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

Detailed Design Specification
CSE 4316: Senior Design I
Spring 2026

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	02.2.2026	AK	Created DDS template and document structure.
0.2	04.2.2026	AK	Added Section 3 Frontend Layer.
0.3	04.2.2026	AK	Added figures for Frontend, Backend, and Data Layer.
0.4	06.2.2026	TT	Completed Section 4 Data Layer descriptions.
0.5	06.2.2026	TT	Drafted Backend Layer Sections 5.1–5.4.
0.6	07.2.2026		Added Backend Layer Sections 5.5–5.8.
0.7	09.2.2026		Completed Backend Layer Sections 5.9–5.14.
0.8	14.2.2026	TT	Completed Backend Layer Sections 5.15–5.22.
0.9	19.2.2026	AK	Reviewed wordings, figures, and formatting across all sections.
1.0	20.2.2026		Proofread and submitted final DDS document.

CONTENTS

List of Figures

List of Tables

1 INTRODUCTION

The MavHousing is a comprehensive platform designed to manage student housing operations at the University of Texas at Arlington. This document provides the **Detailed Design Specification (DDS)**, focusing on the internal design and implementation details that enable the system to support housing applications, room assignments, maintenance requests, payments, and notifications across multiple user roles. It also covers the integration of automated communication and AI-assisted features that improve system responsiveness and user engagement.

Building on the **Software Requirements Specification (SRS)** and the **Architectural Design Specification (ADS)**, this DDS describes the structure of the system, including module responsibilities, data flows, database schemas, class and sequence diagrams, and interface definitions. It provides a clear technical roadmap for developers, testers, and system integrators to implement, integrate, and maintain all components efficiently, ensuring security, reliability, and maintainability throughout the platform.

2 SYSTEM OVERVIEW

The MavHousing system is organized into three primary layers: the Frontend Layer, the Backend Layer, and the Data Layer. These layers work together to provide a structured and maintainable framework for housing operations, including student applications, room assignments, maintenance requests, payments, and notifications. The separation of layers ensures modularity, simplifies maintenance, and allows developers to implement, test, and update components independently.

At a high level, the **Frontend Layer** handles user interactions and presents the interface for students, housing staff, and administrators, ensuring that all actions are intuitive and responsive. The **Backend Layer** manages business logic, authentication and authorization, application processing, and event-driven operations such as notifications and logging. It also coordinates communication between the frontend and data storage, validates requests, and triggers automated tasks where required. The **Data Layer** provides secure and persistent storage for user accounts, application records, uploaded documents, and system logs, ensuring data integrity, consistency, and accessibility across all system components. This layered architecture supports scalability, fault tolerance, and maintainability while allowing new features, such as AI-assisted notifications, to be integrated seamlessly. Figure **??** presents the overall data flow and interaction between these layers.

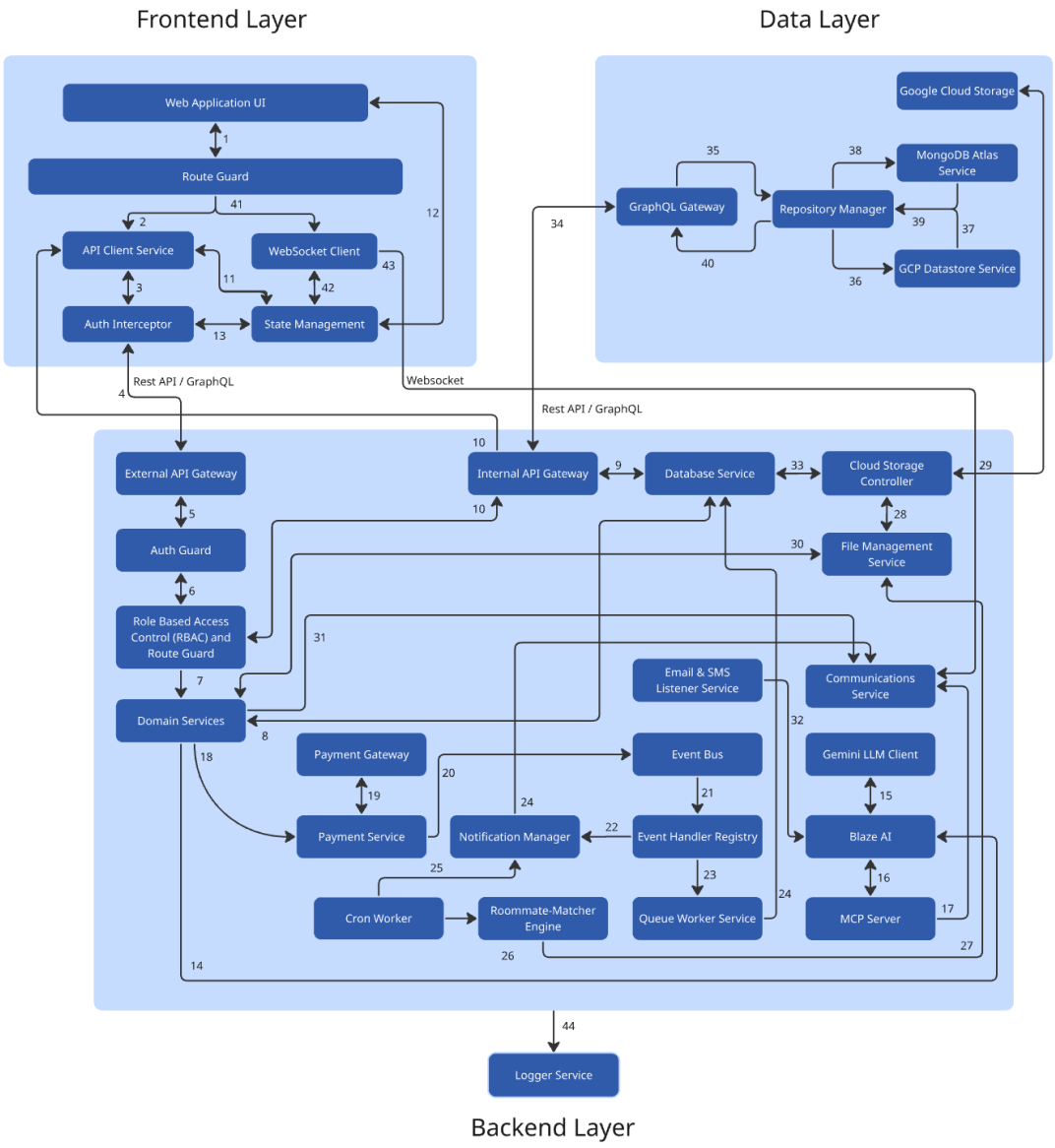

Figure 1: System Architecture

3 Frontend Layer Subsystems

The Frontend Layer manages all user interactions and presents a responsive interface for students, housing staff, and administrators. It is implemented as a web application using Next.js and React components, with Tailwind CSS for styling and Redux for state management. The layer communicates with the backend via RESTful APIs and WebSocket connections for real-time updates. Key subsystems include the Web Application UI for pages and forms, the Route Guard for authentication and role-based access, the API Client Service for managing HTTP requests, the Auth Interceptor for attaching JWT tokens to requests, the WebSocket Client for live notifications, and State Management for consistent data flow. Together, these subsystems provide a modular, secure, and maintainable frontend architecture.

3.1 Layer Hardware

The Frontend Layer does not require any dedicated hardware beyond standard client devices. Users interact with the system through web browsers on desktops, laptops, or mobile devices. No embedded or specialized hardware is required at the layer level, as all processing and rendering are handled by the client device and the backend server.

3.2 Layer Operating System

The Frontend Layer is platform-independent and can operate on any modern operating system that supports web browsers, including Windows, macOS, Linux, Android, and iOS. The system relies on browser capabilities rather than a specific operating system.

3.3 Layer Software Dependencies

The Frontend Layer is implemented using Next.js with React for component-based UI development. Styling is provided through Tailwind CSS, and global state management uses Redux. API requests to the backend are handled via Axios, and real-time notifications are managed using the WebSocket protocol. Authentication and session handling are managed using JWT tokens and an Auth Interceptor. All frontend dependencies are compatible with modern browsers and support modular, maintainable, and scalable application development.

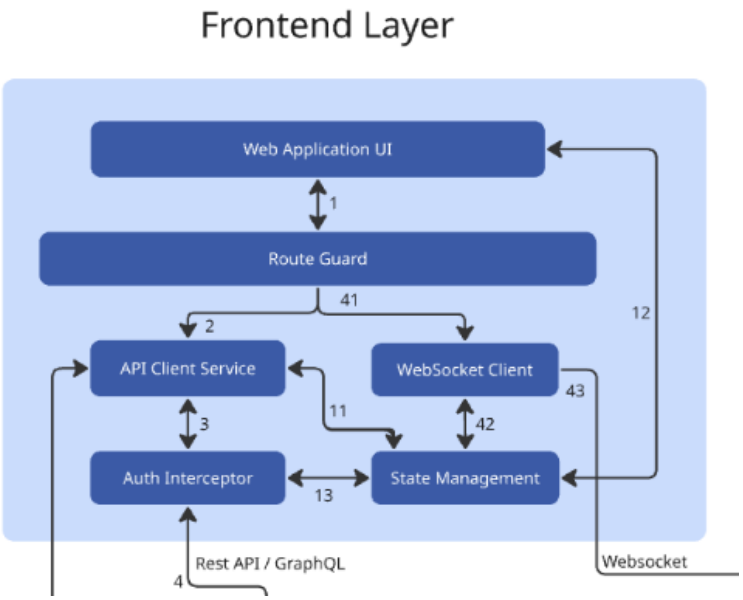

Figure 2: Frontend Layer Subsystems Diagram

3.4 Subsystem 1: Web Application UI

The Web Application UI is a software subsystem responsible for presenting the user interface and handling user interactions in the MavHousing system. It runs entirely in web browsers and provides students, staff, and administrators with access to dashboards, forms, and notifications. This subsystem captures user input, communicates with other frontend subsystems such as the Route Guard, API Client Service, WebSocket Client, and State Management, and displays data received from the backend in real time. It is designed for modularity and responsiveness, ensuring that interface components re-render efficiently based on state changes or user actions.

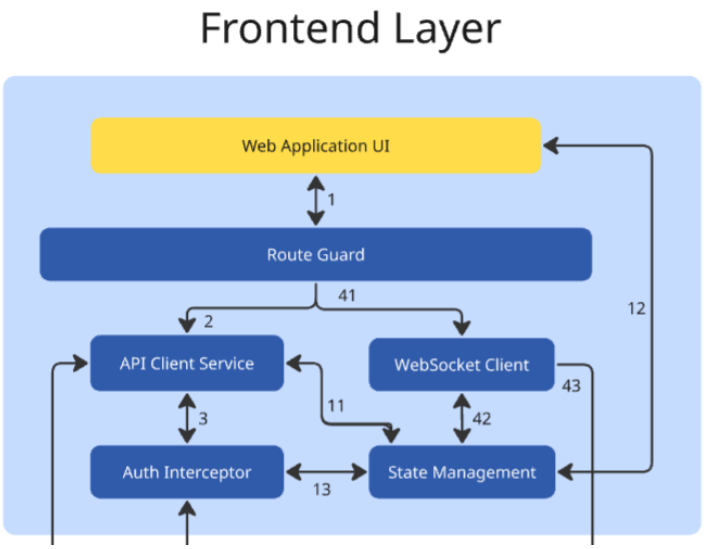

Figure 3: Web Application UI Subsystem

3.4.1 Subsystem Hardware

No dedicated hardware is required for this subsystem beyond standard client devices such as desktops, laptops, or mobile devices running modern web browsers. All processing is performed within the client device.

3.4.2 Subsystem Operating System

The Web Application UI is platform-independent and operates on any modern operating system that supports a compatible web browser, including Windows, macOS, Linux, Android, and iOS.

3.4.3 Subsystem Software Dependencies

This subsystem depends on Next.js for server-side rendering, React for component-based UI design, Tailwind CSS for responsive styling, Redux for state management, Axios for API calls, and WebSocket connections for real-time notifications. JWT is used for session authentication.

3.4.4 Subsystem Programming Languages

The primary programming language used is JavaScript, with JSX for React components and CSS (via Tailwind) for styling.

3.4.5 Subsystem Data Structures

Key data structures include state objects managed by Redux, which store user session information, authentication tokens, application data, and notifications. Components receive these states as props and update them via Redux actions to trigger re-renders.

3.4.6 Subsystem Data Processing

The subsystem processes user inputs such as form submissions, button clicks, and navigation events. It performs client-side validation, updates local state via Redux, and communicates with the backend through REST API calls. WebSocket messages are processed in real time to display notifications, alerts, or status updates. Rendering is optimized through Reacts virtual DOM to efficiently update only components affected by state changes.

3.5 Subsystem 2: Route Guard

The Route Guard is a software subsystem responsible for protecting frontend routes by enforcing authentication and role-based authorization. It works between the Web Application UI and other frontend services to ensure that only permitted users can access protected pages. Whenever a user attempts to navigate to a new route, the Route Guard checks the userâs login status and role information through State Management. Based on this verification, it either allows access, redirects the user to the login page, or displays an access denied message. This subsystem ensures secure navigation control and prevents unauthorized access to restricted system features.

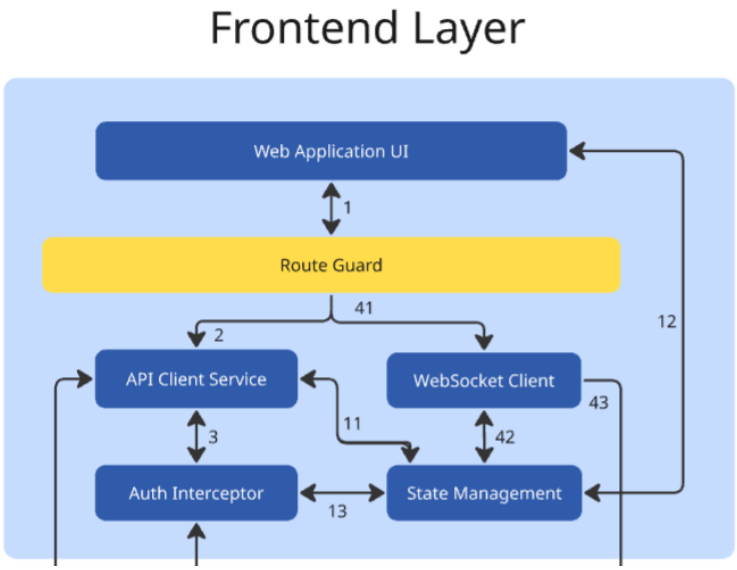

Figure 4: Route Guard Subsystem

3.5.1 Subsystem Hardware

The Route Guard does not require any dedicated hardware. It operates entirely on the client device within the web browser, using the deviceâs standard processing resources.

3.5.2 Subsystem Operating System

This subsystem is platform-independent and functions on any operating system that supports modern web browsers, including Windows, macOS, and Linux.

3.5.3 Subsystem Software Dependencies

Route Guard depends on the frontend routing framework provided by Next.js, along with React for component handling. It also relies on Redux State Management to retrieve authentication and role data, and JWT libraries for validating session tokens.

3.5.4 Subsystem Programming Languages

The subsystem is implemented using JavaScript (ES6+) within the React and Next.js framework environment.

3.5.5 Subsystem Data Structures

Route Guard utilizes authentication state objects stored in Redux. These include user session data such as user ID, authentication token (JWT), and assigned role. Route permission mappings are also

maintained to determine which roles can access specific routes.

3.5.6 Subsystem Data Processing

When a navigation request occurs, Route Guard intercepts the request and retrieves the userâs authentication state from Redux. It validates the presence and status of the JWT token and checks the userâs role against route access rules. Based on this evaluation, it either grants access, redirects the user to the login page, or blocks the request with an authorization error. This processing occurs in real time before page rendering.

3.6 Subsystem 3: API Client Service

The API Client Service is a frontend software subsystem responsible for managing all communication between the user interface and the backend server. It prepares, formats, and sends HTTP requests when users perform actions such as logging in, submitting housing applications, uploading documents, or requesting dashboard data. The service ensures that requests follow proper API structures and routes them through the Auth Interceptor for secure transmission. After receiving backend responses, it processes the returned data and forwards relevant information to State Management so the UI can update accordingly.

