OllamaNet System Architecture Patterns

Overview

OllamaNet is a comprehensive microservices platform for integrating Ollama's large language model capabilities into applications through a clean, modular architecture. The platform consists of five core microservices:

1. **Gateway Service**: API gateway using Ocelot for unified service access
2. **AuthService**: Authentication and user identity management
3. **AdminService**: Administrative functions and platform management
4. **ExploreService**: Model discovery and exploration
5. **ConversationService**: Conversation management and real-time chat

These services are designed with consistent architectural patterns, shared components, and well-defined integration points to create a cohesive ecosystem.

Architectural Approach

Microservices Architecture

The platform follows a microservices architecture with:

- **Service Segregation**: Each service has a specific domain responsibility
- **Independent Deployment**: Services can be deployed and scaled independently
- **API Gateway**: Unified entry point through Gateway service
- **Shared Database**: Common Ollama_DB_layer with service-specific repositories
- **Cross-Service Communication**: REST API calls through Gateway

Clean Layered Architecture

All services implement a consistent layered architecture:

```
┌─────────────────────┐
│    API Layer    │ Controllers handling HTTP requests
├─────────────────────┤
│  Service Layer  │ Business logic implementation
├─────────────────────┤
│    Data Layer   │ Repository pattern for data access
├─────────────────────┤
│ Infrastructure  │ Cross-cutting concerns
└─────────────────────┘
```

- **API Layer**: Controllers that handle HTTP requests/responses
- **Service Layer**: Domain-specific services with business logic
- **Data Layer**: Repository interfaces and implementations
- **Infrastructure Layer**: Cross-cutting concerns (caching, logging, etc.)

Domain-Driven Design

The platform employs domain-driven design principles:

- **Domain Models**: Entity definitions representing business concepts
- **Domain Services**: Logic grouped by domain (User, AIModel, Tag, etc.)
- **Domain Events**: State changes communicated via events
- **Bounded Contexts**: Clear domain boundaries between services

Common Design Patterns

Repository Pattern

All services abstract data access through repositories:

- **IUnitOfWork**: Coordination of multiple repositories
- **Generic Repositories**: Reusable CRUD operations
- **Domain-Specific Repositories**: Specialized data access for entities
- **Query Specifications**: Encapsulated query logic

Dependency Injection

Extensive use of dependency injection across all services:

- **Service Registration**: Via extension methods in ServiceExtensions.cs
- **Constructor Injection**: Dependencies provided via constructors
- **Interface-Based Design**: Programming to abstractions
- **Scoped Lifetimes**: Appropriate service lifetimes (transient, scoped, singleton)

Options Pattern

Configuration management through strongly-typed options:

- **IOptions**: For strongly-typed configuration access
- **Configuration Sections**: Clear organization in appsettings.json
- **Environment-Specific Settings**: Development vs. production configurations
- **No Hard-Coded Values**: All configuration externalized

Caching Strategy

Sophisticated Redis-based caching implemented across services:

- **Two-Tier Architecture**: High-level CacheManager with low-level RedisCacheService
- **Cache-Aside Pattern**: GetOrSetAsync methods with database fallback
- **Domain-Specific TTLs**: Tailored expiration times by data type
- **Cache Invalidation**: Strategic invalidation on data changes
- **Resilience Patterns**: Circuit breakers and retry logic for cache failures

Exception Handling

Consistent exception management approach:

- **Domain-Specific Exceptions**: Custom exception hierarchies
- **Global Exception Handling**: Centralized error processing
- **HTTP Status Mapping**: Exception types mapped to appropriate HTTP codes
- **Structured Error Responses**: Consistent error formats
- **Error Logging**: Comprehensive error details for troubleshooting

Service-Specific Patterns

Gateway Service

Configuration-as-Code

- **Service-Specific Files**: Configuration split by service domain (Auth.json, Admin.json, etc.)
- **Variable Substitution**: Service URLs defined in ServiceUrls.json and referenced with ${variable} syntax
- **Dynamic Reloading**: File watching for configuration changes
- **Unified Configuration**: Multiple files combined into single Ocelot configuration

Request Routing

- **Consistent URL Schema**: Predictable routing patterns
- **Claims Forwarding**: User claims passed to downstream services
- **Authentication Middleware**: JWT validation at gateway level
- **Role-Based Authorization**: Access control based on user roles

AuthService

Security Patterns

- **JWT Authentication**: Token-based security with comprehensive validation
- **Refresh Token Rotation**: One-time use pattern with revocation
- **Secure Cookie Handling**: HTTP-only, secure cookies for refresh tokens
- **ASP.NET Identity**: User and role management framework
- **Password Policy Enforcement**: Secure password requirements

Token Management

- **JWT Generation**: Secure token creation with appropriate claims
- **Token Validation**: Comprehensive validation parameters
- **Token Refresh**: Mechanism for extending sessions
- **Token Revocation**: Ability to invalidate tokens

