OntarioTech Engineering & Applied Science

Faculty of Engineering and Applied Science

Department of Electrical, computer and Software Engineering

SOFE 3650 – Software Design & Architecture

# Deliverable 2: ADD Iteration 1 & 2

Due Date: November 19th, 2025

Course Instructors: Dr Hani Sami

CRN: 45740 - Group #8

| Student Name | Student Number |
|---|---|
| Christopher Lui | 100912564 |
| Geraline Chavez | 100890130 |
| Nicholas Furtado | 100908880 |

## 1. The Design Process

The design process for the AI-Powered Digital Assistant Platform (AIDAP) is guided by the Attribute-Driven Design (ADD) method. This process marks the transition from requirements analysis to architectural reasoning, where the system's functional and quality drivers are translated into design decisions. The goal of this phase is to establish an initial architectural vision that captures the most influential drivers identified in Phase 1, particularly performance, scalability, and maintainability, and to outline how the architecture will evolve through successive iterations. Subsequent steps will progressively refine this high-level structure into concrete components, interactions, and deployment views.

### 1.1 ADD Step 1: Review Inputs

The purpose of this step is to review the main design inputs from Phase 1, which serve as the foundation for the first architectural iteration. These inputs include the functional and quality requirements, constraints, and architectural concerns that will guide all subsequent design decisions.

**Design Inputs Summarys**

| Category | Details |
|---|---|
| **Design Purpose** | To design a cloud-based AI assistant platform (AIDAP) that centralizes academic and administrative information into a single intelligent interface for students, lecturers, and administrators. The goal is to enhance accessibility, ensure seamless integration between institutional systems (LMS, email, and calendar), and deliver rapid, reliable responses through an adaptive and scalable architecture. |
| **Primary Functional Requirements** | UC-1 – Merge and Display Multiple Academic Calendars: Enables integration of different university systems (LMS, email, calendar). This requirement drives the need for an Integration Layer and API Gateway to unify data sources in real time. It impacts interoperability and maintainability. |

| | |
|---|---|
| | UC-4 – Perform Server Maintenance with Zero Downtime: Ensures the system remains available while updates or patches are applied, ensuring minimal downtime. This requirement drives the use of redundant microservices, rolling deployments, and container orchestration (Kubernetes), supporting availability and maintainability.<br><br>UC-5 – Handle High Traffic Efficiently: Supports up to 5,000 concurrent users during peak load (e.g., exams). It drives the choice of a cloud-native microservices architecture and load-balancing mechanisms to achieve performance and scalability. |

**Quality Attributes**

| **Scenario ID** | **Quality Attribute** | **Importance to Customer** | **Difficulty of Implementation** | **Reason** |
|---|---|---|---|---|
| **QA-1** | Performance | High | High | Essential for fast responses under 5,000 users; drives the need for efficient Django views and API Gateway optimization. |
| **QA-2** | Availability | High | Medium | Required for continuous service during maintenance; relies on redundancy and failover mechanisms. |
| **QA-3** | Usability | High | Medium | Important for natural user interaction (text/voice); moderate effort via |

|  |  |  |  |  |
|---|---|---|---|---|
|  |  |  |  | interface design and AI training. |
| **QA-4** | Privacy and Security | High | High | Mandatory institutional policy enforcement; complex because of access-control and data-encryption requirements. |
| **QA-5** | Scalability | High | High | Needed for traffic spikes; container orchestration and load-balancing add complexity. |
| **QA-6** | Maintainability | Medium | Low | Facilitated by modular microservices; lower complexity due to Django's MVT structure. |
| **QA-7** | Adaptability | Medium | Medium | Enhances personalization; requires machine-learning feedback loops. |

From this list, QA-1 (Performance), QA-2 (Availability), QA-5 (Scalability), and QA-6 (Maintainability) are selected as *architectural drivers* because they most strongly influence responsiveness, uptime, scalability, and maintainability, which align with AIDAP's cloud-native goals.

|  |  |
|---|---|
| **Constraints** | All of the constraints (CNST-1 to CNST-6) discussed in Phase 1 are included as drivers. |
| **Concerns** | All architectural concerns (CRN-1 to CRN-8) from Phase 1 are included as drivers. |

### 1.2 Iteration 1: Establishing an Overall System Structure

This section presents the results of the activities performed in the first iteration of the ADD process.
The goal of this iteration is to establish an initial system-level structure for the AI-Powered Digital Assistant Platform (AIDAP) based on the most influential architectural drivers identified in the previous step.

#### 1.2.1 Step 2: Establish the Iteration Goal by Selecting Drivers

This first iteration of the Attribute-Driven Design process is guided by the architectural goal of establishing AIDAP's overall system structure.
The focus is on defining a modular, scalable, and fault-tolerant architecture that ensures high performance and continuous availability, while supporting seamless integration between internal and external university systems.

Although this iteration is primarily driven by the architectural concern CRN-5: Maintain a clearly defined and logically structured architecture, the architect must also account for all the quality, constraint, and concern drivers influencing the system's initial design.

The primary design drivers for this iteration are summarized below:

**Architectural Drivers Considered**

| Type | Driver ID | Influence on Design |
|---|---|---|
| **Quality Attribute** | **QA-1 – Performance** | **The system must handle 5,000 concurrent users and respond to queries within 2 seconds. This drives the need for load balancing, efficient API endpoints, and asynchronous request handling.** |
| **Quality Attribute** | **QA-2 – Availability** | **Continuous operation must be maintained during maintenance and system updates. This leads to redundant deployment, container orchestration, and failover support.** |

| | | |
|---|---|---|
| **Quality Attribute** | **QA-5 – Scalability** | **The architecture must dynamically allocate resources during high-traffic periods, requiring containerization (Docker/Kubernetes) and cloud-native infrastructure.** |
| **Quality Attribute** | **QA-6 – Maintainability** | **Modular microservices and isolated deployment pipelines must be implemented to simplify updates and reduce downtime.** |
| **Constraint** | **CNST-1** | **The system must handle 5,000 concurrent users while maintaining a maximum response time of 2 seconds.** |
| **Constraint** | **CNST-6** | **The system must employ a cloud-native architecture for scalability and portability across environments.** |
| **Architectural Concern** | **CRN-5** | **The architecture must remain logically structured and modular to support long-term scalability and ease of maintenance.** |