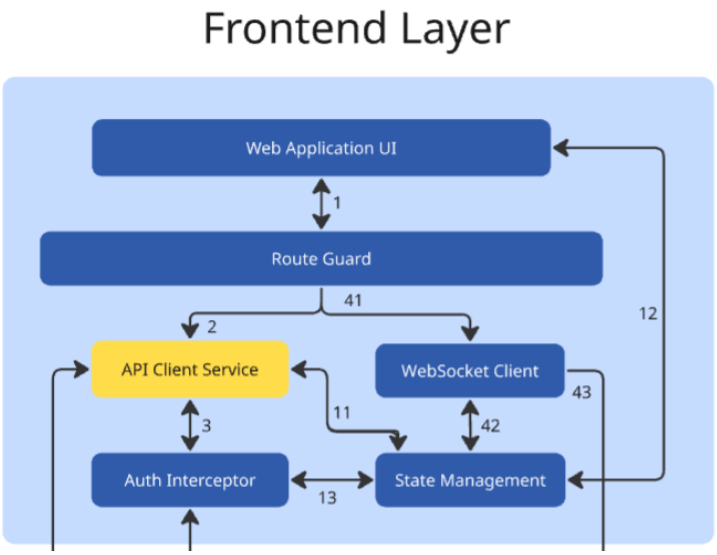

Figure 5: API Client Service Subsystem

3.6.1 Subsystem Hardware

The API Client Service does not require dedicated hardware. It runs within the clientâs web browser environment and utilizes the deviceâs standard computing resources for request processing.

3.6.2 Subsystem Operating System

This subsystem is operating system independent and functions on any OS capable of running modern web browsers, including Windows, macOS, and Linux.

3.6.3 Subsystem Software Dependencies

The subsystem depends on Axios (or Fetch API) for HTTP communication, Next.js routing utilities for endpoint management, and Redux for forwarding processed response data to application state. It also integrates with the Auth Interceptor for token attachment and secure request handling.

3.6.4 Subsystem Programming Languages

The API Client Service is implemented using JavaScript (ES6+) within the React and Next.js frontend framework.

3.6.5 Subsystem Data Structures

The subsystem uses structured request and response objects. Request objects include endpoint URLs, HTTP methods (GET, POST, PUT, DELETE), headers, authentication tokens, and payload data such as form inputs or file metadata. Response objects contain status codes, returned JSON data, authentication tokens, and error messages, which are forwarded to Redux State Management.

3.6.6 Subsystem Data Processing

When a user action triggers a request, the API Client Service constructs the appropriate HTTP request, attaches required headers, and forwards it to the Auth Interceptor. After the backend responds, the service parses the response, extracts relevant data, and identifies success or failure states. Successful data is sent to State Management for UI updates, while error responses are processed and relayed for user notification. File uploads are encoded using multipart/form-data before transmission.

3.7 Subsystem 4: Auth Interceptor

The Auth Interceptor is a frontend security subsystem that manages authentication handling for all outgoing API requests. It works as a middleware layer between the API Client Service and the backend server. Before any request is sent, the interceptor checks whether the endpoint requires authentication. If it is a protected request, the subsystem retrieves the userâs JWT token from State Management and attaches it to the Authorization header in Bearer format. This ensures that the backend can securely verify the userâs identity and permissions before processing the request.

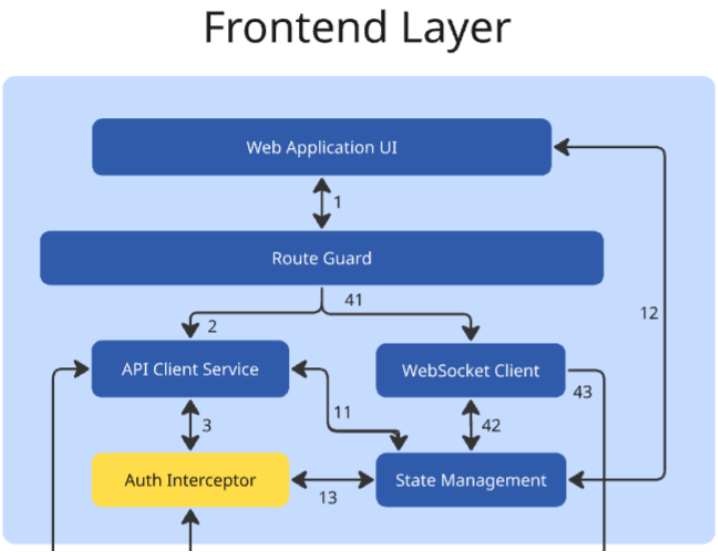

Figure 6: Auth Interceptor Subsystem

3.7.1 Subsystem Hardware

The Auth Interceptor does not rely on specialized hardware. It operates within the clientâs browser environment and uses standard device processing resources.

3.7.2 Subsystem Operating System

This subsystem is platform independent and functions on any operating system that supports modern web browsers, including Windows, macOS, and Linux.

3.7.3 Subsystem Software Dependencies

The subsystem depends on Axios interceptors (or Fetch middleware patterns) for request interception, Redux State Management for token retrieval, and Next.js environment configuration for managing secure API endpoints. It also integrates with the API Client Service for request forwarding.

3.7.4 Subsystem Programming Languages

The Auth Interceptor is implemented using JavaScript (ES6+) within the React and Next.js frontend framework.

3.7.5 Subsystem Data Structures

Key data structures include HTTP request objects containing headers, endpoint URLs, and payload data. JWT token objects stored in State Management include the token string, expiration metadata, and user role claims. Authorization headers follow the structure:

```
Authorization:  Bearer <JWT_Token>
```

3.7.6 Subsystem Data Processing

When an API request is initiated, the Auth Interceptor intercepts the request and evaluates whether authentication is required. If the endpoint is protected, it retrieves the JWT token from State Management and injects it into the request header. The authenticated request is then forwarded to the backend. If the backend returns an authentication error (such as HTTP 401), the interceptor can trigger token refresh workflows or redirect the user to reauthenticate. Public requests bypass token attachment and are forwarded directly.

3.8 Subsystem 5: WebSocket Client

The WebSocket Client is a frontend communication subsystem that enables real-time data exchange between the frontend and backend. Unlike traditional HTTP communication, this subsystem maintains a persistent connection that allows the backend to push instant notifications, alerts, and updates to users without requiring page refreshes. It plays a key role in delivering time-sensitive information such as maintenance updates, announcements, and system alerts. Incoming messages are processed and forwarded to State Management so the user interface can immediately reflect new information.

3.8.1 Subsystem Hardware

The WebSocket Client does not require dedicated hardware. It operates within the userâs web browser and utilizes standard client device networking capabilities.

3.8.2 Subsystem Operating System

This subsystem is operating system independent and functions on any platform that supports modern browsers, including Windows, macOS, and Linux.

3.8.3 Subsystem Software Dependencies

The subsystem relies on WebSocket protocol libraries such as Socket.IO Client or native WebSocket APIs. It also integrates with Next.js frontend services and Redux State Management to distribute real-time data to the UI.

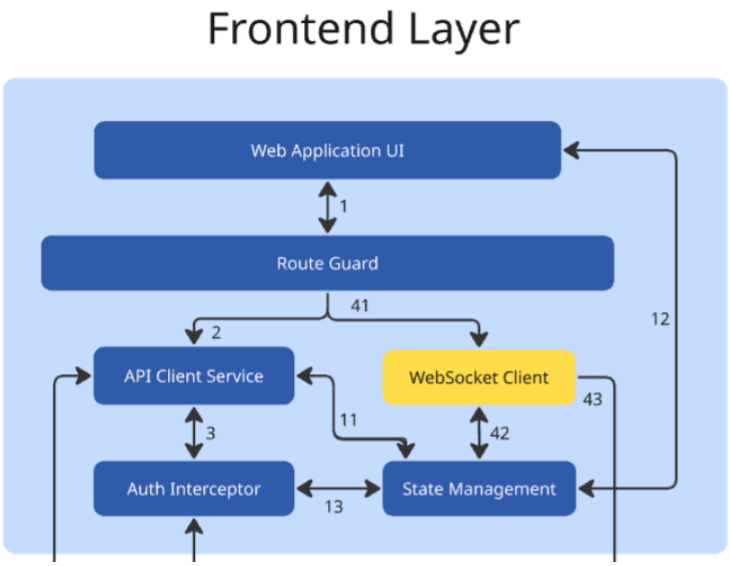

Figure 7: WebSocket Client Subsystem

3.8.4 Subsystem Programming Languages

The WebSocket Client is implemented using JavaScript (ES6+) within the React and Next.js frontend framework.

3.8.5 Subsystem Data Structures

Primary data structures include WebSocket message objects containing notification IDs, message content, timestamps, priority levels, and user targeting metadata. Connection state objects track socket status such as connected, disconnected, or reconnecting.

3.8.6 Subsystem Data Processing

When a user logs in, the subsystem establishes an authenticated WebSocket connection with the backend using the userâs JWT token. It continuously listens for incoming event messages such as maintenance updates or announcements. Received messages are parsed, validated, and transformed into standardized notification objects before being forwarded to State Management. If the connection drops, automatic reconnection logic attempts to restore communication without user intervention.

3.9 Subsystem 6: State Management

The State Management subsystem serves as the central data controller for the frontend layer. It stores and manages all shared application data required across frontend subsystems, including authentication status, JWT tokens, user profile information, housing application data, and real-time notifications. This subsystem ensures that all frontend components operate using consistent and up-to-date information. Whenever new data is received from the backend or WebSocket Client, State Management updates its stored state and automatically notifies the Web Application UI to refresh and reflect the latest system changes.

3.9.1 Subsystem Hardware

State Management does not require dedicated hardware. It operates within the client deviceâs browser memory and uses standard computing resources available on the userâs system.

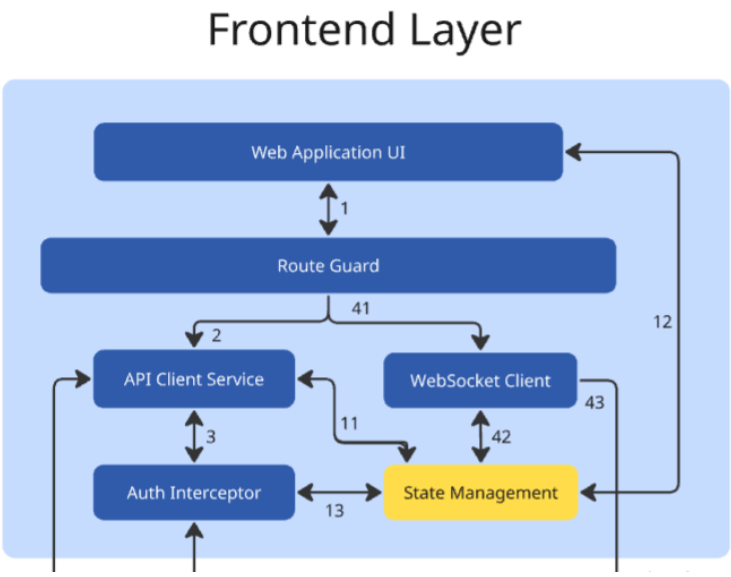

Figure 8: State Management Subsystem

3.9.2 Subsystem Operating System

This subsystem is platform independent and functions on any operating system that supports modern web browsers, including Windows, macOS, and Linux.

3.9.3 Subsystem Software Dependencies

The subsystem relies on state libraries such as Redux Toolkit for centralized state storage and management. It also utilizes browser storage APIs such as localStorage or sessionStorage for session persistence. Integration with React ensures automatic UI re-rendering when state updates occur.

3.9.4 Subsystem Programming Languages

State Management is implemented using JavaScript (ES6+) and TypeScript within the React / Next.js frontend framework.

3.9.5 Subsystem Data Structures

Key data structures include authentication state objects (login status, JWT token, expiration), user profile objects (user ID, email, role), application data records, and notification queues. These structures are organized into Redux slices or state modules for efficient access and updates.

3.9.6 Subsystem Data Processing

The subsystem processes incoming data from API responses and WebSocket messages by validating, structuring, and storing it within the centralized state store. It manages token persistence, monitors authentication changes, and distributes updated data to subscribed components. When state changes occur, it triggers automatic UI updates to ensure users always see the most current system information.

4 Data Layer Subsystems

The Data Layer is responsible for managing all persistent storage, database operations, and file storage services within the MavHousing system. It supports structured and unstructured data storage, secure document handling, and efficient data retrieval for housing applications, user records, maintenance logs, and system communications. This layer operates through cloud-based infrastructure and integrates multiple database technologies to ensure scalability, reliability, and performance.

4.1 Layer Hardware

The Data Layer does not rely on dedicated on-premise hardware. Instead, it operates on cloud-managed infrastructure provided by Google Cloud Platform (GCP) and MongoDB Atlas. All databases, storage services, and processing resources run on virtualized cloud servers maintained by their respective providers. This architecture ensures high availability, automatic scaling, backup management, and fault tolerance without requiring physical hardware deployment by the development team.

4.2 Layer Operating System

Since the Data Layer is hosted on managed cloud platforms, the underlying operating systems are abstracted from the application. Services such as GCP Datastore, Google Cloud Storage, and MongoDB Atlas run on provider-managed Linux-based server environments. Developers interact with these services through secure APIs and dashboards rather than direct operating system access.

4.3 Layer Software Dependencies

The Data Layer depends on several cloud services, database engines, and integration frameworks to function properly. PostgreSQL (via GCP Datastore) is used for structured relational data such as user accounts and housing applications. MongoDB Atlas is used for unstructured data, including logs and event records. Google Cloud Storage is used for storing uploaded documents and files.

Additional dependencies include GraphQL services for flexible data querying, database drivers and ORM libraries for backend communication, and cloud SDKs for secure storage access. These dependencies enable secure data transactions, efficient query processing, and reliable storage management across the system.

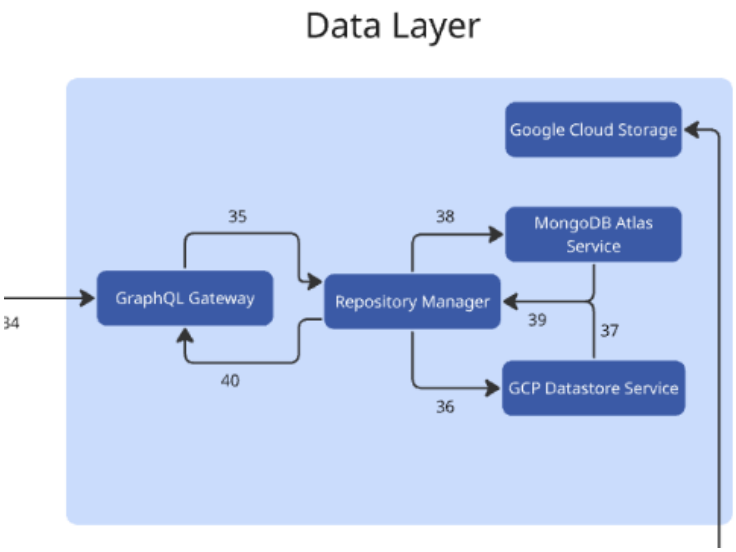

Figure 9: Data Layer subsystem description diagram

4.4 Subsystem 1: GraphQL Gateway

The GraphQL Gateway subsystem is a web service that provides a unified API interface for all data access in the MavHousing system. It receives GraphQL queries and mutations from frontend clients and backend services, validates them against the GraphQL schema, and routes them to the Repository Manager for execution. By abstracting the underlying databases, it provides a type-safe and consistent interface for operations such as housing applications, user management, and notifications.

