AI-powered Smart Notification System

UmiDo

팀명: UmiZoomi

202255636 Kadyrov Adilet

202255634 Yegizbayev Zholan

지도교수: 조환규

제출일: 2025년 9월 18일

부산대학교

Table of Contents

1 Project Overview

1.1 Introduction and Problem Background

In today's connected world, individuals face increasing challenges when managing multiple responsibilities across healthcare, work, and personal domains. Overlapping schedules, conflicting priorities, and critical tasks such as medication intake or work deadlines are often difficult to coordinate effectively. Traditional task management tools, including simple reminder applications and manual scheduling methods, provide limited support for integrated life management and do not adapt well to individual behavior patterns. As a result, users frequently experience inefficiencies, missed commitments, and decreased overall productivity. Our project addresses these issues by developing an AI-powered Smart Notification System that can intelligently coordinate and prioritize tasks across different domains. Using Natural Language Processing (NLP), the system extracts structured information from unstructured user input, automatically resolves scheduling conflicts, and delivers timely alerts. Additionally, we plan to integrate an AI-based chatbot assistant that enables real-time user interaction, providing an accessible and dynamic management experience. While existing solutions such as medication reminder apps and task management platforms partially address these needs, they often lack holistic integration, cross-domain adaptability, and AI-driven prioritization capabilities. However, we have identified several technical and practical challenges that must be addressed. Accurate interpretation of natural language instructions across multiple domains, effective conflict detection algorithms, and seamless handling of multi-role interactions (e.g., doctor to patient, supervisor to employee) require robust backend logic and advanced AI models. Furthermore, ensuring usability for users with varying degrees of digital literacy—particularly elderly individuals—introduces additional design constraints that we are actively working to address. This project aims to create a practical and intelligent system that enhances users' daily lives by offering reliable, adaptive, and privacy-conscious task management. Our goal extends be-

yond providing simple reminders to delivering a proactive assistant capable of intelligently supporting users in managing the complex realities of modern life.

1.2 Project Objectives

The primary objectives for our project in its current stage are as follows:

1. We aim to conduct a thorough analysis of existing task management and notification systems in both medical and non-medical domains, identifying their strengths, limitations, and gaps. This analysis will help establish a solid foundation for understanding the challenges involved in developing an AI-based cross-domain notification system.

2. We will design and implement a secure authentication system that allows various stakeholders such as doctors, supervisors, patients, and agents to interact with the system according to their assigned roles. This involves careful consideration of privacy requirements and access control mechanisms.

3. Our system will incorporate NLP techniques for extracting structured task information from unstructured text input. This functionality aims to simplify task registration and enhance automation, though we recognize that achieving high accuracy across different languages and writing styles presents significant challenges.

4. We plan to develop a system capable of analyzing overlapping schedules and identifying potential task conflicts across healthcare, work, and personal life domains. This includes both scheduling conflicts and drug interaction conflicts, which require integration with medical databases.

5. The system will include a smart chatbot assistant capable of answering user queries regarding tasks, schedules, and reminders. This feature aims to enhance user engagement and accessibility, though we are still evaluating the best framework for implementation.

6. We will implement a notification system that delivers timely alerts for upcoming tasks, missed critical tasks, and escalates notifications when necessary. This system must balance being informative without becoming intrusive to users.

7. While emphasizing automated task processing, the system will also support manual task addition, ensuring consistency between automated and manual task management workflows.

8. Our team will develop mechanisms to monitor user performance in task completion and generate actionable reports for both users and task assigners. This is particularly important for medical compliance monitoring.

9. The system design will prioritize non-functional requirements including multi-domain adaptability, data privacy compliance, and ease of use for users with varying levels of digital literacy.

2 Final Requirements and Constraints

2.0.1 Initial Requirements

The initial requirements for the project were focused on the development of a reliable and intelligent AI-based smart notification system. These requirements included:

- **Data Requirements:** To ensure realistic and practical performance of the system, we required access to diverse user scenarios and task input data reflecting various roles (e.g., doctors, supervisors, patients). Since real-world deployment was not feasible at this stage, we planned to use simulated task data and user interactions across healthcare, work, and personal domains. These simulated datasets would allow testing task parsing, conflict detection, and notification logic under controlled yet realistic conditions.

- **Functional Requirements:** The system needed to support core functionalities, including secure user authentication with role management, intelligent task input with natural language parsing, conflict detection, chatbot interaction, and a robust notification system. Each function had to operate seamlessly within the integrated system to ensure user engagement and task adherence.

- **System Requirements:** For backend implementation, we selected Node.js with Express for the API server and MongoDB for data storage. Firebase services were required for authentication and push notifications. The frontend would be developed using Flutter to ensure cross-platform compatibility. The system architecture also needed to support scalability for future deployment and integration with external platforms.

- **AI Requirements:** Incorporating basic NLP capabilities was essential for extracting structured task details from unstructured input. Initially, we planned to use

rule-based parsing (e.g., regular expressions, keyword matching) with a view to integrating advanced models like BERT for Korean-language support in later stages. The AI assistant (chatbot) also needed to support predefined query-answering based on stored task data.

- **Usability and Accessibility Requirements:** Given the diversity of target users, including elderly patients, the system required a user-friendly interface with simple navigation and support for both mobile and web access. Basic offline functionality and critical alerts needed to be available even with limited connectivity.

2.0.2 Requirements During Development

As the project progressed, several initial assumptions were revisited, leading to adjustments in the technical and functional requirements. These modifications were made to align the system design with practical development considerations and anticipated deployment scenarios.

- **Data Requirements:** Initially, synthetic datasets were created to develop and test the system's task parsing module—particularly for recognizing medication instructions and usage rules. However, the system is ultimately intended to handle real user-provided data, such as manually entered tasks or doctor-uploaded instructions, requiring attention to data privacy and security. All user data will be stored and managed via Firebase services, eliminating the need for a separate database like MongoDB.

- **Functional Requirements:** The role-based access model was revised to simplify user management into three categories: *receiver*, *administrator*, and *mixed*, where mixed-role users can both assign and receive tasks. A drug interaction conflict detection system is planned, which will initially rely on established Drug-Drug Interaction (DDI) databases, with potential future development of AI-based models to detect new or rare interactions. The chatbot feature remains a core component and is planned as an AI-integrated assistant. We are currently evaluating lightweight rule-based frameworks such as Rasa and spaCy for this purpose. However, if these approaches prove insufficient in terms of flexibility, natural language understanding, or integration ease, we are also considering incorporating a large language model (LLM) such as OpenAI's ChatGPT or open-source models from HuggingFace. This would allow us to

provide a more dynamic and context-aware conversational experience, especially for complex user queries or multilingual support. The AI/NLP module for task parsing is currently in development, with ongoing testing on synthetic data. Instead of periodic static reports, the project aims to implement a real-time dashboard—potentially delivered via a web-based platform—that will visualize task adherence and other analytics by leveraging Firestore's real-time database features. Task history tracking is also included as part of the system's core functionality.

