Department of Computer Science & Engineering
The University of Texas at Arlington

Architectural Design Specification
CSE 4316: Senior Design I
Fall 2025

Builders Squad
MavHousing

Aastha Khatri
Alok Jha
Aviral Saxena
Atiqur Rahman
Talha Tahmid
Md Rashidul Alam Sami

Revision History

Revision	Date	Author(s)	Description
0.1	11.09.2025	AK	Initial creation of the Architectural Design Specification (ADS) document with basic structure and placeholders for sections.
0.2	11.12.2025	AK	Added detailed introduction section with system overview, system scope, key requirements, and document structure.
0.3	11.20.2025	AK, AJ, AS	Expanded All layers descriptions with features, functions, and critical interfaces.
1.0	11.21.2025	AJ	Created figures for the ADS.
1.1	11.22.2025	AK	Completed detailed subsystem descriptions for Frontend Layer.
1.1a	11.22.2025	AS	Completed detailed subsystem descriptions for Data Layer.
1.1b	11.22.2025	AS, AJ, AK	Completed detailed subsystem descriptions for Backend Layer.
1.2	11.23.2025	AK, AJ, AS, TT, AR, MS	Finalized ADS document for submission with all subsystem details, diagrams.

Contents

List of Figures

List of Tables

1 Introduction

MavHousing is a comprehensive web-based platform designed to manage student housing operations at the University of Texas at Arlington (UTA). The system centralizes and streamlines all housing-related processes, making essential tasks like housing applications, room assignments, maintenance requests, and payment management faster, more efficient, and transparent for all users. The platform serves students, housing staff, and administrators with role-based access and functionalities tailored to each user type. Additionally, the system includes an AI-powered assistant named Blaze that provides real-time answers to housing-related questions and sends automated reminders for important deadlines and updates. By integrating modern cloud technologies, event-driven architecture, and intelligent automation, MavHousing enhances the convenience and effectiveness of housing operations while ensuring data security and compliance with privacy regulations.

1.1 Product Concept

MavHousing will provide a secure and user-friendly platform for students to apply for housing, submit maintenance requests, and make payments while tracking their application status in real time. Housing staff will manage applications, assign rooms, and process maintenance requests through streamlined workflows. Administrators will oversee system-wide operations with access to analytics, user management, and configuration controls. The system will offer real-time notifications via WebSocket, AI-powered assistance through Blaze, and comprehensive audit logging for compliance. Integration with secure authentication using JWT tokens and role-based access control will enhance data protection and operational efficiency.

1.2 System Scope

The scope of MavHousing encompasses all major housing management functions, serving three distinct user groups:

- **Students:** Students can apply for housing, track their application status, submit maintenance requests, make payments, view room assignments, and interact with the Blaze AI assistant for instant help and information.
- **Housing Staff:** Staff members can review and manage housing applications, assign rooms to students, process maintenance requests, generate reports, and oversee day-to-day housing operations.
- **Administrators:** Administrators have access to system-wide analytics, can oversee all housing operations, manage user accounts, configure system settings, and monitor system performance and usage.

1.3 Key System Requirements

The MavHousing system is designed to meet the following critical requirements:

1. **Security and Access Control:** The system implements secure user authentication using JWT tokens and role-based access control (RBAC) to ensure that students, staff, and administrators can only access features and data appropriate to their roles.
2. **Performance and Reliability:** The system processes housing applications, payments, and maintenance requests quickly and reliably, using event-driven architecture and background job processing to handle high volumes of requests without delays.

3. **User Experience:** The system provides clear, intuitive user interfaces for all user types, with role-based dashboards that adapt to show only relevant information and features for each user.

4. **Room Assignment and Roommate Matching:** The system includes an intelligent unit assignment and roommate matching module that optimizes room allocation based on student preferences such as building choice, room type, budget, and lifestyle habits. A rule-based algorithm enhanced with machine learning calculates compatibility scores between applicants to generate fair and efficient assignments while accounting for gender policies, capacity limits, and accessibility requirements.

5. **Communication and Notifications:** The system provides automated email communication for key user actions such as application submissions, payment confirmations, and maintenance updates, along with real-time in-app messaging and WebSocket-based notifications to ensure timely updates and effective engagement between students, housing staff, and administrators across multiple channels including web portal, email, and Microsoft Teams integration.

6. **AI-Powered Assistance:** The system features Blaze, an AI assistant powered by the Model Context Protocol (MCP) framework integrated with the Gemini large language model. Through the MCP server, Blaze can securely access real-time system data using specialized tools, enabling it to provide context-aware responses to student questions about application status, payment deadlines, and maintenance requests. The assistant sends automated notifications, delivers personalized reminders across multiple platforms including web portal, email, and Microsoft Teams, and helps users navigate complex housing processes through natural language interactions.

7. **Data Management and Compliance:** The system maintains accurate data records with comprehensive auditing and logging capabilities, ensuring compliance with privacy standards such as FERPA (Family Educational Rights and Privacy Act) and providing a complete audit trail of all system actions.

2 System Overview

The architectural design of our software system is organized into three main layers: the Frontend Layer the Data Layer and the Backend Layer. Each layer contains components that share similar responsibilities and work together to support the entire workflow of the application, from user interaction to backend processing and secure data storage. The layers are separated clearly so that each part of the system remains independent, maintainable, and easy to update in the future.

The high-level block diagram below shows how these layers interact with one another.

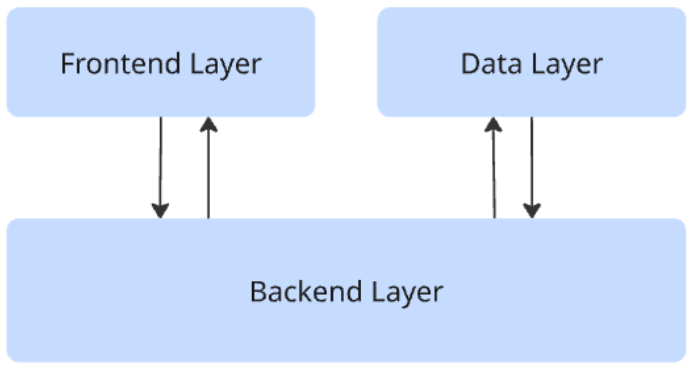

Figure 1: Architectural Layer Diagram for the MavHousing System

2.1 Frontend Layer

The Frontend Layer is the front-facing part of the system. It includes all user interface components that students, staff, and admins interact with. This layer ensures that users can easily perform actions like logging in, navigating pages, submitting housing applications, or receiving notifications.

2.1.1 Features and Functions

- Displays the landing page, dashboards, and forms for students and admins.
- Provides the login screen and handles credential input.
- Handles file uploads (PDFs, documents) for the application process.
- Shows real-time updates to users through WebSocket notifications.
- Supports role-based UI updates, for example, students see a student dashboard, and admins see an admin panel.

2.1.2 Critical Interfaces and Interactions

- Sends login information and form submissions to the Backend Layer through API Client Service.
- Works with the Route Guard to check user authentication and permissions before entering protected pages.
- Interacts with State Management, which stores user sessions, tokens, and role information.
- Receives data and success messages from the backend, such as:

  - application submission confirmation,
  - updated dashboard data,
  - notifications via WebSockets.
- Uses Auth Interceptor to attach JWT tokens to all outgoing requests.

2.2 Data Layer

The Data Layer is responsible for all permanent storage and retrieval of system information. It provides a structured environment for maintaining user records, housing applications, uploaded file URLs, and system logs. This layer ensures that the systemâs data remains safe, consistent, and easy to access.

2.2.1 Features and Functions

- **User Storage:** Stores user accounts, credentials, roles, and identity details.
- **Application Records:** Saves all submitted housing applications, including preferences, timestamps, and document URLs.
- **Document Storage Links:** Stores references to uploaded documents stored in Google Cloud Storage.
- **Logging Database:** Contains audit logs for system actions such as submissions and email events.
- **Data Integrity & Validation:** Ensures that all data passed from the Backend Layer is stored properly and securely.

2.2.2 Critical Interfaces and Interactions

- Communicates with the Backend Layer through the Internal API Gateway.
- Repository Manager executes SQL queries on PostgreSQL (GCP Datastore) for structured data such as users, applications, and sessions.
- Interacts with MongoDB Atlas for unstructured log data.
- Supports CRUD operations requested by Domain Services.
- Returns query results back to the Backend Layer for frontend display and further processing.

2.3 Backend Layer

The Backend Layer is the brain of the system. It coordinates authentication, authorization, application processing, file handling, and event-driven operations such as sending emails or generating logs. It acts as the middle layer between the Frontend Layer and the Data Layer.

2.3.1 Features and Functions

- **Authentication & Authorization:**
  - Auth Guard validates login credentials.
  - RBAC (Role-Based Access Control) checks if the user has permission to perform an action.
- **Domain Services:**
  - Handles all housing application logic.

  - Coordinates file uploads, student actions, and admin approvals.
- **File Ingestion Service:** Ensures uploaded files are valid, safe, and correctly stored.
- **Cloud Storage Management:** Uploads files to Google Cloud Storage and returns secure URLs.
- **Event-Driven Communication:**
  - Sends notifications when applications are submitted.
  - Triggers Queue Worker jobs for emails and logs.
- **Logging System:** Logs major actions like application submitted and stores them in MongoDB.

2.3.2 Critical Interfaces and Interactions

- Receives and validates all incoming requests from the Frontend Layer.
- Sends structured data queries or commands to the Data Layer to fetch or save information.
- Uses Event Bus and Queue Workers to handle long-running tasks such as:
  - sending emails,
  - logging events,
  - notifying staff members.
- Works with the Communications Service to send emails (via SendGrid) and push WebSocket updates.
- Exchanges data with the Data Layer through the Database Service using a highly organized and secure interface.

3 Subsystem Definitions & Data Flow

This section breaks down the layers into smaller subsystems and shows how they interact with each other. The Frontend Layer includes the Web Application UI, Route Guard, API Client Service, Auth Interceptor, State Management, and WebSocket Client. These subsystems send requests and data to the Backend Layer, which contains External API Gateway, Auth Guard, Role Based Access Control (RBAC) and Route Guard, Domain Services, Internal API Gateway, Database Service, File Management Service, Cloud Storage Controller, and Communications Service. The Backend Layer processes authentication, application submissions, file uploads, and notifications. It communicates with the Data Layer through the GraphQL Gateway and Repository Manager, which coordinate access to storage services including GCP Datastore Service, MongoDB Atlas Service, and Google Cloud Storage. These storage services store user records, applications, document links, and logs. Event-driven subsystems such as Event Bus, Event Handler Registry, Queue Worker Service, and Notification Manager handle asynchronous tasks like sending emails and updating logs. All subsystems work together to ensure smooth data flow and proper operation of the housing application system.

