TED UNIVERSITY

High-Level Design Report
Chef Bono

Team Members:

Berkan Gökgöz
Buse Şahin
Yelda Sıla Mumcu
Eren Serdar

1.0 Introduction

Recent advances in artificial intelligence have made it possible to develop smart assistant systems that support users in real-world environments. Cooking is one such area that requires constant attention, time management, and decision-making and information based on both user preferences and environmental conditions. Chef Bono is designed as an AI-powered, real-time, smart cooking assistant that aims to make the home cooking experience enjoyable and fun by providing guidance, personalized recipe suggestions, and interactive feedback through voice and image technologies.

Chef Bono integrates speech recognition, natural language processing, local data persistence, and computer vision into a single desktop application. By combining these technologies, the system can understand user commands, observe and analyze the kitchen environment, retrieve and filter recipes, and provide step-by-step cooking assistance. Prioritizing usability, privacy, and modularity, the system is suitable for single-user, local deployment scenarios.

This High-Level Design Report presents the architectural structure of the Chef Bono project. It describes the current and proposed software architectures, subsystem responsibilities, data management strategies, security considerations, and operational boundaries. This document serves as a technical reference for developers and stakeholders involved in the system's design,

1.1 Purpose of the System

Chef Bono's primary goal is to assist users throughout the entire cooking process by providing intelligent and context-aware guidance. The system enables users to interact via voice commands and responds with both visual and auditory outputs. By analyzing images captured in the kitchen environment and interpreting verbal commands, Chef Bono can suggest appropriate recipes, guide users through preparation steps, and adapt its behavior to user preferences and environmental conditions.

Additionally, the system aims to support dietary needs and nutritional awareness by categorizing recipes based on dietary restrictions and estimated nutritional values. This allows users to make informed decisions while cooking without requiring extensive manual research or advance planning.

1.2 Design Goals

The design of Chef Bono is guided by the following key objectives:

- **Modularity:** Ensure that system components are loosely coupled and independently maintainable
- **Extensibility:** Allow future integration of new AI models, data sources, or interaction mechanisms
- **Usability:** Provide an intuitive, voice-driven user experience with minimal manual input
- **Privacy:** Avoid permanent storage of sensitive audio and visual data
- **Performance:** Deliver timely responses suitable for real-world cooking scenarios

1.3 Definitions, Acronyms, and Abbreviations

The following terms and abbreviations are used throughout this document:

- **AI (Artificial Intelligence):** Computational techniques that enable intelligent behavior
- **STT (Speech-to-Text):** Conversion of spoken language into text
- **TTS (Text-to-Speech):** Conversion of text into spoken audio
- **GUI (Graphical User Interface):** Visual interface for user interaction
- **SQLite:** Lightweight, file-based relational database system
- **API (Application Programming Interface):** Interface for communication between software components

1.4 Overview

Chef Bono is an AI-powered cooking assistant designed to give commands and provide support to users in a home kitchen environment through voice interaction, visual processing, and intelligent decision-making. The system observes the kitchen environment, receives and interprets user commands, and provides contextual guidance throughout the cooking process.

The application is implemented in Python as a system consisting of audio, image, AI processing, and data collection components. Voice commands are processed using speech recognition, kitchen images are analyzed using computer vision techniques, and an AI layer coordinates reasoning, suggestions, and response generation. Recipe data and related metadata are stored locally in an SQLite database.

Chef Bono follows a single-user design approach. Audio and image inputs are processed and discarded after use, while structured recipe information is stored permanently. Overall, the system aims to increase cooking efficiency by providing personalized, real-time support based on user preferences and kitchen conditions.

2. Current software architecture (if any)

Overview:

Python modular micro-services style app coordinating image/audio I/O, AI analysis, scraping, and a SQLite recipe DB; entrypoint is main.py.

Core AI Layer:

AIUtils in ai_utils.py wraps Google Gemini (GenAI) for multimodal prompts, token management, and response parsing.

