International Institute of Information Technology, Hyderabad

CS6.401

Software Engineering

MindChain: Right Help, Right Away![1]

Authors

Sanket Adlak
sanket.adlak@students.iiit.ac.in
2024204005

Priyank Nagarnaik
priyank.nagarnaik@students.iiit.ac.in
2024204011

Aditya Singh Rathore
aditya.sing@students.iiit.ac.in
2024204012

Yash Sonkar
yash.sonkar@research.iiit.ac.in
2024801001

Nitheesh Chandra
nitheeshchandra.y@research.iiit.ac.in
2024801002

Supervisors

Karthik Vaidhyanathan
Course Instructor
karthik.vaidhyanathan@iiit.ac.in

Teaching Assistants
https://karthikv1392.github.io/cs6401_se/staff/

[1]Repo https://github.com/nitheesh-me/MindChain & https://github.com/YashSonkar-PhD-CSE/MindChain_Android

2025-04-17

Contents

Introduction

MindChain is motivated by the perennial challenge of timely and accurate assistance within campus communities, where students, researchers, and event organizers at IIIT Hyderabad frequently seek domain-specific expertise. Traditional channels—such as notice boards, email threads, and generic Q&A forums—support asynchronous exchanges but often suffer from low engagement, delayed responses, and difficulty in matching specialized queries to the right experts. While existing platforms (e.g., university helpdesks and third-party collaboration tools) offer basic messaging and ticket-tracking features, they lack tailored mechanisms for incremental discovery and efficient expert discovery in a dynamic academic setting.

To address these limitations, we present MindChain, a community-driven platform that seamlessly connects help-seekers with IIIT Hyderabad experts through advanced query-matching algorithms, configurable notification policies, and integrated real-time chat. Our approach begins with a formal specification of functional requirements—covering user registration, role-based profiles, query submission, matched recommendation workflows, and communication channels—and extra-functional requirements for scalability, reliability, and user experience. We detail architectural patterns such as pub/sub event handling, modular service decomposition, and real-time messaging protocols. A prototype implementation demonstrates support for incremental notifications, expert availability tracking, and intuitive user interfaces. MindChain's holistic design fosters human-centric interactions, reduces response latency, and establishes a reusable blueprint for campus-scale assistance platforms, laying the groundwork for future enhancements and broader deployment.

Exclusions

Certain aspects were intentionally excluded from the scope of this paper to focus on the core objectives. These exclusions include integration with observability tools and error tracking systems such as Sentry, deployment specifications like CPU and memory limits, additional architectural decision records (ADRs), comprehensive low-level design details, and in-depth research on algorithmic approaches. By narrowing the scope, the work prioritizes a clear, actionable blueprint for addressing immediate community needs while leaving room for future refinements and expansions.

Deliverables

Task 1: Requirements & Subsystems

This section documents the specific functional and extra-functional requirements of the MindChain system and provides an overview of its main subsystems.

Functional Requirements

Core functional requirements that define what the MindChain system must/should do.

1. **User Management**

   FR1: User Registration and Authentication

   The system must allow users to register with their institutional credentials and authenticate using IIIT Hyderabad's authentication system.

   Architectural Significance:
   - Integration with existing institutional authentication systems impacts security architecture.
   - User identity is fundamental to the matching algorithm and permission system.
   - Requires secure data handling for personal and academic information.

FR2: User Profile Management
Users must be able to update and manage profiles containing their academic expertise, research interests, courses, and event coordination experience.

Details:
- *Users* must include interested / expert topics
- Faculty profiles could include teaching areas, research interests, and office hours
- Student profiles could include courses taken, research interests, and technical skills
- Staff profiles could include administrative roles and event coordination experience

FR3: Role-Based Access Control
The system must implement attribute-based access control with distinct permissions for students, faculty, and staff.

Details:
- Experts (students / faculty) can view all queries in their domain of expertise
- Seekers can only view their own queries and responses, unless public queries
- Administrators can manage user accounts and system settings

2. **Query Management**

FR4: Query Submission
Users must be able to submit queries through both the web interface and/or app, specifying category, urgency, and detailed description with title.

Architectural Significance:
- Requires multi-platform integration (web and/or app)
- Query structure directly impacts the matching algorithm's effectiveness
- Must support real-time submission and processing

FR5: Expert Matching Algorithm
The system must automatically match queries to the most relevant experts based on expertise, interests, and availability.

Architectural Significance:
- Core functionality that defines the system's value proposition
- Requires complex algorithm design and optimization
- Must balance accuracy with performance considerations
- Needs to adapt and improve based on feedback and usage patterns

FR6: Query Status Tracking
Users must be able to track the status of their queries (pending, matching, matched, resolved, closed).

Details:
- Status updates must be provided in real-time
- Users should receive notifications when their query status changes
- Query history must be maintained for future reference

3. **Communication Management**

FR7: Real-time Chat
The system must provide a real-time chat interface for communication between seekers and experts.

Architectural Significance:
- Requires real-time messaging infrastructure
- Must support persistent conversations across sessions
- Needs to handle message delivery, read receipts, and notifications
- Must be accessible across all pages of the application

FR8: Notification System

The system must provide incremental notifications to experts about matched queries and to seekers about expert responses.

Architectural Significance:
- Requires real-time notification delivery system
- Must support multiple notification channels (in-app, email)
- Needs to implement intelligent notification batching to minimize disruption
- Critical for the expert routing system to function effectively

FR9: Expert Response Management

Experts must be able to view, accept, decline, ~~or defer queries matched to them~~.

Details:
- Experts should see match percentage and criteria for each query
- System must automatically route queries to the next expert if declined or not responded to within a time limit
- Experts should be able to manage multiple active conversations

4. **Analytics and Feedback**

FR10: Query Analytics

The system must provide analytics on query patterns, response times, and resolution rates.

Details:
- Dashboard for administrators showing system-wide metrics
- Personal analytics for experts showing their response rates and feedback
- Query trend analysis to identify common issues and knowledge gaps

FR11: Feedback System

Users must be able to provide feedback on expert responses and the overall query resolution process.

Details:
- Rating system for expert responses
- Feedback forms for the query resolution process
- Mechanism to improve the matching algorithm based on feedback

Extra-functional Requirements

Quality attributes and constraints that define how the MindChain system should operate.

1. **Performance**

EFR1: Response Time

The system must respond to user interactions within ~ 0.3 second under normal load conditions.

Architectural Significance:
- Impacts the choice of database technology and query optimization
- Requires efficient client-server communication protocols

- May necessitate caching strategies for frequently accessed data
- Influences the overall system architecture and component distribution

EFR2: Scalability
The system must support at least 100 and upto 1000 concurrent users and scale horizontally to accommodate growth.

Architectural Significance:
- Requires stateless design for horizontal scaling
- Influences database sharding and partitioning strategies
- Necessitates load balancing and distributed processing
- Impacts the choice of cloud infrastructure and deployment model

2. **Reliability**

EFR3: Availability
The system must maintain 99.9% uptime during academic hours (8 AM to 8 PM) and 99% uptime during non-academic hours.

Details:
- Maximum allowed downtime of 43.8 minutes per month during academic hours
- Scheduled maintenance should be performed during non-academic hours
- System must implement redundancy for critical components

EFR4: Data Durability
The system must ensure no data loss for completed transactions and maintain chat history for at least one academic year.

Details:
- Regular database backups with point-in-time recovery
- Transaction logging and replay capabilities
- Data archiving strategy for older conversations

3. **Security**

EFR5: Data Privacy
The system must protect user data according to institutional privacy policies and applicable regulations.

Architectural Significance:
- Requires end-to-end encryption for sensitive communications
- Necessitates secure data storage with appropriate access controls
- Impacts authentication and authorization mechanisms
- Requires careful handling of personal and academic information

EFR6: Authentication and Authorization
The system must implement secure authentication and role-based authorization with institutional SSO integration.

Details:
- Support for IIIT Hyderabad's authentication system
- Multi-factor authentication for administrative access
- Fine-grained permission system based on user roles
- Secure session management with appropriate timeouts

4. **Usability**

   EFR7: Accessibility
   The system must comply with WCAG 2.1 AA standards to ensure accessibility for users with disabilities.

   Details:
   - Keyboard navigation support for all features
   - Screen reader compatibility
   - Sufficient color contrast and text sizing
   - Alternative text for images and non-text content

   EFR8: User Experience
   The system must provide an intuitive, responsive interface that minimizes cognitive load and supports efficient task completion.

   Architectural Significance:
   - Requires responsive design and component-based UI architecture
   - Impacts the choice of frontend framework and state management
   - Necessitates efficient client-side rendering and data fetching
   - Influences the design of notification and chat systems

5. **Maintainability**

   EFR9: Extensibility
   The system must be designed to easily incorporate new features, support multi-tenancy to serve multiple independent tenants with data isolation, and integrate seamlessly with additional institutional systems.

   Details:
   - Modular architecture with well-defined interfaces
   - API-first design for all core functionality
   - Plugin system for extending capabilities
   - Comprehensive documentation for developers

   EFR10: Testability
   The system must be designed to facilitate automated testing at unit, integration, and system levels.

   Details:
   - Test coverage of at least 80% for core functionality
   - Support for mocking external dependencies
   - Continuous integration and automated test execution
   - Monitoring and logging for production issues

Subsystem Overview

Breaks down the primary subsystems, explaining their roles and how they integrate to form the cohesive MindChain platform.

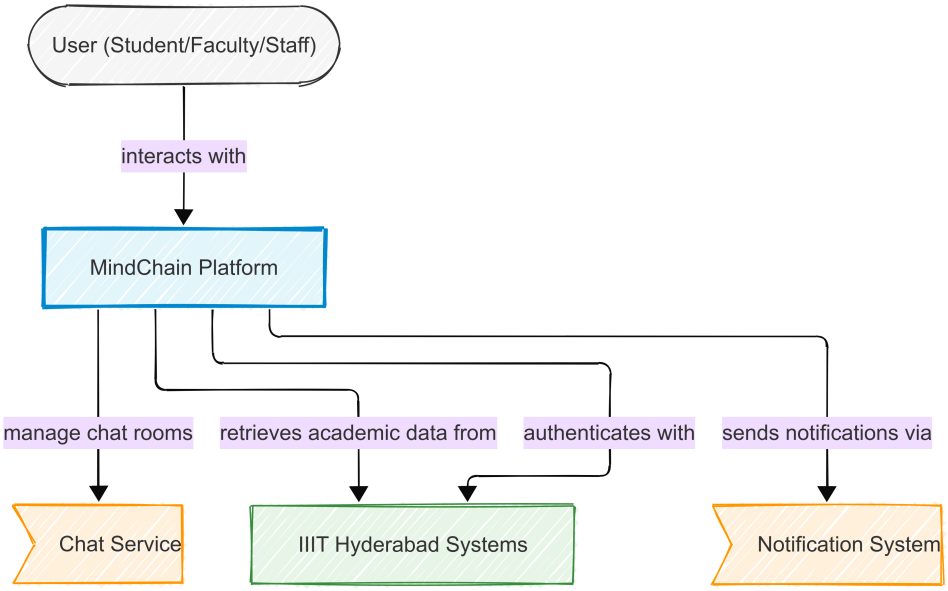

Figure 1: high-level system context diagram showing MindChain and its external interactions