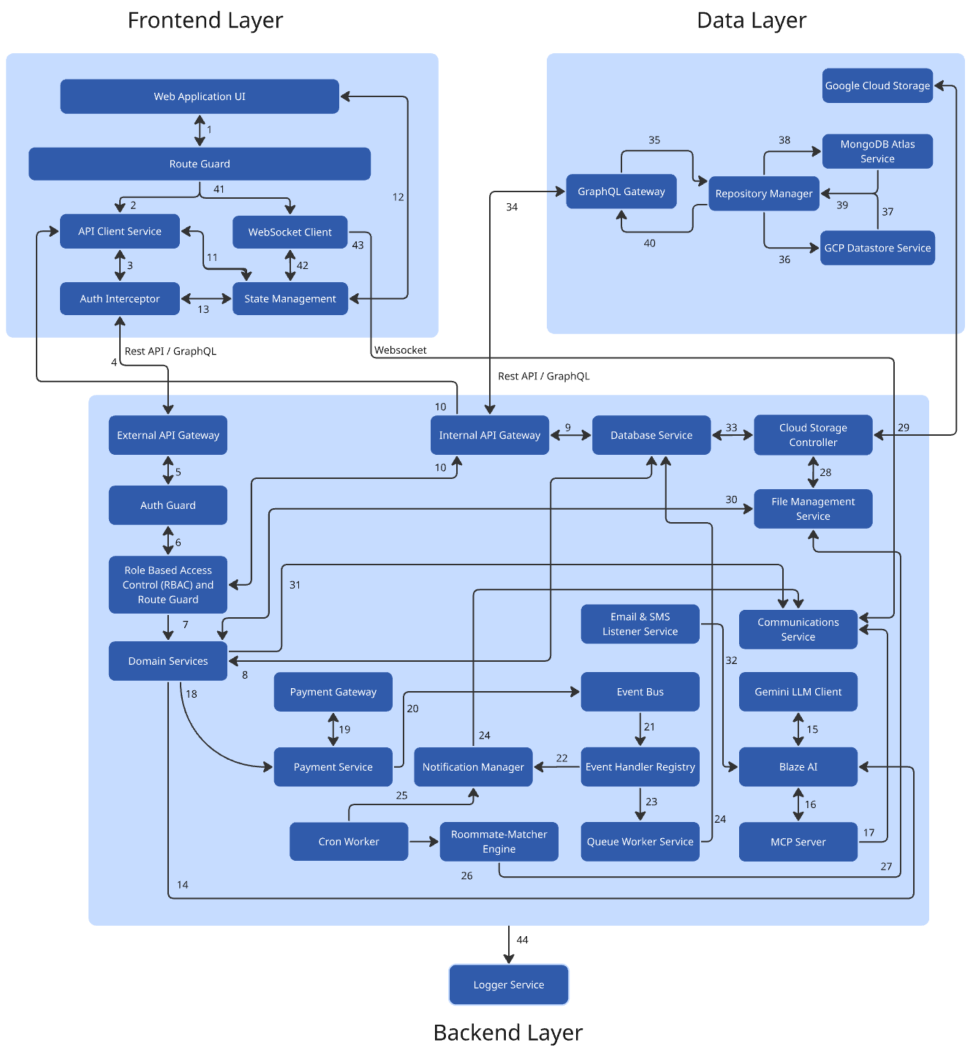

Figure 2: A simple data flow diagram

4 Frontend Layer Subsystems

This section describes each of the six main subsystems that make up the Frontend Layer. These subsystems work together to create the user interface, handle user interactions, manage authentication, communicate with the backend, and store application state. Each subsystem is explained in detail below, including what it does, what assumptions we make about it, its responsibilities, and how it connects to other subsystems through specific interfaces. The Frontend Layer includes the following subsystems:

1. Web Application UI
2. Route Guard
3. API Client Service
4. Auth Interceptor
5. WebSocket Client
6. State Management

4.1 Web Application UI

The Web Application UI is the part of the system that users actually see and interact with in their web browser. It shows all the pages, forms, dashboards, and buttons that students, staff, and admins use. When a user clicks something, types in a form, or navigates to a new page, the Web Application UI captures that action and works with other frontend subsystems to make things happen. It also displays data from the backend and shows real-time notifications when they arrive.

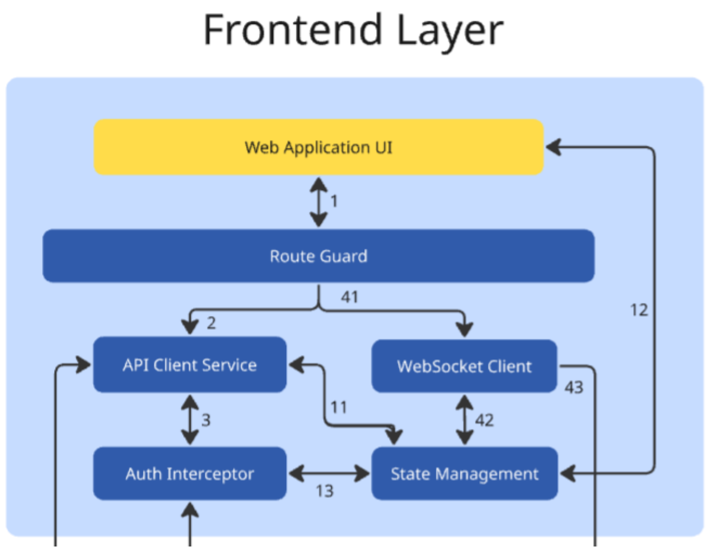

Figure 3: Web Application UI Subsystem

4.1.1 Assumptions

- The UI runs in a modern web browser that supports JavaScript, React (or similar framework), and WebSocket connections.
- User interactions (clicks, form submissions, navigation) are captured and processed in real-time.
- The UI receives state updates from State Management to trigger re-renders when authentication status, user data, or application state changes.

- Route Guard validates all navigation requests before allowing page transitions.
- The UI adapts its display based on user roles (student, staff, admin) retrieved from State Management.

4.1.2 Responsibilities

- Displays landing pages, login screens, dashboards, and application forms based on the userâs authentication state.
- Captures user actions such as form submissions, button clicks, file uploads, and navigation requests.
- Adjusts the interface depending on the userâs role so students and admins see the appropriate dashboard.
- Collects form inputs and uploaded files before sending them to the API Client Service.
- Re-renders UI components when State Management updates authentication or application data.
- Shows real-time notifications and alerts received from the WebSocket Client.

4.1.3 Subsystem Interfaces

Table 2: Web Application UI Subsystem Interfaces

ID	Description	Inputs	Outputs
#1	Bidirectional communication with Route Guard for navigation and authentication checks	Navigation Request Page Access Request	User Credentials Form Data Navigation Intent
#12	Bidirectional communication with State Management for state updates and UI re-renders	Authentication Status User Data (id, email, role) Application State Notification Messages	State Update Requests UI State Queries

4.2 Route Guard

The Route Guard subsystem enforces client-side access control by validating user authentication and authorization before allowing navigation to protected routes. It intercepts all navigation requests from the Web Application UI and queries State Management to verify the user's authentication status and role. Based on this information, Route Guard determines whether the user has sufficient permissions to access the requested route. Authenticated users with appropriate roles are granted access, while unauthorized users are either redirected to the login page or presented with an access denied message. This mechanism ensures role-based route protection, preventing students from accessing administrative interfaces and restricting protected features to authenticated users only.

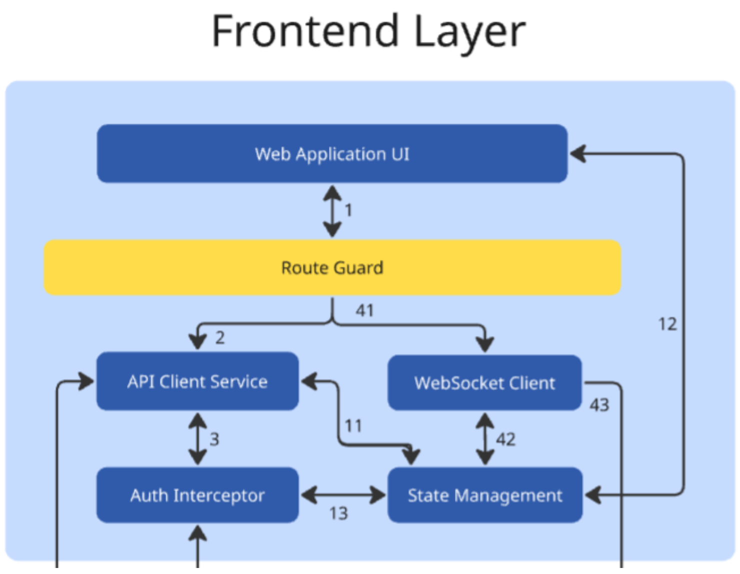

Figure 4: Web Application UI Subsystem

4.2.1 Assumptions

- All navigation requests from Web Application UI pass through Route Guard before page rendering.
- Route Guard queries State Management to verify current authentication status and user role.
- Public routes (like login page) are allowed without authentication, while protected routes require valid authentication and appropriate role.
- Route Guard can redirect unauthenticated users to the login page or deny access with appropriate error messages.
- Authentication and role information stored in State Management is current and reliable.

4.2.2 Responsibilities

- Intercepts navigation requests before a page change is allowed.
- Checks State Management to confirm the user is logged in.
- Verifies the user has the correct role to access the route.
- Allows access to the login page and forwards login attempts to the API Client Service.
- Decides whether to grant or deny access based on authentication and role.
- Sends approved requests to the API Client Service or WebSocket Client.
- Redirects users to the login page or error pages when needed.

4.2.3 Subsystem Interfaces

Table 3: Route Guard Subsystem Interfaces

ID	Description	Inputs	Outputs
#1	Bidirectional communication with Web Application UI for navigation requests and access decisions	Navigation Request Page Access Intent User Credentials (for login)	Access Granted Access Denied Redirect Instruction
#2	Routes authenticated API requests to API Client Service	Login Data Form Submission Requests API Request Intent	Formatted API Request Request Context
#41	Routes WebSocket connection requests to WebSocket Client	WebSocket Connection Intent	WebSocket Connection Request

4.3 API Client Service

The API Client Service is like a messenger that prepares and sends all requests from the frontend to the backend server. When a user submits a form, logs in, or requests data, the API Client Service takes that information and formats it into a proper request that the backend can understand. It works with Route Guard to get the request and with Auth Interceptor to add security tokens. After the backend responds, the API Client Service receives the answer, processes it, and sends the data to State Management so the UI can update and show the results to the user.

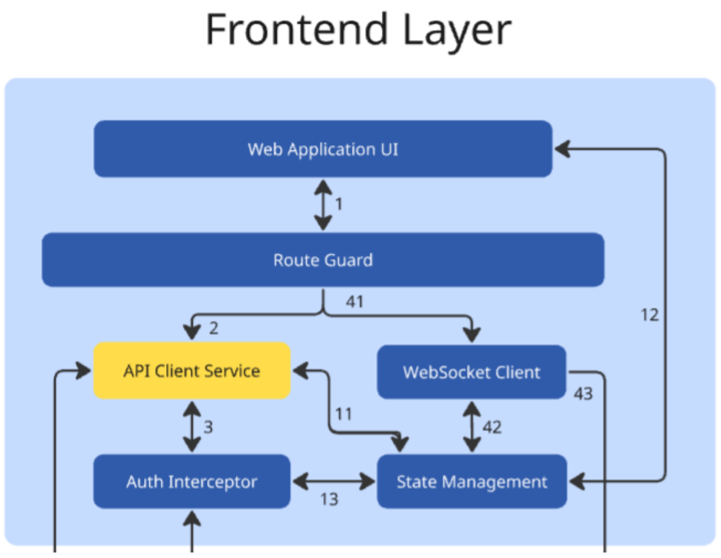

Figure 5: Web Application UI Subsystem

4.3.1 Assumptions

- All API requests from Route Guard are properly formatted before being sent to Auth Interceptor.

- The API Client Service handles both REST API and GraphQL request formats.
- Response data from the backend is valid JSON and can be parsed and stored in State Management.
- Error responses from the backend are properly handled and communicated back to the UI.
- The service can handle file uploads (multipart/form-data) for document submissions.

4.3.2 Responsibilities

- Prepares user data, forms, and file uploads into proper HTTP request structures with correct headers and body content.
- Builds the correct API URLs for actions such as login and application submission.
- Sends all outgoing requests to the Auth Interceptor so authentication tokens can be attached.
- Parses responses returned from the backend and extracts useful data or error messages.
- Sends successful response data to State Management for storage and UI changes.
- Interprets error responses and forwards error details to State Management for display.
- Manages multipart form encoding and ensures files are correctly prepared before sending them through Auth Interceptor.

4.3.3 Subsystem Interfaces

Table 4: API Client Service Subsystem Interfaces

ID	Description	Inputs	Outputs
#2	Receives API requests from Route Guard	Login Credentials Form Data File Uploads API Request Intent	Formatted HTTP Request Request Metadata
#3	Bidirectional communication with Auth Interceptor for authenticated requests	Formatted API Request Request Headers	Authenticated Request with JWT Token HTTP Response Error Response
#10	Sends requests to Internal API Gateway via Auth Interceptor	Authenticated HTTP Request	N/A
#11	Sends response data and state updates to State Management	Authentication Token User Data (id, email, role) Application Data Success/Error Status	N/A

4.4 Auth Interceptor

The Auth Interceptor subsystem manages authentication token injection for all outgoing HTTP requests. It acts as a middleware layer between the API Client Service and the backend, intercepting requests to determine whether authentication credentials are required. For protected endpoints, the Auth Interceptor retrieves the JWT token from State Management and attaches it to the request's Authorization header in Bearer token format, enabling the backend to validate the user's identity and permissions. Public endpoints such as login and registration are allowed to pass through without token attachment, as users have not yet been authenticated.

