Faculty of Computers and Information
Kafr El-Sheikh University

OLLAMANET

By

Ibrahim Ahmed Ibrahim

Ahmed Abdulrahim Hamed

Esraa Ali Sultan

Asmaa Ahmed Ayad

Khaled Mohamed Abdul-Hafez

Khoulod Khaled Bakr

Supervisors

Dr. Reda Hussein Mabrouk

Undergraduate Project

Submitted to the textbfFaculty of Computers and Information Kafr El-Sheikh University as a partial fulfillment for BSc.

Information Systems Department
June 2025

Acknowledgments

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Abstract text goes here...

If you've just opened up this template, you should check out Chapter 1 for a quick introduction.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: your, keywords, go, here

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

OllamaNet is a comprehensive AI platform built on a modern microservices architecture that enables its users to explore, interact with, and manage various AI models through a coherent ecosystem of services. The platform provides both administrative capabilities and end-user experiences for AI-powered conversations and model discovery, all supported by a robust database layer.

The platform is designed to address the growing need for accessible, well-organized AI model interactions while maintaining security, scalability, and performance through specialized microservices that handle distinct aspects of the system's functionality.

A key purpose of OllamaNet is to provide developers with a customizable and scalable infrastructure for working with Large Language Models (LLMs), offering an alternative to existing Python-based libraries that may lack performance and customization options. By leveraging C# and .NET, the platform delivers superior robustness, maintainability, and extensibility, opening pathways for custom extensions and enterprise-grade implementations.

Terminology

OllamaNet: Comprehensive AI platform with microservices architecture for LLM interaction

Microservice: Independent, specialized service with specific domain responsibilities

LLM: Large Language Model, an AI model used for text generation and understanding

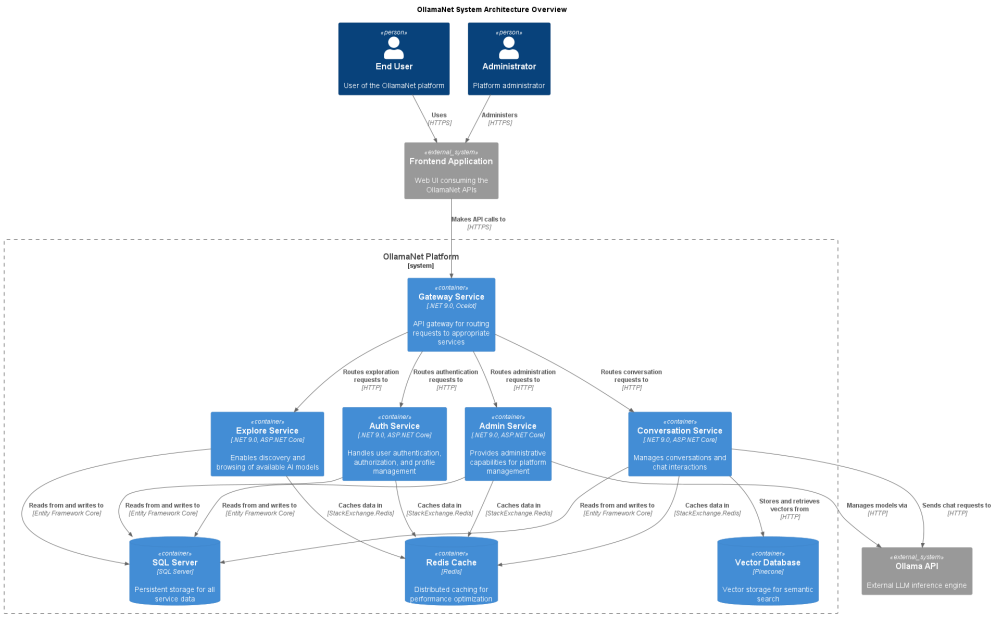

Figure 1.1 – OllamaNet Architecture Overview

1.2 Problem Statement

Traditional AI model deployment and interaction platforms often face several key challenges:

1. **Complexity in Administration**: Managing AI models, users, and permissions typically requires complex administrative interfaces.

2. **Context Loss in Conversations**: Users often lose conversation history and context when interacting with AI models.

3. **Inefficient Discovery**: Finding the right AI model for specific needs can be difficult without proper categorization and search capabilities.

4. **Security Concerns**: Maintaining proper authentication and authorization across AI services presents security challenges.

5. **Performance Bottlenecks**: AI interactions can suffer from latency issues, especially without proper caching and optimization strategies.

6. **Data Management Complexity**: Handling the persistence of conversations, user data, and model information requires sophisticated data access patterns.

7. **Reliability Constraints**: Many existing solutions use non-relational databases for speed of development, sacrificing data reliability and relationship integrity in the process.

8. **Limited Customization**: Popular Python-based LLM libraries prioritize simplicity over extensibility, limiting developers' ability to customize the infrastructure.

OllamaNet addresses these challenges through its specialized microservice architecture, providing a robust, extensible, and maintainable solution for AI model interaction.

Terminology

API Gateway: Component that routes requests to appropriate microservices

Context Preservation: Maintaining conversation history and state across interactions

Caching: Storing frequently accessed data to improve performance

1.3 Objectives and Goals

The OllamaNet platform aims to achieve the following objectives:

1. **Provide Comprehensive Administration**: Deliver complete control over platform resources through the AdminService.

2. **Enable Secure Authentication**: Implement robust user authentication and authorization via the AuthService.

3. **Facilitate Model Discovery**: Allow users to browse and evaluate AI models through the ExploreService.

4. **Support Rich Conversations**: Enable persistent, organized conversations with AI models via the ConversationService.

5. **Ensure Data Integrity**: Maintain consistent data operations through the DB Layer.

6. **Optimize Performance**: Implement caching strategies and efficient data access patterns across services.

7. **Enhance User Experience**: Deliver a responsive, intuitive interface for all platform interactions.

8. **Enable Custom Inference**: Provide flexible AI model inference capabilities through the InferenceService.

These objectives are achieved through a carefully designed microservices architecture that emphasizes separation of concerns, domain-driven design, and robust integration patterns.

Terminology

Domain-Driven Design: Software development approach that focuses on the core domain and domain logic

Separation of Concerns: Design principle for separating software into distinct sections

Integration Pattern: Standardized approach for connecting different components or services

1.4 Project Scope

OllamaNet encompasses a full-stack solution with the following components:

1.4.1 Microservices Backend

- **AdminService**: Central control point for platform administration, managing users, AI models, tags, and inference operations.

- **AuthService**: Comprehensive authentication and authorization service handling user registration, login, password management, and role-based access control.

- **ExploreService**: Model discovery and browsing service allowing users to search, filter, and explore available AI models and their capabilities.

- **ConversationService**: Conversation management service enabling persistent, organized interactions with AI models, including real-time streaming responses.

- **InferenceService**: Flexible inference engine service for interacting with Ollama models, exposed via ngrok for accessibility.
- **Gateway**: API gateway service routing client requests to appropriate microservices, handling authentication and authorization.
- **DB Layer**: Shared data access infrastructure implementing the repository and unit of work patterns for consistent data operations.

1.4.2 Frontend Applications

- Administrative interfaces for platform management
- End-user interfaces for conversation and model exploration

1.4.3 Infrastructure Components

- SQL Server database for persistence
- Redis for distributed caching
- JWT-based authentication system
- Integration with the Ollama inference engine
- RabbitMQ for service discovery

Terminology

Repository Pattern: Design pattern that mediates between the domain and data mapping layers

Unit of Work: Pattern that maintains a list of objects affected by a business transaction

JWT: JSON Web Token, a secure method for representing claims between parties

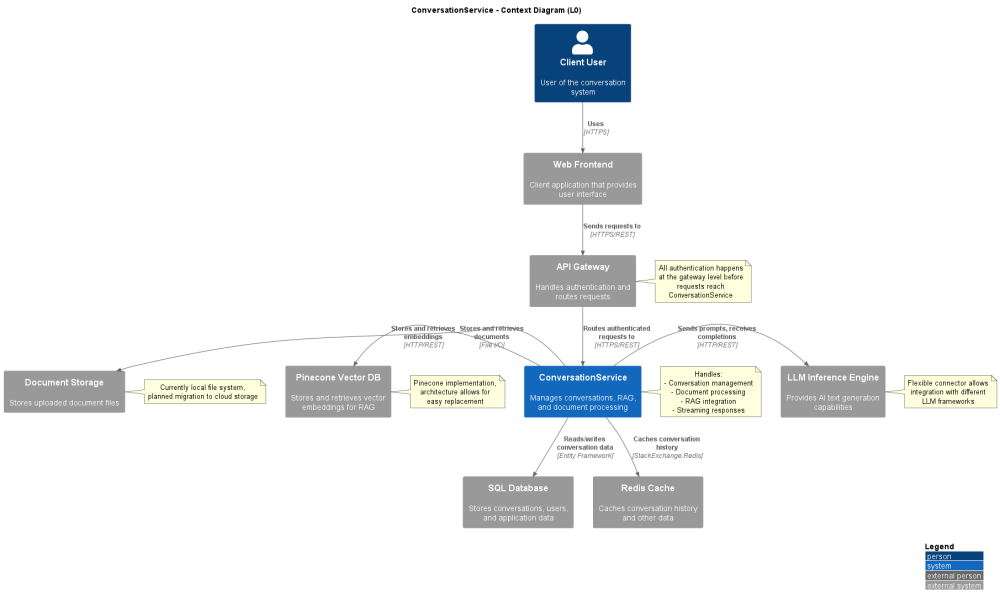

Figure 1.2 – OllamaNet Platform Components

Redis: In-memory data structure store used for caching

RabbitMQ: Message broker software for service-to-service communication

1.5 Implementation Strategy

OllamaNet follows a structured implementation approach:

1. **Service Isolation**: Each microservice addresses a specific domain with clear boundaries, following domain-driven design principles.

2. **Shared Data Layer**: A common DB layer provides consistent data access patterns across all services.

3. **API-First Design**: RESTful APIs with comprehensive documentation via Swagger/OpenAPI ensure clear service interfaces.

4. **Progressive Enhancement**: Services evolve through phased migrations and improvements, with clear versioning.

5. **Performance Optimization**: Strategic caching and efficient data access patterns ensure responsive user experiences.

6. **Security Integration**: Consistent authentication and authorization across services protect resources.

7. **Domain-Driven Design**: Service organization based on business domains and use cases ensures alignment with business needs.

8. **Notebook-First Inference**: The InferenceService uses a notebook-based architecture for flexibility and accessibility.

This implementation strategy enables the platform to be both robust and flexible, catering to enterprise-grade requirements while maintaining developer accessibility and extensibility.

Terminology

REST: Representational State Transfer, an architectural style for designing networked applications

Swagger/OpenAPI: Framework for API documentation and specification

Notebook-First Architecture: Implementation approach that prioritizes interactive development environments

1.6 Report Structure

This documentation is organized to provide a comprehensive understanding of the OllamaNet platform:

- **Chapter 2**: Explores the background and theoretical foundations of microservices architecture
- **Chapter 3**: Details the requirements analysis and domain modeling
- **Chapter 4**: Describes the overall system architecture and communication patterns
- **Chapter 5**: Examines the database layer and data management strategies
- **Chapter 6**: Provides detailed designs for each microservice

- **Chapter 7**: Outlines the frontend architecture and integration
- **Chapter 8**: Explains the testing strategy across services
- **Chapter 9**: Covers implementation details and development practices
- **Chapter 10**: Presents system evaluation and performance metrics
- **Chapter 11**: Concludes with lessons learned and future directions

Each chapter builds on the previous to create a complete picture of the OllamaNet platform's architecture, implementation, and value proposition.

Terminology

Architecture: The high-level structure of a software system

Domain Model: Conceptual model of the domain that incorporates both behavior and data

Communication Pattern: Standard approach for service interactions in a distributed system

Chapter 2

Background & Literature Review

2.1 Theoretical Background

The OllamaNet platform is built on several key theoretical foundations that inform its architecture and design:

2.1.1 Microservices Architecture

Microservices architecture represents a modern approach to software development where applications are composed of small, independent services that communicate over well-defined APIs. This architectural style has gained prominence as an alternative to monolithic applications, offering benefits such as:

- **Independent Deployability**: Each service can be deployed, upgraded, and scaled independently
- **Technology Diversity**: Different services can use different technologies based on their specific requirements
- **Focused Development Teams**: Teams can focus on specific business domains and services
- **Resilience**: Failures in one service can be isolated without affecting the entire system
- **Scalability**: Services can be individually scaled based on demand

Figure 2.1 – Monolithic vs Microservices Architecture

Figure 2.2 – Domain-Driven Design Bounded Contexts

For OllamaNet, microservices architecture enables the separation of concerns between different aspects of the platform (authentication, administration, conversations, exploration, inference) while allowing for independent evolution of each component.