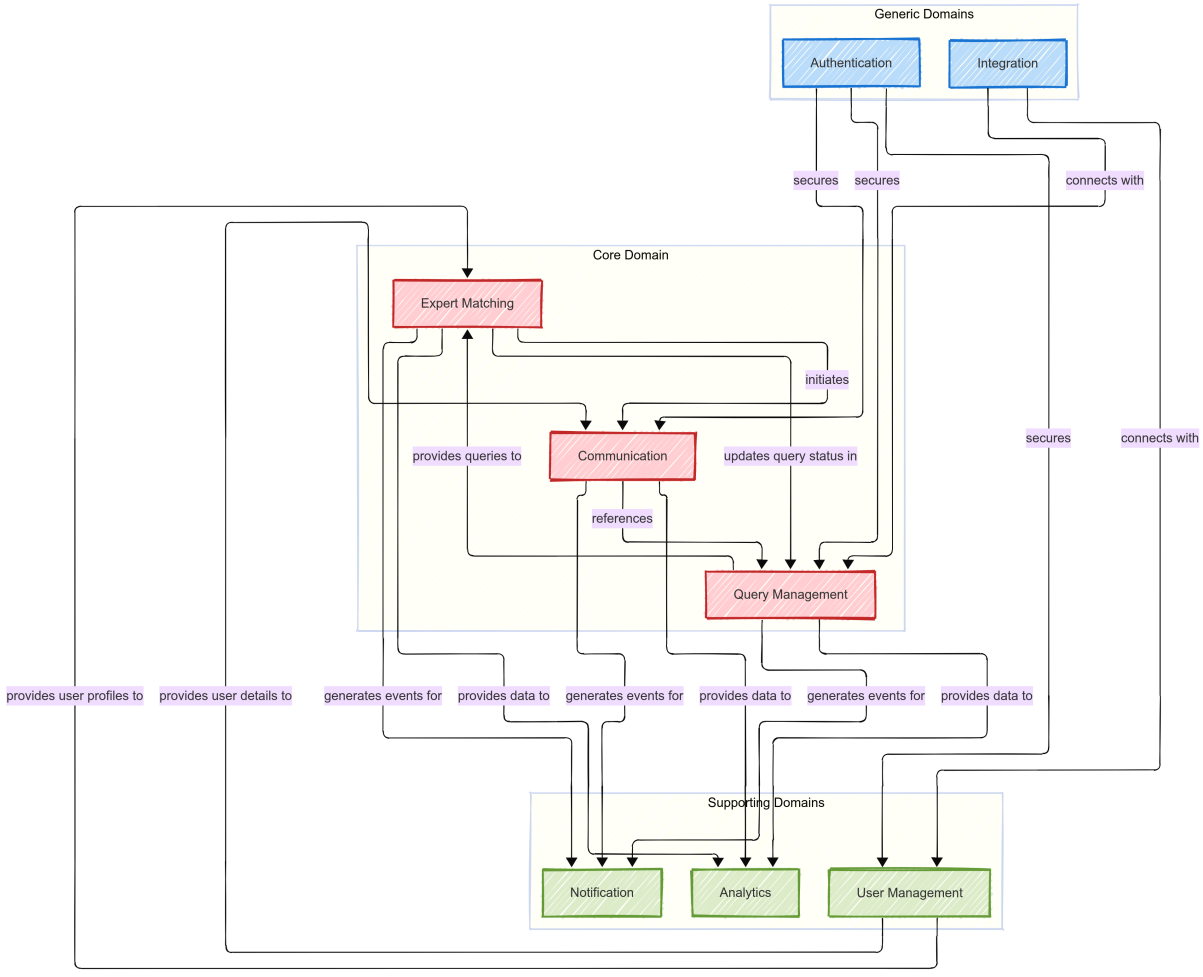

Figure 2: Strategic Domain Model: Bounded Context Map and their relations

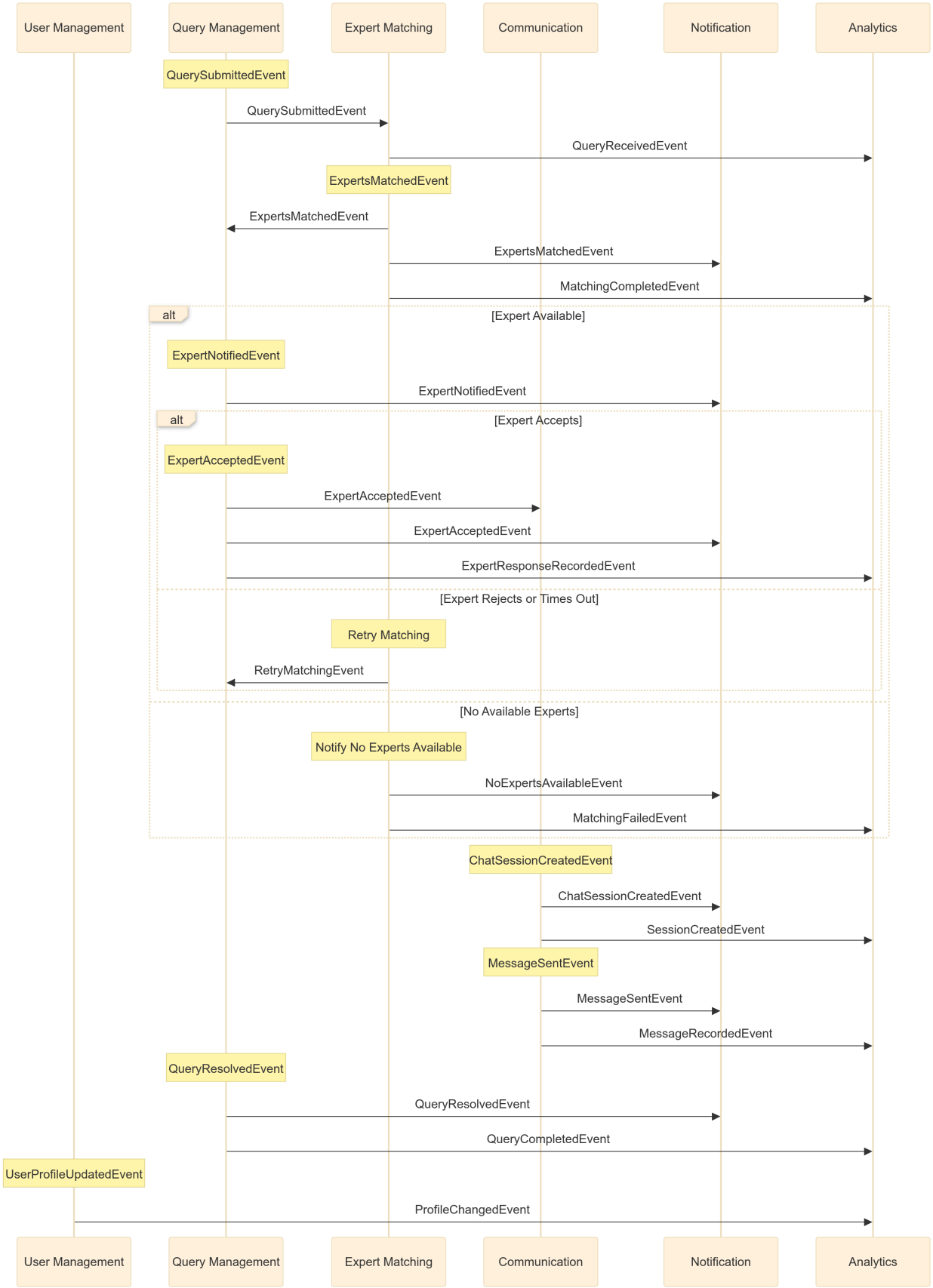

Figure 3: High level Event Flow: domain events flow between bounded contexts:

1. **SS0: Authentication Subsystem**

   The Authentication Subsystem ensures secure access to the MindChain platform by managing user authentication and session handling. It integrates with institutional authentication systems and supports multi-factor authentication (MFA) for enhanced security.

   1. *Key Components:*
      - Login Service: Handles user login requests and validates credentials
      - Session Manager: Manages user sessions, including token generation and expiration
      - MFA Service: Implements multi-factor authentication for added security
      - Authentication Gateway: Acts as a unified entry point for authentication requests
      - Audit Logger: Tracks authentication events for security and compliance
   2. *Interfaces:*
      - To User Interface: Provides login and MFA interfaces
      - To Institutional Authentication System: Integrates with OAuth -SAML for institutional SSO
      - To User Management Subsystem: Validates user roles and permissions
      - To Security Subsystem: Shares authentication logs for monitoring and threat detection

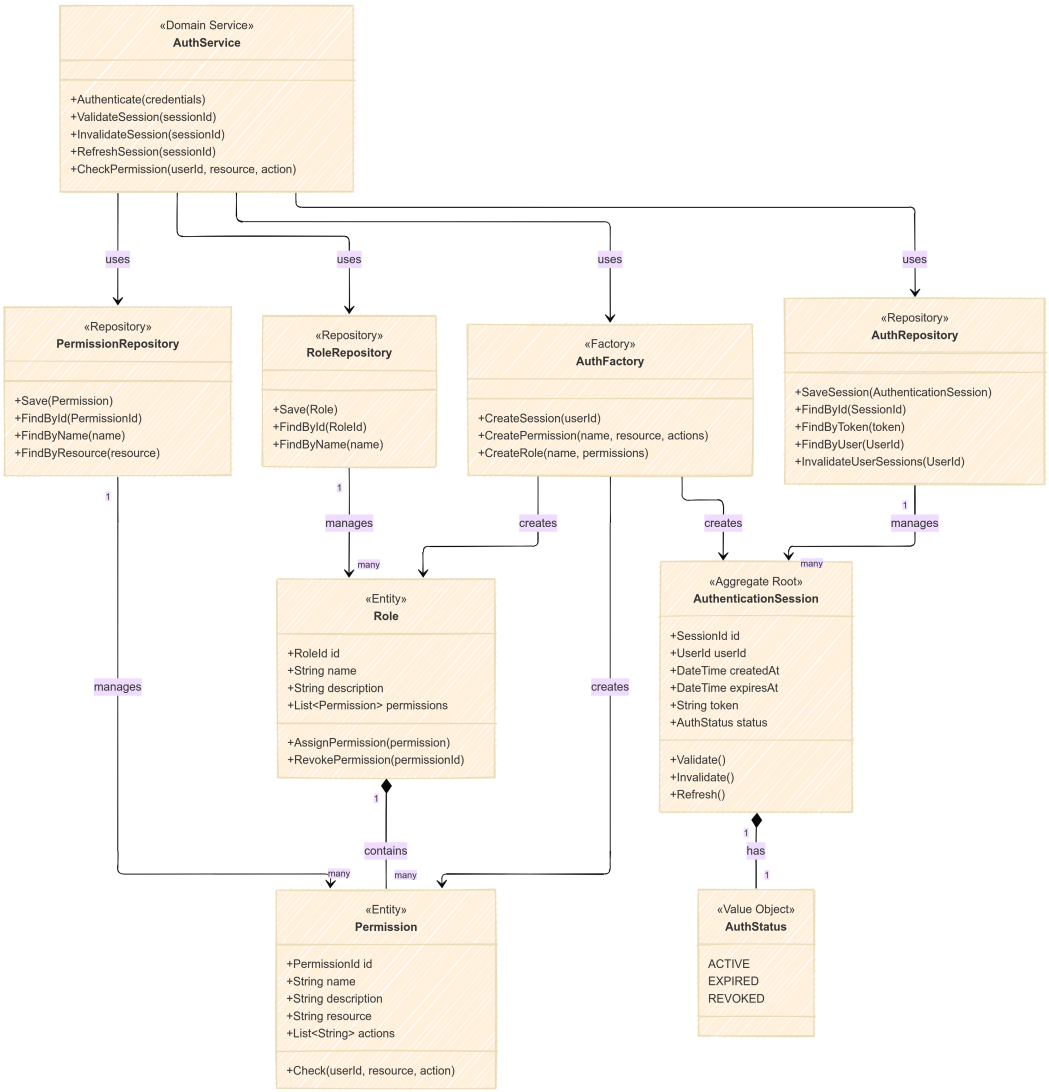

Figure 4: Authentication Subsystem : Detailed Bounded Contexts

2. **SS1: User Management Subsystem**

   The User Management Subsystem handles user authentication, profile management, and Attribute-based access control (ABAC). It integrates with IIIT Hyderabad's authentication system and maintains user profiles with academic expertise, research interests, and other relevant information.

   1. *Key Components:*
      - Authentication Service: Handles user login, session management, and integration with institutional SSO
      - Profile Manager: Manages user profiles, expertise data, and preferences
      - Authorization Service: Implements Attribute-based access control and permission management
      - User Data Store: Securely stores user information and credentials
   2. *Interfaces:*
      - To Authentication System: OAuth/SAML integration with IIIT Hyderabad's authentication
      - To Query Matching Subsystem: Provides expertise data for matching algorithm
      - To Communication Subsystem: Provides user identity and contact information

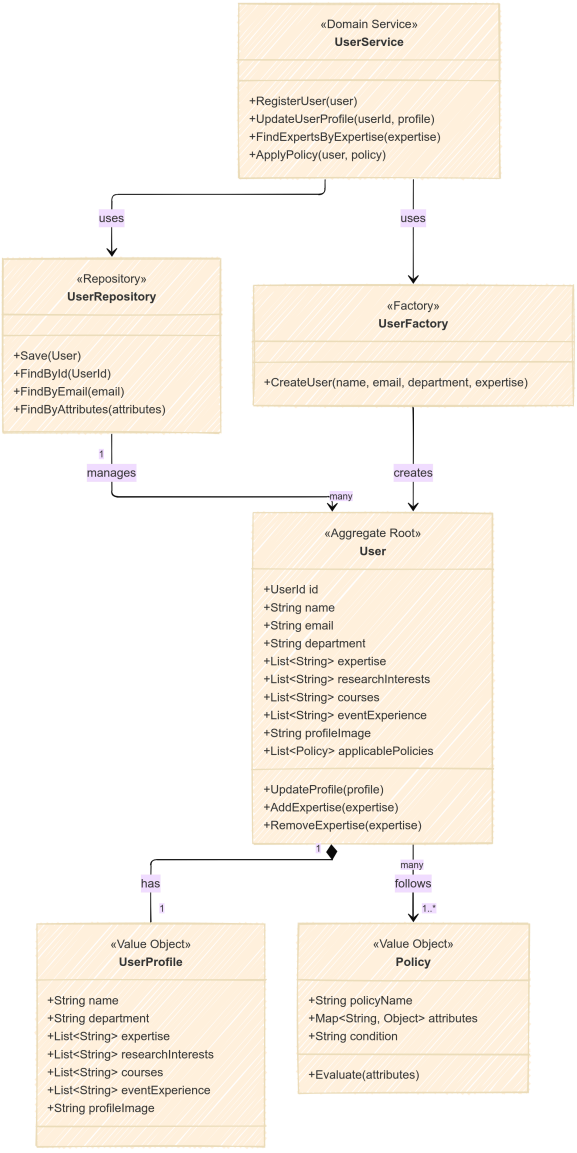

Figure 5: User management Subsystem : Detailed Bounded Contexts

3. **SS2: Query Management Subsystem**
   The Query Management Subsystem handles the creation, tracking, and resolution of user queries. It manages the lifecycle of queries from submission to resolution, including categorization, prioritization, and status tracking.

   1. *Key Components:*
      - Query Submission Service: Processes query submissions from web and app interfaces
      - Query Tracker: Manages query status and lifecycle
      - Category Manager: Handles query categorization and tagging
      - Query Data Store: Stores query details, status, and history
   2. *Interfaces:*
      - To User Interface: Provides query submission and tracking functionality
      - To Expert Matching Subsystem: Sends queries for expert matching
      - To Analytics Subsystem: Provides query data for analysis

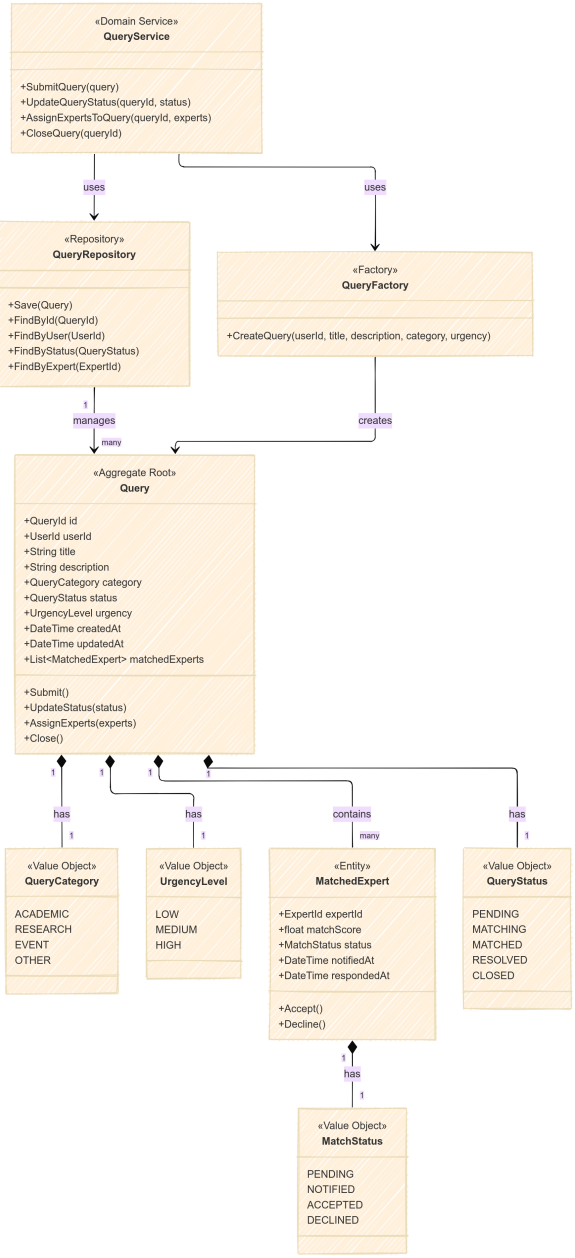

Figure 6: Query management Subsystem : Detailed Bounded Contexts

4. **SS3: Expert Matching Subsystem**
   The Expert Matching Subsystem is the core intelligence of MindChain, matching queries to the most relevant experts based on expertise, research interests, and availability. It implements sophisticated algorithms to ensure optimal matching and handles expert notification and response management.

   1. *Key Components:*
      - Matching Engine: Implements the core matching algorithm
      - Expertise Analyzer: Processes and indexes expertise data
      - Expert Router: Manages the routing of queries to experts, including fallback mechanisms
      - Availability Manager: Tracks expert availability and workload
   2. *Interfaces:*
      - To Query Management Subsystem: Receives queries for matching
      - To User Management Subsystem: Accesses expertise data
      - To Notification Subsystem: Triggers expert notifications
      - To Analytics Subsystem: Provides matching performance data

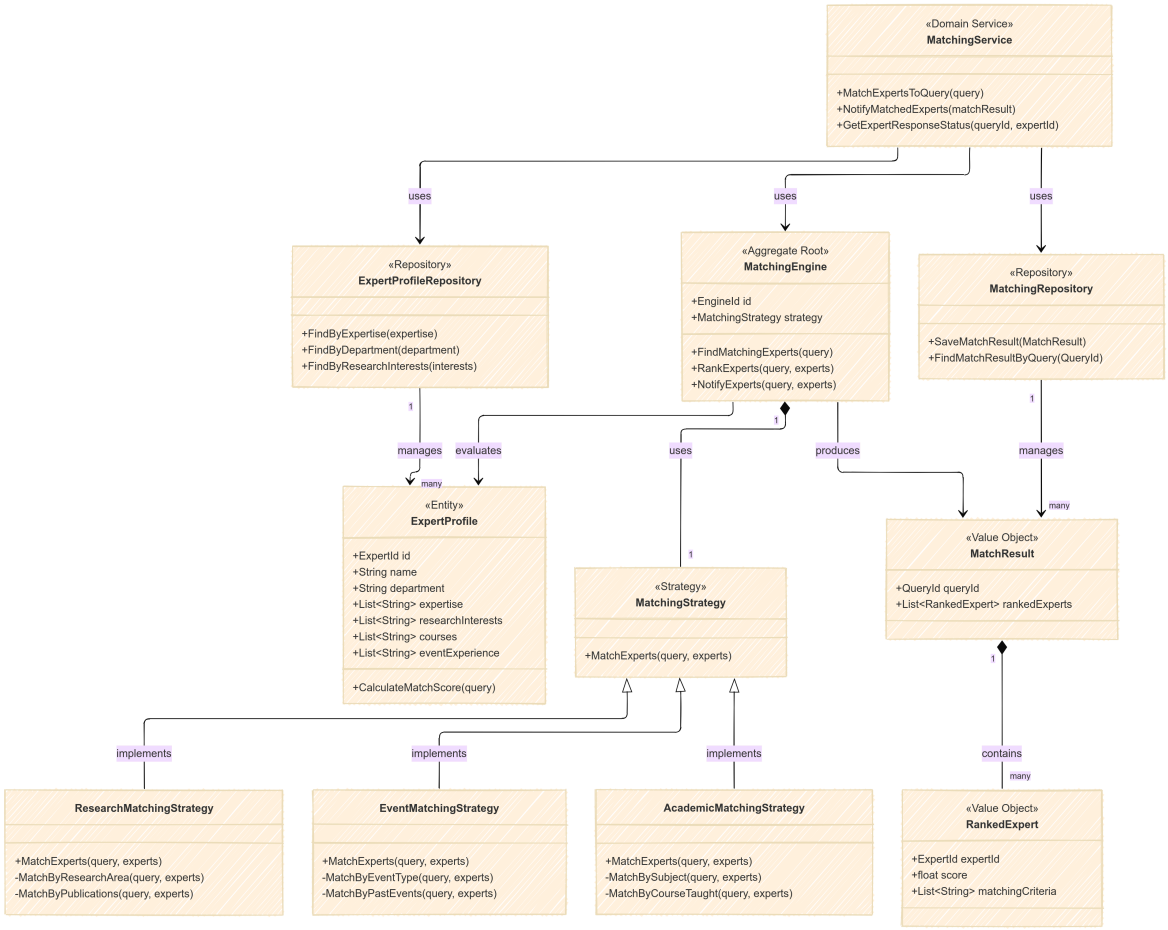

Figure 7: Expert Matching Subsystem : Detailed Bounded Contexts

5. **SS4: Communication Subsystem**
   The Communication Subsystem handles all real-time and asynchronous communication between users and experts. It provides chat functionality, message delivery, and conversation management across the platform.

