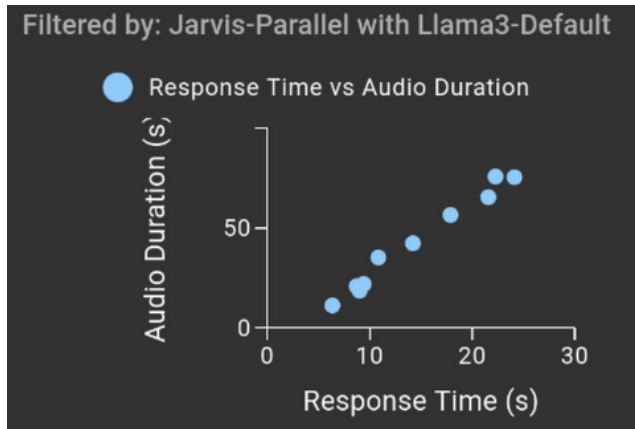


Fig 5 - Parallel Execution.

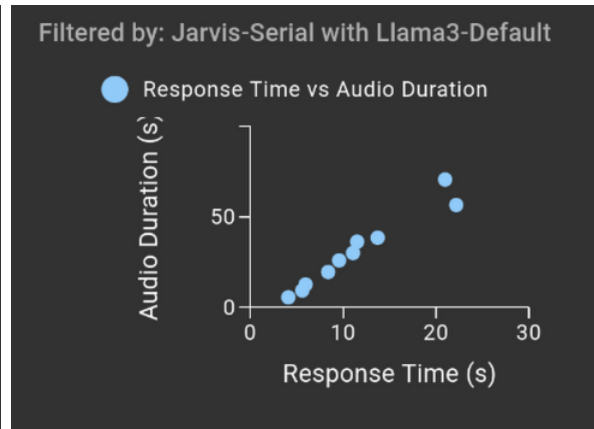


Fig 6 - Serial Execution.

As expected, the longer the duration, the longer the execution time for both, less so for Serial as it scales better.

5.2 Frontend Testing

The MirAI project involves a lot of user interaction via a graphical interface on the browser. As such, the testing strategy focused primarily on **unit testing for UI components**, **manual system testing**, and **interface behaviour workflow** testing. The frontend testing efforts aimed to confirm that individual components render and behave correctly under different states and user interactions, while also verifying that integrated flows.

5.2.1 Testing Frameworks

Testing was conducted using the following testing stack:

- **Jest** for test execution and snapshot testing
- **@testing-library/react** for simulating user interactions

5.2.2 Unit Testing

Tests were created to validate component logic, props handling, rendering outcomes, and edge case scenarios. For example, all the abstraction page layers render components intendedly and handle props input. Testing was also conducted on partial core components such as Settings.js and STTConfig.js, which play a key role in enabling voice features.

Here are summary of test case we have covered:

- Settings Component
 - Renders settings form correctly.
 - Loads and displays existing access key.
 - Handles save action correctly.
 - Handle error states correctly.
 - Toggle password visibility.
- Navigation Component
 - Renders navigation with all main menu items.
 - Handles tab changes correctly.
 - Highlights active tab correctly.
- STTConfig
 - Renders all UI components.
 - Updates form input correctly.
 - Display error messages appropriately.
- SettingsPage
 - Renders all main tabs.
 - Switches between tabs correctly.
 - Maintains tab state after navigation.
- GlobalChatPage
 - Renders the GlobalChat component with config.
 - Passes config prop to GlobalChat component.
- APIModuleConfigPage
 - Renders the page container with correct styling.
 - Renders the APIModuleConfig component.
- AgentConfigurationPage
 - Renders the page container with correct styling.
 - Renders the AgentConfiguration component.
- ConversationsPage
 - Renders the page container with correct styling.
 - Renders the Conversations component.

5.2.3 System Testing

System testing was informal due to time constraint and large scope, we conducted functional testing through the actual running interface. Key user flows such as sending a message in the global chat, saving configuration changes in the settings page and receiving TTS-generated responses were tested through development phases. These flows involved integration between local models (wake word, STT), WebSocket communication, backend endpoints and server models (TTS, LLM). Although we did not cover it by automated tests, their behaviour will be validated during demo runs.

5.2.4 Ad Hoc & Manual Testing

During every feature development, we manually tested UI interactions, state updates, and API integrations to verify responsiveness and correctness. This includes direct interaction with the running application, it is especially important for speech related features, which are harder to automate and require microphone input, real time model loading, and asynchronous state management. For these cases, we manually verified the expected result by comparing visible UI behavior and browser console logs.