Terminology

Microservice: An architectural style that structures an application as a collection of services that are independently deployable

API Gateway: A server that acts as an API front-end, receiving API requests and routing them to appropriate services

2.1.2 Domain-Driven Design

Domain-Driven Design (DDD) is a software development approach that focuses on modeling the software to match the business domain. It involves:

- **Ubiquitous Language**: A common language shared between developers and domain experts
- **Bounded Contexts**: Explicit boundaries between different domain models
- **Entities and Value Objects**: Clear modeling of domain objects based on identity and characteristics
- **Aggregates**: Clusters of domain objects treated as a unit for data changes
- **Domain Services**: Operations that don't naturally belong to any specific entity

OllamaNet employs DDD principles to ensure each microservice accurately represents its domain model (administration, authentication, conversations, exploration, inference) with appropriate boundaries and clear separation of concerns. This approach resulted in well-defined service boundaries that align with business capabilities.

Terminology

Bounded Context: A conceptual boundary within which a particular domain model applies

Aggregate: A cluster of domain objects that can be treated as a single unit

Ubiquitous Language: A common language used by developers and domain experts to describe domain concepts

2.1.3 API Design and RESTful Principles

The OllamaNet platform follows RESTful API design principles across all its services:

- **Resource-Oriented**: APIs are designed around resources (conversations, models, users)
- **Standard HTTP Verbs**: GET, POST, PUT, DELETE are used for CRUD operations
- **Stateless Communication**: Each request contains all necessary information
- **Standardized Response Formats**: Consistent JSON responses with appropriate HTTP status codes
- **Versioning**: API versioning to support evolution without breaking changes
- **Clear Endpoint Naming**: Intuitive and descriptive endpoint paths

These principles ensure consistency across services and ease of integration for client applications.

2.1.4 Authentication and Authorization Theories

OllamaNet implements a robust authentication and authorization system using established security patterns:

- **Token-Based Authentication**: Using JWT (JSON Web Tokens) for secure, stateless authentication
- **Claims-Based Identity**: User information stored as claims within tokens

- **Role-Based Access Control**: Permissions assigned based on user roles
- **Refresh Token Pattern**: Long-lived refresh tokens used to obtain new access tokens
- **Token Validation**: Comprehensive validation of token signature, expiry, and claims
- **Gateway Authentication**: Centralized authentication at the API Gateway level

2.1.5 Caching Strategies and Patterns

The platform implements several caching strategies to optimize performance:

- **Distributed Caching**: Redis used as a shared cache across services
- **Cache-Aside Pattern**: Data retrieved from cache first, falling back to the database
- **Expiration Policies**: Time-based expiration tailored to data volatility
- **Cache Invalidation**: Strategies to update or invalidate cached data when modified
- **Multi-Level Caching**: In-memory and distributed caching used in tandem
- **Resilient Caching**: Graceful fallback when cache is unavailable

2.1.6 Distributed Systems Concepts

As a distributed system, OllamaNet addresses several core distributed computing challenges:

- **Service Discovery**: Using message brokers (RabbitMQ) for dynamic service URL updates
- **Consistency Models**: Ensuring data consistency across services
- **Failure Handling**: Strategies for graceful degradation when components fail
- **Distributed Tracing**: Request tracing across service boundaries
- **Eventual Consistency**: Accepting temporary inconsistency for system availability
- **Circuit Breaking**: Preventing cascading failures across service calls

Terminology

REST: Representational State Transfer, an architectural style for distributed hypermedia systems

JWT: JSON Web Token, a compact, URL-safe means of representing claims to be transferred between two parties

Caching: Storing copies of data in a high-speed data store to reduce database load

2.2 Similar Systems Analysis

Several existing AI platforms offer similar functionality to OllamaNet, though often with different architectural approaches:

OpenAI API Platform

- **Architecture**: Centralized API services with limited customization
- **Strengths**: Enterprise-grade security, high-reliability, extensive model selection
- **Limitations**: Limited local deployment options, proprietary technology stack, higher cost
- **Comparison**: OllamaNet provides greater customization, local deployment, and cost advantages

Hugging Face Spaces

- **Architecture**: Model hosting platform with standardized deployment
- **Strengths**: Wide model availability, community-driven, integrated UI components
- **Limitations**: Less focus on enterprise features, limited conversation persistence
- **Comparison**: OllamaNet offers stronger administrative controls and conversation management

LangChain

- **Architecture**: Python framework for LLM development, not a complete platform
- **Strengths**: Extensive integrations, component-based architecture, rapid prototyping
- **Limitations**: Python-centric, less focus on enterprise deployment, limited built-in UI
- **Comparison**: OllamaNet provides a complete, production-ready system vs. a development framework

LocalAI

- **Architecture**: Local API server for open-source models
- **Strengths**: Self-hosted, privacy-focused, open-source
- **Limitations**: Limited administrative features, Python-based implementation
- **Comparison**: OllamaNet offers more comprehensive microservices with better separation of concerns

Ollama (Base System)

- **Architecture**: Single-service model server
- **Strengths**: Easy setup, rapidly growing ecosystem, excellent command-line interface
- **Limitations**: Limited administrative features, basic conversation management
- **Comparison**: OllamaNet extends Ollama's capabilities with robust microservices around it

A key advantage of OllamaNet over these systems is its comprehensive microservices approach with strong domain separation, robust database structure, and enterprise-grade features while maintaining the ability to leverage open-source models.

2.3 Technologies Evaluation

The OllamaNet platform has been built using a carefully selected technology stack:

Backend Framework

ASP.NET Core (.NET 9.0) was chosen as the primary backend framework for all microservices due to:

- Performance advantages and scalability
- Comprehensive support for RESTful API development
- Strong typing and compile-time safety
- Rich ecosystem of libraries and tools
- Cross-platform capabilities
- Superior robustness compared to popular Python-based LLM libraries
- Enhanced customizability and extensibility for enterprise implementations
- Better thread management for handling concurrent LLM operations

The decision to use C# and .NET over Python (the most common language for LLM applications) was deliberate, prioritizing long-term maintainability, performance, and enterprise-grade stability over the rapid prototyping advantages of Python libraries. This choice enables developers to build custom extensions and integrations with greater confidence in the system's reliability and scalability.

Database Technologies

SQL Server serves as the primary database technology, offering:

- Strong ACID compliance for transactional integrity
- Robust performance for relational data

- Comprehensive tooling and management capabilities
- Strong integration with Entity Framework Core
- Superior data reliability compared to non-relational alternatives

OllamaNet deliberately employs a relational database schema rather than trending non-relational approaches. While non-relational databases offer faster development cycles and simpler initial setup, the project prioritizes data reliability, relationship integrity, and consistent query performance. This approach ensures conversations, user data, and model information maintain their referential integrity and can be reliably retrieved with consistent performance characteristics, even as the data grows in complexity and volume.

Redis provides distributed caching capabilities, delivering:

- High-performance in-memory data storage
- Support for various data structures
- Pub/Sub capabilities for real-time features
- Distributed caching across services

Authentication & Authorization

JWT (JSON Web Tokens) with refresh token functionality was implemented for:

- Stateless authentication between services
- Secure transmission of claims
- Support for token expiration and renewal
- Cross-service authorization

API Documentation

Swagger/OpenAPI was selected for API documentation because it offers:

- Interactive API exploration and testing
- Automatic documentation generation
- Client code generation capabilities
- Standardized API specifications

ORM Solution

Entity Framework Core serves as the object-relational mapping solution, providing:

- Clean abstraction over database operations
- LINQ support for type-safe queries
- Migrations for database schema evolution
- Comprehensive relationship mapping

Additional Libraries and Tools

- **FluentValidation**: Comprehensive request validation
- **Polly**: Resilience policies for external service calls
- **OllamaSharp**: Client library for Ollama API integration
- **StackExchange.Redis**: Redis client for distributed caching
- **RabbitMQ Client**: Message broker integration for service discovery
- **Jupyter Notebook** (for InferenceService): Interactive development environment
- **ngrok** (for InferenceService): Secure tunneling for notebook-based services

Figure 2.3 – Deployment Comparison Between Architectures

Terminology

ORM: Object-Relational Mapping, a technique for converting data between incompatible type systems

ACID: Atomicity, Consistency, Isolation, Durability - properties of database transactions that guarantee validity

Pub/Sub: Publish/Subscribe pattern where senders don't send messages directly to receivers

2.4 Architectural Patterns

2.4.1 Monolithic vs Microservices Comparison

OllamaNet chose a microservices architecture over a monolithic approach after careful consideration of trade-offs:

Table 2.1 – Comparison between Monolithic and Microservices Approaches

Aspect	**Monolithic Approach**	**Microservices Approach**	**OllamaNet Decision**
Deployment	Simple but all-or-nothing	Complex but granular	Microservices for deployment flexibility
Development	Simple coordination but coupled	Independent but requires interfaces	Microservices for team autonomy
Scaling	Vertical scaling of entire application	Horizontal scaling of specific services	Microservices for targeted scaling
Technology	Single technology stack	Technology diversity	Microservices with consistent .NET stack
Resilience	Single point of failure	Isolated failures	Microservices for fault isolation
Complexity	Lower initial complexity	Higher distributed complexity	Microservices with careful boundary design

Figure 2.4 – OllamaNet Microservices Architecture

2.4.2 Microservices Architecture Overview

OllamaNet implements a microservices architecture with the following characteristics:

- **Service Boundaries**: Services are divided along clear domain boundaries (administration, authentication, conversations, exploration, inference)
- **API Gateway Pattern**: Gateway service provides a single entry point for clients
- **Shared Data Layer**: Common DB layer for consistent data access patterns across services
- **Event-Driven Communication**: Services communicate through events for loose coupling
- **Distributed Caching**: Redis-based caching for performance optimization
- **Authentication Integration**: JWT-based authentication shared across services
- **Service Discovery**: Dynamic discovery of service endpoints, especially critical for the notebook-based InferenceService

2.4.3 Design Patterns in Microservices

OllamaNet implements several design patterns to solve common challenges in microservices architecture:

- **Repository Pattern**: Abstracts data access logic from business logic across all services
- **Unit of Work Pattern**: Coordinates operations across multiple repositories
- **Mediator Pattern**: Decouples request handling from business logic
- **Circuit Breaker Pattern**: Prevents cascading failures across services
- **Retry Pattern**: Automatic retry with exponential backoff for transient failures
- **API Gateway Pattern**: Service-specific APIs with consistent patterns

Figure 2.5 – Design Patterns Implementation

- **Service Discovery Pattern**: Dynamic discovery of service endpoints via RabbitMQ
- **Caching Strategy Pattern**: Multi-level caching for performance optimization
- **Configuration Management Pattern**: Centralized configuration with dynamic updates
- **Decorator Pattern**: Authentication and authorization implemented as decorators
- **Strategy Pattern**: Different services have different routing strategies

A unique aspect of OllamaNet's design is the notebook-first architecture of the InferenceService, which combines Python flexibility with the robustness of .NET microservices. This service uses ngrok tunneling and RabbitMQ-based service discovery to expose Ollama LLM capabilities from any cloud notebook environment, while maintaining secure integration with the broader platform.