   1. *Key Components:*
      - Chat Service: Manages real-time messaging between users

- Message Store: Persists chat history and message data
- Conversation Manager: Handles conversation state and metadata
- Read Receipt Service: Tracks message delivery and read status

2. *Interfaces:*
   - To User Interface: Provides chat UI components and real-time updates
   - To Notification Subsystem: Triggers message notifications
   - To Query Management Subsystem: Updates query status based on communication

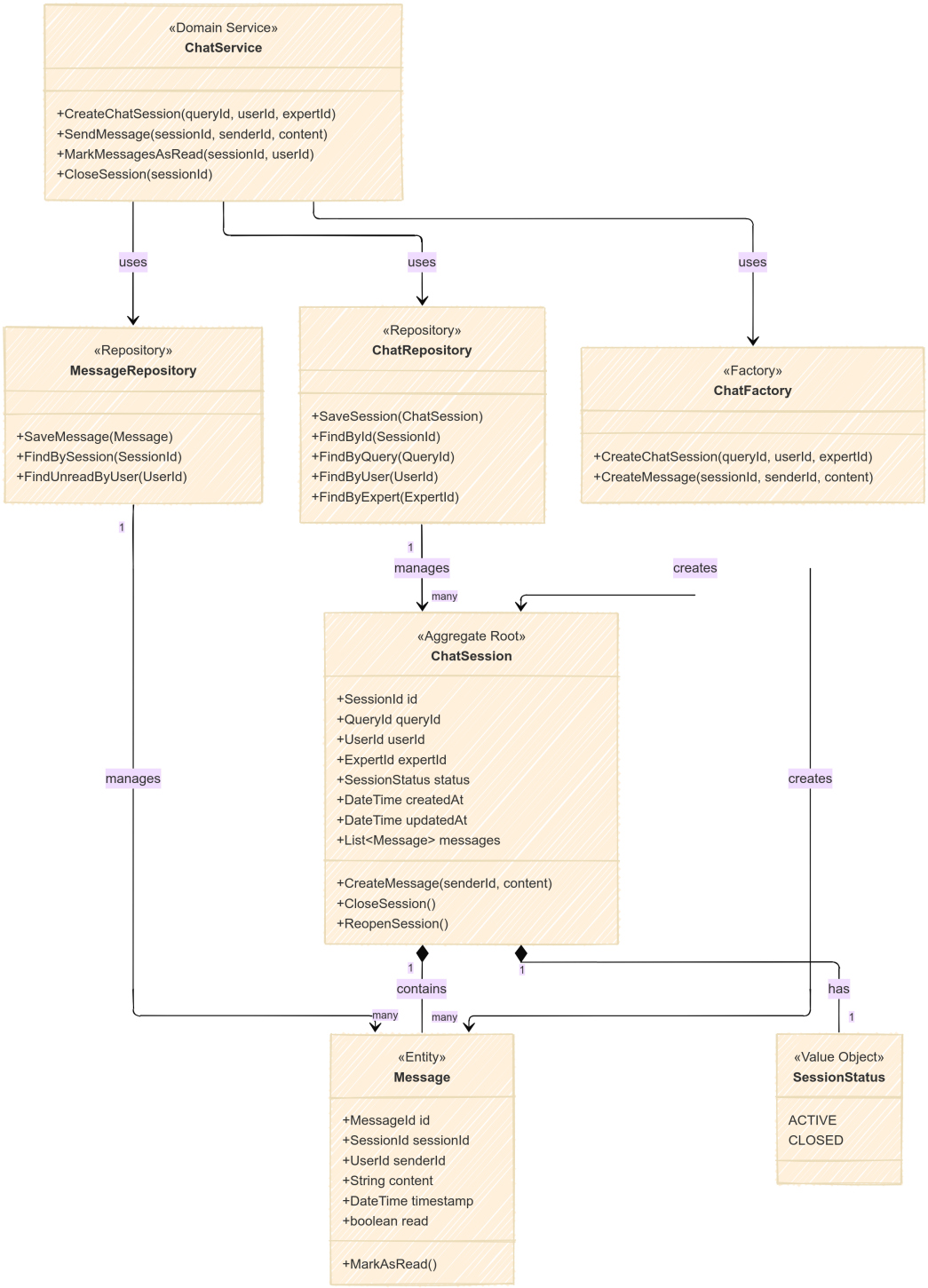

Figure 8: Communication Subsystem : Detailed Bounded Contexts

6. **SS5: Notification Subsystem**

   The Notification Subsystem manages the delivery of notifications to users across multiple channels. It implements intelligent notification batching to minimize disruption while ensuring timely delivery of important information.

   1. *Key Components:*
      - Notification Manager: Coordinates notification creation and delivery
      - Channel Adapters: Handles delivery through different channels (in-app, email)
      - Batching Service: Implements intelligent notification grouping and timing
      - Notification Store: Persists notification history and status
   2. *Interfaces:*
      - To User Interface: Delivers in-app notifications
      - To Expert Matching Subsystem: Receives expert match events
      - To Communication Subsystem: Receives message events
      - To External Services: Connects to email and push notification services

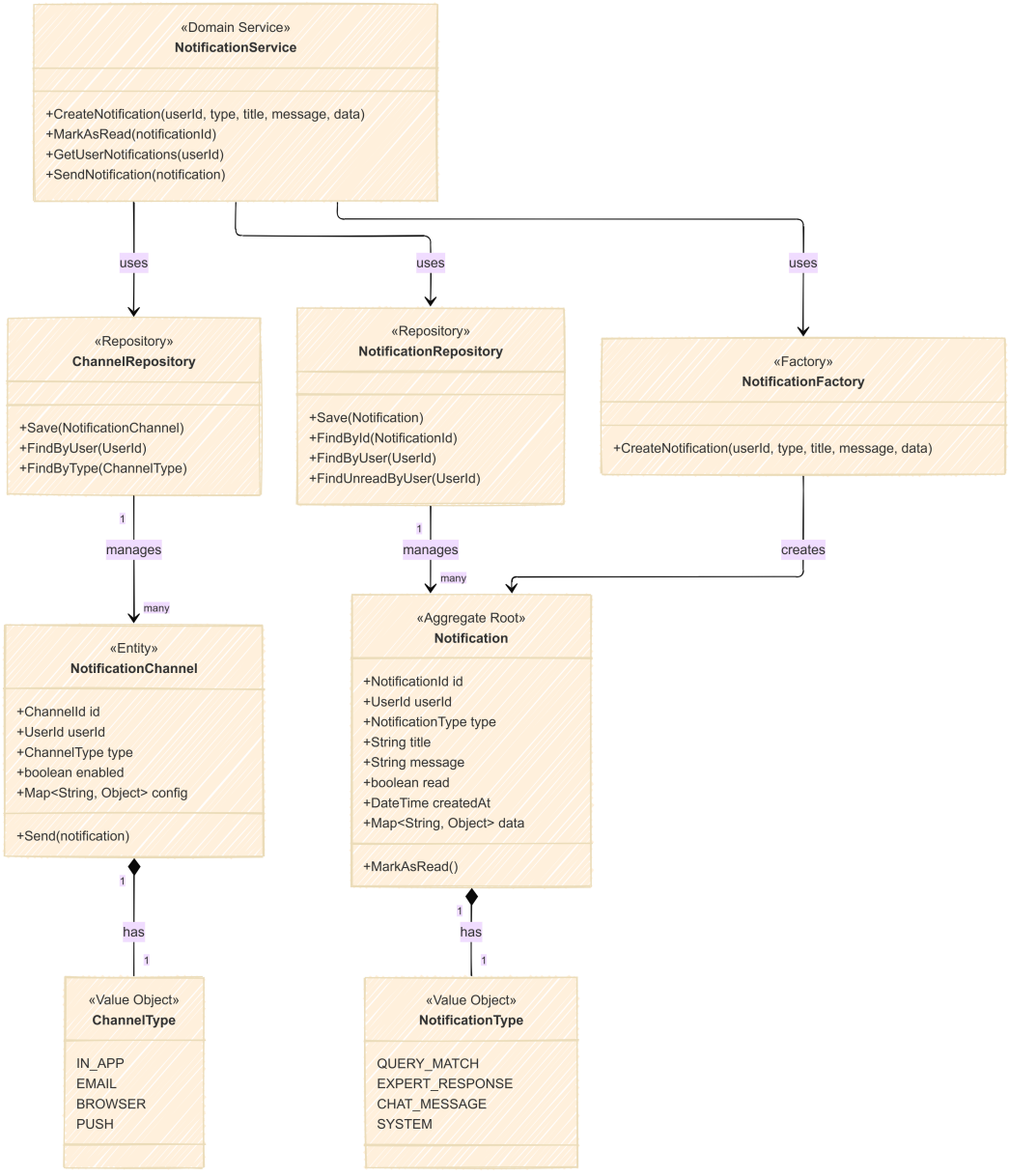

Figure 9: Notification Subsystem : Detailed Bounded Contexts

7. **SS6: Analytics and Feedback Subsystem**
   The Analytics and Feedback Subsystem collects, processes, and visualizes data about system usage, query patterns, and user feedback. It provides insights for system improvement and helps measure the effectiveness of the platform.

   1. *Key Components:*
      - Data Collection Service: Gathers usage and performance metrics
      - Feedback Manager: Processes user ratings and feedback
      - Analytics Engine: Analyzes collected data for insights
      - Reporting Service: Generates dashboards and reports
   2. *Interfaces:*
      - To User Interface: Provides analytics dashboards and feedback forms
      - To Query Management Subsystem: Collects query data
      - To Expert Matching Subsystem: Provides feedback for algorithm improvement
      - To Communication Subsystem: Collects communication metrics

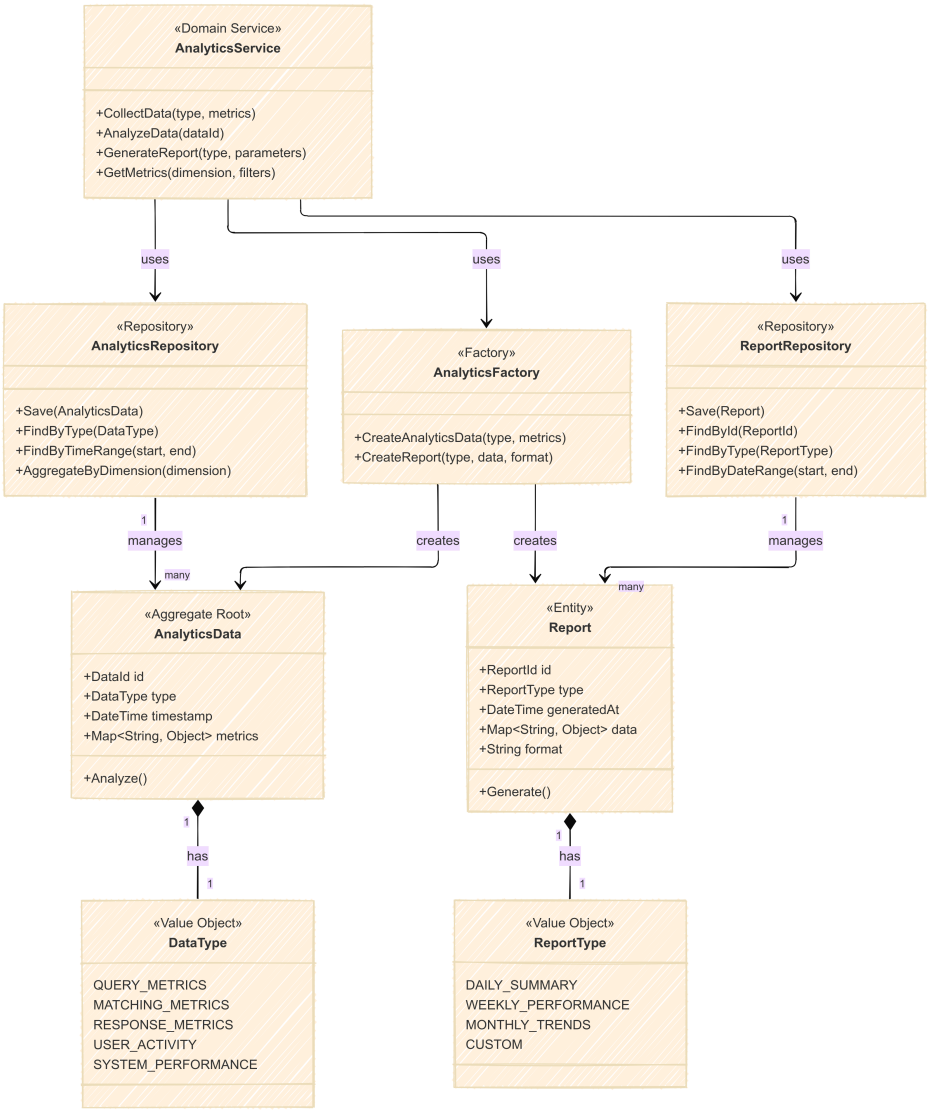

Figure 10: Analytics Subsystem : Detailed Bounded Contexts

8. **SS7: Integration Subsystem**
   The Integration Subsystem manages connections with external systems and services, including the browser extensions institutional databases, and third-party services. It provides APIs and adapters for seamless integration.

