Lumina AI: Comprehensive Technical Documentation

Table of Contents

System Overview

Lumina AI is a comprehensive, enterprise-grade artificial intelligence platform designed to provide powerful agentic capabilities with a focus on quality, robustness, and ethical considerations. The system integrates multiple AI providers, offers sophisticated collaboration mechanisms, and provides a robust workflow orchestration engine, all while maintaining strong governance controls.

The platform is designed to solve complex problems autonomously with minimal human intervention, while ensuring transparency, privacy, and safety. Lumina AI can operate across various domains and use cases, from content generation and data analysis to complex decision-making and process automation.

Design Philosophy

Lumina AI was built with the following core principles:

1. **Quality First**: Prioritizing high-quality outputs and reliable performance
2. **Robustness**: Ensuring system stability even under unexpected conditions
3. **Agentic Capabilities**: Enabling autonomous problem-solving and task completion
4. **Ethical Considerations**: Embedding governance and ethical guidelines at the core
5. **Extensibility**: Designing for easy integration of new capabilities and providers

Target Use Cases

- Enterprise AI deployment and management
- Multi-agent collaborative problem solving
- Workflow automation with AI components
- Secure and compliant AI operations

- Cross-provider AI orchestration

Core Architecture

Lumina AI follows a microservices architecture pattern, with distinct services handling specific domains of functionality:

Microservices Architecture

1. **Collaboration Service**: Enables multi-agent collaboration through dynamic team formation, shared context management, and negotiation protocols
2. **Workflow Service**: Orchestrates complex AI workflows with step execution, monitoring, and integration capabilities
3. **Deployment Service**: Manages deployment across environments with configuration management and pipeline execution
4. **Provider Service**: Integrates with multiple AI providers with dynamic selection and fallback mechanisms
5. **Governance Service**: Enforces ethical guidelines with content evaluation and policy management

Each service is built using Spring Boot (Java) for backend services with RESTful APIs, while specialized AI components are implemented in Python for maximum flexibility and access to AI libraries.

Technology Stack

- **Backend Services**: Java 17, Spring Boot 3.x, Spring Data JPA
- **AI Components**: Python 3.10+, PyTorch, TensorFlow, Hugging Face Transformers
- **Database**: PostgreSQL (relational data), MongoDB (document store), Pinecone (vector database)
- **Message Broker**: Apache Kafka for asynchronous communication
- **API Gateway**: Spring Cloud Gateway
- **Authentication**: OAuth2/JWT with Spring Security
- **Containerization**: Docker, Kubernetes
- **CI/CD**: GitHub Actions, Jenkins

System Interaction Flow

1. Client applications interact with Lumina AI through the API Gateway
2. The API Gateway routes requests to appropriate microservices
3. Microservices process requests, potentially communicating with other services
4. AI operations are delegated to specialized Python components
5. Results are returned through the API Gateway to client applications

Key Technical Features

Advanced Multi-Agent Collaboration System

The Collaboration Service enables multiple AI agents to work together effectively on complex tasks.

Dynamic Team Formation

- **Capability-Based Selection**: Assembles teams based on required capabilities for a task
- **Performance History**: Considers past performance in similar tasks
- **Compatibility Analysis**: Evaluates agent compatibility for effective collaboration
- **Adaptive Sizing**: Determines optimal team size based on task complexity
- **Role Assignment**: Assigns specific roles to agents based on their strengths

Implementation:

```java
public class AdvancedTeamFormationService {
    private final AgentRepository agentRepository;
    private final TaskRepository taskRepository;
    private final PerformanceHistoryRepository performanceRepository;

    public Team formTeam(Task task, TeamRequirements requirements) {
        // Analyze task requirements
        Set<Capability> requiredCapabilities = analyzeTaskRequirements(task);

        // Find agents with matching capabilities
        List<Agent> candidateAgents = agentRepository.findByCapabilitiesIn(requiredCapabilities);

        // Filter by performance history
        candidateAgents = filterByPerformanceHistory(candidateAgents, task.getType());

        // Analyze compatibility
        List<Agent> compatibleAgents = analyzeCompatibility(candidateAgents);

        // Determine optimal team size
        int optimalSize = determineOptimalTeamSize(task.getComplexity());

        // Select final team members
        List<Agent> teamMembers = selectTeamMembers(compatibleAgents, optimalSize);

        // Assign roles
        Map<Agent, Role> agentRoles = assignRoles(teamMembers, task);

        // Create and return team
        return new Team(teamMembers, agentRoles, task);
    }

    // Additional methods...
}
```

Shared Context Management

- **Versioned Context**: Maintains versioned context information
- **Conflict Resolution**: Resolves conflicts in context updates
- **Selective Sharing**: Controls what context is shared with which agents
- **Context Synchronization**: Ensures all agents have up-to-date context
- **Context Persistence**: Maintains context across sessions

Implementation:

```java
public class AdvancedSharedContextService {
    private final SharedContextRepository contextRepository;
    private final ContextVersionRepository versionRepository;
    private final ContextAccessRepository accessRepository;

    public SharedContext createContext(String name, String description, String ownerId) {
        SharedContext context = new SharedContext();
        context.setName(name);
        context.setDescription(description);
        context.setOwnerId(ownerId);
        context.setCreatedAt(LocalDateTime.now());
        SharedContext savedContext = contextRepository.save(context);

        // Create initial version
        ContextVersion initialVersion = new ContextVersion();
        initialVersion.setContextId(savedContext.getId());
        initialVersion.setVersionNumber(1);
        initialVersion.setCreatedAt(LocalDateTime.now());
        initialVersion.setCreatedBy(ownerId);
        versionRepository.save(initialVersion);

        // Set owner access
        ContextAccess ownerAccess = new ContextAccess();
        ownerAccess.setContextId(savedContext.getId());
        ownerAccess.setAgentId(ownerId);
        ownerAccess.setAccessLevel(AccessLevel.OWNER);
        accessRepository.save(ownerAccess);

        return savedContext;
    }

    public ContextVersion updateContext(String contextId, String agentId, Map<String, Object> changes)
        // Verify access
        ContextAccess access = accessRepository.findByContextIdAndAgentId(contextId, agentId)
            .orElseThrow(() -> new AccessDeniedException("Agent does not have access to this context"))

        if (access.getAccessLevel() == AccessLevel.READ_ONLY) {
            throw new AccessDeniedException("Agent has read-only access to this context");
        }

        // Get latest version
        ContextVersion latestVersion = versionRepository.findTopByContextIdOrderByVersionNumberDesc(con
            .orElseThrow(() -> new NotFoundException("Context not found"));

        // Create new version
        ContextVersion newVersion = new ContextVersion();
        newVersion.setContextId(contextId);
        newVersion.setVersionNumber(latestVersion.getVersionNumber() + 1);
        newVersion.setCreatedAt(LocalDateTime.now());
        newVersion.setCreatedBy(agentId);
        newVersion.setData(mergeChanges(latestVersion.getData(), changes));

        return versionRepository.save(newVersion);
    }

    // Additional methods...
}
```

Negotiation Protocol

- **Structured Communication**: Defined protocol for agent communication
- **Conflict Detection**: Identifies conflicts in agent proposals
- **Consensus Building**: Mechanisms for reaching agreement
- **Fallback Resolution**: Strategies when consensus cannot be reached

- **Negotiation History**: Tracks negotiation process for transparency

Implementation:

```java
public class AdvancedNegotiationService {
    private final NegotiationRepository negotiationRepository;
    private final ProposalRepository proposalRepository;
    private final AgentRepository agentRepository;

    public Negotiation startNegotiation(String topic, List<String> participantIds, NegotiationParameter
        // Verify participants exist
        List<Agent> participants = agentRepository.findAllById(participantIds);
        if (participants.size() != participantIds.size()) {
            throw new NotFoundException("One or more participants not found");
        }

        // Create negotiation
        Negotiation negotiation = new Negotiation();
        negotiation.setTopic(topic);
        negotiation.setParticipantIds(participantIds);
        negotiation.setParameters(parameters);
        negotiation.setStatus(NegotiationStatus.ACTIVE);
        negotiation.setStartedAt(LocalDateTime.now());

        return negotiationRepository.save(negotiation);
    }

    public Proposal submitProposal(String negotiationId, String agentId, ProposalContent content) {
        // Verify negotiation exists and is active
        Negotiation negotiation = negotiationRepository.findById(negotiationId)
            .orElseThrow(() -> new NotFoundException("Negotiation not found"));

        if (negotiation.getStatus() != NegotiationStatus.ACTIVE) {
            throw new IllegalStateException("Negotiation is not active");
        }

        // Verify agent is a participant
        if (!negotiation.getParticipantIds().contains(agentId)) {
            throw new AccessDeniedException("Agent is not a participant in this negotiation");
        }

        // Create proposal
        Proposal proposal = new Proposal();
        proposal.setNegotiationId(negotiationId);
        proposal.setAgentId(agentId);
        proposal.setContent(content);
        proposal.setSubmittedAt(LocalDateTime.now());

        Proposal savedProposal = proposalRepository.save(proposal);

        // Check for conflicts
        List<Proposal> existingProposals = proposalRepository.findByNegotiationId(negotiationId);
        List<Conflict> conflicts = detectConflicts(savedProposal, existingProposals);

        // Update proposal with conflicts
        if (!conflicts.isEmpty()) {
            savedProposal.setConflicts(conflicts);
            savedProposal = proposalRepository.save(savedProposal);
        }

        // Check for consensus
        boolean consensusReached = checkConsensus(negotiation, existingProposals);
        if (consensusReached) {
            negotiation.setStatus(NegotiationStatus.CONSENSUS_REACHED);
            negotiation.setCompletedAt(LocalDateTime.now());
            negotiationRepository.save(negotiation);
        }

        return savedProposal;
    }

    // Additional methods...
}
```

Workflow Orchestration Engine

The Workflow Service enables the definition, execution, and monitoring of complex AI workflows.