Terminology

Repository Pattern: A design pattern that mediates between the domain and data mapping layers

Unit of Work: A pattern that maintains a list of objects affected by a business transaction

Circuit Breaker: A design pattern that detects failures and prevents further requests to failing components

Chapter 3

Requirements Analysis

3.1 Stakeholder Analysis

3.1.1 Identification of Key Stakeholders

The OllamaNet platform serves a diverse group of stakeholders with varying needs and interests:

1. **End Users**
    - Regular users seeking AI-powered conversations
    - Knowledge workers and researchers
    - Content creators and developers
    - Students and educators
2. **Administrative Personnel**
    - Platform administrators
    - System operators
    - Security administrators
    - DevOps engineers
3. **Organization Stakeholders**

Figure 3.1 – Stakeholder Analysis Diagram

- IT departments
- Management teams
- Security and compliance teams
- Model development teams

4. **Technical Stakeholders**
    - Frontend application developers
    - Integration partners
    - API consumers
    - Infrastructure providers
5. **Governance Stakeholders**
    - Data privacy officers
    - Compliance managers
    - Legal representatives
    - Information security officers

3.1.2 Stakeholder Needs and Concerns

End User Needs

- Intuitive and responsive user interface
- Persistent conversation history
- Well-organized conversation management
- Fast and accurate AI responses

- Data privacy and security
- Seamless authentication
- Consistent experience across sessions
- Search capabilities for past interactions
- Access to a variety of AI models

Administrative Personnel Needs

- Comprehensive user management capabilities
- Fine-grained access control
- System monitoring and analytics
- Model deployment and management tools
- Platform configuration controls
- Efficient troubleshooting capabilities
- Audit and compliance features
- Task automation

Organization Stakeholders' Concerns

- Total cost of ownership
- Compliance with organizational policies
- Data security and privacy
- User productivity and efficiency
- Integration with existing systems
- Customization capabilities

- Governance and oversight

Technical Stakeholders' Needs

- Well-documented APIs
- Reliable service availability
- Consistent data models
- Proper error handling
- Scalable architecture
- Performance optimization
- Testability and debugging support

Governance Stakeholders' Concerns

- Regulatory compliance (GDPR, CCPA, etc.)
- Data residency requirements
- Audit trails and reporting
- Security controls and monitoring
- Risk management
- Intellectual property protection

3.1.3 Priority Matrix of Stakeholder Requirements

The following matrix presents the relative priority of key requirements across different stakeholder groups, rated as High (H), Medium (M), or Low (L) priority:

Table 3.1 – Priority Matrix of Stakeholder Requirements

Requirement	End Users	Admins	Organization	Technical	Governance
Conversation Persistence	H	L	M	L	M
Model Discovery	H	M	M	L	L
User Authentication	M	H	H	M	H
Admin Controls	L	H	H	L	M
Performance	H	M	M	H	L
Security	M	H	H	M	H
API Access	L	M	L	H	L
Data Management	M	H	M	M	H
Scalability	L	H	H	H	L
Compliance	L	M	H	L	H

3.1.4 Conflicting Requirements and Resolution Approach

Several areas of potential conflict have been identified between different stakeholders' requirements:

1. **Performance vs. Security**
    - **Conflict**: End users prioritize fast response times, while security stakeholders require thorough validation and protection measures that may introduce latency.
    - **Resolution**: Implement performance-optimized security measures, such as caching JWT validation results, using distributed authentication, and implementing parallel processing where possible.
2. **Usability vs. Compliance**
    - **Conflict**: Seamless user experience may conflict with compliance requirements for explicit consents and disclosures.
    - **Resolution**: Design user-friendly compliance features that integrate organically into the workflow, such as progressive disclosure of terms and just-in-time consent mechanisms.
3. **Flexibility vs. Governance**
    - **Conflict**: Technical stakeholders want maximum flexibility and customization, while governance stakeholders require standardization and controlled processes.

- **Resolution**: Implement a modular architecture with clear extension points, coupled with governance policies on which aspects can be customized and how changes are managed.

4. **Notebook-First Architecture vs. Reliability**

   - **Conflict**: The InferenceService's notebook-first architecture provides flexibility but creates challenges for reliability and management.
   - **Resolution**: Implement robust service discovery mechanisms, monitoring, and auto-recovery while preserving the flexibility of the notebook environment.

5. **Centralized vs. Distributed Data**

   - **Conflict**: Performance and user experience benefit from distributed caching, while governance and management prefer centralized control.
   - **Resolution**: Implement a hybrid approach with a primary centralized database of record and distributed caching with clear invalidation strategies.

Terminology

Stakeholder: Any person, group, or organization with an interest in or concern about the project

Requirements Priority Matrix: A visualization showing the relative importance of requirements to different stakeholders

3.2 Functional Requirements

3.2.1 Core Platform Capabilities

The OllamaNet platform must provide the following core capabilities:

1. **User Authentication and Authorization**

   - The system shall provide secure user registration and login facilities

- The system shall support role-based access control
- The system shall implement JWT-based authentication with refresh token support
- The system shall enforce appropriate authorization checks across all services

2. **Conversation Management**

- The system shall enable creation, retrieval, update, and deletion of conversations
- The system shall maintain conversation history persistently
- The system shall support organizing conversations in folders
- The system shall provide search functionality for past conversations
- The system shall enable real-time streaming of AI responses

3. **AI Model Interaction**

- The system shall connect to the Ollama inference engine
- The system shall support multiple AI models
- The system shall maintain context during conversation
- The system shall provide streaming responses for real-time interaction
- The system shall support document-based context enhancement

4. **Administration and Governance**

- The system shall provide tools for user management
- The system shall enable AI model configuration and deployment
- The system shall support categorization through a tagging system
- The system shall maintain audit logs for administrative actions
- The system shall provide monitoring capabilities

5. **Content Discovery**

- The system shall enable browsing and searching for AI models

Figure 3.2 – Core Platform Capabilities

- The system shall provide detailed model information
- The system shall support filtering models by tags and categories
- The system shall optimize discovery through caching

3.2.2 Service-Specific Functionality Requirements

AuthService Requirements

1. **User Account Management**
   - FR-AUTH-01: The service shall allow registration of new users with email, password, and basic profile information
   - FR-AUTH-02: The service shall validate email addresses through confirmation mechanisms
   - FR-AUTH-03: The service shall support password reset functionality
   - FR-AUTH-04: The service shall enforce password complexity requirements
2. **Authentication Mechanisms**
   - FR-AUTH-05: The service shall authenticate users via username/email and password
   - FR-AUTH-06: The service shall issue JWT tokens upon successful authentication
   - FR-AUTH-07: The service shall support refresh tokens for maintaining sessions
   - FR-AUTH-08: The service shall implement token revocation for security purposes
3. **Authorization Management**
   - FR-AUTH-09: The service shall support role assignment to users
   - FR-AUTH-10: The service shall provide APIs for role management
   - FR-AUTH-11: The service shall enforce role-based access control
   - FR-AUTH-12: The service shall validate tokens and claims for all protected endpoints

AdminService Requirements

1. **User Administration**
    - FR-ADMIN-01: The service shall provide CRUD operations for user management
    - FR-ADMIN-02: The service shall support role assignment and revocation
    - FR-ADMIN-03: The service shall enable account status management (activation/deactivation)
    - FR-ADMIN-04: The service shall support bulk operations for administrative efficiency
2. **Model Administration**
    - FR-ADMIN-05: The service shall enable registration of AI models with metadata
    - FR-ADMIN-06: The service shall provide model update and deletion capabilities
    - FR-ADMIN-07: The service shall support model categorization through tags
    - FR-ADMIN-08: The service shall allow model activation and deactivation
3. **Tag Management**
    - FR-ADMIN-09: The service shall provide CRUD operations for tags
    - FR-ADMIN-10: The service shall enable association of tags with models
    - FR-ADMIN-11: The service shall support hierarchical tag relationships
    - FR-ADMIN-12: The service shall provide search and filtering for tags
4. **Model Deployment**
    - FR-ADMIN-13: The service shall connect to the Ollama API for model operations
    - FR-ADMIN-14: The service shall support model installation with progress tracking
    - FR-ADMIN-15: The service shall provide model information retrieval
    - FR-ADMIN-16: The service shall enable model removal and cleanup

ConversationService Requirements

1. **Conversation Management**
    - FR-CONV-01: The service shall provide CRUD operations for conversations
    - FR-CONV-02: The service shall support conversation title management
    - FR-CONV-03: The service shall enable conversation search and filtering
    - FR-CONV-04: The service shall manage conversation archiving and deletion
2. **Chat Functionality**
    - FR-CONV-05: The service shall enable sending messages to AI models
    - FR-CONV-06: The service shall support streaming responses for real-time interaction
    - FR-CONV-07: The service shall maintain conversation context for improved responses
    - FR-CONV-08: The service shall track and report token usage
3. **Organization Features**
    - FR-CONV-09: The service shall provide folder CRUD operations
    - FR-CONV-10: The service shall support moving conversations between folders
    - FR-CONV-11: The service shall enable note-taking associated with conversations
    - FR-CONV-12: The service shall support tagging and categorization of conversations
4. **Document Integration**
    - FR-CONV-13: The service shall enable document uploading and management
    - FR-CONV-14: The service shall process documents for content extraction
    - FR-CONV-15: The service shall use document content for context enhancement
    - FR-CONV-16: The service shall support multiple document formats (PDF, TXT, DOCX)
5. **Feedback Collection**
    - FR-CONV-17: The service shall provide mechanisms for users to rate AI responses

- FR-CONV-18: The service shall collect optional feedback comments
- FR-CONV-19: The service shall store feedback for future analysis
- FR-CONV-20: The service shall associate feedback with specific AI responses

ExploreService Requirements

1. **Model Discovery**
   - FR-EXPL-01: The service shall provide a catalog of available AI models
   - FR-EXPL-02: The service shall support browsing models by categories
   - FR-EXPL-03: The service shall enable searching for models by keywords
   - FR-EXPL-04: The service shall support filtering models by various attributes
2. **Model Information**
   - FR-EXPL-05: The service shall provide detailed model metadata
   - FR-EXPL-06: The service shall display model capabilities and specifications
   - FR-EXPL-07: The service shall show associated tags and categories
   - FR-EXPL-08: The service shall include model usage information when available
3. **Performance Optimization**
   - FR-EXPL-09: The service shall implement caching for frequently accessed model data
   - FR-EXPL-10: The service shall optimize search operations for performance
   - FR-EXPL-11: The service shall use pagination for large result sets
   - FR-EXPL-12: The service shall implement cache invalidation strategies

InferenceService Requirements

1. **Model Inference**
   - FR-INF-01: The service shall provide API endpoints for AI model inference

- FR-INF-02: The service shall support text completion requests
- FR-INF-03: The service shall enable streaming responses
- FR-INF-04: The service shall maintain connection with the Ollama backend

2. **Service Discovery**

- FR-INF-05: The service shall dynamically publish its endpoint URL
- FR-INF-06: The service shall use RabbitMQ for service discovery messages
- FR-INF-07: The service shall implement secure tunneling via ngrok
- FR-INF-08: The service shall provide health check endpoints

3. **Notebook Integration**

- FR-INF-09: The service shall operate in cloud notebook environments
- FR-INF-10: The service shall handle environment initialization
- FR-INF-11: The service shall manage Ollama and ngrok processes
- FR-INF-12: The service shall provide interactive operation controls

3.2.3 API Requirements

The OllamaNet platform's APIs must adhere to the following requirements:

1. **REST Principles**

- The APIs shall follow RESTful design practices
- The APIs shall use standard HTTP verbs appropriately (GET, POST, PUT, DELETE)
- The APIs shall return appropriate HTTP status codes
- The APIs shall implement proper resource naming conventions

2. **API Documentation**

- All APIs shall be documented using Swagger/OpenAPI

- API documentation shall include endpoint descriptions, parameters, and response formats
- API documentation shall be available through interactive endpoints
- API documentation shall include example requests and responses

3. **Request/Response Format**
   - APIs shall accept and return data in JSON format
   - APIs shall implement consistent request validation
   - APIs shall provide clear error messages and codes
   - APIs shall use pagination for large result sets

4. **Security Controls**
   - APIs shall require authentication for protected resources
   - APIs shall validate JWT tokens for protected endpoints
   - APIs shall implement appropriate CORS policies
   - APIs shall sanitize inputs to prevent injection attacks

5. **Versioning**
   - APIs shall support versioning to maintain backward compatibility
   - API versions shall be included in the URL path
   - API changes shall be documented between versions
   - Deprecated API versions shall be clearly marked

3.2.4 Integration Requirements

The OllamaNet platform requires the following integration capabilities:

1. **Service-to-Service Communication**
   - Services shall communicate via well-defined APIs

- Services shall handle communication errors gracefully
- Services shall implement circuit breakers for resilience
- Services shall validate data received from other services

2. **External System Integration**

- The platform shall integrate with Ollama for model inference
- The platform shall support ngrok for dynamic service exposure
- The platform shall implement RabbitMQ for messaging and service discovery
- The platform shall provide extensibility points for future integrations