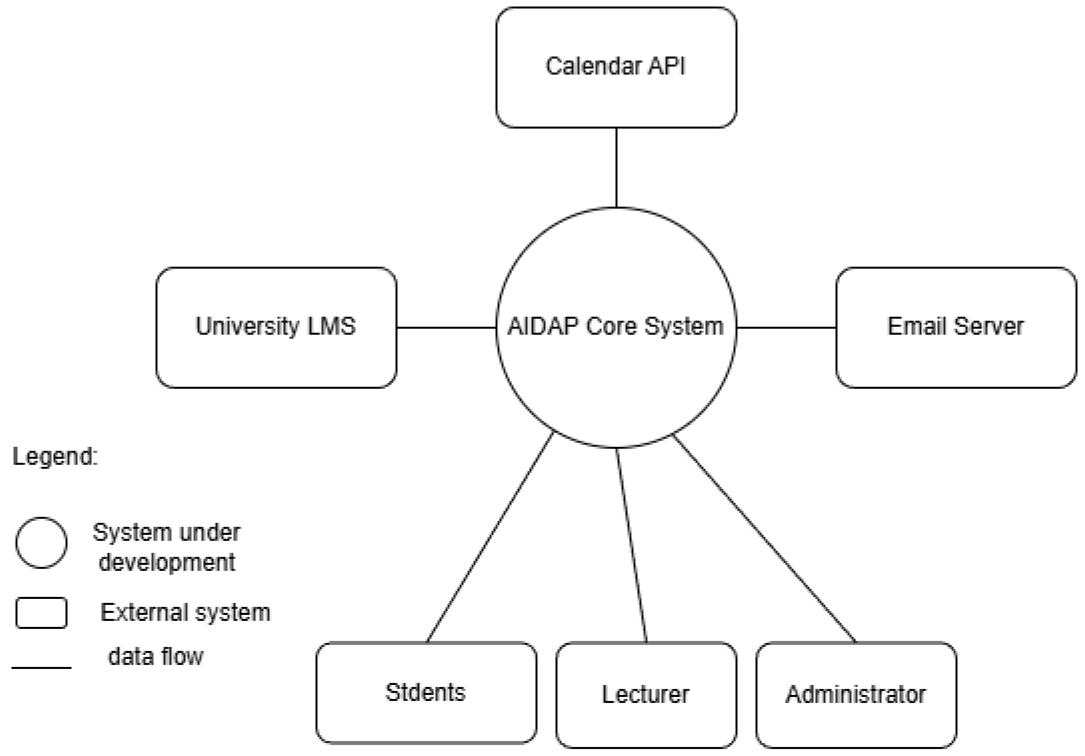


**Figure 1– Context Diagram for the AIDAP System**

This diagram shows the external entities (users and connected systems) interacting with the AIDAP Core System. Lines represent data exchange, consistent with the ADD context view format.

### 1.2.2 Step 3: Choose One or More Elements of the System to Refine

This is a greenfield development effort; therefore, in this iteration, the element to refine is the entire AIDAP Core System. The purpose of this step is to define the system boundaries and identify its external entities, which include institutional systems and user roles.
The Context Diagram (Figure 1) illustrates these relationships, showing how AIDAP interacts with connected systems, such as the University LMS, Calendar API, and Email Server, as well as end-users, Students, Lecturers, and Administrators. This context definition serves as the foundation for further architectural decomposition in subsequent iterations.

### 1.2.3 Step 4: Choose One or More Design Concepts That Satisfy the Selected Drivers

In this iteration, the selected design concepts define the high-level structure of the AIDAP system. They are based on the main architectural drivers such as scalability, availability, and maintainability. These decisions establish the foundation for system decomposition and guide future design refinements.

| **Design Decisions & Locations** | **Rationale** |
|---|---|
| Establish AIDAP core system as primary element of refinement | Defining the system boundary is essential before decomposition (CRN-5) |
| Define external entities | (External Entities include: Students, lecturers, administrators, LMS, registration system, Calendar API, Email Server, etc) Identification of actors defines how AIDAP must interact with external institutional systems. This supports maintainability of the system (QA-6). |
| Adopt a cloud-native architecture for the Core System | Required by CNST-6.<br>Enables dynamic scaling, and better system availability. It ensures the platform can support high amount of concurrent users (5000) and continuous operations (QA - 2, QA - 5) |
| Use containerization for all deployable components | Supports portable deployment and efficient scaling with an architecture that is cloud native. Directly contributes to performance and maintainability (QA - 1, QA - 6) |
| Structure the system using modular microservices | Allows for independent scaling and deployment while also ensuring faults are isolated. It is vital for maintainability and keeps the architecture modular (QA - 6, CRN - 5). |
| Introduce API gateway as access point for interactions | Centralize authentication, routing, and monitoring. Improves performance, simplifying the process of external integration. It also allows for a single entrypoint across the system services (QA - 1). |
| Require processing using message queues | This reduces latency and ensures that the system meets the response constraints. It also enhances fault tolerance and supports system scalability. (CNST-1, QA-2, QA-5). |
| Define the systems initial high level functional domains | Establishes the structure needed for iteration 2 decomposition. These domains include the Chat UI, AI Engine Domain, Integration Domain, and User/Profile Domain. This ensures the system architecture remains logically structured, modular, and maintainable (CRN-5, QA-6). |

**1.2.4 Step 5: Instantiate Architectural Elements, Allocate Responsibilities and Define Interfaces**
The instantiation design decisions considered and made are summarized in the following table:

| **Design Decisions &** | **Rationale** |
|---|---|

| **Locations** | |
|---|---|
| Create initial domain model | Identifies the main AIDAP entities (User, Academic Calendar, LMS Data, Sessions, Messages). This clarifies the foundation of the system and speeds up later design steps. |
| Identify use cases inside domain objects | Maps UC-1 (Merge Calendars), UC-4 (Zero-downtime Maintenance), UC-5 (High-Traffic Handling) to the domain model to ensure all key behaviors are supported. |
| Decompose domain objects into functional domains | Splits the system into clear areas (Chat UI, AI Engine, Integration, User/Profile). Helps distribute workload evenly and supports modular microservice design. |
| Instantiate API Gateway module | Provides a single controlled entry point for all users and external university systems. Handles routing, authentication, and monitoring. Supports performance and maintainability. |
| Create core microservice modules | Instantiates containerized modules for Chat UI (Django), AI Engine, Integration Service, and User/Profile Service. The User/Profile Service handles authentication, user data, and authorization logic. |
| Introduce Message Queue for async tasks | Supports notification delivery, background processing, and traffic spikes. Ensures fast responses during peak load (5000+ users). |
| Allocate storage components | Defines an SQL RDBMS for structured data and Object Storage for large files/archives. Ensures reliable data persistence and backup handling. |
| Define preliminary REST interfaces | Establishes the basic communication paths between the API Gateway and each microservice. Detailed interfaces will be refined in Iteration 2. |