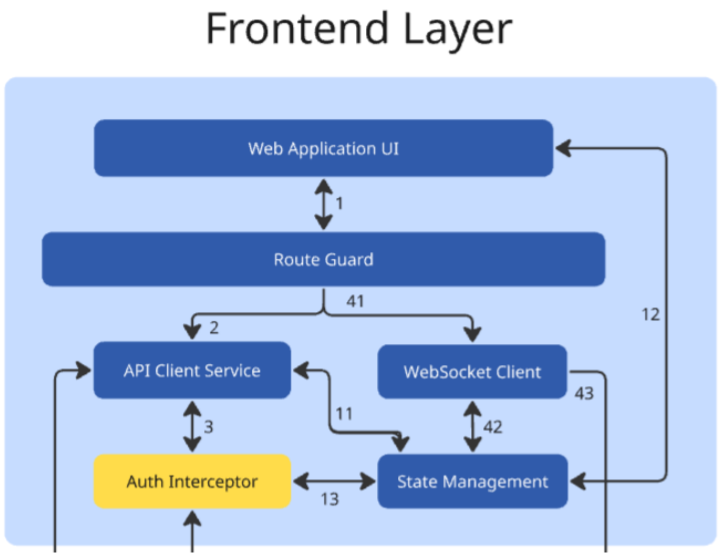

Figure 6: Auth Interceptor subsystem

4.4.1 Assumptions

- All authenticated API requests require a valid JWT token in the Authorization header (Bearer token format).
- Login and registration endpoints do not require authentication tokens and should be allowed to pass through without token attachment.
- JWT tokens stored in State Management are valid and not expired (token expiration is handled by the backend).
- The interceptor can distinguish between public endpoints (login, register) and protected endpoints (application submission, dashboard data).
- If no token is available for a protected endpoint, the request should be rejected or the user should be redirected to login.

4.4.2 Responsibilities

- Intercepts all HTTP requests from API Client Service before they are sent to the backend.
- Determines whether the requested endpoint requires authentication based on the endpoint URL or request metadata.

- Queries State Management to retrieve the current JWT token for authenticated requests.
- Attaches the JWT token to the Authorization header in Bearer token format for protected endpoints.
- Allows login and registration requests to pass through without token attachment.
- Sends the authenticated or unauthenticated request to the backend using REST API or GraphQL.
- Forwards backend responses back to API Client Service for processing.
- Coordinates with State Management to refresh expired tokens if the backend returns a 401 response.

4.4.3 Subsystem Interfaces

Table 5: Auth Interceptor Subsystem Interfaces

ID	Description	Inputs	Outputs
#3	Bidirectional communication with API Client Service for request/response handling	Formatted HTTP Request Request Headers Endpoint URL	Authenticated Request with JWT Token HTTP Response Error Response
#4	Sends authenticated requests to backend External API Gateway via REST API or GraphQL	HTTP Request with Authorization Header Request Body Request Method	HTTP Response Status Code Response Body Error Response
#13	Bidirectional communication with State Management to retrieve JWT token for authenticated requests	Token Retrieval Request Authentication Status Query	JWT Token Token Expiration Status Authentication State

4.5 WebSocket Client

The WebSocket Client subsystem maintains a real-time, two-way connection with the backend server using WebSocket technology. Unlike regular HTTP requests that happen once and close, WebSocket stays open so the backend can send instant updates, notifications, and messages to the frontend without the user needing to refresh the page. This subsystem receives real-time notifications from the Communications Service in the backend and sends them to State Management so the UI can show popup messages and alerts immediately.

4.5.1 Assumptions

- The user's browser supports WebSocket connections and can maintain a persistent connection with the backend server.

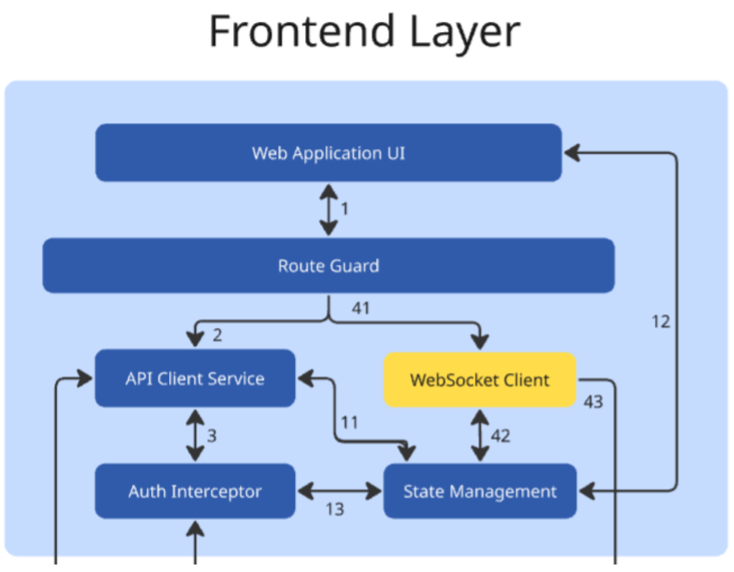

Figure 7: WebSocket Client Subsystem

- The WebSocket connection is established after the user logs in and remains active while the user is on the website.
- If the WebSocket connection drops, the client will try to reconnect automatically.
- Messages received from the backend are in a format that can be understood and processed by the frontend.
- The WebSocket connection uses the same authentication token (JWT) as regular API requests to verify the user's identity.

4.5.2 Responsibilities

- Establishes and maintains a WebSocket connection with the backend server when the user logs in.
- Receives instant messages, notifications, and updates from the backend Communications Service through the WebSocket connection.
- Processes incoming WebSocket messages, extracts notification content, and prepares the data for State Management.
- sends real-time notifications and updates to State Management for immediate UI rendering.
- Checks the health of the WebSocket connection and automatically reconnects if it disconnects.
- Includes the user's JWT token during WebSocket initialization to ensure only authorized users receive messages.
- Converts incoming WebSocket data into a consistent structure usable by other frontend subsystems.

4.5.3 Subsystem Interfaces

Table 6: WebSocket Client Subsystem Interfaces

ID	Description	Inputs	Outputs
#42	Bidirectional communication with State Management for real-time updates	Notification Messages Real-Time Updates Connection Status	Notification Data Update Events Connection State
#43	Sends WebSocket communication to backend Internal API Gateway via Websocket protocol	WebSocket Connection Request Authentication Token	WebSocket Message Notification Payload Event Data
#41	Receives WebSocket connection requests from Route Guard	WebSocket Connection Intent Authentication Token	Connection Status Connection Error

4.6 State Management

The State Management subsystem acts as the central storage and information hub for the entire frontend application. It stores all important data that the frontend needs to remember, such as whether the user is logged in, their authentication token, user information (like email and role), application data, and real-time notifications. When any part of the frontend needs to know the current state of the application, it asks State Management. When new data arrives from the backend or WebSocket, State Management stores it and notifies the UI to update what the user sees.

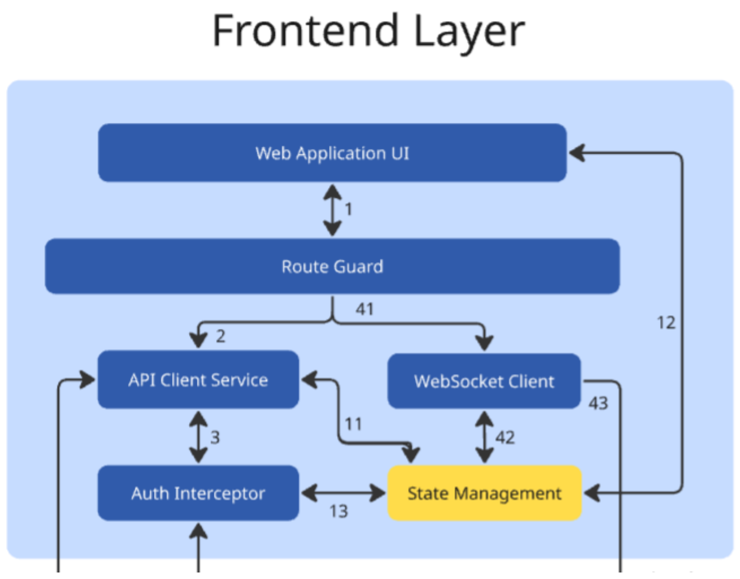

Figure 8: State Management Subsystem

4.6.1 Assumptions

- State Management stores data in the browser's memory and can also use browser storage (like localStorage) to persist data across page refreshes.
- All frontend subsystems trust State Management as the single source of truth for application state.
- When State Management updates, it automatically notifies all subsystems that are listening for changes.
- Authentication tokens and user data stored in State Management are kept secure and not exposed to other websites or scripts.
- State Management can handle multiple simultaneous requests from different subsystems without conflicts.

4.6.2 Responsibilities

- Stores the user's authentication state, including login status and JWT token, so other subsystems can verify access.
- Manages user information such as user ID, email, role, and profile data.
- Provides the stored JWT token to Auth Interceptor when needed for authenticated API requests.
- Stores application records, form data, and other backend response data for use across the UI.
- Maintains a queue of notifications received from WebSocket Client for real-time updates.
- Notifies Web Application UI and other subsystems when stored data changes, triggering re-renders and UI updates.
- Responds to queries from Route Guard, Auth Interceptor, and others regarding authentication status, user role, and stored data.
- Saves important data, such as authentication tokens, to browser storage to preserve login sessions across browser restarts.

4.6.3 SUBSYSTEM INTERFACES

Table 7: State Management Subsystem Interfaces

ID	Description	Inputs	Outputs
#11	Receives data from API Client Service after successful API responses	Authentication Token User Data (id, email, role) Application Data Success/Error Status	N/A
#12	Bidirectional communication with Web Application UI for state updates	N/A	Authentication Status User Data Application State Notification Messages
#13	Receives authentication token requests from Auth Interceptor	Token Retrieval Request Authentication Status Query	JWT Token Token Expiration Status Authentication State
#42	Bidirectional communication with WebSocket Client for real-time updates	Notification Messages Real-Time Updates Connection Status	Notification Data Update Events Connection State

5 Data Layer Subsystems

The Data Layer is responsible for all data storage and database interactions in the system. It lets other layers (Frontend and Backend) work with data without needing to know where or how it is stored. The Data Layer includes the following subsystems:

1. GraphQL Gateway
2. Repository Manager
3. MongoDB Atlas Service
4. GCP Datastore Service
5. Google Cloud Storage

Each subsystem is described below.

5.1 GraphQL Gateway

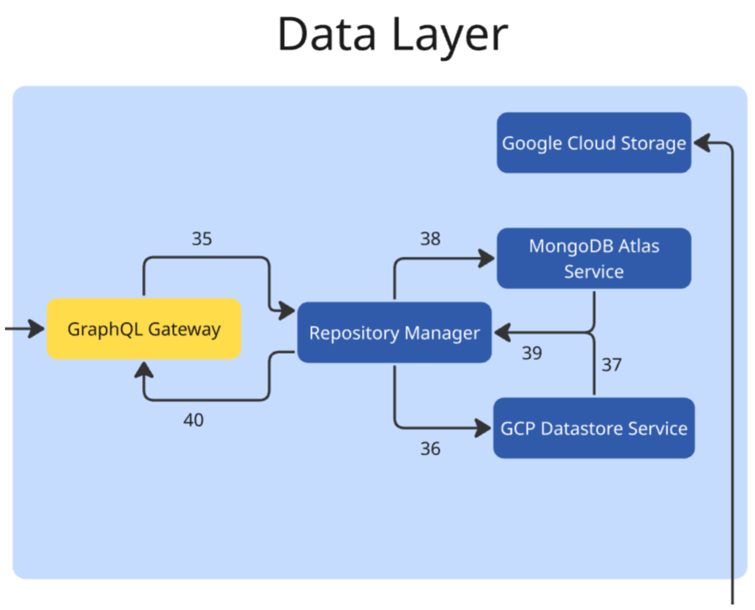

Figure 9: GraphQL Gateway Subsystem

The GraphQL Gateway subsystem provides a unified API interface for data access across all system layers. It receives GraphQL queries and mutations from client applications and backend services, validates them against the defined schema, and routes them to the Repository Manager for execution. This subsystem acts as the primary data access layer, abstracting the underlying storage mechanisms and providing a consistent, type-safe interface for all data operations throughout the MavHousing system.

5.1.1 Assumptions

- Other layers only use GraphQL or REST/GraphQL to request data.
- The GraphQL schema is managed here so updates dont break clients.
- The gateway never talks directly to a databaseâalways through the Repository Manager.
- Security checks are done before requests arrive here.