Workflow Definition

- **Declarative Definitions**: JSON/YAML-based workflow definitions
- **Step Configuration**: Detailed configuration for each workflow step
- **Transition Rules**: Conditional transitions between steps
- **Templating**: Reusable workflow templates
- **Versioning**: Version control for workflow definitions

Implementation:

```
public class WorkflowDefinitionService {
    private final WorkflowDefinitionRepository definitionRepository;
    private final WorkflowTemplateRepository templateRepository;

    public WorkflowDefinition createDefinition(WorkflowDefinition definition) {
        // Validate definition
        validateDefinition(definition);

        // Set creation metadata
        definition.setCreatedAt(LocalDateTime.now());
        definition.setVersion(1);

        return definitionRepository.save(definition);
    }

    public WorkflowDefinition updateDefinition(String id, WorkflowDefinition updatedDefinition) {
        // Find existing definition
        WorkflowDefinition existingDefinition = definitionRepository.findById(id)
            .orElseThrow(() -> new NotFoundException("Workflow definition not found"));

        // Create new version
        WorkflowDefinition newVersion = new WorkflowDefinition();
        newVersion.setName(updatedDefinition.getName());
        newVersion.setDescription(updatedDefinition.getDescription());
        newVersion.setSteps(updatedDefinition.getSteps());
        newVersion.setTransitions(updatedDefinition.getTransitions());
        newVersion.setInputSchema(updatedDefinition.getInputSchema());
        newVersion.setOutputSchema(updatedDefinition.getOutputSchema());
        newVersion.setCreatedAt(LocalDateTime.now());
        newVersion.setCreatedBy(updatedDefinition.getCreatedBy());
        newVersion.setVersion(existingDefinition.getVersion() + 1);
        newVersion.setPreviousVersionId(existingDefinition.getId());

        // Validate new version
        validateDefinition(newVersion);

        return definitionRepository.save(newVersion);
    }

    public WorkflowDefinition createFromTemplate(String templateId, Map<String, Object> parameters) {
        // Find template
        WorkflowTemplate template = templateRepository.findById(templateId)
            .orElseThrow(() -> new NotFoundException("Workflow template not found"));

        // Create definition from template
        WorkflowDefinition definition = new WorkflowDefinition();
        definition.setName(template.getName());
        definition.setDescription(template.getDescription());

        // Apply parameters to template
        definition.setSteps(applyParametersToSteps(template.getSteps(), parameters));
        definition.setTransitions(applyParametersToTransitions(template.getTransitions(), parameters));

        definition.setCreatedAt(LocalDateTime.now());
        definition.setVersion(1);
        definition.setTemplateId(templateId);

        // Validate definition
        validateDefinition(definition);

        return definitionRepository.save(definition);
    }

    // Additional methods...
}
```

Execution Engine

- **Stateful Execution**: Maintains execution state across steps
- **Context Management**: Passes context between workflow steps
- **Error Handling**: Sophisticated error handling and recovery
- **Parallel Execution**: Support for parallel step execution
- **Conditional Branching**: Dynamic workflow paths based on conditions

Implementation:

```
public class WorkflowExecutionEngine {
    private final WorkflowDefinitionRepository definitionRepository;
    private final WorkflowInstanceRepository instanceRepository;
    private final StepExecutionRepository stepExecutionRepository;
    private final ExecutionContextRepository contextRepository;

    public WorkflowInstance startWorkflow(String definitionId, Map<String, Object> input) {
        // Find definition
        WorkflowDefinition definition = definitionRepository.findById(definitionId)
            .orElseThrow(() -> new NotFoundException("Workflow definition not found"));

        // Validate input against schema
        validateInput(input, definition.getInputSchema());

        // Create workflow instance
        WorkflowInstance instance = new WorkflowInstance();
        instance.setDefinitionId(definitionId);
        instance.setStatus(WorkflowStatus.RUNNING);
        instance.setStartedAt(LocalDateTime.now());

        WorkflowInstance savedInstance = instanceRepository.save(instance);

        // Create execution context
        ExecutionContext context = new ExecutionContext();
        context.setInstanceId(savedInstance.getId());
        context.setData(input);
        contextRepository.save(context);

        // Find initial step
        WorkflowStep initialStep = findInitialStep(definition);

        // Execute initial step
        executeStep(savedInstance, initialStep, context);

        return savedInstance;
    }

    public void continueWorkflow(String instanceId, String stepId, StepExecutionResult result) {
        // Find instance
        WorkflowInstance instance = instanceRepository.findById(instanceId)
            .orElseThrow(() -> new NotFoundException("Workflow instance not found"));

        if (instance.getStatus() != WorkflowStatus.RUNNING) {
            throw new IllegalStateException("Workflow is not running");
        }

        // Find step execution
        StepExecution stepExecution = stepExecutionRepository.findByInstanceIdAndStepId(instanceId, step
            .orElseThrow(() -> new NotFoundException("Step execution not found"));

        // Update step execution
        stepExecution.setStatus(StepStatus.COMPLETED);
        stepExecution.setCompletedAt(LocalDateTime.now());
        stepExecution.setOutput(result.getOutput());
        stepExecutionRepository.save(stepExecution);
```

```
        // Update context
        ExecutionContext context = contextRepository.findByInstanceId(instanceId)
            .orElseThrow(() -> new NotFoundException("Execution context not found"));
        context.setData(mergeContextData(context.getData(), result.getContextUpdates()));
        contextRepository.save(context);

        // Find definition
        WorkflowDefinition definition = definitionRepository.findById(instance.getDefinitionId())
            .orElseThrow(() -> new NotFoundException("Workflow definition not found"));

        // Find next steps
        List<WorkflowStep> nextSteps = findNextSteps(definition, stepId, result, context);

        if (nextSteps.isEmpty()) {
            // No more steps, workflow is complete
            instance.setStatus(WorkflowStatus.COMPLETED);
            instance.setCompletedAt(LocalDateTime.now());
            instanceRepository.save(instance);
        } else {
            // Execute next steps
            for (WorkflowStep nextStep : nextSteps) {
                executeStep(instance, nextStep, context);
            }
        }
    }

    // Additional methods...
}
```

Monitoring

- **Real-time Status**: Live monitoring of workflow execution
- **Metrics Collection**: Performance and execution metrics
- **Alerting**: Configurable alerts for workflow issues
- **Visualization**: Graphical representation of workflow execution
- **Audit Logging**: Comprehensive logging for compliance

Implementation:

```
public class WorkflowMonitoringService {
    private final WorkflowInstanceRepository instanceRepository;
    private final StepExecutionRepository stepExecutionRepository;
    private final WorkflowDefinitionRepository definitionRepository;
    private final AlertService alertService;

    public WorkflowStatus getWorkflowStatus(String instanceId) {
        WorkflowInstance instance = instanceRepository.findById(instanceId)
            .orElseThrow(() -> new NotFoundException("Workflow instance not found"));

        return buildWorkflowStatus(instance);
    }

    public List<StepStatus> getStepStatuses(String instanceId) {
        // Verify instance exists
        if (!instanceRepository.existsById(instanceId)) {
            throw new NotFoundException("Workflow instance not found");
        }

        List<StepExecution> stepExecutions = stepExecutionRepository.findByInstanceId(instanceId);

        return stepExecutions.stream()
            .map(this::buildStepStatus)
            .collect(Collectors.toList());
    }

    public WorkflowMetrics getWorkflowMetrics(String instanceId) {
        WorkflowInstance instance = instanceRepository.findById(instanceId)
            .orElseThrow(() -> new NotFoundException("Workflow instance not found"));

        List<StepExecution> stepExecutions = stepExecutionRepository.findByInstanceId(instanceId);

        return calculateMetrics(instance, stepExecutions);
    }

    public void configureAlert(String definitionId, AlertConfiguration configuration) {
        // Verify definition exists
        if (!definitionRepository.existsById(definitionId)) {
            throw new NotFoundException("Workflow definition not found");
        }

        alertService.configureAlert(definitionId, configuration);
    }

    // Additional methods...
}
```

Enterprise Deployment System

The Deployment Service manages the deployment of Lumina AI components across different environments.