**The results of these instantiation decisions are recorded in the next step.**

### 1.2.5 Step 6: Sketch Views and Record Design Decisions

**This is the modified version of the Rich Client architecture based on the decision made in step 5.**

**Figure 2- Rich Client architecture diagram**

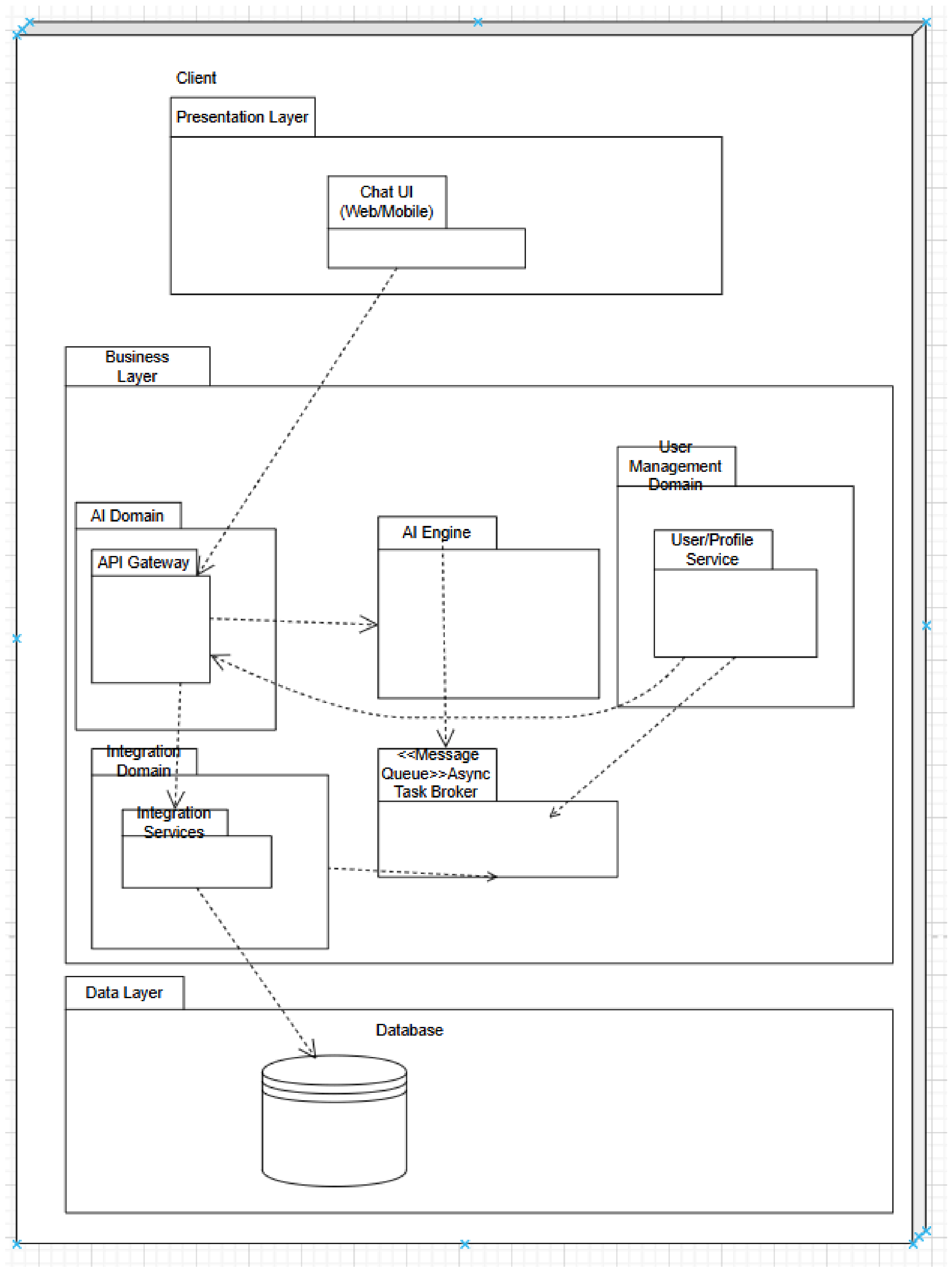
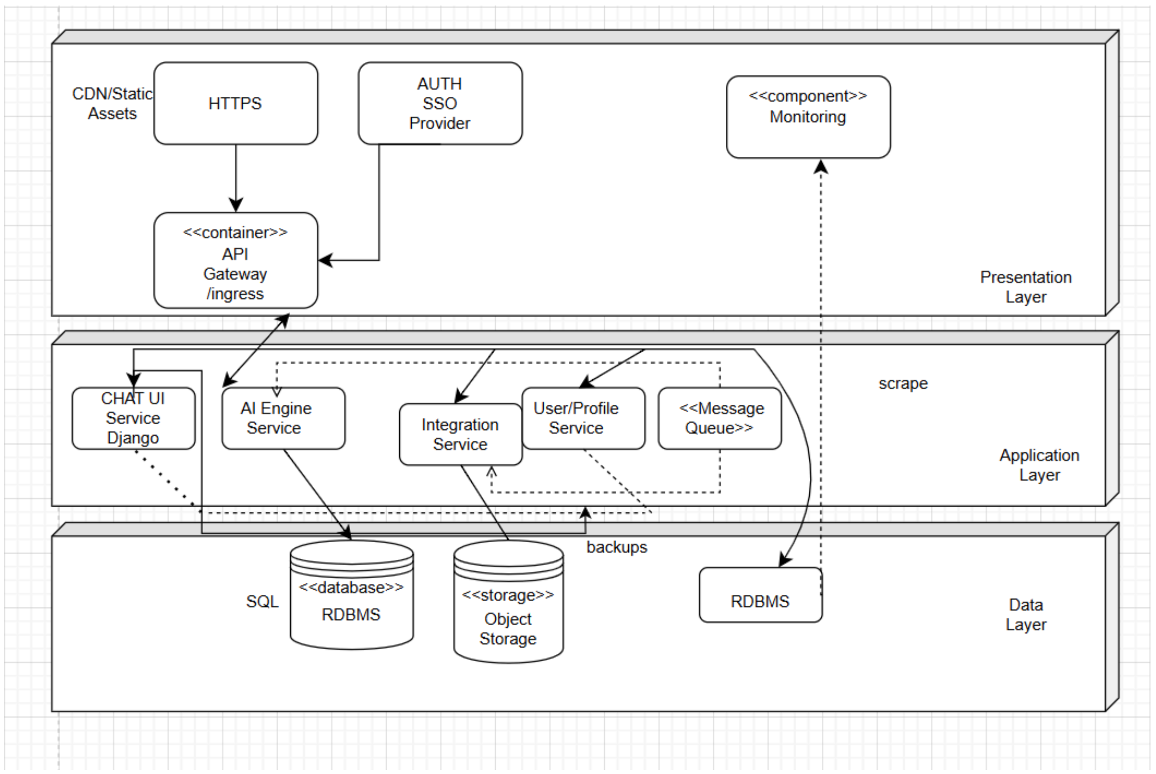


**Table based on the elements in the rich client architecture**

| Element | Responsibility |
|---|---|
| **Client Side** | Represents the user-facing environment where the system is accessed through web or mobile platforms. It is responsible for capturing user input and displaying the system’s responses in an intuitive and interactive way. |
| **Chat UI (Web/Mobile)** | Provides the main interaction interface for users. It allows them to communicate with the system through natural language and forwards queries to the backend via the API Gateway. |
| **API Gateway** | Serves as the central access point for all client-side requests. It manages request routing, authentication, and communication between the client and the AI Engine. This design simplifies external access and improves system security and scalability. |
| **AI Engine** | Represents the core processing component of the system. It analyzes user requests, applies intelligent logic or machine learning models, and generates responses based on system data and integrated services. |
| **User/Profile Service** | Manages user authentication, authorization, profile data, and access control policies. Ensures secure user interactions with the system. |
| **Message Queue** | Handles asynchronous communication between services by decoupling time-sensitive tasks such as notifications and background processing from real-time interactions. |
| **Integration Services** | Handles communication with external systems and internal services, such as academic databases, scheduling services, or analytics modules. It |

| | |
|---|---|
| | ensures seamless data exchange between the AI Engine and other subsystems. |
| **Database** | Responsible for storing and managing all persistent system data, including user records, interaction history, and domain-specific information. It supports data consistency, retrieval, and long-term storage. |

