SAPIENZA
UNIVERSITÀ DI ROMA

Design and Implementation of an Enterprise-Grade AI Agent Orchestration Platform: The Cineca Agentic Platform

Facoltà di Ingegneria dell'Informazione,
Informatica e Statistica
Master of Science in Data Science

Arman Feili
ID number 2101835

Advisor	Co-Advisors
Prof. Marco Raoul Marini	Dr. Valerio Venanzi
	Dr. Giuseppe Melfi
	Dr. Marco Puccini

Academic Year 2025/2026

Thesis not yet defended

Design and Implementation of an Enterprise-Grade AI Agent Orchestration Platform: The Cineca Agentic Platform
Master's Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Website: https://github.com/armanfeili

Author's email: feili.2101835@studenti.uniroma1.it

Abstract

The rapid advancement of Large Language Models (LLMs) has opened new possibilities for building intelligent software agents capable of reasoning, planning, and executing complex tasks. However, deploying AI agents in enterprise environments presents significant challenges, including multi-tenancy requirements, security and governance needs, reliability concerns, and integration with existing infrastructure.

This thesis presents the design and implementation of the Cineca Agentic Platform, an enterprise-grade AI agent orchestration system developed in collaboration with CINECA, Italy's national supercomputing center. The platform addresses the gap between research-oriented agent frameworks and production-ready enterprise systems by providing a comprehensive solution for agent orchestration, tool integration, and knowledge graph querying.

The key contributions of this work include: (1) a novel three-layer architecture separating core backend services, data persistence, and presentation concerns; (2) a native implementation of the Model Context Protocol (MCP) with 34 tools across 17 categories; (3) a Natural Language to Cypher pipeline enabling intuitive graph database querying with multi-layer safety validation; (4) a comprehensive security framework featuring OpenID Connect (OIDC) and JSON Web Token (JWT) authentication, role-based access control (RBAC), multi-tenancy isolation, and personally identifiable information (PII) protection; (5) a production-ready observability stack with Prometheus metrics, OpenTelemetry tracing, and structured logging; and (6) an asynchronous job system supporting long-running agent tasks with real-time progress streaming via Server-Sent Events (SSE).

The platform is implemented using Python and FastAPI for the backend, PostgreSQL and Redis for data management, Memgraph for graph storage, and dual frontends built with Next.js and Streamlit. In the evaluation scenarios described in Chapter 12, the system demonstrates capability to handle enterprise workloads while maintaining security, reliability, and extensibility.

This work contributes to the emerging field of enterprise AI agent systems by providing a reference architecture and open-source implementation that bridges the gap between academic agent research and industrial deployment requirements.

Contents

Chapter 1

Introduction

1.1 Motivation and Context

The landscape of artificial intelligence has undergone a transformative shift with the advent of Large Language Models (LLMs). Models such as GPT-4 [1], LLaMA [2], and their successors have demonstrated unprecedented capabilities in natural language understanding, reasoning, and generation. These advances have catalyzed a new paradigm in software development: the construction of *AI agents*—autonomous software entities capable of perceiving their environment, reasoning about goals, and executing multi-step actions to accomplish complex tasks [5].

1.1.1 The Rise of AI Agents in Enterprise Applications

The progression from simple chatbots to sophisticated AI agents represents a fundamental evolution in how organizations leverage artificial intelligence. Unlike traditional conversational interfaces that merely respond to queries, AI agents can:

- **Plan and decompose** complex tasks into manageable sub-tasks
- **Invoke external tools** to gather information or execute actions
- **Reason about intermediate results** and adapt their approach
- **Maintain context** across multi-turn interactions
- **Query structured knowledge** from databases and knowledge graphs

This capability has driven significant interest in deploying AI agents across enterprise domains, from customer service and document processing to scientific research and data analysis. Organizations increasingly seek to augment human expertise with AI systems that can autonomously handle routine inquiries, synthesize information from multiple sources, and provide actionable insights.

1.1.2 The High-Performance Computing Context

Research institutions and supercomputing centers face unique challenges in managing computational resources, scientific workflows, and the vast amounts of data generated by modern research. High-Performance Computing (HPC) environments, such as those operated by national supercomputing centers, support diverse user communities spanning bioinformatics, physics simulation, climate modeling, and many other domains.

In these environments, researchers and operators must navigate:

- Complex job scheduling systems with intricate resource allocation policies
- Heterogeneous software stacks spanning multiple programming languages and frameworks
- Regulatory compliance requirements for data privacy and research integrity
- Multi-institutional collaborations requiring fine-grained access control
- The need to rapidly onboard new users and adapt to evolving research requirements

AI agents offer a compelling solution for many of these challenges by providing intuitive interfaces that abstract away underlying complexity while maintaining the precision and auditability that research environments demand.

1.1.3 CINECA: Italy's National Supercomputing Center

CINECA (Consorzio Interuniversitario del Nord Est per il Calcolo Automatico) [19] serves as Italy's premier supercomputing consortium, providing computational infrastructure and services to universities, research institutions, and industrial partners across the nation and beyond. Founded in 1969, CINECA has evolved to become one of Europe's leading HPC centers, operating some of the world's most powerful supercomputers and supporting thousands of researchers across diverse scientific disciplines.