Multi-Environment Support

- **Environment Isolation**: Separate development, staging, and production environments
- **Environment-Specific Configuration**: Tailored settings for each environment
- **Promotion Workflow**: Controlled promotion between environments
- **Environment Templates**: Standardized environment configurations
- **Environment Health Monitoring**: Continuous monitoring of environment health

Implementation:

```
public class DeploymentService {
    private final DeploymentRepository deploymentRepository;
    private final DeploymentComponentRepository componentRepository;
    private final ConfigurationRepository configurationRepository;

    public Deployment createDeployment(Deployment deployment) {
        // Validate deployment
        validateDeployment(deployment);

        // Set creation metadata
        deployment.setStatus(DeploymentStatus.CREATED);
        deployment.setCreatedAt(LocalDateTime.now());

        return deploymentRepository.save(deployment);
    }

    public Deployment startDeployment(String deploymentId, String userId) {
        // Find deployment
        Deployment deployment = deploymentRepository.findById(deploymentId)
            .orElseThrow(() -> new NotFoundException("Deployment not found"));

        if (deployment.getStatus() != DeploymentStatus.CREATED) {
            throw new IllegalStateException("Deployment is not in CREATED state");
        }

        // Update status
        deployment.setStatus(DeploymentStatus.RUNNING);
        deployment.setStartedAt(LocalDateTime.now());
        deployment.setStartedBy(userId);

        // Start deployment process
        startDeploymentProcess(deployment);

        return deploymentRepository.save(deployment);
    }

    public Deployment promoteDeployment(String deploymentId, String targetEnvironment, String userId) {
        // Find deployment
        Deployment deployment = deploymentRepository.findById(deploymentId)
            .orElseThrow(() -> new NotFoundException("Deployment not found"));

        if (deployment.getStatus() != DeploymentStatus.COMPLETED) {
            throw new IllegalStateException("Deployment is not in COMPLETED state");
        }

        // Create new deployment for target environment
        Deployment promotedDeployment = new Deployment();
        promotedDeployment.setName(deployment.getName() + " (promoted to " + targetEnvironment + ")");
        promotedDeployment.setDescription("Promoted from " + deployment.getEnvironment() + " to " + targ
        promotedDeployment.setEnvironment(targetEnvironment);
        promotedDeployment.setComponents(deployment.getComponents());
        promotedDeployment.setCreatedAt(LocalDateTime.now());
        promotedDeployment.setCreatedBy(userId);
        promotedDeployment.setStatus(DeploymentStatus.CREATED);
        promotedDeployment.setSourceDeploymentId(deploymentId);

        // Apply environment-specific configurations
        applyEnvironmentConfigurations(promotedDeployment, targetEnvironment);

        return deploymentRepository.save(promotedDeployment);
    }

    // Additional methods...
}
```

Configuration Management

- **Secure Credential Storage**: Encrypted storage for sensitive credentials
- **Configuration Versioning**: Version history for configurations
- **Environment Variables**: Support for environment-specific variables
- **Configuration Validation**: Validation of configuration against schemas
- **Configuration Inheritance**: Hierarchical configuration with overrides

Implementation:

```
public class ConfigurationService {
    private final ConfigurationRepository configurationRepository;
    private final EncryptionService encryptionService;

    public Configuration createConfiguration(Configuration configuration) {
        // Validate configuration
        validateConfiguration(configuration);

        // Encrypt sensitive values
        encryptSensitiveValues(configuration);

        // Set creation metadata
        configuration.setCreatedAt(LocalDateTime.now());
        configuration.setVersion(1);

        return configurationRepository.save(configuration);
    }

    public Configuration updateConfiguration(String id, Configuration updatedConfiguration) {
        // Find existing configuration
        Configuration existingConfiguration = configurationRepository.findById(id)
            .orElseThrow(() -> new NotFoundException("Configuration not found"));

        // Create new version
        Configuration newVersion = new Configuration();
        newVersion.setName(updatedConfiguration.getName());
        newVersion.setDescription(updatedConfiguration.getDescription());
        newVersion.setEnvironment(updatedConfiguration.getEnvironment());
        newVersion.setValues(updatedConfiguration.getValues());
        newVersion.setCreatedAt(LocalDateTime.now());
        newVersion.setCreatedBy(updatedConfiguration.getCreatedBy());
        newVersion.setVersion(existingConfiguration.getVersion() + 1);
        newVersion.setPreviousVersionId(existingConfiguration.getId());

        // Encrypt sensitive values
        encryptSensitiveValues(newVersion);

        // Validate new version
        validateConfiguration(newVersion);

        return configurationRepository.save(newVersion);
    }

    public Map<String, Object> resolveConfiguration(String environment, String component) {
        // Find base configuration
        Configuration baseConfig = configurationRepository.findByEnvironmentAndComponentAndIsBase(envir
            .orElseThrow(() -> new NotFoundException("Base configuration not found"));

        // Find overrides
        List<Configuration> overrides = configurationRepository.findByEnvironmentAndComponentAndIsBaseO

        // Merge configurations
        Map<String, Object> resolvedConfig = new HashMap<>(baseConfig.getValues());
        for (Configuration override : overrides) {
            resolvedConfig.putAll(override.getValues());
        }

        // Decrypt sensitive values
        decryptSensitiveValues(resolvedConfig);

        return resolvedConfig;
    }

    // Additional methods...
}
```

Pipeline Engine

- **Multi-Stage Pipelines**: Sequential stages for deployment
- **Pipeline Steps**: Individual steps within each stage
- **Conditional Execution**: Conditional steps based on criteria
- **Rollback Support**: Automated rollback on failure
- **Pipeline Templates**: Reusable pipeline templates

Implementation:

```
public class PipelineService {
    private final PipelineRepository pipelineRepository;
    private final PipelineStageRepository stageRepository;
    private final PipelineStepRepository stepRepository;

    public Pipeline createPipeline(Pipeline pipeline) {
        // Validate pipeline
        validatePipeline(pipeline);

        // Set creation metadata
        pipeline.setCreatedAt(LocalDateTime.now());

        Pipeline savedPipeline = pipelineRepository.save(pipeline);

        // Save stages and steps
        for (PipelineStage stage : pipeline.getStages()) {
            stage.setPipelineId(savedPipeline.getId());
            PipelineStage savedStage = stageRepository.save(stage);

            for (PipelineStep step : stage.getSteps()) {
                step.setStageId(savedStage.getId());
                stepRepository.save(step);
            }
        }
        return savedPipeline;
    }

    public Pipeline executePipeline(String pipelineId, Map<String, Object> parameters) {
        // Find pipeline
        Pipeline pipeline = pipelineRepository.findById(pipelineId)
            .orElseThrow(() -> new NotFoundException("Pipeline not found"));

        // Validate parameters
        validateParameters(parameters, pipeline.getParameterSchema());

        // Update pipeline status
        pipeline.setStatus(PipelineStatus.RUNNING);
        pipeline.setStartedAt(LocalDateTime.now());
        pipeline.setParameters(parameters);

        Pipeline updatedPipeline = pipelineRepository.save(pipeline);

        // Execute pipeline asynchronously
        executePipelineAsync(updatedPipeline);

        return updatedPipeline;
    }

    public Pipeline getPipelineStatus(String pipelineId) {
        return pipelineRepository.findById(pipelineId)
            .orElseThrow(() -> new NotFoundException("Pipeline not found"));
    }

    // Additional methods...
}
```

API Gateway

- **Request Routing**: Intelligent routing to appropriate services
- **Authentication**: Centralized authentication and authorization
- **Rate Limiting**: Protection against excessive requests
- **Request/Response Logging**: Comprehensive logging for auditing
- **API Documentation**: Automatic API documentation generation

Implementation:

```
public class ApiGatewayConfig {
    private final ApiAuthenticationInterceptor authInterceptor;
    private final ApiRequestLoggingInterceptor loggingInterceptor;
    private final ApiRateLimitingInterceptor rateLimitingInterceptor;

    @Bean
    public WebMvcConfigurer apiGatewayConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addInterceptors(InterceptorRegistry registry) {
                registry.addInterceptor(authInterceptor).addPathPatterns("/api/**");
                registry.addInterceptor(loggingInterceptor).addPathPatterns("/api/**");
                registry.addInterceptor(rateLimitingInterceptor).addPathPatterns("/api/**");
            }
        };
    }

    @Bean
    public RouterFunction<ServerResponse> routerFunction() {
        return RouterFunctions
            .route(RequestPredicates.GET("/api/deployments/**"), this::routeToDeploymentService)
            .andRoute(RequestPredicates.GET("/api/configurations/**"), this::routeToConfigurationServic
            .andRoute(RequestPredicates.GET("/api/pipelines/**"), this::routeToPipelineService)
            .andRoute(RequestPredicates.GET("/api/providers/**"), this::routeToProviderService)
            .andRoute(RequestPredicates.GET("/api/governance/**"), this::routeToGovernanceService);
    }

    // Additional methods...
}
```

Advanced Provider Integration

The Provider Service enables seamless integration with multiple AI providers.