Image Analysis:

Image capture/IO handled by camera.py and analysis helpers in KitchenImageAnalyzer; AIUtils also provides _analyze_kitchen_image.

DB Layer:

Simple SQLite access via Database in db_query_handler.py; data read utilities in read_database.py; persistent store is recipes.db.

Data Ingestion:

Web scraping and batch insertion implemented in row_adder.py which uses AIUtils._add_flags to categorize the recipes before saving, vegan, gluten free, protein score, etc.

Interface & I/O:

GUI front-end in interface.py (Tkinter); TTS and STT stubs/utilities in tts.py and stt.py; audio transcription uses speech_recognition.

Utilities & Scripts:

Misc helpers and the main orchestration in main.py for targeted workflows and testing.

External Dependencies:

Google GenAI SDK, OpenCV, Pillow, BeautifulSoup, requests, speech_recognition, tkinter, sqlite3, python-dotenv

; API key read from environment (GOOGLE_API_KEY).

Data Flow:

User (CLI/GUI) -> capture/image file -> AIUtils (GenAI) -> parsed JSON -> SQL generation -> Database.execute_query / persistence; scraping provides bulk data to DB.

Security & Ops:

Secrets via env vars; network calls to GenAI and external sites; no access control, no migrations or backups for recipes.db.

Extensibility Points / Risks:

Centralized AIUtils is single point for API changes; token/error handling partially implemented; add async handling for blocking IO (GenAI, scraping), validate/sanitize scraped data, and add API-key protection and DB schema migrations.

database_structure.sql holds the structure of the database.

3. Proposed software architecture

3.1 Overview

Purpose: Provide an extensible, modular assistant that captures audio/video, converts speech, invokes AI logic, persists results, and plays audio responses.

Architecture style: Layered + modular micro-services style within a single-process app with clear module boundaries.

Primary data flow: Capture (mic/camera) -> utilize modules of stt.py and camera.py -> understand user intent ai_utils.p -> based on the intent, trigger one of the modules: tts.py (to respond), camera.py(to observe), db_query_handler.py (to generate query to fetch recipes from database) ->output (tts.py, interface.py).

3.2 Subsystem decomposition

UI / management:

-Responsibility: Start/stop sessions, route events, manage user interaction lifecycle.
  Files: main.py, interface.py
  Interface: Event-driven API (start/stop, on Result).
-Capture / Input Adapters:
  Responsibility: Acquire raw data from hardware and provide standardized payloads.

Files: stt.py (audio capture + STT wrapper), camera.py
Data formats: PCM/WAV/mp3 audio, image frames (jpeg,png).

-Speech-to-Text (STT) & Text-to-Speech (TTS):
Responsibility: Convert audio→text and text→audio via pluggable providers.
Files: stt.py, tts.py
Interfaces: Async methods transcribe(audio)->text and speak(text)->audio_stream.

-AI Utilities:
Responsibility: Process text, apply assistant logic, find parameters of sql search queries out of user intent (may include usage of vector space for words), generate sql queries, trigger action based on user intent and provide user-friendly output.
Files: ai_utils.py
Interfaces: ask_ai(text)->response.
generate_sql_query(self, user_input, db_path)

-Data Persistence / DB Layer:
Responsibility: Schema management, queries, inserts, and retrieval for sessions/results.
Files: database_structure.sql, db_query_handler.py, read_database.py, row_adder.py
Storage: Local SQLite (or optional remote DB); SQL schema described in database_structure.sql.

-Integration & Helpers:
Responsibility: Shared utilities, config, async helpers, logging.
Files: ai_utils.py (shared functions), results.txt, README.md

-Testing / Future Requirements:
Responsibility: Test harnesses, planned features and requirements.
Files: future_requirements.txt, tests to be added.

General usage of every subsystem file:

row_adder.py will use api as well all web scraping to fetch recipes which must include preperation and ingredients.Then Then row_adder.py will call add_flag functions of ai_utils.py to categorize the recipes(vegan, gluten-free, protein score, carbs score, etc.) 5 by 5. Then row_adder.py then adds them to the sql lite database.