- **System Requirements:** The decision was made to currently unify the backend and database infrastructure under Firebase services. Authentication, data storage (using Firestore), backend logic (via Cloud Functions), and push notifications (via Firebase Cloud Messaging) are fully integrated within the Firebase ecosystem. However, we are also considering the future integration of MongoDB and Node.js to support more advanced server-side logic and flexible data modeling if required. Flutter will be used to develop the application for Android, iOS, and web platforms, enabling a consistent user experience across devices. The system will likely be hosted on Firebase, although previous options such as AWS EC2 or Heroku remain under consideration based on scalability and cost factors. The deployment of AI/NLP components—whether client-side or server-side—remains under evaluation and will be determined based on performance and security considerations.

- **Usability and Accessibility Requirements:** Simplified user interface features are planned to ensure accessibility, particularly for users with limited technical proficiency. An offline mode is under development, allowing basic functionalities like task viewing and input even without an active internet connection. Offline data will be locally stored and synchronized with the server when connectivity is restored. Certain features may remain limited in offline mode to ensure data consistency.

2.0.3 Final Requirements

- **Data Requirements:** Initially, synthetic datasets were created to develop and test the system's task parsing module—particularly for recognizing medication instructions and usage rules. However, the system is ultimately intended to handle real user-provided data, such as manually entered tasks or doctor-uploaded instructions, requiring attention to data privacy and security. All user data will be stored and

managed via Firebase services, eliminating the need for a separate database like MongoDB.

- **Functional Requirements:** The role-based access model was finalized into four categories: Doctor, Manager, Normal, and Mixed. Doctors assign medical tasks such as prescriptions, Managers assign administrative or work-related tasks, Normal users receive and manage assigned tasks, and Mixed users are able to both assign and receive tasks, providing flexibility for supervisors or coordinators. A drug interaction conflict detection system was implemented using a machine learning model trained on an established Drug-Drug Interaction (DDI) database, sourced from the Ministry of Food and Drug Safety (MFDS) to ensure patient safety, with future plans to extend this capability through AI-based models capable of identifying rare or newly emerging interactions. The chatbot assistant was realized as a sophisticated, multi-stage AI-powered module, moving beyond earlier evaluations of lightweight rule-based frameworks such as Rasa and spaCy. The system first processes user input with fast, rule-based matching before leveraging a large language model (LLM) for complex queries, enabling dynamic, context-aware, and multilingual conversational support. The notification system delivers timely alerts for upcoming tasks, missed deadlines, and critical medical events using Firebase Cloud Messaging, designed to remain informative without overwhelming users. Instead of static periodic reports, a real-time dashboard was developed to visualize task adherence, completion rates, and system analytics. For data storage, the system uses Microsoft Azure SQL Database as the primary backend for reliable and scalable data management, while SQLite is employed on the client side to support offline functionality. This ensures that users can view and input tasks without network connectivity, with synchronization to Azure SQL once the device reconnects. Finally, task history tracking was incorporated as a core feature, allowing users to review completed, ongoing, and missed tasks for accountability and long-term monitoring.

- **System Requirements:** The final decision was made to deploy the backend using a Node.js server hosted on Microsoft Azure App Services, providing improved scalability, security, and flexibility for server-side logic. The primary database is a Microsoft Azure SQL Database, which ensures reliable cloud-based data storage and centralized consistency. To support offline functionality, the system also integrates

SQLite as a local database, allowing users to view and create tasks without internet connectivity, with synchronization to Azure SQL once the device reconnects. Authentication and role-based access control are managed securely through the Node.js backend. Push notifications are delivered through Firebase Cloud Messaging (FCM), ensuring timely alerts across platforms. The client application is developed in Flutter, enabling cross-platform deployment for Android, iOS, and web with a consistent user experience. AI/NLP components, including the chatbot and task-parsing modules, are deployed on the backend to ensure secure handling of user data, with flexibility to extend to external APIs or open-source models for advanced natural language understanding. Discovered Constraints and Development Adjustments: Although Firebase Firestore provided convenient integration with authentication, storage, and push notifications, it also presented limitations for our project's long-term requirements. Firestore's schema-less NoSQL model allowed fast prototyping, but it lacked the relational consistency, complex query support, and transactional reliability that our system required, especially for handling medical and work-related data where strict consistency is essential. In contrast, Microsoft Azure SQL Database offers a robust relational model with strong ACID guarantees, ensuring that tasks, schedules, and drug-interaction records remain consistent even under concurrent access. Additionally, Azure SQL integrates seamlessly with Node.js backend services hosted on Microsoft Azure App Services, improving security and compliance by consolidating hosting, database, and authentication under a single enterprise-grade environment. While Firebase's pricing model was attractive during early development, Azure SQL provides more predictable scaling for structured data and better alignment with enterprise deployment scenarios. To support offline functionality, we combined Azure SQL with SQLite on the client side, enabling local task storage and synchronization once network connectivity is restored.

- **Usability and Accessibility Requirements:** Simplified user interface features are planned to ensure accessibility, particularly for users with limited technical proficiency. An offline mode is under development, allowing basic functionalities like task viewing and input even without an active internet connection. Offline data will be locally stored and synchronized with the server when connectivity is restored. Certain features may remain limited in offline mode to ensure data consistency.

2.1 Realistic Constraints and Countermeasures

2.1.1 Initial Constraints

At the beginning of the project, several realistic constraints were identified based on the system's intended functionality and the anticipated technical environment. These included:

- **Limited Access to Real User Data:** The system's ability to process real user input—especially medical instructions and task data—was initially constrained by the lack of real-world datasets. As a result, synthetic data was used for early development and testing of the AI/NLP modules.
- **Challenges in Multi-Domain Task Management:** Ensuring seamless handling of tasks across different life domains (healthcare, work, personal) posed a risk of creating overly complex conflict detection mechanisms, potentially affecting system performance.
- **Integration Complexity with Multiple Services:** The initial plan to integrate various services (Node.js backend, MongoDB, Firebase) increased the system's complexity and risked introducing inconsistencies in data flow and user management.
- **Varying User Digital Literacy:** Considering that users may include elderly individuals or people with low technical proficiency, there was a concern about usability and the need for a simple, intuitive user interface.
- **Network Dependency and Offline Access:** As the system was initially planned to be cloud-based, ensuring uninterrupted functionality in environments with unstable or no internet connection was identified as a significant challenge.

2.1.2 Countermeasures for Initial Constraints

- Synthetic datasets were generated to support early development of AI parsing logic, with plans to gradually incorporate anonymized real user data after deployment.
- A modular design for conflict detection was adopted, allowing flexible handling of domain-specific constraints and reducing system complexity.