Unified API Interface

- **Provider Abstraction**: Common interface across providers
- **Model Registry**: Centralized registry of available models
- **Capability Discovery**: Automatic discovery of provider capabilities
- **API Versioning**: Support for multiple API versions
- **Request/Response Mapping**: Standardized mapping between providers

Implementation:

```
public class ProviderService {
    private final ProviderRepository providerRepository;
    private final ModelRepository modelRepository;
    private final ProviderCapabilityRepository capabilityRepository;

    public Provider registerProvider(Provider provider) {
        // Validate provider
        validateProvider(provider);

        // Set creation metadata
        provider.setCreatedAt(LocalDateTime.now());

        Provider savedProvider = providerRepository.save(provider);

        // Discover capabilities
        List<ProviderCapability> capabilities = discoverCapabilities(savedProvider);
        for (ProviderCapability capability : capabilities) {
            capability.setProviderId(savedProvider.getId());
            capabilityRepository.save(capability);
        }

        // Discover models
        List<Model> models = discoverModels(savedProvider);
        for (Model model : models) {
            model.setProviderId(savedProvider.getId());
            modelRepository.save(model);
        }

        return savedProvider;
    }

    public List<Provider> getProvidersByCapability(String capability) {
        return providerRepository.findByCapabilitiesContaining(capability);
    }

    public List<Model> getModelsByCapability(String capability) {
        return modelRepository.findByCapabilitiesContaining(capability);
    }

    public Provider updateApiKey(String providerId, String keyName, String keyValue) {
        // Find provider
        Provider provider = providerRepository.findById(providerId)
            .orElseThrow(() -> new NotFoundException("Provider not found"));

        // Update API key
        Map<String, String> apiKeys = provider.getApiKeys();
        if (apiKeys == null) {
            apiKeys = new HashMap<>();
        }
        apiKeys.put(keyName, keyValue);
        provider.setApiKeys(apiKeys);
        provider.setUpdatedAt(LocalDateTime.now());

        return providerRepository.save(provider);
    }

    // Additional methods...
}
```

Dynamic Model Selection

- **Task-Based Selection**: Selects models based on task requirements
- **Performance Metrics**: Considers performance metrics in selection
- **Cost Optimization**: Balances performance and cost
- **Availability Checking**: Verifies model availability before selection
- **Capability Matching**: Matches task requirements to model capabilities

Implementation:

```
public class ModelService {
    private final ModelRepository modelRepository;
    private final ProviderRepository providerRepository;
    private final PerformanceMetricRepository metricRepository;

    public Model selectModelForTask(TaskRequirements requirements) {
        // Find models with required capabilities
        List<Model> candidateModels = modelRepository.findByCapabilitiesContainingAll(requirements.getR

        // Filter by context size if specified
        if (requirements.getMinContextSize() != null) {
            candidateModels = candidateModels.stream()
                .filter(model -> model.getMaxContextSize() >= requirements.getMinContextSize())
                .collect(Collectors.toList());
        }

        // Get performance metrics for candidate models
        Map<String, List<PerformanceMetric>> modelMetrics = new HashMap<>();
        for (Model model : candidateModels) {
            List<PerformanceMetric> metrics = metricRepository.findByModelIdAndTaskType(model.getId(),
            modelMetrics.put(model.getId(), metrics);
        }

        // Calculate scores based on performance, cost, and availability
        Map<Model, Double> modelScores = new HashMap<>();
        for (Model model : candidateModels) {
            double performanceScore = calculatePerformanceScore(model, modelMetrics.get(model.getId()),
            double costScore = calculateCostScore(model, requirements);
            double availabilityScore = calculateAvailabilityScore(model);

            // Weighted score
            double totalScore = (performanceScore * requirements.getPerformanceWeight()) +
                               (costScore * requirements.getCostWeight()) +
                               (availabilityScore * requirements.getAvailabilityWeight());

            modelScores.put(model, totalScore);
        }

        // Select model with highest score
        return modelScores.entrySet().stream()
            .max(Map.Entry.comparingByValue())
            .map(Map.Entry::getKey)
            .orElseThrow(() -> new NoSuitableModelException("No suitable model found for the given requ
    }

    // Additional methods...
}
```

Parameter Optimization

- **Task-Specific Optimization**: Optimizes parameters for specific tasks
- **Learning from History**: Improves parameters based on past results
- **A/B Testing**: Compares different parameter sets
- **Parameter Templates**: Reusable parameter templates for common tasks
- **Custom Parameter Validation**: Validates parameters against model constraints

Implementation:

```
public class ParameterOptimizationService {
    private final ModelRepository modelRepository;
    private final RequestRepository requestRepository;
    private final ParameterTemplateRepository templateRepository;

    public Map<String, Object> optimizeParameters(String modelId, String taskType, Map<String, Object>
        // Find model
        Model model = modelRepository.findById(modelId)
            .orElseThrow(() -> new NotFoundException("Model not found"));

        // Get parameter constraints
        Map<String, ParameterConstraint> constraints = model.getParameterConstraints();

        // Get historical requests for this model and task type
        List<Request> historicalRequests = requestRepository.findByModelIdAndTaskTypeOrderByCreatedAtDe

        // Extract successful parameters
        List<Map<String, Object>> successfulParameters = historicalRequests.stream()
            .filter(request -> request.getStatus() == RequestStatus.COMPLETED && request.getQualityScor
            .map(Request::getParameters)
            .collect(Collectors.toList());

        // Optimize parameters
        Map<String, Object> optimizedParameters = new HashMap<>(baseParameters);

        // Apply historical learning
        if (!successfulParameters.isEmpty()) {
            optimizedParameters = applyHistoricalLearning(optimizedParameters, successfulParameters);
        }

        // Apply task-specific optimizations
        optimizedParameters = applyTaskSpecificOptimizations(optimizedParameters, taskType);

        // Validate against constraints
        validateParameters(optimizedParameters, constraints);

        return optimizedParameters;
    }

    public ParameterTemplate createTemplate(ParameterTemplate template) {
        // Validate template
        validateTemplate(template);

        // Set creation metadata
        template.setCreatedAt(LocalDateTime.now());

        return templateRepository.save(template);
    }

    // Additional methods...
}
```

Robust Error Handling

- **Error Classification**: Categorizes errors for appropriate handling
- **Retry Strategies**: Sophisticated retry strategies with backoff
- **Fallback Mechanisms**: Graceful fallbacks when providers fail
- **Error Reporting**: Detailed error reporting for diagnosis
- **Circuit Breaking**: Prevents cascading failures

Implementation:

```java
public class ProviderErrorHandlingService {
    private final CircuitBreakerRegistry circuitBreakerRegistry;
    private final RetryRegistry retryRegistry;
    private final ProviderRepository providerRepository;

    public <T> T executeWithErrorHandling(String providerId, Supplier<T> operation) {
        // Find provider
        Provider provider = providerRepository.findById(providerId)
            .orElseThrow(() -> new NotFoundException("Provider not found"));

        // Get circuit breaker for provider
        CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker(providerId);

        // Get retry for provider
        Retry retry = retryRegistry.retry(providerId);

        // Execute with circuit breaker and retry
        try {
            return Decorators.ofSupplier(operation)
                .withCircuitBreaker(circuitBreaker)
                .withRetry(retry)
                .decorate()
                .get();
        } catch (Exception e) {
            // Classify error
            ProviderErrorType errorType = classifyError(e);

            // Handle based on error type
            switch (errorType) {
                case AUTHENTICATION:
                    throw new ProviderAuthenticationException("Authentication failed for provider: " + 
                case RATE_LIMIT:
                    throw new ProviderRateLimitException("Rate limit exceeded for provider: " + provide
                case TIMEOUT:
                    throw new ProviderTimeoutException("Request timed out for provider: " + provider.ge
                case INVALID_REQUEST:
                    throw new ProviderInvalidRequestException("Invalid request for provider: " + provid
                case SERVICE_UNAVAILABLE:
                    throw new ProviderUnavailableException("Provider service unavailable: " + provider.
                default:
                    throw new ProviderException("Error executing operation on provider: " + provider.ge
            }
        }
    }

    // Additional methods...
}
```

Tool Use Framework

- **Tool Registry**: Centralized registry of available tools
- **Tool Execution**: Standardized tool execution interface
- **Tool Result Handling**: Processing and validation of tool results
- **Tool Chaining**: Support for chaining multiple tools

- **Tool Documentation**: Automatic tool documentation generation

Implementation:

```java
public class ToolService {
    private final ToolRepository toolRepository;
    private final ToolExecutionRepository executionRepository;

    public Tool registerTool(Tool tool) {
        // Validate tool
        validateTool(tool);

        // Set creation metadata
        tool.setCreatedAt(LocalDateTime.now());

        return toolRepository.save(tool);
    }

    public List<Tool> getToolsByCapability(String capability) {
        return toolRepository.findByCapabilitiesContaining(capability);
    }

    public ToolExecution executeTool(String toolId, Map<String, Object> parameters) {
        // Find tool
        Tool tool = toolRepository.findById(toolId)
            .orElseThrow(() -> new NotFoundException("Tool not found"));

        // Validate parameters
        validateParameters(parameters, tool.getParameterSchema());

        // Create execution record
        ToolExecution execution = new ToolExecution();
        execution.setToolId(toolId);
        execution.setParameters(parameters);
        execution.setStatus(ToolExecutionStatus.RUNNING);
        execution.setStartedAt(LocalDateTime.now());

        ToolExecution savedExecution = executionRepository.save(execution);

        // Execute tool asynchronously
        executeToolAsync(tool, parameters, savedExecution.getId());

        return savedExecution;
    }

    public ToolExecution getToolExecutionResult(String executionId) {
        return executionRepository.findById(executionId)
            .orElseThrow(() -> new NotFoundException("Tool execution not found"));
    }

    // Additional methods...
}
```

Ethical AI Governance Framework

The Governance Service ensures that Lumina AI operates ethically and in compliance with regulations.