The organization's mission encompasses:

- Providing world-class computational resources for scientific research
- Developing and maintaining software tools for data management and analysis
- Supporting bioinformatics and life sciences research initiatives
- Enabling secure, multi-tenant access for diverse user communities
- Advancing the state of the art in high-performance computing applications

Within this context, there exists a compelling need for intelligent systems that can assist researchers in navigating complex computational workflows, querying interconnected research data, and managing resources efficiently. The intersection of AI agent capabilities with HPC requirements presents both significant opportunities and substantial technical challenges.

1.1.4 The Gap Between Research and Production

Despite the rapid proliferation of AI agent frameworks and tools, a significant gap persists between research-oriented prototypes and production-ready enterprise systems. Popular frameworks such as LangChain [4] and AutoGPT [6] have demonstrated impressive capabilities in controlled settings, yet deploying these systems in enterprise environments reveals critical limitations:

1. **Security and Governance**: Research frameworks often lack comprehensive authentication, authorization, and audit capabilities required for enterprise deployments.
2. **Multi-Tenancy**: Most existing tools assume single-tenant deployments, making them unsuitable for organizations serving multiple user communities with distinct access requirements.
3. **Reliability and Observability**: Production systems require robust monitoring, health checks, and graceful degradation—capabilities rarely prioritized in research-focused implementations.
4. **Tool Integration Standards**: The absence of standardized protocols for tool integration leads to fragmented ecosystems and vendor lock-in.
5. **Knowledge Graph Integration**: While many frameworks support document retrieval, few provide native integration with graph databases for structured knowledge querying.

This thesis addresses these gaps by presenting the design and implementation of an enterprise-grade AI agent orchestration platform developed in collaboration with CINECA. The platform bridges the divide between academic agent research and industrial deployment requirements, providing a reference architecture for organizations seeking to deploy AI agents in production environments.

1.1.5 The Model Context Protocol Opportunity

The emergence of the Model Context Protocol (MCP) [7] presents a timely opportunity to address the tool integration challenge. MCP provides a standardized specification for how AI systems discover and invoke external tools, enabling interoperability across different agent implementations and reducing the fragmentation that has plagued the ecosystem.

By adopting MCP principles, the Cineca Agentic Platform establishes a foundation for extensible tool integration that can evolve with the rapidly advancing field of AI agents. This protocol-first approach ensures that investments in tool development remain valuable as the underlying AI technologies continue to improve.

1.1.6 Chapter Summary

This chapter has established the motivation for developing an enterprise-grade AI agent orchestration platform by:

- Describing the transformative potential of LLM-powered AI agents
- Contextualizing the unique requirements of HPC and research environments
- Introducing CINECA as the institutional setting for this work
- Identifying the critical gaps between research frameworks and production requirements
- Highlighting the opportunity presented by emerging tool integration standards

The following sections of this introductory chapter will elaborate on specific use cases, stakeholder requirements, the formal problem statement, research objectives, and the contributions of this thesis.

1.2 Primary Use Cases and Stakeholders

The Cineca Agentic Platform is designed to serve diverse user communities with varying technical expertise and operational responsibilities. This section identifies the primary stakeholders and articulates the core use cases that drive the platform's design.

1.2.1 Stakeholder Analysis

The platform addresses the needs of four primary stakeholder groups, each with distinct requirements and interaction patterns.

End Users (Researchers and Analysts)

End users represent the primary consumers of the platform's AI agent capabilities. This stakeholder group includes:

- **Researchers** who need to query complex datasets and knowledge graphs without deep technical expertise in query languages
- **Data analysts** seeking to synthesize information across multiple sources and generate insights
- **Domain experts** who require natural language interfaces to interact with computational systems
- **Students and trainees** learning to navigate research infrastructure

Key Requirements:

- Intuitive, conversational interfaces for complex queries
- Clear explanations of how answers were derived
- Reliable, accurate responses grounded in authoritative data sources
- Reasonable response times for interactive use
- Access controls that respect data sensitivity and user authorization

Administrators and Operators

Platform administrators and operators are responsible for the day-to-day management and oversight of the system. This group includes:

- **System administrators** managing infrastructure, deployments, and configurations
- **Tenant administrators** responsible for their organization's users and resources
- **Operations engineers** monitoring system health and responding to incidents
- **Support staff** assisting end users and troubleshooting issues

Key Requirements:

- Comprehensive dashboards for monitoring system health and performance
- Tools for managing users, tenants, and access permissions
- Ability to configure LLM providers and model defaults
- Access to audit logs for compliance and troubleshooting
- Mechanisms for controlling resource consumption and enforcing quotas

Security and Compliance Officers

Security and compliance stakeholders ensure that the platform meets organizational and regulatory requirements. This group includes:

- **Security officers** responsible for threat assessment and security policy
- **Compliance managers** ensuring adherence to data protection regulations
- **Auditors** requiring evidence of proper access control and data handling
- **Privacy officers** concerned with personally identifiable information (PII) protection

Key Requirements:

- Robust authentication using industry-standard protocols (OpenID Connect/JSON Web Tokens)
- Fine-grained authorization with role-based access control (RBAC)
- Complete audit trails for all significant operations
- PII detection and redaction capabilities
- Multi-tenant isolation guaranteeing data segregation
- Rate limiting to prevent abuse and ensure fair resource allocation

Developers and Integrators

Developers and integrators extend the platform's capabilities and integrate it with other systems. This group includes:

- **Platform developers** building new features and tools
- **Integration engineers** connecting the platform to external services
- **Tool authors** creating domain-specific capabilities
- **DevOps engineers** managing deployment pipelines and infrastructure

Key Requirements:

- Well-documented APIs with comprehensive OpenAPI specifications
- Extensible tool framework with clear integration patterns
- Comprehensive test suites and development fixtures
- Support for multiple LLM providers with consistent interfaces
- Deployment flexibility across different infrastructure configurations

1.2.2 Core Use Cases

Based on stakeholder analysis and requirements gathering, the platform addresses five primary use cases.

UC1: Conversational Agent Interaction (Chat)

Description: End users engage in natural language conversations with AI agents to obtain information, perform tasks, or receive guidance. The agent maintains conversational context across multiple turns and can invoke tools as needed.

Actors: End Users

Flow:

1. User authenticates and accesses the chat interface
2. User enters a natural language prompt or question
3. System classifies intent and routes to appropriate processing pipeline
4. Agent reasons about the query, potentially invoking tools or querying data sources
5. Agent generates a response with supporting evidence
6. User reviews response and may continue the conversation
7. All interactions are logged for audit purposes

Success Criteria:

- Response accuracy meets domain-specific quality thresholds
- Response latency remains within acceptable bounds (typically under 30 seconds for complex queries)
- Conversational context is maintained across session turns

UC2: Knowledge Graph Question Answering (Graph Q&A)

Description: Users query structured knowledge graphs using natural language, with the system automatically translating queries into Cypher and executing them against the graph database.

Actors: End Users, Researchers

Flow:

1. User submits a natural language question about entities or relationships
2. System normalizes the query and extracts key entities
3. Natural Language to Cypher (NL-to-Cypher) pipeline generates a Cypher query
4. Safety validation ensures query compliance (read-only, tenant-scoped, bounded)
5. Query executes against Memgraph graph database
6. Results are summarized into a natural language response
7. Query, results, and summary are logged for audit

Success Criteria:

- Generated Cypher queries are syntactically correct and semantically appropriate
- Safety validation rejects all unauthorized or potentially harmful queries
- Results accurately reflect the underlying graph data
- Response includes appropriate citations to source entities

UC3: Long-Running Background Jobs

Description: Users and systems submit tasks that require extended processing times, receiving real-time progress updates through event streaming.

Actors: End Users, Administrators, Automated Systems

Flow:

1. Client submits a job request with type and parameters
2. System validates request and creates job record with unique identifier
3. Job enters queue for worker processing
4. Worker picks up job and begins execution
5. Progress events are streamed via Server-Sent Events (SSE)
6. On completion, final results are persisted and client notified
7. Job lifecycle events are recorded for audit

Success Criteria:

- Jobs execute reliably without data loss
- Progress updates reflect actual execution state
- Cancellation requests are honored promptly
- Failed jobs provide actionable error information

UC4: System Monitoring and Administration

Description: Administrators monitor system health, manage configurations, and perform operational tasks through dedicated management interfaces.

Actors: Administrators, Operators

Flow:

1. Administrator authenticates with elevated privileges
2. Dashboard displays real-time health metrics and KPIs
3. Administrator reviews component status (databases, LLM providers, workers)
4. Administrator performs management actions (tenant configuration, model updates, etc.)
5. Changes are validated and applied with appropriate logging
6. System confirms successful configuration changes

Success Criteria:

- Health information is accurate and current
- Configuration changes take effect without system restart where possible
- Administrative actions are fully audited
- Role-based permissions prevent unauthorized operations

UC5: Tool Discovery and Invocation

Description: Users and agents discover available tools, understand their capabilities, and invoke them with appropriate parameters.

Actors: End Users, Agents, Developers

Flow:

1. Client requests list of available tools
2. System returns tool metadata filtered by caller's permissions
3. Client retrieves detailed schema for specific tool
4. Client constructs invocation request with validated parameters
5. System authorizes invocation based on RBAC and tool-specific policies
6. Tool executes and returns structured results
7. Invocation details are logged for audit

Success Criteria:

- Tool discovery returns only authorized tools
- Schema validation prevents invalid invocations
- Tool execution respects resource limits and timeouts
- Results conform to documented output schemas

1.2.3 Use Case Prioritization

Table 1.1 summarizes the relative priority and frequency of each use case, informing architectural decisions throughout the platform design.

Table 1.1. Use case prioritization matrix.

Use Case	Primary Stakeholder	Priority	Frequency
UC1: Chat	End Users	High	Very High
UC2: Graph Q&A	End Users, Researchers	High	High
UC3: Background Jobs	All	Medium	Medium
UC4: Monitoring/Admin	Administrators	High	Medium
UC5: Tool Invocation	Agents, Developers	High	Very High

1.2.4 User Interface Mapping

The platform provides two distinct user interfaces to address the diverse needs identified in stakeholder analysis:

Agent Chat UI (Next.js) A modern, responsive web application targeting end users for conversational interactions. Built with Next.js, React, and TypeScript, this interface provides real-time message display, step-by-step execution visualization, and model selection capabilities.