Our aim is to save around 10k recipes to 100k recipes.

camera.py will capcture images of the user’s kitchen table. Stt.py will capture user audio and delete the audio immediately if the command selected by user is not said initially. Tts.py will get its input from ai_utils.py to read the output out loud to the user.

ai_utils.py will process user command, Kitchen image, trigger program scenarios based of user command or kitchen image, create sql queries based on commands, generate text to be played to user as audio.

db_query_handler.py will get the sql code from ai_utils.py and fetch the data and give it to main.py or ai_utils.py

database_structure.sql will hold the structure of the database.

current_frame.jpg will be the image holder of that moment

recipes.db will hold the sql lite database

.venv folder will hold the external python libraries

.env will hold the api keys

.gitignore will ignore specific files and folders that can't be transferred through git.

3.3 Hardware/software mapping

-Client hardware:
Microphone: Captured by stt.py for audio input.
Speaker / Headphones: Output audio produced by tts.py.
Camera: Managed by camera.py for image/video input.
CPU / Memory: Computing and storing program memory.

-Developer hardware (for future):
GPU with more than 6gb vram to run llms locally and provide cloud service.

-Local software runtime:
Platform: multi-platform development
Runtime: Python 3.x (dependencies in requirements.txt).
DB: SQLite accessed via db_query_handler.py and schema in database_structure.sql.
Files / Formats: WAV/PCM/mp3 audio, JSON for internal message payloads, SQL for persistence.
-Security & Ops mapping:
Secrets: API keys stored in env vars or OS keychain; loader in startup code (main.py).

3.4 Persistent Data Management

Chef Bono implements a lightweight yet robust data persistence strategy using SQLite as its primary storage engine. The system maintains recipe data, user preferences, and cooking session states through a structured relational database approach.

3.4.1 Database Architecture

The core of the persistence layer is built around the recipes.db SQLite database, which stores structured recipe information along with AI-generated metadata. The database schema is designed to support efficient querying while maintaining data integrity through proper typing and constraints.

The Database class in db_query_handler.py provides a simple abstraction layer that handles connection management and query execution. Each operation opens a new connection, executes the query, commits changes if necessary, and properly closes the connection to prevent resource leaks. This pattern ensures thread safety and reduces the risk of database corruption during concurrent operations.

3.4.2 Recipe Storage and Retrieval

Recipes are stored with comprehensive metadata including nutritional scores, dietary flags, and preparation metrics. The RowAdder component in row_adder.py demonstrates the data ingestion pipeline: recipes are scraped from external sources, processed through the AI module to generate classification flags, validated for completeness, and then inserted into the database.

The storage structure supports boolean flags for dietary restrictions (vegetarian, vegan, gluten-free, dairy-free, nut-free), health objectives (weight loss, muscle gain, balanced), and continuous scores for nutritional content (carbohydrates, protein, fat, calories, sugar) as well as preparation characteristics (time and difficulty). This rich metadata enables sophisticated recipe filtering and recommendation capabilities.

3.4.3 Data Validation and Integrity

Before inserting recipes into the database, the system validates that all required fields are present and non-null. The required_fields check in row_adder.py ensures data completeness, preventing partial or corrupted recipe entries from entering the system. This validation layer acts as a quality gate, maintaining the reliability of the recipe database.

3.4.4 Query Generation and Execution

The system leverages AI capabilities to translate natural language queries into SQL statements. When a user requests recipes with specific criteria, the AIUtils.generate_sql_query() method The Database.execute_query() method then executes these queries and returns structured results that can be presented to the user through the TTS interface.

3.4.5 Data Persistence Limitations and Future Enhancements

The current implementation stores recipe data and metadata but does not persist user cooking session history, timer states, or personalized preferences across application restarts. Additionally, there is no mechanism for syncing data across devices or backing up to cloud storage. Future enhancements should include session state persistence, user profile storage, and optional cloud synchronization for cross-device recipe access.