   1. *Key Components:*
      - API Gateway: Provides unified access to system functionality
      - Institutional Data Connector: Integrates with IIIT Hyderabad's academic databases
      - External Service Adapters: Connects to third-party services as needed
   2. *Interfaces:*
      - To Institutional Systems: Connects to academic databases and authentication systems
      - To All Subsystems: Provides external data and services as needed

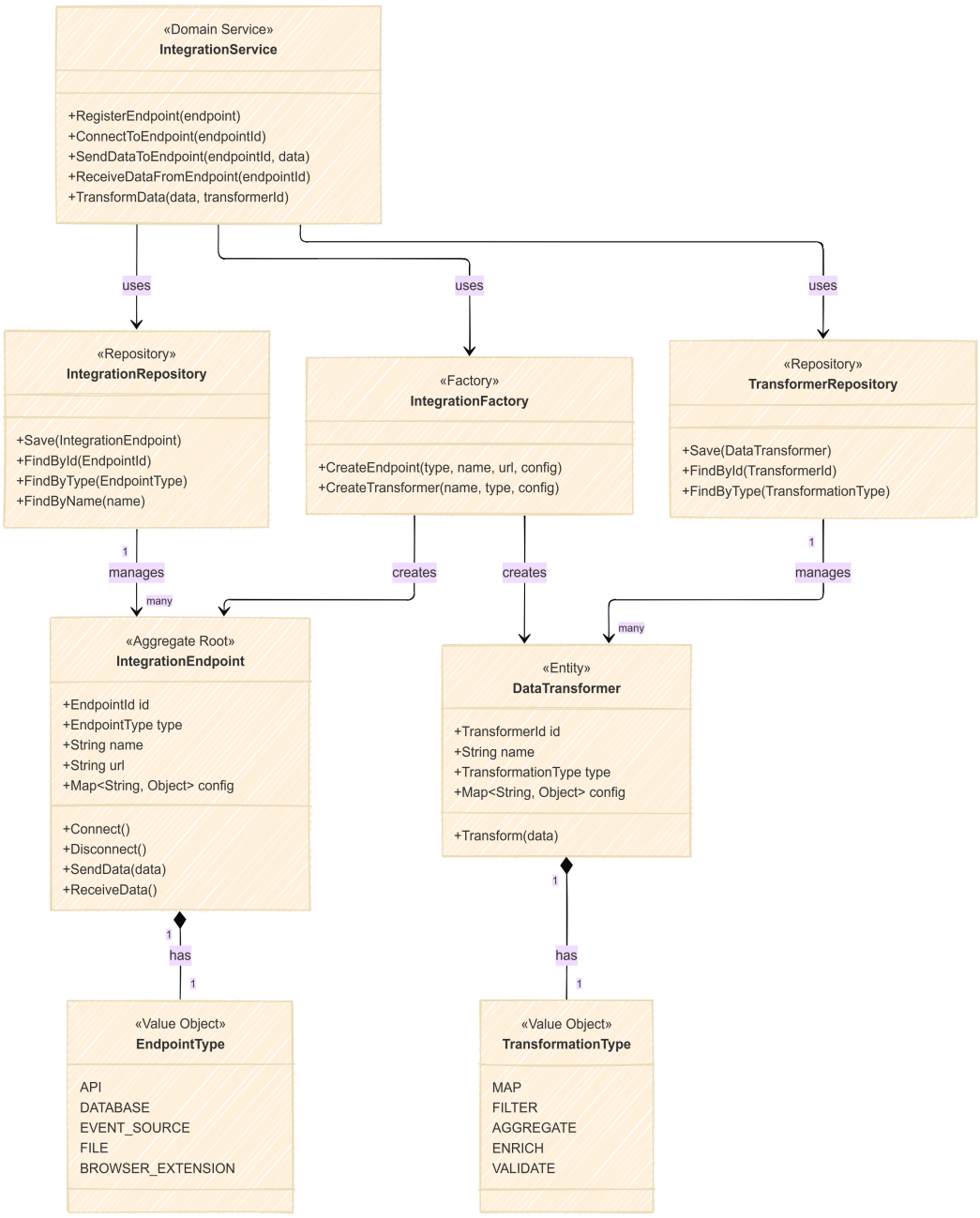

Figure 11: Integration Subsystem : Detailed Bounded Contexts

Task 2: Architecture Framework

This section outlines the architectural framework of MindChain, including stakeholder identification following IEEE 42010 and major design decisions documented as Architecture Decision Records (ADRs).

Stakeholder Identification

Following IEEE 42010 standard to identify stakeholders, their concerns, and the viewpoints and views addressing these concerns.

IEEE 42010 is a standard for architecture description that provides a framework for identifying stakeholders, their concerns, and the viewpoints and views that address these concerns. This framework helps ensure that the architecture meets the needs of all stakeholders and addresses their specific concerns.

Key Components of IEEE 42010:
Stakeholders Individuals, groups, or organizations with interests in or concerns about the system
Concerns Interests or requirements that stakeholders have regarding the system
Viewpoints Perspectives from which to view the system that address specific concerns
Views Representations of the system from specific viewpoints

Stakeholder Categories

The stakeholders of the MindChain system can be categorized into four main groups: Users, Operators, Developers, and Business. Each group has its own set of concerns and interests in the system.

Category	Stakeholder	Description	Primary Concerns
Users	Students	Undergraduate and graduate students at IIIT Hyderabad who seek academic and research support; some students can be recognized as experts	• Ease of submitting queries • Quick and accurate expert matching • Timely responses to questions • User-friendly interface • If expert, get matched and allow to answer • Privacy of academic information
	Faculty	Professors and instructors who provide expertise and answer student queries	• Relevant query matching based on expertise • Manageable workload and notification frequency • Efficient communication tools • Don't want to be overwhelmed with notifications
	Staff	Administrative and support staff who assist with event coordination and administrative queries	• Clear categorization of administrative queries • Integration with institutional systems • Efficient workflow management • Reporting and analytics

Category	Stakeholder	Description	Primary Concerns
Operators	System Administrators	IT staff responsible for maintaining and operating the MindChain platform	• System reliability and uptime • Performance monitoring • Security management • Backup and recovery • Ease of maintenance
	Department Coordinators	Staff who oversee the use of MindChain within specific departments	• Department-specific analytics • Faculty workload management • Query routing and assignment • Integration with departmental processes
	IT Support Staff	Personnel who provide technical support to users of the system	• Troubleshooting tools • User management capabilities • System diagnostics • Documentation and knowledge base
Developers	Software Developers	Engineers who build and maintain the MindChain platform	• Code maintainability and quality • Development efficiency • Technical documentation • Testing and debugging capabilities • Integration with development tools
	UX/UI Designers	Designers responsible for the user experience and interface	• Design system consistency • Accessibility compliance • User flow optimization • Responsive design implementation
	Data Scientists	Specialists who develop and improve the matching algorithm	• Data quality and availability • Algorithm performance metrics • Model training and evaluation • Integration with the matching subsystem
	QA Engineers	Quality assurance specialists who test the system	• Testability of components • Test automation support • Bug tracking and reporting • Performance testing capabilities
Business	Institute Leadership	Directors, deans, and other leadership at IIIT Hyderabad	• Alignment with institutional goals • Return on investment • Reputation and image • Compliance with policies • Strategic value

Category	Stakeholder	Description	Primary Concerns
	Project Sponsors	Individuals or groups funding the MindChain project	• Budget adherence • Timeline compliance • Feature delivery • Project success metrics
	Regulatory Compliance	Legal and compliance officers ensuring adherence to regulations	• Data privacy compliance • Security standards • Prevention and mitigation of cyber crimes • Accessibility requirements • Audit trails and reporting

Viewpoints

Viewpoint	Description	Addressed Concerns	Primary Stakeholders
Functional Viewpoint	Describes the system's functional elements, their responsibilities, interfaces, and interactions	• Feature completeness • System capabilities • Functional requirements • Integration points	Students, Faculty, Staff, Developers
Information Viewpoint	Describes how the system stores, manages, and manipulates data	• Data integrity • Data privacy • Information flow • Data persistence	Data Scientists, System Administrators, Regulatory Compliance
Deployment Viewpoint	Describes the environment in which the system will be deployed and the dependencies on its environment	• System reliability • Scalability • Performance • Infrastructure requirements	System Administrators, IT Support Staff, Developers
Development Viewpoint	Describes the architecture that supports the software development process	• Code maintainability • Development efficiency • Testing strategy • Module organization	Software Developers, QA Engineers
Usability Viewpoint	Describes the user interface, user experience, and accessibility aspects of the system	• User-friendly interface • Accessibility • User flow optimization • Responsive design	Students, Faculty, Staff, UX/UI Designers
Security Viewpoint	Describes how the system protects sensitive data and prevents unauthorized access	• Data protection • Authentication • Authorization • Audit trails	System Administrators, Regulatory Compliance, Institute Leadership
Performance Viewpoint	Describes the performance characteristics of the system and how they affect user experience	• Response time • Throughput • Resource utilization • Scalability	Students, Faculty, System Administrators, Developers

Key Architecture Views

For each viewpoint need to mention: Model Kinds,Conventions, Correspondence Rules are not mentioned in this document!

Logical View

The logical view represents the functional elements of the system and their relationships. It addresses concerns related to system functionality, organization, and structure.

Key Diagrams:

- Component diagram showing the main subsystems[2]
- Class diagrams for key domain models[3]
- Sequence diagrams for critical interactions[4]

Process View

The process view addresses concerns related to concurrency, distribution, performance, and scalability. It shows how the runtime elements of the system interact.

Key Diagrams:

- Activity diagrams for key workflows

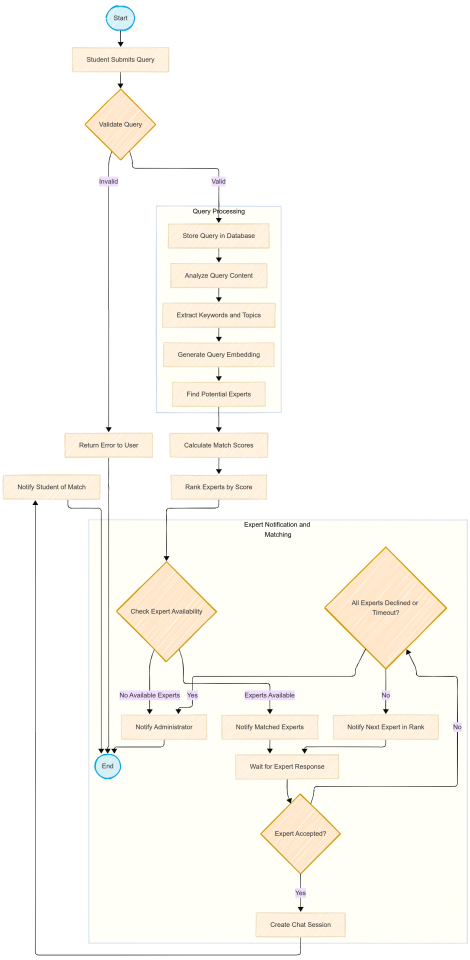

Figure 12: Activity Diagram for Expert matching

[2]Mentioned in previous sections
[3]Mentioned in previous sections
[4]Mentioned in previous sections

- State diagrams for complex processes

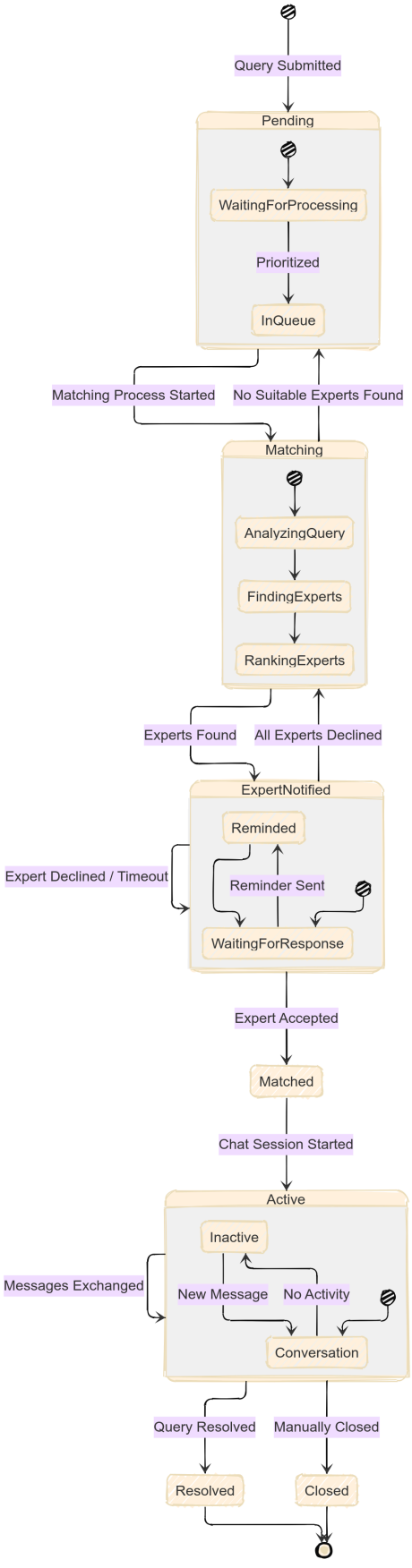

Figure 13: Activity Diagram for Expert matching

- Communication diagrams for real-time interactions

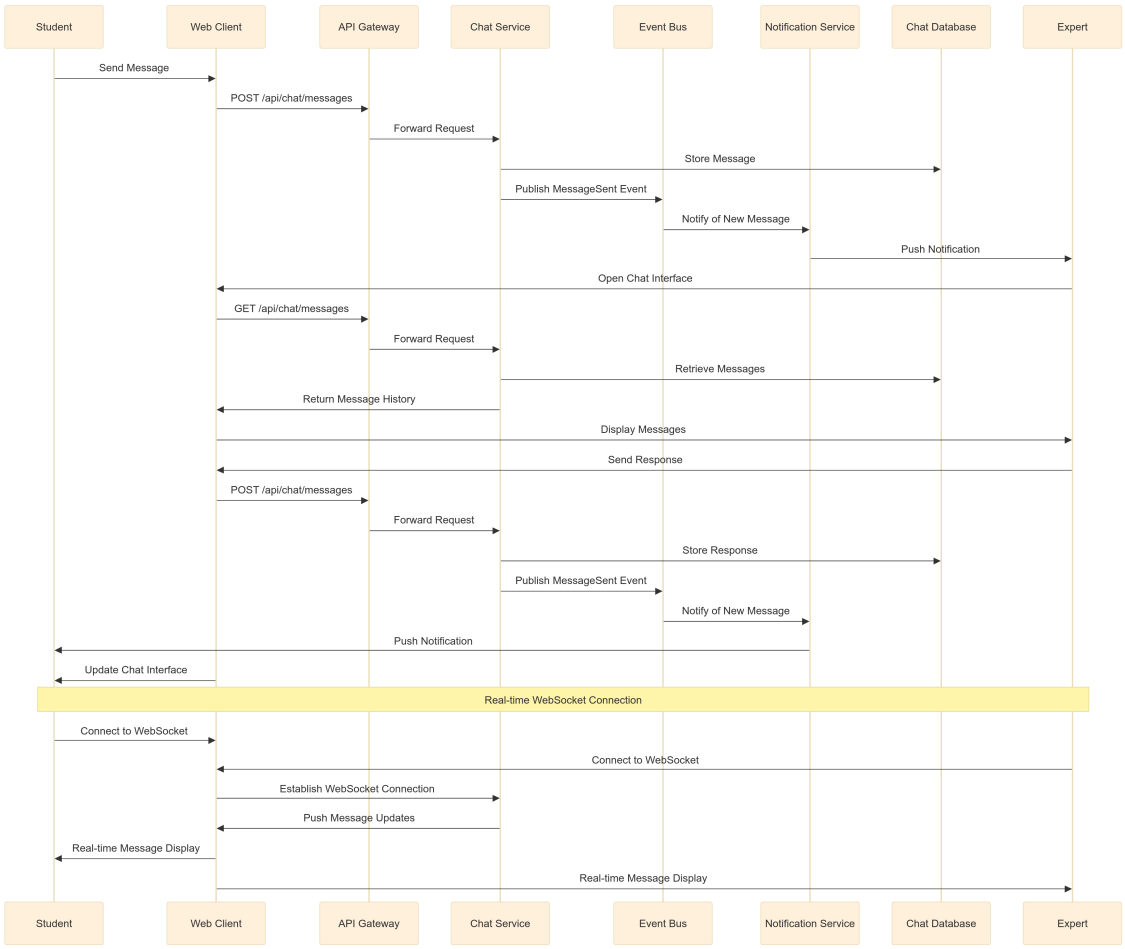

Figure 14: Communication Diagram for Expert matching

Development View

The development view addresses concerns related to software management, reuse, constraints, and development environment. It shows the organization of modules and components.

Key Diagrams:

- Package diagrams showing code organization
- Module dependency diagrams
- Build and deployment pipeline diagrams

Physical View

The physical view addresses concerns related to system topology, distribution, and deployment. It shows how software elements are mapped to hardware.