**Figure 3- Deployment Diagram**



**Responsibilities of the elements in the deployment diagram**

| Element | Responsibility |
|---|---|

| | |
|---|---|
| **CDN / Static Assets** | Distributes static frontend resources such as CSS, images, and scripts to reduce server load and improve performance. |
| **HTTPS Interface** | Enables secure communication between clients and the system using encrypted protocols. |
| **AUTH SSO Provider** | Provides authentication and single sign-on services for users. |
| **API Gateway (Ingress)** | Acts as the main entry point for client requests, handling routing and security enforcement. |
| **Chat UI Service (Django)** | Manages user interaction and sends requests to backend services. |
| **AI Engine Service** | Processes user queries using AI models and generates intelligent responses. |
| **Integration Service** | Manages communication with external systems and internal services. |
| **User/Profile Service** | Handles user account management and profile-related information. |

|  |  |
|---|---|
| **Message Queue (RabbitMQ / Kafka)** | Enables asynchronous communication between services to support background tasks and high traffic processing. |
| **RDBMS (SQL Database)** | Stores structured system data such as user records, logs, and system information. |
| **Object Storage** | Stores large files, system assets, and backup data. |
| **Monitoring Component** | Collects system logs, performance metrics, and service health information for monitoring and diagnostics. |

**Relationships between some elements in the deployment diagram**

| **Relationship** | **Description** |
|---|---|
| Client → API Gateway | Client requests are routed to the system through the API Gateway over HTTPS. |
| API Gateway → AI Engine Service | The API Gateway forwards user queries to the AI Engine for processing. |

| | |
|---|---|
| API Gateway → User/Profile Service | The API Gateway routes authentication and user-related requests to the User/Profile Service. |
| AI Engine → Message Queue | AI tasks are offloaded to the Message Queue for asynchronous execution. |
| Integration Service → Message Queue | Integration tasks are handled asynchronously using the Message Queue. |
| Core Services → RDBMS | All main services store and retrieve structured data from the relational database. |
| Core Services → Object Storage | Services use object storage for large files and backup handling. |
| System Services → Monitoring Component | All services send logs and performance metrics to the Monitoring system. |
| Object Storage → RDBMS (Backups) | Database backups are stored in the object storage system. |

### 1.2.6 Step 7: Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

| **Not Addressed** | **Partially Addressed** | **Completely Addressed** | **Design Decisions Made During the Iteration** |
|---|---|---|---|

|  |  |  |  |
|---|---|---|---|
|  | **UC-1** |  | API Gateway selected to support calendar merging requests efficiently. |
|  | **UC-4** |  | Microservice decomposition prepared for zero-downtime updates. |
|  | **UC-5** |  | Cloud-native, containerized services chosen to support high-traffic load. |
|  |  | **QA-1** | Message Queue introduced to improve performance under 5,000+ concurrent users. |
|  |  | **QA-2** | Redundant, container-based deployment strategy introduced for availability. |
|  |  | **QA-5** | Modular microservices allow independent scaling during peak load. |
|  |  | **QA-6** | Clear functional domains and service separation support maintainability. |
|  | **CNST-1** |  | High-traffic constraint addressed through asynchronous processing and efficient routing. |
|  |  | **CNST-6** | Cloud-native architecture enforced using containers and scalable storage. |
|  | **CRN-5** |  | Architecture structured into presentation, application, and data layers. |

## 1.3 Iteration 2: Refined Architectural Design

Iteration 2 represents the more refined and detailed design decisions we have made prior to development of the system. This step will allow us to form a coherent strategy before beginning the project, which will streamline the development process and prevent overlap.

### 1.3.1. Step 2. Establish Iteration Goal by Selecting Driver

The goal is to identify specific structures that aid functionality. Clarifying the use cases to be focused on in this iteration allows us to focus on the task at hand and not get sidetracked. In iteration 2, we decided to focus on the use cases we did not take into consideration in iteration 1.

| **Primary Use Cases** | **Description** |
|---|---|
| **UC-2: Maintaining Student Privacy** | A student requests to view their friend's grade for the recent midterm. The AI model detects that the user making the request is not authorized to view the material they are asking for, so it politely, but firmly, declines. |
| **UC-3: Detecting Drop in Participation** | The AI model notifies the lecturer that participation with assignments has decreased significantly over the last month. The lecturer requests to view the average GPA of his class and sees that it has lowered. They make an announcement regarding extra credit for bonus assignments. |
| **UC-6: Referencing Chat History** | The student uses the assistant to help with studying for an exam over the course of a week. The assistant is able to pick up where they left off the previous session by analyzing the previous messages in the chat. The assistant is also able to explain concepts in a way that the student can understand, based on their chat history. |

### 1.3.2. Step 3. Choose one or more elements of the system to refine

In this iteration, the main element to refine is the self-learning capabilities of the AI model. Since the purpose of this assistant is to guide students and lecturers through their academic careers with as little friction as possible, it is essential that the model is able to train effectively over numerous user interactions.

Another important element to refine is the security and privacy of users when interacting with the model. We must ensure that sensitive information such as grades are not given away by the assistant, even when directly prompted. To accomplish this, we will need a way to direct data flow to the correct sections of the system, to allow for careful filtering of outputs from the assistant.

### 1.3.3 Step 4. Choose one or more design concepts that satisfy the selected drivers

This step highlights the decision concepts used in the planning process. Each of these concepts were carefully selected to ensure the most efficient initial modelling as possible. We made our decisions based on the amount of detail relative to the time it would take to complete each step.

| **Design Decision** | **Explanation** |
|---|---|
| **Initial Domain Model** | A functioning domain model is crucial to the development of any system, even more so for ones pertaining to AI. Without identifying clear objects and entities within the system, development will be chaotic and random, contributing to working time and costs. Also important to distinguish differences between classes since privacy is a large concern. |
| **Mapping of domain objects to use cases** | Organizing the entities, operations and data flow of each use case is essential to ensuring all architectural drivers have been accounted for.<br>An effective alternative to using mapping tables would be a traceability matrix, although this method could become difficult to manage with many drivers to consider. |
| **Decompose domain objects into aggregates and value objects** | Breaking the large entities within each domain object into smaller components lessens the possibility of leaking private data, as well as simplifies the implementation of frameworks later on in development.<br>The alternative would be to skip decomposition entirely, but this is not recommended as domain objects may be unmanageable and tightly coupled. |
| **Organizing system into layered architecture** | Using the layered architecture style ensures a clear separation of concerns, which is important due to the private nature of certain objects. Isolating sensitive information from |

| | |
|---|---|
| | the machine-learning algorithms will also allow for more organized workflow.<br>An alternative we considered was blackboard architecture, as it is highly effective at analyzing data and integration, which could be a solid choice considering UC-3. Besides that, however, blackboard architecture is not typically used for the basic chat retrieval aspects of the system. |
| **Implementing Django and React frameworks** | Our back-end framework of choice is Django, not only because of its familiarity, but also due to its numerous built-in features such as user authentication. React can create dynamic graphs and analytics for our front-end, as well as interactive UI elements for grades and alerts.<br>Another alternative we considered was flask, for its simplicity in integrating AI models, but has far fewer built in functions compared to Django. |
| **Data Transfer Objects** | Combines with layered architecture to provide a structured method for transferring data between layers. This enforces privacy by monitoring what information is shared with certain layers.<br>The alternative to DTOs would simply be to directly use domain models, but we decided against it due to it being less efficient overall. |