3.5 Access Control and Security

Given that Chef Bono is designed as a single-user desktop application with local data storage, its security model focuses on privacy protection and safe resource access rather than traditional multi-user authentication and authorization mechanisms.

3.5.1 Privacy-First Design

The system implements a privacy-first approach by processing all sensitive data locally. Camera feeds and microphone inputs are analyzed in real-time without being stored permanently. The AI model processes audio-visual streams on-demand and discards the raw data immediately after analysis, ensuring that no recordings of the user's kitchen activities are retained.

3.5.2 API Key Management

The application requires a Google Gemini API key to function, which is stored in a .env file and loaded through the python-dotenv library. This approach keeps sensitive credentials out of the source code and version control systems. The API key is never logged or transmitted except to Google's official Gemini API endpoints over encrypted HTTPS connections.

The ai_utils.py module loads the API key during initialization through os.getenv("GOOGLE_API_KEY"), ensuring that the key remains isolated from other system components. Users must create their own .env file with valid credentials before running the application.

3.5.3 External Service Communication

All communication with external services occurs over encrypted channels. The Gemini API interactions use HTTPS exclusively, and the web scraping functionality in row_adder.py accesses public recipe websites through standard HTTPS requests. The system does not transmit personally identifiable information to external services beyond what is necessary for API functionality.

3.5.4 Local Resource Access Control

The application requires permission to access the camera and microphone, which are controlled by the operating system's permission model. On modern operating systems, users must explicitly grant these permissions when the application first attempts to access these resources. The camera.py module initializes the camera through OpenCV's cv2.VideoCapture(0), triggering the system's permission dialog if not previously granted.

3.5.5 Data Minimization and Retention

In accordance with privacy principles similar to KVKK and GDPR, Chef Bono minimizes data collection and retention. The system does not store:

- Raw camera frames beyond the temporary current_frame.jpg used for immediate analysis
- Audio recordings from voice commands
- Personal user information or cooking history
- Usage analytics or telemetry data

The only persistent data consists of the recipe database and user-created content, both stored locally on the user's device.

3.5.6 Security Limitations and Recommendations

The current implementation lacks several security features recommended for production deployment:

- No encryption of the local database, leaving recipe data unencrypted on disk
- No secure credential storage beyond basic environment variables
- No input sanitization for SQL queries generated by AI, potentially vulnerable to AI-generated SQL injection
- No integrity checking for downloaded recipes or AI-generated metadata

For production use, the system should implement database encryption, secure credential storage using operating system keychains, comprehensive input validation for AI-generated SQL queries, and integrity verification for external data sources.

3.6 Global Software Control

Chef Bono operates as a centralized orchestration system where a main control component coordinates multiple specialized modules to deliver seamless cooking assistance. This architectural pattern enables clear separation of concerns while maintaining cohesive system behavior.

3.6.1 Orchestration Architecture

The system employs a micro services style architecture with the main orchestrator serving as the central coordination point. While the current codebase shows individual modules (ai_utils.py, camera.py, db_query_handler.py, stt.py, interface.py), the intended design places a main controller that manages the lifecycle and interactions of these components.

The orchestrator maintains references to all system modules, handles initialization and cleanup, manages the conversation flow, and routes data between components based on user actions and system state. This centralized control pattern simplifies error handling and ensures consistent behavior across different operational modes.

3.6.2 Operational Mode Management

Chef Bono supports multiple operational modes that determine system behavior:

Limited Surveillance Mode: The camera captures images only when explicitly requested by the user or when the AI needs to verify a specific cooking step. This mode minimizes computational overhead and provides a less intrusive user experience.

Full-Time Surveillance Mode: The camera continuously monitors the cooking process, capturing frames at regular intervals (approximately every 3-10 seconds as specified in NFR1). The AI analyzes these frames proactively to detect potential issues, hazards, or opportunities for guidance.

The mode selection affects multiple system components simultaneously: the camera capture frequency, the AI analysis trigger conditions, and the TTS intervention patterns. The orchestrator ensures that all modules operate consistently within the selected mode.