Policy Management

- **Policy Definition**: Defines governance policies with rules
- **Regional Compliance**: Policies tailored to regional requirements
- **Policy Enforcement**: Mechanisms to enforce policies
- **Policy Auditing**: Tracks policy application and exceptions
- **Policy Versioning**: Version control for policies

Implementation:

```java
public class GovernancePolicyService {
    private final GovernancePolicyRepository policyRepository;

    public GovernancePolicy createPolicy(GovernancePolicy policy) {
        // Validate policy
        validatePolicy(policy);

        // Set creation metadata
        policy.setCreatedAt(LocalDateTime.now());

        return policyRepository.save(policy);
    }

    public List<GovernancePolicy> getPoliciesByRegion(Region region) {
        return policyRepository.findByApplicableRegion(region);
    }

    public List<GovernancePolicy> getEnabledPoliciesByRegion(Region region) {
        return policyRepository.findByApplicableRegionAndEnabledTrue(region);
    }

    public GovernancePolicy updatePolicy(String policyId, GovernancePolicy updatedPolicy) {
        // Find existing policy
        GovernancePolicy existingPolicy = policyRepository.findById(policyId)
            .orElseThrow(() -> new NotFoundException("Governance policy not found"));

        // Update fields
        existingPolicy.setName(updatedPolicy.getName());
        existingPolicy.setDescription(updatedPolicy.getDescription());
        existingPolicy.setPolicyDefinition(updatedPolicy.getPolicyDefinition());
        existingPolicy.setEnforcementRules(updatedPolicy.getEnforcementRules());
        existingPolicy.setPriority(updatedPolicy.getPriority());
        existingPolicy.setUpdatedAt(LocalDateTime.now());
        existingPolicy.setUpdatedBy(updatedPolicy.getUpdatedBy());

        return policyRepository.save(existingPolicy);
    }

    public GovernancePolicy enablePolicy(String policyId) {
        // Find policy
        GovernancePolicy policy = policyRepository.findById(policyId)
            .orElseThrow(() -> new NotFoundException("Governance policy not found"));
        // Enable policy
        policy.setEnabled(true);
        policy.setUpdatedAt(LocalDateTime.now());

        return policyRepository.save(policy);
    }

    // Additional methods...
}
```

Content Evaluation

- **Multi-Dimensional Evaluation**: Evaluates content across multiple dimensions
- **Safety Scoring**: Scores content for safety concerns
- **Privacy Protection**: Identifies and protects private information
- **Transparency Metrics**: Measures transparency of AI operations
- **Content Filtering**: Filters inappropriate or harmful content

Implementation:

```
public class ContentEvaluationService {
    private final ContentEvaluationRepository evaluationRepository;
    private final GovernancePolicyService policyService;

    public ContentEvaluation evaluateContent(String content, ContentType contentType, String requestId,
        // Create evaluation record
        ContentEvaluation evaluation = new ContentEvaluation();
        evaluation.setContent(content);
        evaluation.setContentType(contentType);
        evaluation.setRequestId(requestId);
        evaluation.setUserId(userId);
        evaluation.setModelId(modelId);
        evaluation.setProviderId(providerId);
        evaluation.setEvaluatedAt(LocalDateTime.now());

        // Evaluate safety
        double safetyScore = evaluateSafety(content, contentType);
        evaluation.setSafetyScore(safetyScore);

        // Evaluate privacy
        double privacyScore = evaluatePrivacy(content, contentType);
        evaluation.setPrivacyScore(privacyScore);

        // Evaluate transparency
        double transparencyScore = evaluateTransparency(content, contentType, modelId);
        evaluation.setTransparencyScore(transparencyScore);

        // Identify flags
        Map<String, Double> flags = identifyFlags(content, contentType);
        evaluation.setFlags(flags);

        // Generate evaluation details
        String evaluationDetails = generateEvaluationDetails(safetyScore, privacyScore, transparencySco
        evaluation.setEvaluationDetails(evaluationDetails);
        return evaluationRepository.save(evaluation);
    }

    public List<ContentEvaluation> getEvaluationsBelowSafetyThreshold(double threshold) {
        return evaluationRepository.findBySafetyScoreLessThan(threshold);
    }

    public List<ContentEvaluation> getEvaluationsBelowPrivacyThreshold(double threshold) {
        return evaluationRepository.findByPrivacyScoreLessThan(threshold);
    }

    public List<ContentEvaluation> getEvaluationsWithFlagAboveThreshold(String flag, double threshold) 
        return evaluationRepository.findByFlagAboveThreshold(flag, threshold);
    }

    // Additional methods...
}
```

User Consent Management

- **Consent Collection**: Collects and records user consent
- **Consent Verification**: Verifies consent before processing
- **Consent Revocation**: Supports revoking previously given consent
- **Consent History**: Maintains history of consent changes
- **Purpose-Specific Consent**: Granular consent for specific purposes

Implementation:

```
public class UserConsentService {
    private final UserConsentRepository consentRepository;

    public UserConsent recordConsent(String userId, ConsentType consentType, Map<String, Boolean> conse
        // Create consent record
        UserConsent consent = new UserConsent();
        consent.setUserId(userId);
        consent.setConsentType(consentType);
        consent.setConsentDetails(consentDetails);
        consent.setSource(source);
        consent.setGrantedAt(LocalDateTime.now());

        return consentRepository.save(consent);
    }

    public boolean verifyConsent(String userId, ConsentType consentType, String purpose) {
        // Find latest consent
        UserConsent latestConsent = consentRepository.findTopByUserIdAndConsentTypeOrderByGrantedAtDesc
            .orElse(null);

        if (latestConsent == null) {
            return false;
        }

        // Check if consent is revoked
        if (latestConsent.isRevoked()) {
            return false;
        }

        // Check purpose-specific consent
        Map<String, Boolean> consentDetails = latestConsent.getConsentDetails();
        if (consentDetails != null && consentDetails.containsKey(purpose)) {
            return consentDetails.get(purpose);
        }

        // Default to general consent
        return true;
    }

    public UserConsent revokeConsent(String userId, ConsentType consentType, String reason) {
        // Find latest consent
        UserConsent latestConsent = consentRepository.findTopByUserIdAndConsentTypeOrderByGrantedAtDesc
            .orElseThrow(() -> new NotFoundException("No consent record found"));

        // Create revocation record
        UserConsent revocation = new UserConsent();
        revocation.setUserId(userId);
        revocation.setConsentType(consentType);
        revocation.setRevoked(true);
        revocation.setRevocationReason(reason);
        revocation.setRevokedAt(LocalDateTime.now());
        revocation.setPreviousConsentId(latestConsent.getId());

        return consentRepository.save(revocation);
    }

    // Additional methods...
}
```

Transparency Records

- **Operation Logging**: Logs all AI operations
- **Decision Explanation**: Provides explanations for AI decisions
- **Confidence Metrics**: Reports confidence levels for outputs
- **Source Attribution**: Attributes sources used in generation
- **User Access**: Allows users to access their transparency records

Implementation:

```
public class TransparencyRecordService {
    private final TransparencyRecordRepository recordRepository;

    public TransparencyRecord createRecord(String requestId, String userId, String modelId, String opera
        // Create transparency record
        TransparencyRecord record = new TransparencyRecord();
        record.setRequestId(requestId);
        record.setUserId(userId);
        record.setModelId(modelId);
        record.setOperation(operation);
        record.setDetails(details);
        record.setCreatedAt(LocalDateTime.now());
```

```java
        return recordRepository.save(record);
    }

    public TransparencyRecord addExplanation(String recordId, String explanation, List<String> factors)
        // Find record
        TransparencyRecord record = recordRepository.findById(recordId)
            .orElseThrow(() -> new NotFoundException("Transparency record not found"));

        // Add explanation
        record.setExplanation(explanation);
        record.setFactors(factors);
        record.setUpdatedAt(LocalDateTime.now());

        return recordRepository.save(record);
    }

    public TransparencyRecord addSourceAttribution(String recordId, List<SourceAttribution> sources) {
        // Find record
        TransparencyRecord record = recordRepository.findById(recordId)
            .orElseThrow(() -> new NotFoundException("Transparency record not found"));

        // Add source attribution
        record.setSources(sources);
        record.setUpdatedAt(LocalDateTime.now());

        return recordRepository.save(record);
    }

    public List<TransparencyRecord> getRecordsByUserId(String userId) {
        return recordRepository.findByUserId(userId);
    }

    // Additional methods...
}
```

Technical Implementation Details

Data Model

Lumina AI uses a sophisticated data model with multiple storage mechanisms:

Relational Database (PostgreSQL)

- **Core Entities**: Stores structured data for all core entities
- **Relationships**: Maintains relationships between entities
- **Transactional Data**: Handles transactional operations
- **User Data**: Stores user profiles and preferences
- **Configuration Data**: Stores system configuration