Key Diagrams:

- Deployment diagrams showing hardware configuration

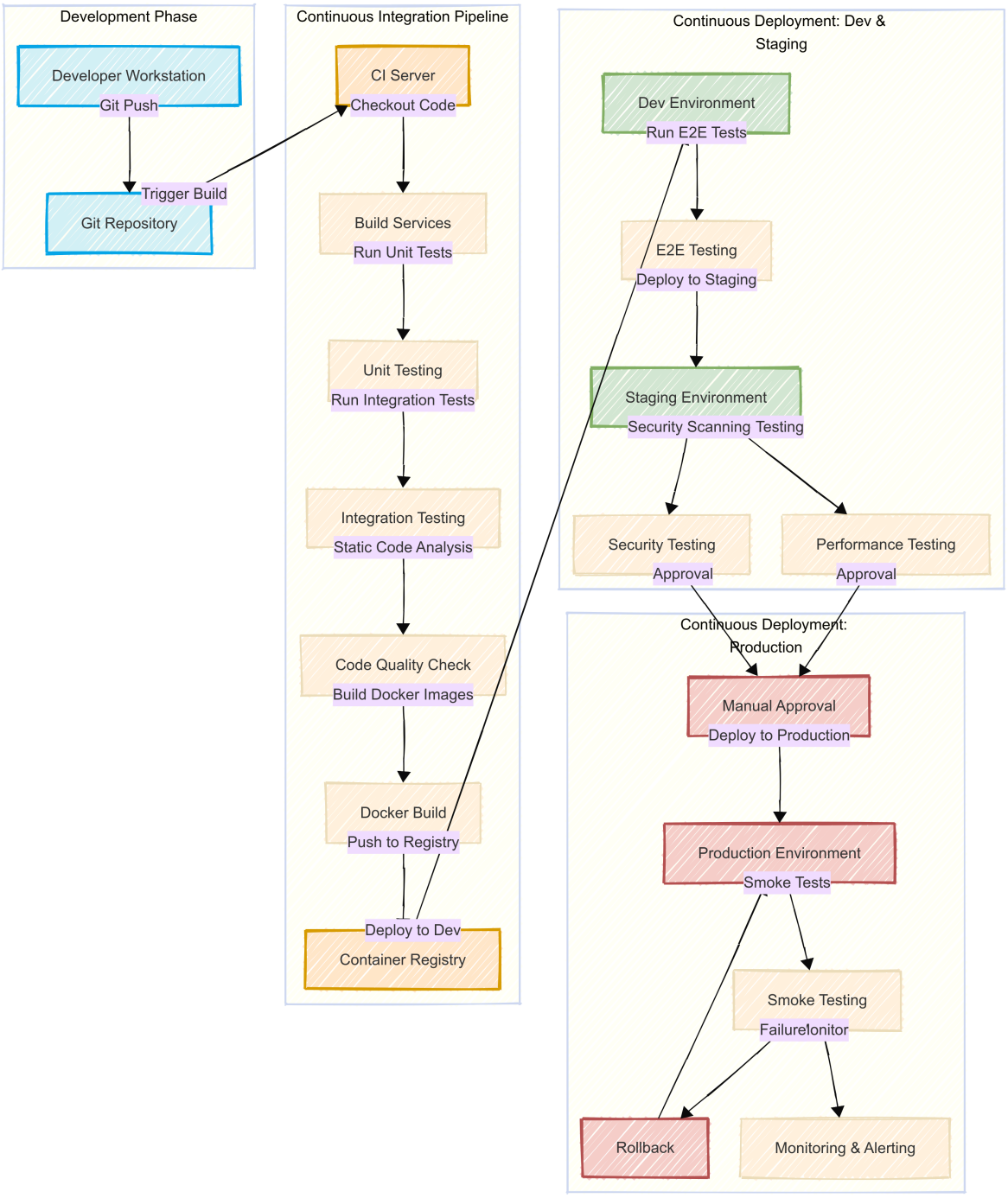

Figure 15: Deployment Diagram for MindChain

- Network topology diagrams
- Infrastructure architecture diagrams

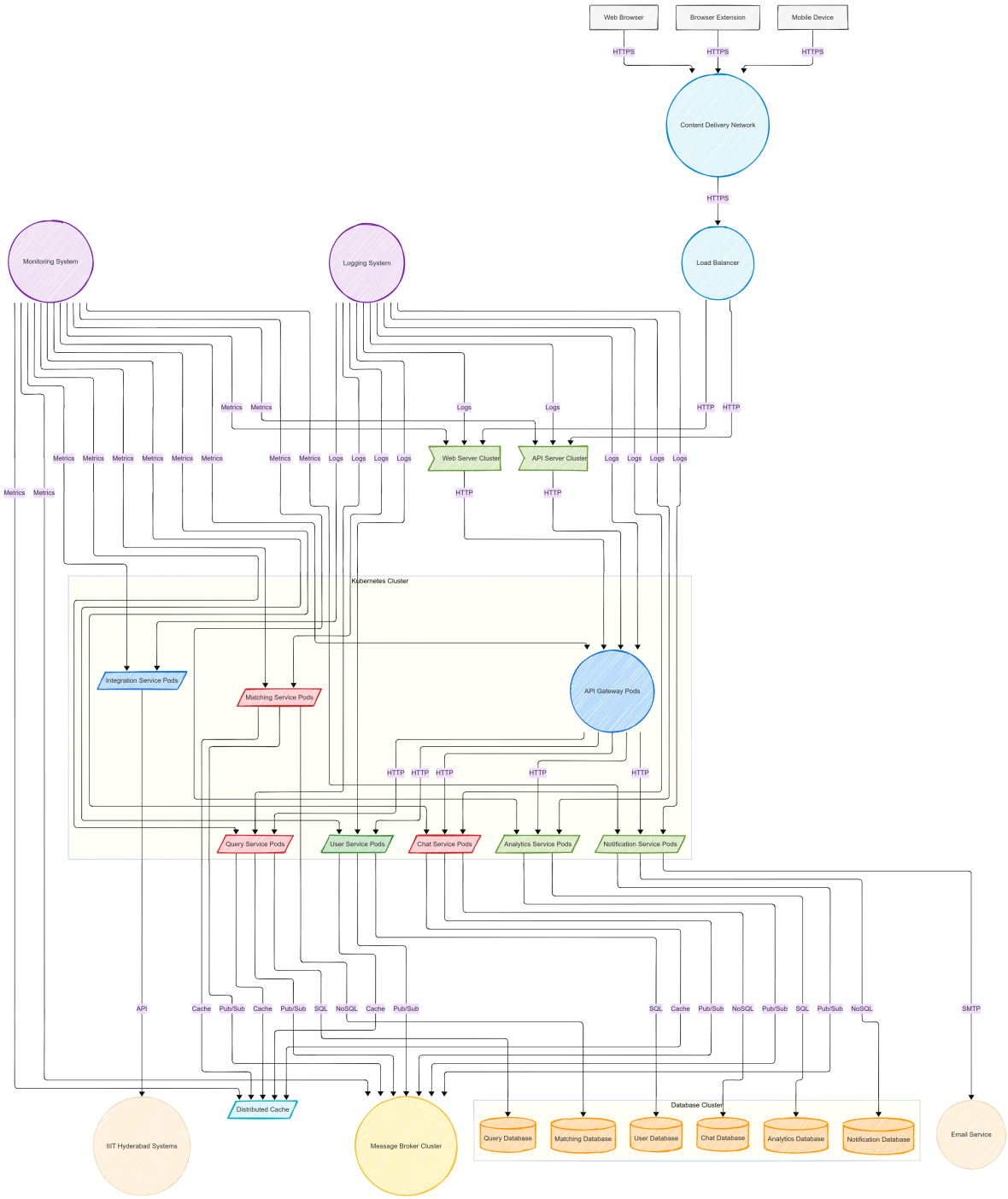

Figure 16: Different components involved in deployment

Scenarios

Scenarios illustrate how the various architectural elements work together to fulfill key requirements. They serve as a validation mechanism for the architecture.

Key Diagrams:

- Use case diagrams for main system functions

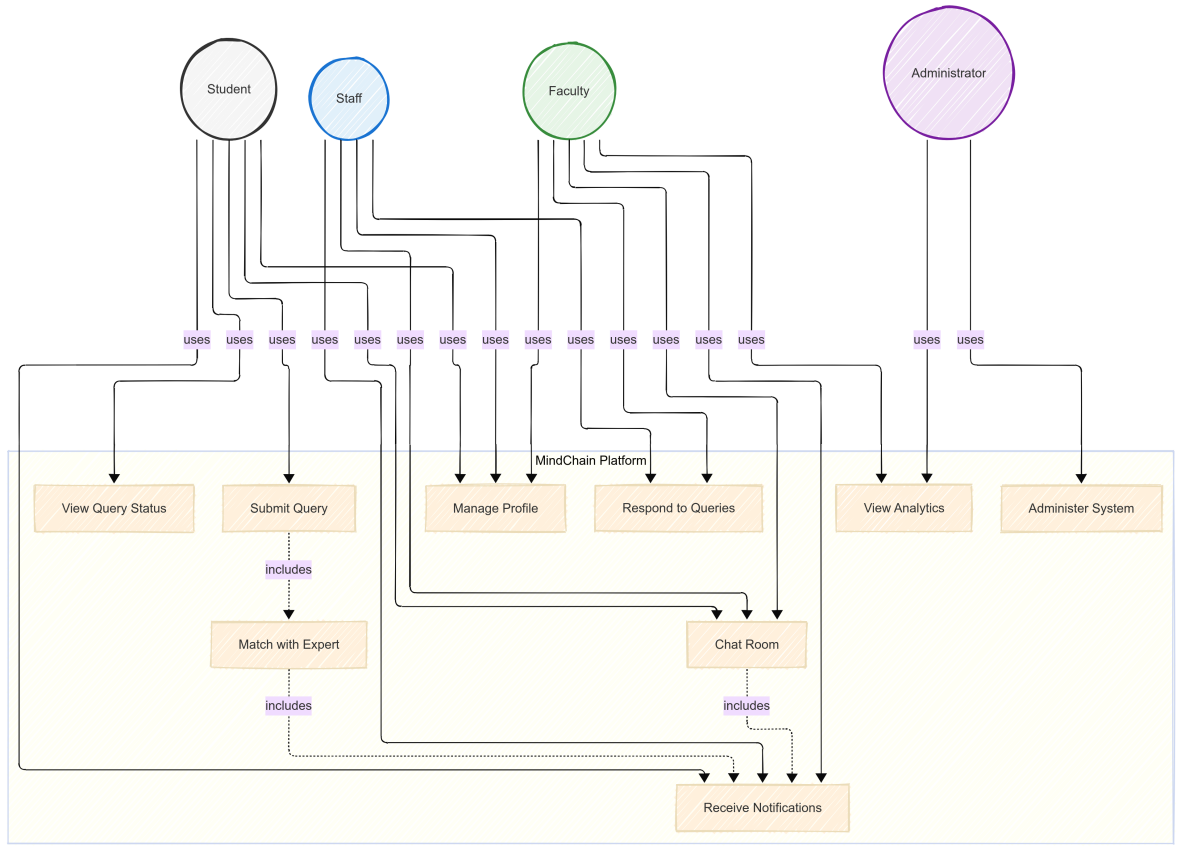

Figure 17: Use Case Diagram for MindChain

- User journey maps for key workflows

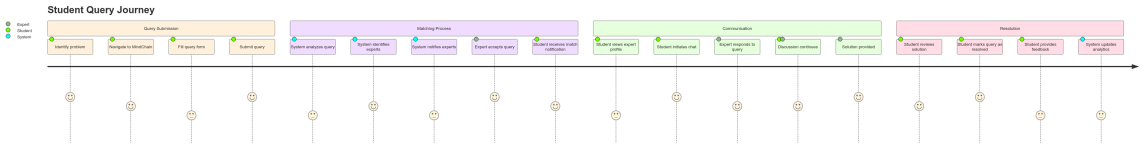

Figure 18: Use Case Diagram for MindChain

- Sequence diagrams for specific scenarios

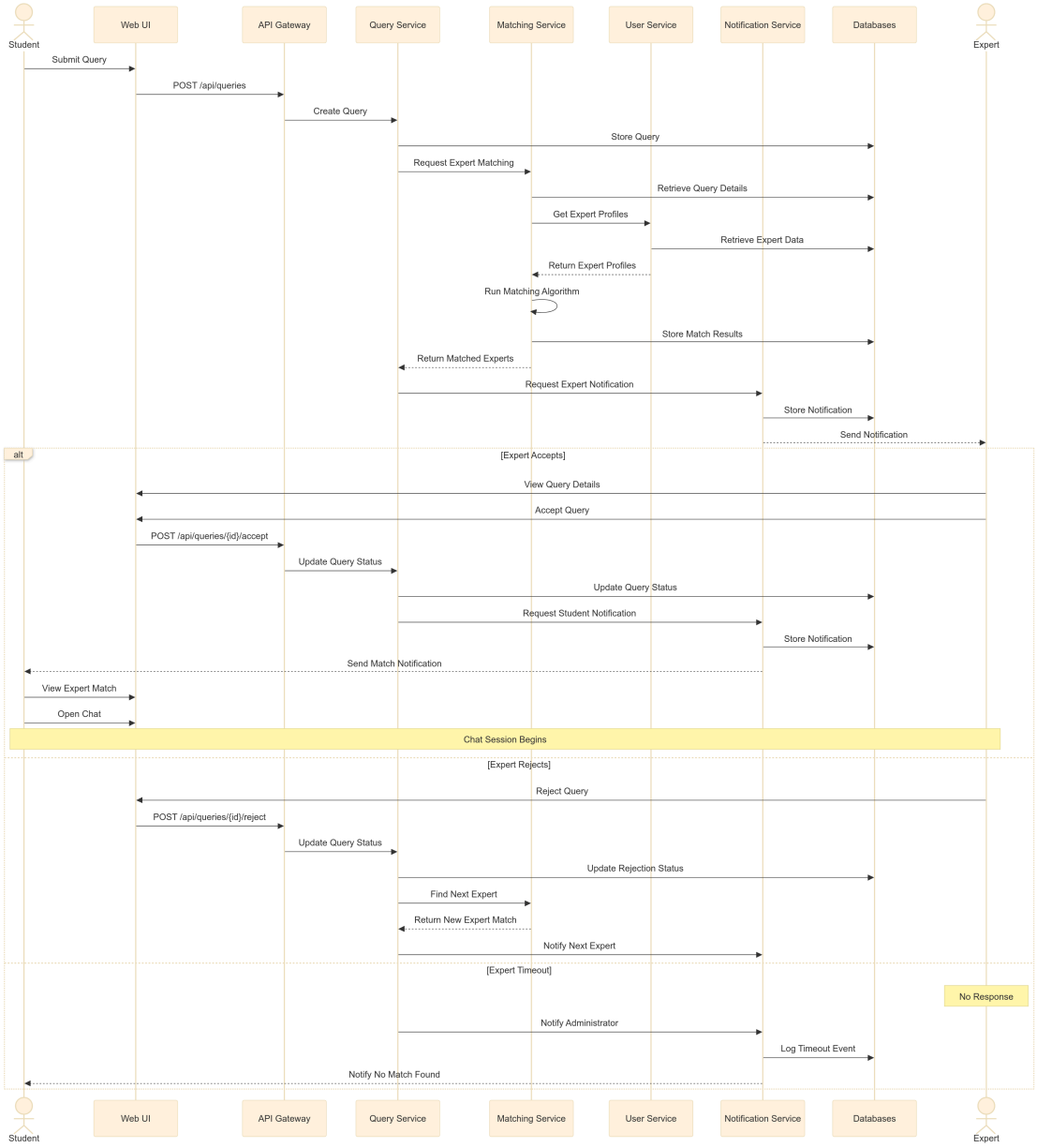

Figure 19: Sequence Diagram for MindChain

Major Design Decisions(* are accepted)

Architecture Decision Records (ADRs) documenting significant design decisions for the MindChain platform.

What are ADRs?

Architecture Decision Records (ADRs) are documents that capture important architectural decisions made along with their context and consequences. They provide a record of architectural decisions and the rationale behind them, serving as valuable documentation for current and future team members.

Key Components of an ADR:

Title A descriptive title for the decision
Status The current status of the decision (proposed, accepted, deprecated, etc.)
Context The background and situation that led to the need for a decision
Decision The specific architectural decision that was made
Consequences The resulting effects of the decision, both positive and negative
Alternatives Other options that were considered and why they were rejected

ADR 001: Microservices Architecture for MindChain*

```
Status: Accepted

Context
-------
MindChain requires a scalable, maintainable architecture that can support multiple
distinct functionalities (user management, query matching, real-time communication,
etc.) while allowing for independent development, deployment, and scaling of
different components. The system needs to handle varying loads across different
features and must be able to evolve over time as new requirements emerge.

We need to decide on the overall architectural style for the MindChain platform,
considering factors such as scalability, maintainability, development efficiency, and
deployment flexibility.

Decision
--------
We will adopt a microservices architecture for the MindChain platform, with the
following key characteristics:

- The system will be decomposed into seven core microservices aligned with the
identified subsystems: User Management, Query Management, Expert Matching,
Communication, Notification, Analytics, and Integration.
- Each microservice will have its own database, following the database-per-service
pattern to ensure loose coupling.
- Services will communicate primarily through asynchronous messaging using a message
broker (RabbitMQ) for event-driven interactions.
- RESTful APIs will be used for synchronous service-to-service communication when
necessary.
- An API Gateway will be implemented to provide a unified entry point for client
applications.
- Services will be containerized using Docker and orchestrated with Kubernetes for
deployment and scaling.

Consequences
------------
Positive:
- Independent scalability of services based on demand (e.g., the Communication
service can scale during peak usage times)
- Technology flexibility allowing different services to use the most appropriate tech
stack
- Enhanced fault isolation preventing failures in one service from cascading to
others
- Parallel development enabling multiple teams to work on different services
simultaneously
- Easier maintenance and updates with the ability to deploy changes to individual
services
Negative:
- Increased operational complexity in deployment, monitoring, and management
- Potential performance overhead due to network communication between services
- Challenges in maintaining data consistency across service boundaries
- More complex testing and debugging processes
- Higher initial development cost compared to a monolithic approach

Alternatives Considered
-----------------------
```

Monolithic Architecture:
  A single, unified codebase containing all functionality would be simpler to develop initially and would avoid the complexity of service communication. However, it was rejected due to concerns about scalability, long-term maintainability, and the ability to evolve different parts of the system independently.