3.6.3 State Management and Flow Control

The system maintains several state variables that govern its behavior:

- Current recipe and step progression
- Active timers and their remaining durations
- Camera monitoring status (active/inactive)
- User interaction context (waiting for response, processing command, providing guidance)

The orchestrator tracks these states and ensures smooth transitions between different operational phases. For example, when a user requests the next recipe step, the orchestrator coordinates: stopping any active speech output, updating the step counter, retrieving the next instruction from the database, generating the TTS output, monitoring for the completion of the current step, and handling any intervening questions or commands.

3.6.4 Inter-Module Communication

Modules communicate through clearly defined interfaces managed by the orchestrator. The AI module receives text from STT and images from the camera, then generates responses that are routed to TTS for speech output or to the database handler for recipe queries. This explicit routing prevents tight coupling between modules and enables independent testing and development.

The AIUtils class serves as a bridge between raw inputs and structured outputs, translating natural language and visual information into actionable system commands. The orchestrator interprets these outputs and dispatches appropriate actions to other modules.

3.6.5 Process Control Commands

Users can control the cooking process through voice commands that the orchestrator recognizes and handles:

- **Continue**: Proceeds to the next recipe step or resumes a paused process
- **Stop**: Halts the current recipe and cancels active timers
- **Restart**: Returns to the beginning of the current recipe
- **Repeat**: Repeats the current instruction through TTS

The interface.py module provides a graphical representation of these controls, though the primary interaction method is voice-based. The orchestrator ensures that these commands are processed consistently regardless of the current system state.

3.6.6 Error Handling and Recovery

Global error handling is implemented at the orchestrator level to provide graceful degradation when components fail. If the camera becomes unavailable, the system continues operating in voice-only mode. If the API rate limit is exceeded, the AI module implements exponential backoff and retry logic, and the orchestrator queues user requests rather than failing immediately.

The ai_utils.py module demonstrates this pattern with its _fix_exceptions() method that handles API errors like rate limiting (429) and service unavailability (503). The orchestrator extends this approach to handle broader system failures, ensuring that partial functionality remains available even when individual components encounter problems.

3.6.7 Configuration and Extensibility

The system uses environment variables and configuration files to control global parameters such as API endpoints, model selection, and performance thresholds. The settings.json file configures the Python environment and package dependencies, while the .env file stores sensitive credentials.

This configuration-driven approach enables easy customization without code modification and facilitates testing with different parameter combinations. The modular architecture allows new components to be integrated by extending the orchestrator's initialization and routing logic without modifying existing modules.

3.7 Boundary Conditions

Chef Bono operates within specific technical and operational constraints that define the limits of its capabilities and establish expected behavior in edge cases.

3.7.1 Input Processing Limits

Voice Command Recognition: The STT module processes audio through Google's speech recognition service, which requires clear audio and standard pronunciation. The system may fail to recognize commands in noisy kitchen environments with running appliances, when the user speaks too quickly or quietly, or when using heavy accents or non-standard language constructs. The NFR1 specification indicates a 3-10 second processing window for voice commands, meaning real-time conversational flow is not guaranteed.

Image Analysis Boundaries: The camera-based analysis depends on adequate lighting, appropriate camera positioning, and unobstructed views of the cooking surface. The system cannot accurately analyze: ingredients in opaque containers, food items outside the camera's field of view, subtle color changes in poor lighting conditions, or food textures obscured by steam or smoke. The current camera.py implementation captures frames without validation of image quality or content visibility.

Recipe Data Quality: The recipe retrieval system scrapes data from external sources without comprehensive validation. Recipes with incomplete ingredient lists, ambiguous preparation instructions, or non-standard formatting may result in poor cooking guidance. The row_adder.py module filters out recipes missing required fields, but cannot verify the accuracy or quality of the content itself.

3.7.2 API and Network Constraints

Gemini API Rate Limits: The system is constrained by Google Gemini's API rate limits and quotas. The ai_utils.py module implements retry logic for 429 (rate limit exceeded) and 503 (service unavailable) errors, but cannot guarantee uninterrupted service during periods of high usage or when quotas are exhausted. Users may experience delays or temporary loss of AI-assisted features when limits are reached.