- The backend architecture was simplified by transitioning fully to Firebase services, reducing integration complexity while ensuring consistency in user management, data storage, and notification handling.
- The user interface was designed with simplification in mind, incorporating clear navigation, user guidance, and support for varying levels of digital literacy.
- Offline mode functionality was planned, with features allowing task viewing and input during offline periods, and automatic synchronization when the device reconnects.

2.1.3 Constraints During Development

As the project progressed, new constraints emerged, some stemming from practical development considerations:

- **Firebase Usage Limits and Pricing:** While Firebase's free plan suffices for early development, scaling to handle real users may require a transition to a pay-as-you-go model, necessitating cost management planning.
- **Uncertainty in AI/NLP Deployment Model:** Deciding between client-side and server-side AI/NLP deployment presents trade-offs between performance, data privacy, and system resource utilization. This choice remains under evaluation.
- **Database Schema Flexibility:** The evolving nature of feature implementation has required continuous adjustments to the Firestore database schema, emphasizing the need for flexible design and rigorous testing.
- **Chatbot Framework Selection:** Multiple chatbot frameworks (e.g., Rasa, spaCy) are under evaluation to find the best fit for the system's requirements, leading to potential integration and maintenance challenges.
- **User Experience with Offline Features:** Ensuring a seamless offline experience without compromising data consistency remains a technical challenge, particularly when syncing changes after reconnection.

2.1.4 Countermeasures for Constraints During Development

- Monitor Firebase usage closely during development and conduct scaling tests to anticipate potential costs and optimize system resource usage.

- Conduct comparative evaluations of AI/NLP deployment options to select the most suitable model based on system constraints, user privacy considerations, and performance.
- Apply iterative schema design practices with regular database reviews to minimize disruption from structural changes.
- Test multiple chatbot solutions with realistic use cases to determine the most compatible framework, focusing on ease of integration and long-term maintenance.
- Implement robust local data caching strategies and conflict resolution mechanisms to ensure reliable offline operation and smooth synchronization.

2.1.5 Final Constraints and Adjustments

As the system reached its final stage, several realistic constraints were confirmed as enduring challenges that could not be completely eliminated, though mitigation strategies were applied:

- **Limited Access to Real User Data:** The system continued to rely on synthetic datasets for testing AI/NLP modules, as the use of sensitive medical and scheduling data required privacy safeguards. This limited the ability to validate performance on real-world inputs during development.
- **Schema Rigidity in SQL:** The relational model provides consistency but makes rapid feature iteration more difficult. Adding new features often requires schema adjustments and migration planning, which may slow down development.
- **Offline Synchronization Complexity:** Although SQLite enables offline access, synchronization conflicts remain a technical constraint. In cases where multiple users edit the same record offline, perfect conflict-free merging is not always possible, and resolution strategies must be applied.
- **Chatbot Framework Selection:** During development, multiple frameworks such as Rasa and spaCy were evaluated. However, the final system adopts a hybrid design: lightweight rule-based and fuzzy matching for common queries, with a Gemini API LLM fallback for complex or ambiguous requests. This provides both efficiency and context-aware flexibility..

- **User Accessibility and Digital Literacy:** Despite efforts to simplify the UI, supporting elderly or low-tech users remains a constraint, as advanced features like chatbot-based input may still require additional training or onboarding.

2.1.6 Countermeasures for Final Constraints

To address the enduring constraints identified in the final stage of development, the following countermeasures were applied:

- **Limited Access to Real User Data:** Synthetic datasets were refined to better simulate realistic user behavior. In addition, plans were made to gradually incorporate anonymized and ethically sourced real user data after deployment to improve system accuracy while maintaining privacy.

- **Schema Rigidity in SQL:** A modular schema design was adopted in Microsoft Azure SQL, supported by version-controlled migration scripts. This approach reduces disruption during updates and makes it easier to extend the system with new features.

- **Offline Synchronization Complexity:** A combination of UUID-based identifiers and timestamp-based versioning was implemented to support offline operation. UUIDs ensure that tasks created on different devices remain globally unique, even without connectivity, eliminating the risk of ID collisions. For updates to existing tasks, timestamp-based conflict resolution is applied so that the most recent change is preserved. In cases where concurrent edits cannot be safely resolved automatically, users are notified and prompted to confirm the correct version manually.

- **AI/NLP Deployment Trade-offs:** A hybrid chatbot approach was adopted, where lightweight preprocessing with keyword extraction and fuzzy matching is used to quickly handle routine queries, while more complex or ambiguous inputs are routed to the Gemini API LLM for context-aware responses. This balances latency and efficiency for common cases with the flexibility and accuracy of advanced NLP, while keeping sensitive data processing secure on the backend..

- **User Accessibility and Digital Literacy:** The user interface was refined with simplified navigation, clear task categories, and consistent design patterns. The chatbot was enhanced with guided prompts, natural error-handling, and multilingual support to improve accessibility for elderly and less tech-savvy users.

3 Team Member Progress

Table 3.1: Team Progress by Members

Name	Final Contributions
Kadyrov Adilet	**AI/NLP Module Development** • **Data Curation & Augmentation**: Curated and augmented a diverse dataset from prescriptions to improve model generalization. • **Neural Network Parsing**: Fine-tuned a **transformer-based model** for advanced medication and instruction parsing. • **Conversational AI Fine-Tuning**: Developed a conversational AI system by fine-tuning a custom language model for context-aware responses. • **Evaluation & MLOps**: Established a rigorous evaluation framework and MLOps pipeline for automated model retraining and performance monitoring. **Flutter UI Development** • Designed UI structure and navigation flow. • Developed core Flutter UI components for mobile and web. **Documentation and Reporting** • Wrote and compiled project documentation, requirements, and design reports.

Yegizbayev Zholan	**Firebase Integration & Database Development** • Integrated Firebase Cloud Messaging for notifications. • Implemented secure user authentication and role-based access control. **Backend Logic with Cloud Functions** • Planned server-side/offline mode data synchronization logic. • Deployed the final backend application to Microsoft Azure App Services. **Offline Mode Implementation** • Designed offline mode strategy with local data caching. • Started implementing data sync with Firestore upon reconnection.
Both Members	**System Architecture Design** • Designed and implemented the relational database schema on Microsoft Azure SQL Database • Developed the complete backend server using Node.js and Express.js. • Defined system structure using Firebase backend and Flutter frontend. • Evaluated AI/NLP deployment strategy options. **Feature Testing** • Tested authentication, task input flow, and Firebase service integration.