Control Panel UI (Streamlit) A comprehensive administrative dashboard targeting operators and administrators. Built with Streamlit for rapid development and Python integration, this interface provides system monitoring, job management, tool exploration, and configuration capabilities.

This dual-interface strategy ensures that each stakeholder group receives an experience optimized for their specific workflows and technical expectations.

1.3 Industrial and Institutional Context

This thesis emerges from a collaboration between Sapienza Università di Roma and CINECA (see Section 1.3 for institutional details). This section elaborates on the institutional context, constraints, and opportunities that shaped the platform's design.

1.3.1 CINECA Organizational Profile

CINECA (Consorzio Interuniversitario del Nord Est per il Calcolo Automatico) was established in 1969 as an interuniversity consortium for automatic computation. Today, it operates as Italy's primary high-performance computing center, serving:

- Over 70 Italian universities and research institutions as consortium members
- Thousands of researchers across physics, chemistry, life sciences, engineering, and social sciences
- European research collaborations through PRACE (Partnership for Advanced Computing in Europe)
- Public administration entities requiring computational services
- Industrial partners seeking HPC capabilities for research and development

CINECA operates several of Europe's most powerful supercomputers, including systems ranked among the world's top 500, and maintains extensive infrastructure for data storage, networking, and scientific software.

1.3.2 High-Performance Computing Environment Characteristics

HPC environments present distinctive characteristics that influence software architecture decisions:

Resource Heterogeneity: HPC centers manage diverse computational resources including traditional CPU clusters, GPU-accelerated systems, specialized hardware for specific scientific applications, and high-capacity storage systems. Software must accommodate this heterogeneity gracefully.

Job Scheduling Complexity: Computational resources are allocated through sophisticated scheduling systems (such as SLURM, PBS, or LSF) that balance competing priorities, enforce resource quotas, and optimize utilization. Users interact with these systems through command-line interfaces that can present substantial learning curves.

Data Sensitivity Spectrum: Research data spans a wide sensitivity spectrum, from publicly available datasets to highly restricted clinical or defense-related information. Access control must be precise and auditable.

Multi-Institutional Collaboration: Research projects frequently involve collaborators from multiple institutions, each with their own authentication systems, policies, and compliance requirements. Federated identity and fine-grained access control are essential.

Long-Running Workflows: Scientific computations often run for hours, days, or even weeks. Systems must support long-lived operations with checkpointing, resumption, and progress tracking capabilities.

1.3.3 Institutional Constraints

The collaboration with CINECA introduced several constraints that directly influenced platform architecture:

Security and Compliance Requirements

As a national research infrastructure, CINECA must comply with European data protection regulations (GDPR), national security requirements, and research ethics standards. The platform must:

- Implement strong authentication using institutional identity providers
- Provide complete audit trails for all data access and system operations
- Support data residency requirements (data must remain within European jurisdiction)
- Enable compliance with sector-specific regulations (e.g., health data protection)

Multi-Tenancy Mandate

CINECA serves diverse user communities that must be logically isolated from one another:

- Different universities may require separate tenant environments
- Research projects may need dedicated spaces with project-specific configurations
- Industrial partners require isolation from academic users
- Administrative boundaries must be enforceable and auditable

The platform's multi-tenancy model, with tenant-scoped data isolation and per-tenant configuration defaults, directly addresses this constraint.

Infrastructure Integration Requirements

The platform must integrate with existing CINECA infrastructure:

- Identity management systems (LDAP, SAML, institutional IdPs via OIDC)
- Existing databases and data warehouses
- Monitoring and alerting systems (Prometheus ecosystem)
- Container orchestration platforms (Docker, Kubernetes)
- Network security policies and firewall configurations

Operational Excellence Expectations

As a production service supporting research activities, the platform must meet operational excellence standards:

- High availability with defined Service Level Objectives (SLOs) for uptime and response time
- Comprehensive monitoring with proactive alerting
- Graceful degradation under component failures
- Clear operational procedures and runbooks
- Capacity for horizontal scaling to meet demand

1.3.4 Bioinformatics Application Domain

While the platform is designed as a general-purpose agent orchestration system, initial deployment focuses on bioinformatics applications within CINECA's life sciences computing initiatives. This domain presents specific requirements:

Knowledge Graph Queries: Researchers need to query interconnected biological data—genes, proteins, pathways, diseases, publications, and researchers—using natural language. The graph database integration with NL-to-Cypher capabilities directly serves this need.

Workflow Orchestration: Bioinformatics analyses typically involve multi-step pipelines with dependencies. The job system with progress streaming supports these long-running workflows.

Data Privacy: Clinical and genetic data require strict access controls and audit logging. The security framework addresses these requirements.

Reproducibility: Scientific analyses must be reproducible. Complete logging of agent reasoning, tool invocations, and data access supports this requirement.

1.3.5 Enterprise Context Beyond CINECA

While developed for CINECA, the platform architecture addresses challenges common to many enterprise environments:

Financial Services: Multi-tenant access controls, audit requirements, and data protection needs mirror those in banking and insurance.