3. **Frontend Integration**

- The platform shall provide APIs suitable for frontend consumption
- The platform shall implement appropriate CORS settings
- The platform shall support real-time features through streaming APIs
- The platform shall implement API rate limiting for fairness

4. **Data Synchronization**

- The platform shall maintain data consistency across services
- The platform shall implement appropriate caching strategies
- The platform shall handle concurrent modifications appropriately
- The platform shall provide mechanisms for data reconciliation

3.2.5 Security and Access Control Requirements

1. **Authentication**

- The platform shall implement JWT-based authentication
- The platform shall support refresh tokens for session maintenance
- The platform shall enforce token expiration and validation

- The platform shall provide secure password management

2. **Authorization**

   - The platform shall implement role-based access control
   - The platform shall enforce authorization at the API Gateway level
   - The platform shall propagate user claims to downstream services
   - The platform shall validate permissions for protected operations

3. **Data Protection**

   - The platform shall encrypt sensitive data at rest
   - The platform shall use HTTPS for all communications
   - The platform shall implement proper data isolation between users
   - The platform shall provide secure credential storage

4. **Audit and Compliance**

   - The platform shall maintain audit logs for security events
   - The platform shall log authentication attempts and failures
   - The platform shall track administrative actions
   - The platform shall support compliance reporting

3.2.6 Data Management Requirements

1. **Data Storage**

   - The platform shall use SQL Server for relational data storage
   - The platform shall implement proper schema design
   - The platform shall maintain referential integrity
   - The platform shall support data migration and evolution

2. **Data Access**

- The platform shall implement the repository pattern for data access
- The platform shall use the unit of work pattern for transaction management
- The platform shall provide efficient query capabilities
- The platform shall support pagination for large data sets

3. **Caching**

- The platform shall implement Redis for distributed caching
- The platform shall use appropriate cache expiration policies
- The platform shall maintain cache consistency
- The platform shall provide fallback mechanisms for cache failures

4. **Data Lifecycle**

- The platform shall support soft deletion for most entities
- The platform shall implement data archiving strategies
- The platform shall provide data export capabilities
- The platform shall handle data purging for compliance

Terminology

Functional Requirement: A requirement that specifies what the system should do

Service Capability: A discrete function or set of functions provided by a service

3.3 Non-Functional Requirements

3.3.1 Performance Requirements

The OllamaNet platform must meet the following performance requirements:

1. **Response Time**

- NFR-PERF-01: API endpoints shall respond within 200ms for non-computational operations
- NFR-PERF-02: Authentication operations shall complete within 500ms
- NFR-PERF-03: Database queries shall execute within 100ms for 95% of requests
- NFR-PERF-04: Static resource delivery shall occur within 50ms
- NFR-PERF-05: AI inference first-token response shall occur within 1 second

2. **Throughput**

- NFR-PERF-06: The platform shall support at least 100 concurrent users per instance
- NFR-PERF-07: The platform shall process at least 50 API requests per second per instance
- NFR-PERF-08: The platform shall manage at least 25 concurrent conversation sessions per instance
- NFR-PERF-09: The administrative API shall handle at least 20 requests per second
- NFR-PERF-10: Each service shall be capable of horizontal scaling to increase throughput

3. **Caching Performance**

- NFR-PERF-11: Redis cache access shall complete within 10ms
- NFR-PERF-12: Cache hit ratio shall exceed 80% for frequently accessed data
- NFR-PERF-13: JWT validation cache shall maintain a hit ratio above 90%
- NFR-PERF-14: Cache invalidation shall propagate within 5 seconds
- NFR-PERF-15: Cache warm-up shall complete within 30 seconds of service start

4. **Resource Utilization**

- NFR-PERF-16: Services shall utilize less than 70% CPU under normal load
- NFR-PERF-17: Memory consumption shall not exceed 2GB per service instance

- NFR-PERF-18: Database connections shall be pooled with a maximum of 100 connections
- NFR-PERF-19: Network bandwidth usage shall remain below 50Mbps under normal operations
- NFR-PERF-20: Disk I/O shall remain below 70% utilization

3.3.2 Scalability Requirements

The OllamaNet platform must satisfy the following scalability requirements:

1. **Horizontal Scaling**
   - NFR-SCAL-01: All services shall support horizontal scaling through multiple instances
   - NFR-SCAL-02: The platform shall support auto-scaling based on predefined metrics
   - NFR-SCAL-03: Service instances shall be stateless to facilitate scaling
   - NFR-SCAL-04: The platform shall maintain performance under load with linear resource addition
   - NFR-SCAL-05: Node addition shall not require system restart
2. **Database Scalability**
   - NFR-SCAL-06: The database shall support partitioning for data growth
   - NFR-SCAL-07: The system shall support read replicas for query scaling
   - NFR-SCAL-08: Database operations shall use appropriate indexing for scale
   - NFR-SCAL-09: The platform shall implement database connection pooling
   - NFR-SCAL-10: Query patterns shall be optimized for large data volumes
3. **Caching Scalability**
   - NFR-SCAL-11: Redis cache shall support cluster mode for horizontal scaling
   - NFR-SCAL-12: Cache size shall be configurable based on deployment environment

Figure 3.3 – OllamaNet Scalability Architecture

- NFR-SCAL-13: The platform shall implement multiple cache levels for scalability
- NFR-SCAL-14: Cache eviction policies shall optimize for memory utilization
- NFR-SCAL-15: The platform shall gracefully handle cache server failures

4. **Load Handling**

- NFR-SCAL-16: The platform shall implement rate limiting to prevent overload
- NFR-SCAL-17: The platform shall queue excess requests rather than reject them
- NFR-SCAL-18: The platform shall degrade gracefully under extreme load
- NFR-SCAL-19: Backend services shall implement backpressure mechanisms
- NFR-SCAL-20: The platform shall support traffic prioritization

3.3.3 Security Requirements

The OllamaNet platform must comply with the following security requirements:

1. **Authentication and Authorization**

- NFR-SEC-01: User passwords shall be stored using industry-standard hashing algorithms (bcrypt)
- NFR-SEC-02: JWT tokens shall expire after 15 minutes of inactivity
- NFR-SEC-03: Refresh tokens shall expire after 30 days
- NFR-SEC-04: Failed login attempts shall be limited to prevent brute force attacks
- NFR-SEC-05: Role-based access control shall be enforced for all protected resources

2. **Data Protection**

- NFR-SEC-06: All communications shall use TLS 1.3 or higher
- NFR-SEC-07: Sensitive data shall be encrypted at rest using AES-256

- NFR-SEC-08: Database credentials shall be stored in secure environment variables
- NFR-SEC-09: Production environments shall enforce strict network isolation
- NFR-SEC-10: Data access shall follow the principle of least privilege

3. **API Security**

- NFR-SEC-11: APIs shall validate all inputs to prevent injection attacks
- NFR-SEC-12: CORS policies shall restrict access to approved origins
- NFR-SEC-13: API endpoints shall implement rate limiting
- NFR-SEC-14: Error responses shall not expose implementation details
- NFR-SEC-15: API security headers shall be implemented (HSTS, X-XSS-Protection, etc.)

4. **Audit and Compliance**

- NFR-SEC-16: Security-related events shall be logged with appropriate details
- NFR-SEC-17: Audit logs shall be immutable and tamper-evident
- NFR-SEC-18: The platform shall support regulatory compliance reporting
- NFR-SEC-19: Privileged operations shall require additional authentication
- NFR-SEC-20: Regular security assessments shall be supported

5. **Vulnerability Management**

- NFR-SEC-21: The platform shall not use components with known vulnerabilities
- NFR-SEC-22: The platform shall be designed to mitigate OWASP Top 10 vulnerabilities
- NFR-SEC-23: Security patches shall be applicable without service interruption
- NFR-SEC-24: Dependencies shall be regularly updated to address security issues
- NFR-SEC-25: Security testing shall be integrated into the development process

3.3.4 Reliability and Availability Requirements

The OllamaNet platform must meet the following reliability and availability requirements:

1. **System Availability**
    - NFR-REL-01: The platform shall maintain 99.9% uptime (excluding planned maintenance)
    - NFR-REL-02: Planned maintenance shall occur during off-peak hours
    - NFR-REL-03: No single point of failure shall exist in the production environment
    - NFR-REL-04: The system shall support rolling updates without downtime
    - NFR-REL-05: The platform shall detect and restart failed service instances automatically
2. **Fault Tolerance**
    - NFR-REL-06: The platform shall implement circuit breakers for external service calls
    - NFR-REL-07: Services shall fail gracefully when dependencies are unavailable
    - NFR-REL-08: The system shall implement retry logic with exponential backoff
    - NFR-REL-09: The platform shall maintain data integrity during partial system failures
    - NFR-REL-10: The system shall recover automatically from most failure scenarios
3. **Data Reliability**
    - NFR-REL-11: The database shall use transaction isolation to prevent data corruption
    - NFR-REL-12: The platform shall implement data backups with point-in-time recovery
    - NFR-REL-13: Data integrity constraints shall be enforced at the database level
    - NFR-REL-14: The system shall detect and log data inconsistencies
    - NFR-REL-15: Critical data changes shall be audited with before/after values
4. **Disaster Recovery**
    - NFR-REL-16: The platform shall support database failover to standby replicas

- NFR-REL-17: Recovery Point Objective (RPO) shall be less than 5 minutes
- NFR-REL-18: Recovery Time Objective (RTO) shall be less than 30 minutes
- NFR-REL-19: Disaster recovery procedures shall be documented and tested
- NFR-REL-20: The system shall support geographic redundancy for critical components

3.3.5 Maintainability Requirements

The OllamaNet platform must adhere to the following maintainability requirements:

1. **Code Quality**
   - NFR-MAIN-01: Code shall follow language-specific style guidelines
   - NFR-MAIN-02: Code complexity metrics shall remain below specified thresholds
   - NFR-MAIN-03: Test coverage shall exceed 80% for critical components
   - NFR-MAIN-04: Static code analysis shall be part of the build process
   - NFR-MAIN-05: Code shall be properly commented and documented
2. **Deployment**
   - NFR-MAIN-06: The platform shall support containerized deployment
   - NFR-MAIN-07: Configuration shall be externalized from code
   - NFR-MAIN-08: The system shall support environment-specific configuration
   - NFR-MAIN-09: Deployment shall be automated via CI/CD pipelines
   - NFR-MAIN-10: Deployments shall be reversible through rollback mechanisms
3. **Monitoring and Diagnostics**
   - NFR-MAIN-11: Services shall expose health check endpoints
   - NFR-MAIN-12: The system shall generate appropriate logs for troubleshooting
   - NFR-MAIN-13: Performance metrics shall be collected and available for analysis

- NFR-MAIN-14: Error conditions shall trigger appropriate alerts
- NFR-MAIN-15: The platform shall support distributed tracing

4. **Extensibility**

   - NFR-MAIN-16: The system architecture shall support adding new services
   - NFR-MAIN-17: APIs shall be versioned to support evolution
   - NFR-MAIN-18: The database schema shall support extension without redesign
   - NFR-MAIN-19: User interface components shall be modular and reusable
   - NFR-MAIN-20: The system shall support plugin architectures where appropriate

3.3.6 Compatibility and Interoperability Requirements

The OllamaNet platform must satisfy the following compatibility and interoperability requirements:

1. **Client Compatibility**

   - NFR-COMP-01: Frontend applications shall work on modern browsers (Chrome, Firefox, Safari, Edge)
   - NFR-COMP-02: The system shall support responsive design for mobile and desktop access
   - NFR-COMP-03: APIs shall be accessible from common programming languages
   - NFR-COMP-04: The platform shall support common authentication mechanisms (JWT, OAuth)
   - NFR-COMP-05: Public interfaces shall follow industry standards for maximum compatibility

2. **Integration Compatibility**

   - NFR-COMP-06: The platform shall use standard protocols for integration (REST, AMQP)

- NFR-COMP-07: Data exchange shall use standardized formats (JSON, XML)
- NFR-COMP-08: APIs shall provide Swagger/OpenAPI documentation
- NFR-COMP-09: The platform shall support standard database connection mechanisms
- NFR-COMP-10: Integration points shall be well documented for third-party developers

3. **Environment Compatibility**

- NFR-COMP-11: The platform shall operate in common cloud environments (AWS, Azure, GCP)
- NFR-COMP-12: </rewritten$_{file}$ >

Chapter 4

System Architecture

4.1 Overall Architecture

4.1.1 High-level System Architecture Overview

The OllamaNet platform is built on a modern microservices architecture that separates concerns into distinct, independently deployable services. Each service focuses on a specific domain within the system, communicating through well-defined APIs and messaging patterns.