### 1.3.4. Step 5. Instantiate Architectural Elements, Allocate Responsibilities, and Define Interfaces

The instantiation design decisions made in this iteration are summarized in the following table:

| **Design Decision** | **Explanation** |
|---|---|
| **Domain instantiation** | Each entity within the system must be identified and modeled clearly. This must be done thoroughly to ensure that all drivers are included within the design. Only an initial model is required, so that we are able to begin |

| | |
|---|---|
| | development as soon as possible. |
| **Map drivers to concrete services** | Dedicated service classes should be implemented for each use case identified. This will prevent tight coupling, as well as simplify development while maintaining privacy of data. |
| **Decompose domain objects into layer-specific modules** | Each use case must be accounted for and satisfied within the modules. The decomposition of domain objects will allow for thorough inspection of each module, ensuring that no use case has been neglected. This task is best suited for multiple team members, as to make sure nothing is left unchecked. |
| **Implement DTOs to define boundaries between layers** | Introduce DTOs for each layer to control the data that flows between boundaries. Creating separate DTOs for lecturers, students and other entities guarantee that sensitive information is kept within its own boundary. This also allows for simplified API implementation and looser coupling. |
| **Connect Components using Django** | Connect architectural elements using chosen frameworks, allowing it to become runtime executable. Django uses app configuration to register modules, URL routing to map views to endpoints, and APIViews to expose services. |
| **Create browser-based interface using React** | Conceptual screens and data flows are implemented as functional react components. UI components are wired to backend contracts and services. React also enforces privacy and access routes in the client side. |

### 1.3.5. Step 6 - Sketch Views and Record Design Decisions

As a result of the decisions made in step 5, several diagrams are created.