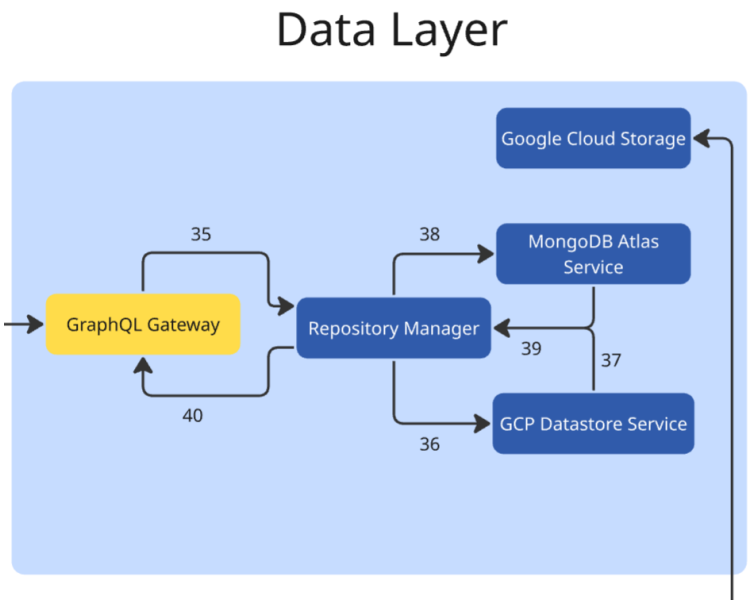

Figure 10: GraphQL Gateway Subsystem

4.4.1 Subsystem Hardware

The subsystem runs on virtual servers hosted on Google Cloud Platform or similar cloud infrastructure. It does not require dedicated physical hardware, as all processing is handled in cloud-hosted Node.js instances.

4.4.2 Subsystem Operating System

The servers hosting the GraphQL Gateway run a Linux-based operating system (e.g., Ubuntu 22.04) in containerized or VM environments.

4.4.3 Subsystem Software Dependencies

The service depends on the Node.js runtime for execution, NestJS for modular backend architecture, Apollo Server and GraphQL for query parsing and schema validation, Prisma Client for database access through the Repository Manager, and GraphQL tools for schema stitching and structured error handling.

4.4.4 Subsystem Programming Languages

Implemented primarily in TypeScript for type safety and maintainability, with JavaScript used at runtime.

4.4.5 Subsystem Data Structures

Data structures include GraphQL query and mutation objects that encapsulate requested fields, variables, and operation types, resolver request objects containing parsed query information and user con-

text, and GraphQL response payloads formatted as JSON objects with the requested data or structured errors.

4.4.6 Subsystem Data Processing

The subsystem parses and validates incoming GraphQL queries and mutations against the schema, routes valid requests to the Repository Manager for execution on the appropriate storage service, aggregates results from multiple sources if needed, and returns them in the correct GraphQL response format. Errors, including schema mismatches or database issues, are converted into structured GraphQL error messages for the client.

4.5 Subsystem 2: Repository Manager

The Repository Manager subsystem is a web service backend module responsible for coordinating all data storage and retrieval operations in MavHousing. It implements the repository pattern to abstract database access logic, allowing other subsystems, like the GraphQL Gateway, to perform data operations without knowing the specifics of the underlying storage systems. It determines whether to use MongoDB Atlas for document-based data, GCP Datastore for relational entities, or Google Cloud Storage for binary files, aggregates results, and returns them in a consistent format.

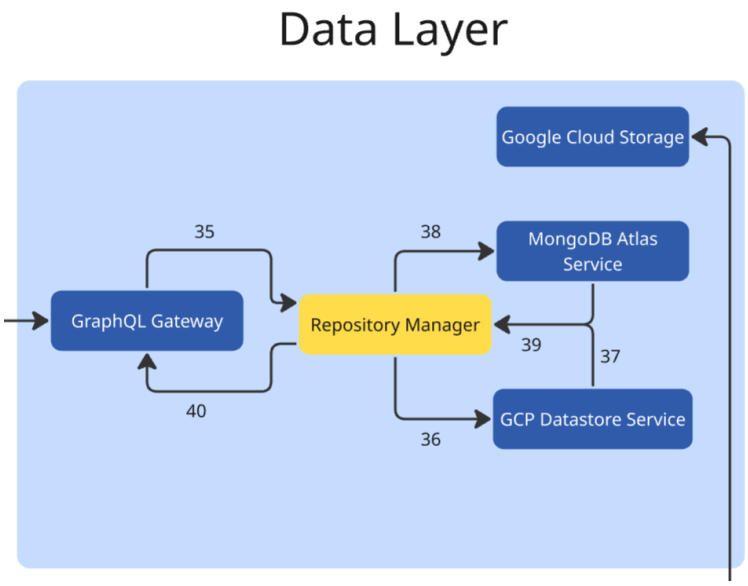

Figure 11: Repository Manager Subsystem

4.5.1 Subsystem Hardware

The subsystem runs on cloud-hosted servers or containerized instances and does not require dedicated physical hardware. It relies on virtualized compute resources to handle all processing tasks.

4.5.2 Subsystem Operating System

The subsystem uses Linux-based operating systems, such as Ubuntu, deployed in containerized or virtual machine environments.

4.5.3 Subsystem Software Dependencies

This subsystem depends on the Node.js runtime for execution and the NestJS framework for building a modular backend architecture. It also relies on Prisma ORM for relational database interactions, the

MongoDB Node Driver for document-based operations, and Google Cloud SDKs for managing storage services. Additionally, optional caching libraries, such as Redis, can be used to improve performance.

4.5.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript to ensure type safety and maintainable code. JavaScript is used for runtime execution after TypeScript compilation.

4.5.5 Subsystem Data Structures

The subsystem uses repository request objects that encapsulate query types, target storage locations, and parameters. Entity objects represent relational records from GCP Datastore, while document objects represent JSON documents from MongoDB Atlas. Metadata objects are used for binary files stored in Google Cloud Storage, and aggregated response objects combine results from multiple storage sources to provide consistent responses.

4.5.6 Subsystem Data Processing

The subsystem routes requests to the appropriate storage service and converts repository requests into MongoDB queries, Datastore queries, or Cloud Storage operations. It aggregates results into a unified format suitable for the GraphQL Gateway, handles errors and retries for failed operations, and optionally caches frequently accessed data to improve performance.

4.6 Subsystem 3: MongoDB Atlas Service

The MongoDB Atlas Service subsystem is a cloud-hosted NoSQL database that provides document storage for semi-structured and dynamic data in the MavHousing system. It stores user profiles, application logs, audit trails, and other JSON-like documents. Mongoose is used as an Object Document Mapper (ODM) to define schemas, validate data, and execute queries. MongoDB Atlas ensures horizontal scalability, high availability, and automated backups for reliability.

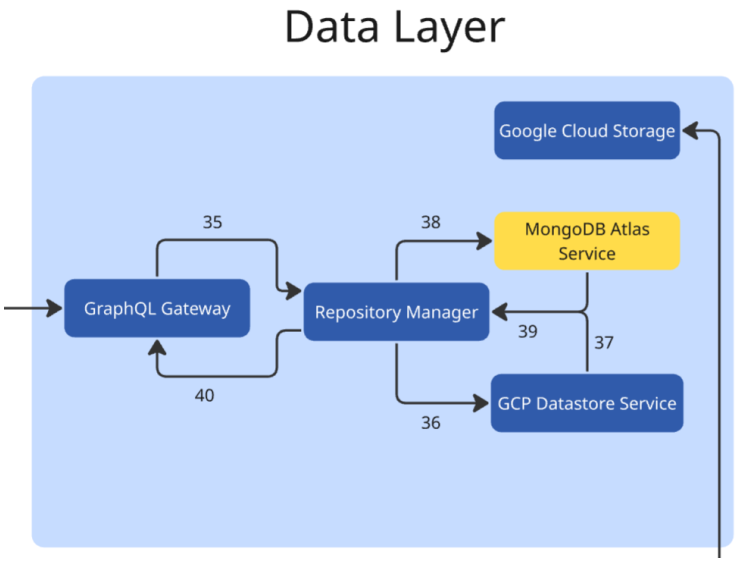

Figure 12: MongoDB Atlas Service Subsystem

4.6.1 Subsystem Hardware

The subsystem runs on cloud-managed virtual servers provided by MongoDB Atlas. No dedicated on-premises hardware is required, as all instances are fully managed in the cloud.

4.6.2 Subsystem Operating System

Managed by MongoDB Atlas, the underlying servers typically run Linux-based operating systems in containerized or virtualized environments.

4.6.3 Subsystem Software Dependencies

The service relies on the managed MongoDB Atlas database, Mongoose ODM for schema definition and validation, and cloud SDKs for monitoring, backups, and administrative tasks.

4.6.4 Subsystem Programming Languages

Database interactions and scripting are primarily implemented in JavaScript and TypeScript through Mongoose.

4.6.5 Subsystem Data Structures

Key data structures include document objects containing JSON-like data with nested fields and arrays, query objects defining filter conditions, updates, and aggregation pipelines, and schema definitions that enforce validation rules and relationships between documents.

4.6.6 Subsystem Data Processing

The subsystem handles CRUD operations on collections, enforces schema validation via Mongoose, executes queries including aggregation and indexing for efficient retrieval, manages replication and horizontal scaling for high availability, and supports automated backup and recovery processes provided by MongoDB Atlas.

4.7 Subsystem 4: GCP Datastore Service

The GCP Datastore Service subsystem is a cloud-hosted relational database used for structured, transactional data in the MavHousing system. It stores user accounts, housing applications, room assignments, payment records, and other relational entities. TypeORM is used as the Object-Relational Mapper (ORM) to define entity models, handle migrations, and execute type-safe queries. This service provides strong consistency for transactional operations and integrates with the Repository Manager for unified data access.

4.7.1 Subsystem Hardware

The subsystem runs on cloud-managed virtual machines provided by Google Cloud Platform, with no dedicated on-premises hardware required.

4.7.2 Subsystem Operating System

The underlying servers are Linux-based and operate in containerized or virtualized environments managed by GCP.

4.7.3 Subsystem Software Dependencies

The service depends on PostgreSQL as the database engine, TypeORM for object-relational mapping, and GCP client libraries for authentication, monitoring, and management.

4.7.4 Subsystem Programming Languages

Database interaction and scripting are primarily implemented in TypeScript and JavaScript using TypeORM.

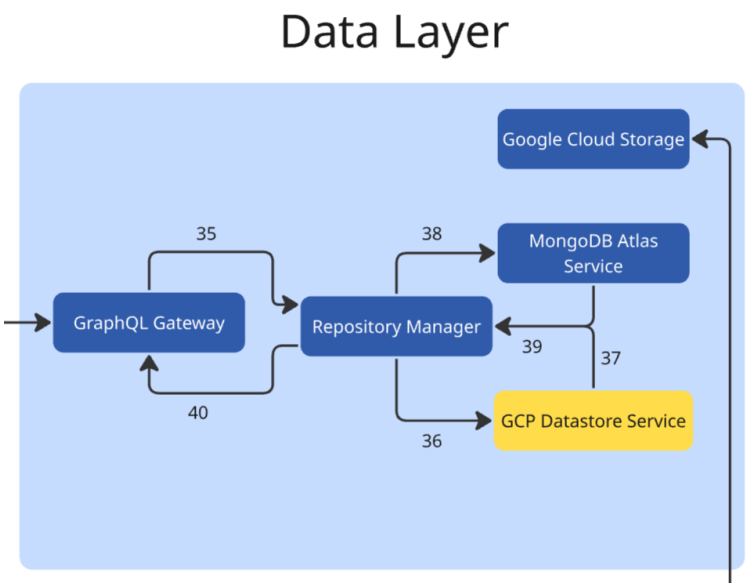

Figure 13: GCP Datastore Service Subsystem

4.7.5 Subsystem Data Structures

Key data structures include entity models representing database tables such as User, HousingApplication, and Payment. Query objects define fetch, update, and delete operations, while transaction objects bundle multiple operations to ensure atomicity and data consistency.

4.7.6 Subsystem Data Processing

The subsystem handles CRUD operations on relational tables, executes transactions for atomic operations, maps database rows to application entities via TypeORM, performs optimized queries for fast filtering and sorting, and manages schema migrations through TypeORM migration scripts to safely evolve the database structure.

4.8 Subsystem 5: Google Cloud Storage Service

The Google Cloud Storage (GCS) Service subsystem provides scalable object storage for large files, media assets, and document attachments in the MavHousing system. It manages uploads, downloads, deletions, and lifecycle policies for housing application attachments, lease agreements, maintenance photos, and system backups. Integration with the Repository Manager allows metadata tracking and secure URL generation, while access control is enforced through Google Cloud IAM policies.

4.8.1 Subsystem Hardware

The subsystem runs entirely on cloud-managed infrastructure provided by Google Cloud, so no dedicated on-premises hardware is required.

4.8.2 Subsystem Operating System

The underlying servers are Linux-based and run in containerized or virtualized environments managed by GCP.

4.8.3 Subsystem Software Dependencies

The service depends on the Google Cloud Storage SDK and client libraries for Node.js and TypeScript. It also integrates with the Repository Manager for metadata handling and uses GCPâs logging and mon-

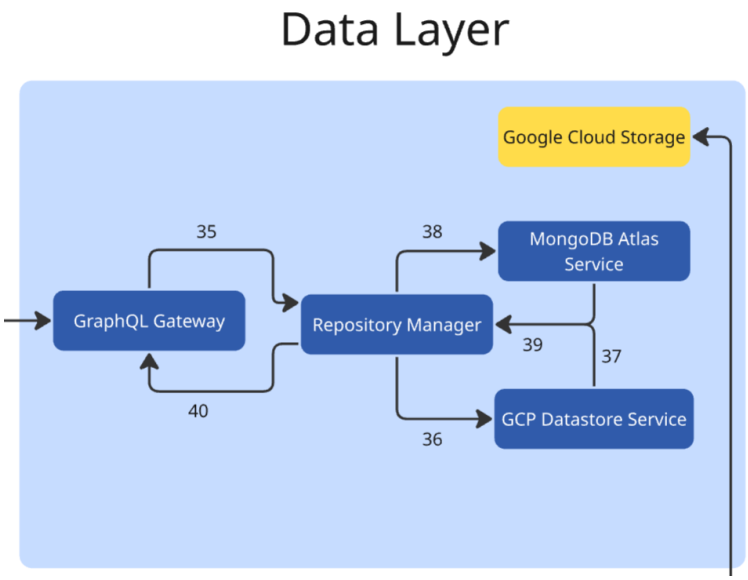

Figure 14: Google Cloud Storage Service Subsystem

itoring tools for observability.

4.8.4 Subsystem Programming Languages

The subsystem is implemented in TypeScript and JavaScript for API integration, cloud functions, and service orchestration.

4.8.5 Subsystem Data Structures

Data structures include Blob objects for the stored files, metadata objects containing file attributes and permissions, and file streams used for uploading or downloading binary content.

4.8.6 Subsystem Data Processing

File uploads and downloads are handled efficiently using chunked or streamed transfers. Lifecycle policies ensure automatic cleanup of obsolete files. Metadata is associated with the Repository Manager to track file ownership and permissions, and signed URLs provide secure, temporary access to files.

5 Backend Layer Subsystems

The Backend Layer handles core business logic, API routing, authentication, payments, notifications, file management, and AI services. It acts as a bridge between the frontend and data layers, exposing APIs, performing background jobs, and ensuring secure and reliable processing of all application operations.

5.1 Layer Hardware

The backend layer does not require dedicated hardware. All services run on standard cloud servers or virtual machines provided by the deployment environment. Computing resources such as CPU, memory, and network bandwidth are allocated according to service requirements, and scaling is handled automatically by the cloud infrastructure when needed.

5.2 Layer Operating System

Backend services are platform independent and typically run on Linux-based operating systems, such as Ubuntu or Debian, which provide stability, security, and compatibility with Node.js and other server-side frameworks.

5.3 Layer Software Dependencies

The backend relies on Node.js as the runtime environment and uses Express or NestJS for routing and microservice orchestration. Logging is handled through Winston, while Sentry provides error monitoring and observability. TypeORM and Mongoose manage database interactions for relational and document-based data, respectively. Additional dependencies include GraphQL libraries for API endpoints, JWT for authentication, and integration libraries for cloud services such as Google Cloud Storage, email/SMS providers, and payment gateways. Task scheduling and background jobs use cron or queue-based systems for timed or asynchronous operations.