The architecture consists of the following key components:

(a) **API Gateway**: Serves as the entry point for all client requests, handling routing, authentication, and cross-cutting concerns.

(b) **Auth Service**: Manages user authentication, authorization, and account management.

(c) **Admin Service**: Provides administrative capabilities for user and model management.

(d) **Explore Service**: Enables discovery and browsing of available AI models.

(e) **Conversation Service**: Handles conversation management and interaction with AI models.

(f) **Inference Service**: Connects to the Ollama engine for AI model inference.

(g) **Database Layer**: Provides data persistence across services.

(h) **RabbitMQ**: Facilitates asynchronous communication and service discovery.

4.1.2 Architectural Principles and Goals

The OllamaNet architecture adheres to the following principles:

(a) **Service Independence**: Each service can be developed, deployed, and scaled independently.

(b) **Domain-Driven Design**: Services are organized around business domains rather than technical functions.

(c) **API-First Design**: All services expose well-defined APIs with consistent patterns.

(d) **Resilience by Design**: The system is designed to handle failures gracefully.

(e) **Security at Every Layer**: Authentication and authorization are enforced consistently.

(f) **Scalability**: Services can be scaled independently based on demand.

(g) **Observability**: The system provides monitoring, logging, and diagnostics capabilities.

The architectural goals include:

(a) **Maintainability**: Clear separation of concerns makes the system easier to maintain.

(b) **Extensibility**: New features can be added with minimal impact on existing components.

(c) **Performance**: The architecture optimizes for responsive user experiences.

(d) **Security**: The system protects user data and prevents unauthorized access.

(e) **Reliability**: The system remains available and consistent even during partial failures.

4.1.3 System Topology and Deployment View

The OllamaNet system is designed for flexible deployment across various environments:

Figure 4.1 – OllamaNet System Topology and Deployment View

The deployment architecture supports:

(a) **Containerization**: All services are containerized for consistent deployment.

(b) **Horizontal Scaling**: Services can be scaled out by adding instances.

(c) **Environment Isolation**: Development, testing, and production environments are isolated.

(d) **Cloud Deployment**: The system can be deployed to various cloud providers.

(e) **On-Premises Deployment**: The system can also be deployed on-premises.

4.1.4 Key Architectural Decisions and Rationales

Table 4.1 – Key Architectural Decisions and Rationales

Decision	**Rationale**
Microservices Architecture	Enables independent development, deployment, and scaling of components
API Gateway Pattern	Provides a single entry point for clients, simplifying client integration
Domain-Driven Design	Aligns services with business domains for better maintainability
JWT Authentication	Enables stateless authentication across services
SQL Server Database	Provides robust relational data storage with strong consistency
Redis Caching	Improves performance by caching frequently accessed data
RabbitMQ Messaging	Enables asynchronous communication and service discovery
Notebook-First Inference	Allows flexible deployment of inference capabilities in cloud environments

4.2 Service Decomposition Strategy

4.2.1 Microservice Boundaries and Responsibilities

The OllamaNet platform is decomposed into services based on business domains and responsibilities:

(a) **Auth Service**

- ▶ User registration and authentication
- ▶ JWT token issuance and validation
- ▶ Role and permission management
- ▶ User profile management

(b) **Admin Service**

- ▶ User account administration
- ▶ AI model management and deployment
- ▶ Tag and category management
- ▶ System configuration and monitoring

(c) **Explore Service**

- ▶ AI model discovery and browsing
- ▶ Search and filtering capabilities
- ▶ Model metadata presentation
- ▶ Caching for performance optimization

(d) **Conversation Service**

- ▶ Conversation management and organization
- ▶ Chat interaction with AI models
- ▶ Document processing for context enhancement
- ▶ Folder and organization features

(e) **Inference Service (Spicy Avocado)**

- ▶ AI model inference and response generation

Figure 4.2 – OllamaNet Bounded Contexts in Domain-Driven Design

- Service discovery via RabbitMQ
- Notebook-first architecture for cloud deployment
- Integration with Ollama engine

(f) **Gateway Service**

- Request routing to appropriate services
- Authentication and authorization enforcement
- Cross-cutting concerns like CORS and rate limiting
- Request/response transformation

4.2.2 Domain-Driven Design Application

The service decomposition follows Domain-Driven Design principles:

(a) **Bounded Contexts**: Each service represents a distinct bounded context with its own domain model.

(b) **Ubiquitous Language**: Each service uses consistent terminology within its domain.

(c) **Aggregates**: Domain entities are organized into aggregates with clear boundaries.

(d) **Domain Events**: Services communicate through domain events when appropriate.

4.2.3 Service Granularity Decisions

The service granularity in OllamaNet balances several factors:

(a) **Business Domain Alignment**: Services align with distinct business capabilities.

(b) **Team Ownership**: Services can be owned by specific teams.

(c) **Deployment Independence**: Services can be deployed independently.

(d) **Scalability Requirements**: Services can be scaled based on their specific load patterns.

Figure 4.3 – OllamaNet Service Dependencies

For example, the Inference Service is separated from the Conversation Service because:

- It has different scaling requirements (compute-intensive)
- It has a unique deployment model (notebook-first architecture)
- It integrates with external systems (Ollama engine)

4.2.4 Service Composition and Dependencies

Services in OllamaNet have the following dependencies:

(a) **Auth Service**: No dependencies on other services

(b) **Admin Service**: Depends on Auth Service for authentication

(c) **Explore Service**: Depends on Auth Service for authentication

(d) **Conversation Service**:

- Depends on Auth Service for authentication
- Depends on Inference Service for AI model responses

(e) **Inference Service**: No direct dependencies on other services

(f) **Gateway Service**: Depends on all services for routing

4.3 Service Discovery and Registry

4.3.1 Service Discovery Mechanisms

OllamaNet implements service discovery using RabbitMQ, particularly for the dynamic discovery of Inference Service endpoints:

(a) **Publisher-Subscriber Pattern**: Services publish their endpoints to RabbitMQ topics.

(b) **Topic Exchange**: A "service-discovery" exchange routes messages based on routing keys.

Figure 4.4 – Service Discovery Sequence Diagram

(c) **Routing Keys**: Structured keys like "inference.url.changed" identify message types.

(d) **Message Format**: JSON messages contain service URLs and metadata.

4.3.2 Dynamic Service URL Configuration

The system handles dynamic URL configuration through several mechanisms:

(a) **Initial Configuration**: Services start with default URLs from configuration files.

(b) **Runtime Updates**: Services receive URL updates via RabbitMQ messages.

(c) **Configuration Service**: A central service manages and distributes configuration.

(d) **Caching**: Updated URLs are cached in Redis for persistence across restarts.

```
public async Task UpdateBaseUrl(string newUrl)
{
    if (string.IsNullOrEmpty(newUrl) || _currentBaseUrl == newUrl)
        return;

    if (!_urlValidator.IsValid(newUrl))
    {
        _logger.LogWarning("Received invalid URL update: {Url}", newUrl);
        return;
    }

    _currentBaseUrl = newUrl;
    await _redisCacheService.SetStringAsync(CACHE_KEY, newUrl);
    _logger.LogInformation("InferenceEngine URL updated to: {Url}", newUrl);

    BaseUrlChanged?.Invoke(newUrl);
}
```

Figure 4.5 – Gateway Architecture Components

4.3.3 Service Registration Approaches

Services register themselves through different mechanisms:

(a) **Static Registration**: Most services have static endpoints defined in configuration.

(b) **Dynamic Registration**: The Inference Service dynamically registers its endpoint.

(c) **Health Checks**: Services provide health check endpoints for availability monitoring.

(d) **Service Metadata**: Registration includes service metadata like version and capabilities.

4.3.4 Service Health Monitoring

The system monitors service health through several approaches:

(a) **Health Check Endpoints**: Each service exposes a `/health` endpoint.

(b) **Circuit Breakers**: Services implement circuit breakers to detect and handle failures.

(c) **Heartbeats**: Services send periodic heartbeats to indicate liveness.

(d) **Logging and Monitoring**: Centralized logging captures service health events.

4.4 API Gateway

4.4.1 Gateway Architecture Using Ocelot

The OllamaNet Gateway is built using the Ocelot API Gateway library, providing a robust and flexible routing solution:

Key components of the Gateway architecture include:

(a) **Ocelot Core**: Handles request routing based on configuration.

(b) **Middleware Pipeline**: Processes requests through a series of middleware components.

(c) **Configuration System**: Manages routing and other gateway settings.

(d) **Authentication Integration**: Validates JWT tokens and enforces security.

4.4.2 Routing Configuration and Management

The Gateway uses a modular configuration approach:

(a) **Service-Specific Configurations**: Each service has its own configuration file.

(b) **Variable Substitution**: Service URLs are defined centrally and referenced via variables.

(c) **Dynamic Reloading**: Configuration changes are detected and applied without restart.

(d) **Aggregation**: Individual configurations are combined at runtime.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/auth/{everything}",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "${Services.Auth.Host}",
          "Port": 443
        }
      ],
      "UpstreamPathTemplate": "/api/auth/{everything}",
      "UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE" ]
    }
  ]
}
```

4.4.3 Authentication and Authorization at Gateway Level

The Gateway handles authentication and authorization through:

(a) **JWT Validation**: Validates tokens issued by the Auth Service.

(b) **Role-Based Authorization**: Enforces access control based on user roles.

(c) **Claims Forwarding**: Extracts user claims and forwards them to downstream services.

(d) **Centralized Policy Enforcement**: Applies consistent security policies across all services.

```
public async Task InvokeAsync(HttpContext context)
{
    if (context.User.Identity?.IsAuthenticated ?? false)
    {
        // Extract claims from the authenticated user
        var userId = context.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        var email = context.User.FindFirst(ClaimTypes.Email)?.Value;
        var roles = context.User.FindAll(ClaimTypes.Role).Select(c => c.Value);

        // Add claims as headers to the request
        if (!string.IsNullOrEmpty(userId))
            context.Request.Headers.Add("X-User-Id", userId);

        if (!string.IsNullOrEmpty(email))
            context.Request.Headers.Add("X-User-Email", email);

        if (roles.Any())
            context.Request.Headers.Add("X-User-Roles", string.Join(",", roles)
    }

    // Continue processing the request
```

```
        await _next(context);
    }
```

4.4.4 Request/Response Transformation

The Gateway performs several transformations on requests and responses:

(a) **URL Rewriting**: Rewrites URLs based on routing configuration.

(b) **Header Manipulation**: Adds, removes, or modifies HTTP headers.

(c) **Response Aggregation**: Combines responses from multiple services when needed.

(d) **Content Transformation**: Transforms request and response content when required.

4.4.5 Cross-cutting Concerns Handled at Gateway

The Gateway handles several cross-cutting concerns:

(a) **CORS**: Configures and enforces Cross-Origin Resource Sharing policies.

(b) **Rate Limiting**: Prevents abuse by limiting request rates.

(c) **Logging**: Logs requests and responses for auditing and troubleshooting.

(d) **Error Handling**: Provides consistent error responses across services.

(e) **Request Tracing**: Adds correlation IDs for request tracking across services.

4.5 Communication Patterns

4.5.1 Synchronous Communication

4.5.1.1 REST API Design and Implementation

OllamaNet services communicate primarily through RESTful APIs:

(a) **Resource-Oriented Design**: APIs are organized around resources.

(b) **Standard HTTP Methods**: GET, POST, PUT, DELETE are used appropriately.

(c) **Status Codes**: Proper HTTP status codes indicate success or failure.

(d) **Content Negotiation**: APIs support multiple content types (primarily JSON).