5.1.2 Responsibilities

- Understand and process GraphQL requests.
- Check if requests match the schema.
- Send requests to the Repository Manager.
- Combine results from different sources if needed.
- Return clear responses to the caller.
- Convert errors into GraphQL error messages.

5.1.3 Subsystem Interfaces

Table 8: GraphQL Gateway Interfaces

ID	Description	Inputs	Outputs
34	Receives GraphQL requests from Internal API Gateway	GraphQL Query GraphQL Mutation Request Context	Query Validation Schema Check
35	Request from higher layers (GraphQL query/mutation)	GraphQL Query GraphQL Mutation	Resolver Requests to Repository Manager
40	Response back to higher layers	Data Objects from Repository Manager	GraphQL Response Payload

5.2 Repository Manager

The Repository Manager subsystem coordinates all data persistence operations across the MavHousing platform. It implements the repository pattern to abstract data access logic, determining the appropriate storage service based on data type and access patterns. This subsystem routes requests to MongoDB Atlas for document-based data, GCP Datastore for relational entities, or Google Cloud Storage for binary files, then aggregates and returns results in a consistent format. By encapsulating storage implementation details, the Repository Manager allows other system components to perform data operations without direct knowledge of the underlying database technologies or cloud storage infrastructure.

5.2.1 Assumptions

- Data models are defined here.
- Each storage service is chosen for a specific type of data.
- The Repository Manager keeps responses consistent.
- Only valid and checked requests reach this subsystem.

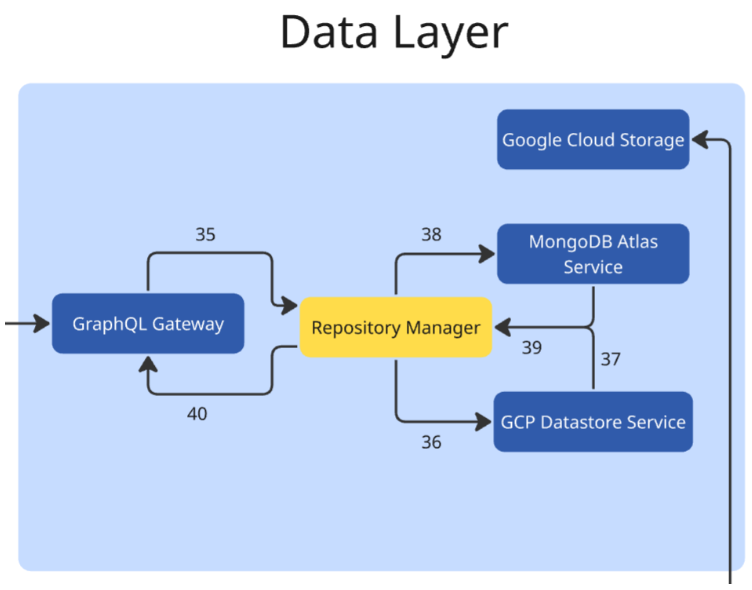

Figure 10: Repository Manager Subsystem

5.2.2 Responsibilities

- Handle all data storage and retrieval requests.
- Decide which storage service to use.
- Convert requests into the right kind of database query.
- Manage caching and retries if needed.
- Handle errors and try backups or fallback options.

5.2.3 Subsystem Interfaces

Table 9: Repository Manager Interfaces

ID	Description	Inputs	Outputs
35	Requests from GraphQL Gateway	Resolver Call Model Lookup	Normalized Dataset
36	Interaction with GCP Datastore	Datastore Query	Entity Records
38	Interaction with MongoDB Atlas	Mongo Query	Document Result Set
39	Routing/aggregation result	Mongo + Datastore responses	Composite Data Model
40	Sends aggregated results to GraphQL Gateway	N/A	Normalized Dataset Composite Data Model

5.3 MongoDB Atlas Service

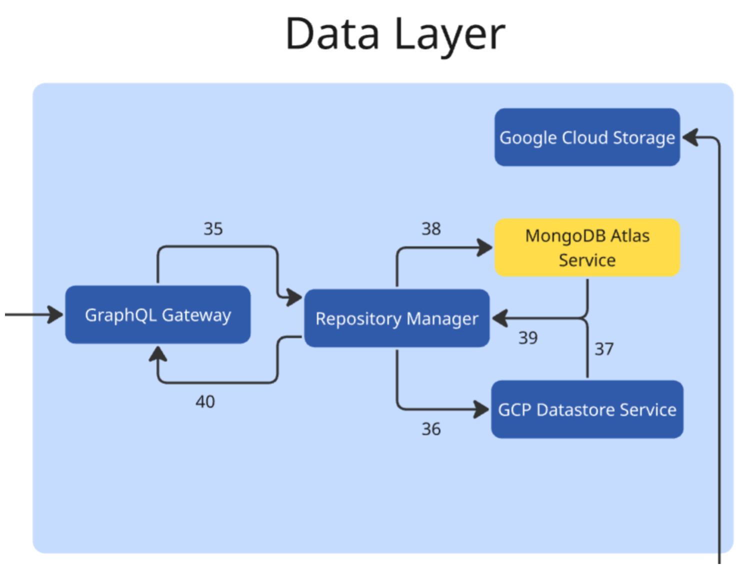

Figure 11: MongoDB Atlas Service

The MongoDB Atlas Service subsystem provides NoSQL document storage for semi-structured and dynamic data such as user profiles, application logs, and audit trails. It utilizes Mongoose as the Object Document Mapper (ODM) to define schemas, validate data models, and execute queries against MongoDB collections. This service excels at storing data with flexible schemas that may evolve over time, offering horizontal scalability and high availability through MongoDB Atlas's managed cloud infrastructure. The document-oriented model supports complex nested structures and enables efficient querying of unstructured or variably-structured data.

5.3.1 Assumptions

- Used for semi-structured, document-based data.
- Backups and copies are managed by MongoDB Atlas.
- Most operations are for single documents.
- Repository Manager checks data format before saving.

5.3.2 Responsibilities

- Store, update, and fetch documents.
- Run searches and aggregation queries.
- Keep data safe with backups.
- Scale for high availability and performance.

5.3.3 Subsystem Interfaces

Table 10: MongoDB Atlas Service Interfaces

ID	Description	Inputs	Outputs
38	Read/write requests from Repository Manager	Mongo Query Document Insert/Update	Query Results Write Status
39	Exchanges data with Repository Manager	Data Query	Query Result

5.4 GCP Datastore Service

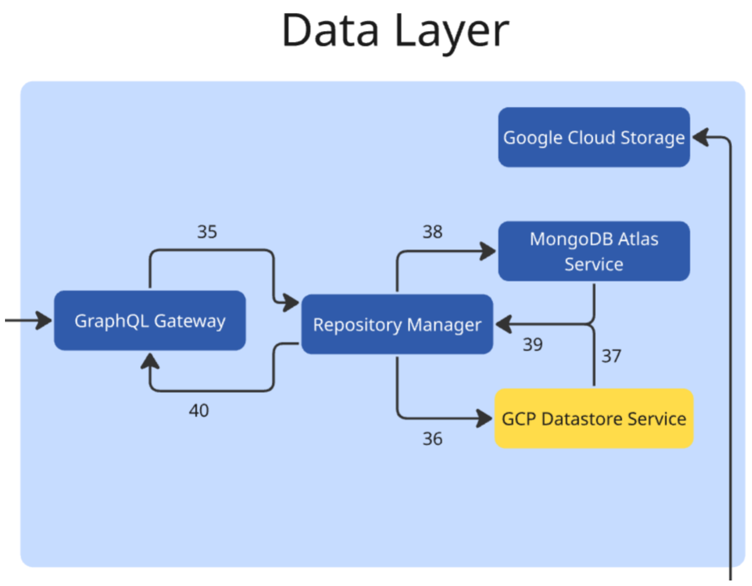

Figure 12: GCP Datastore Service Subsystem

The GCP Datastore Service subsystem manages structured relational data using PostgreSQL as the underlying database engine. It employs TypeORM as the Object-Relational Mapper (ORM) to define entity models, manage database migrations, and execute type-safe queries. This service is optimized for transactional data requiring strong consistency guarantees, such as user accounts, housing applications, room assignments, and payment records.

5.4.1 Assumptions

- Stores well-structured, simple data.
- Reads are always correct and up-to-date.
- Big, multi-row transactions are avoided.

5.4.2 Responsibilities

- Provide fast and reliable data lookups.

- Keep data organized for quick access.
- Store important configuration data.
- Support simple updates.

5.4.3 Subsystem Interfaces

Table 11: GCP Datastore Service Interfaces

ID	Description	Inputs	Outputs
36	Structured entity fetch/store via Repository Manager	Key Lookup Entity Save	Entity Data Operation Status
37	Service output flow toward Repository Manager	Entity Records	Final Data Objects

5.5 Google Cloud Storage Service

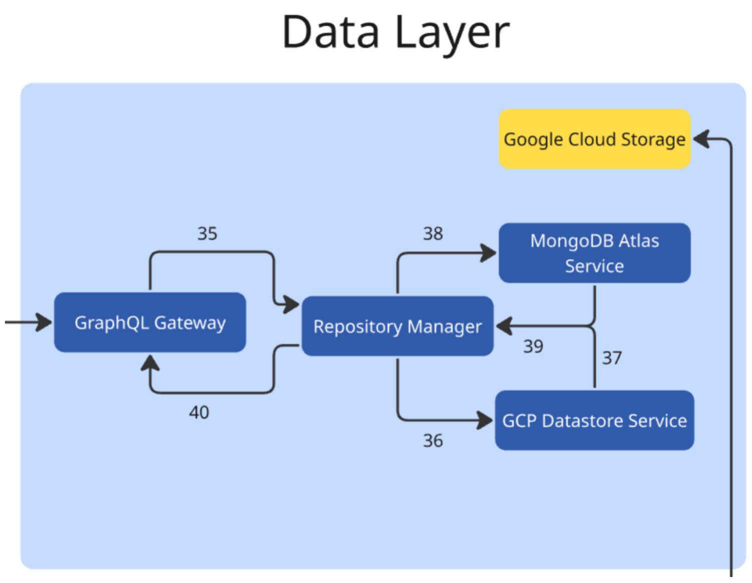

Figure 13: Google Cloud Storage Service Subsystem

The Google Cloud Storage Service subsystem handles binary large object (BLOB) storage for files, images, documents, and media assets that do not fit structured database models. It manages file uploads, downloads, deletions, and lifecycle policies for documents such as housing application attachments, maintenance request photos, lease agreements, and system backups. This service provides scalable, durable object storage with configurable access controls, automatic replication for high availability, and integration with the Repository Manager for metadata tracking and secure URL generation.

5.5.1 Assumptions

- Used for storing files and large blobs of data.
- Repository Manager handles file metadata, while files are stored and accessed directly.

- Access control is managed by Google Cloud permissions.

5.5.2 Responsibilities

- Store, retrieve, and delete files and blobs.
- Manage file lifecycles and automatic cleanup.
- Keep files available and reliable.
- Support logging and integration with other services.

5.5.3 Subsystem Interfaces

Table 12: Google Cloud Storage Interfaces

ID	Description	Inputs	Outputs
29	File management and storage operations	File Upload Request File Metadata	Blob Handle File Download Stream

6 Backend Layer Subsystems

The Backend Layer manages all the core business logic, API routing, background jobs, external communication, and interactions with both storage and the outside world. It connects the frontend and data layers, performing tasks like authentication, notifications, payment processing, file management, and communication with AI services.

The main subsystems are:

1. External API Gateway
2. Auth Guard
3. Role Based Access Control (RBAC) and Route Guard
4. Domain Services
5. Internal API Gateway
6. Database Service
7. Cloud Storage Controller
8. File Management Service
9. Payment Gateway
10. Payment Service
11. Notification Manager
12. Email SMS Listener Service
13. Event Bus
14. Event Handler Registry
15. Cron Worker
16. Roommate-Matcher Engine
17. Queue Worker Service
18. Communications Service
19. Gemini LLM Client
20. Blaze AI
21. MCP Server

Each subsystem is described below.

6.1 External API Gateway

Handles incoming API requests from outside the system. It forwards requests to the proper internal services and ensures only valid traffic is allowed.

6.1.1 Assumptions

- Only trusted sources reach this gateway.
- API keys or authentication tokens are required.

6.1.2 Responsibilities

- Accept and route external requests.
- Enforce rate limiting and logging.
- Forward requests to the Auth Guard and RBAC.