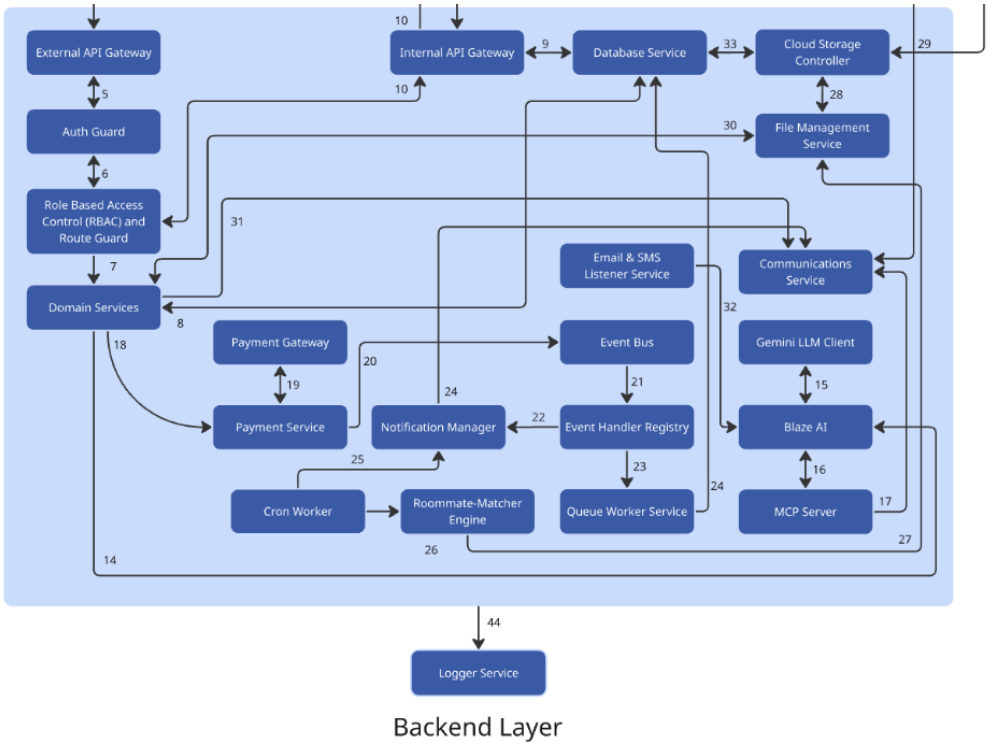

Figure 15: Backend subsystem description diagram

5.4 Subsystem 1: External API Gateway

The External API Gateway is a web service that handles all incoming API requests from outside the MavHousing system. It provides a single entry point for clients, performing initial validation, authentication, and routing of requests to the appropriate internal services. The gateway enforces security measures such as API key validation, token verification, and rate limiting, ensuring that only authorized and trusted traffic reaches backend services like the Auth Guard, RBAC module, and internal APIs. By centralizing these responsibilities, the gateway maintains a consistent interface for external clients and simplifies monitoring, logging, and observability.

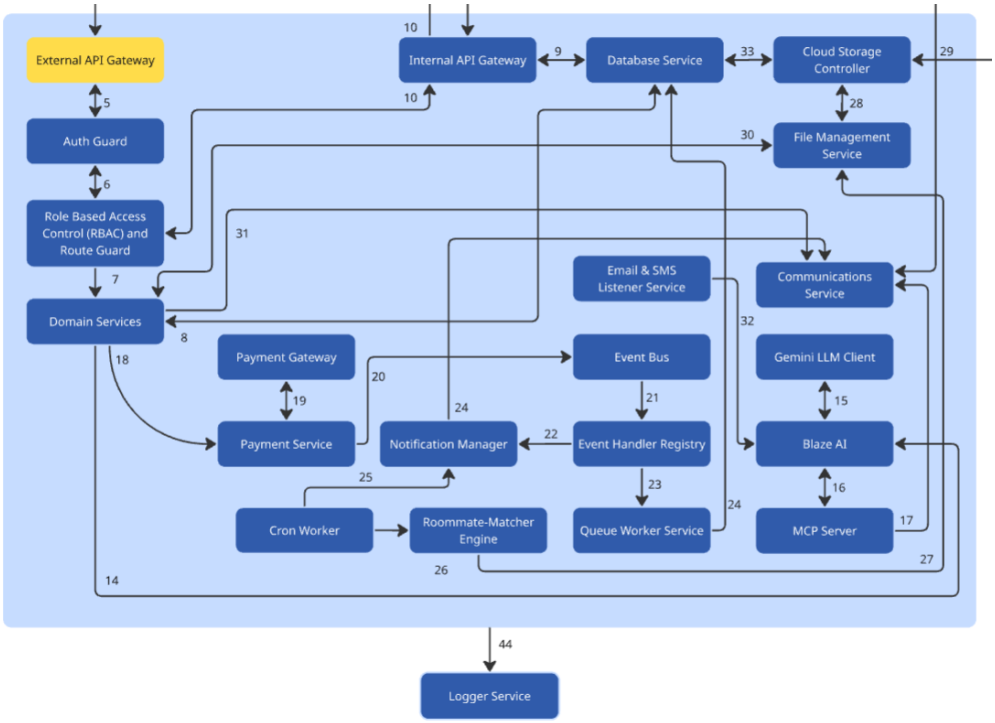

Figure 16: External API Gateway Subsystem

5.4.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or container instances provided by Google Cloud Platform. No dedicated hardware is required, and compute resources scale automatically with request load to maintain availability.

5.4.2 Subsystem Operating System

Servers hosting the gateway run a Linux-based operating system (e.g., Ubuntu 22.04) in containerized or virtualized environments to ensure stability, security, and efficient resource utilization.

5.4.3 Subsystem Software Dependencies

The gateway relies on the Node.js runtime and NestJS framework for modular backend services. Apollo Server and Express are used for handling HTTP and GraphQL requests, while middleware libraries handle logging, authentication, and request validation. Cloud SDKs and monitoring tools such as Winston and Sentry provide observability, error tracking, and logging for incoming API traffic.

5.4.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript for type safety and maintainability, with JavaScript used at runtime after compilation.

5.4.5 Subsystem Data Structures

Key data structures include request objects containing HTTP/GraphQL payloads, headers, authentication tokens, and client metadata. Response objects encapsulate status codes, result data, or error messages. Rate limiting and logging data structures track request counts and timestamps for observability and security enforcement.

5.4.6 Subsystem Data Processing

Incoming requests are first validated for authentication and format, then routed to the appropriate backend service. Requests may be enriched with context information such as user session details. The gateway performs rate limiting, error handling, and logging. For GraphQL queries, requests are parsed and forwarded to the internal API or repository layer, while REST calls are translated into internal service calls. Failed requests are handled with retries, structured error responses, or blocked according to security rules. This ensures secure, reliable, and efficient handling of all external traffic.

5.5 Subsystem 2: Auth Guard

The Auth Guard subsystem is a backend service module responsible for verifying user authentication and controlling access to all protected resources in the MavHousing system. It ensures that incoming requests are properly authenticated via tokens, session data, or credentials before allowing them to proceed to the RBAC module or other internal services. By centralizing authentication checks, the Auth Guard protects sensitive endpoints, enforces security policies, and provides standardized error responses for unauthenticated or unauthorized requests.

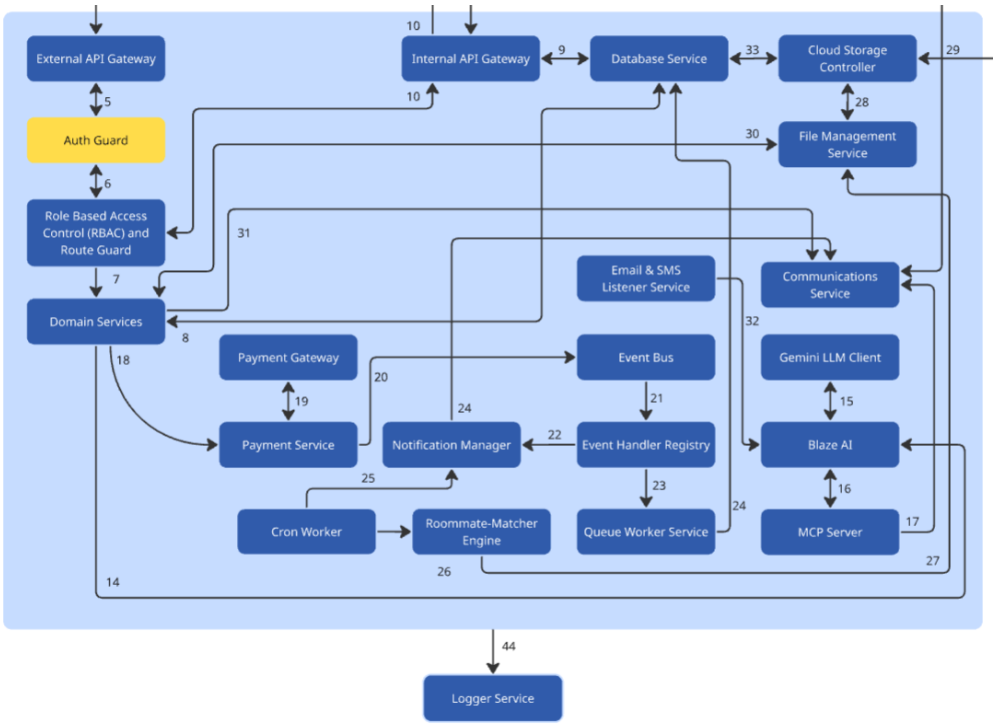

Figure 17: Auth Guard Subsystem

5.5.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or container instances, leveraging standard cloud compute resources. No dedicated physical hardware is required, and scaling is handled by the cloud provider.

5.5.2 Subsystem Operating System

Servers hosting the Auth Guard run a Linux-based OS (e.g., Ubuntu 22.04) within containerized or virtualized environments to ensure stability and security.

5.5.3 Subsystem Software Dependencies

The Auth Guard relies on the Node.js runtime and NestJS framework for modular backend services. Libraries for JWT token validation, OAuth 2.0, and session management are used, along with middleware for logging, request parsing, and error handling. Observability tools like Winston and Sentry provide monitoring and error tracking.

5.5.4 Subsystem Programming Languages

Implemented primarily in TypeScript for type safety, maintainability, and consistency across backend services. JavaScript is used at runtime after compilation.

5.5.5 Subsystem Data Structures

Key data structures include authentication request objects containing headers, JWT tokens, session identifiers, and user metadata. Response objects encapsulate authentication and authorization results, including status codes, allowed roles, or error messages. Context objects store temporary session or request-specific data during processing.

5.5.6 Subsystem Data Processing

Incoming requests are first checked for valid authentication tokens or session identifiers. Tokens are decoded and verified for integrity, expiration, and user identity. Once validated, the subsystem forwards the request to the RBAC module along with the user context to determine authorization. Requests failing authentication are blocked, logged, and returned with standardized error responses. The Auth Guard may also handle token refresh, session renewal, and logging of authentication attempts for auditing and security purposes.

5.6 Subsystem 3: Role Based Access Control (RBAC) and Route Guard

The RBAC and Route Guard subsystem is a backend service module that enforces authorization policies for the MavHousing system. It evaluates each authenticated request to determine if the userâs role (student, staff, or administrator) permits access to the requested resources or operations. By centralizing role checks, it ensures secure, consistent access control across all internal services and endpoints.

5.6.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or container instances. No dedicated hardware is required.

5.6.2 Subsystem Operating System

The subsystem uses a Linux-based operating system (e.g., Ubuntu 22.04) within containerized or virtual machine environments.

5.6.3 Subsystem Software Dependencies

The subsystem relies on the Node.js runtime, the NestJS framework, and role/permission management libraries. Logging and monitoring are handled via tools such as Winston or Sentry.

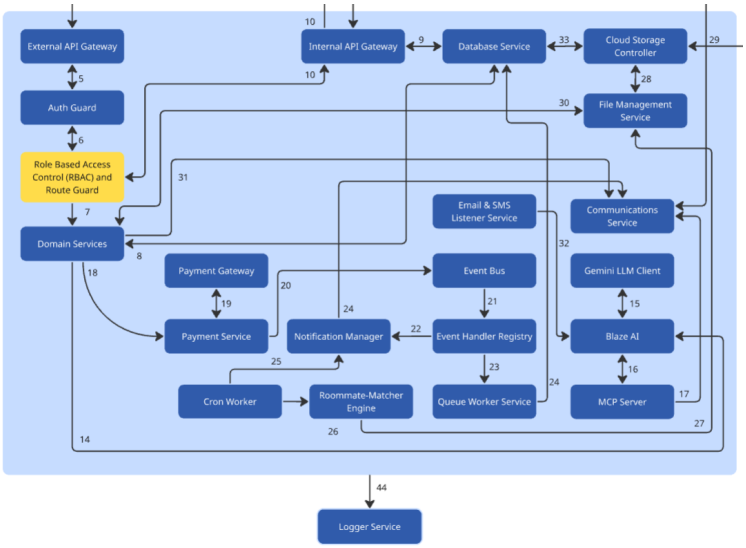

Figure 18: Role Based Access Control (RBAC) and Route Guard Subsystem

5.6.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript, with JavaScript used at runtime.

5.6.5 Subsystem Data Structures

Authorization request objects include user ID, session information, and requested resource metadata. Role check results indicate permitted actions or access denial. Context objects may store temporary role validation data during request processing.

5.6.6 Subsystem Data Processing

The subsystem receives authenticated requests from the Auth Guard, checks the userâs role and permissions against requested resources, and either forwards the request to domain services or blocks it. All access decisions are logged for auditing and security purposes.

5.7 Subsystem 4: Domain Services

The Domain Services subsystem is a backend service module that implements the core business logic for the MavHousing system. It manages workflows such as housing application processing, room assignments, maintenance requests, and payment operations. Acting as the central point for business rules, it coordinates between internal services, database operations, file management, and payment processing to ensure consistency, integrity, and correct application of policies.

5.7.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or containers. No dedicated hardware is required.

5.7.2 Subsystem Operating System

The subsystem uses a Linux-based operating system (e.g., Ubuntu 22.04) within containerized or virtual machine environments.

5.7.3 Subsystem Software Dependencies

The subsystem relies on the Node.js runtime, NestJS for modular service architecture, and libraries for logging, validation, and asynchronous messaging (e.g., Winston, Sentry, or RabbitMQ/Cloud Pub-Sub).

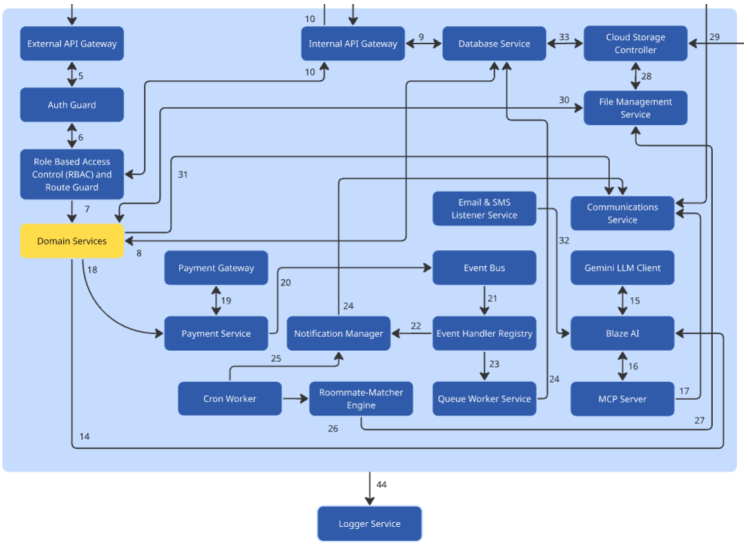

Figure 19: Domain Services Subsystem

5.7.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript, with JavaScript used at runtime.

5.7.5 Subsystem Data Structures

The subsystem includes business request objects containing user context, operation type, and resource data. File operation requests and payment request objects encapsulate related metadata. Event objects contain the payload for asynchronous processing and notifications.