Vector Database (Pinecone)

- **Embeddings**: Stores vector embeddings for semantic search
- **Similarity Search**: Enables finding similar content
- **Clustering**: Supports clustering of related content

- **Dimensionality Reduction**: Optimizes storage and retrieval
- **Incremental Updates**: Supports efficient updates to vectors

Document Store (MongoDB)

- **Unstructured Data**: Stores unstructured content
- **Large Context Windows**: Handles large context windows
- **Flexible Schema**: Adapts to varying data structures
- **Hierarchical Data**: Stores nested and hierarchical data
- **Binary Data**: Stores binary data like images and audio

API Design

Lumina AI provides multiple API interfaces for different use cases:

RESTful APIs

- **Resource-Based**: Organized around resources
- **Standard Methods**: Uses standard HTTP methods
- **Stateless**: Maintains no client state on the server
- **Cacheable**: Supports caching for improved performance
- **Layered System**: Supports intermediary servers

Example Endpoint:

```
GET /api/providers/{providerId}/models
```

Response:

```json
{
  "models": [
    {
      "id": "model-123",
      "name": "GPT-4",
      "provider": "OpenAI",
      "capabilities": ["text-generation", "code-generation"],
      "maxContextSize": 8192,
      "inputTokenCost": 0.01,
      "outputTokenCost": 0.03
    },
    {
      "id": "model-456",
      "name": "Claude 3",
      "provider": "Anthropic",
      "capabilities": ["text-generation", "reasoning"],
      "maxContextSize": 100000,
      "inputTokenCost": 0.008,
      "outputTokenCost": 0.024
    }
  ],
  "total": 2,
  "page": 1,
  "pageSize": 10
}
```

GraphQL

- **Query Language**: Flexible query language
- **Single Endpoint**: One endpoint for all operations
- **Client-Specified Queries**: Client specifies exactly what data it needs
- **Strong Typing**: Strongly typed schema
- **Introspection**: Self-documenting API

Example Query:

```
query {
  provider(id: "provider-123") {
    name
    models {
      id
      name
      capabilities
      maxContextSize
    }
  }
}
```

Response:

```json
{
  "data": {
    "provider": {
      "name": "OpenAI",
      "models": [
        {
          "id": "model-123",
          "name": "GPT-4",
          "capabilities": ["text-generation", "code-generation"],
          "maxContextSize": 8192
        },
        {
          "id": "model-789",
          "name": "GPT-4-Vision",
          "capabilities": ["text-generation", "image-understanding"],
          "maxContextSize": 8192
        }
      ]
    }
  }
}
```

Streaming APIs

- **Server-Sent Events**: Real-time updates from server
- **WebSockets**: Bidirectional communication
- **Chunked Responses**: Incremental delivery of large responses
- **Long Polling**: Fallback for environments without WebSocket support
- **Heartbeat Mechanism**: Keeps connections alive

Example WebSocket Message:

```json
{
  "type": "token",
  "content": "Hello",
  "index": 0
}
```

Security Architecture

Lumina AI implements a comprehensive security architecture:

Authentication

- **OAuth2/JWT**: Industry-standard authentication
- **Multi-Factor Authentication**: Additional security layer
- **API Keys**: For service-to-service authentication
- **Session Management**: Secure session handling
- **Token Revocation**: Immediate revocation of compromised tokens

Authorization

- **Role-Based Access Control**: Access based on roles
- **Attribute-Based Access Control**: Fine-grained access control
- **Resource-Level Permissions**: Permissions at the resource level
- **Dynamic Permissions**: Context-aware permission evaluation
- **Permission Auditing**: Tracks permission changes

Data Protection

- **Encryption at Rest**: All sensitive data encrypted in storage
- **Encryption in Transit**: TLS for all communications
- **Field-Level Encryption**: Encryption of specific sensitive fields
- **Key Rotation**: Regular rotation of encryption keys
- **Secure Key Management**: Hardware security modules for key storage

API Security

- **Rate Limiting**: Protection against excessive requests
- **Request Validation**: Validates all incoming requests
- **Input Sanitization**: Prevents injection attacks
- **CORS Configuration**: Controls cross-origin requests
- **Security Headers**: Implements security headers

Scalability Considerations

Lumina AI is designed for horizontal scalability:

Horizontal Scaling

- **Stateless Services**: Services maintain no client state
- **Load Balancing**: Distributes load across instances
- **Service Discovery**: Automatic discovery of service instances
- **Auto-Scaling**: Automatically scales based on load
- **Regional Deployment**: Deployment across multiple regions

Asynchronous Processing

- **Message Queues**: Decouples producers and consumers
- **Event-Driven Architecture**: Services react to events
- **Background Processing**: Handles long-running tasks in background
- **Scheduled Tasks**: Executes tasks on schedule

- **Retry Mechanisms**: Automatically retries failed operations

Caching

- **Multi-Level Caching**: Caching at multiple levels
- **Distributed Cache**: Shared cache across instances
- **Cache Invalidation**: Strategies for keeping cache fresh
- **Cache Warming**: Pre-populates cache for common queries
- **Time-to-Live**: Automatic expiration of cached items

Load Balancing

- **Round-Robin**: Simple distribution of requests
- **Least Connections**: Routes to least busy instance
- **Resource-Based**: Routes based on instance resources
- **Sticky Sessions**: Maintains session affinity when needed
- **Health Checks**: Routes only to healthy instances

Component Implementation

Collaboration Service

The Collaboration Service enables multiple AI agents to work together effectively.

Key Classes

- **Team**: Represents a group of agents working together
- **Agent**: Represents an individual AI agent
- **SharedContext**: Shared information between agents
- **Negotiation**: Structured communication between agents
- **Proposal**: Suggestion made during negotiation

Database Schema

```sql
CREATE TABLE teams (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL,
    task_id VARCHAR(36)
);

CREATE TABLE agents (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    capabilities JSONB,
    provider_id VARCHAR(36),
    model_id VARCHAR(36),
    created_at TIMESTAMP NOT NULL
);

CREATE TABLE team_members (
    team_id VARCHAR(36) NOT NULL,
    agent_id VARCHAR(36) NOT NULL,
    role VARCHAR(255),
    joined_at TIMESTAMP NOT NULL,
    PRIMARY KEY (team_id, agent_id),
    FOREIGN KEY (team_id) REFERENCES teams(id),
    FOREIGN KEY (agent_id) REFERENCES agents(id)
);

CREATE TABLE shared_contexts (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    owner_id VARCHAR(36) NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP
);

CREATE TABLE context_versions (
    id VARCHAR(36) PRIMARY KEY,
    context_id VARCHAR(36) NOT NULL,
    version_number INT NOT NULL,
    data JSONB,
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL,
    FOREIGN KEY (context_id) REFERENCES shared_contexts(id)
);

CREATE TABLE context_access (
    context_id VARCHAR(36) NOT NULL,
    agent_id VARCHAR(36) NOT NULL,
    access_level VARCHAR(20) NOT NULL,
    granted_at TIMESTAMP NOT NULL,
    granted_by VARCHAR(36) NOT NULL,
    PRIMARY KEY (context_id, agent_id),
    FOREIGN KEY (context_id) REFERENCES shared_contexts(id)
);

CREATE TABLE negotiations (
    id VARCHAR(36) PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    participant_ids JSONB NOT NULL,
    parameters JSONB,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP
);

CREATE TABLE proposals (
    id VARCHAR(36) PRIMARY KEY,
    negotiation_id VARCHAR(36) NOT NULL,
    agent_id VARCHAR(36) NOT NULL,
    content JSONB NOT NULL,
    conflicts JSONB,
    submitted_at TIMESTAMP NOT NULL,
    FOREIGN KEY (negotiation_id) REFERENCES negotiations(id)
```

```
);
```

API Endpoints

- `POST /api/teams`: Create a new team
- `GET /api/teams/{teamId}`: Get team details
- `POST /api/teams/{teamId}/members`: Add member to team
- `POST /api/contexts`: Create a shared context
- `GET /api/contexts/{contextId}`: Get context details
- `PUT /api/contexts/{contextId}`: Update context
- `POST /api/negotiations`: Start a negotiation
- `POST /api/negotiations/{negotiationId}/proposals`: Submit a proposal

Workflow Service

The Workflow Service orchestrates complex AI workflows.