Healthcare: Patient data privacy, regulatory compliance, and multi-institutional collaboration requirements align with healthcare IT challenges.

Manufacturing: Integration with existing enterprise systems, long-running processes, and operational monitoring match manufacturing IT needs.

Government: Security classifications, multi-agency collaboration, and compliance requirements parallel government IT environments.

The architectural patterns and implementation strategies presented in this thesis thus have applicability beyond the specific CINECA context, providing a reference for enterprise AI agent deployments across sectors.

1.3.6 Technology Environment

The platform operates within a technology environment characterized by:

- **Container-Native Deployment:** All services are containerized for deployment on Docker and Kubernetes platforms.
- **Python Ecosystem:** The backend leverages Python's rich ecosystem for machine learning, data processing, and web services.
- **Open Source Foundation:** Core infrastructure components (PostgreSQL, Redis, Memgraph) are open-source, avoiding vendor lock-in.

- **Cloud-Agnostic Design:** While deployable on cloud platforms, the architecture avoids dependencies on specific cloud provider services.
- **Local LLM Support:** Integration with Ollama enables deployment of language models on local infrastructure, addressing data sovereignty concerns.

This technology environment reflects both CINECA's operational preferences and broader industry trends toward open, portable, and privacy-respecting AI infrastructure.

1.4 Problem Statement

The development of production-ready AI agent systems for enterprise environments presents a complex set of interrelated challenges. This section formally articulates the problem space that the Cineca Agentic Platform addresses.

1.4.1 The Production Deployment Gap

Problem Statement: *Existing AI agent frameworks, while demonstrating impressive capabilities in research and prototyping contexts, lack the comprehensive security, governance, observability, and operational features required for deployment in enterprise and research institution environments.*

This gap manifests across multiple dimensions, each presenting distinct technical challenges.

1.4.2 Challenge 1: Security and Access Control

Enterprise environments require sophisticated security controls that extend far beyond simple authentication. Specific challenges include:

Multi-Layer Authentication: Organizations typically employ federated identity systems with multiple identity providers. AI agent platforms must integrate seamlessly with these systems using standards like OpenID Connect (OIDC) while maintaining security across different trust domains.

Fine-Grained Authorization: Different users require different levels of access to data, tools, and operations. Role-Based Access Control (RBAC) must be comprehensive enough to express complex permission models while remaining manageable. The challenge intensifies when AI agents invoke tools on behalf of users—the agent's actions must respect the invoking user's permissions.

Tool-Level Security: MCP tools that access databases, invoke external services, or modify system state present security risks if not properly controlled. Each tool invocation must be authorized based on the caller's identity, the tool's sensitivity, and the specific operation being performed.

Audit and Compliance: Regulatory frameworks (GDPR, HIPAA, institutional policies) require complete audit trails of data access and system operations. Every agent interaction, tool invocation, and data query must be logged in tamper-evident fashion with sufficient context for compliance verification.

1.4.3 Challenge 2: Multi-Tenancy Requirements

Research institutions and enterprises typically serve multiple distinct user communities that must be isolated from one another:

Data Isolation: Each tenant's data must be invisible and inaccessible to other tenants. This isolation must be enforced at the database level, not merely at the application level, to prevent accidental or malicious data leakage.

Configuration Isolation: Different tenants may require different default configurations—preferred LLM models, rate limits, tool access policies, and operational parameters. The platform must support per-tenant customization while maintaining system-wide defaults.

Resource Isolation: Tenants should not be able to monopolize shared resources (API rate limits, LLM quotas, database connections) to the detriment of other tenants. Fair resource allocation requires explicit quota management.

Administrative Delegation: Organizations often delegate tenant administration to designated administrators within each tenant. The platform must support this administrative hierarchy without compromising cross-tenant isolation.

1.4.4 Challenge 3: Knowledge Graph Integration

While many AI agent frameworks support document retrieval through Retrieval-Augmented Generation (RAG), structured knowledge stored in graph databases presents different integration challenges:

Natural Language to Query Translation: Users expect to query knowledge graphs using natural language, yet graph databases require structured query languages (Cypher for property graphs). Translating natural language to correct, efficient, and safe graph queries is non-trivial.

Query Safety: Unlike read-only document retrieval, graph databases support mutations. AI-generated queries must be validated to ensure they:

- Contain only read operations (no writes or deletes)
- Respect tenant boundaries (queries cannot access other tenants' data)
- Have bounded execution time and result size (preventing denial of service)
- Use valid syntax and semantics

Result Synthesis: Graph query results are often complex structures (paths, aggregations, multiple related entities). These results must be synthesized into coherent natural language responses that accurately represent the underlying data.

1.4.5 Challenge 4: Operational Reliability

Production deployments require operational excellence that research prototypes rarely address:

Health Monitoring: Systems must expose comprehensive health information through Kubernetes-compatible probes (liveness, readiness, startup) and detailed component health endpoints. This information drives automated recovery and load balancing decisions.