```
[ApiController]
[Route("api/[controller]")]
public class ConversationsController : ControllerBase
{
    [HttpGet]
    public async Task<ActionResult<IEnumerable<ConversationDto>>> GetConversati
    {
        // Implementation
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<ConversationDto>> GetConversation(Guid id)
    {
        // Implementation
    }

    [HttpPost]
    public async Task<ActionResult<ConversationDto>> CreateConversation(CreateC
    {
        // Implementation
    }

    // Other endpoints
}
```

4.5.1.2 Request-Response Patterns

Services implement several request-response patterns:

Figure 4.6 – Message Broker Communication Patterns

(a) **Synchronous Request-Response**: Client sends a request and waits for a response.

(b) **Streaming Responses**: Used for real-time AI model responses.

(c) **Pagination**: Large result sets are paginated for efficiency.

(d) **Filtering and Sorting**: Clients can specify filters and sort orders.

4.5.1.3 HTTP/HTTPS Communication

All service-to-service communication uses HTTPS with:

(a) **TLS Encryption**: All traffic is encrypted using TLS.

(b) **Certificate Validation**: Services validate certificates for security.

(c) **Connection Pooling**: HTTP clients use connection pooling for efficiency.

(d) **Timeout Management**: Appropriate timeouts prevent resource exhaustion.

4.5.2 Asynchronous Communication

4.5.2.1 Message Broker Usage (RabbitMQ)

OllamaNet uses RabbitMQ for asynchronous communication:

(a) **Topic Exchange**: Messages are routed based on routing keys.

(b) **Durable Messaging**: Messages persist across broker restarts.

(c) **Dead Letter Queues**: Failed messages are sent to dead letter queues for handling.

(d) **Consumer Acknowledgments**: Messages are acknowledged when processed successfully.

4.5.2.2 Event-Driven Communication

The system uses event-driven communication for:

(a) **Service Discovery**: Inference Service publishes URL updates.

(b) **State Changes**: Services publish events when important state changes occur.

(c) **Asynchronous Processing**: Long-running operations use events for completion notification.

(d) **Integration Events**: Cross-service business processes use events for coordination.

4.5.2.3 Message Formats and Standards

Messages follow these standards:

(a) **JSON Format**: All messages use JSON for compatibility.

(b) **Metadata Inclusion**: Messages include metadata like timestamp and version.

(c) **Schema Validation**: Messages are validated against schemas.

(d) **Versioning**: Message formats include version information for compatibility.

```
{
  "newUrl": "https://example-inference-engine.ngrok-free.app/",
  "timestamp": "2023-08-15T14:30:00Z",
  "serviceId": "inference-engine",
  "version": "1.0"
}
```

4.5.2.4 Publishing and Subscribing Mechanisms

The system implements these messaging patterns:

(a) **Publish-Subscribe**: One publisher, multiple subscribers.

(b) **Work Queues**: Tasks distributed among multiple workers.

(c) **RPC**: Request-reply pattern over messaging.

(d) **Broadcast**: Messages sent to all interested parties.

```
public async Task PublishInferenceUrlUpdate(string newUrl)
{
    var message = new InferenceUrlUpdateMessage
```

```
{
    NewUrl = newUrl,
    Timestamp = DateTime.UtcNow
};

var factory = new ConnectionFactory
{
    HostName = _options.HostName,
    Port = _options.Port,
    UserName = _options.UserName,
    Password = _options.Password,
    VirtualHost = _options.VirtualHost
};

using var connection = factory.CreateConnection();
using var channel = connection.CreateModel();

channel.ExchangeDeclare(
    exchange: _options.Exchange,
    type: ExchangeType.Topic,
    durable: true);

var json = JsonSerializer.Serialize(message);
var body = Encoding.UTF8.GetBytes(json);

channel.BasicPublish(
    exchange: _options.Exchange,
    routingKey: _options.InferenceUrlRoutingKey,
    basicProperties: null,
    body: body);
```

```
}
```

4.6 Cross-Cutting Concerns

4.6.1 Authentication & Authorization

4.6.1.1 JWT Implementation

OllamaNet implements JWT-based authentication:

(a) **Token Issuance**: The Auth Service issues JWT tokens upon successful login.

(b) **Token Validation**: The Gateway and services validate tokens before processing requests.

(c) **Claims-Based Identity**: Tokens contain claims about the user's identity and roles.

(d) **Refresh Tokens**: Long-lived refresh tokens enable session persistence.

4.6.1.2 Role-Based Access Control

The system implements role-based access control:

(a) **Role Definitions**: Users are assigned roles (Admin, User, etc.).

(b) **Permission Mapping**: Roles map to permissions for specific operations.

(c) **Gateway Enforcement**: The Gateway enforces role requirements for routes.

(d) **Service-Level Checks**: Services perform additional authorization checks as needed.

```
{
  "RoleAuthorization": [
    {
      "PathTemplate": "/api/admin/*",
      "RequiredRole": "Admin"
    },
    {
      "PathTemplate": "/api/conversations/*",
```

```
      "RequiredRole": "User"
    }
  ]
}
```

4.6.1.3 Claims Forwarding Between Services

User claims are forwarded between services:

(a) **Gateway Extraction**: The Gateway extracts claims from JWT tokens.

(b) **Header Injection**: Claims are added as HTTP headers to downstream requests.

(c) **Service Validation**: Services validate and use the forwarded claims.

(d) **Consistent Identity**: User identity is maintained across service boundaries.

4.6.2 Logging & Monitoring

4.6.2.1 Centralized Logging Approach

OllamaNet implements centralized logging:

(a) **Structured Logging**: All logs use a structured format (JSON).

(b) **Log Levels**: Appropriate log levels (Debug, Info, Warning, Error) are used.

(c) **Correlation IDs**: Requests are tracked across services using correlation IDs.

(d) **Contextual Information**: Logs include relevant context for troubleshooting.

4.6.2.2 Monitoring Strategies

The system is monitored through:

(a) **Health Checks**: Services expose health check endpoints.

(b) **Metrics Collection**: Key performance metrics are collected.

(c) **Alerting**: Alerts are triggered for critical issues.

(d) **Dashboard Visualization**: Metrics are visualized in dashboards.

4.6.2.3 Observability Features

Observability is achieved through:

(a) **Distributed Tracing**: Requests are traced across service boundaries.

(b) **Performance Metrics**: Response times, throughput, and error rates are tracked.

(c) **Resource Utilization**: CPU, memory, and network usage are monitored.

(d) **Business Metrics**: Key business metrics are tracked for insights.

4.6.3 Resilience Patterns

4.6.3.1 Circuit Breaker Patterns

Circuit breakers prevent cascading failures:

(a) **Failure Detection**: Tracks failures in downstream service calls.

(b) **Open Circuit**: Stops calls to failing services after threshold is reached.

(c) **Half-Open State**: Allows test calls to check if service has recovered.

(d) **Closed Circuit**: Normal operation when service is healthy.

```
// Configure HTTP client with circuit breaker
services.AddHttpClient<IInferenceEngineConnector, InferenceEngineConnector>()
    .AddPolicyHandler(GetCircuitBreakerPolicy());

private IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(
            handledEventsAllowedBeforeBreaking: 5,
            durationOfBreak: TimeSpan.FromSeconds(30)
        );
}
```

4.6.3.2 Retry Policies

Retry policies handle transient failures:

(a) **Retry Count**: Specifies how many retries to attempt.

(b) **Backoff Strategy**: Implements exponential backoff between retries.

(c) **Retry Triggers**: Defines which errors trigger retries.

(d) **Timeout**: Sets maximum time for operation with retries.

```
private IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.TooManyReq
        .WaitAndRetryAsync(
            retryCount: 3,
            sleepDurationProvider: retryAttempt => TimeSpan.FromSeconds(Math.Po
        );
}
```

4.6.3.3 Timeout Management

The system manages timeouts at multiple levels:

(a) **Request Timeouts**: HTTP requests have appropriate timeouts.

(b) **Operation Timeouts**: Complex operations have overall timeouts.

(c) **Graceful Degradation**: Services degrade gracefully when timeouts occur.

(d) **User Feedback**: Users are informed about timeouts when appropriate.

4.6.3.4 Fallback Strategies

Fallback strategies provide alternatives when operations fail:

(a) **Cached Data**: Return cached data when live data is unavailable.

(b) **Default Values**: Use sensible defaults when actual values cannot be retrieved.

(c) **Degraded Functionality**: Provide limited functionality rather than complete failure.

(d) **User Notification**: Inform users when fallbacks are used.

Terminology

API Gateway Component that acts as an entry point for client requests to microservices

Circuit Breaker Pattern that prevents cascading failures across services

JWT JSON Web Token used for authentication between services

Microservice Independent deployable service with a specific domain responsibility

RabbitMQ Message broker used for asynchronous communication and service discovery

Chapter 5

Database Layer

5.1 Database Architecture

5.1.1 Overall Database Design Philosophy

The OllamaNet platform employs a robust database architecture that balances the needs of a microservices ecosystem with the benefits of relational data integrity. The database layer is designed around several key principles:

(a) **Shared Database with Logical Separation**: While microservices often employ separate databases, OllamaNet uses a shared SQL Server database with logical separation to maintain data integrity while providing service isolation.

(b) **Repository Pattern**: Data access is abstracted through repositories, providing a consistent interface for services to interact with the database.

(c) **Unit of Work Pattern**: Transactions across multiple repositories are coordinated through a Unit of Work implementation, ensuring data consistency.

(d) **Domain-Driven Design**: The database schema reflects the domain model, with clear entity boundaries and relationships that map to business concepts.

(e) **Soft Delete**: Entities implement soft deletion to preserve historical data while allowing logical removal.

5.1.2 Database Per Service Pattern Consideration

The OllamaNet architecture considered the Database-per-Service pattern, which is common in microservices architectures, but ultimately chose a shared database approach for several reasons:

(a) **Data Integrity**: A shared database ensures referential integrity across services, which is particularly important for user data and relationships.

(b) **Simplified Deployment**: A single database reduces operational complexity compared to managing multiple databases.

(c) **Query Efficiency**: Cross-service queries can be performed more efficiently within a single database.

(d) **Transaction Support**: ACID transactions across related data are easier to implement in a shared database.

However, to maintain service independence, the architecture implements logical separation through:

(a) **Service-Specific Repositories**: Each service accesses only its relevant entities through dedicated repositories.

(b) **Schema Namespacing**: Database tables are organized into logical groups aligned with service boundaries.

(c) **Access Control**: Services are restricted to their own data domains through repository interfaces.

5.1.3 Shared Database Implementation

The shared database implementation uses Entity Framework Core as an ORM, with the following components:

(a) **Central DbContext**: A single `MyDbContext` class that extends `IdentityDbContext<Applicati` and includes all entity DbSets.

(b) **Entity Configuration**: Entity relationships and constraints are configured in the `OnModelCreating` method of the DbContext.

(c) **Service-Specific Repositories**: Each service has dedicated repositories that access only the entities relevant to that service.

(d) **Unit of Work Coordinator**: A central `UnitOfWork` class coordinates operations across repositories and manages transactions.

```
public class MyDbContext : IdentityDbContext<ApplicationUser>
{
    public MyDbContext(DbContextOptions<MyDbContext> options) : base(options) {

    public DbSet<AIModel> AIModels { get; set; }
    public DbSet<AIResponse> AIResponses { get; set; }
    public DbSet<Attachment> Attachments { get; set; }
    public DbSet<Conversation> Conversations { get; set; }
    public DbSet<ConversationPromptResponse> ConversationPromptResponses { get;
    public DbSet<Feedback> Feedbacks { get; set; }
    public DbSet<ModelTag> ModelTags { get; set; }
    public DbSet<Prompt> Prompts { get; set; }
    public DbSet<RefreshToken> RefreshTokens { get; set; }
    public DbSet<SystemMessage> SystemMessages { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Configure ModelTag as a join entity
        modelBuilder.Entity<ModelTag>().HasKey(mt => new { mt.ModelId, mt.TagId

        // Other entity configurations...
    }
}
```

5.1.4 Physical vs. Logical Database Separation

OllamaNet employs logical separation within a physically shared database:

(a) **Physical Sharing**: All services access the same SQL Server instance and database.

(b) **Logical Separation**:

- Each service accesses only its domain-specific entities
- Repository interfaces expose only relevant operations
- Service boundaries are enforced at the application level

This approach provides several benefits:

- **Data Consistency**: Ensures consistent data across services.
- **Simplified Transactions**: Enables ACID transactions across related entities.
- **Efficient Queries**: Allows for efficient joins between related entities.
- **Reduced Operational Complexity**: Simplifies database management and deployment.

5.1.5 Design Principles and Patterns

The database layer implements several key design principles and patterns:

Repository Pattern

Abstracts data access logic and provides a consistent interface for working with entities.