AdminService

Domain Organization

- **User Domain**: User management services and DTOs
- **AIModel Domain**: Model administration services
- **Tag Domain**: Categorization services
- **Inference Domain**: Model deployment services

- **Domain-Specific Validators**: Input validation by domain

Streaming Capability

- **Server-Sent Events**: For model installation progress
- **IAsyncEnumerable**: Asynchronous stream of progress updates
- **Progress Reporting**: IProgress for status updates
- **Real-Time Feedback**: Immediate installation status

ExploreService

Resilient Caching

- **Circuit Breaker Pattern**: Graceful fallback for cache failures
- **Retry Pattern**: Exponential backoff for transient failures
- **Timeout Handling**: Configurable timeouts for operations
- **Exception Conversion**: Infrastructure to domain exception mapping

Performance Optimization

- **Efficient Data Retrieval**: Optimized database queries
- **Strategic Caching**: Data cached based on access patterns
- **Performance Monitoring**: Stopwatch for critical operations
- **Resource Management**: Careful management of connections and resources

ConversationService

RAG System Architecture

- **Clean Separation**: Infrastructure and service layers
- **Vector Database Integration**: Pinecone for semantic search
- **Document Processing**: Multi-format support via specialized processors
- **Context Retrieval**: Semantic search for conversation enhancement

Real-Time Chat

- **Streaming Responses**: Server-sent events for live responses
- **Message History**: Persistent conversation storage
- **Conversation Organization**: Folder structure for organization
- **Background Processing**: Post-streaming conversation saving

Service Discovery

- **Dynamic Configuration**: Runtime URL updates via RabbitMQ
- **Resilience Patterns**: Polly-based retry and circuit breaker
- **Message-Based Updates**: Topic exchange for configuration changes
- **Persistent Configuration**: Redis for configuration storage

Cross-Cutting Concerns

Authentication & Authorization

- **JWT-Based Authentication**: Consistent token validation
- **Role-Based Authorization**: Admin and User roles
- **Claims-Based Identity**: User identification from claims
- **Token Security**: Proper signing and validation

Caching Infrastructure

- **Redis Cache**: Distributed caching via Upstash
- **Cache Key Management**: Consistent key generation
- **TTL Strategies**: Domain-specific expiration times
- **Cache Failure Handling**: Graceful degradation on failures

Logging & Monitoring

- **Structured Logging**: Consistent log format across services
- **Performance Metrics**: Timing information for critical operations
- **Error Tracking**: Comprehensive error details
- **Context Enrichment**: Request correlation and user information

Validation Framework

- **FluentValidation**: Consistent validation framework
- **Domain-Specific Validators**: Organized by domain
- **Conditional Validation**: Context-aware validation rules
- **Validation Response**: Standardized error feedback

Configuration Management

- **Environment-Specific Settings**: Development vs. production
- **Strongly-Typed Options**: IOptions pattern
- **Sensitive Data Handling**: Secure storage considerations
- **Service Registration Extensions**: Consistent registration patterns

Integration Patterns

Service Communication

- **REST API**: Primary communication mechanism
- **Gateway Routing**: All requests through Gateway service
- **Authentication Propagation**: JWT tokens for identity
- **Error Handling**: Consistent error responses

Shared Database

- **Common Schema**: Shared database design
- **Service-Specific Repositories**: Data access abstraction
- **Connection Management**: Efficient database connections
- **Transaction Handling**: Atomic operations with UnitOfWork

Redis Integration

- **Distributed Caching**: Shared Redis instance
- **Performance Optimization**: Reduced database load
- **Consistent Configuration**: Similar setup across services
- **Connection Resilience**: Handling Redis unavailability

Message-Based Communication

- **RabbitMQ**: For service discovery and configuration updates
- **Topic Exchange**: Topic-based message routing
- **Durable Queues**: Message persistence
- **Consumer Design**: Background service consumers

Data Consistency

- **Unit of Work**: Atomic database operations
- **Cache Synchronization**: Cache invalidation on changes
- **Input Validation**: Data integrity through validation
- **Domain-Specific Logic**: Business rules in domain services
- **Exception Handling**: Proper transaction management

Security Architecture

- **JWT Authentication**: Secure token handling
- **Role-Based Access**: Appropriate authorization
- **Input Validation**: Against injection attacks
- **HTTPS Enforcement**: Secure communication
- **Secure Configuration**: Protection of sensitive data
- **Document Security**: Content validation and access control

Extensibility Points

- **Middleware Pipeline**: Custom middleware integration
- **Service Registration**: Extension methods for services
- **Configuration Providers**: Customizable configuration sources
- **Validation Rules**: Custom validation logic
- **Exception Handlers**: Specialized error processing

Performance Considerations

- **Caching Strategy**: Strategic data caching
- **Query Optimization**: Efficient database access
- **Connection Pooling**: Database connection reuse
- **Asynchronous Operations**: Non-blocking I/O
- **Background Processing**: Offloading long-running tasks
- **Streaming Implementation**: Efficient data streaming