Observability: The three pillars of observability—metrics, logs, and traces—must be implemented comprehensively:

- **Metrics:** Prometheus-compatible metrics for request rates, latencies, error rates, resource utilization, and domain-specific indicators
- **Logs:** Structured logging with correlation IDs enabling request tracing across service boundaries
- **Traces:** Distributed tracing with OpenTelemetry for end-to-end request visualization

Graceful Degradation: Component failures (database unavailability, LLM provider outages, network partitions) should not cause complete system failure. Circuit breakers, fallback strategies, and graceful degradation paths must be designed into the architecture.

Long-Running Operations: AI agent tasks may run for extended periods. The system must support:

- Asynchronous job submission with unique identifiers
- Real-time progress streaming
- Cooperative cancellation
- Worker heartbeats and failure recovery

1.4.6 Challenge 5: LLM Provider Management

Depending on a single LLM provider creates operational risk and limits flexibility:

Provider Abstraction: Applications should not be tightly coupled to specific LLM providers. A consistent interface must abstract provider-specific details while supporting diverse providers (OpenAI, Azure OpenAI, Ollama, HuggingFace, etc.).

Resilience: LLM providers experience outages, rate limiting, and performance degradation. The platform must implement:

- Circuit breakers to prevent cascade failures
- Provider fallback chains for automatic failover
- Retry strategies with exponential backoff
- Health monitoring per provider

Cost Management: LLM API calls incur costs that can accumulate rapidly. The platform should:

- Track token consumption and estimated costs
- Support budget limits and alerts
- Enable cost allocation to tenants and projects
- Optimize by caching responses where appropriate

Model Selection: Different tasks may require different models (faster models for simple tasks, more capable models for complex reasoning). The platform should support:

- Dynamic model selection based on task characteristics
- Per-tenant default model configuration
- Model capability matching (context length, function calling support)

1.4.7 Challenge 6: Tool Ecosystem Extensibility

A valuable AI agent platform must be extensible with domain-specific tools:

Standardized Tool Protocol: Tools should follow a consistent protocol (MCP) enabling discovery, schema validation, invocation, and result handling. This consistency reduces integration friction and improves maintainability.

Tool Authorization: Different tools have different sensitivity levels. The platform must support per-tool authorization policies that integrate with the overall RBAC system.

Tool Lifecycle: Tools must be discoverable, describable (via schemas), invokable, and auditable. The tool framework should support this complete lifecycle without requiring modifications to core platform code.

1.4.8 Problem Summary

The challenges described above are not merely technical inconveniences—they represent fundamental barriers to deploying AI agents in production enterprise environments. Existing frameworks typically address one or two of these challenges while leaving others unaddressed or delegated to implementers.

Table 1.2. Challenge coverage in existing frameworks versus requirements.

Challenge	LangChain	LlamaIndex	AutoGen	Semantic Kernel	Required
Security/RBAC	Partial	Partial	No	Partial	Full
Multi-Tenancy	No	No	No	No	Full
Graph Integration	Plugin	Plugin	No	No	Native
Observability	Plugin	Plugin	No	Partial	Native
Job System	No	No	No	No	Native
Provider Resilience	No	No	No	Partial	Native

This thesis addresses the problem of providing a comprehensive, integrated solution that addresses all six challenge areas while remaining extensible and maintainable. The solution must be production-ready—not a proof-of-concept—capable of supporting real workloads in the CINECA environment.

1.5 Research Objectives

This thesis pursues a primary objective supported by several sub-objectives that collectively address the challenges identified in Section 1.4. The objectives are formulated to be specific, measurable, and verifiable through the evaluation presented in Chapter 12.

1.5.1 Primary Objective

Primary Objective: *Design and implement an enterprise-grade AI agent orchestration platform that integrates comprehensive security, multi-tenancy, graph database querying, observability, and extensible tool capabilities into a production-ready system suitable for deployment at research institutions and enterprises.*

This objective encompasses the full lifecycle from architectural design through implementation to production deployment, with the Cineca Agentic Platform serving as the concrete realization.

1.5.2 Sub-Objectives

The primary objective decomposes into six sub-objectives, each addressing a specific challenge area.

SO1: Security and Governance Architecture

Objective: Design and implement a comprehensive security framework providing:

1. OIDC/JWT-based authentication with support for federated identity providers
2. Role-Based Access Control (RBAC) with scope-based permissions
3. Per-tool authorization policies integrated with the RBAC system
4. Comprehensive audit logging for compliance and forensics
5. PII detection and redaction capabilities
6. Intent-based filtering to block potentially dangerous operations

Success Criteria:

- All API endpoints enforce authentication and authorization
- Audit logs capture all significant operations with sufficient context
- PII scrubbing correctly identifies and redacts sensitive data patterns
- Security mechanisms do not introduce unacceptable latency overhead (<50ms per request)

SO2: Multi-Tenancy Implementation

Objective: Implement native multi-tenancy with:

1. Database-level tenant isolation for all data entities
2. Per-tenant configuration defaults (models, rate limits, policies)
3. Tenant-scoped API endpoints with header-based tenant identification
4. Administrative delegation within tenant boundaries
5. Cross-tenant data leakage prevention at all system layers

Success Criteria:

- Tenant A cannot access Tenant B's data through any API path
- Per-tenant configurations correctly override system defaults
- Tenant isolation is enforced at database query level (not just application level)
- Multi-tenant operations scale linearly with tenant count

SO3: Graph Database Integration with NL-to-Cypher

Objective: Develop a Natural Language to Cypher pipeline enabling:

1. Translation of natural language questions to syntactically correct Cypher queries
2. Multi-layer safety validation (syntax, read-only, tenant boundaries, resource limits)
3. Query execution against Memgraph graph database
4. Result summarization into natural language responses
5. Deterministic test mode for reliable pipeline testing

Success Criteria:

- Generated Cypher queries pass syntax validation at >95% rate
- Safety validation correctly rejects all mutation attempts
- Tenant filters are automatically injected into all generated queries
- Query execution respects configured timeout and result limits
- Test mode enables deterministic, LLM-independent testing

SO4: Operational Observability Stack

Objective: Implement comprehensive observability through:

1. Prometheus-compatible metrics covering HTTP requests, agent runs, jobs, tools, and LLM calls
2. Structured logging with request ID correlation across all components
3. OpenTelemetry distributed tracing for end-to-end request visualization
4. Kubernetes-compatible health probes (liveness, readiness, startup)
5. Component-level health checks with degradation detection

Success Criteria:

- All key operations are instrumented with timing and outcome metrics
- Log entries include correlation IDs enabling request tracing
- Trace spans correctly propagate across async boundaries
- Health probes accurately reflect system readiness
- Degraded states (e.g., provider unavailability) are correctly reported

SO5: LLM Provider Resilience and Management

Objective: Design an LLM provider management system featuring:

1. Provider abstraction supporting multiple LLM backends (OpenAI, Ollama, Azure)
2. Circuit breaker pattern for provider failure isolation
3. Automatic provider fallback with configurable chains
4. Cost tracking and budget management
5. Default Model Resolver with hierarchy (user → tenant → system)
6. Provider health monitoring and reporting

Success Criteria:

- Provider failures are isolated within circuit breaker timeouts
- Fallback chains automatically route to healthy providers
- Cost estimates are computed for all LLM calls
- Model defaults correctly resolve through the configuration hierarchy
- Provider health status is exposed via metrics and health endpoints

SO6: Extensible Tool Ecosystem

Objective: Develop an MCP-compatible tool framework supporting:

1. Tool registration with schema-based input/output validation
2. Tool discovery API with filtering by category and authorization
3. Per-tool RBAC with scope requirements
4. Comprehensive tool invocation logging
5. Straightforward patterns for adding new tools
6. Test fixtures for tool development

Success Criteria:

- Tool discovery returns only tools the caller is authorized to invoke
- Input validation rejects malformed invocation requests
- Tool invocations are fully logged with inputs, outputs, and timing
- New tools can be added by following documented patterns
- Tool test coverage exceeds 80% across all tool categories

1.5.3 Non-Objectives

To clarify the scope of this work, the following are explicitly *not* objectives:

- Advancing the state of the art in LLM architectures or training methods
- Developing novel natural language understanding algorithms
- Creating a general-purpose workflow orchestration engine
- Building a comprehensive RAG (Retrieval-Augmented Generation) library
- Replacing existing agent frameworks for all use cases

The focus remains on the integration, orchestration, and operationalization of existing AI capabilities within an enterprise-grade platform architecture.

1.5.4 Objective Traceability

Table 1.3 maps each sub-objective to the challenges it addresses and the evaluation sections where it is verified.

Table 1.3. Objective traceability matrix.

Sub-Objective	Addresses Challenge	Primary Chapter	Evaluation
SO1: Security	Challenge 1	Chapter 8	Section 12.4
SO2: Multi-Tenancy	Challenge 2	Chapter 8	Section 12.2
SO3: NL-to-Cypher	Challenge 3	Chapter 5	Section 12.2
SO4: Observability	Challenge 4	Chapter 9	Section 12.3
SO5: LLM Resilience	Challenge 5	Chapter 5	Section 12.5
SO6: Tool Ecosystem	Challenge 6	Chapter 6	Section 12.2

1.5.5 Validation Approach

Each objective will be validated through a combination of:

Functional Testing: Automated tests verifying that implemented features meet specifications. The test suite includes over 2,700 test functions covering all major subsystems.

Performance Benchmarking: Quantitative measurements of latency, throughput, and resource utilization under realistic workloads.

Security Assessment: Evaluation of authentication, authorization, and data protection mechanisms against defined threat scenarios.

Operational Evaluation: Assessment of monitoring, alerting, and operational procedures in production-like environments.

Comparative Analysis: Comparison with existing frameworks to demonstrate relative capabilities and limitations.

The evaluation chapter (Chapter 12) presents detailed results for each validation approach.

1.6 Scope and Non-Goals

Clearly delineating what falls within and outside the scope of this thesis is essential for understanding its contributions and limitations. This section explicitly defines the boundaries of the work.