```
public interface IRepository<T> where T : class
{
    Task<IEnumerable<T>> GetAllAsync();
    Task<T> GetByIdAsync(string id);
    Task<bool> AddAsync(T entity);
    Task<bool> UpdateAsync(T entity);
    Task<bool> DeleteAsync(string id);
    Task<bool> SoftDeleteAsync(string id);
```

```
    Task<bool> ExistsAsync(string id);
}
```

Unit of Work Pattern

Coordinates operations across multiple repositories and provides a single point for committing changes.

```
public interface IUnitOfWork : IDisposable
{
    IAIModelRepository AIModels { get; }
    IAIResponseRepository AIResponses { get; }
    // Other repositories...

    Task<int> SaveChangesAsync();
}
```

Identity Integration

Extends ASP.NET Identity with custom user properties and relationships.

Soft Delete Pattern

Implements logical deletion through an `IsDeleted` flag on entities.

Query Specification Pattern

Encapsulates query logic for complex filtering and sorting.

Eager Loading Strategy

Uses explicit loading of related entities to avoid N+1 query problems.

5.2 Data Models

5.2.1 Entity Relationship Diagrams

The OllamaNet database schema includes several interconnected entities that support the platform's functionality:

5.2.2 Schema Designs

The database schema is organized around several key domains:

(a) **User Management Domain**:

- ApplicationUser (extends IdentityUser)
- RefreshToken

(b) **Model Management Domain**:

- AIModel
- Tag
- ModelTag (junction entity)

(c) **Conversation Domain**:

- Conversation
- ConversationPromptResponse
- Prompt
- AIResponse
- Attachment
- Feedback

(d) **System Configuration Domain**:

- SystemMessage

Each entity includes standard fields for tracking creation, modification, and deletion:

```
// Common fields found in most entities
```

```
public string Id { get; set; }
public DateTime CreatedAt { get; set; }
public bool IsDeleted { get; set; }
```

5.2.3 Domain Model to Database Mapping

The OllamaNet platform maps its domain model to the database schema using Entity Framework Core's fluent API and data annotations:

(a) **Entity Configuration**:

- Primary keys defined using the `[Key]` attribute or fluent API
- Foreign keys established through the fluent API
- Navigation properties configured for relationships

(b) **Relationship Mapping**:

- One-to-Many: Defined through navigation properties and foreign keys
- Many-to-Many: Implemented using junction entities (e.g., ModelTag)
- One-to-One: Configured with unique foreign keys

(c) **Property Mapping**:

- Data types specified through attributes or fluent API
- Required fields marked with the `[Required]` attribute
- String length constraints applied where appropriate

Example of entity configuration in the DbContext:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Configure ModelTag as a join entity
    modelBuilder.Entity<ModelTag>().HasKey(mt => new { mt.ModelId, mt.TagId });
```

```
        modelBuilder.Entity<ModelTag>()
            .HasOne(mt => mt.Model)
            .WithMany(m => m.ModelTags)
            .HasForeignKey(mt => mt.ModelId);

        modelBuilder.Entity<ModelTag>()
            .HasOne(mt => mt.Tag)
            .WithMany(t => t.ModelTags)
            .HasForeignKey(mt => mt.TagId);
    }
```

5.2.4 Database Constraints and Validations

The OllamaNet database implements several types of constraints and validations:

(a) **Primary Key Constraints**: Each entity has a unique identifier, typically a string containing a GUID.

(b) **Foreign Key Constraints**: Relationships between entities are enforced through foreign keys.

(c) **Required Field Constraints**: Critical fields are marked as required to prevent null values.

(d) **String Length Constraints**: String fields have appropriate length constraints.

(e) **Unique Constraints**: Applied to fields that must be unique, such as usernames and email addresses.

(f) **Default Values**: Some fields have default values, such as `IsDeleted = false` and `CreatedAt = DateTime.UtcNow`.

(g) **Check Constraints**: Used to enforce business rules, such as valid status values.

These constraints are implemented through a combination of Entity Framework Core configurations and database-level constraints.

5.2.5 Soft Delete Implementation

OllamaNet implements soft deletion across its entities to preserve historical data while allowing logical removal:

(a) **IsDeleted Flag**: Each entity includes an `IsDeleted` boolean property.

(b) **Repository Filtering**: Repositories automatically filter out entities where `IsDeleted = true`.

```
public async Task<IEnumerable<Tag>> GetAllAsync()
{
    return await _context.Tags
        .Where(t => !t.IsDeleted)
        .ToListAsync();
}
```

(c) **Soft Delete Operation**: Instead of physically removing entities, the `SoftDeleteAsync` method sets `IsDeleted = true`.

```
public async Task<bool> SoftDeleteAsync(string id)
{
    var tag = await _context.Tags.FindAsync(id);
    if (tag == null)
        return false;

    tag.IsDeleted = true;
    _context.Tags.Update(tag);
    return true;
}
```

(d) **Query Extensions**: Extension methods automatically apply soft delete filtering to queries.

This approach provides several benefits:

- Preserves historical data for auditing and analysis

- Allows for data recovery if needed
- Maintains referential integrity across related entities
- Simplifies compliance with data retention requirements

5.3 Data Consistency Strategies

5.3.1 Eventual Consistency Approaches

While OllamaNet primarily uses a shared database with strong consistency, it also implements eventual consistency patterns for specific scenarios:

(a) **Cache Synchronization**: Redis caching is used with appropriate invalidation strategies to ensure eventual consistency between the cache and the database.

(b) **Read-Your-Writes Consistency**: The system ensures that after a write operation, subsequent reads will reflect the changes, even when caching is involved.

(c) **Background Processing**: Some operations are performed asynchronously, with eventual consistency guaranteed through reliable message processing.

(d) **Optimistic Concurrency**: Entity Framework's optimistic concurrency control is used to detect and resolve conflicts when multiple services update the same data.

5.3.2 Saga Pattern Implementation

For complex operations that span multiple services, OllamaNet implements a simplified version of the Saga pattern:

(a) **Coordinated Transactions**: The Unit of Work pattern coordinates transactions within a single service.

(b) **Compensating Transactions**: For cross-service operations, compensating transactions are implemented to roll back changes if a step fails.

(c) **Event-Driven Coordination**: Services publish events to signal completion of their part of a distributed transaction.

This approach helps maintain data consistency across service boundaries while avoiding distributed transactions.

5.3.3 Distributed Transactions Handling

OllamaNet avoids distributed transactions where possible, but implements several strategies for maintaining consistency across services:

(a) **Service Composition**: Complex operations are composed at the API level rather than using distributed transactions.

(b) **Event-Driven Updates**: Services subscribe to events from other services to update their data accordingly.

(c) **Idempotent Operations**: APIs are designed to be idempotent, allowing safe retries of failed operations.

(d) **Consistency Verification**: Background processes verify and reconcile data consistency across services.

5.3.4 Concurrency Control Mechanisms

The database layer implements several concurrency control mechanisms:

(a) **Optimistic Concurrency**: Entity Framework's optimistic concurrency control is used to detect conflicts during updates.

```
modelBuilder.Entity<AIModel>()
    .Property(p => p.RowVersion)
    .IsRowVersion();
```

(b) **Pessimistic Locking**: For critical operations, explicit database locks are used to prevent concurrent modifications.

(c) **Transaction Isolation Levels**: Appropriate transaction isolation levels are set based on the operation's requirements.

```
using var transaction = await _context.Database.BeginTransactionAsync(Isolat
try
```

```
{
    // Perform operations
    await _context.SaveChangesAsync();
    await transaction.CommitAsync();
}
catch
{
    await transaction.RollbackAsync();
    throw;
}
```

5.3.5 Error Handling and Rollback Strategies

The database layer implements robust error handling and rollback strategies:

(a) **Transaction Scope**: Operations that modify multiple entities are wrapped in transactions to ensure atomicity.

```
public async Task<bool> CreateConversationWithPromptAsync(Conversation conve
{
    // Add entities
    await _unitOfWork.Conversations.AddAsync(conversation);
    await _unitOfWork.Prompts.AddAsync(prompt);
    await _unitOfWork.AIResponses.AddAsync(response);

    // Create relationship
    var cpr = new ConversationPromptResponse
    {
        Id = Guid.NewGuid().ToString(),
        ConversationId = conversation.Id,
        PromptId = prompt.Id,
        AIResponseId = response.Id,
```

```
            CreatedAt = DateTime.UtcNow
        };

        await _unitOfWork.ConversationPromptResponses.AddAsync(cpr);

        // Save all changes in a single transaction
        await _unitOfWork.SaveChangesAsync();

        return true;
    }
```

(b) **Exception Handling**: Exceptions during database operations are caught and handled appropriately, with transactions rolled back when necessary.

(c) **Retry Logic**: Transient errors are handled with retry logic to improve resilience.

(d) **Logging**: Database errors are logged with sufficient context for troubleshooting.

5.4 Database Technologies

5.4.1 SQL Server Implementation Details

OllamaNet uses SQL Server as its primary database technology:

(a) **Version**: SQL Server 2019 or later, supporting modern features like JSON support and improved performance.

(b) **Connection Management**: Connection pooling is configured for optimal performance and resource utilization.

(c) **Indexing Strategy**: Appropriate indexes are created for frequently queried fields to optimize performance.

(d) **Query Optimization**: Complex queries are optimized using query hints and execution plan analysis.

(e) **Security Configuration**: SQL Server is configured with appropriate security settings, including encryption and access controls.

5.4.2 Entity Framework Core Configuration

Entity Framework Core is configured for optimal performance and functionality:

(a) **DbContext Configuration**: The `MyDbContext` is configured with appropriate options for tracking, batching, and logging.

```
services.AddDbContext<MyDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnectio
        sqlOptions =>
        {
            sqlOptions.EnableRetryOnFailure(
                maxRetryCount: 5,
                maxRetryDelay: TimeSpan.FromSeconds(30),
                errorNumbersToAdd: null);
        })
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking)
    .EnableSensitiveDataLogging(isDevelopment));
```

(b) **Entity Configuration**: Entities are configured using a combination of data annotations and the fluent API.

(c) **Lazy Loading**: Lazy loading is disabled by default to prevent N+1 query problems, with explicit loading used when needed.

(d) **Query Filters**: Global query filters are applied for soft delete and multi-tenancy.

```
modelBuilder.Entity<Conversation>()
    .HasQueryFilter(c => !c.IsDeleted);
```

(e) **Performance Optimization**: Query compilation is cached, and query execution is optimized through appropriate includes and projections.

5.4.3 Redis Caching Implementation

OllamaNet uses Redis for distributed caching to improve performance:

(a) **Cache Architecture**: A multi-level caching approach with in-memory and Redis caches.

(b) **Cache-Aside Pattern**: Data is retrieved from the cache first, with database fallback when needed.

```
public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, Tim
{
    try
    {
        var cachedValue = await _redisCacheService.GetAsync<T>(key);
        if (cachedValue != null)
        {
            _logger.LogDebug("Cache hit for key: {Key}", key);
            return cachedValue;
        }

        _logger.LogDebug("Cache miss for key: {Key}", key);
        var result = await factory();

        if (result != null)
        {
            await _redisCacheService.SetAsync(key, result, expiry ?? _defaul
        }

        return result;
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "Cache operation failed for key: {Key}, falli
        return await factory();
    }
```

```
}
```

(c) **Cache Invalidation**: When data changes, related cache entries are invalidated to maintain consistency.

(d) **Cache Resilience**: The system gracefully handles Redis unavailability by falling back to the database.

(e) **Cache Optimization**: Frequently accessed data is cached with appropriate expiration policies.

5.4.4 Query Optimization Techniques

OllamaNet implements several query optimization techniques:

(a) **Indexing Strategy**: Appropriate indexes are created for frequently queried fields.