Modular Monolith:
  A compromise approach with a single deployment unit but clear internal module boundaries was considered. While this would reduce operational complexity, it would still limit independent scaling and technology choices, which are important for the diverse requirements of MindChain's subsystems.

Serverless Architecture:
  A fully serverless approach using functions-as-a-service was evaluated for its scaling benefits and reduced operational overhead. However, it was deemed less suitable for the stateful, long-running processes required by the real-time communication and matching features of MindChain.

ADR 002: Real-time Communication Implementation*

```
Status: Accepted

Context
-------
MindChain requires robust real-time communication capabilities for its chat interface
and notification system. The platform needs to support persistent conversations
between users and experts, real-time status updates, and immediate delivery of
notifications. The solution must be scalable to handle thousands of concurrent users
while maintaining low latency.

We need to decide on the technology and approach for implementing real-time
communication features in MindChain, considering factors such as performance,
reliability, scalability, and developer experience.

Decision
--------
We will implement real-time communication using WebSockets as the primary protocol,
with Socket.IO as the implementation library, supported by a Redis-based message
broker for scaling. Specifically:

- Socket.IO will be used on both client and server sides to provide WebSocket
communication with automatic fallback to HTTP long-polling when WebSockets are not
available.
- A Redis pub/sub system will be implemented to enable horizontal scaling of the
WebSocket servers.
- Messages will be persisted in a MongoDB database for message history and offline
delivery.
- The system will implement room-based communication for chat sessions and user-
specific channels for notifications.
- A presence system will track online/offline status and typing indicators.
- Message delivery acknowledgments and read receipts will be implemented to ensure
reliability.

Consequences
------------
Positive:
- Low-latency, bidirectional communication enabling true real-time interactions
- Reliable message delivery with acknowledgment mechanisms
- Scalable architecture that can handle thousands of concurrent connections
- Graceful degradation to HTTP long-polling when WebSockets are not supported
- Comprehensive feature set including presence detection and typing indicators
Negative:
- Increased server resource usage compared to request-response patterns
- Additional complexity in deployment and scaling WebSocket servers
- Potential challenges with load balancers and proxies that aren't configured for
WebSockets
- Need for careful connection management to prevent resource leaks
- Increased complexity in handling offline/reconnection scenarios

Alternatives Considered
-----------------------
HTTP Polling:
  Regular HTTP requests to check for updates would be simpler to implement but would
create unnecessary server load and wouldn't provide true real-time capabilities. This
approach was rejected due to performance concerns and poor user experience.
```

Server-Sent Events (SSE):
  SSE would provide real-time updates from server to client but lacks the bidirectional communication needed for chat features. It was considered for notifications only but rejected in favor of a unified approach for all real-time features.

GraphQL Subscriptions:
  Using GraphQL subscriptions would integrate well with a GraphQL API but would add complexity if the rest of the API uses REST. This approach was considered but deemed less mature and potentially more complex than Socket.IO for our specific requirements.

MQTT:
  A lightweight publish-subscribe protocol designed for IoT would be efficient but would require additional client-side libraries and might be less suitable for web applications. It was rejected as being optimized for different use cases than ours.

ADR 003: Expert Matching Algorithm Approach*

```
Status: Accepted

Context
-------
The core value proposition of MindChain is its ability to match queries to the most
relevant experts based on expertise, research interests, and availability. The
matching algorithm needs to be accurate, efficient, and adaptable to different types
of queries and expertise domains. It must also consider factors such as expert
workload, availability, and past performance.

We need to decide on the approach and technology for implementing the expert matching
algorithm, considering factors such as accuracy, performance, explainability, and
adaptability.

Decision
--------
We will implement a hybrid matching algorithm that combines vector embeddings for
semantic matching with a rule-based scoring system for contextual factors.
Specifically:

- Query and expertise descriptions will be converted to vector embeddings using a
pre-trained language model (BERT or similar) to capture semantic meaning.
- Cosine similarity between query vectors and expert expertise vectors will form the
base matching score.
- Additional rule-based factors will be applied to adjust the base score, including:
  - Expert availability and current workload
  - Historical response rate and quality
  - Specific keyword matching for technical terms
  - Departmental and course affiliations
- The system will use a weighted scoring model that can be tuned based on feedback
and performance data.
- Match explanations will be generated to show why a particular expert was matched to
a query.
- The algorithm will be implemented as a separate microservice with its own scaling
capabilities.

Consequences
------------
Positive:
- High-quality semantic matching that understands the meaning behind queries
- Flexibility to incorporate multiple factors beyond text similarity
- Explainable results that can show why a match was made
- Ability to improve over time through feedback and performance data
- Scalable architecture that can handle complex matching operations
Negative:
- Higher computational requirements for generating and comparing vector embeddings
- Complexity in tuning the weights for different factors
- Need for significant training data to optimize the algorithm
- Potential cold-start issues for new experts with limited history
- Challenges in maintaining the balance between semantic and rule-based factors

Alternatives Considered
-----------------------
Keyword-Based Matching:
  A simpler approach using keyword extraction and matching would be less
```

computationally intensive but would miss semantic relationships and synonyms. This approach was rejected due to its limited understanding of natural language queries.

Pure Machine Learning Model:
  A fully ML-based approach using a trained classifier could potentially provide high accuracy but would require extensive training data and would be less explainable. This approach was rejected due to the initial lack of training data and the need for explainability.

Graph-Based Matching:
  Using a knowledge graph to represent expertise and queries would capture complex relationships but would be more complex to implement and maintain. This approach was considered promising for future iterations but too complex for the initial implementation.

Manual Assignment:
  Having department coordinators manually assign queries would ensure human judgment but wouldn't scale and would introduce delays. This approach was rejected as it wouldn't fulfill the real-time matching requirement.

ADR 004: Data Storage Strategy*

Status: Accepted

Context

MindChain needs to store and manage various types of data, including user profiles, queries, chat messages, notifications, and analytics. Different data types have different access patterns, consistency requirements, and scaling needs. The data storage strategy must support the microservices architecture while ensuring data integrity, performance, and scalability.

We need to decide on the data storage technologies and patterns for MindChain, considering factors such as data model complexity, query patterns, consistency requirements, and scaling needs.

Decision

We will adopt a polyglot persistence approach, using different database technologies for different services based on their specific data requirements. Specifically:

- User Management Service: PostgreSQL for structured user profile data with complex relationships and strong consistency requirements.
- Query Management Service: PostgreSQL for structured query data with transaction support and complex querying needs.
- Expert Matching Service: Elasticsearch for efficient vector similarity searches and full-text query capabilities.
- Communication Service: MongoDB for flexible schema chat messages with high write throughput and document-oriented structure.
- Notification Service: Redis for high-performance, in-memory notification storage with time-to-live support.
- Analytics Service: ClickHouse for columnar storage optimized for analytical queries and time-series data.
- Integration Service: Redis for caching and temporary storage of integration data.

Each service will own its database, with no direct database access from other services. Cross-service data access will be handled through service APIs or event-driven communication.

Consequences

Positive:
- Optimized data storage for different access patterns and requirements
- Independent scaling of databases based on service needs
- Improved performance through specialized database technologies
- Enhanced resilience with isolated database failures
- Flexibility to evolve data models independently for each service
Negative:
- Increased operational complexity in managing multiple database technologies
- Challenges in maintaining data consistency across different databases
- Need for specialized knowledge across different database systems
- More complex backup, monitoring, and disaster recovery processes
- Potential for data duplication across services

Alternatives Considered

Single Relational Database:

Using a single PostgreSQL database for all services would simplify operations and ensure strong consistency but would create tight coupling between services and limit scalability. This approach was rejected due to its conflict with microservices principles and scaling limitations.

Single NoSQL Database:
Using a single MongoDB or similar NoSQL database for all services would provide schema flexibility but would not be optimized for all access patterns and would still create coupling. This approach was rejected as it wouldn't provide the specialized capabilities needed for different data types.

Database-per-Service with Same Technology:
Using the same database technology (e.g., PostgreSQL) for all services but with separate instances would reduce operational complexity but wouldn't optimize for different data requirements. This approach was considered but rejected in favor of more specialized optimization.

Shared Database with Schema-per-Service:
Using a single database instance with separate schemas for each service would reduce infrastructure costs but would create potential coupling and scaling issues. This approach was rejected as it wouldn't provide true isolation between services.

ADR 005: Use of Blockchain for Data Integrity

```
Status: Rejected

Context
-------
MindChain handles sensitive data, including user profiles, queries, and analytics.
Ensuring data integrity and preventing unauthorized modifications are critical
requirements. Blockchain technology was considered as a potential solution due to its
immutability and distributed ledger capabilities.

The proposal involved using a private blockchain to store critical data, such as
query logs, audit trails, and analytics, to ensure tamper-proof records and enhance
trust in the system.

Decision
--------
We decided **not** to adopt blockchain technology for data integrity in MindChain for
the following reasons:

- The complexity and overhead of maintaining a private blockchain are not justified
for the current use case.
- Blockchain's immutability conflicts with the need for data updates and deletions
(e.g., GDPR compliance for user data).
- The performance overhead of blockchain transactions would negatively impact system
responsiveness.
- Existing database technologies (e.g., PostgreSQL with audit logging) can meet the
data integrity requirements more efficiently.

Consequences
------------
Positive:
- Avoids the operational complexity and resource overhead of blockchain.
- Simplifies compliance with data privacy regulations like GDPR.
- Reduces development and infrastructure costs.

Negative:
- Missed opportunity to leverage blockchain's tamper-proof capabilities.
- Potential perception of weaker data integrity compared to blockchain-based
solutions.

Alternatives Considered
-----------------------
1. **Audit Logging in Relational Databases**:
   Using PostgreSQL's built-in audit logging features to track changes to critical
data. This approach was chosen as it provides sufficient data integrity with minimal
overhead.

2. **Hash-Based Integrity Checks**:
   Storing cryptographic hashes of critical data in a separate database to detect
unauthorized modifications. This approach was considered but deemed unnecessary given
the adequacy of audit logging.

3. **Third-Party Integrity Services**:
   Using external services to verify data integrity. This approach was rejected due
to concerns about vendor lock-in and additional costs.
```

4. **Blockchain**:
   A private blockchain for storing critical data was rejected due to the reasons outlined above.

ADR 006: Expansion to Multi-Tenant Architecture

Status: In Review

Context

MindChain was initially designed to serve a single organization (IIIT Hyderabad). However, there is growing interest from other institutions and organizations to adopt the platform. To support this expansion, the system must evolve into a multi-tenant architecture that can serve multiple organizations while maintaining data isolation, scalability, and customizability.

The multi-tenant architecture must address the following requirements:
- Data isolation to ensure that each organization's data is securely separated.
- Scalability to handle varying loads from multiple tenants.
- Customizability to allow tenants to configure certain features according to their needs.
- Efficient resource utilization to minimize operational costs.

Decision

We propose transitioning MindChain to a **multi-tenant architecture** with the following key characteristics:

- **Tenant Isolation**: Each tenant's data will be logically separated using a database-per-tenant approach or schema-per-tenant approach, depending on the service's requirements.
- **Shared Infrastructure**: Core services (e.g., API Gateway, Authentication) will be shared across tenants to optimize resource usage.
- **Tenant Configuration**: A configuration service will be implemented to allow tenants to customize features, branding, and access control settings.
- **Tenant Identification**: All requests will include a tenant identifier (e.g., subdomain, header, or token) to route and process requests appropriately.
- **Scalability**: Kubernetes will be used to dynamically scale services based on tenant-specific loads.
- **Monitoring and Billing**: A centralized monitoring and billing system will track resource usage and generate tenant-specific reports.

Consequences

Positive:
- Enables MindChain to expand to multiple organizations, increasing its reach and impact.
- Ensures data isolation and security for each tenant.
- Provides flexibility for tenants to customize the platform to their needs.
- Optimizes resource utilization by sharing infrastructure where possible.
- Supports scalability to handle varying loads across tenants.

Negative:
- Increased complexity in managing tenant-specific configurations and data isolation.
- Higher operational overhead for monitoring, billing, and scaling multiple tenants.
- Potential performance impact if shared infrastructure is not properly scaled.
- Requires significant development effort to refactor existing services for multi-tenancy.

Alternatives Considered
---

1. **Single-Tenant Instances**:
   Deploying separate instances of MindChain for each organization would ensure complete isolation but would result in high operational costs and inefficiencies. This approach was rejected due to scalability concerns.

2. **Hybrid Multi-Tenancy**:
   Combining shared infrastructure for some services (e.g., Authentication) with isolated instances for others (e.g., Query Management) was considered. While this approach offers flexibility, it adds complexity and was deemed less efficient than a fully multi-tenant architecture.

3. **Federated Architecture**:
   Allowing each organization to host its own instance of MindChain while federating certain services (e.g., Analytics) was considered. This approach was rejected due to the operational burden on tenant organizations and the lack of centralized control.

Task 3: Tactics & Patterns

This section details the architectural tactics employed to address quality attributes and the design patterns implemented in the MindChain system.

Architectural Tactics

Describes tactics implemented to address extra-functional requirements, ensuring system quality.

Performance Tactics

Tactics employed to optimize response time, throughput, and resource utilization.

1. **Caching Strategy**

Focus Response Time Optimization

MindChain implements a multi-level caching strategy to reduce database load and improve response times for frequently accessed data.

Implementation Details

- In-memory caching for user profiles and expertise data
- Redis-based distributed caching for query matching results
- Browser-side caching for static assets and UI components
- Time-based cache invalidation strategies to maintain data freshness

Quality Attribute Addressed This tactic directly addresses **EFR1 (Response Time)** by ensuring that frequently accessed data is available without expensive database queries or computations, reducing average response time by up to **70%** for cached operations.

2. **Asynchronous Processing**

Focus Resource Management

MindChain employs asynchronous processing for computationally intensive operations to prevent blocking and improve overall system responsiveness.

Implementation Details

- Message queues for expert matching operations
- Background processing for notification batching and delivery
- Scheduled tasks for analytics and data aggregation
- Event-driven architecture for decoupling time-intensive operations

Quality Attribute Addressed This tactic addresses both **EFR1 (Response Time)** and **EFR2 (Scalability)** by ensuring that resource-intensive operations don't block the main request-response cycle, allowing the system to remain responsive even under heavy computational load.

Security Tactics

Tactics employed to optimize response time, throughput, and resource utilization.

1. **Defense in Depth**

Focus Multi-layered Protection

MindChain implements multiple layers of security controls to protect sensitive academic and personal data.

Implementation Details

1. Network-level protection with WAF and DDoS mitigation
2. Application-level security with input validation and output encoding
3. Database-level security with encryption and access controls

4. Regular security audits and penetration testing

Quality Attribute Addressed This tactic addresses **EFR5 (Data Privacy)** and **EFR6 (Authentication and Authorization)** by creating multiple security barriers that must be breached for an attacker to access sensitive information.

2. **Least Privilege**

Focus Access Control

MindChain enforces the principle of least privilege, ensuring users and system components have only the minimum access rights necessary.

Implementation Details
- Role-based access control for different user types (students, faculty, staff)
- Fine-grained permissions for system operations
- Contextual access controls based on query relevance and expertise
- Service-to-service authentication with limited scopes

Quality Attribute Addressed This tactic addresses **EFR6 (Authentication and Authorization)** by minimizing the potential damage from compromised accounts or services, limiting access to only what is necessary for each role or component.