4 Design

4.1 Design Details and Changes

4.1.1 System Design Approach

During the development process, we made several key design decisions to better align with practical development needs and system scalability requirements. Initially, we planned to use a Node.js and MongoDB-based server architecture. However, to simplify early development and ensure seamless integration of core services, we finalized the system architecture using a Node.js backend hosted on Microsoft Azure. The backend handles business logic and user authentication, communicating with a Google Cloud Services and Microsoft Azure SQL Database for all data operations. As the project continues to evolve, we are considering the possibility of extending the backend with additional MongoDB and Node.js components to support more complex server-side logic and increase system flexibility. Firebase's compatibility with external services makes such integration feasible if future system demands or deployment scenarios require it. Additionally, while our original design focused on a mobile-first approach using Flutter for Android and iOS applications, we decided to extend this to include web platforms as well. Flutter's cross-platform capabilities made it feasible to maintain a consistent user interface across devices, with platform-specific integrations being used only when necessary for optimal performance. Another significant addition to our system architecture is the development of an administrative web portal. This platform serves as a management dashboard where administrators can assign tasks, monitor user adherence, and review analytics through real-time dashboards connected to the Firestore database. This addition reflects our evolving understanding of stakeholder needs, particularly for institutional users such as hospitals or care organizations.

4.1.2 Data Handling and Storage

Our system's data handling strategy is built on a robust relational database hosted on Microsoft Azure SQL. Data is structured into tables with defined schemas and relationships to ensure data integrity and consistency, which is crucial for managing medical and work-related tasks. This structure supports efficient querying and role-based data access control. To support offline functionality, we implemented a local storage mechanism on the client-side using the sqflite database. This local database stores critical user data and tasks temporarily when the device is offline and synchronizes with Azure SQL when connectivity is restored. Given the potential handling of Personally Identifiable Information (PII) and Protected Health Information (PHI), particularly under the Korean Personal Information Protection Act (PIPA), our security is handled through the backend and the inherent security features of the Azure platform

Example Data Structures and Storage

To clarify the data handling approach, the following table details the schema for the primary ‘tasks‘ table within the Azure SQL database. This relational structure ensures that all task-related information is stored consistently.

Column Name	**Description**
`task_uuid`	Primary key for the task, a unique identifier for offline synchronization.
`sender_uuid`	Foreign key for the user who assigned the task.
`assignee_uuid`	Foreign key for the user who must complete the task.
`name`	The title or name of the task.
`description`	A detailed description of the task.
`status`	The current status (e.g., 'pending', 'completed').
`priority`	The priority level of the task (e.g., 1 for high).
`created_at`	Timestamp when the task was created.
`updated_at`	Timestamp of the last modification.
`valid_until`	The due date for the task.
`start_date`	The start date for the task.

Table 4.1: Schema for the `tasks` table in Azure SQL

Additionally, the medication list was obtained from the official database of the Ministry of Food and Drug Safety in Korea[1], ensuring medical accuracy and compliance with national standards.

[1]Official data source provided by the Ministry of Food and Drug Safety, Republic of Korea.

id	description	created_at	valid_at
1	U2FsdGVkX1+J7WV9K9Y+3Q==	2025-07-14 09:00:00	2025-07-21 09:00:00
2	QWxkK09tdUxkRFl0aE4yZzA==	2025-07-14 10:00:00	2025-07-16 17:00:00
3	Q2lwQW53MjMxY2dSZGFjQjY=	2025-07-14 08:30:00	2025-07-28 13:00:00

Table 4.2: Working Tasks with Encrypted Descriptions

4.1.3 Frontend and User Interface Design

Our frontend is designed to deliver a consistent experience across mobile and web platforms using Flutter's cross-platform capabilities. Although some platform-specific adjustments are necessary for optimal performance, the core user interface remains unified.

For task input, we designed a simplified form that incorporates intuitive components like time pickers and medication suggestion fields. The system performs immediate validation checks for task conflicts, such as overlapping times or duplicate entries. Users are alerted with clear conflict messages and prompted to adjust or confirm their inputs.

The system now includes a fully integrated drug-drug interaction (DDI) detection feature. This functionality actively analyzes medication combinations and provides real-time warnings about potentially harmful conflicts. By flagging these issues before a task is saved, the system directly supports medication safety and enhances user adherence.

4.1.4 AI/NLP System Design

Drug–Drug Interaction (DDI) Module

Abstract The Drug–Drug Interaction (DDI) Module was designed to overcome the limitations of relying on commercial APIs such as DrugBank, which restricted access. To ensure independence, reliability, and local relevance, we built a proprietary dataset beginning with official Korean drug data from the Ministry of Food and Drug Safety (MFDS). The pipeline transforms this dataset through normalization, feature engineering, evidence extraction, and machine learning analysis to predict and classify drug interaction risks.

Introduction Drug–Drug Interactions are a major factor in prescription safety. Access restrictions to commercial APIs, such as the rejection of DrugBank access, created the need for an autonomous solution. Our approach was to construct a comprehensive dataset of Korean medications and develop a machine learning pipeline for DDI detection. The resulting module integrates seamlessly with the Parser and Chatbot modules to provide reliable, locally hosted, and explainable interaction checking.

System Architecture and Workflow The DDI pipeline consists of four major phases, starting from normalized medication data and progressing through interaction prediction.

Table 4.3: DDI Pipeline Workflow

Stage	Description
Data Collection & Normalization	Build dataset from MFDS drug list; map Korean/English names and generics; include chemical structures and pharmacological attributes.
Feature Engineering	Generate all drug pairs; compute similarity scores, biological targets, and shared side effects.
Signal Detection & Labeling	Extract potential interactions from biomedical literature, co-prescription patterns, and adverse event reports (FAERS).
Model Training & Analysis	Train ML models to classify interaction likelihood and severity; interpret feature importance.

Data Collection and Normalization The foundation of the module is a self-constructed dataset derived from the MFDS. Each medication record was standardized to unify identifiers and enable downstream analysis.

Table 4.4: Normalization Dimensions

Attribute	Example (Korean)	Example (English)	Purpose
Brand Name	타이레놀정	Tylenol Tablet	Market-level identification
Generic Name (Korean)	아세트아미노펜	–	Local generic standardization
Generic Name (English)	–	Acetaminophen	Global mapping
Chemical Structure	C8H9NO2	C8H9NO2	Enables chemical similarity

Feature Engineering Normalized data was expanded into structured feature sets by generating all possible drug–drug pairs. For each pair, multiple quantitative and categorical features were computed.

Signal Detection and Labeling Potential interactions were identified through a combination of textual and statistical evidence. Automated processes extracted signals, which were then curated for training purposes.

Table 4.5: Key Feature Categories

Feature Category	**Description**	**Example**
Chemical Similarity	Overlap of structural fingerprints	Tanimoto coefficient
Biological Targets	Shared proteins, enzymes, or receptors	CYP3A4 inhibition
Side Effect Overlap	Common adverse reactions	Nausea, hepatotoxicity
Co-Prescription Frequency	Frequency of pair in reporting systems	High co-use in FAERS
Literature Mentions	Co-occurrence in biomedical text	Joint mention in PubMed abstracts

Table 4.6: Evidence Sources

Source	**Description**	**Contribution**
PubMed Abstracts	Biomedical literature	Co-mentions of drugs, reported interactions
Korean Medical Journals	Local domain knowledge	Regional prescribing context
FAERS	FDA Adverse Event Reporting System	Statistical co-occurrence of adverse events
Local Pharmacovigilance Reports	MFDS monitoring	Nationally relevant signals

Model Training and Analysis The structured dataset of labeled drug pairs was used to train multiple machine learning models. Accuracy and interpretability were considered equally important.