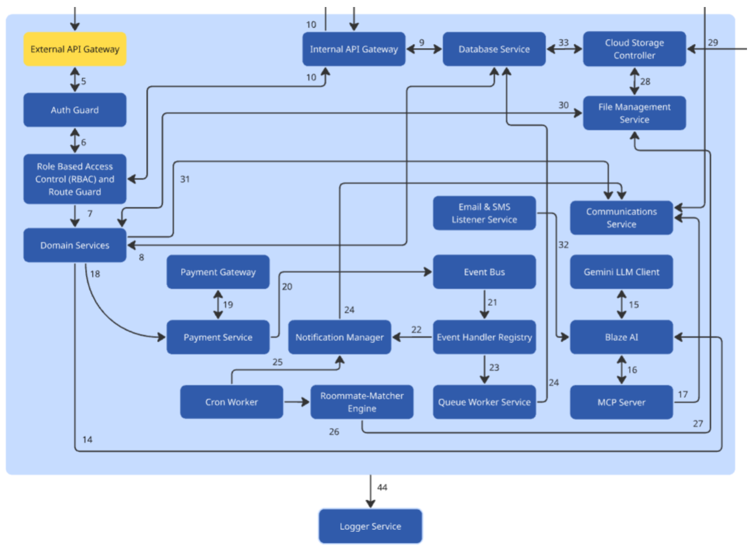

Figure 14: External API Gateway Subsystem

Table 13: External API Gateway Interfaces

ID	Description	Inputs	Outputs
4	Receives API requests from frontend through Auth Interceptor	Rest API / GraphQL Request	Authenticated Request
5	Checks authentication with Auth Guard	Authenticated Request	Verification Result

6.2 Auth Guard

Protects the system by checking user authentication and permissions before letting requests go further.

6.2.1 Assumptions

- Works with tokens or session data.
- Returns errors for unauthenticated users.

6.2.2 Responsibilities

- Verify authentication on every request.
- Block or pass requests to RBAC.

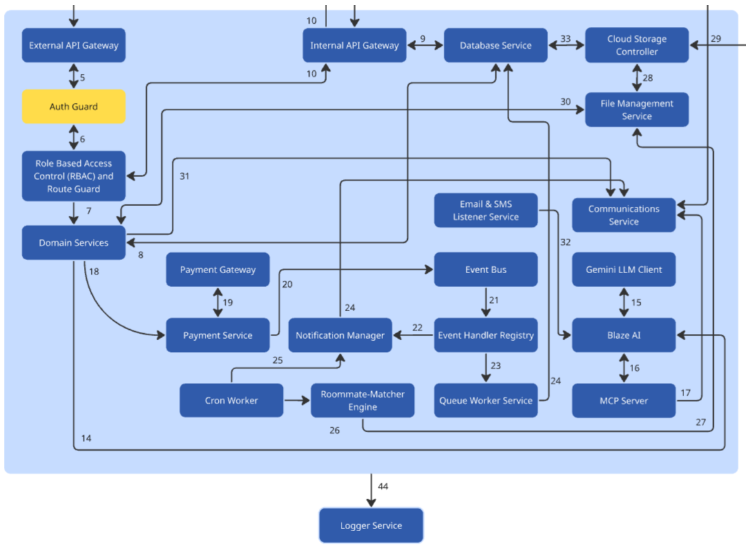

Figure 15: Auth Guard Subsystem

Table 14: Auth Guard Interfaces

ID	Description	Inputs	Outputs
5	Verifies authentication with External API Gateway	API Request	Authentication Result
6	Checks authorization with RBAC	Authenticated Request	Authorization Request

6.3 Role Based Access Control (RBAC) and Route Guard

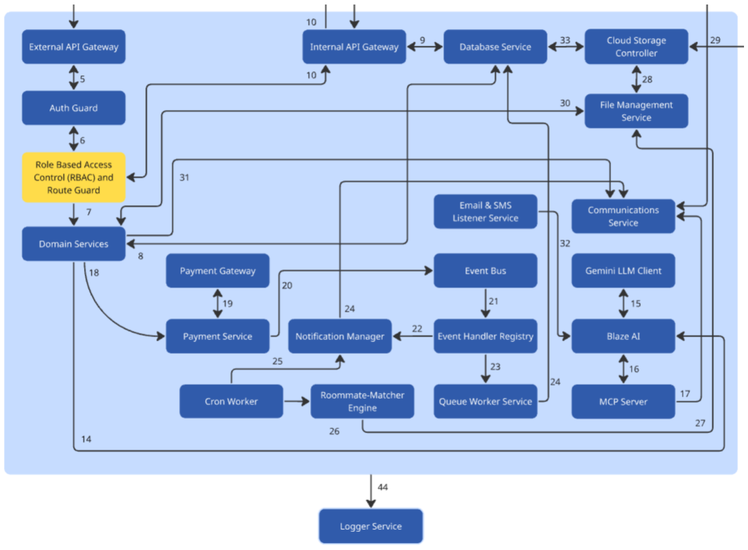

Figure 16: Role Based Access Control (RBAC) and Route Guard Subsystem

The RBAC and Route Guard subsystem enforces authorization policies by validating user roles and permissions against requested resources. It evaluates each authenticated request to ensure users can only access features and data appropriate to their assigned role (student, staff, or administrator).

6.3.1 Assumptions

- User roles are defined in the system.

6.3.2 Responsibilities

- Enforce access policies.
- Allow or deny requests based on user role.

Table 15: RBAC and Route Guard Interfaces

ID	Description	Inputs	Outputs
6	Exchanges authorization with Auth Guard	Authorization Request	Role Check
7	Sends authorized requests to Domain Services	Role Check	Authorized Request
10	Forwards authorized requests to Internal API Gateway	Authorized Request	Rest API / GraphQL Request

6.4 Domain Services

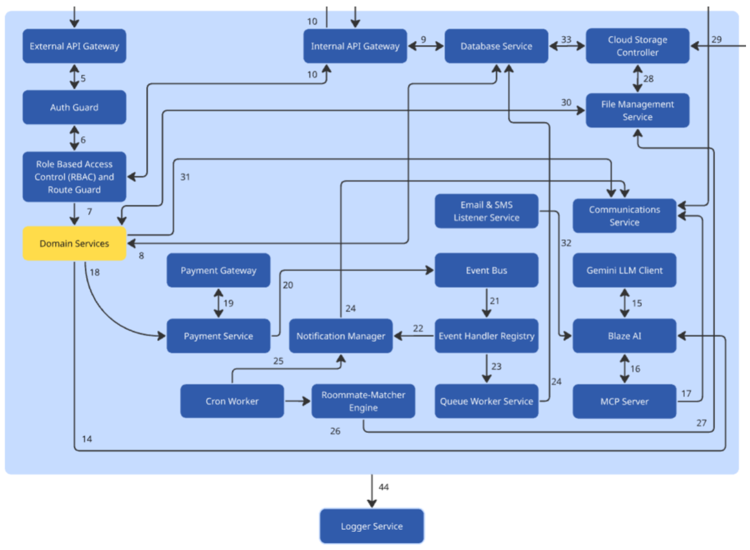

Figure 17: Domain Services Subsystem

The Domain Services subsystem encapsulates the core business logic and rules governing housing operations, including application processing, room assignment workflows, maintenance request handling, and payment management. It coordinates interactions between multiple backend subsystems to execute complex operations while maintaining business rule consistency and application state integrity.

6.4.1 Assumptions

- Business rules and housing policies are configurable and can be modified by administrators without requiring code changes.
- All incoming requests have been authenticated and authorized by Auth Guard and RBAC before reaching Domain Services.
- Database operations performed by Domain Services maintain transactional integrity through the Database Service layer.
- Domain Services act as the single source of truth for all business logic, with no business rules duplicated in other subsystems

6.4.2 Responsibilities

- Perform main application operations.
- Act as a central logic point for other subsystems.

Table 16: Domain Services Interfaces

ID	Description	Inputs	Outputs
7	Receives authorized requests from RBAC	Authorized Request	Business Request
8	Sends requests to Internal API Gateway	Business Request	Internal Request
14	Sends events to Event Handler Registry	Business Event	Event Message
31	Sends file operations to File Management Service	Business Request	File Request
30	Exchanges file operations with Cloud Storage Controller	File Operation Request	Storage Request
18	Sends payment requests to Payment Gateway	Business Request	Payment Request

6.5 Internal API Gateway

The Internal API Gateway subsystem exposes internal backend APIs to external layers, specifically providing REST and GraphQL endpoints for the Frontend Layer and Data Layer to communicate with backend services. It routes incoming requests from the API Client Service and GraphQL Gateway to the appropriate Domain Services, Database Service, or other backend subsystems.

6.5.1 Assumptions

- All requests from the Frontend Layer pass through the External API Gateway for authentication before being routed here.
- The Internal API Gateway provides a stable interface contract, isolating external clients from internal backend implementation changes.
- API versioning is supported to maintain backward compatibility as the system evolves.

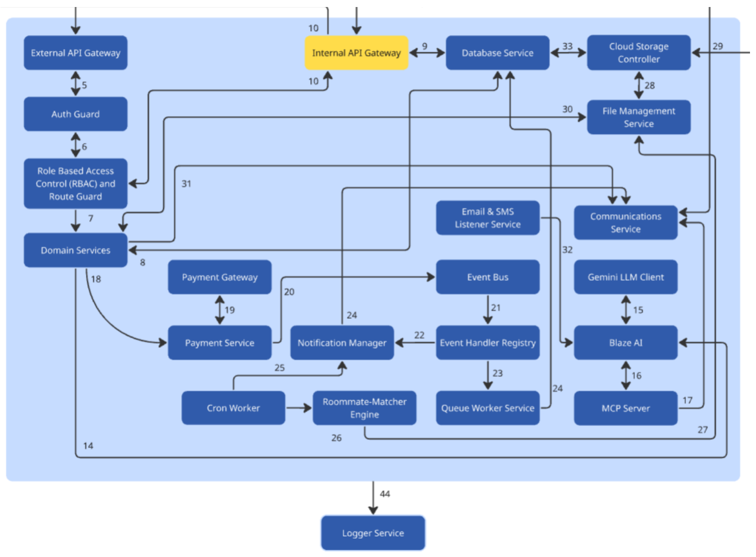

Figure 18: Internal API Gateway Subsystem

6.5.2 Responsibilities

- Expose REST and GraphQL endpoints for frontend and data layer access to backend functionality.
- Route incoming API requests to the appropriate backend subsystem based on endpoint and request type.
- Transform request and response formats between external API contracts and internal service interfaces.
- Track and log all API calls for monitoring, debugging, and audit purposes.
- Implement rate limiting and request throttling to protect backend services from overload.

Table 17: Internal API Gateway Interfaces

ID	Description	Inputs	Outputs
9	Sends queries to Database Service	Routing Request	Data Query
10	Receives requests from External API Gateway	Rest API / GraphQL Request	Routing Request
34	Sends data requests to GraphQL Gateway	Query Request	Rest API / GraphQL Request

6.6 Database Service

The Database Service subsystem provides a unified interface for accessing structured data stored in PostgreSQL and MongoDB databases. It abstracts database-specific implementation details from other

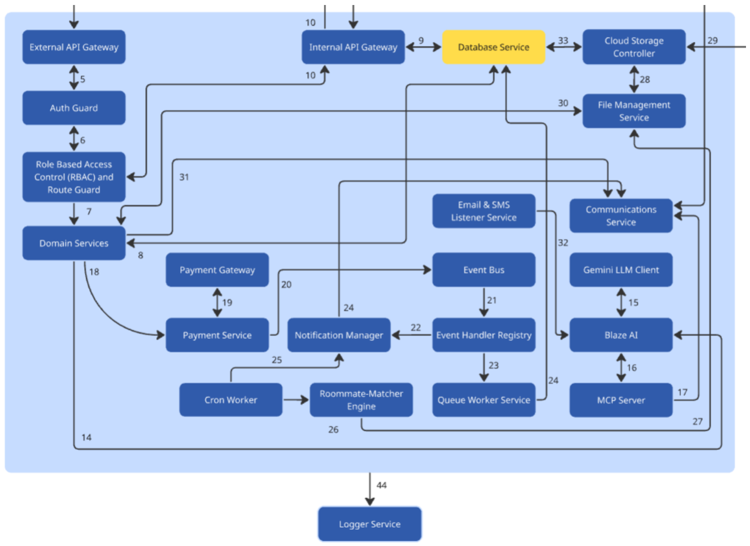

Figure 19: Database Service Subsystem

backend services, offering a consistent API for executing queries, transactions, and data manipulation operations across both relational and document-based storage systems.