(b) **Query Projection**: Queries project only the required fields rather than loading entire entities.

```
public async Task<IEnumerable<ModelDto>> GetAllModelsAsync()
{
    return await _context.AIModels
        .Where(m => !m.IsDeleted)
        .Select(m => new ModelDto
        {
            Id = m.Id,
            Name = m.Name,
            Description = m.Description,
            // Only select needed properties
        })
        .ToListAsync();
}
```

(c) **Eager Loading**: Related entities are loaded using explicit includes to avoid N+1 query problems.

```
public async Task<Conversation> GetConversationWithDetailsAsync(string id)
{
    return await _context.Conversations
        .Include(c => c.ConversationPromptResponses)
            .ThenInclude(cpr => cpr.Prompt)
        .Include(c => c.ConversationPromptResponses)
            .ThenInclude(cpr => cpr.AIResponse)
        .FirstOrDefaultAsync(c => c.Id == id && !c.IsDeleted);
}
```

(d) **Pagination**: Large result sets are paginated to improve performance.

```
public async Task<IEnumerable<Conversation>> GetConversationsPagedAsync(
    string userId, int pageNumber, int pageSize)
{
    return await _context.Conversations
        .Where(c => c.UserId == userId && !c.IsDeleted)
        .OrderByDescending(c => c.CreatedAt)
        .Skip((pageNumber - 1) * pageSize)
        .Take(pageSize)
        .ToListAsync();
}
```

(e) **Query Caching**: Frequently executed queries are cached to reduce database load.

5.4.5 Connection Management

OllamaNet implements effective connection management strategies:

(a) **Connection Pooling**: Entity Framework Core's connection pooling is configured for optimal performance.

(b) **Connection Resilience**: Retry logic is implemented for transient connection failures.

(c) **Connection Monitoring**: Connection usage is monitored to detect leaks and performance issues.

(d) **Connection Timeout**: Appropriate connection timeouts are configured to prevent resource exhaustion.

(e) **Connection Security**: Connections use encrypted communication and minimal privilege accounts.

5.5 Data Migration and Versioning

5.5.1 Migration Strategy

OllamaNet uses Entity Framework Core migrations for database schema evolution:

(a) **Migration Generation**: Migrations are generated using the EF Core Tools when entity models change.

```
dotnet ef migrations add AddConversationStatus
```

(b) **Migration Application**: Migrations are applied during application startup or through deployment scripts.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // Apply migrations at startup
    using (var scope = app.ApplicationServices.CreateScope())
    {
        var dbContext = scope.ServiceProvider.GetRequiredService<MyDbContext
        dbContext.Database.Migrate();
    }

    // Other configuration...
}
```

(c) **Migration Testing**: Migrations are tested in development and staging environments before production deployment.

(d) **Rollback Strategy**: Each migration has a corresponding down method for rollback if needed.

5.5.2 Schema Evolution Approach

OllamaNet follows a careful approach to schema evolution:

(a) **Additive Changes**: Prefer adding new tables or columns rather than modifying existing ones.

(b) **Backward Compatibility**: Maintain backward compatibility with existing code when possible.

(c) **Phased Deployment**: Complex schema changes are deployed in phases to minimize risk.

(d) **Data Migration**: Include data migration logic when schema changes affect existing data.

```
migrationBuilder.Sql(@"
    UPDATE Conversations
    SET Status = 'Active'
    WHERE Status IS NULL
");
```

5.5.3 Backward Compatibility Considerations

The database layer maintains backward compatibility through several strategies:

(a) **Default Values**: New columns have sensible default values to support existing code.

(b) **Nullable Columns**: New columns are added as nullable when appropriate.

(c) **View Compatibility**: Database views are used to maintain compatibility with legacy queries.

(d) **API Versioning**: API versions are maintained to support clients using older data models.

5.5.4 Deployment Practices for Database Changes

OllamaNet follows these best practices for database deployments:

(a) **Automated Migrations**: Database migrations are part of the automated deployment pipeline.

(b) **Validation Scripts**: Pre-deployment validation scripts verify that migrations can be applied safely.

(c) **Backup Strategy**: Database backups are created before applying migrations.

(d) **Deployment Windows**: Database changes are deployed during low-traffic periods.

(e) **Monitoring**: Database performance is monitored during and after migration application.

5.5.5 Database Versioning Approach

The database schema is versioned using several mechanisms:

(a) **Migration History**: EF Core's migration history table tracks applied migrations.

(b) **Semantic Versioning**: Database schema versions follow semantic versioning principles.

(c) **Documentation**: Schema changes are documented with each migration.

(d) **Version Compatibility**: The application verifies database schema compatibility at startup.

```
public void EnsureDatabaseCompatibility(MyDbContext context)
{
    var pendingMigrations = context.Database.GetPendingMigrations();
    if (pendingMigrations.Any())
    {
        throw new Exception($"Database is out of date. {pendingMigrations.Co
    }
}
```

Terminology

Repository Pattern Design pattern that mediates between the domain model and data source

Unit of Work Pattern that maintains a list of objects affected by a business transaction

Entity Framework Core Object-relational mapper used for database access

Soft Delete Pattern for marking records as deleted without physically removing them

Appendix A

An Example Appendix

As an appendix, this should contain some content that's not really required for the argument in the main body of the thesis, but is clearly relevant and supports the work.

A.1 Code Listings

The `listings` package allows you to include code listings or other formatted text with some parsing to make them more readable than simply calling `\input{}` on the code file.

Listing A.1 – MATLAB script for interactive radians to degrees converter

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
```

```
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train␣Epoch:␣{}␣[{}/{}␣({:.0f}%)]\tLoss:␣{:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break


def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
```

```
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))


def main():
    # Training settings
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=64, metavar='N',
                        help='input batch size for training (default: 64)')
    parser.add_argument('--test-batch-size', type=int, default=1000, metavar='N',
                        help='input batch size for testing (default: 1000)')
    parser.add_argument('--epochs', type=int, default=14, metavar='N',
                        help='number of epochs to train (default: 14)')
    parser.add_argument('--lr', type=float, default=1.0, metavar='LR',
                        help='learning rate (default: 1.0)')
    parser.add_argument('--gamma', type=float, default=0.7, metavar='M',
                        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--no-cuda', action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--no-mps', action='store_true', default=False,
                        help='disables macOS GPU training')
    parser.add_argument('--dry-run', action='store_true', default=False,
                        help='quickly check a single pass')
    parser.add_argument('--seed', type=int, default=1, metavar='S',
                        help='random seed (default: 1)')
    parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                        help='how many batches to wait before logging training status')
    parser.add_argument('--save-model', action='store_true', default=False,
                        help='For Saving the current Model')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()
    use_mps = not args.no_mps and torch.backends.mps.is_available()

    torch.manual_seed(args.seed)

    if use_cuda:
        device = torch.device("cuda")
    elif use_mps:
        device = torch.device("mps")
```

```
    else:
        device = torch.device("cpu")

    train_kwargs = {'batch_size': args.batch_size}
    test_kwargs = {'batch_size': args.test_batch_size}
```

A number of languages are supported with basic syntax highlighting and formatting.

A.2 Multi-Page Tables

The `supertabular` package allows tables to span multiple pages using the `supertabular` environment (in place of `tabular`). This has already been used in the Nomenclature Section in the front matter, allowing the notation to span multiple pages if necessary. Table A.1 shows an example of a table spanning two pages. Note that such tables are no longer floating elements (i.e. there's no `table` environment anymore), and the header/footer for the whole table, and ones repeated on each new page, can be defined through `supertabular` macros rather than as part of the table to copy headers across each page.

Table A.1 – This table is especially long, so it's been turned into a `supertabular` environment allowing it to span multiple pages.

first × **second** = **RHS**
$1 \times 1 = 1$
$1 \times 2 = 2$
$1 \times 3 = 3$
$1 \times 4 = 4$
$1 \times 5 = 5$
$1 \times 6 = 6$
$1 \times 7 = 7$
$1 \times 8 = 8$
$2 \times 1 = 2$

continued on next page

continued from previous page

first	×	**second**	=	**RHS**
2	×	2	=	4
2	×	3	=	6
2	×	4	=	8
2	×	5	=	10
2	×	6	=	12
2	×	7	=	14
2	×	8	=	16
3	×	1	=	3
3	×	2	=	6
3	×	3	=	9
3	×	4	=	12
3	×	5	=	15
3	×	6	=	18
3	×	7	=	21
3	×	8	=	24

A.3 Landscape Tables

If your table is especially wide, it may be better to switch it to the landscape orientation. One way of doing this is with the `rotating` package, which implements (among other things) two new environments: `sidewaystable` and `sidewaysfigure`[1]. The way this package achieves this is most useful for *printed results*, as it only rotates the environment on the page (but does not convert the page into landscape orientation)—for electronic viewing of a PDF, it may be useful to rotate the whole page since it's not often easy for the reader to rotate their screen (assuming the sideways content takes up the whole page). One advantage of this package's implementation of sideways environments is that it supports `twoside` page layout, and will rotate the sideways environment

[1] I find `sidewaysfigure` less useful, as it tends to be easy enough to rotate the figure before inclusion, but if the caption/figure are complex it may be useful to have them oriented in the same way

such that the bottom is towards the outside of the double-page layout in such cases.

An example of a `sidewaystable` is shown in Table A.2—if you're reading this as a PDF on your computer, you'll probably find it difficult to read as it's sideways on your screen.

Table A.2 – This table is so wide that I decided it should be in the landscape orientation to allow it to fit nicely on one page. You may of course find it easier (for the reader) to reconsider the content and layout of the table, or convert it to a graphical representation, as large walls of data tend to be hard to really interpret well. Almost certainly, you'd only have such large tables in an appendix.

Item	Total	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
SA Mission Distance (m)	1101.4	49.4	81.5	34.2	78.8	98.8	70.8	16.0	61.4	14.9	52.1	24.3	83.3	170.3	143.5	20.7	30.1	21.4	99.2
SA Traversed Distance (m)	3244.1	53.9	86.8	90.7	92.1	120.8	74.3	46.4	63.8	15.6	55.3	27.4	127.2	222.4	987.1	273.0	167.0	235.4	505.0
MA Mission Distance (m)	1083.9	59.1	81.5	34.2	78.8	98.8	70.8	16.0	61.4	14.9	52.1	24.3	83.3	170.3	143.5	20.7	30.1	21.4	81.8
MA Traversed Distance (m)	2343.1	61.8	84.6	70.7	84.6	116.6	72.5	46.3	62.6	15.5	53.5	25.8	129.4	213.4	147.1	268.2	174.5	233.8	482.3
Ratio (SA/MA)	1.38	0.872	1.03	1.28	1.09	1.04	1.03	1	1.02	1.01	1.03	1.06	0.983	1.04	6.71	1.02	0.957	1.01	1.05
SA Mission Est. Time (s)	734.3	32.9	54.4	22.8	52.5	65.9	47.2	10.7	40.9	9.9	34.8	16.2	55.5	113.5	95.7	13.8	20.1	14.3	66.2
SA Traversal Time (s)	2436.0	43.5	61.6	70.1	68.0	89.6	55.3	35.2	45.3	12.7	39.8	21.8	93.1	161.8	731.0	204.7	146.1	174.3	382.1
MA Mission Est. Time (s)	1083.9	59.1	81.5	34.2	78.8	98.8	70.8	16.0	61.4	14.9	52.1	24.3	83.3	170.3	143.5	20.7	30.1	21.4	81.8
MA Traversal Time (s)	2411.3	64.2	84.8	73.6	86.8	119.1	74.2	47.2	63.5	16.1	54.3	28.1	132.8	215.2	148.2	271.3	203.4	239.9	488.7
Ratio (SA/MA)	1.01	0.677	0.726	0.953	0.784	0.753	0.746	0.744	0.714	0.786	0.734	0.776	0.701	0.752	4.93	0.755	0.718	0.727	0.782
SA Cost (Exp. Map)	1019924.1	1540.6	49041.0	5094.7	86202.5	98847.3	0.0	7974.8	1773.9	459.0	7825.8	1120.0	4131.3	14618.2	447306.7	19380.5	109612.1	33188.1	131807.7
MA Cost (Exp. Map)	916661.5	6032.4	81939.1	7722.2	73856.9	198949.4	11654.8	6191.5	1398.9	564.8	13336.9	1123.8	6129.8	68857.2	39422.7	3213.2	172319.0	93562.5	130386.5
Ratio (SA/MA)	1.11	0.255	0.599	0.66	1.17	0.497	0	1.29	1.27	0.813	0.587	0.997	0.674	0.212	11.3	6.03	0.636	0.355	1.01
SA Cost (Ground Truth)	26891.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	26891.4	0.0	0.0
MA Cost (Ground Truth)	28400.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	28400.0	0.0	0.0
Ratio (SA/MA)	0.947	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0.947	–	–

Map Configuration	SA Coverage (m^2)	MA Coverage (m^2)	Ratio (MA/SA)
Expanded Cost Map	28108	28229	1