5.7.6 Subsystem Data Processing

The subsystem receives authorized requests from RBAC, validates business rules, and orchestrates multiple backend operations such as database updates, file storage, payment processing, and event generation. It ensures transactional integrity, applies housing policies, and forwards results or events to downstream services like the Internal API Gateway, File Management Service, Payment Gateway, or Event Handler Registry.

5.8 Subsystem 5: Internal API Gateway

The Internal API Gateway subsystem is a backend service module that exposes REST and GraphQL endpoints for internal clients, including the Frontend Layer and Data Layer. It routes incoming requests from the External API Gateway, API Client Service, or GraphQL Gateway to the appropriate backend subsystems, such as Domain Services, Database Service, or File Management. The gateway abstracts internal service details, enforces API contracts, and ensures backward compatibility through versioning.

5.8.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or container instances. No dedicated physical hardware is required.

5.8.2 Subsystem Operating System

The subsystem uses a Linux-based operating system (e.g., Ubuntu 22.04) in containerized or virtual machine environments.

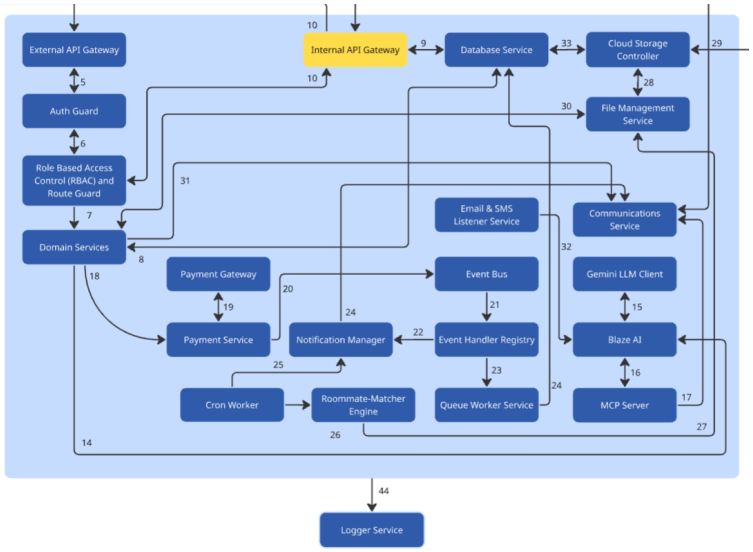

Figure 20: Internal API Gateway Subsystem

5.8.3 Subsystem Software Dependencies

The subsystem depends on the Node.js runtime, NestJS for modular API architecture, Apollo Server for GraphQL handling, and middleware libraries for logging, validation, and rate limiting. Cloud monitoring and observability tools such as Sentry or Stackdriver may also be integrated.

5.8.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript, with JavaScript used at runtime.

5.8.5 Subsystem Data Structures

Includes routing request objects containing endpoint, method, parameters, and user context. Response objects encapsulate payload data, status codes, and metadata. GraphQL query objects are used to communicate with downstream services.

5.8.6 Subsystem Data Processing

The subsystem receives authenticated requests from the External API Gateway, parses and validates them, and determines the target backend subsystem. Requests are transformed to match internal service interfaces, routed accordingly, and the responses are formatted to comply with the external API contract. Logging, throttling, and error handling are applied to ensure observability and system reliability.

5.9 Subsystem 6: Database Service

The Database Service subsystem is a backend web service that provides a unified interface for accessing both relational and document-based databases. It abstracts database-specific implementation details from other backend services, allowing Domain Services, the Internal API Gateway, and other subsystems to perform queries, transactions, and data manipulations without worrying about the underlying database engine. This service ensures ACID-compliant transactions for PostgreSQL and eventual consistency for MongoDB, while pooling connections and managing concurrent access efficiently.

5.9.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or containerized instances; no dedicated on-premises hardware is required.

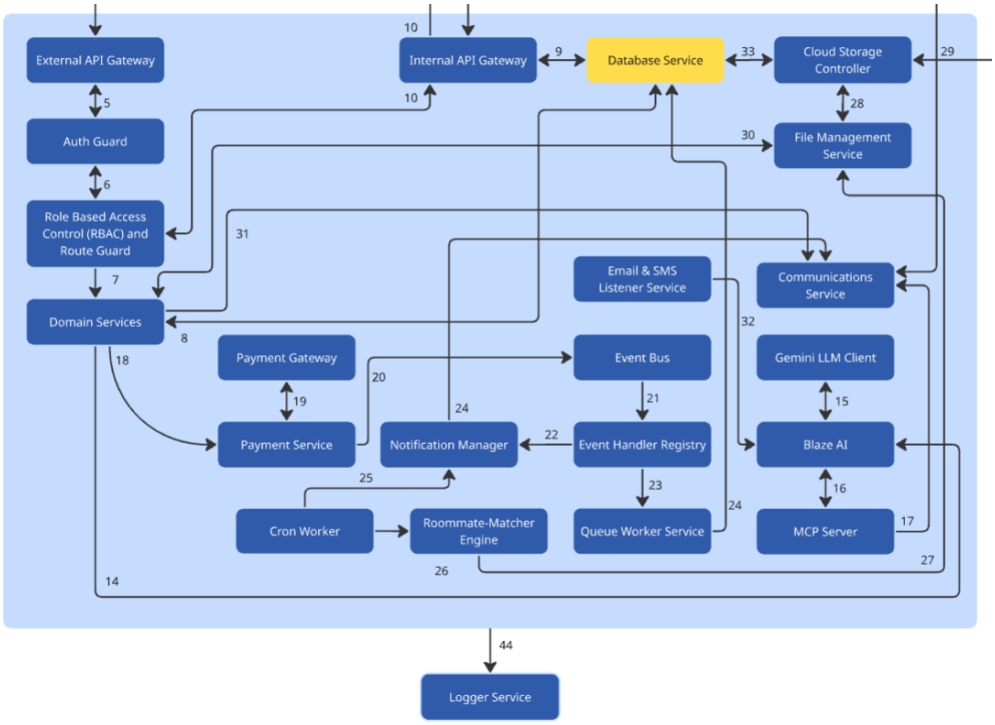

Figure 21: Database Service subsystem

5.9.2 Subsystem Operating System

The subsystem is managed by the cloud provider, typically using Linux-based OS environments such as Ubuntu or Debian.

5.9.3 Subsystem Software Dependencies

The subsystem depends on PostgreSQL and MongoDB drivers, connection pooling libraries, and database client SDKs. Integration with configuration management tools ensures secure access to credentials and connection strings.

5.9.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript and JavaScript for backend services interacting with both relational and document databases.

5.9.5 Subsystem Data Structures

Key data structures include query request objects, transaction objects for PostgreSQL, document objects for MongoDB, and response payloads formatted for consumption by Domain Services or API gateways.

5.9.6 Subsystem Data Processing

The subsystem processes incoming queries and transactions by routing them to the appropriate database engine, managing connection pooling, executing queries or aggregation pipelines, enforcing transactional integrity, and returning results in a consistent format. It also handles database updates triggered by events from the Event Bus and coordinates with the Cloud Storage Controller for file-related operations.

5.10 Subsystem 7: Cloud Storage Controller

The Cloud Storage Controller subsystem is a backend web service responsible for managing all interactions with Google Cloud Storage (GCS). It provides a unified interface for backend services to upload, download, and manage files without directly handling cloud SDK complexity. The subsystem handles authentication, request formatting, error handling, and generates signed URLs for secure temporary access to files such as housing documents, maintenance photos, and lease agreements. It also coordinates with the File Management Service to ensure that metadata and database records are updated in sync with storage operations.

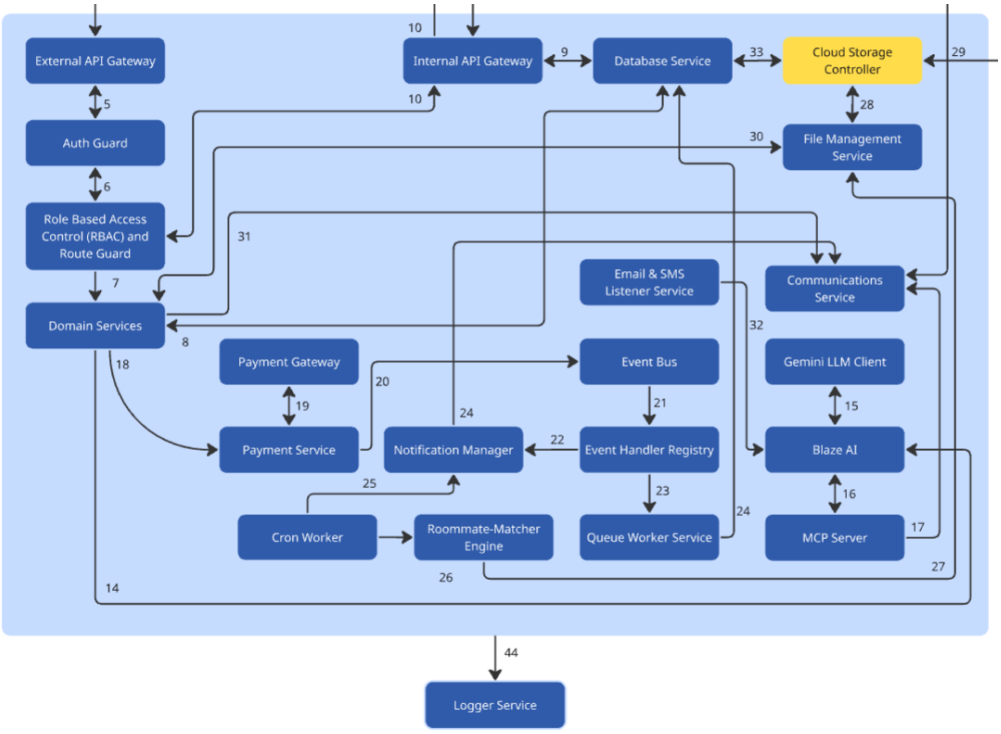

Figure 22: Cloud Storage Controller subsystem

5.10.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or containerized instances; no dedicated hardware is required.

5.10.2 Subsystem Operating System

The subsystem is managed by the cloud provider, typically using Linux-based environments such as Ubuntu or Debian.

5.10.3 Subsystem Software Dependencies

The subsystem relies on the Google Cloud Storage SDK and client libraries for Node.js or TypeScript. It integrates with the File Management Service for metadata updates and with logging and monitoring tools provided by the cloud platform.

5.10.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript and JavaScript for backend API communication and cloud storage operations.

5.10.5 Subsystem Data Structures

Key data structures include file storage request objects, storage response objects, blob metadata objects, and signed URL objects that encapsulate file identifiers, access permissions, and expiration timestamps.

5.10.6 Subsystem Data Processing

The subsystem handles file upload and download requests by formatting API calls for Google Cloud Storage, managing authentication tokens, streaming data, and generating signed URLs. It validates file operations, forwards metadata to the File Management Service for database updates, and includes automatic retry logic for transient network errors to ensure reliability and data integrity.

5.11 Subsystem 8: File Management Service

The File Management Service subsystem is a backend web service responsible for coordinating the lifecycle of all files and documents within the MavHousing system. It manages file metadata, validates file types and sizes, enforces access permissions, and ensures that all files are properly linked to the corresponding database entities. The subsystem acts as an intermediary between Domain Services and the Cloud Storage Controller, handling business logic related to file operations while delegating storage and retrieval tasks to the Cloud Storage Controller.

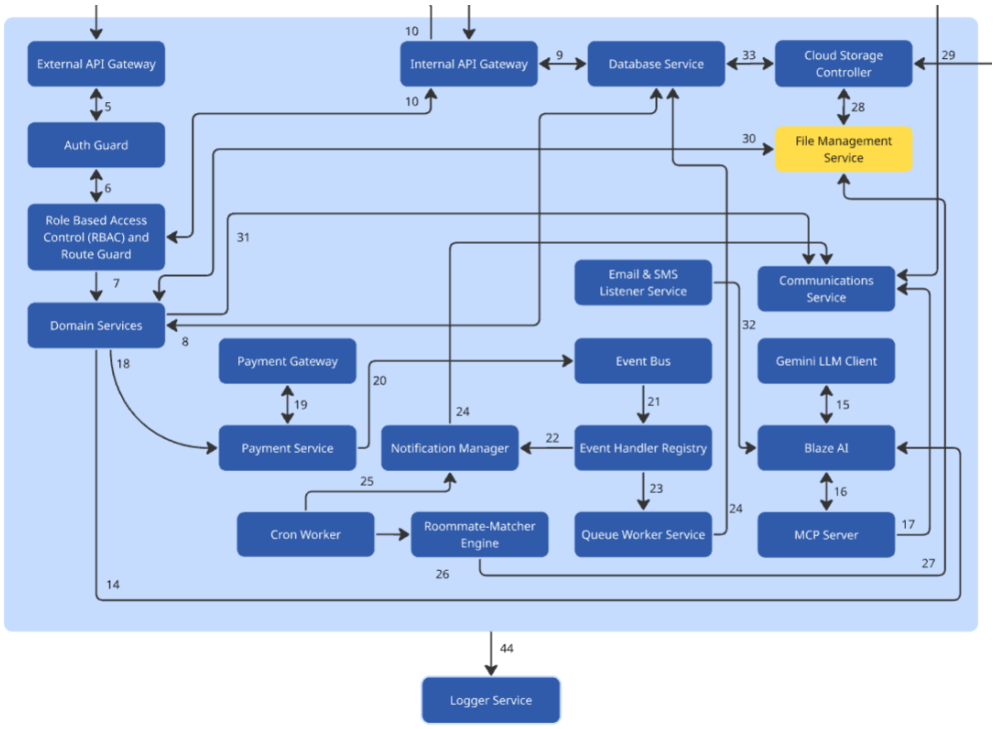

Figure 23: File Management Service subsystem

5.11.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines or containerized instances; no dedicated physical hardware is required.

5.11.2 Subsystem Operating System

It operates on Linux-based server environments managed by the cloud provider, such as Ubuntu or Debian.

5.11.3 Subsystem Software Dependencies

The subsystem depends on the Node.js runtime and the NestJS framework for modular backend services. It integrates with the Cloud Storage Controller SDK for storage operations and uses logging and monitoring tools provided by the cloud environment.

5.11.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript to handle file operations, metadata management, and backend API interactions.

5.11.5 Subsystem Data Structures

Key data structures include file operation request objects, file metadata objects, storage request objects, and file status objects. These encapsulate information such as file identifiers, permissions, size, type, and links to related database entities.

5.11.6 Subsystem Data Processing

The subsystem processes file create, read, update, and delete requests by validating inputs, enforcing access rules, and generating storage requests for the Cloud Storage Controller. It ensures metadata consistency in the database, tracks file lifecycle events, and manages error handling and retries for failed operations.

5.12 Subsystem 9: Payment Gateway

The Payment Gateway subsystem is a backend web service that manages communication with third-party payment processors to handle billing, transactions, and refunds. It securely transmits payment requests, receives responses or callbacks from providers, and standardizes transaction results for internal backend services such as the Payment Service.

5.12.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is required.

5.12.2 Subsystem Operating System

It operates on Linux-based server environments managed by the cloud provider.

5.12.3 Subsystem Software Dependencies

The subsystem depends on Node.js and NestJS for backend services, integrates with third-party payment SDKs (e.g., Stripe, PayPal), and uses secure HTTPS connections for all transactions.

5.12.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for API integration, request handling, and response processing.