Key Classes

- **WorkflowDefinition**: Defines a workflow
- **WorkflowStep**: Individual step in a workflow
- **WorkflowTransition**: Transition between steps
- **WorkflowInstance**: Running instance of a workflow
- **StepExecution**: Execution of a workflow step

Database Schema

```sql
CREATE TABLE workflow_definitions (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    version INT NOT NULL,
    previous_version_id VARCHAR(36),
    template_id VARCHAR(36),
    input_schema JSONB,
    output_schema JSONB,
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL
);

CREATE TABLE workflow_steps (
    id VARCHAR(36) PRIMARY KEY,
    definition_id VARCHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    type VARCHAR(50) NOT NULL,
    configuration JSONB,
    is_initial BOOLEAN NOT NULL,
    is_final BOOLEAN NOT NULL,
    FOREIGN KEY (definition_id) REFERENCES workflow_definitions(id)
);

CREATE TABLE workflow_transitions (
    id VARCHAR(36) PRIMARY KEY,
    definition_id VARCHAR(36) NOT NULL,
    source_step_id VARCHAR(36) NOT NULL,
    target_step_id VARCHAR(36) NOT NULL,
    condition TEXT,
    priority INT NOT NULL,
    FOREIGN KEY (definition_id) REFERENCES workflow_definitions(id),
    FOREIGN KEY (source_step_id) REFERENCES workflow_steps(id),
    FOREIGN KEY (target_step_id) REFERENCES workflow_steps(id)
);

CREATE TABLE workflow_instances (
    id VARCHAR(36) PRIMARY KEY,
    definition_id VARCHAR(36) NOT NULL,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    FOREIGN KEY (definition_id) REFERENCES workflow_definitions(id)
);

CREATE TABLE step_executions (
    id VARCHAR(36) PRIMARY KEY,
    instance_id VARCHAR(36) NOT NULL,
    step_id VARCHAR(36) NOT NULL,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    input JSONB,
    output JSONB,
    error TEXT,
    FOREIGN KEY (instance_id) REFERENCES workflow_instances(id),
    FOREIGN KEY (step_id) REFERENCES workflow_steps(id)
);

CREATE TABLE execution_contexts (
    id VARCHAR(36) PRIMARY KEY,
    instance_id VARCHAR(36) NOT NULL,
    data JSONB,
    updated_at TIMESTAMP NOT NULL,
    FOREIGN KEY (instance_id) REFERENCES workflow_instances(id)
);
```

API Endpoints

- `POST /api/workflows/definitions`: Create workflow definition
- `GET /api/workflows/definitions/{definitionId}`: Get definition details
- `POST /api/workflows/instances`: Start workflow instance
- `GET /api/workflows/instances/{instanceId}`: Get instance details
- `POST /api/workflows/instances/{instanceId}/steps/{stepId}/complete`: Complete step
- `GET /api/workflows/instances/{instanceId}/context`: Get execution context

Deployment Service

The Deployment Service manages the deployment of Lumina AI components.

Key Classes

- **Deployment**: Represents a deployment
- **DeploymentComponent**: Component being deployed
- **Configuration**: Environment-specific configuration
- **Pipeline**: Deployment pipeline
- **Infrastructure**: Target infrastructure

Database Schema

```sql
CREATE TABLE deployments (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    environment VARCHAR(50) NOT NULL,
    status VARCHAR(20) NOT NULL,
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL,
    started_at TIMESTAMP,
    started_by VARCHAR(36),
    completed_at TIMESTAMP,
    source_deployment_id VARCHAR(36)
);

CREATE TABLE deployment_components (
    id VARCHAR(36) PRIMARY KEY,
    deployment_id VARCHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    type VARCHAR(50) NOT NULL,
    version VARCHAR(50) NOT NULL,
    configuration_id VARCHAR(36),
    status VARCHAR(20) NOT NULL,
    FOREIGN KEY (deployment_id) REFERENCES deployments(id)
);

CREATE TABLE configurations (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    environment VARCHAR(50) NOT NULL,
    component VARCHAR(255),
    values JSONB NOT NULL,
    is_base BOOLEAN NOT NULL,
    priority INT,
    version INT NOT NULL,
    previous_version_id VARCHAR(36),
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL
);

CREATE TABLE pipelines (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    status VARCHAR(20) NOT NULL,
    parameters JSONB,
    created_at TIMESTAMP NOT NULL,
    started_at TIMESTAMP,
    completed_at TIMESTAMP
);

CREATE TABLE pipeline_stages (
    id VARCHAR(36) PRIMARY KEY,
    pipeline_id VARCHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    order_index INT NOT NULL,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    FOREIGN KEY (pipeline_id) REFERENCES pipelines(id)
);

CREATE TABLE pipeline_steps (
    id VARCHAR(36) PRIMARY KEY,
    stage_id VARCHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    type VARCHAR(50) NOT NULL,
    configuration JSONB,
    order_index INT NOT NULL,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    output TEXT,
    error TEXT,
    FOREIGN KEY (stage_id) REFERENCES pipeline_stages(id)
);

CREATE TABLE infrastructures (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    type VARCHAR(50) NOT NULL,
    environment VARCHAR(50) NOT NULL,
    configuration JSONB,
    status VARCHAR(20) NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP
);
```

API Endpoints

- `POST /api/deployments`: Create deployment
- `GET /api/deployments/{deploymentId}`: Get deployment details
- `POST /api/deployments/{deploymentId}/start`: Start deployment
- `POST /api/configurations`: Create configuration
- `GET /api/configurations/{configurationId}`: Get configuration
- `POST /api/pipelines`: Create pipeline
- `POST /api/pipelines/{pipelineId}/execute`: Execute pipeline

Provider Service

The Provider Service integrates with multiple AI providers.

Key Classes

- **Provider**: Represents an AI provider
- **Model**: AI model from a provider
- **ProviderRequest**: Request to a provider
- **Tool**: Tool that can be used by AI
- **ToolExecution**: Execution of a tool

Database Schema

```
CREATE TABLE providers (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    type VARCHAR(50) NOT NULL,
    base_url VARCHAR(255) NOT NULL,
    api_keys JSONB,
    capabilities JSONB,
    enabled BOOLEAN NOT NULL,
    priority INT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP
);

CREATE TABLE models (
    id VARCHAR(36) PRIMARY KEY,
    provider_id VARCHAR(36) NOT NULL,
    name VARCHAR(255) NOT NULL,
    version VARCHAR(50),
    description TEXT,
    capabilities JSONB,
    max_context_size INT,
    parameter_constraints JSONB,
    input_token_cost DECIMAL(10, 6),
    output_token_cost DECIMAL(10, 6),
    enabled BOOLEAN NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP,
    FOREIGN KEY (provider_id) REFERENCES providers(id)
);

CREATE TABLE provider_requests (
    id VARCHAR(36) PRIMARY KEY,
    provider_id VARCHAR(36) NOT NULL,
    model_id VARCHAR(36) NOT NULL,
    user_id VARCHAR(36),
    request_type VARCHAR(50) NOT NULL,
    content TEXT NOT NULL,
    parameters JSONB,
    status VARCHAR(20) NOT NULL,
    created_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    response TEXT,
    error TEXT,
    token_count_input INT,
    token_count_output INT,
    cost DECIMAL(10, 6),
    FOREIGN KEY (provider_id) REFERENCES providers(id),
    FOREIGN KEY (model_id) REFERENCES models(id)
);

CREATE TABLE tools (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    type VARCHAR(50) NOT NULL,
    capabilities JSONB,
    parameter_schema JSONB,
    result_schema JSONB,
    implementation_class VARCHAR(255),
    enabled BOOLEAN NOT NULL,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP
);

CREATE TABLE tool_executions (
    id VARCHAR(36) PRIMARY KEY,
    tool_id VARCHAR(36) NOT NULL,
    request_id VARCHAR(36),
    parameters JSONB,
    status VARCHAR(20) NOT NULL,
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    result JSONB,
    error TEXT,
    FOREIGN KEY (tool_id) REFERENCES tools(id),
    FOREIGN KEY (request_id) REFERENCES provider_requests(id)
);
```

API Endpoints

- `POST /api/providers`: Register provider
- `GET /api/providers/{providerId}`: Get provider details
- `POST /api/providers/{providerId}/models`: Register model
- `GET /api/providers/{providerId}/models`: Get provider models
- `POST /api/requests`: Send request to provider
- `GET /api/requests/{requestId}`: Get request details
- `POST /api/tools`: Register tool
- `POST /api/tools/{toolId}/execute`: Execute tool

Governance Service

The Governance Service ensures ethical operation of Lumina AI.