Table 4.7: Model Evaluation

Model	**Strengths**	**Weaknesses**	**Accuracy**
Logistic Regression	Simple, interpretable	Limited non-linear capture	81.25%
Random Forest	Robust, handles complex data	Less transparent	89.09%
Neural Networks	Captures complex dependencies	Requires large training set	90.11%
K-Nearest Neighbors (KNN)	Simple, easy to implement, and non-parametric	Sensitive to irrelevant features and data scaling	72.00%
Gradient Boosting	High predictive accuracy	Computationally heavier	96.12%

Output Format and Integration Predictions are provided in JSON format to ensure compatibility with other system components. Outputs include both the predicted likelihood and severity classification, as well as evidence references.

Table 4.8: JSON Schema Specification (DDI Output)

JSON Path	Type	Description	Required
drug1_id	int	The unique identifier for the first drug, which corresponds to a record in the medications list.	Yes
drug2_id	int	The unique identifier for the second drug, which corresponds to a record in the medications list.	Yes
interaction_score	float	Predicted probability (0–1)	Yes
level	string	Classified severity: Minor, Moderate, Severe	Yes

Limitations and Future Work The current dataset is limited by the coverage of official MFDS drug lists and available evidence sources. Rare drugs and newly introduced medications may initially lack interaction records. Future work will expand evidence integration to include electronic health record (EHR) co-prescription data, explore transformer-based biomedical models for literature mining, and improve multilingual mapping for global applicability.

Design Decisions Summary

Table 4.9: Key Design Decisions

Component	Choice	Rationale
Data Source	MFDS official database	Reliable, up-to-date, Korean-focused
Normalization	Korean + English names, generics, structures	Enables bilingual and chemical-level mapping
Feature Set	Similarity, targets, side effects, literature, co-prescription	Multi-dimensional risk signals
Modeling	Gradient Boosting + Random Forest	Accuracy + robustness

Conclusion The DDI Module was created in response to the rejection of DrugBank API access, ensuring full independence and data sovereignty. By building a proprietary dataset from the MFDS, enriching it with chemical and pharmacological attributes, and

applying machine learning, the system provides reliable predictions of interaction risks. Its modular design supports integration with the Parser and Chatbot modules, enabling safe, explainable, and scalable prescription management.

Prescription Parser Module

Abstract The Prescription Parser Module is a multi-stage pipeline designed to extract structured data from prescription images. It converts raw images into validated JSON outputs, accurately identifying patient information, medications, dosages, frequency, duration, and instructions. The system leverages image encoding, advanced information extraction, fuzzy matching, and NLP-based instruction analysis to handle the complexities of handwritten and printed Korean prescriptions.

Introduction Traditional OCR approaches often fail to handle the variability and nuances of prescription text. The parser addresses this challenge by integrating specialized preprocessing, a domain-specific information extractor, and a fine-tuned NLP model. This enables high-accuracy extraction while producing structured, machine-readable outputs.

System Architecture and Workflow The parser pipeline is organized into four sequential stages:

Table 4.10: Prescription Parser Workflow

Stage	**Description**
Image Encoding	Convert uploaded prescription images to Base64 text for secure transmission to the information extraction API.
Information Extraction	Use the Upstage Information Extractor with a JSON Schema to identify entities like patient.name, medications.name, dosage, frequency, and instructions.
Medication Matching	Apply RapidFuzz fuzzy string matching against a normalized drug list. Thresholds: 80 for specific drug names, 75 for generic names.
Instruction Analysis	Convert raw instruction text (e.g., "아침, 저녁식후 복용") into structured JSON using a fine-tuned NLP model.

Information Extraction Accuracy This section evaluates the performance of different information extraction services on Korean prescriptions.

Table 4.11: Information Extractor Performance

Service	**Accuracy on Korean Prescriptions**
Upstage Information Extractor	95.2%
Microsoft Azure AI Document Intelligence	80.5%
Amazon Textract	75.1%
Langextract	54.2%

Instruction Analysis Models This table compares the accuracy of various NLP models evaluated for parsing prescription instructions into structured data. Based on its superior performance, the **Fine-tuned KoBART** model was selected for the final implementation.

Table 4.12: NLP Models for Instruction Analysis

Model	**Description**	**Accuracy**
Fine-tuned KoBART	Sequence-to-sequence transformer; converts text instructions to JSON	94.48%
KoBERT	Contextual encoder; suitable for NER tasks	83.25%
CNN	Captures local patterns in text; less effective on long dependencies	65.77%
CRF	Statistical sequence labeling; identifies entities like MEAL, RELATION	55.94%

Data Generation and NLP Training To train the instruction analysis model, a large synthetic dataset of Korean prescription instructions was generated. Template-based instructions were augmented with linguistic variability using the Gemini API, producing diverse, natural-sounding instructions paired with structured JSON labels. This enabled robust model performance across both canonical and real-world instruction formats.

JSON Output Schema The following table outlines the structured JSON output format generated by the parser.

Table 4.13: Prescription Parser JSON Schema

JSON Path	Type	Description	Required
patient	array	List of patient objects	Yes
medications	array	List of medication objects	Yes
id	number	Unique identifier for the medication	Yes
extractor_name	string	Name of the medication as extracted from UpStage	Yes
dosage	number	Dosage (may include decimals)	Yes
frequency	integer	Times medication is taken per day	Yes
duration	integer	Number of days to take medication	Yes
instructions	string	Raw instruction text	Yes
instruction_details	object	Parsed details from the raw instructions	No
matched_name	string	Name of the medication after fuzzy matching	No

Conclusion The Prescription Parser Module combines advanced information extraction, fuzzy matching, and NLP-based instruction analysis to transform raw prescription images into structured, validated JSON outputs. Its modular and multi-stage design ensures high accuracy, robustness, and seamless integration with the DDI and Chatbot modules.

Chatbot Module

Abstract The Chatbot Module provides an intelligent conversational interface for interacting with the AI/NLP system. It is designed as a multi-tiered pipeline that efficiently handles routine queries while maintaining robust capabilities for complex or ambiguous requests. This architecture allows the chatbot to provide accurate, context-aware responses in a variety of scenarios, including prescription guidance, medication instructions, and interaction inquiries.

Introduction A conversational interface is essential for enabling users to access prescription and interaction information without requiring technical expertise. The Chatbot Module integrates rule-based matching, fallback strategies, and a large language model (LLM) API to balance efficiency with high-level reasoning. It serves as the user-facing endpoint of the AI/NLP system, complementing the Parser and DDI modules.