Token Context Limits: Each Gemini model has input and output token limits (tracked via input_token_limit, output_token_limit, and context_window_limit in ai_utils.py). Long conversations or complex recipe instructions may exceed these limits, triggering the _shrink_chat_history() method that removes older conversation turns. This can result in loss of context and potentially inconsistent guidance if important earlier information is discarded.

Network Dependency: The system requires active internet connectivity for AI processing, recipe retrieval, and voice recognition. Offline operation is not supported, and network interruptions will cause immediate failure of these features. The local SQLite database provides some offline capability for previously cached recipes, but without AI analysis or voice interaction.

3.7.3 Performance and Resource Boundaries

Processing Latency: NFR1 specifies 3-10 second response times for various operations. These boundaries mean that Chef Bono cannot provide instantaneous feedback, which may be problematic for time-sensitive cooking scenarios like temperature monitoring or rapid technique correction. The system processes approximately 1 frame per 3-10 seconds, limiting its ability to track fast-moving actions or detect momentary hazards.

Memory Constraints: NFR8 establishes a 4GB memory limit during normal operation. Complex recipes with extensive history, multiple concurrent AI analyses, or large numbers of cached images could approach this limit, potentially causing performance degradation or crashes. The current implementation does not include active memory monitoring or cleanup mechanisms.

Hardware Requirements: The system assumes standard laptop hardware without GPU acceleration. This limits the complexity of computer vision models that can be used and constrains real-time analysis capabilities. The camera.py module uses CPU-based OpenCV operations which may struggle with high-resolution video or complex scene analysis.

3.7.4 Operational Environment Constraints

Single-User Design: Chef Bono is designed for single-user operation on a single device. Multiple users cannot interact simultaneously, and there is no support for collaborative cooking scenarios. The system maintains a single conversation context and cannot distinguish between different speakers.

Kitchen Environment Assumptions: The system assumes a relatively controlled environment with standard kitchen lighting, manageable background noise, and typical home cooking scales. Industrial kitchens, outdoor cooking setups, or unusual environmental conditions may exceed the system's adaptability.

Recipe Complexity Limits: While the system can handle standard home cooking recipes, it may struggle with: highly specialized techniques requiring precise timing at subsecond scales, recipes with numerous simultaneous parallel processes, professional-level plating and presentation instructions, or recipes requiring equipment not visually identifiable through the camera.

3.7.5 Data Accuracy and Reliability Boundaries

AI Hallucination Risk: The Gemini-based AI may occasionally generate plausible but incorrect information about cooking techniques, food safety, or recipe modifications. The system has no built-in fact-checking mechanism beyond the AI model's training, and users should exercise judgment when following AI-generated advice.

Cooking Analysis Limitations: The visual analysis of cooking doneness, ingredient preparation quality, and food safety relies on surface-level image analysis. The system cannot: measure internal food temperatures, detect pathogenic contamination, assess ingredient freshness beyond visual cues, or guarantee food safety compliance.

Recipe Classification Accuracy: The AI-generated flags for dietary restrictions and nutritional scores in row_adder.py are estimates based on ingredient lists and may not account for: cross-contamination during preparation, hidden ingredients in processed items, variation in ingredient sources (e.g., whether a product contains traces of allergens), or precise nutritional values requiring laboratory analysis.

3.7.6 Security and Privacy Boundaries

As detailed in section 3.5, the system has limited security controls:

- No multi-user authentication or access control
- No encryption of local data at rest
- Dependence on third-party services (Google) for core functionality
- Limited audit logging or activity tracking
- No formal security testing or vulnerability assessment

3.7.7 Boundary Condition Handling Strategy

The system implements defensive programming practices to handle boundary conditions gracefully:

- Input validation and sanitization where implemented
- Try-catch error handling in critical operations
- Exponential backoff for API failures
- Graceful degradation when optional features fail
- User notifications for known limitations

However, comprehensive boundary testing and formal specification of edge case behaviors remain areas for future development. The current implementation prioritizes core functionality over exhaustive edge case handling.