Key Classes

- **GovernancePolicy**: Defines governance policy
- **ContentEvaluation**: Evaluation of content
- **GovernanceAudit**: Audit of governance decisions
- **UserConsent**: User consent record

- **TransparencyRecord**: Record of AI operation

Database Schema

```
CREATE TABLE governance_policies (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    type VARCHAR(50) NOT NULL,
    scope VARCHAR(50) NOT NULL,
    policy_definition TEXT NOT NULL,
    enforcement_rules JSONB,
    applicable_regions JSONB,
    enabled BOOLEAN NOT NULL,
    priority INT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    created_by VARCHAR(36) NOT NULL,
    updated_at TIMESTAMP,
    updated_by VARCHAR(36)
);

CREATE TABLE content_evaluations (
    id VARCHAR(36) PRIMARY KEY,
    request_id VARCHAR(36),
    user_id VARCHAR(36),
    model_id VARCHAR(36),
    provider_id VARCHAR(36),
    content_type VARCHAR(50) NOT NULL,
    content TEXT NOT NULL,
    safety_score DECIMAL(4, 3) NOT NULL,
    privacy_score DECIMAL(4, 3) NOT NULL,
    transparency_score DECIMAL(4, 3) NOT NULL,
    flags JSONB,
    evaluation_details TEXT,
    evaluated_at TIMESTAMP NOT NULL
);

CREATE TABLE governance_audits (
    id VARCHAR(36) PRIMARY KEY,
    policy_id VARCHAR(36) NOT NULL,
    request_id VARCHAR(36),
    user_id VARCHAR(36),
    action VARCHAR(50) NOT NULL,
    decision VARCHAR(50) NOT NULL,
    reason TEXT,
    details JSONB,
    created_at TIMESTAMP NOT NULL,
    FOREIGN KEY (policy_id) REFERENCES governance_policies(id)
);

CREATE TABLE user_consents (
    id VARCHAR(36) PRIMARY KEY,
    user_id VARCHAR(36) NOT NULL,
    consent_type VARCHAR(50) NOT NULL,
    consent_details JSONB,
    source VARCHAR(255),
    revoked BOOLEAN NOT NULL DEFAULT FALSE,
    revocation_reason TEXT,
    granted_at TIMESTAMP NOT NULL,
    revoked_at TIMESTAMP,
    previous_consent_id VARCHAR(36)
);

CREATE TABLE transparency_records (
    id VARCHAR(36) PRIMARY KEY,
    request_id VARCHAR(36),
    user_id VARCHAR(36),
    model_id VARCHAR(36),
    operation VARCHAR(255) NOT NULL,
    details JSONB,
    explanation TEXT,
    factors JSONB,
    sources JSONB,
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP
);
```

API Endpoints

- `POST /api/governance/policies`: Create policy
- `GET /api/governance/policies/{policyId}`: Get policy details
- `POST /api/governance/evaluate`: Evaluate content
- `GET /api/governance/evaluations/{evaluationId}`: Get evaluation
- `POST /api/governance/consents`: Record user consent
- `GET /api/governance/consents/{userId}`: Get user consents
- `POST /api/governance/transparency`: Create transparency record
- `GET /api/governance/transparency/{userId}`: Get user records

Testing Strategy

Lumina AI implements a comprehensive testing strategy:

Unit Testing

- **Service Layer Tests**: Tests for service classes
- **Controller Layer Tests**: Tests for REST controllers
- **Repository Layer Tests**: Tests for data access
- **Utility Class Tests**: Tests for utility functions
- **Mock Dependencies**: Uses mocks for dependencies

Example Unit Test:

```
@SpringBootTest
public class DeploymentServiceTest {

    @Mock
    private DeploymentRepository deploymentRepository;

    @InjectMocks
    private DeploymentService deploymentService;
    private Deployment testDeployment;
    private final String deploymentId = UUID.randomUUID().toString();

    @BeforeEach
    public void setup() {
        MockitoAnnotations.openMocks(this);

        testDeployment = new Deployment();
        testDeployment.setId(deploymentId);
        testDeployment.setName("Test Deployment");
        testDeployment.setDescription("Test Deployment Description");
        testDeployment.setStatus(DeploymentStatus.CREATED);
        testDeployment.setEnvironment("development");
        testDeployment.setCreatedAt(LocalDateTime.now());
        testDeployment.setCreatedBy("test-user");
    }

    @Test
    public void testCreateDeployment() {
        when(deploymentRepository.save(any(Deployment.class))).thenReturn(testDeployment);

        Deployment createdDeployment = deploymentService.createDeployment(testDeployment);

        assertNotNull(createdDeployment);
        assertEquals(testDeployment.getName(), createdDeployment.getName());
        assertEquals(testDeployment.getDescription(), createdDeployment.getDescription());
        assertEquals(DeploymentStatus.CREATED, createdDeployment.getStatus());
        verify(deploymentRepository, times(1)).save(any(Deployment.class));
    }

    // Additional tests...
}
```

Integration Testing

- **API Endpoint Tests**: Tests for API endpoints
- **Database Integration**: Tests with actual database
- **Service Interaction**: Tests service interactions
- **External Service Mocking**: Mocks external services
- **Test Containers**: Uses containers for dependencies

Performance Testing

- **Load Testing**: Tests under expected load
- **Stress Testing**: Tests under extreme load
- **Endurance Testing**: Tests over extended periods
- **Spike Testing**: Tests with sudden load increases
- **Scalability Testing**: Tests scaling capabilities

Security Testing

- **Penetration Testing**: Identifies vulnerabilities
- **Static Analysis**: Analyzes code for security issues
- **Dependency Scanning**: Checks dependencies for vulnerabilities

- **Authentication Testing**: Tests authentication mechanisms
- **Authorization Testing**: Tests access control

Current Status and Future Work

Completed Components

1. **Collaboration Service**: Fully implemented with dynamic team formation, shared context management, and negotiation protocols
2. **Workflow Service**: Complete implementation with workflow definition, execution engine, and monitoring
3. **Deployment Service**: Implemented with configuration management, pipeline engine, and API gateway
4. **Provider Service**: Implemented with unified API, dynamic selection, and fallback mechanisms
5. **Governance Service**: Implemented with policy management, content evaluation, and transparency records
6. **Documentation**: Comprehensive documentation for all components
7. **Test Suite**: Unit tests for all service and controller layers

Remaining Work

1. **Integration Testing**: End-to-end testing across services
2. **Performance Optimization**: Profiling and optimization for high-throughput scenarios
3. **Deployment Automation**: CI/CD pipeline for automated testing and deployment
4. **Monitoring Infrastructure**: Comprehensive monitoring and alerting
5. **User Interface**: Admin dashboard for system management and monitoring
6. **Extended Provider Support**: Integration with additional AI providers
7. **Advanced Analytics**: Usage analytics and performance metrics dashboard

Technical Assessment

Lumina AI represents a sophisticated AI platform with enterprise-grade features. The architecture follows modern best practices with microservices, clear separation of concerns, and robust API design. The system's strengths include:

1. **Flexibility**: The provider-agnostic approach allows for easy integration of new AI models
2. **Robustness**: Comprehensive error handling and fallback mechanisms
3. **Governance**: Strong ethical controls with regional compliance
4. **Scalability**: Architecture designed for horizontal scaling
5. **Extensibility**: Clear extension points for new capabilities

The implementation is production-ready for core functionality, with the remaining work focused on operational aspects, extended capabilities, and user interfaces rather than fundamental architectural changes.

Deployment Guide

Prerequisites

- Docker and Docker Compose
- Kubernetes cluster (for production)
- PostgreSQL database
- MongoDB instance
- Pinecone account
- Apache Kafka cluster

Development Environment Setup

1. Clone the repository:

```
git clone https://github.com/kimhons/lumina-ai.git
cd lumina-ai
```

2. Set up environment variables:

```
cp .env.example .env
# Edit .env with your configuration
```

3. Start development environment:

```
docker-compose -f docker-compose.dev.yml up -d
```

4. Build and run services:

```
./gradlew bootRun
```

Production Deployment

1. Build Docker images:

```
./gradlew jibDockerBuild
```

2. Deploy to Kubernetes:

```
kubectl apply -f kubernetes/
```

3. Configure ingress:

```
kubectl apply -f kubernetes/ingress.yml
```

4. Verify deployment:

```
kubectl get pods
```

API Reference

Collaboration Service API

Teams

- `POST /api/teams`
  - Create a new team
  - Request Body: Team object
  - Response: Created Team
- `GET /api/teams/{teamId}`
  - Get team details
  - Path Parameters: teamId
  - Response: Team object

Shared Context

- `POST /api/contexts`
  - Create a shared context
  - Request Body: SharedContext object
  - Response: Created SharedContext
- `GET /api/contexts/{contextId}`
  - Get context details
  - Path Parameters: contextId
  - Response: SharedContext with latest version

Workflow Service API

Workflow Definitions

- `POST /api/workflows/definitions`
  - Create workflow definition
  - Request Body: WorkflowDefinition object
  - Response: Created WorkflowDefinition
- `GET /api/workflows/definitions/{definitionId}`
  - Get definition details
  - Path Parameters: definitionId
  - Response: WorkflowDefinition object

Workflow Instances

- `POST /api/workflows/instances`
  - Start workflow instance
  - Request Body: { definitionId, input }
  - Response: Created WorkflowInstance
- `GET /api/workflows/instances/{instanceId}`
  - Get instance details
  - Path Parameters: instanceId
  - Response: WorkflowInstance with steps

Provider Service API

Providers

- `POST /api/providers`
  - Register provider
  - Request Body: Provider object
  - Response: Created Provider
- `GET /api/providers/{providerId}`
  - Get provider details
  - Path Parameters: providerId
  - Response: Provider object

Models

- `GET /api/providers/{providerId}/models`
  - Get provider models
  - Path Parameters: providerId
  - Response: List of Model objects
- `POST /api/requests`
  - Send request to provider
  - Request Body: { providerId, modelId, content, parameters }
  - Response: Created ProviderRequest

Governance Service API

Policies

- `POST /api/governance/policies`
  - Create policy
  - Request Body: GovernancePolicy object
  - Response: Created GovernancePolicy
- `GET /api/governance/policies/{policyId}`

    - Get policy details
    - Path Parameters: policyId
    - Response: GovernancePolicy object

Content Evaluation

- `POST /api/governance/evaluate`
    - Evaluate content
    - Request Body: { content, contentType, requestId, userId, modelId, providerId }
    - Response: ContentEvaluation object