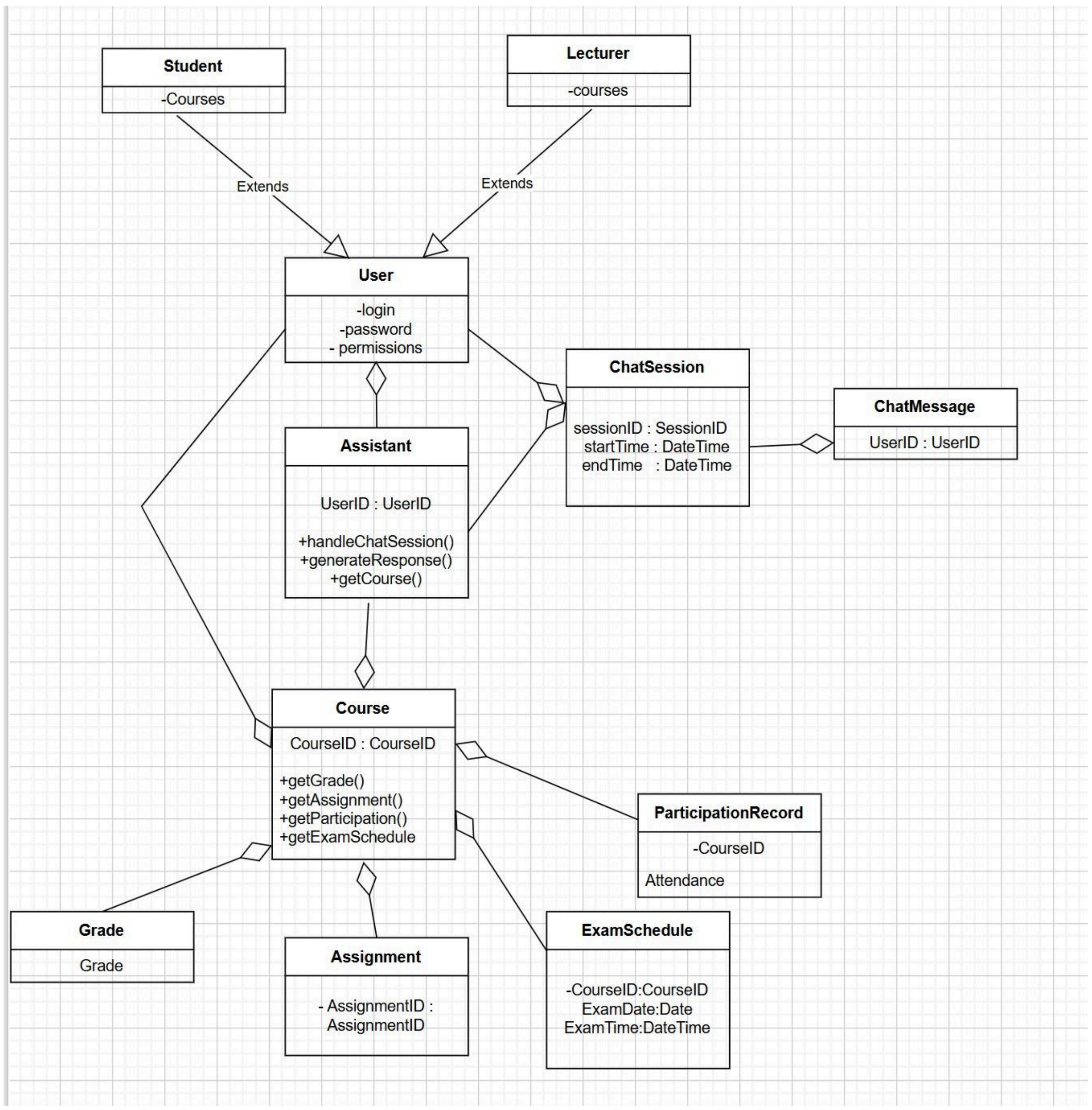


**Figure 4 - Initial Domain Model**

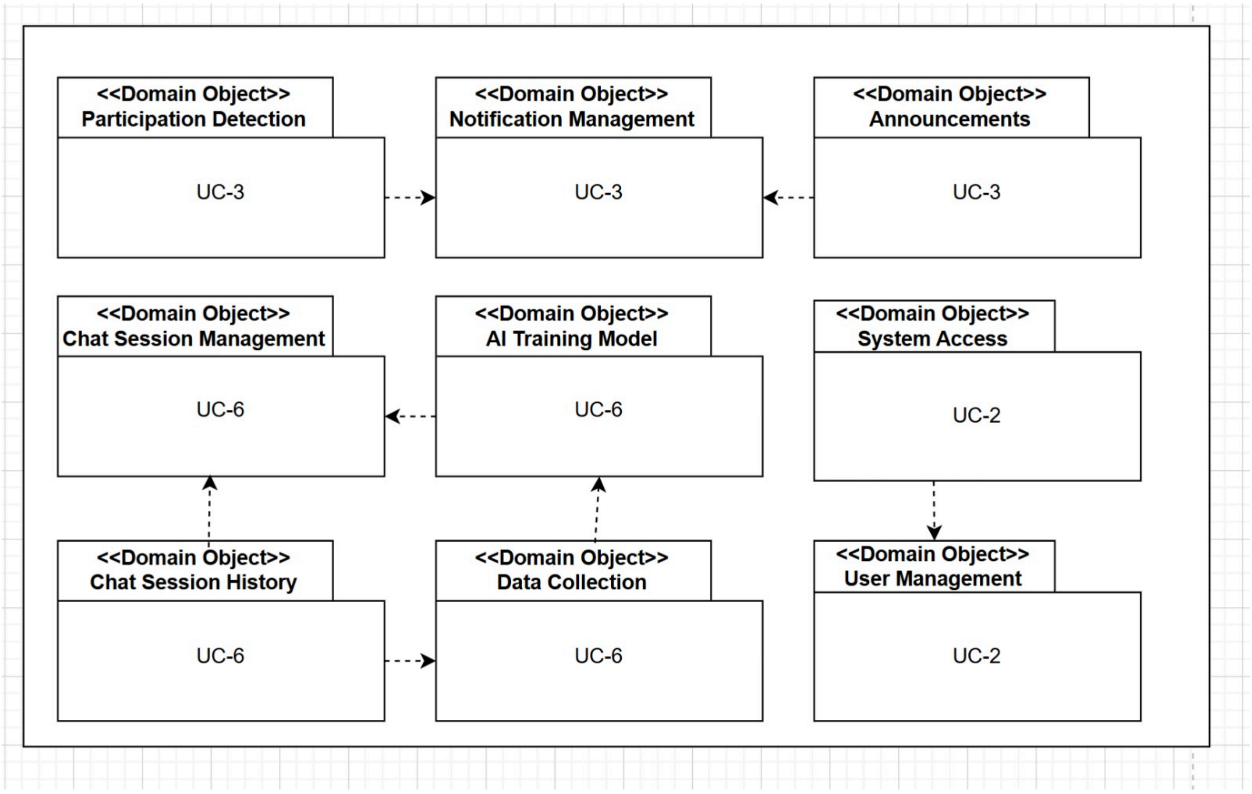


**Figure 5 - Domain objects associated with the use case**

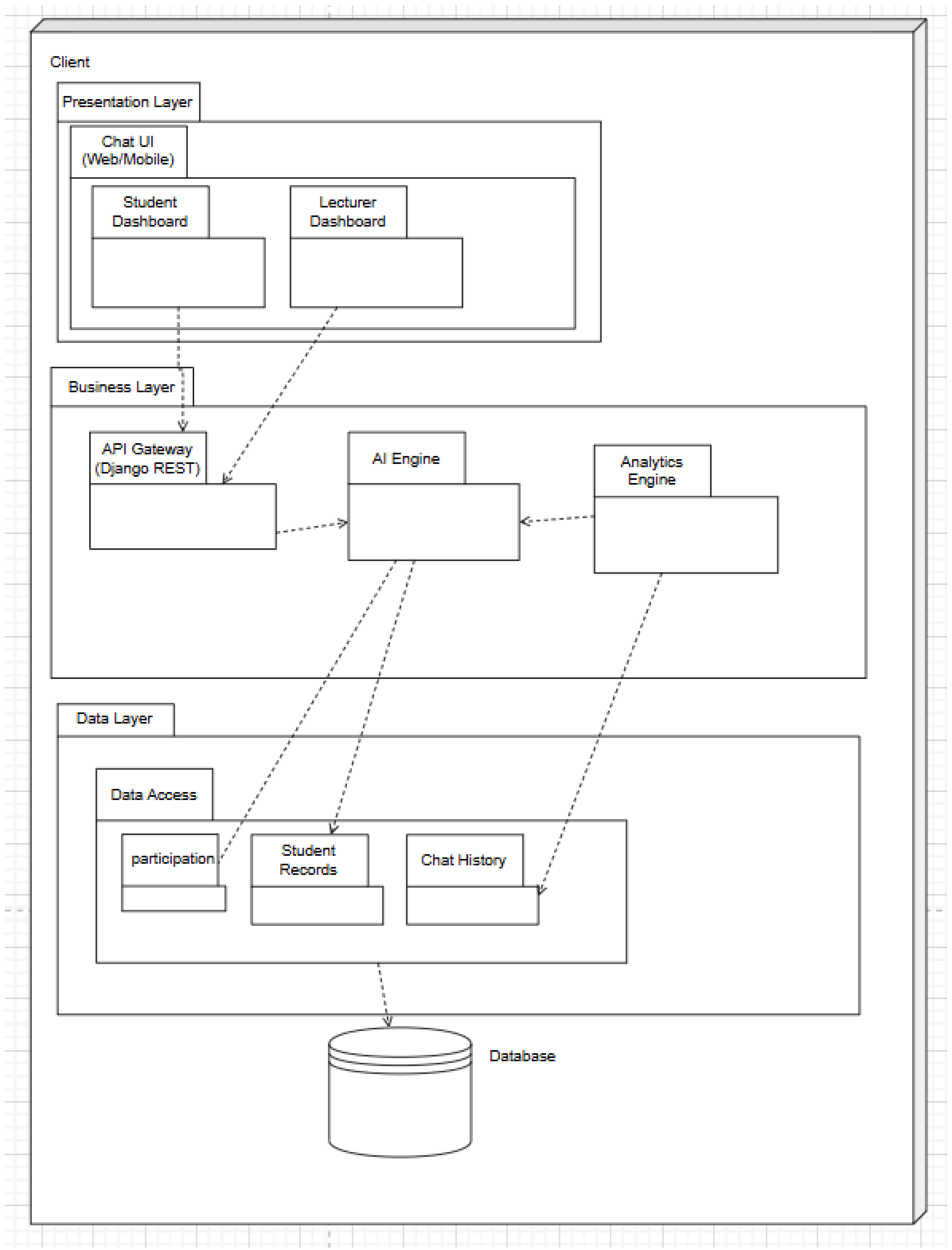


**Figure 6 - Modules that support the primary use cases**

| Element | Responsibility |
|---|---|
| Chat UI (Web/Mobile) | User interface for the assistant, available on the website and on mobile |
| Student Dashboard | Student view of the dashboard |
| Lecturer Dashboard | Lecturer view of the dashboard |
| API Gateway (Django REST) | Allows users to interact with the assistant |
| AI Engine | Uses machine learning to find patterns in data to make decisions |
| Analytics Engine | Processes data using queries and statistical methods |
| Participation | Contains data about attendance and submissions during lectures |
| Student Records | Contains data concerning grades for all students |
| Chat History | Contains previous conversations with the assistant to train the model |

**UC - 2: Maintaining Student Privacy**

Figure # shows an initial sequence diagram for UC - 2 (Maintaining student privacy). It shows how the system checks user credentials prior to accessing and providing sensitive information. Upon the user's request, the AIDAP system verifies student credentials and passes the information to the record system, where it then confirms that the requested information is not equal to the account information making the request. The AIDAP system then sends a polite prompt to the user letting them know that they are unauthorized to access the requested information.