5.2.4 Testing Coverage

We also implemented a code coverage analysis via Jest's `coverage` flag. This produces a report artifact for every integration in the project shown below:

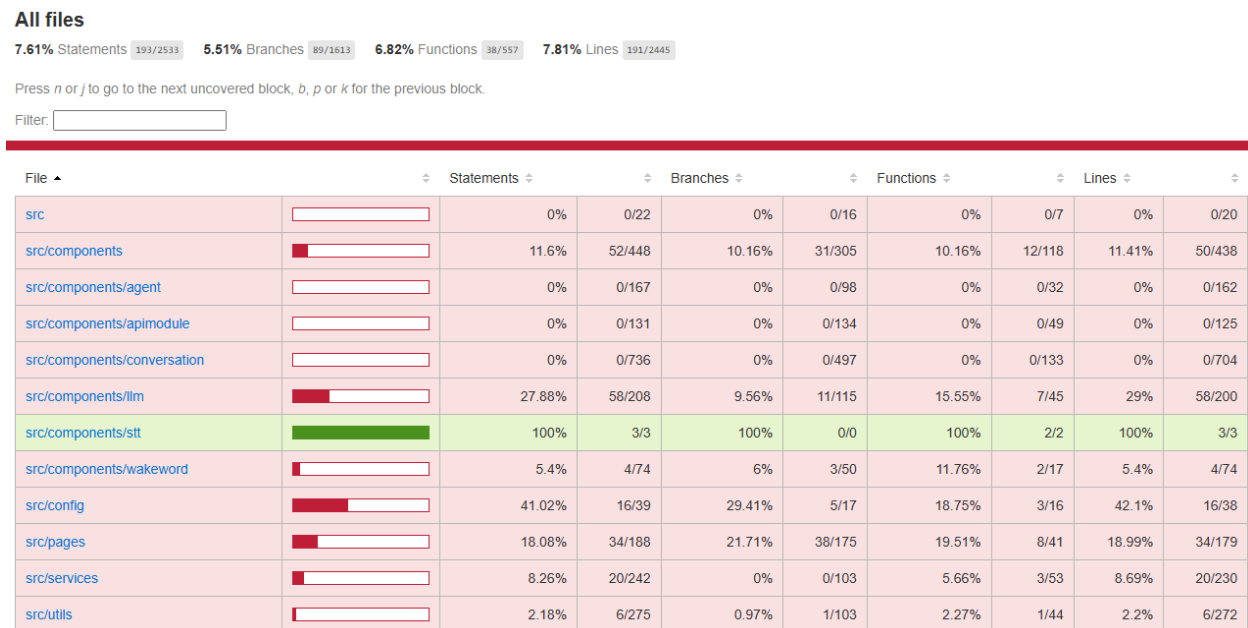


Figure 10- Frontend Testing Coverage Report

This latest coverage results show that:

- Statement coverage sits at **7.61%**, mostly driven by targeted unit tests.
- The `src/components/stt` directory achieved **100% coverage** as it is small and highly testable.
- Larger, complex components such as `conversation` and `agent` had minimal coverage due to time constraints and testing complexity.
- The `config` folder achieved **42% line coverage**, due to testable logic in `fetchAPI()` and URL resolution helpers.

6. Problems Encountered and Solutions

During the development of MirAI, several technical design challenges occurred due to inexperience in the field. These challenges influenced overall architecture decisions and required trade-offs between performance, flexibility, and reliability. This gave the team a huge opportunity to improve and gain experience from it.

6.1 Wake Word Training and Detection

One of the earliest challenges involved **training a custom wake word**, specifically the non-English term “**Mirai**”. It was one of the motivations for this project to offer flexibility that other cloud assistants on the market could not provide. While open-source tools like **OpenWakeWord** offer custom wake word training, the tool is primarily optimized for English-based phonemes and requires a huge amount of training time. Furthermore, OpenWakeWord needs to run on the **backend server**, as its required significant processing power to perform well, it introduces **audio streaming latency** and increases the complexity to achieve real time responsiveness.

Solution: To address this, the project now implements **Picovoice Porcupine** for wake word detection, which runs entirely in the **client browser**. This removes the need for audio to be streamed to the backend and results in latency and slow response time. It also provides the solution to train custom wake words quicker for different languages other than English. Although this meant MirAI would have to rely on Picovoice access key and limitation to free account, it ensured consistent cross-device compatibility and real-time performance.

6.2 Speech-to-Text Performance

The initial implementation used **Vosk** as MirAI local STT engine, but it required audio to be streamed to the **backend server**, which introduced latency and unpleasant user experience, especially for short interactions. Additionally, handling WebSocket-based audio input and decoding server-side added significant overhead.

Solution: The team transitioned to using **Picovoice Leopard**, a fully client side STT engine which optimized for **browser use** and **privacy concern**. This achieves a low latency transcription directly in the user’s browser, less dependency of client’s hardware, and lessens the bandwidth needed to send data to the backend. The result was a smoother transcription, and better feedback timing.

7. Results

MirAI is now a functional, expandable voice assistant platform capable of local speech processing, dynamic personality switching, and modular API integration. It operates across a central backend and multiple frontend clients, with all conversation data stored in a local MongoDB instance. The system supports voice interaction using browser based STT and wake word detection, allowing users to customize their own assistant personalities, voice, and custom wake word. The assistant can engage in

contextual conversations and respond with both text and synthesized speech. MirAI archiving all this without the need of cloud services for core functionality.