System Architecture and Workflow The chatbot pipeline is organized into five sequential stages:

Table 4.14: Chatbot Pipeline Workflow

Stage	Description
User Input Processing	Capture and preprocess user input for linguistic analysis.
Primary Matching	Quickly match input to predefined instructions using keyword and pattern recognition.
Secondary Matching (Fallback)	Apply a more flexible matching algorithm to handle slight variations in phrasing.
Gemini API (Fallback of Fallback)	Utilize a large language model to handle ambiguous or unmatched inputs, generating context-aware responses.
Query Execution & Response Generation	Execute SQL queries to retrieve required information and formulate a human-like response for the user.

User Input Processing The first stage captures raw text input from the user and preprocesses it for downstream analysis. This includes tokenization, normalization, and removal of extraneous characters, ensuring consistent input for the matching algorithms.

Primary and Secondary Matching This two-tiered strategy is designed to balance speed with robustness. The primary stage employs **exact keyword-based matching** for high-confidence resolution. This approach is exceptionally fast and efficient for common queries, but its brittleness means it fails on minor typos or rephrasing. If no match is found, the system proceeds to the secondary stage, which leverages **fuzzy string matching algorithms, such as Levenshtein distance**. This method calculates a similarity score, allowing it to accommodate variations in phrasing, typos, and minor spelling differences. While computationally more intensive, this fallback mechanism significantly improves the user experience and ensures a high success rate even with imperfect user input. The effectiveness of this pipeline relies on a carefully tuned similarity threshold to avoid inaccurate matches.

LLM Fallback Integration For inputs that remain unresolved after primary and secondary matching, the Gemini API is invoked. This LLM-based assistant is capable of understanding context, generating human-like responses, and interpreting complex or open-ended queries. This ensures that even unusual or nuanced requests are addressed appropriately.

Query Execution and Response Generation Once the user input has been mapped to a known instruction or resolved via the LLM, the system executes relevant SQL queries to retrieve data from the underlying database. The final response is formatted in natural language, providing a clear and concise answer that integrates information from the Parser and DDI modules as needed.

JSON Output Schema The following table outlines the structured JSON output format generated by the Chatbot.

Table 4.15: Chatbot Response JSON Schema

JSON Path	Type	Description	Required
user_input	string	Raw text input provided by the user	Yes
matched_instruction	string	Instruction matched via primary or secondary matching	Cond.
llm_response	string	Generated response from Gemini API if fallback was used	Cond.
query_results	array	Data retrieved from database queries	Cond.
response_text	string	Final human-readable response	Yes
processing_path	string	Path taken in the pipeline (primary, secondary, or LLM fallback)	Yes

Conclusion The Chatbot Module combines rule-based matching, flexible fallbacks, and a large language model to provide a conversational interface that is both efficient and intelligent. Its layered architecture ensures rapid responses for common queries while retaining robust understanding and context awareness for complex or ambiguous requests, completing the AI/NLP system's user-facing functionality.

4.1.5 Messaging and Communication System Design

The messaging system is designed to facilitate secure and direct communication between assigned users, such as a doctor and a patient, or a manager and an employee. This feature complements the automated notification system by enabling real-time, human-to-human interaction, which is crucial for discussing complex tasks, clarifying instructions, or addressing concerns.

Purpose and Scope

The primary purpose of the messaging module is to provide a dedicated, secure channel for task-related communication. This prevents the need for external messaging apps and keeps all relevant discussions within the system for better organization and accountability. Key features include:

- **Direct Messaging:** Users can initiate one-on-one chats with others involved in a shared task.
- **Contextual Conversations:** Message threads are tied to specific tasks, allowing users to easily reference task details while communicating.
- **Attachment Support:** Users can send images or documents, such as a new prescription photo or a work-related file.

User Interface and Experience

The messaging interface is designed for simplicity and ease of use, with a familiar chat-like layout. When a user opens a task, an integrated chat icon or tab is visible. Tapping it reveals the conversation history for that specific task. This contextual design ensures that all communication is directly relevant to the task at hand, reducing confusion and improving information retrieval.

Integration with Task Management

The messaging system is deeply integrated with the core task management logic. When a message is sent, a soft notification is generated on the recipient's side to alert them of the new communication. These notifications are handled via our push notification infrastructure to ensure timeliness.

4.1.6 Notification System and Conflict Detection

The system's notification logic is built upon Firebase Cloud Messaging (FCM) and Firebase Cloud Functions. Task reminders and critical alerts are sent based on both scheduled timings and event-driven triggers. The conflict detection logic is embedded within the task management flow, where the system checks for overlaps, excessive task loads, and planned drug-drug interaction (DDI) conflicts.

4.1.7 Offline Mode Handling

Recognizing the importance of system availability in low-connectivity environments, offline mode support represents a key feature of our design. The system employs a client-side local database (sqflite) for storing task data and essential information. When a device reconnects to the internet, a synchronization mechanism communicates with our Node.js backend to update the central Azure SQL Database with locally stored data. While core functionalities such as task viewing and entry are supported offline, certain features including real-time notifications and conflict checks requiring server-side logic remain limited until synchronization occurs. We continue to refine this functionality to ensure seamless user experience and data consistency, particularly addressing challenges that arise when offline changes conflict with server-side updates.

4.1.8 Key Design Changes from Initial Plan

Several significant architectural and design changes were made throughout the project's lifecycle to meet the final requirements:

- **Transition to a Relational Database:** The backend data store was transitioned from Firebase Firestore to a Microsoft Azure SQL Database. This change was crucial for implementing robust, transaction-safe operations and enforcing strict data consistency, which are essential for the application's reliability.

- **Finalized AI and Chatbot Integration:** The chatbot's development began with initial plans focused on implementing a rule-based matching logic. This foundational work has since evolved into a robust, multi-stage processing pipeline. The system now first captures and pre-processes user input, then attempts a Primary Matching to quickly match to predefined instructions. If that fails, a Secondary Matching fallback attempts a more flexible match. For any remaining ambiguous or unmatched inputs, the system leverages the Gemini API as a final, powerful fallback. The process concludes with Query Execution & Response Generation, where the system runs a SQL query and formulates a final response.

- **Expanded Frontend Scope:** The application scope was expanded from a mobile-only app to a unified mobile and web platform using Flutter, which also included the development of a dedicated web interface for administrators.

- **Robust Offline Mode Implementation:** A sophisticated offline mode was implemented using a local SQLite database, with a synchronization mechanism that communicates with the Node.js backend to ensure data consistency with the central Azure SQL database.