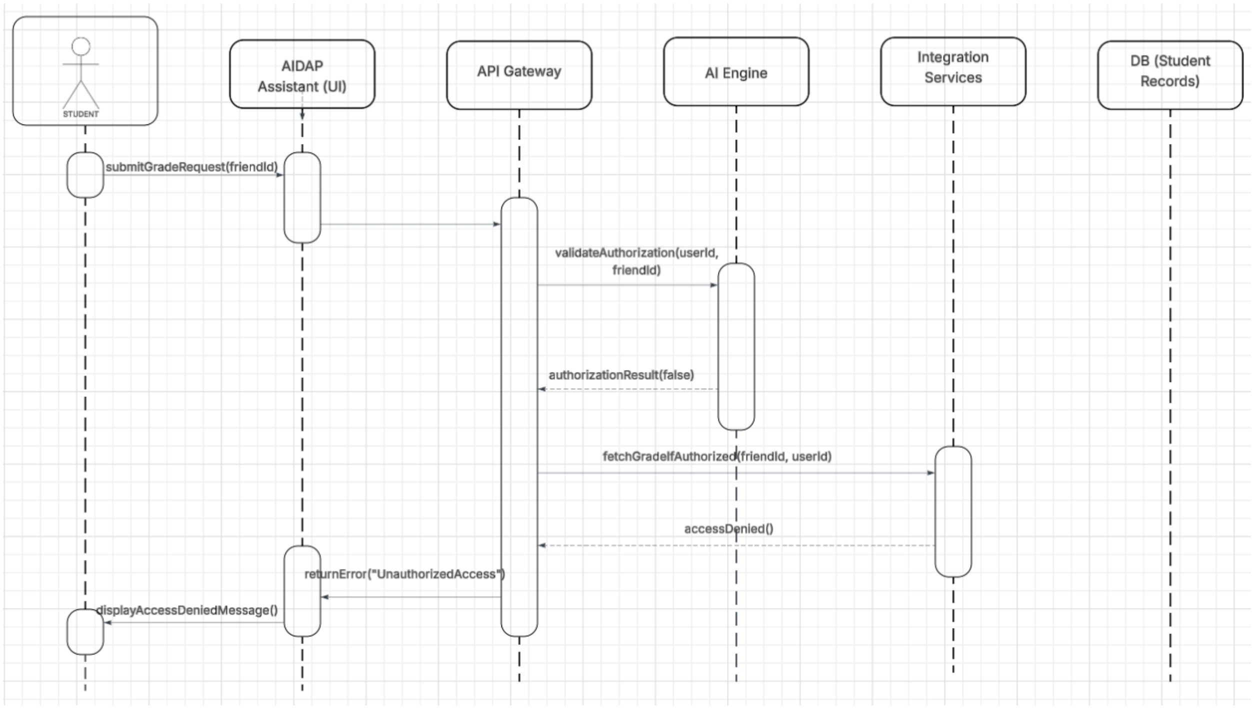

STUDENT
AIDAP Assistant (UI)
API Gateway
AI Engine
Integration Services
DB (Student Records)
submitGradeRequest(friendId)
validateAuthorization(userId, friendId)
authorizationResult(false)
fetchGradeIfAuthorized(friendId, userId)
accessDenied()
returnError("UnauthorizedAccess")
displayAccessDeniedMessage()

**Figure 7 Sequence Diagram for UC - 2**

From the interactions presented and identified in the sequence diagram above, initial methods for interfaces of all the interacting elements present can be identified:

| **Method Name** | **Description** |
|---|---|
| **Element:** AIDAP Assistant (UI) | |
| `submitGradeRequest(friendId)` | Student submits request to view their friends grades |
| `displayAccessDeniedMessage()` | User receives a prompt stating their access is denied |
| **Element:** API Gateway | |
| `returnError("UnauthorizedAccess")` | Returns authorization failure |
| **Element:** AI Engine | |
| `validateAuthorization(userId, friendId)` | Compares and determines if logged in account and requested account are equal or vary |
| **Element:** Integration Services | |

| | |
|---|---|
| `fetchGradeIfAuthorized(friendId, userId)`<br><br>`accessDenied()` | Attempts to fetch grade if accounts are equal, if not it denies access<br><br>Replies with a denied access |

**UC - 3: Detecting Drop in Participation**

Figure # shows an initial sequence diagram for UC - 3 (Detecting Drop in Participation). It shows how the system periodically monitors user participation within courses and provides lecturers with notifications on its observations. The analytics tracker built into the AIDAP system tracks the user participation across courses and when a drop is present, it generates a notification and prompts the lecturer of its observations. The lecturer is then able to access GPA information from the students record system from which they are able to use the AIDAP system to generate and publish a message to all of the users registered within the course.

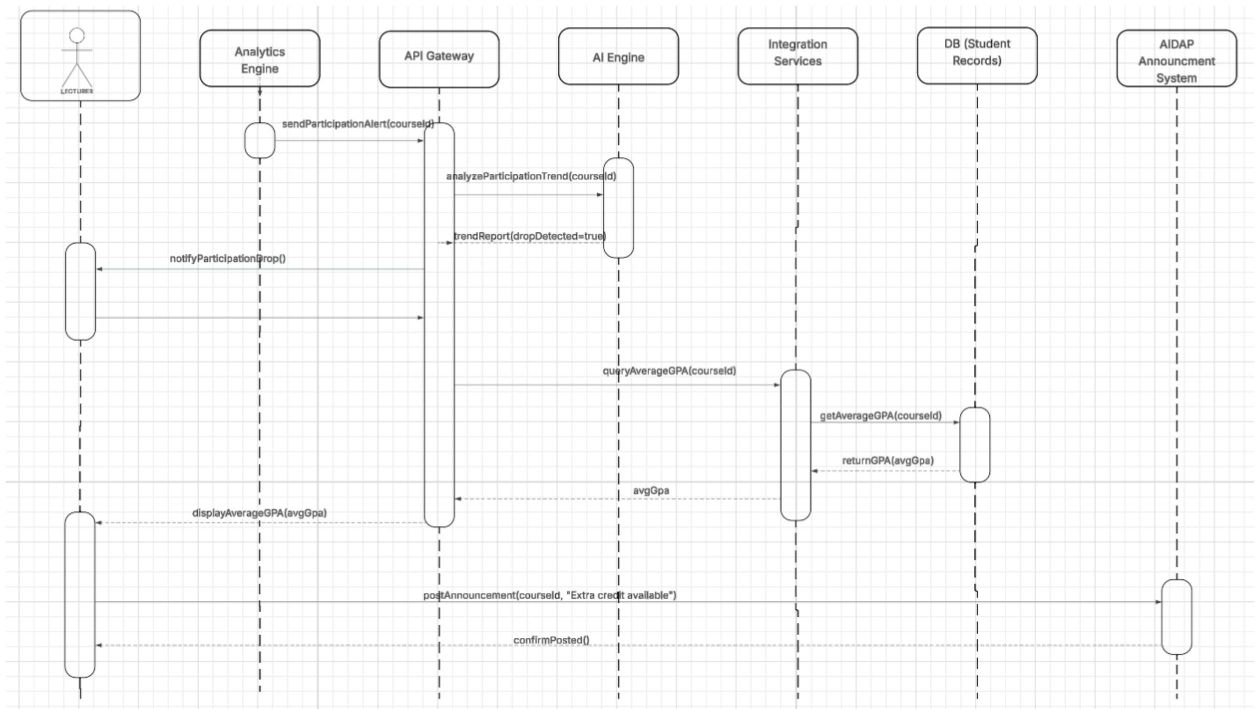

LECTURER
Analytics Engine
API Gateway
AI Engine
Integration Services
DB (Student Records)
AIDAP Announcment System
sendParticipationAlert(courseId)
analyzeParticipationTrend(courseId)
trendReport(dropDetected=true)
notifyParticipationDrop()
queryAverageGPA(courseId)
getAverageGPA(courseId)
returnGPA(avgGpa)
avgGpa
displayAverageGPA(avgGpa)
postAnnouncement(courseId, "Extra credit available")
confirmPosted()