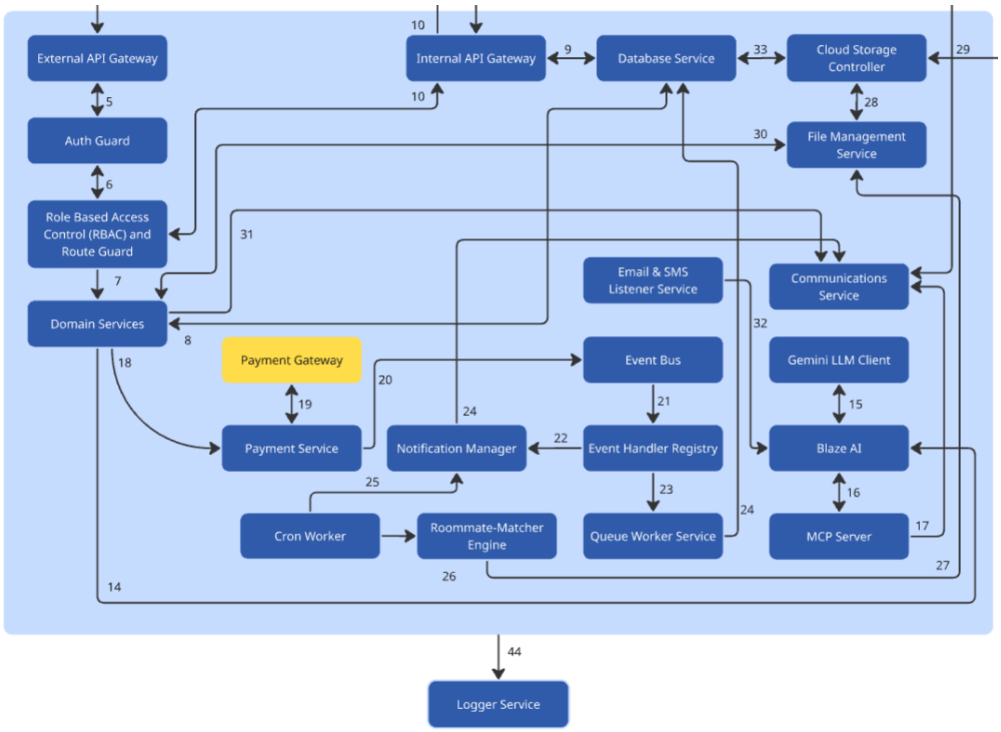

Figure 24: Payment Gateway subsystem

5.12.5 Subsystem Data Structures

Key data structures include payment request objects, transaction response objects, and callback objects containing fields such as amount, currency, status, and transaction ID.

5.12.6 Subsystem Data Processing

The subsystem validates incoming payment requests, formats and sends them to external payment providers, receives and parses provider responses, handles errors or retries, and forwards standardized transaction results to the Payment Service.

5.13 Subsystem 10: Payment Service

The Payment Service subsystem is a backend web service responsible for implementing the business logic related to payments, invoices, and refunds. It validates incoming payment requests, records transactions in the database, triggers notifications, and coordinates with the Payment Gateway to process external transactions securely.

5.13.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is required.

5.13.2 Subsystem Operating System

It operates on Linux-based server environments managed by the cloud provider.

5.13.3 Subsystem Software Dependencies

The subsystem depends on Node.js and NestJS for backend logic, integrates with database services for transaction recording, and interacts with Payment Gateway and Notification Manager APIs.

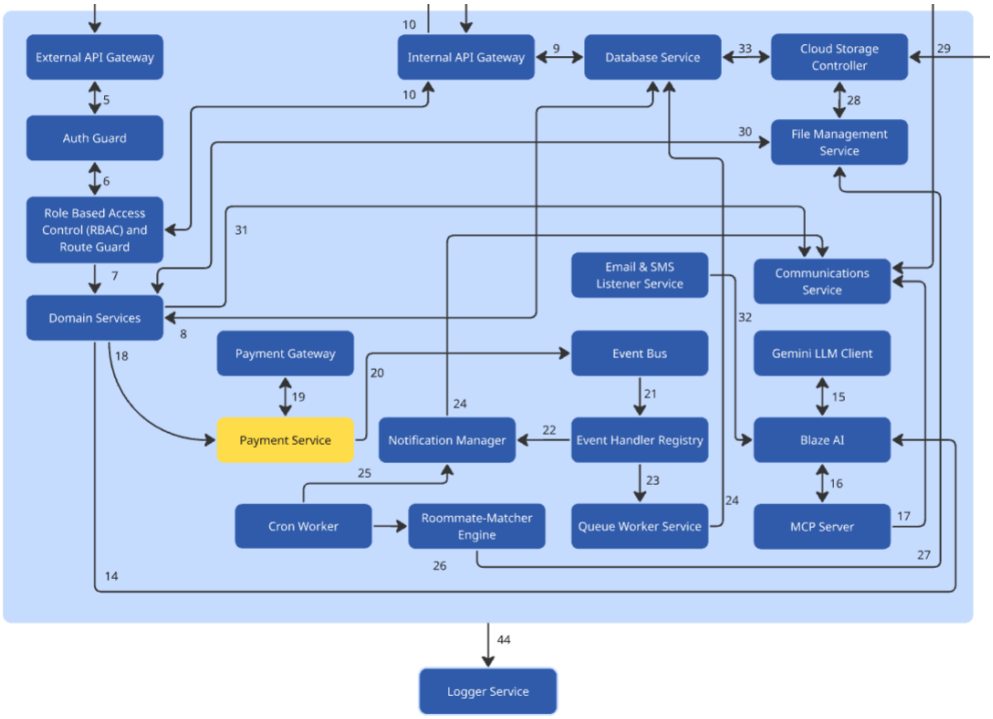

Figure 25: Payment Service subsystem

5.13.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for API handling, business logic, and transaction processing.

5.13.5 Subsystem Data Structures

Key data structures include payment request objects, transaction records, invoice objects, refund objects, and notification events with fields such as amount, status, currency, user ID, and transaction ID.

5.13.6 Subsystem Data Processing

The subsystem validates payment requests, ensures transaction integrity, records payments and refunds in the database, triggers notifications via the Notification Manager, and communicates with the Payment Gateway for external payment processing and status updates.

5.14 Subsystem 11: Notification Manager

The Notification Manager subsystem is a backend web service responsible for orchestrating notifications across multiple channels, including email, SMS, in-app alerts, and push notifications. It manages notification queuing, prioritization, delivery scheduling, and status tracking, ensuring timely communication with students, staff, and administrators while coordinating with the Communications Service for actual message delivery.

5.14.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is required.

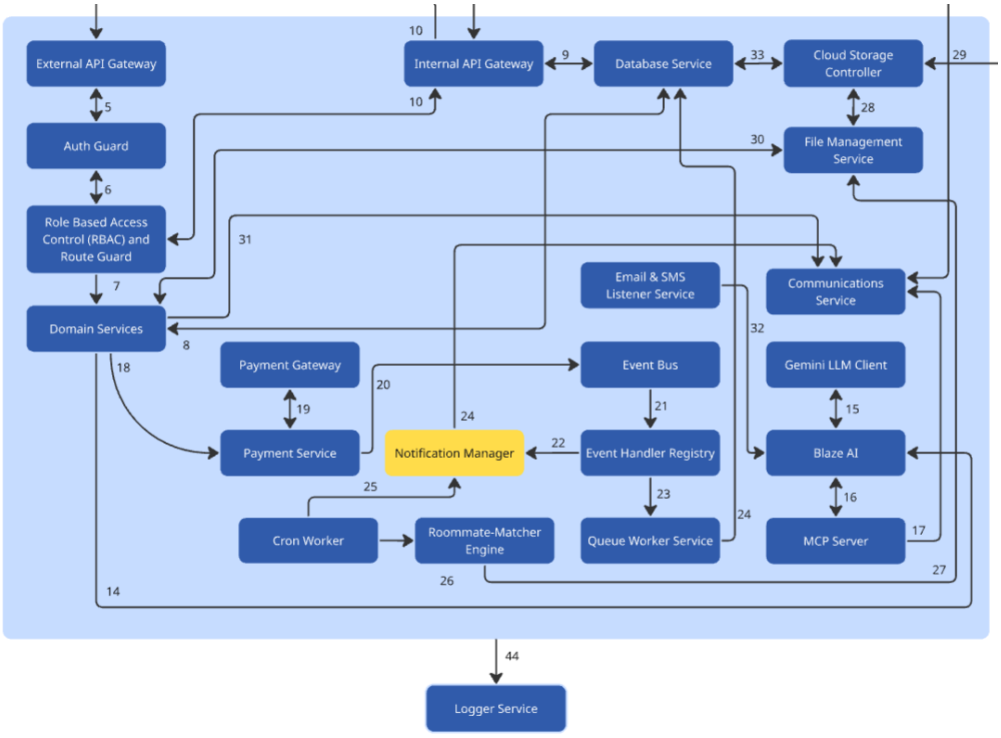

Figure 26: Notification Manager subsystem

5.14.2 Subsystem Operating System

It operates on Linux-based server environments managed by the cloud provider.

5.14.3 Subsystem Software Dependencies

The subsystem depends on Node.js and NestJS for backend logic, and it integrates with the Cron Worker for scheduling, the Event Bus for event-driven communication, and the Communications Service for delivering messages.

5.14.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for handling notification logic, scheduling, and API communication.

5.14.5 Subsystem Data Structures

The subsystem uses notification task objects, scheduled task objects, event objects, and delivery status records. Each object contains fields such as recipient ID, message content, channel type, priority, scheduled time, and delivery status.

5.14.6 Subsystem Data Processing

It queues and prioritizes notifications, schedules delivery based on time or urgency, triggers communication via the Communications Service, batches notifications when appropriate, and updates status logs to track delivery success or failure.

5.15 Subsystem 12: Email & SMS Listener Service

The Email & SMS Listener Service is a backend web service that monitors incoming messages from email and SMS channels. It parses message content, extracts metadata, and forwards relevant information

to services such as Blaze AI for processing, ensuring that user queries or communications are correctly routed within the MavHousing system.

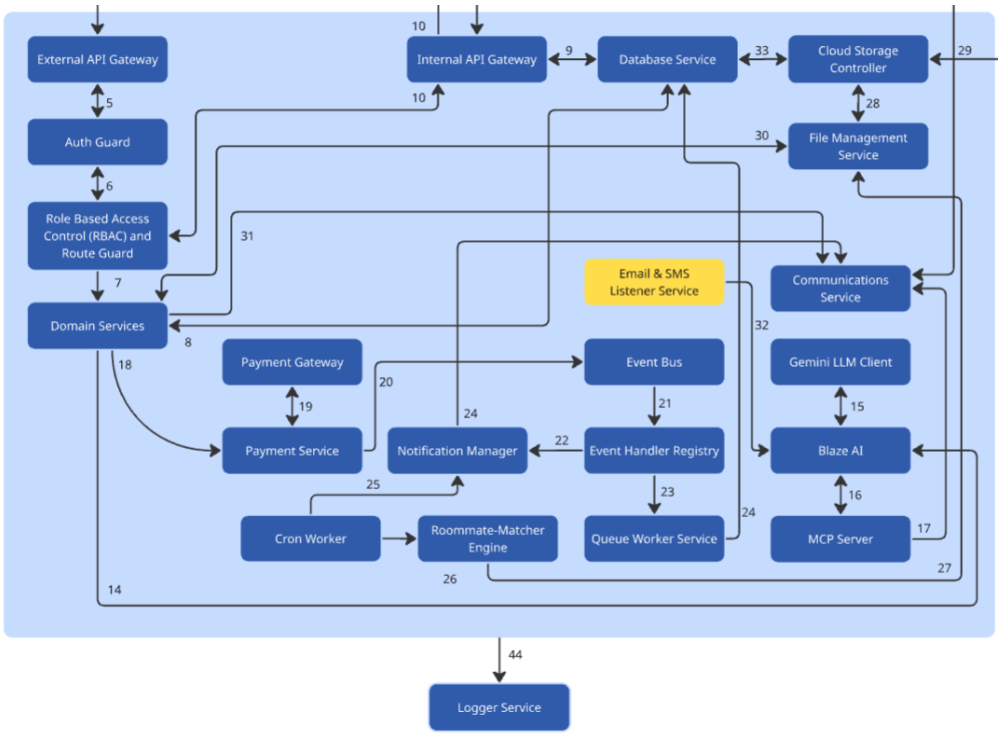

Figure 27: Email & SMS Listener Service subsystem

5.15.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is required.

5.15.2 Subsystem Operating System

It operates on Linux-based environments managed by the cloud provider.

5.15.3 Subsystem Software Dependencies

The subsystem uses Node.js and NestJS for backend logic. It integrates with email and SMS provider SDKs for message retrieval and communicates with Blaze AI or other processing services via internal APIs or messaging queues.

5.15.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for message parsing, routing, and API communication.

5.15.5 Subsystem Data Structures

Handles message objects containing sender, recipient, timestamp, message content, and metadata; routes structured objects representing user queries and contact information to downstream services.

5.15.6 Subsystem Data Processing

Continuously polls or listens for incoming messages, parses content and metadata, classifies messages for processing, routes them to appropriate services, and logs message receipt and routing status for

monitoring and auditing purposes.

5.16 Subsystem 13: Event Bus

The Event Bus is a backend service that acts as a centralized message broker for asynchronous communication between services. It receives events from publishers like Domain Services, Payment Service, and File Management Service, and ensures reliable delivery to all subscriber services, supporting retry mechanisms and event persistence.

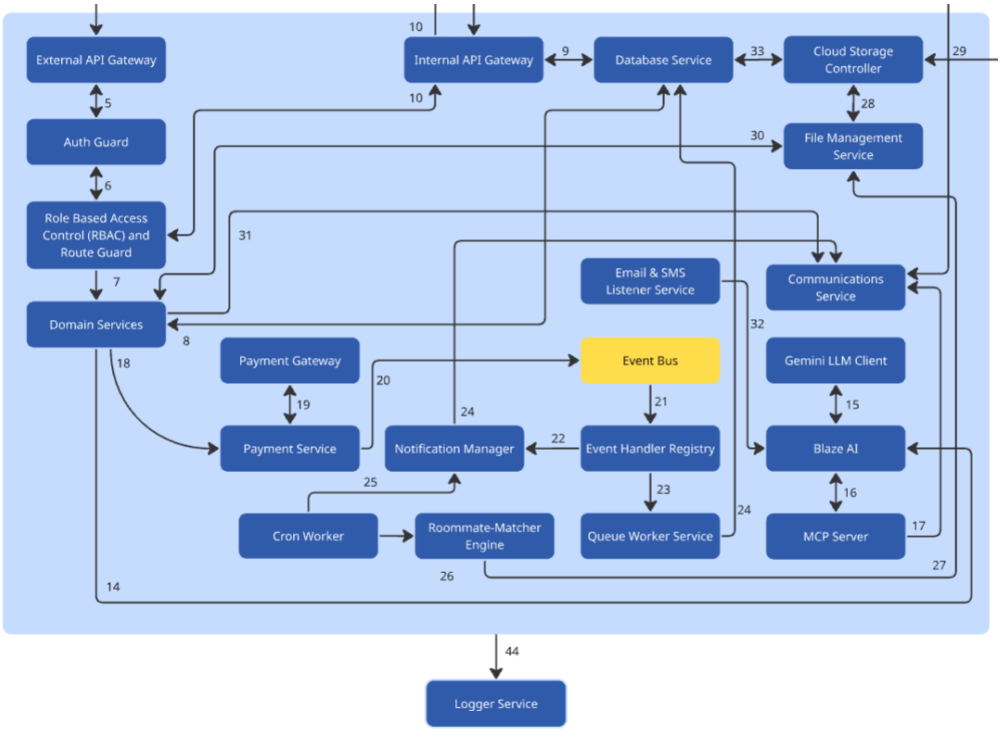

Figure 28: Event Bus subsystem

5.16.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or managed containerized instances; no dedicated hardware is required.

5.16.2 Subsystem Operating System

It operates on Linux-based environments managed by the cloud provider.

5.16.3 Subsystem Software Dependencies

The subsystem depends on messaging frameworks such as RabbitMQ, Kafka, or a managed cloud pub/sub service. It also uses Node.js or other backend runtime for event processing and routing logic.

5.16.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for event handling, queuing, and distribution logic.

5.16.5 Subsystem Data Structures

Events are structured objects containing type, payload, timestamp, source service, and delivery metadata. Subscriber queues or topics store these events until they are successfully processed.

5.16.6 Subsystem Data Processing

The subsystem handles event ingestion, persistence, and distribution to multiple subscribers. It implements retry with exponential backoff for failed deliveries, ensures ordering guarantees where needed, and logs events for monitoring, auditing, and debugging.