6.6.1 Assumptions

- Data models and entity schemas are shared between the Database Service, Repository Manager, and Domain Services through a common data layer definition.
- All database connections are pooled and managed efficiently to handle concurrent requests from multiple backend subsystems.
- Transactional operations maintain ACID properties for relational data in PostgreSQL and eventual consistency for document data in MongoDB.
- Database credentials and connection strings are securely stored in environment variables and accessed through configuration management.

6.6.2 Responsibilities

- Execute data queries from Domain Services, Internal API Gateway, and other backend subsystems against PostgreSQL for structured data and MongoDB for logs and semi-structured data.
- Connect with storage and data layers.
- Coordinate with the Repository Manager in the Data Layer to abstract storage implementation details from business logic.

Table 18: Database Service Interfaces

ID	Description	Inputs	Outputs
8	Receives data queries from Domain Services	Data Query Request	Query Result
9	Exchanges data queries with Internal API Gateway	Data Query	Query Result
24	Receives events from Event Bus	Event	Database Update
33	Exchanges data with Cloud Storage Controller	Storage Request	Storage Response

6.7 Cloud Storage Controller

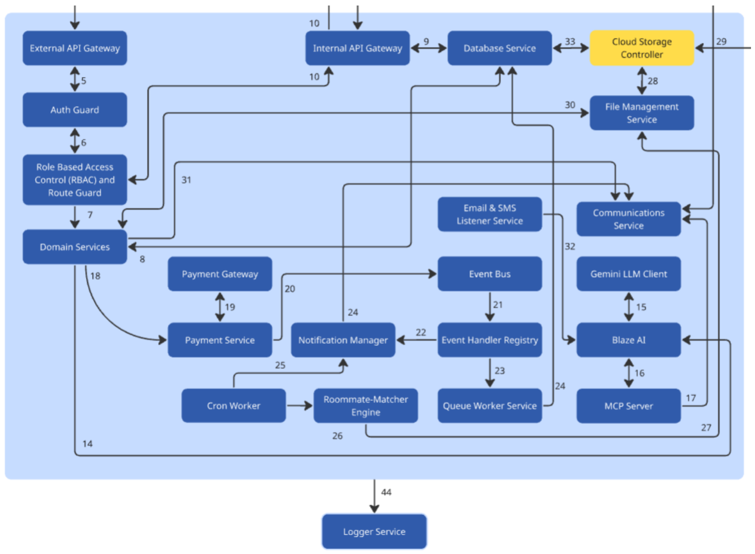

Figure 20: Cloud Storage Controller Subsystem

The Cloud Storage Controller subsystem manages interactions with Google Cloud Storage for file and blob storage operations. It handles authentication, request formatting, and API communication with GCS, providing backend services with a simplified interface for uploading, downloading, and managing files without directly handling cloud provider SDK complexity.

6.7.1 Assumptions

- Connected to cloud storage providers.
- Network connectivity to Google Cloud Storage endpoints is reliable with automatic retry logic for transient failures.
- Google Cloud Storage buckets are pre-configured with appropriate access controls, lifecycle policies, and regional redundancy settings.

6.7.2 Responsibilities

- Handle file storage operations.

- Upload files to Google Cloud Storage buckets, generating unique object keys and returning public or signed URLs for access.
- Download files from GCS by retrieving objects based on their storage keys and streaming content to requesting services.
- Generate pre-signed URLs with expiration times for secure, temporary access to private files such as student documents or lease agreements.
- Forward file operation requests and metadata to the File Management Service for business logic processing and database record updates.

Table 19: Cloud Storage Controller Interfaces

ID	Description	Inputs	Outputs
28	Receives file storage requests	File Storage Request	Storage Response
33	Exchanges data with Database Service	Storage Request	Storage Response
29	Uploads files to Google Cloud Storage	File Data	Upload Confirmation

6.8 File Management Service

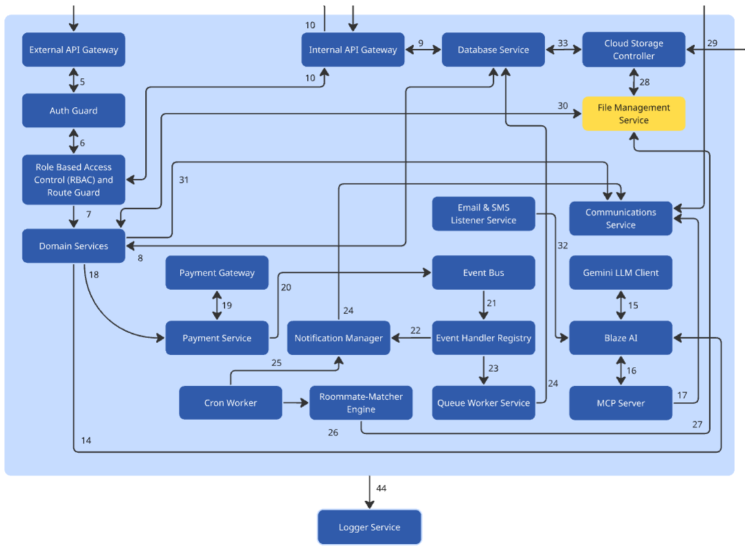

Figure 21: File Management Service Subsystem

The File Management Service subsystem coordinates the complete lifecycle of files and documents within the MavHousing system, including housing applications, maintenance request attachments, lease agreements, and identification documents. It manages file metadata, enforces access permissions, validates file types and sizes, and maintains database records linking files to their associated entities.

6.8.1 Assumptions

- All file operations are coordinated through the Cloud Storage Controller, which handles the actual storage layer interactions.
- EXIF Metadata are retained to maintain transparency.
- Validate uploaded files against allowed file types and maximum file limit.

6.8.2 Responsibilities

- Maintain file metadata and access rules.
- Manage file lifecycle (create, update, delete).

Table 20: File Management Service Interfaces

ID	Description	Inputs	Outputs
28	Sends file storage requests to Cloud Storage Controller	File Operation Request	File Storage Request
30	Exchanges file operations with Cloud Storage Controller	File Operation	Storage Request
27	Receives file requests from Domain Services	File Request	File Status

6.9 Payment Gateway

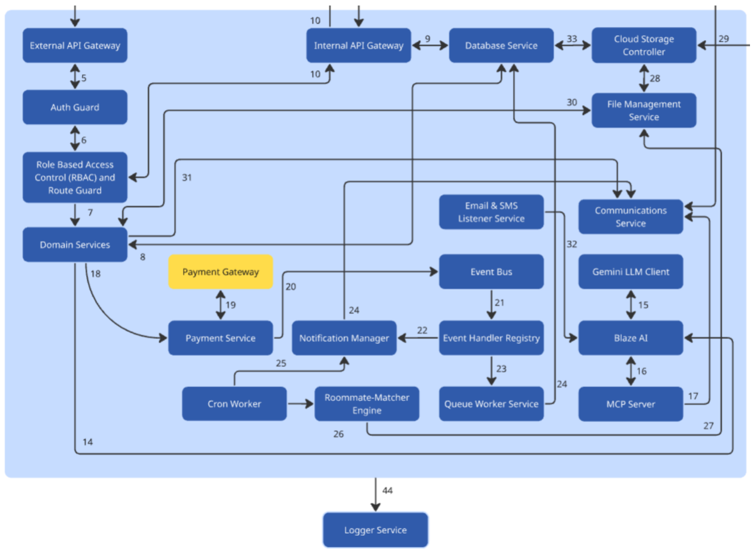

Figure 22: Payment Gateway Subsystem

Handles communication with third-party payment processors for billing and transactions.

6.9.1 Assumptions

- Supports multiple payment providers.

6.9.2 Responsibilities

- Process payment requests.
- Handle callbacks and responses.

Table 21: Payment Gateway Interfaces

ID	Description	Inputs	Outputs
19	Exchanges payment data with Payment Service	Payment Process	Payment Result

6.10 Payment Service

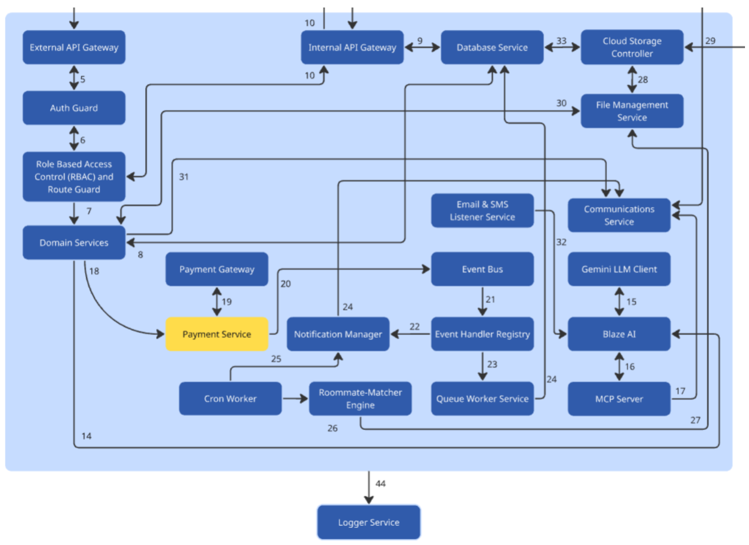

Figure 23: Payment Service Subsystem

Implements the business logic for handling payments, invoices, and refunds.

6.10.1 Assumptions

- Payment methods are securely stored.

6.10.2 Responsibilities

- Validate and record transactions.
- Connect with Payment Gateway and Notification Manager.

Table 22: Payment Service Interfaces

ID	Description	Inputs	Outputs
19	Exchanges payment data with Payment Gateway	Payment Result	Payment Confirmation
20	Sends notifications to Notification Manager	Payment Event	Notification Request
18	Receives payment requests from Domain Services	Payment Request	Payment Process

6.11 Notification Manager

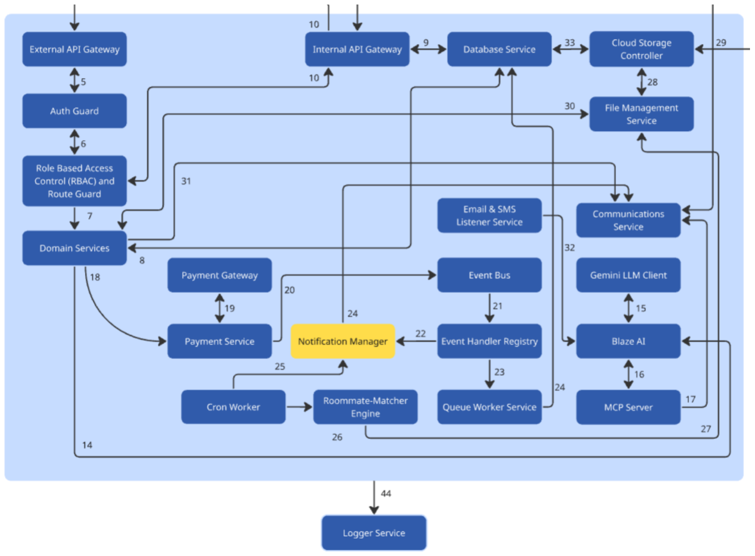

Figure 24: Notification Manager Subsystem

The Notification Manager subsystem orchestrates the delivery of notifications across multiple channels including email, SMS, in-app alerts, and push notifications. It manages notification queuing, priority handling, delivery scheduling, and status tracking to ensure timely and reliable communication with students, staff, and administrators.

6.11.1 Assumptions

- Multiple notification channels are supported with fallback mechanisms if primary channels fail.
- The service coordinates with the Communications Service for actual message delivery across channels.
- Notification templates are configurable and support variable substitution for personalized content.

6.11.2 Responsibilities

- Schedule notification delivery for future times, such as rent reminders sent three days before due dates or move-in checklists sent one week before move-in.

- Coordinate with the Communications Service to execute actual message delivery across email, SMS, WebSocket, and push notification channels.
- Prioritize urgent notifications such as emergency maintenance or security alerts for immediate delivery ahead of routine notifications.
- Aggregate similar notifications to prevent notification fatigue, such as batching multiple maintenance updates into a daily digest.