5 Production Deployment and Functionality

5.1 System Deployment Architecture

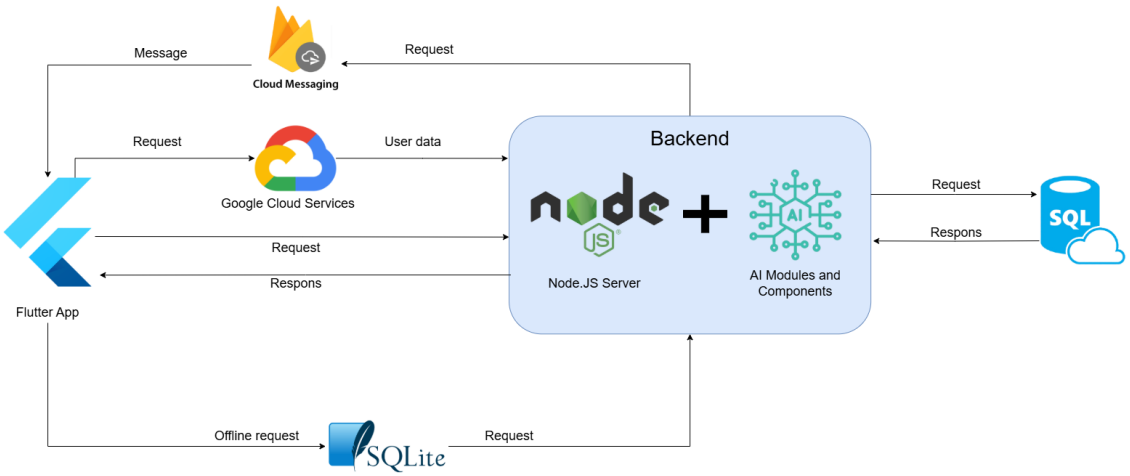

Figure 5.1: Flutter App workflow design

The system's architecture, as illustrated in Figure 5.1, is designed around a centralized backend that communicates with a client application and various cloud services to deliver a robust and responsive user experience. The workflow is composed of several key interactions:

1. **Client-Server Communication:** The primary interaction occurs between the Flutter App (the client) and the Node.js Server (the backend). When a user performs an action, such as creating a new task, the Flutter App sends a Request to the backend's API. The backend processes the request, interacts with the database, and returns a Respons to the app to update the user interface.

2. **Database Interaction:** The backend is responsible for all data persistence. It sends `Requests` to and receives Responses from a cloud-based SQL database, which stores all critical information, including user accounts, tasks, and schedules.

3. **Push Notifications:** To provide timely alerts, the Node.js Server sends a Request to Firebase Cloud Messaging (FCM) when a notification needs to be triggered. FCM then delivers a push Message directly to the user's device via the Flutter App.

4. **External Cloud Services:** The architecture integrates with Google Cloud Services to handle specific functionalities such as user authentication or data handling. The Flutter App can send a `Request` to these services, which then securely provide User data to the backend for processing.

5. **Offline Functionality:** A key feature of the system is its ability to function offline. When the user has no internet connection, the Flutter App makes an Offline request to a local SQLite database on the device, allowing the user to continue viewing and managing tasks. Once connectivity is restored, the local data is synchronized with the backend server.

5.2 Application Functionality and User Interface

6 Literature Review

6.1 Comparison with Existing Systems

To understand the unique value proposition of the UmiDo system, a thorough analysis of existing solutions in the market was conducted. We compared our system against two prominent applications: Medisafe, a dedicated medication reminder and health management app, and Trello, a popular project management and collaboration tool. This comparative analysis highlights the key features where existing solutions fall short and demonstrates how UmiDo's integrated approach addresses these critical gaps.

As shown in the table below, while Medisafe excels in medical reminders, it lacks functionality for broader task management and collaborative work. Conversely, Trello is highly effective for work-related tasks but offers no specialized support for medical needs or real-time health-related checks. Our system, UmiDo, is designed to be a holistic solution that combines the strengths of both domains while introducing advanced AI and real-time conflict detection capabilities.

6.2 Summary of Findings

The comparative analysis confirms that no single existing application provides the comprehensive, integrated functionality of UmiDo. Traditional solutions are specialized for a single domain, forcing users to manage their personal, work, and health-related tasks across multiple disconnected platforms.

Our system's unique value lies in its ability to seamlessly integrate these domains through AI-powered, natural language-based task entry and a sophisticated DDI detection module. This holistic approach not only simplifies the user's daily life but also enhances safety and accountability, particularly in a medical context. The multi-role access control and secure, offline-first design further distinguish UmiDo as a reliable and adaptable

Table 6.1: Comparative Analysis of Task Management and Notification Systems

Feature	**Medisafe**	**Trello**	**UmiDo**
Medical task reminders (pills, prescriptions)	O	X	O
Work/project task management	X	O	O
Drug–Drug Interaction (DDI) safety checking	X	X	O
AI/NLP for natural task input (text, speech, prescription parsing)	X	X	O
Chatbot assistance	X	X	O
Cross-domain task integration (health + work + personal)	X	Partial	O
Offline support with local DB sync	Limited	O	O
Real-time conflict detection (schedule + medication)	X	X	O
Role-based access (doctor, manager, patient, normal)	X	Limited	O
Privacy & security (PIPA/HIPAA compliance focus)	O	Limited	O

solution for the complex demands of modern life.

7 System Implementation

This chapter details the technical implementation of the UmiDo system, covering the technological stack, backend and frontend architecture, and the implementation of its core functionalities.

7.1 Technological Stack

The system was developed using a modern, scalable technology stack chosen to meet the project's requirements for cross-platform availability, real-time communication, and robust data management.

- **Backend:** Node.js with the Express.js framework.
- **Database:** Microsoft Azure SQL Database for primary data storage and SQLite for client-side offline storage.
- **Frontend:** Flutter for cross-platform (iOS, Android, Web) development.
- **Cloud Services:** Microsoft Azure App Services for hosting the backend, and Firebase Cloud Messaging (FCM) for push notifications.
- **AI/NLP:** Integration with a Large Language Model (LLM) via its API for the chatbot functionality.

7.2 Backend Implementation

The backend, built with Node.js, serves as the central hub for all business logic, data processing, and communication between the client and the database.

7.2.1 Server and API

The server was developed using Express.js, a minimal and flexible Node.js web application framework. It exposes a RESTful API for the Flutter client to consume. Key responsibilities of the API include handling user authentication, task management (CRUD operations), and processing chatbot queries.

Code 7.1: Example of an Express.js route for creating a task.

```
// POST /api/tasks - Create a new task
router.post('/tasks', isAuthenticated, async (req, res) => {
        const {assignee_id, name, description, valid_until} = req.body;
        const sender_id = req.user.id;

        try {
                // Insertion of the new task into the Azure SQL DB
                const newTask = await db.createTask({
                        sender_id,
                        assignee_id,
                        name,
                        description,
                        valid_until
                });
                res.status(201).json(newTask);
        } catch (error) {
                res.status(500).send('Error creating task.');
        }
});
```

7.2.2 User Authentication and Security

User authentication is managed by the backend using JSON Web Tokens (JWT). When a user logs in, the server validates their credentials and issues a JWT. This token is then sent with every subsequent API request to authenticate the user and authorize access to resources based on their role (e.g., Doctor, Manager, Normal). Passwords are securely stored in the database using the `bcrypt` hashing algorithm.