5.17 Subsystem 14: Event Handler Registry

The Event Handler Registry is a backend service that maintains a dynamic mapping of event types to their corresponding handler services. It allows services to register or deregister handlers without downtime and provides the Event Bus with routing information to ensure events are delivered to the correct processing services.

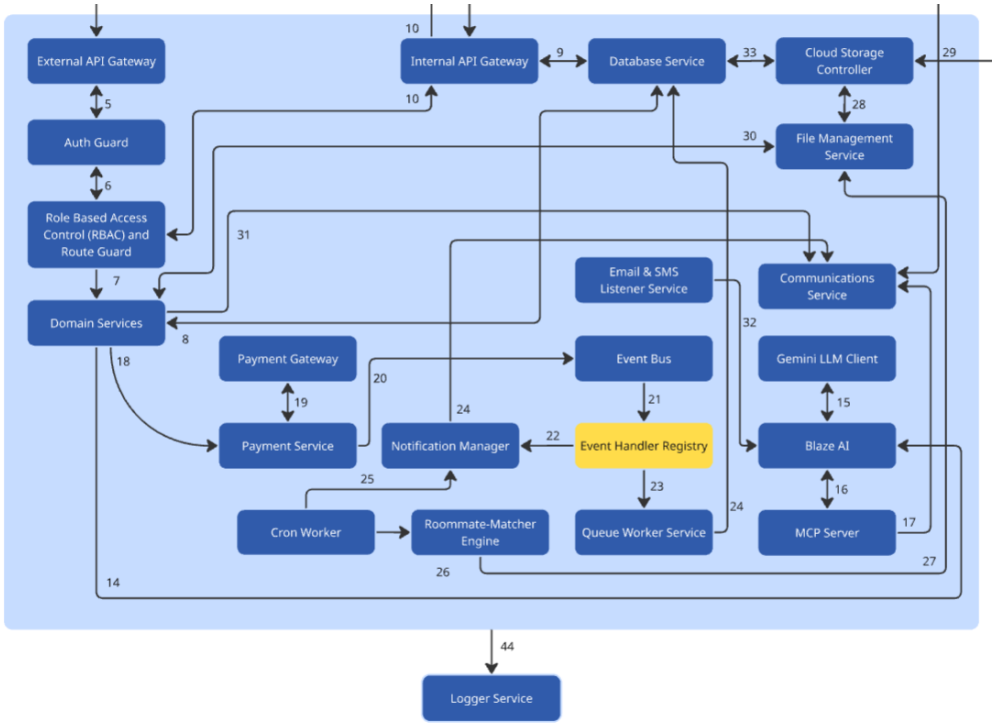

Figure 29: Event Handler Registry subsystem

5.17.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is required.

5.17.2 Subsystem Operating System

It operates on Linux-based environments managed by the cloud provider.

5.17.3 Subsystem Software Dependencies

The subsystem relies on backend frameworks such as Node.js and TypeScript, data storage for the registry (in-memory or persistent storage like Redis), and messaging integration with the Event Bus for subscription updates.

5.17.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for registry management, event mapping, and handler routing logic.

5.17.5 Subsystem Data Structures

The subsystem stores event-handler mappings as objects containing event type, handler service endpoint, subscription metadata, and processing rules. It supports multiple handlers per event type.

5.17.6 Subsystem Data Processing

It processes incoming events from the Event Bus, looks up registered handlers, and routes the events accordingly. The subsystem supports dynamic handler registration, deregistration, and parallel dispatch to multiple subscribers while ensuring reliable forwarding and logging for auditing and monitoring.

5.18 Subsystem 15: Cron Worker

The Cron Worker is a backend service that executes scheduled background jobs at configured intervals. It handles recurring tasks such as sending reminders, generating reports, performing maintenance operations, and cleaning up data.

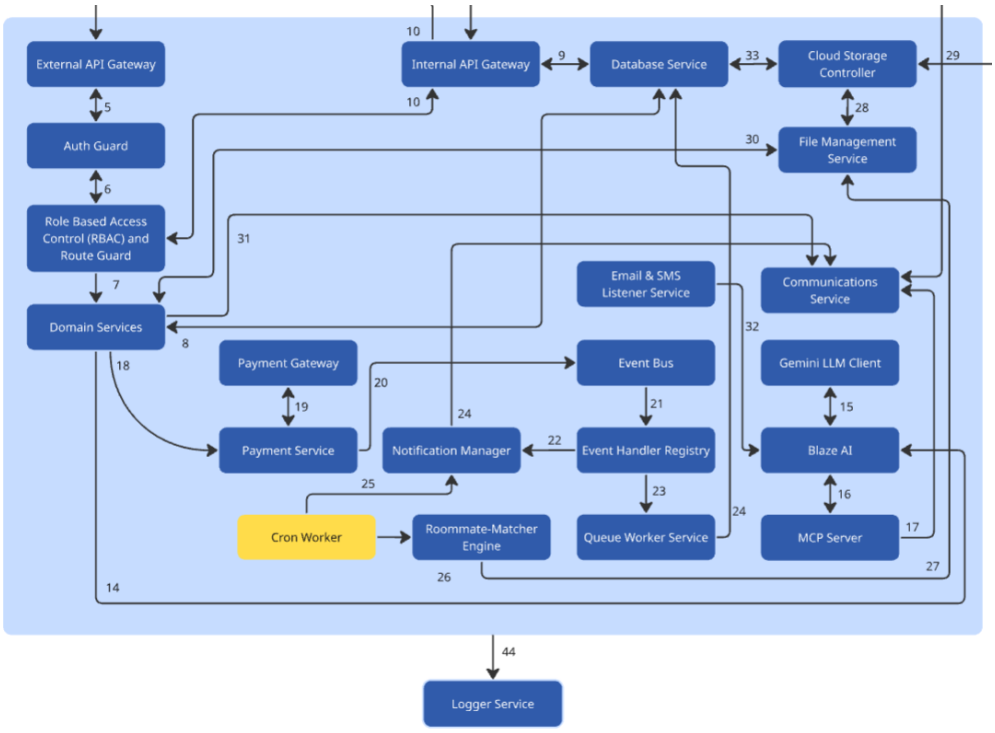

Figure 30: Cron Worker subsystem

5.18.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized instances; no dedicated hardware is needed.

5.18.2 Subsystem Operating System

It operates on Linux-based environments provided by the cloud platform.

5.18.3 Subsystem Software Dependencies

The subsystem depends on backend runtime environments such as Node.js and TypeScript, cron scheduling libraries, logging utilities, and connectivity to other backend services including the Notification Manager and Roommate-Matcher Engine.

5.18.4 Subsystem Programming Languages

It is implemented primarily in TypeScript and JavaScript for scheduling logic, task execution, and monitoring.

5.18.5 Subsystem Data Structures

The subsystem stores job definitions that include cron expressions, task type, payload data, execution metadata, and retry policies. It supports distributed locking flags to prevent duplicate execution.

5.18.6 Subsystem Data Processing

Evaluates scheduled jobs based on cron expressions, triggers tasks at the configured time, monitors execution results, handles retries for failed tasks, and dispatches events or requests to downstream services like Notification Manager and Roommate-Matcher Engine.

5.19 Subsystem 16: Roommate-Matcher Engine

The Roommate-Matcher Engine is a backend service that predicts student roommate compatibility using historical data. It considers preferences, habits, and policies to suggest optimal pairings.

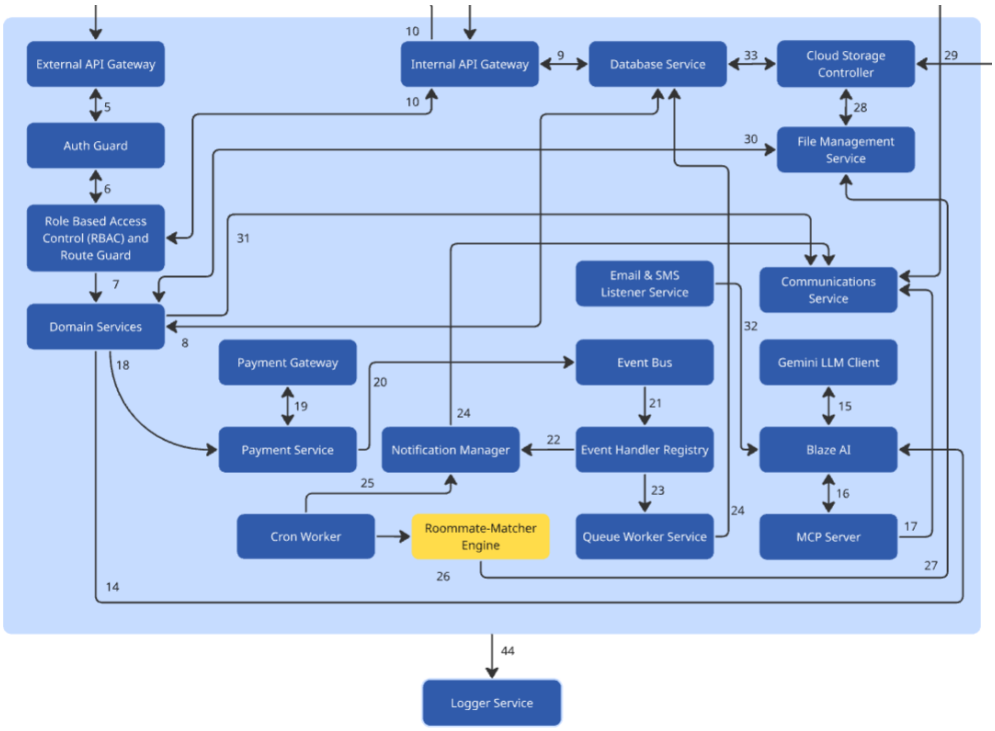

Figure 31: Roommate-Matcher Engine subsystem

5.19.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual machines; no dedicated hardware is required.

5.19.2 Subsystem Operating System

It operates within Linux-based containers or virtual servers.

5.19.3 Subsystem Software Dependencies

The subsystem requires machine learning libraries, data processing tools, and connectivity to the Domain Services and Notification Manager.

5.19.4 Subsystem Programming Languages

It is implemented in Python or TypeScript depending on the ML framework and integration requirements.

5.19.5 Subsystem Data Structures

The subsystem stores student profiles, compatibility scores, match results, and administrator override flags.

5.19.6 Subsystem Data Processing

It analyzes student preferences and historical data, computes compatibility scores, generates roommate match suggestions, and forwards the results to the Notification Manager and Domain Services.

5.20 Subsystem 17: Queue Worker Service

The Queue Worker Service subsystem is responsible for processing jobs and messages from queues. It allows tasks to be executed asynchronously in the background, improving system responsiveness and ensuring that long-running operations do not block real-time services.

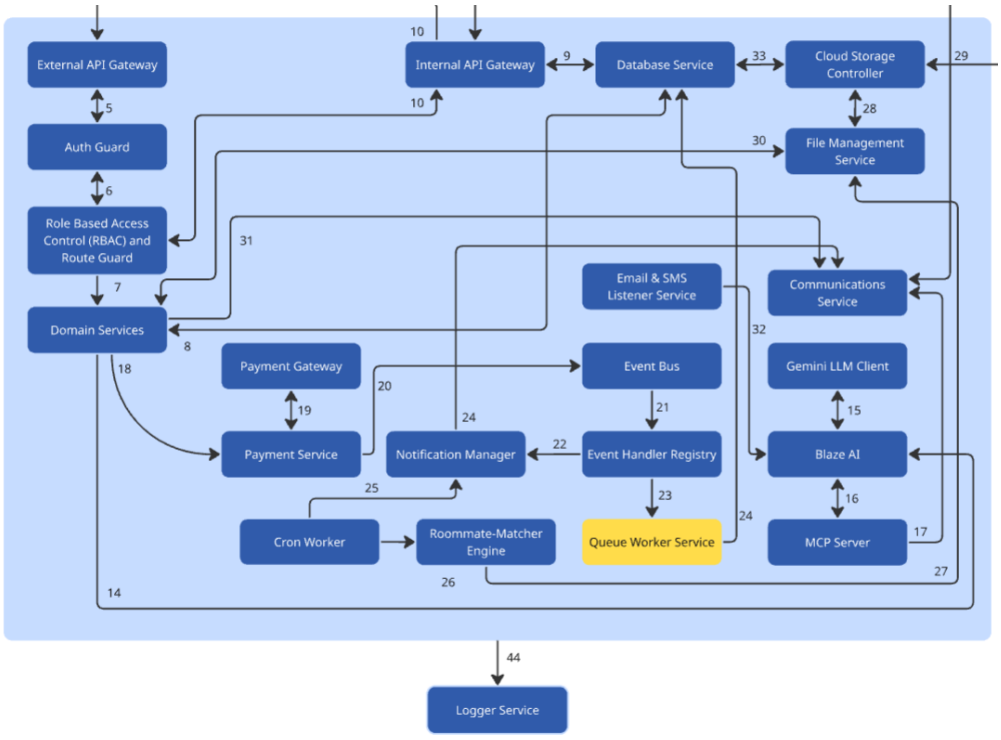

Figure 32: Queue Worker Service subsystem

5.20.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized environments. No dedicated hardware is required.

5.20.2 Subsystem Operating System

It operates on Linux-based containers or virtual machines managed by the cloud provider.

5.20.3 Subsystem Software Dependencies

The subsystem depends on queue management libraries such as RabbitMQ or Redis Queue. It also requires integration with the Event Handler Registry for job retrieval and with the Notification Manager for reporting results.

5.20.4 Subsystem Programming Languages

The Queue Worker Service is implemented primarily in TypeScript or Python, depending on the chosen queue framework and backend integration requirements.

5.20.5 Subsystem Data Structures

The subsystem processes job objects, task messages, event payloads, and result records. Each job object contains information about the task type, input data, and metadata for tracking execution status.

5.20.6 Subsystem Data Processing

The subsystem fetches tasks from the queues and executes them according to the defined logic. After processing, it updates the job status and forwards results or notifications to the Notification Manager and any other dependent subsystems to complete the workflow.

5.21 Subsystem 18: Communications Service

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

5.21.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containerized environments. No dedicated hardware is required.

5.21.2 Subsystem Operating System

It operates on Linux-based virtual machines or containers managed by the cloud provider.

5.21.3 Subsystem Software Dependencies

The subsystem depends on messaging SDKs or APIs for email, SMS, and WebSocket services. It also integrates with the Notification Manager and Domain Services for message generation and delivery.

5.21.4 Subsystem Programming Languages

The Communications Service is primarily implemented in TypeScript or Python, depending on the chosen messaging libraries and backend architecture.

5.21.5 Subsystem Data Structures

The subsystem processes message request objects, including the message template, recipient information, event data, and delivery metadata. It also handles delivery status objects for logging and auditing purposes.