Table 23: Notification Manager Interfaces

ID	Description	Inputs	Outputs
24	Sends events to Event Bus	Notification Process	Event
22	Receives notification tasks from Event Handler Registry	Notification Task	Notification Request
25	Receives scheduled notifications from Cron Worker	Scheduled Task	Notification Request

6.12 Email & SMS Listener Service

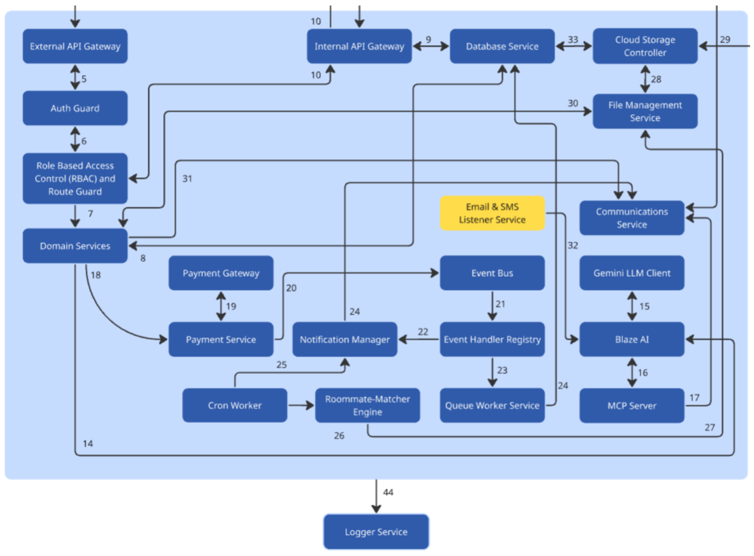

Figure 25: Email & SMS Listener Service Subsystem

Listens for inbound email and SMS messages, then routes them to appropriate services.

6.12.1 Assumptions

- Connected to email/SMS providers.

6.12.2 Responsibilities

- Receive and process incoming messages.

- Notify other services of new messages.

Table 24: Email & SMS Listener Service Interfaces

ID	Description	Inputs	Outputs
32	Routes incoming email/SMS messages to Blaze AI for processing	Incoming Email (External) Incoming SMS (External) Message Metadata	Message Routed User Query Contact Info

6.13 Event Bus

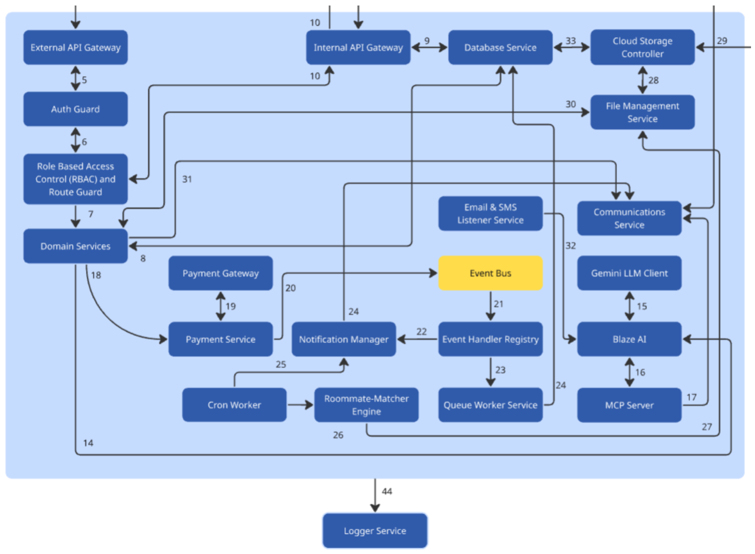

Figure 26: Event Bus Subsystem

The Event Bus subsystem provides a centralized message broker for asynchronous event-driven communication across backend services.

6.13.1 Assumptions

- Accept and queue events from publisher services such as Domain Services, Payment Service, and File Management Service.
- Implement retry mechanisms with exponential backoff for failed event deliveries.

6.13.2 Responsibilities

- Distribute events to subscribers.
- Ensure delivery even if some services are down.

Table 25: Event Bus Interfaces

ID	Description	Inputs	Outputs
20	Receives payment events from Payment Gateway	Payment Event Transaction Status Payment Confirmation	Event Queued Event Log
21	Distributes events to Event Handler Registry for routing	N/A	Event Event Type Event Payload

6.14 Event Handler Registry

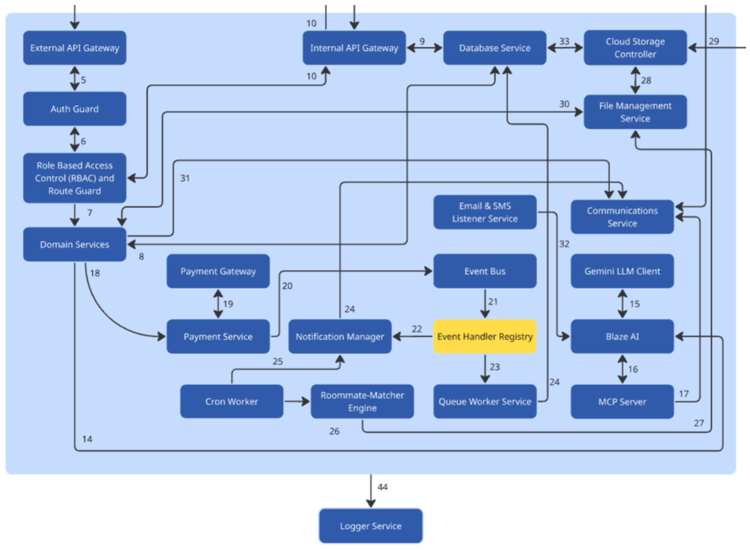

Figure 27: Event Handler Registry Subsystem

The Event Handler Registry subsystem maintains a registry of event handlers and their subscriptions, mapping event types to the services that should process them. It acts as the configuration layer for the event-driven architecture, enabling dynamic registration and deregistration of event handlers without system downtime.

6.14.1 Assumptions

- Handlers can be updated or changed as needed.
- Multiple handlers can subscribe to the same event type for parallel processing.

6.14.2 Responsibilities

- Maintain a registry mapping event types to their corresponding handler services and endpoints.
- Supply the Event Bus with current subscription information for event routing decisions.

Table 26: Event Handler Registry Interfaces

ID	Description	Inputs	Outputs
21	Receives events from Event Bus for handler routing	Event Event Type Event Payload	Handler Mapping Routing Decision
22	Routes notification events to Notification Manager	N/A	Notification Event Handler Invocation Event Forwarding
23	Routes background job events to Queue Worker Service	N/A	Job Event Task Assignment Event Forwarding

6.15 Cron Worker

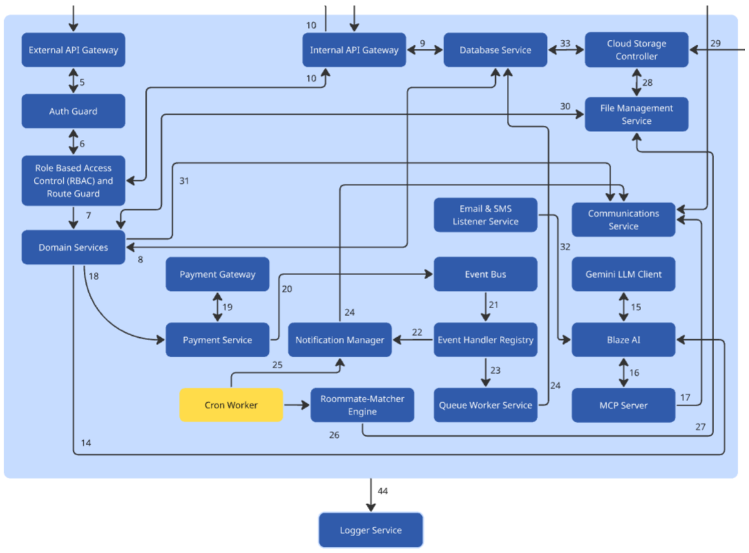

Figure 28: Cron Worker Subsystem

The Cron Worker subsystem executes scheduled background jobs at configured intervals, handling recurring tasks such as rent payment reminders, application deadline notifications, maintenance report generation, and data cleanup operations.

6.15.1 Assumptions

- Scheduling rules are configured through environment variables or administrative interfaces using cron expression syntax.
- Failed jobs are logged with error details and can be manually retried by administrators.
- The Cron Worker has access to all necessary backend services and databases required for scheduled tasks.

- Only one instance of each scheduled job runs at a time to prevent duplicate processing through distributed locking mechanisms.

6.15.2 Responsibilities

- Start tasks at set intervals.
- Monitor job status and results.

Table 27: Cron Worker Interfaces

ID	Description	Inputs	Outputs
25	Sends scheduled notification tasks to Notification Manager	Scheduled Task Time Trigger	Notification Job Reminder Task Deadline Alert
26	Triggers roommate matching jobs to Roommate-Matcher Engine	Scheduled Task Time Trigger	Matching Job Request Batch Processing Task

6.16 Roommate-Matcher Engine

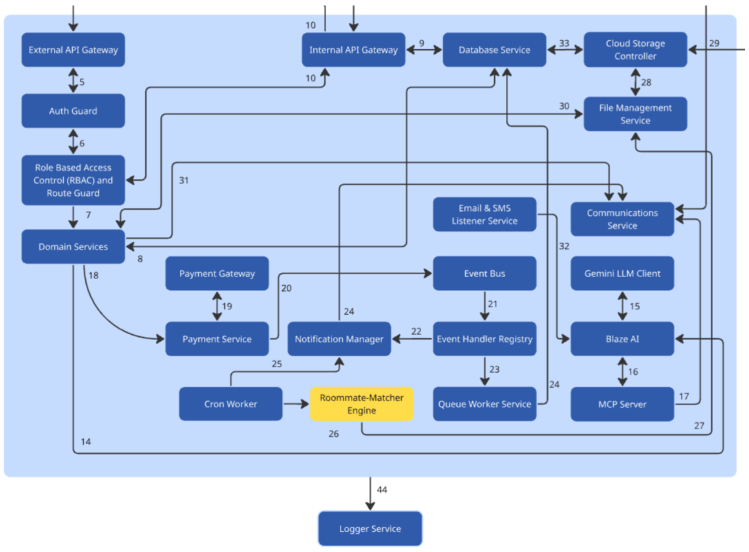

Figure 29: Roommate-Matcher Engine Subsystem

The Roommate-Matcher Engine subsystem is a machine learning model trained on historical housing and roommate pairing data to predict compatibility between students. The model analyzes compatibility factors including lifestyle preferences, sleep schedules, study habits, cleanliness standards, and housing priorities to generate compatibility scores and suggest optimal roommate pairings that maximize satisfaction while respecting policy constraints.

6.16.1 Assumptions

- Uses data from multiple services.
- Administrators can review and override model-suggested matches before final room assignments are confirmed.

6.16.2 Responsibilities

- Match and suggest roommates.
- Update and store match results.

Table 28: Roommate-Matcher Engine Interfaces

ID	Description	Inputs	Outputs
26	Sends match notifications to Notification Manager	Match Result	Notification Request
27	Receives file requests from Domain Services	File Request	File Status

6.17 Queue Worker Service

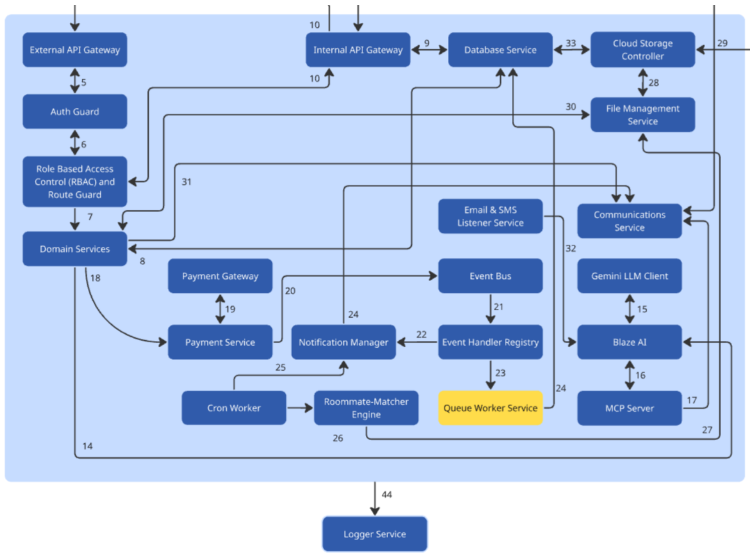

Figure 30: Queue Worker Service Subsystem

Processes jobs or messages from a queue, allowing tasks to be handled in the background.