Scalability Tactics
Tactics employed to handle growing workloads and user base efficiently.

1. **Horizontal Scaling**

Focus Capacity Management

MindChain is designed to scale horizontally by adding more instances of services rather than increasing the capacity of existing instances.

Implementation Details
- Stateless microservices that can be replicated across multiple instances
- Load balancing to distribute requests across service instances
- Auto-scaling based on CPU utilization and request volume
- Containerization for consistent deployment across environments

Quality Attribute Addressed This tactic addresses **EFR2 (Scalability)** by allowing the system to handle increased load by adding more resources in a linear fashion, without requiring architectural changes.

2. **Data Partitioning**

Focus Database Scalability

MindChain implements data partitioning strategies to distribute database load and improve query performance at scale.

Implementation Details
- Horizontal sharding of user data based on department or faculty
- Time-based partitioning for historical query and chat data
- Read replicas for analytics and reporting workloads
- Polyglot persistence with specialized databases for different data types

Quality Attribute Addressed This tactic addresses **EFR2 (Scalability)** and **EFR1 (Response Time)** by ensuring that database operations remain efficient even as data volumes grow, preventing database bottlenecks.

Maintainability Tactics
Tactics employed to facilitate system evolution, updates, and long-term maintenance.

1. **Service Isolation**

Focus Modularity

MindChain isolates functionality into independent services that can be developed, tested, and deployed separately.

Implementation Details
- Clear service boundaries with well-defined interfaces
- Domain-driven design principles for service organization
- Independent deployment pipelines for each service
- Versioned APIs to manage service evolution

Quality Attribute Addressed This tactic addresses **EFR9 (Extensibility)** and **EFR10 (Testability)** by allowing individual services to evolve independently, reducing the risk and complexity of system changes.

2. **Comprehensive Monitoring**

Focus Observability

MindChain implements extensive monitoring and observability features to facilitate troubleshooting and maintenance.

Implementation Details
- Distributed tracing across service boundaries
- Centralized logging with context-aware correlation
- Real-time metrics and dashboards for system health
- Automated alerting for anomaly detection

Quality Attribute Addressed This tactic addresses **EFR3 (Availability)** and **EFR10 (Testability)** by providing visibility into system behavior, enabling rapid identification and resolution of issues.

Tactics Summary
Overview of architectural tactics and their relationship to quality attributes.

Tactic	Category	Primary Quality Attributes	Implementation Approach
Caching Strategy	Performance	Response Time	Multi-level caching with time-based invalidation
Asynchronous Processing	Performance	Response Time, Scalability	Message queues and background processing
Defense in Depth	Security	Data Privacy, Authentication	Multi-layered security controls

Tactic	Category	Primary Quality Attributes	Implementation Approach
Least Privilege	Security	Authentication and Authorization	Role-based and contextual access controls
Horizontal Scaling	Scalability	Scalability	Stateless services with load balancing
Data Partitioning	Scalability	Scalability, Response Time	Sharding and specialized databases
Service Isolation	Maintainability	Extensibility, Testability	Microservices with clear boundaries
Comprehensive Monitoring	Maintainability	Availability, Testability	Distributed tracing and centralized logging

Implementation Patterns
Design patterns utilized in the MindChain platform to solve common architectural challenges.

Architectural Patterns
High-level patterns that define the overall structure of the MindChain system.

1. **Microservices Architecture**

Focus System Structure

MindChain is built using a microservices architecture, decomposing the application into small, independently deployable services organized around business capabilities.

Implementation Details
- Six core microservices aligned with subsystems
- API Gateway for client communication
- Service discovery for dynamic service location
- Independent data stores for each service

Benefits
- Independent development and deployment cycles
- Technology flexibility for different services
- Improved fault isolation
- Targeted scaling of high-demand services

2. **Event-Driven Architecture**

Focus Communication Pattern

MindChain uses an event-driven architecture for asynchronous communication between services, enabling loose coupling and improved responsiveness.

Implementation Details
- Event bus for publishing and subscribing to events
- Event sourcing for critical state changes
- Command-Query Responsibility Segregation (CQRS) for complex domains
- Idempotent event handlers for reliability

Benefits
- Decoupled services that can evolve independently
- Improved system resilience through asynchronous processing
- Better scalability for event producers and consumers
- Enhanced auditability through event history

Structural Patterns
Patterns that define how classes and objects are composed to form larger structures.

1. **Repository Pattern**

Focus Data Access

MindChain uses the Repository pattern to abstract the data layer and provide a collection-like interface for accessing domain objects.

Implementation Details
- Generic repository interfaces for common operations
- Specialized repositories for complex domain objects
- Separation of query and command repositories
- Unit of work pattern for transaction management

Benefits
- Decoupling of business logic from data access details
- Simplified testing through repository mocking
- Consistent data access patterns across the application
- Ability to switch underlying data stores with minimal impact

2. **Adapter Pattern**

Focus Integration

MindChain uses the Adapter pattern to integrate with external systems and services while maintaining a consistent internal interface.

Implementation Details
- External system adapters for institutional data sources
- API adapters for third-party services
- Legacy system integration adapters
- Protocol adapters for different communication methods

Benefits
- Isolation of external system dependencies
- Simplified testing through adapter mocking
- Ability to swap external systems with minimal impact
- Consistent error handling for external interactions

Behavioral Patterns
Patterns that define how objects interact and communicate with each other.

1. **Observer Pattern**

Focus Event Notification

MindChain uses the Observer pattern to implement the notification system, allowing objects to subscribe to events and receive updates.

Implementation Details
- Event publishers for system state changes
- Multiple subscriber types for different notification channels
- Filtering mechanisms for notification relevance
- Batching strategies for notification delivery

Benefits
- Loose coupling between event sources and consumers
- Support for multiple notification channels
- Dynamic subscription management
- Scalable notification delivery

2. **Strategy Pattern**

Focus Algorithm Selection

MindChain uses the Strategy pattern to implement the expert matching algorithm, allowing different matching strategies to be selected based on context.

Implementation Details
- Common interface for all matching strategies
- Specialized strategies for different query types
- Context-aware strategy selection
- Composite strategies for complex matching scenarios

Benefits
- Flexibility to change matching algorithms at runtime
- Encapsulation of algorithm-specific logic
- Simplified testing of individual strategies
- Easy addition of new matching algorithms

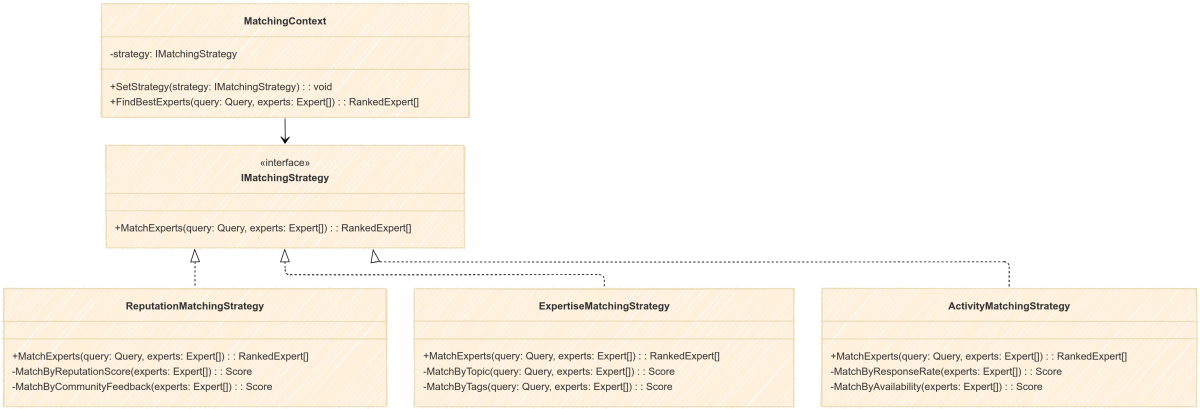

Figure 20: Strategy pattern for Matching algorithms

Integration Patterns
Patterns that define how different systems and services communicate and share data.

1. **API Gateway Pattern**

Focus Client Communication

MindChain uses the API Gateway pattern to provide a unified entry point for client applications, handling cross-cutting concerns and routing.

Implementation Details

- Centralized authentication and authorization
- Request routing to appropriate microservices
- Response aggregation from multiple services
- Rate limiting and traffic management

Benefits

- Simplified client integration
- Consistent security enforcement
- Reduced client-server communication
- Improved monitoring and analytics

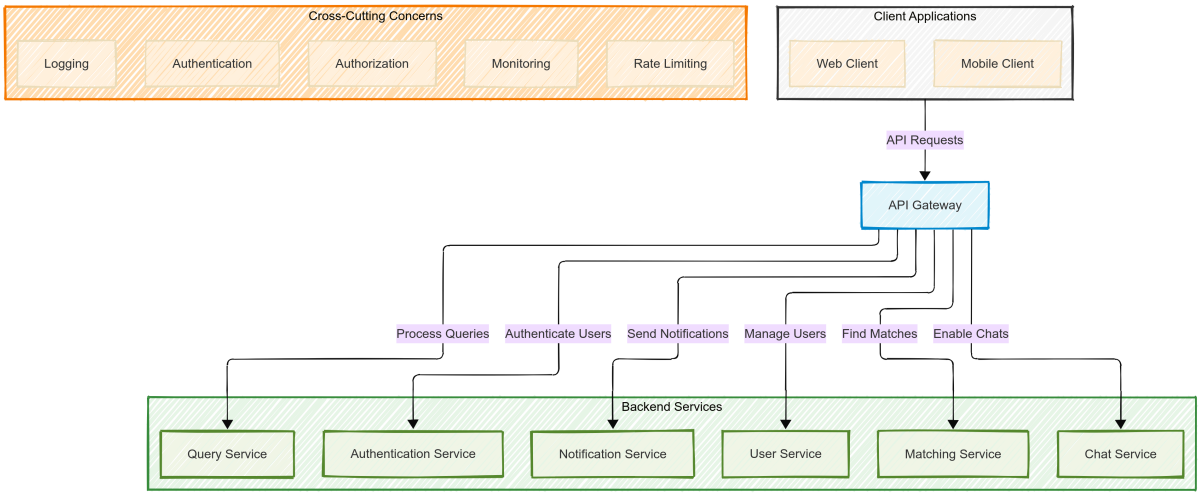

Figure 21: API gateway pattern

2. **Circuit Breaker Pattern**

Focus Fault Tolerance

MindChain uses the Circuit Breaker pattern to prevent cascading failures and improve system resilience when communicating with external services.

Implementation Details

- Failure threshold monitoring for external calls
- Automatic circuit opening on failure threshold breach
- Half-open state for testing recovery
- Fallback mechanisms for degraded functionality

Benefits

- Prevention of cascading failures
- Reduced latency from failing services
- Automatic recovery testing
- Graceful degradation under failure conditions

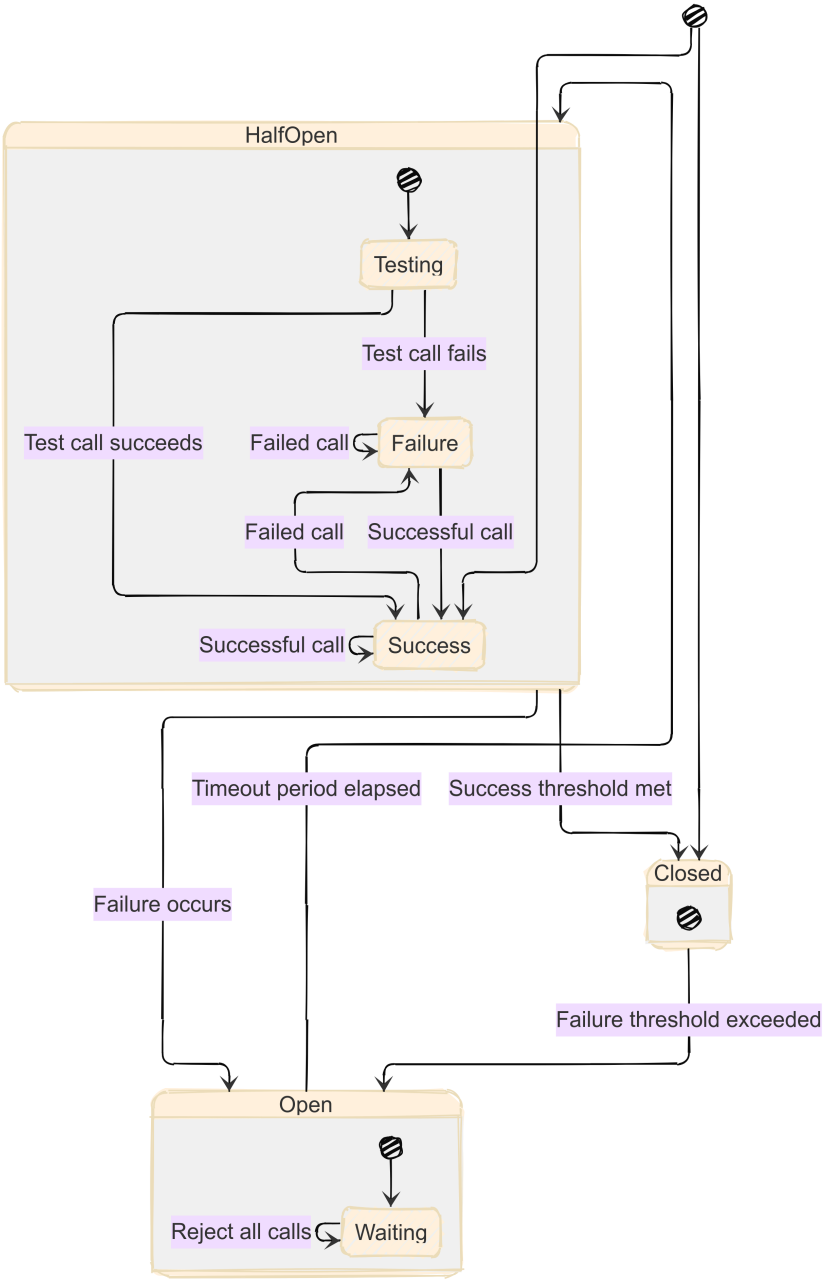

Figure 22: Circuit Breaker State Diagram

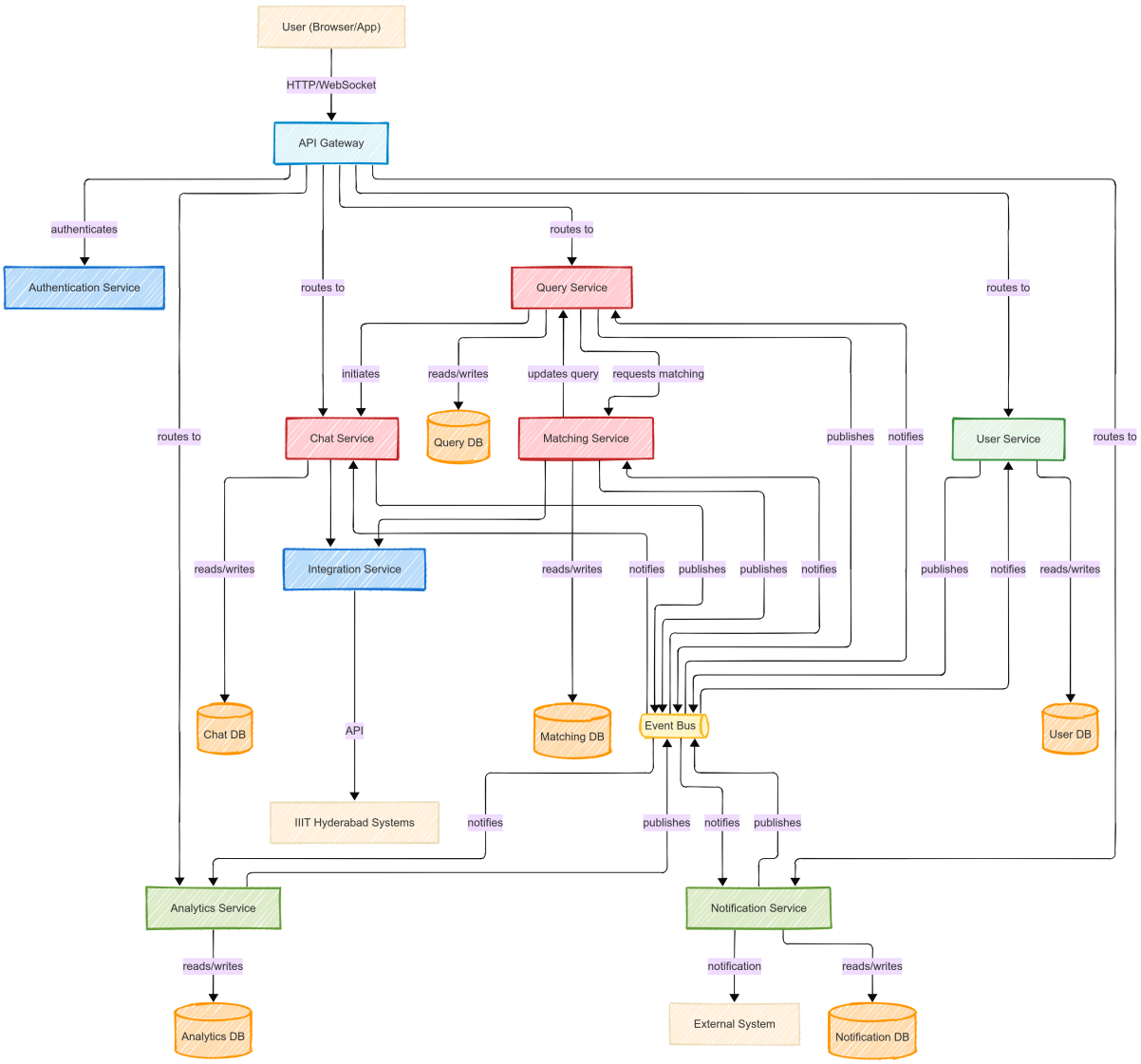

Figure 23: Final High Level Architecture Diagram for Mind Chain

Task 4: Implementation & Analysis

This section covers the development of MindChain prototype and provide a comparative analysis of the implemented architecture against alternatives

Prototype Development

Details the development of the MindChain prototype, showcasing key workflows and functionalities.