**Figure 8 Sequence Diagram for UC - 3**

From the interactions presented and identified in the sequence diagram above, initial methods for interfaces of all the interacting elements present can be identified:

| **Method Name** | **Description** |
|---|---|

| | |
|---|---|
| **Element:** Analytics Engine | |
| `sendParticipationAlert(courseId)` | Sends an alert that participation within the course has dropped |
| **Element:** API Gateway | |
| `notifyParticipationDrop()` | Notifies the lecturer of the drop in participation |
| **Element:** AI Engine | |
| `analyzeParticipationTrend(courseId)` | Confirms and analyzes the trends in the course participation |
| **Element:** Integration Services | |
| `queryAverageGPA(courseId)` | Retrieves the classes GPA |
| **Element:** DB (Student Records) | |
| `getAverageGPA(courseId)` | Executes GPA query |
| **Element:** AIDAP Announcement System | |
| `postAnnouncement(courseId, message)` | Posts extra credit announcement to the course registrants |

**UC - 6: Referencing Chat History**

Figure # shows an initial sequence diagram for UC - 6 (Referencing Chat History).

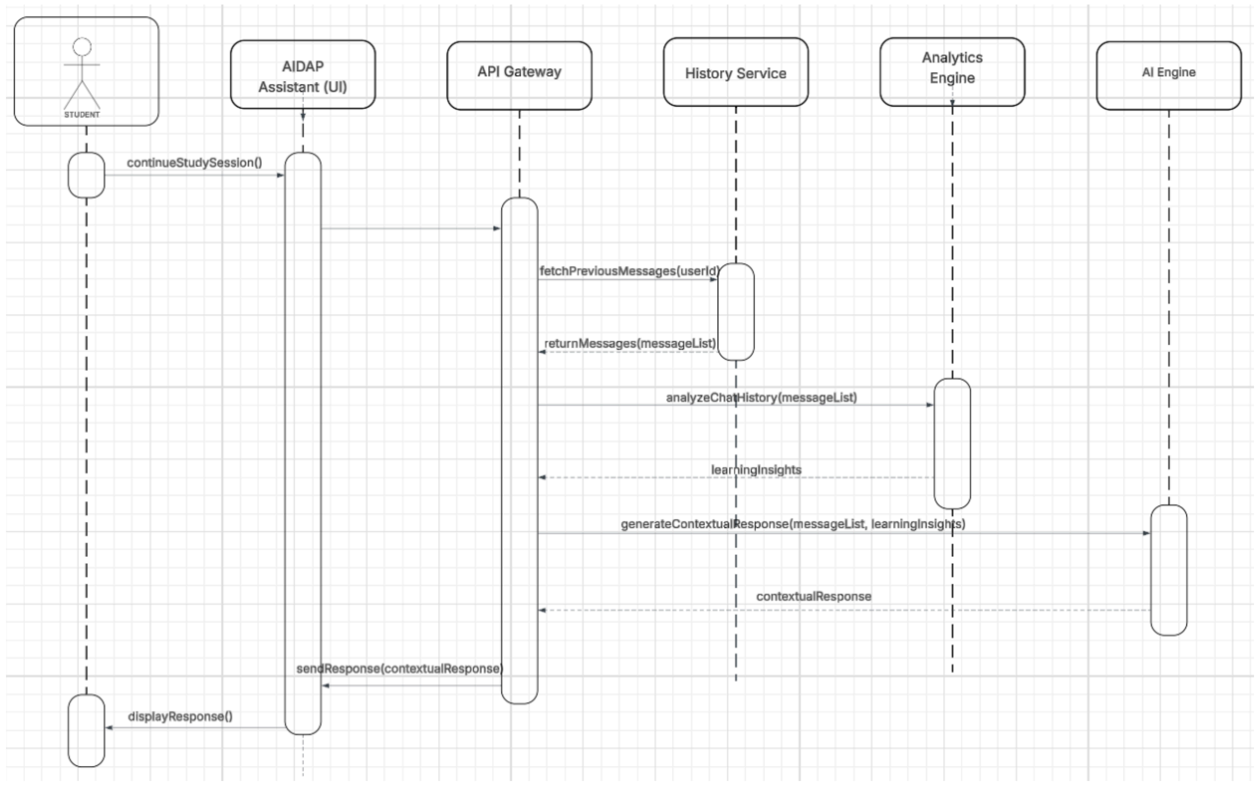

STUDENT
AIDAP Assistant (UI)
API Gateway
History Service
Analytics Engine
AI Engine
continueStudySession()
fetchPreviousMessages(userId)
returnMessages(messageList)
analyzeChatHistory(messageList)
learningInsights
generateContextualResponse(messageList, learningInsights)
contextualResponse
sendResponse(contextualResponse)
displayResponse()

**Figure 9 Sequence Diagram for UC - 2**

From the interactions presented and identified in the sequence diagram above, initial methods for interfaces of all the interacting elements present can be identified:

| **Method Name** | **Description** |
|---|---|
| **Element:** AIDAP Assistant (UI) | |
| `continueStudySession()`<br><br>`displayResponse()` | Student resumes a past study session<br><br>Displays response |
| **Element:** API Gateway | |
| `sendResponse(contextualResponse)` | Sends the final generated response to the display |
| **Element:** History Service | |
| `fetchPreviousMessages(userId)` | Retrieves the most recent message to continue the chat |

| **Element:** Analytics Engine | |
|---|---|
| `analyzeChatHistory(messageList)` | Analyzes past content for learning patterns |
| **Element:** AI Engine | |
| `generateContextualResponse(messageList, learningInsights)` | Generates the next set of responses based on past learning and analyzed patterns |

### 1.3.6 Step 7 - Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

| **Not Addressed** | **Partially Addressed** | **Completely Addressed** | **Design Decisions Made During the Iteration** |
|---|---|---|---|
| | UC - 2 | | Required modules that support Student privacy have been identified. |
| | UC - 3 | | Required modules that support analytics, alerts, and retrieval have been identified. |
| | UC - 6 | | Required modules supporting history retrieval and analysis have been identified. |
| QA - 1 | | | No relevant decisions made, as it is necessary to identify the element that participate in the Use Case that is associated in the scenario |
| QA - 2 | | | No relevant decisions made, as it is necessary to identify the element that participate in the Use Case that is associated in the scenario |
| | QA - 5 | | Scalability partially addressed through layered architecture. |
| | QA - 6 | | Maintainability partially addressed through modular services, and clear layering. |
| | CNST - 5 | | Performance partially supported through layered architecture and separation of concerns. |

| | | | |
|---|---|---|---|
| CNST - 6 | | | No relevant decisions made, as it is necessary to identify the element that participate in the Use Case that is associated in the scenario |
| | | CRN - 5 | Logically structured architecture with clearly defined layers, and domain decompositions. Completely supports modularity and maintainability. |