6.17.1 Assumptions

- Works with multiple queues.

6.17.2 Responsibilities

- Fetch and process queued jobs.
- Report results to Notification Manager or other subsystems.

Table 29: Queue Worker Service Interfaces

ID	Description	Inputs	Outputs
23	Receives jobs from Event Handler Registry	Job Queue Task Message Event Data	Task Result Processing Status
24	Sends processed notifications to Notification Manager	N/A	Notification Request Processed Event Delivery Task

6.18 Communications Service

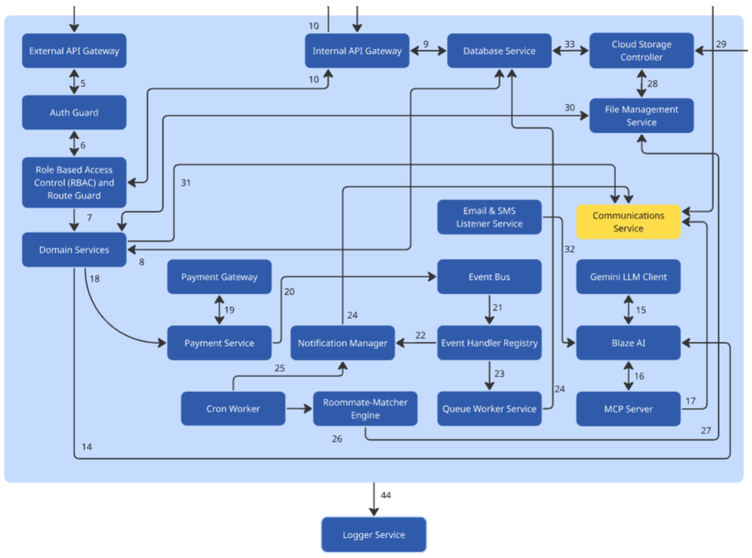

Figure 31: Communications Service Subsystem

The Communications Service subsystem manages all outbound and inbound communications across multiple channels including email, SMS, in-app messaging, and WebSocket push notifications. It provides a unified interface for sending notifications and maintains message delivery logs for audit and troubleshooting purposes.

6.18.1 Assumptions

- Connects to different messaging platforms.
- Message templates support dynamic variable substitution for personalized content.

6.18.2 Responsibilities

- Send and receive communications.
- Log or archive messages as needed.
- Implement rate limiting to prevent abuse and ensure compliance.

Table 30: Communications Service Interfaces

ID	Description	Inputs	Outputs
17	Receives messaging requests from MCP Server	Message Request Tool Response User Context	Email Sent SMS Sent Delivery Status
24	Receives notification delivery requests from Notification Manager	Notification Payload Channel Info User Contact Info	Email Sent SMS Sent Push Notification Sent
31	Receives communication requests from Domain Services	Message Template Event Data Recipient List	Email Sent SMS Sent Delivery Confirmation
43	Receives WebSocket communication requests from WebSocket Client	WebSocket Message Real-time Update Connection Info	WebSocket Message Sent Real-time Notification Delivered

6.19 Gemini LLM Client

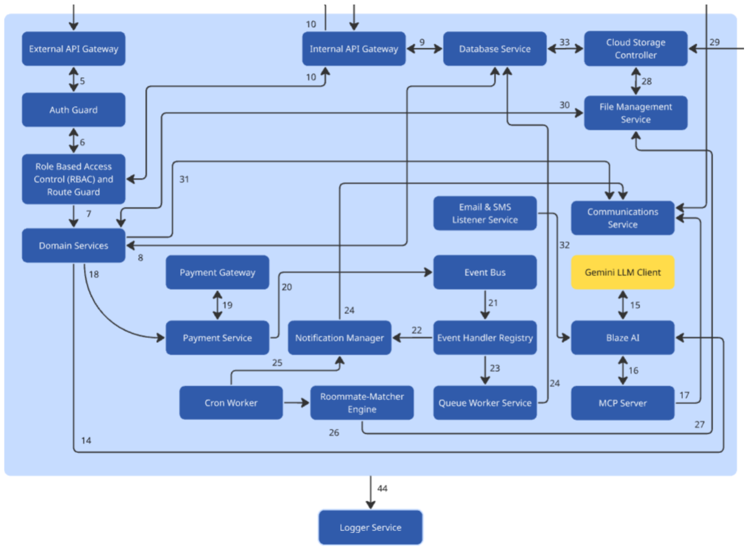

Figure 32: Gemini LLM Client Subsystem

The Gemini LLM Client subsystem provides an interface to Google's Gemini large language model, enabling natural language processing capabilities for the Blaze AI assistant. It manages API authentication, request formatting, response parsing, and error handling for all interactions with the external LLM service.

6.19.1 Assumptions

- Requires API keys for external access.
- API usage is monitored to stay within budget constraints and usage limits.

6.19.2 Responsibilities

- Submit data or queries to the LLM.
- Process and store responses.

Table 31: Gemini LLM Client Interfaces

ID	Description	Inputs	Outputs
15	Interfaces with Gemini LLM API	LLM Request Query Data	LLM Response Result Data

6.20 Blaze AI

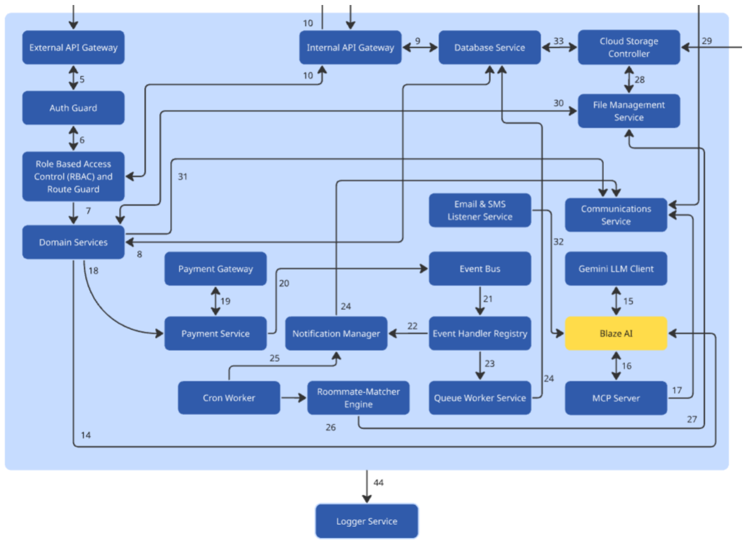

Figure 33: Blaze AI Subsystem

The Blaze AI subsystem implements the intelligent assistant that answers student questions, provides housing information, and automates notifications about deadlines and updates. It combines the Gemini LLM Client for natural language understanding with the MCP Server for secure access to real-time system data through specialized tools.

6.20.1 Assumptions

- The MCP Server provides tools that allow Blaze to query application status, payment history, maintenance requests, and housing availability in real-time.
- User queries are preprocessed to extract intent and relevant entities before being sent to the LLM.
- The system maintains conversation context across multiple exchanges to provide coherent multi-turn interactions.
- AI-generated responses are monitored for accuracy and appropriateness before delivery to users.

6.20.2 Responsibilities

- Process natural language queries from students through web portal, email, or Microsoft Teams integration.
- Determine user intent and extract relevant entities such as application IDs, room numbers, or payment amounts from queries.
- Invoke appropriate MCP tools to retrieve real-time data including application status, room assignments, payment balances, and maintenance ticket updates.
- Escalate complex queries that cannot be answered automatically to housing staff with relevant context.

Table 32: Blaze AI Interfaces

ID	Description	Inputs	Outputs
16	Exchanges data with Gemini LLM Client	LLM Request	LLM Response
14	Receives events from Event Handler Registry	Event	AI Task Request
15	Exchanges data with Communications Service	Message Request	AI Response
32	Receives messages from Email & SMS Listener Service	Incoming Message	Message Processing

6.21 MCP Server

The MCP Server subsystem implements the Model Context Protocol framework, providing Blaze AI with secure, structured access to backend system data through a collection of specialized tools. Each tool exposes specific functionality such as querying application status, retrieving payment information, or fetching maintenance request details, enabling the AI assistant to provide accurate, context-aware responses based on real-time data.

6.21.1 Assumptions

- MCP tools are defined with clear input/output schemas that the LLM can understand and invoke correctly.
- The MCP Server validates all tool requests to prevent unauthorized data access or malicious parameter injection.

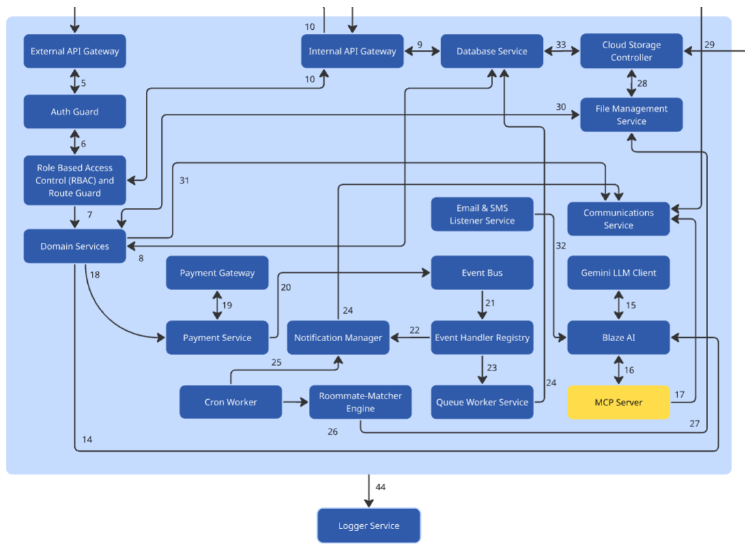

Figure 34: MCP Server Subsystem

6.21.2 Responsibilities

- Expose MCP-compliant tools for querying housing applications, room assignments, payment records, maintenance requests, and building availability.
- Validate tool invocation requests from Blaze AI, ensuring proper authentication tokens and authorized data access based on user roles.
- Execute tool logic by coordinating calls to appropriate backend services such as Database Service, Domain Services, or File Management Service.
- Return structured, schema-compliant responses containing requested data or appropriate error messages.

Table 33: MCP Server Interfaces

ID	Description	Inputs	Outputs
16	Bidirectional communication with Blaze AI for tool invocations	Tool Invocation Request User Context Query Parameters	Tool Response Structured Data Error Messages
17	Communication with Queue Worker Service for asynchronous tool execution	Tool Execution Job Service Registration Health Report	Job Status Service List Health Status

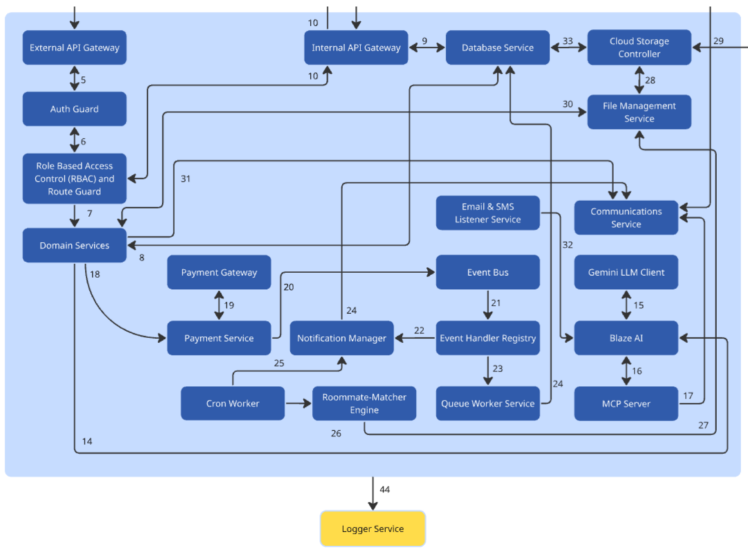

Figure 35: Logger Service Subsystem

6.22 Logger Service

The Logger Service subsystem provides centralized logging infrastructure for all backend subsystems, collecting, storing, and indexing log entries and performance metrics. It enables system monitoring, troubleshooting, security auditing, and performance analysis through structured log aggregation and query capabilities.

6.22.1 Assumptions

- Centralized log storage.

6.22.2 Responsibilities

- Receive logs and store or forward them.
- Make logs available for monitoring and analysis.

Table 34: Logger Service Interfaces

ID	Description	Inputs	Outputs
44	Collects logs and metrics	Log Entry Metric Report	Log Storage Log Query Result

References