Core Functionality & Architectural Design

The MindChain prototype demonstrates the following core functionality, aligned with the architectural design principles:

Context-Aware Query Matching The prototype implements the vector embedding-based matching algorithm that analyzes query content and matches it with expert profiles based on academic expertise, research interests, and past interactions.

Real-Time Communication WebSocket-based real-time chat functionality enables immediate interaction between students and matched experts, with message persistence and history retrieval.

Incremental Notifications Smart notification batching and prioritization system that minimizes disruption while ensuring timely delivery of important updates.

Architectural Design Highlights

Microservices Architecture Decomposed into seven specialized services with clear boundaries and responsibilities.

Event-Driven Communication Asynchronous event-based communication between services for improved scalability and resilience.

Polyglot Persistence Specialized databases for different data types and access patterns (relational for structured data, document for profiles, graph for relationships).

API Gateway Unified entry point for client applications with authentication, routing, and request aggregation

Key Workflows

The prototype showcases the following key workflows that are essential to MindChain's operation:

Query Submission & Expert Matching Workflow

1. User submits a query through the web interface or app
2. Query is processed and enriched with contextual information
3. Matching algorithm analyzes query content and generates vector embeddings
4. System identifies and ranks potential experts based on expertise match
5. Top-ranked experts receive notifications about the new query
6. Experts can accept or decline the query
7. Upon acceptance, a communication channel is established

Real-Time Communication Workflow

1. User and expert join the established communication channel
2. WebSocket connection is established for real-time messaging
3. Messages are encrypted end-to-end for security
4. Messages are persisted to the database for history and continuity
5. Users receive real-time typing indicators and read receipts
6. File sharing and rich media support is available
7. Chat history is accessible for future reference

Notification Management Workflow

1. System events trigger notification creation
2. Notifications are prioritized based on urgency and user preferences
3. Similar notifications are batched to reduce interruptions
4. Notifications are delivered through multiple channels (in-app, email, push)
5. User interaction with notifications is tracked for future optimization
6. Notification preferences can be customized by users

Technologies & Tools

The MindChain prototype leverages the following technologies and tools:

Frontend Technologies

- React with Next.js for server-side rendering and routing
- TypeScript for type safety and developer experience
- Tailwind CSS for responsive and consistent UI design
- Socket.IO client for real-time communication
- React Query for efficient data fetching and caching

Backend Technologies

- Python with FastAPI for API services
- Socket.IO server for WebSocket communication
- PostgreSQL for structured data storage
- MongoDB for document storage (user profiles, queries)
- Redis for caching and pub/sub messaging
- ~~Neo4j for graph relationships (expertise network)~~

Android Native Application
- Kotlin for the application code
- Jetpack Compose as UI framework
- Dagger-Hilt for dependency injection

AI & Machine Learning
- TensorFlow for vector embeddings generation
- BERT-based models for semantic understanding
- Custom matching algorithms for expert recommendation
- Scikit-learn for classification and clustering

DevOps & Infrastructure
- Docker for containerization
- Kubernetes for orchestration and scaling
- ~~GitHub Actions for CI/CD pipelines~~
- ~~Prometheus and Grafana for monitoring~~
- ~~ELK Stack for centralized logging~~
- ~~Terraform for infrastructure as code~~

Development Tools
- VS Code with specialized extensions for collaborative development
- Postman for API testing and documentation
- ~~Jest and React Testing Library for automated testing~~
- ~~Storybook for component development and documentation~~
- ~~Figma for UI/UX design and prototyping~~

Architecture Analysis

This page provides a detailed analysis of MindChain's architecture, comparing it with alternative approaches and evaluating its performance against key extra-functional requirements.

Microservices vs. Monolithic Architecture

We compared our implemented microservices architecture against a monolithic alternative to evaluate the trade-offs and validate our architectural decisions.

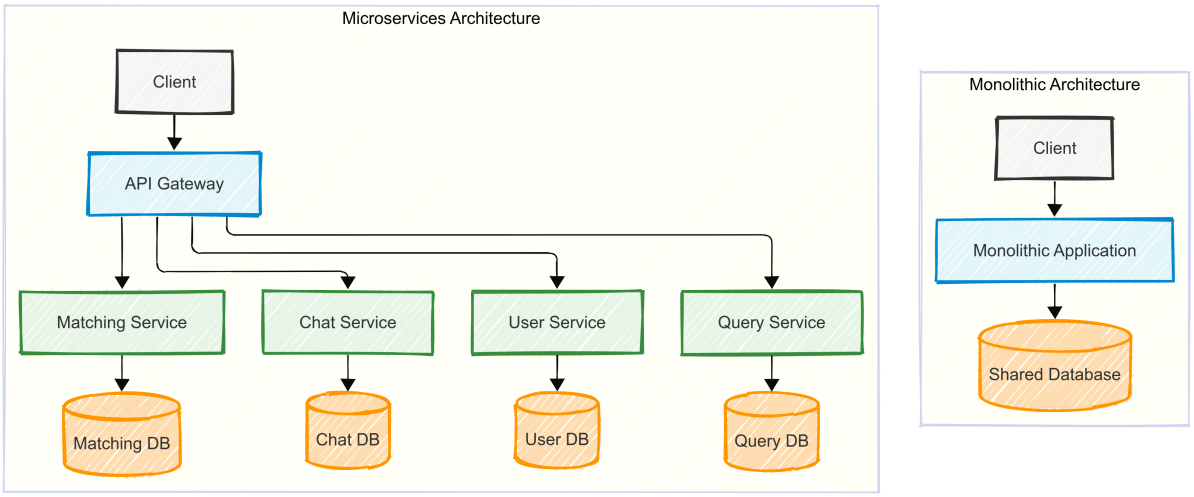

Figure 24: Different components involved in deployment

Aspect	**Microservices Architecture (Implemented; only essential)**	**Monolithic Architecture (Alternative)**
Development Complexity	Higher initial complexity due to distributed nature	Lower initial complexity, simpler setup
Scalability	Independent scaling of services based on demand	Entire application must scale together
Deployment	Independent deployment of services	Entire application deployed at once
Technology Flexibility	Different technologies can be used for different services	Single technology stack for the entire application
Fault Isolation	Failures contained within individual services	Failures can affect the entire application
Team Organization	Teams can work independently on different services	Teams must coordinate more closely
Operational Complexity	Higher due to distributed nature	Lower, simpler to monitor and manage
Resource Utilization	Higher overhead due to service replication	More efficient resource usage

🗒️ **Observation**

While the microservices architecture introduces additional complexity, it provides significant advantages in terms of scalability, fault isolation, and team autonomy that are critical for MindChain's long-term evolution.

Performance Analysis

We conducted a comprehensive quantitative evaluation of three architectural deployments—monolithic, microservices without autoscaling, and microservices with autoscaling—for MindChain's prototype. To capture end-to-end performance and reliability, we measured five Service Level Indicators (SLIs):

1. Latency (P50, P95, P99)
2. Throughput (requests per second)
3. Error Rate (percentage of failed requests)
4. Availability (service uptime under stress)
5. Resource Utilization (CPU & memory usage)

Performance SLIs Used dummy data and load generators to do stress testing.

Latency for query submission

Load Level	Monolith P50 / P95 / P99 (ms)	MS w/o AS P50 / P95 / P99 (ms)	MS w/ AS P50 / P95 / P99 (ms)
Light (≤100)	110 / 180 / 300	145 / 220 / 360	135 / 210 / 350
Medium (101–500)	240 / 400 / 650	260 / 430 / 700	255 / 420 / 680
Heavy (>500)	750 / 1,200 / 2,000	450 / 700 / 1,050	440 / 680 / 1,000

🗒️ **Observation**

Auto scaling reduces tail latency by 5–8% under heavy load, closing the performance gap with the monolith at P95/P99.

- Bottleneck analysis is not done at this stage, to identify if database is the cause of issue / running on single instance infra.

Throughput

Architecture	Max Throughput (req/s)
Monolith	800
MS w/o AS (fixed pods)	1,200
MS w/ AS (elastic)	1,800 (at max usage)

Error Rate: primarily timeouts / connection failures

Load Level	Monolith (%)	MS w/o AS (%)	MS w/ AS (%)
Light	0.05	0.08	0.06
Medium	0.5	0.7	0.6
Heavy	4.2	3.5	1.8

🗒️ **Observation**

Autoscaling reduces error rates under stress by redistributing load to new instances, halving the heavy-load error rate compared to fixed microservices.

Availability: Kind of hard to reproduce! getting different values but he trend is same

Architecture	Availability under ≥2,000 connections
Monolith	82%
MS w/o AS	92%
MS w/ AS	98%

> 🗒️ **Observation**
>
> Autoscaling enhances uptime by provisioning additional service instances before saturation.

Operational SLIs

CPU & Memory Utilization: auto scaling at 50% usage; adjusting this varies results

Metric	Monolith Avg.	MS w/o AS Avg.	MS w/ AS Avg.
CPU Utilization	80%	70%	65%
Memory Utilization	85%	75%	70%

> 🗒️ **Observation**
>
> Elastic scaling in the autoscaled microservices cluster maintains headroom, reducing resource peaks by 5–10%.

Saturation (Queue Length)

- Monolith: Queue depth spikes > 100 under heavy load.
- MS w/o AS: Spikes to ~ 60 when fixed service instances hit capacity.
- MS w/ AS: Peaks remain < 40 thanks to horizontal pod autoscaling.

Architectural Trade-offs

Based on our analysis, we identified the following key trade-offs between the microservices and monolithic architectures:

Performance Trade-offs

Network Overhead vs. Scalability The microservices architecture introduces network communication overhead, slightly reducing performance for simple operations. However, it enables independent scaling of services, resulting in better performance under high load and for complex operations.

Specialized Optimization vs. Global Optimization Microservices allow for specialized optimization of individual services (e.g., the matching algorithm service), while the monolith enables more holistic optimizations across the entire codebase.

Maintainability Trade-offs

Development Complexity vs. Long-term Maintainability The microservices architecture has higher initial development complexity but offers better long-term maintainability through clearer service boundaries and independent evolution.

Technology Diversity vs. Consistency Microservices enable using different technologies for different services, allowing optimal tool selection but increasing the knowledge requirements for the development team.

Deployment Complexity vs. Deployment Flexibility Microservices require more sophisticated deployment infrastructure but enable independent deployment of services, reducing the risk and impact of deployments.

Cost Trade-offs

Infrastructure Costs vs. Development Costs The microservices architecture has higher baseline infrastructure costs but can reduce development costs through team autonomy and parallel development.

Operational Complexity vs. Operational Flexibility Microservices require more sophisticated monitoring and management tools, increasing operational costs, but provide better operational flexibility and fault isolation.

Initial Investment vs. Long-term Scalability Microservices require higher initial investment in infrastructure and tooling but offer better long-term scalability and cost efficiency as the system grows and evolves.

Decision Rationale

After careful consideration of these trade-offs, we chose the microservices architecture for MindChain based on the following key factors:

Anticipated Growth MindChain is expected to grow significantly in terms of user base and feature set, making the scalability advantages of microservices critical.

Team Structure The development team is organized into specialized groups that align well with the service boundaries, enabling more efficient parallel development.

Specialized Optimization Different components of MindChain have vastly different performance characteristics and requirements, making specialized optimization valuable.

Fault Isolation The critical nature of the platform for academic support makes fault isolation particularly important to maintain overall system availability.

Technology Evolution The rapidly evolving nature of AI and matching algorithms makes the ability to update individual services independently particularly valuable.

> 🗒️ **Conclusion**
>
> While the microservices architecture introduces additional complexity, our quantitative analysis confirms that it provides significant advantages in terms of scalability, performance under load, and long-term maintainability that align with MindChain's requirements and growth trajectory.
>
> - *Next Steps*: Integrate real traffic simulations, refine HPA thresholds.

Reflections

Challenges Faced

1. Team Coordination: Misaligned communication led to delays and inconsistencies, highlighting the need for better collaboration tools and clearer documentation.
2. Matching Algorithm: The current implementation is basic, suitable for the MVP, but requires optimization for scalability and quality with more data.
3. UI/UX Balancing: Balancing aesthetics and usability required iterative user feedback, which delayed finalization.

Lessons Learned

1. Document Design Decisions: ADRs clarified decisions and reduced confusion, proving essential for long-term maintainability.
2. Prioritize Deliverability: Focused on timelines and team expertise over ideal solutions, ensuring practical progress.
3. Iterative Feedback: Regular stakeholder feedback helped refine implementation and user experience.

4. Scalability Mindset: Even MVP designs must consider future growth to avoid technical debt.

Key Takeaways

1. Data quality is crucial for algorithm development and UI precision.
2. Automated testing and robust monitoring reduce post-deployment issues.
3. Comprehensive onboarding and documentation enhance team efficiency.
4. User-centric, iterative design ensures relevance and usability.

These insights emphasize pragmatic planning, communication, and adaptability for future projects.

References

[1] J. Martinez-Gil, A. L. Paoletti, G. Rácz, A. Sali, and K.-D. Schewe, "Accurate and efficient profile matching in knowledge bases," *Data & Knowledge Engineering*, vol. 117, pp. 195–215, 2018, doi: https://doi.org/10.1016/j.datak.2018.07.010.

[2] N. Nikzad–Khasmakhi, M. Balafar, and M. Reza Feizi–Derakhshi, "The state-of-the-art in expert recommendation systems," *Engineering Applications of Artificial Intelligence*, vol. 82, pp. 126–147, 2019, doi: https://doi.org/10.1016/j.engappai.2019.03.020.

[3] D.-R. Liu, Y.-H. Chen, W.-C. Kao, and H.-W. Wang, "Integrating expert profile, reputation and link analysis for expert finding in question-answering websites," *Information Processing & Management*, vol. 49, no. 1, pp. 312–329, 2013, doi: https://doi.org/10.1016/j.ipm.2012.07.002.

[4] M. Shahriari, S. Parekodi, and R. Klamma, "Community-aware Ranking Algorithms for Expert Identification in Question-Answer Forums," 2015, p. . doi: 10.1145/2809563.2809592.

[5] D. Poltorak, "Choose your own architecture \vert ITNEXT", *Medium*, Apr. 2025, Accessed: Apr. 17, 2025. [Online]. Available: https://itnext.io/choose-your-own-architecture-92c56b12f7b0