7.3 Frontend Implementation

The frontend was developed using Flutter, allowing for a single codebase to target mobile (iOS, Android) and web platforms, ensuring a consistent user experience across all devices.

7.3.1 State Management and API Integration

The Flutter application is structured using a state management solution to separate the UI from the business logic. The app communicates with the backend via HTTP requests to the REST API. Data fetched from the API is parsed into local data models and used to build the UI, providing the user with a dynamic and responsive experience.

Code 7.2: Example of a Flutter function to fetch tasks from the backend API.

```
Future<List<Task>> fetchTasks() async {
  final token = await getAuthToken(); // Retrieve JWT
  final response = await http.get(
  Uri.parse('https://umido-api.azurewebsites.net/api/tasks'),
  headers: {
    'Content-Type': 'application/json',
        'Authorization': 'Bearer $token',
  },
  );

  if (response.statusCode == 200) {
    List jsonResponse = json.decode(response.body);
        return jsonResponse.map((task)=>Task.fromJson(task)).toList();
  } else {
    throw Exception('Failed to load tasks');
  }
}
```

7.4 Core Feature Implementation

7.4.1 AI-Powered Chatbot

The chatbot functionality is implemented by integrating a third-party Large Language Model (LLM). The workflow is as follows:

1. The user types a message in the Flutter app's chat interface.
2. The app sends the message to a dedicated endpoint on the Node.js backend.
3. The backend formats the request and forwards it to the LLM API, including relevant context about the user's tasks.
4. The LLM processes the request and returns a natural language response.
5. The backend sends this response back to the Flutter app, which displays it to the user.

7.4.2 Implementation of the Messaging Module

The messaging module is a key component of our system's real-time communication strategy, relying on a combination of backend services and a real-time database.

Backend and Real-time Data

The messaging functionality is built on a real-time data architecture to ensure low-latency message delivery. The primary data for message storage is handled by our Microsoft Azure SQL Database. We use a dedicated table for messages, which includes the sender's ID, recipient's ID, message content, timestamp, and a foreign key to the associated task.

Code 7.3: Example of a message object data structure.

```
// Message Data Structure
const messageSchema = {
        message_id: 'UUID',
        task_id: 'UUID',
        sender_id: 'UUID',
        recipient_id: 'UUID',
        content: 'TEXT',
        timestamp: 'DATETIME',
        attachment_url: 'TEXT'
};
```

When a message is sent from the client, a request is made to a dedicated API endpoint on our Node.js backend. The server then validates the request, records the message in the database, and initiates the notification process.

Frontend Implementation (Flutter)

The Flutter client uses a reactive programming approach to display messages in real-time. When a user is on the chat screen, the app listens for new messages by polling the backend or using a real-time data listener (if a service like Firestore were to be integrated for this specific feature). This ensures that messages appear instantly for all participants in the conversation, providing a fluid user experience.

Push Notifications via FCM

For new messages, a push notification is sent to the recipient's device to alert them, even if the app is closed. This is implemented via Firebase Cloud Messaging (FCM). When the Node.js backend receives a new message, it sends a request to the FCM API with the

recipient's device token. FCM then delivers the notification to the target device, ensuring that critical messages are never missed.

7.4.3 Offline Mode and Data Synchronization

To ensure the app is functional without an internet connection, an offline-first approach was implemented.

- **Local Storage:** All tasks are stored locally in a **SQLite** database on the user's device. The app reads from and writes to this local database for all primary operations.
- **Synchronization:** When the app detects an internet connection, a background synchronization service is initiated. It checks for any new or modified tasks in the local SQLite database that have not yet been pushed to the server. These changes are then sent to the backend API to be saved in the central Azure SQL Database, ensuring data consistency across devices.

7.5 Task Performance and Interim Results

As of the current reporting period, the project has made substantial progress across multiple core development areas. The collaborative efforts of team members have resulted in both backend and frontend modules approaching MVP (Minimum Viable Product) status, while AI components undergo active testing and refinement.

7.5.1 Final Implemented Modules

AI/NLP Task Parsing:

The AI/NLP component has progressed beyond its initial evaluation phase. A robust dataset has been developed to train and test the models, ensuring high accuracy. The rule-based parsing logic has been refined and integrated with an advanced medication matching system powered by RapidFuzz. Furthermore, the chatbot assistant has been fully developed, implementing a multi-tiered architecture that efficiently handles user queries with predefined logic and uses the Gemini API as a powerful fallback for ambiguous inputs.

Backend and Database Infrastructure:

User authentication with role-based access is fully functional.The final system is powered by a Node.js backend deployed on Microsoft Azure, which handles all business logic, authentication, and API requests. All data is securely stored in a relational Microsoft Azure

SQL Database. Firebase Cloud Messaging has been partially implemented for real-time notifications, with additional serverless functions in development for task validation and syncing.

Frontend Development:

Core UI components using Flutter are built for Android, iOS, and web. The interface supports task input, time scheduling, and visual validation feedback. UI testing for older users and accessibility constraints is underway.

Offline Mode:

The offline mode, using a local SQLite database, is fully functional. A robust synchronization mechanism ensures data consistency between the client and the central Azure SQL database upon reconnection.

Security and Compliance:

For privacy-sensitive data, encryption-at-source is in place, ensuring data is encrypted before it leaves the client device. This approach meets strict PIPA requirements and provides end-to-end protection for all confidential information.

7.5.2 Final Evaluation and Results.

System Stability:

End-to-end testing of the fully integrated system was completed. The application demonstrated high stability under simulated user load.

Usability Testing:

Usability testing was conducted with a group of representative users. Feedback from these sessions was overwhelmingly positive, highlighting the application's intuitive design and ease of use. The insights gathered led to direct improvements in the user interface, such as adjustments to font sizes and button placements to enhance accessibility.

Conflict Detection:

The Drug-Drug Interaction (DDI) conflict detection system was fully implemented and rigorously tested. It proved to be highly effective, reliably identifying potential medication conflicts based on our established test cases.

Team Coordination:

Members have effectively divided tasks by technical specialty, maintaining regular version control and documentation updates.

8 Development Plan

8.1 Detailed Development plan

구분	5 월				6 월				7 월				8 월				9 월			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Define requirements, finalize architecture, set up dev env	■	■	■																	
Design Firestore schema, implement Firebase Authentication			■	■	■	■														
Flutter UI screens + navigation					■	■	■	■	■											
AI-based NLP task parsing							■	■	■	■	■	■	■							
Conflict detection													■	■	■					
Chatbot integration													■	■	■	■				
Firebase Cloud Functions & App Management													■	■	■	■				
System testing, error correction, UI refinements																■	■	■		
Final documentation, prepare for final demo and submission																■	■	■	■	

Figure 8.1: Development Plan Table