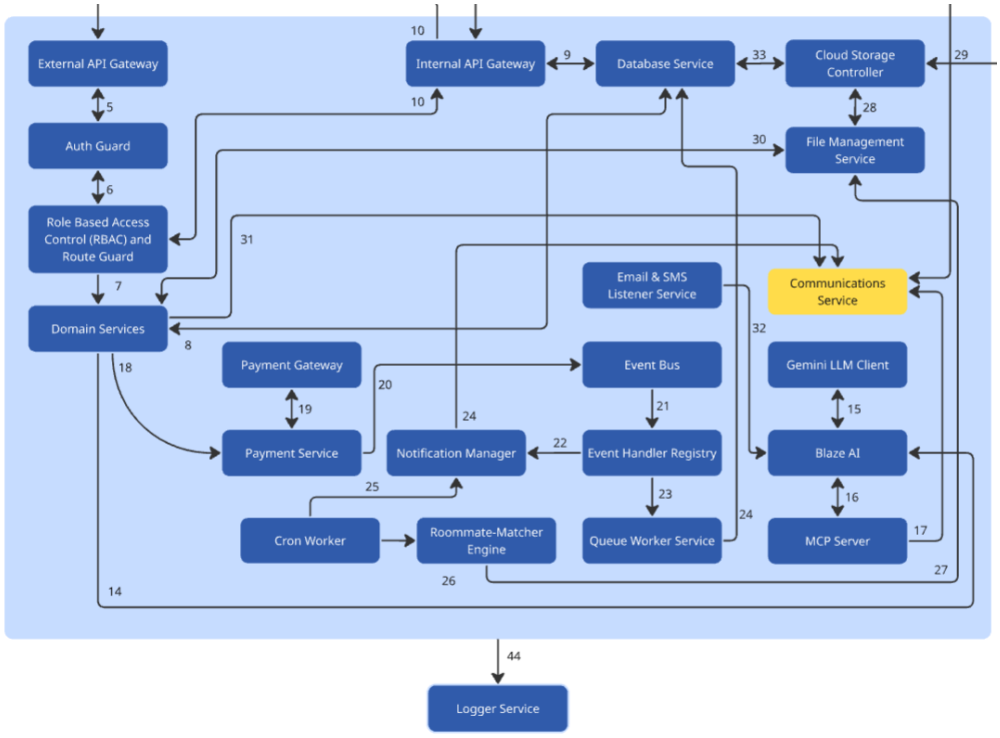

Figure 33: Communications Service subsystem

5.21.6 Subsystem Data Processing

The subsystem receives communication requests from the Notification Manager, Domain Services, and WebSocket clients. It formats messages according to the channel and template, sends them to the respective messaging platform, and logs the delivery status for tracking and audit purposes.

5.22 Subsystem 19: Gemini LLM Client

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 handles API authentication, request formatting, response parsing, and error management for all interactions with the external LLM service.

5.22.1 Subsystem Hardware

The subsystem runs on cloud-hosted virtual servers or containers and does not require dedicated physical hardware.

5.22.2 Subsystem Operating System

It operates on a Linux-based virtual machine or container environment managed by the cloud provider.

5.22.3 Subsystem Software Dependencies

The subsystem depends on HTTP client libraries for API requests, JSON parsers for processing responses, and authentication libraries for secure API key management.

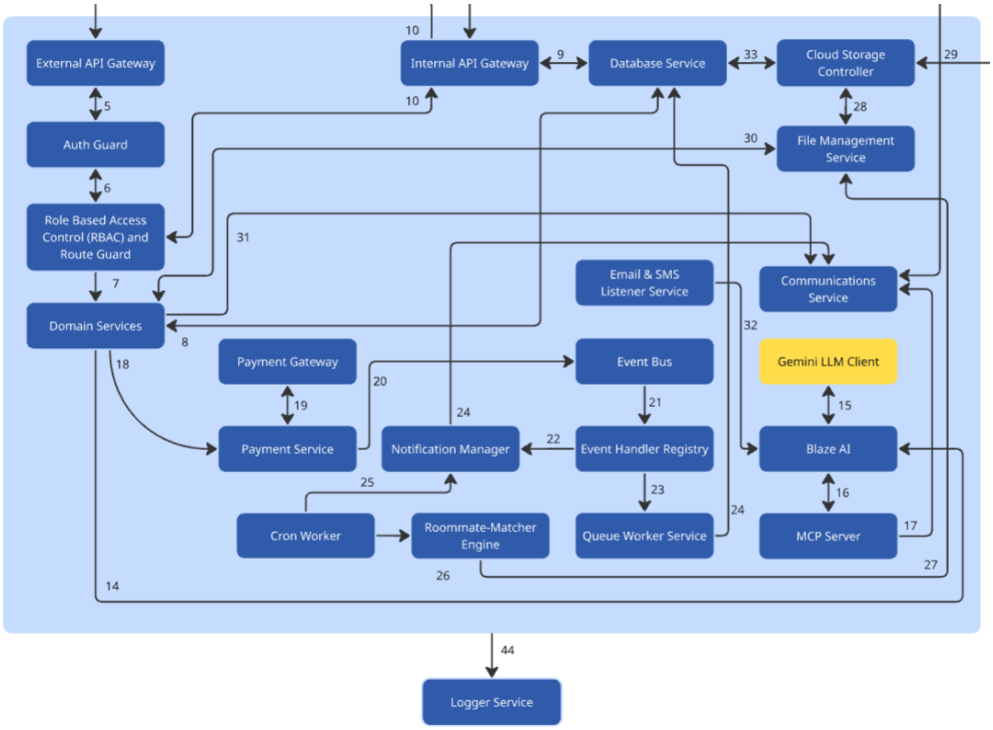

Figure 34: Gemini LLM Client subsystem

5.22.4 Subsystem Programming Languages

The subsystem is primarily implemented in TypeScript or Python, depending on integration requirements with the Blaze AI backend.

5.22.5 Subsystem Data Structures

It processes request objects that contain query data, context information, and user session identifiers. Response objects include the LLM output, confidence scores, and any metadata for logging and debugging.

5.22.6 Subsystem Data Processing

The subsystem formats queries for the Gemini LLM API, submits them, and waits for a response. Upon receiving the response, it parses the output, extracts relevant information, and forwards it to Blaze AI for further processing or storage. Error handling ensures retries or fallback mechanisms in case of failed requests.

5.23 Subsystem 20: Blaze AI

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

5.23.1 Subsystem Hardware

Blaze AI runs on cloud-hosted virtual machines or containerized environments and does not require dedicated physical hardware.

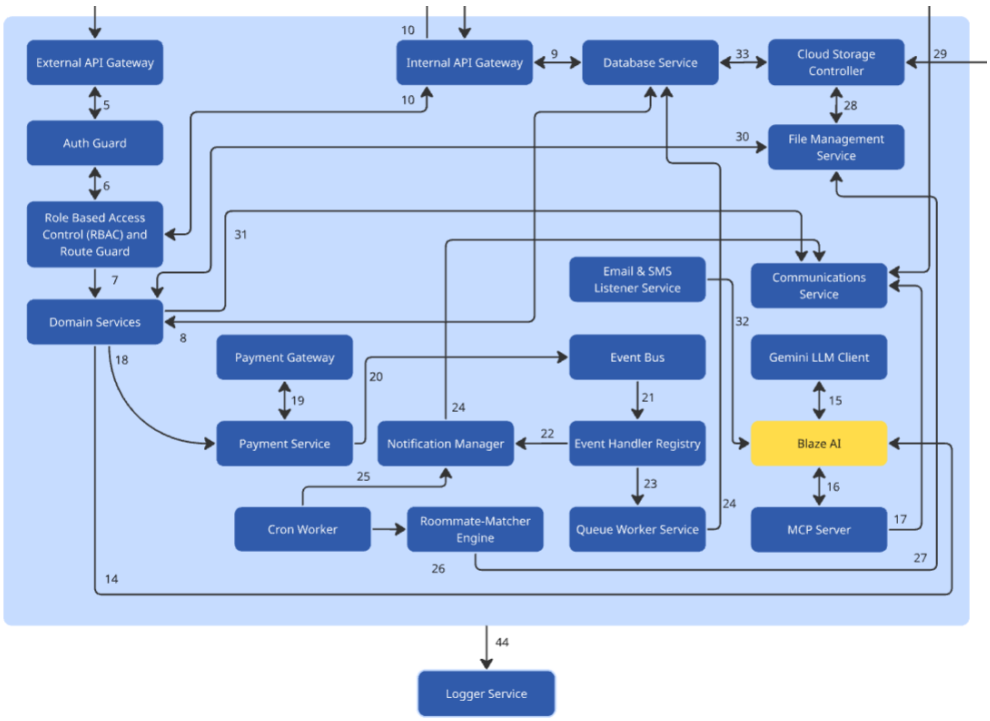

Figure 35: Blaze AI subsystem

5.23.2 Subsystem Operating System

The subsystem operates on Linux-based containers or virtual machines, managed by the hosting cloud provider.

5.23.3 Subsystem Software Dependencies

It relies on the Gemini LLM Client, HTTP/REST libraries for MCP Server communication, JSON parsers, and authentication modules for secure tool access.

5.23.4 Subsystem Programming Languages

The subsystem is implemented primarily in TypeScript and Python to integrate AI processing with backend services.

5.23.5 Subsystem Data Structures

Blaze AI processes structured request objects containing user queries, conversation context, and extracted entities. Response objects include generated natural language answers, metadata about query processing, and any relevant real-time data retrieved via MCP tools.

5.23.6 Subsystem Data Processing

User queries are preprocessed to extract intent and relevant entities before sending to the LLM. Blaze AI forwards queries to the Gemini LLM Client, receives generated responses, and combines them with real-time data from the MCP Server. Responses are then delivered via communications channels or escalated to staff if they cannot be answered automatically. The subsystem maintains conversation context to enable coherent multi-turn interactions and ensures responses are monitored for accuracy.

5.24 Subsystem 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 specialized tools. Each tool exposes functionality such as querying application status, retrieving payment information, or fetching maintenance request details, enabling the AI assistant to generate accurate, context-aware responses based on real-time data.

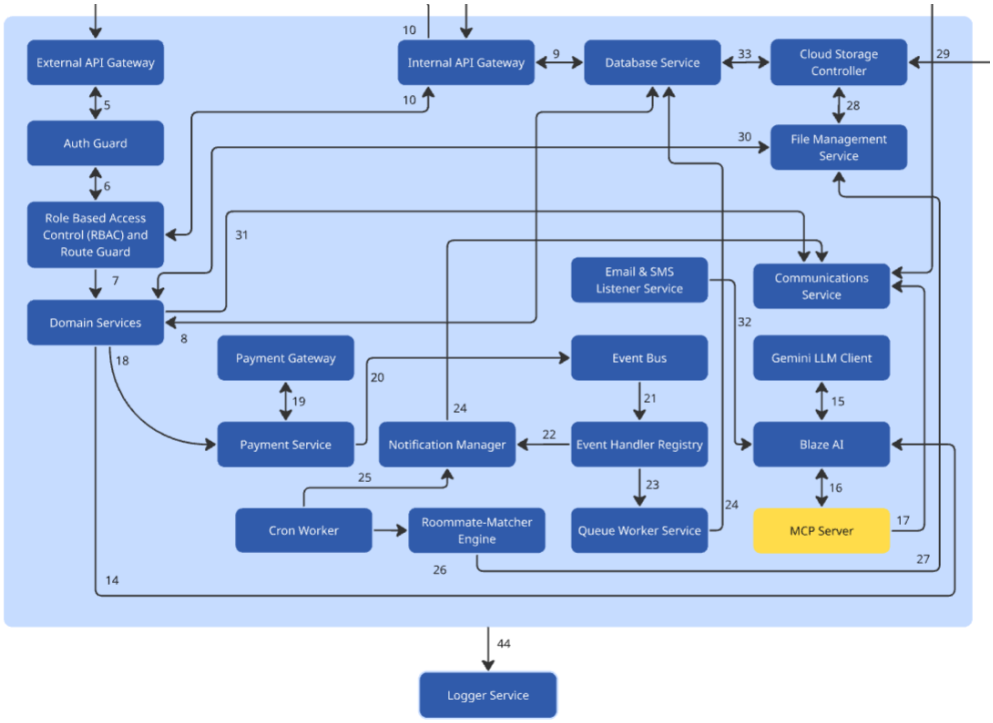

Figure 36: MCP Server subsystem

5.24.1 Subsystem Hardware

The MCP Server runs on cloud-hosted virtual machines or containerized environments and does not require dedicated physical hardware.

5.24.2 Subsystem Operating System

It operates on Linux-based containers or virtual machines provided by the cloud infrastructure.

5.24.3 Subsystem Software Dependencies

The subsystem depends on REST/HTTP libraries for API communication, JSON parsers for structured data handling, authentication and authorization modules, and tool validation libraries.

5.24.4 Subsystem Programming Languages

It is primarily implemented in TypeScript and Python to enable secure API endpoints and integration with backend services.

5.24.5 Subsystem Data Structures

MCP Server handles structured tool invocation objects that include user context, query parameters, and authentication tokens. Response objects contain structured data results, error messages, and metadata

about tool execution.

5.24.6 Subsystem Data Processing

Incoming requests from Blaze AI are validated for authentication and authorization. The server routes each request to the appropriate tool, coordinating calls to backend services such as Database Service, Domain Services, or File Management Service. Responses are formatted according to the MCP schema and returned to Blaze AI. Asynchronous tool executions are queued through the Queue Worker Service when necessary, with status updates and results tracked for reliability.

5.25 Subsystem 22: Logger Service

The Logger Service subsystem provides centralized logging infrastructure for all backend subsystems. It collects, stores, and indexes log entries and performance metrics, enabling monitoring, troubleshooting, security auditing, and performance analysis through structured log aggregation and query capabilities.

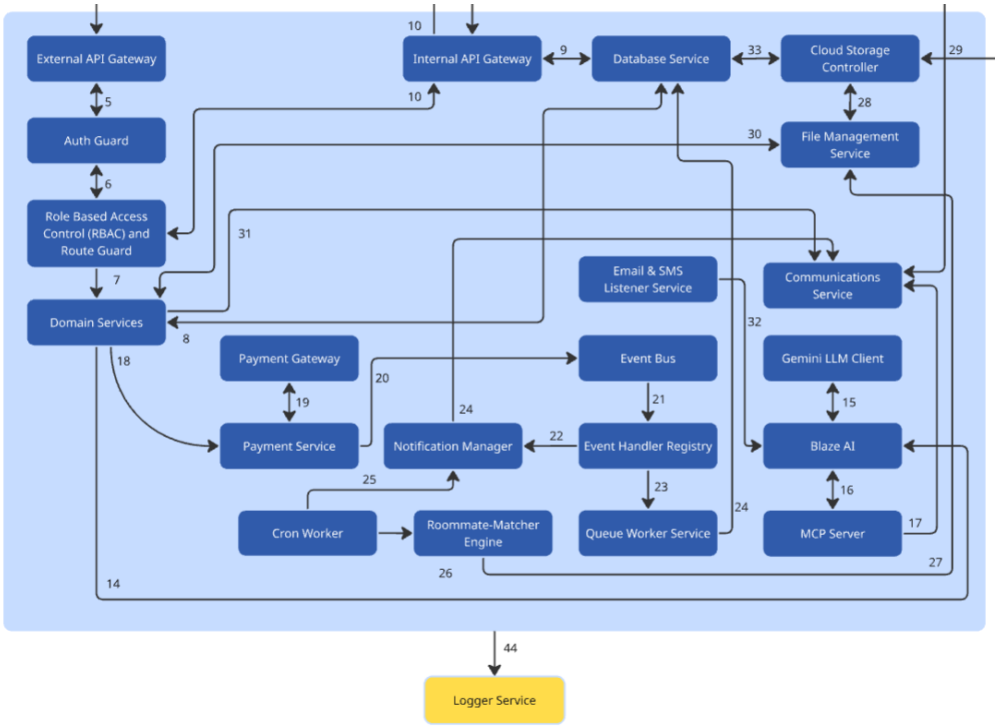

Figure 37: Logger Service subsystem

5.25.1 Subsystem Hardware

The Logger Service runs on cloud-hosted virtual machines or containerized environments and does not require dedicated physical hardware.

5.25.2 Subsystem Operating System

It operates on Linux-based containers or virtual machines provided by the cloud infrastructure.

5.25.3 Subsystem Software Dependencies

The subsystem depends on logging libraries (e.g., Winston, Log4j), time-series databases, indexing tools (e.g., Elasticsearch), and monitoring dashboards for querying and visualization.

5.25.4 Subsystem Programming Languages

It is primarily implemented in TypeScript and Python to enable structured log collection, storage, and retrieval APIs.

5.25.5 Subsystem Data Structures

Logger Service handles log entries and metric reports as structured objects containing timestamp, subsystem identifier, log level, message content, and optional metadata for traceability.

5.25.6 Subsystem Data Processing

Incoming log entries and metrics are validated and parsed before storage. Logs are indexed for fast querying and can be forwarded to monitoring dashboards or alerting systems. The service supports aggregation, filtering, and search of logs and metrics to support debugging, auditing, and performance analysis.

6 APPENDIX A

Include any additional documents (CAD design, circuit schematics, etc) as an appendix as necessary.

References