7.1 Feature

Feature	Description
Custom Agent Configuration	Users can create agents with unique personalities and voices, including voice files and profile pictures.
Voice Interaction	Wake word and STT enabled in-browser using Picovoice SDK for fully local processing.
API Module System	Users can define voice-triggered APIs with variables and test them via the UI.
Chat History Storage	All conversations are stored in MongoDB, with metadata like response time, model used, and user feedback.
Real-time Response Streaming	Audio responses are streamed via WebSocket and served with minimal latency.
Multidevice Compatibility	A single backend supports multiple frontend clients (browsers), allowing flexible deployment.
Ollama Model Integration	Users can pull any available LLM from Ollama (e.g., llama3, mistral, etc.) via the UI, enabling flexible model swapping with agents.
RAG Enhanced LLM Responses	The assistant can perform Retrieval-Augmented Generation (RAG) using SearXNG for up-to-date external information during conversations.
Live Statistics Dashboard	Displays real-time metrics on agent usage, message ratings, and LLM performance.

7.2 Use Case Execution Summaries

Field	Details
Use Case ID	1
Title	Query Weather Info
Primary Actor	User

Goal	Retrieve up-to-date weather information using voice command.
Preconditions	Picovoice's access key is valid and internet/API is available.
Main Flow	<ol style="list-style-type: none"> 1. User says "Hey Jervis, what's the weather in Dublin?" 2. Wake word is detected and STT processes speech. 3. NLP classifies it as a weather query. 4. API Module fetches data from the weather endpoint. 5. LLM summarizes the response (optional). 6. TTS synthesizes the spoken reply. 7. Response is played via speaker.
Alternate Flow	
Postconditions	Transcript is delivered and the user receives a response.

Field	Details
Use Case ID	2
Title	Configure Agent's Wake Word
Primary Actor	User
Goal	Upload or select a custom wake word model for a specific agent.
Preconditions	User has access to the Agent Configuration page.
Main Flow	<ol style="list-style-type: none"> 1. User opens the Agent Configuration tab. 2. Create a new agent. 3. Uploads .pv file or selects from predefined list. 4. Configuration is saved and applied.
Alternate Flow	Wake word model failed to initialize(invalid type) → error message shown.
Postconditions	Wake word model updated and used by detector.

Field	Details
Use Case ID	3
Title	Create Custom API Module
Primary Actor	Power User

Goal	Extend MirAI with a new voice-triggered API integration.
Preconditions	User have access to the API Module Config page.
Main Flow	<ol style="list-style-type: none"> 1. User opens the API Module Config tab. 2. Fills in trigger words and endpoint URL. 3. Defines variable mappings. 4. Saves the module. 5. Optionally tests it with example query.
Alternate Flow	API endpoint is unreachable → warning shown on test.
Postconditions	New module is stored and active.

8. Future Work

8.1 Testing

While the project has implemented baseline unit testing, there remains large room for improvement, especially in system testing and end-to-end validation.

For the backend, future testing efforts will concentrate on:

1. Increasing coverage for `api_module_service`.
2. Adding more tests for FastAPI routers via request/response validation.
3. Conversing untested services such as `global_conversation`, `settings`, and `statistics`.

For the frontend, future testing efforts will concentrate on:

1. Coverage of more complex interaction flows (e.g., streaming back audio via `Conversation Voice`, starting a conversation via `ConversationDetails`)
2. Mocked WebSocket and API service tests to verify live updates and asynchronous flows.
3. Snapshot testing for UI components to detect unintended changes.

Covering these tests will promote long term reliability as new features emerge or adapt for other platforms.

8.2 Wake Word alternative

Currently, MirAI uses Picovoice Porcupine for client side wake word detection, having a dependency on external platform. While it offer excellent performance and privacy by running offline in the browser, its free tier account allow only one custom wake word training every 30 days, therefore removing the flexibility of creating multiple different custom wake word at start. Future iterations could explore alternative wake word solutions.

8.3 Smart Home Integration

MirAI includes a functional API module system with the ability to send and receive data from external APIs based on user queries. Due to **lack of smart home hardware**, we were unable to test modules that control real-world devices such as smart light.

In the future, we plan to:

- Implement modules for common smart home platforms (e.g., Home Assistant, Philips Hue, Tuya).
- Test commands like “turn off the living room light” through MirAI’s voice interface.
- Allow modules to trigger context-aware responses, such as status reports or automated routines.

Once hardware is available, this will become a milestone of MirAI's usefulness as a **home assistant** rather than just a chat assistant.

8.3 Global multi-agent chat

Due to the current state of the picovoice porcupine model, we were unable to utilize listening it to listen to multiple wake words simultaneously, as it is limited to one at a time.

This bottleneck, did not allow us to complete the Global Chat feature as we wanted to, meaning we confine it to only using text based input to enable inter-agent communication.

For future work, we would like to find a solution to allow the user to interact with multiple agent personalities in the same conversation over STT.