4. Subsystem Services

Chef Bono is structured as a modular framework made up of various interworking subsystems, with each having a specific collection of services to manage. This division facilitates a clear understanding of tasks, enhances maintenance, and allows for future enhancements to the system without needing major changes to its structure.

4.1 User Interaction Subsystem

The User Interaction Subsystem oversees all interactions between the user and the system. It acts as the main gateway through which users control the application and obtain feedback.

This subsystem enables users to:

- Choose the operation mode (restricted or full-time monitoring)
- Issue commands via voice or graphical interface

- Obtain responses from the system, along with cooking guidelines and confirmations

User entries are captured through the GUI found in interface.py and through vocal input processed by stt.py. The gathered input is sent to the AI processing layer for analysis. The output created by the system is shown visually and optionally through the Text-to-Speech feature.

4.2 Recipe Retrieval and Management Subsystem

The Recipe Retrieval Subsystem is tasked with gathering, organizing, and delivering the recipe information needed by the system.

Recipes are sourced from open-source online resources, employing a mix of API integration and regulated web scraping managed in row_adder.py. The fetched data is processed to confirm that each recipe includes essential information, such as ingredients and preparation instructions.

The subsystem offers:

- Organized recipe storage
- Metadata enhancement through AI-based categorization
- Prompt retrieval of recipes based on user inquiries and preferences

This subsystem guarantees the consistency and dependability of recipe information utilized throughout the system.

4.3 AI Processing Subsystem

The AI Processing Subsystem serves as the main intelligence base of Chef Bono. Its primary implementation is found in ai_utils.py and includes all connections with the external AI service.

This subsystem carries out:

- Understanding of user commands in natural language
- Image assessment of the cooking space
- Categorization of recipes and ingredients
- Formulation of SQL inquiries based on user intentions
- Development of system responses and cooking advice

By consolidating AI interactions in one module, the system reduces the connection between components and permits the AI backend to be changed or updated with little effect on other subsystems.

4.4 Vision and Audio Subsystems

The Vision Subsystem, executed in camera.py, captures photos of the cooking environment and transmits them to the AI module for evaluation. The frequency of image capturing is determined by the chosen operating mode.

The Audio Subsystem consists of:

- Speech-to-Text (STT) for recording user commands
- Text-to-Speech (TTS) for providing auditory feedback

These subsystems function on demand and do not keep raw audio or image information beyond immediate handling, in keeping with the system's privacy-focused design.

4.5 Data Persistence Subsystem

The Data Persistence Subsystem oversees all long-term data storage using SQLite. This is managed through db_query_handler.py and associated database utilities.

The subsystem delivers:

- Long-lasting storage of recipes and metadata
- Execution of queries and retrieval of results
- Data verification during entry

This structure guarantees efficient, reliable data storage without the need for external database services.

5. Glossary

AI (Artificial Intelligence): Computational methods that allow the system to understand user inputs, images, and create intelligent responses.

API: A specified interface facilitating communication between software components and external services.

SQLite: A compact, file-based relational database employed for local data storage.

STT (Speech-to-Text): A technology that translates spoken user commands into written text.

TTS (Text-to-Speech): A technology that changes system responses into audible speech.

Recipe Fetching: The act of obtaining recipe information from outside sources.

Parsing: Transforming unorganized web data into organized formats.

Surveillance Mode: Operating modes that specify how often cameras are used.

Metadata: Attributes generated by AI, including dietary markers and nutritional ratings.

6. References

MediaWiki API Documentation – https://www.mediawiki.org/wiki/API

Wikibooks Cookbook – https://en.wikibooks.org/wiki/Cookbook

Python Software Foundation – https://docs.python.org

SQLite Documentation – https://www.sqlite.org/docs.html

Google Gemini API – https://ai.google.dev/gemini-api/docs