1.6.1 In-Scope Elements

The following elements are within the scope of this thesis and are addressed through design, implementation, and evaluation:

Agent Orchestration Engine

- Multi-step agent run execution with planning and tool invocation
- Intent classification pipeline (pattern-based and LLM fallback)
- Session management for conversational context
- Agent run lifecycle management (pending $\rightarrow$ running $\rightarrow$ completed/failed/cancelled)
- Step-by-step execution recording for auditability
- Task planning for complex task decomposition

MCP Tool Framework

- Tool registry with 34 tools across 17 categories
- Tool discovery API with schema retrieval
- Tool invocation with input validation and output handling
- Per-tool RBAC and audit logging
- Patterns for tool extension and customization

Graph Database Integration

- Memgraph graph database integration
- NL-to-Cypher pipeline for natural language querying
- Multi-layer safety validation for generated queries
- Graph schema management and introspection
- Result summarization for user-friendly responses

Security Framework

- OIDC/JWT authentication with JWKS key management
- Role-Based Access Control with scope-based permissions
- Multi-tenant data isolation at database level
- Rate limiting with sliding window algorithm
- PII detection and redaction
- Comprehensive audit logging

Observability Stack

- Prometheus metrics for all major subsystems
- Structured logging with correlation IDs
- OpenTelemetry distributed tracing
- Kubernetes-compatible health probes
- Component health monitoring

Background Job System

- Asynchronous job queue with Redis backend
- Worker processes with heartbeat monitoring
- SSE-based progress streaming
- Job cancellation and graceful shutdown
- Scheduled background tasks (APScheduler)

User Interfaces

- Next.js Agent Chat UI for end-user interactions
- Streamlit Control Panel for administration
- API client implementations in TypeScript and Python

Deployment Infrastructure

- Docker Compose configuration for development and production
- NGINX reverse proxy configuration
- Prometheus and Grafana monitoring setup
- Database initialization and migration scripts

1.6.2 Out-of-Scope Elements (Non-Goals)

The following elements are explicitly **not** within the scope of this thesis:

Advanced RAG Pipeline

While the platform supports graph-based knowledge querying through NL-to-Cypher, it does **not** implement:

- Document ingestion and chunking pipelines
- Vector embedding generation and storage
- Semantic similarity search over unstructured documents
- Hybrid retrieval combining vector and keyword search

Rationale: The focus is on structured knowledge in graph form, which is the primary data representation at CINECA. Document RAG is a complementary capability that could be added in future work but is not essential for the initial deployment context.

Multi-Agent Collaboration

The platform implements single-agent orchestration but does **not** include:

- Agent-to-agent communication protocols
- Multi-agent task delegation and coordination
- Hierarchical agent architectures (manager/worker patterns)
- Agent negotiation and conflict resolution

Rationale: Multi-agent systems introduce significant complexity in coordination, debugging, and security. The single-agent architecture sufficiently addresses current requirements while providing a foundation for future multi-agent extensions.

Advanced Agent Architectures

The platform implements the ReAct (Reasoning + Acting) pattern but does **not** include:

- Plan-and-execute architectures with separate planning and execution phases
- Tree-of-thought reasoning with branching exploration
- Self-reflection and critique loops
- Constitutional AI safety patterns

Rationale: The ReAct pattern provides a solid foundation for tool-using agents. More sophisticated architectures could be implemented within the existing framework but are not required for the current use cases.

Real-Time Streaming Responses

The platform supports Server-Sent Events (SSE) for job progress but does **not** implement:

- Token-by-token LLM response streaming to users
- WebSocket-based bidirectional communication
- Real-time collaborative editing features

Rationale: The current polling-based approach provides adequate user experience for the target use cases. Streaming would improve perceived responsiveness but adds implementation complexity that was deprioritized.

Code Execution Sandbox

The platform does **not** include:

- Sandboxed code execution environments for agent-generated code
- Container-based isolation for untrusted code
- Language-specific interpreters for code execution tools

Rationale: Safe code execution requires sophisticated sandboxing that is outside the scope of this work. Agents can invoke pre-defined tools but cannot execute arbitrary code.

Conversation Memory Systems

Beyond session-scoped context, the platform does **not** implement:

- Long-term episodic memory across sessions
- Semantic memory with embedding-based retrieval
- Summary memory for conversation compression
- Entity memory for tracking mentioned entities over time

Rationale: Session-based context suffices for the current use cases. Advanced memory systems could enhance user experience but require additional infrastructure (vector stores) and careful privacy considerations.

MLOps and Model Training

The platform does **not** include:

- Model fine-tuning or training pipelines
- Experiment tracking and model versioning
- A/B testing infrastructure for model comparison
- Model deployment automation

Rationale: The platform consumes pre-trained models via APIs. Model training and MLOps are separate concerns addressed by dedicated platforms (MLflow, Kubeflow, etc.).

1.6.3 Boundary Cases

Some capabilities exist in limited form within the platform:

Caching: LLM response caching is supported but not extensively optimized. Semantic deduplication of similar prompts is not implemented.

Distributed Deployment: While the architecture supports horizontal scaling, distributed locking for multi-worker job processing is not fully implemented.

Provider-Specific Features: The platform uses a common provider interface, which means provider-specific features (e.g., OpenAI function calling syntax) are abstracted rather than directly exposed.

Graph Analytics: Basic graph analytics (centrality, paths) are available through tools, but advanced algorithms (community detection, link prediction) are not implemented.

1.6.4 Scope Summary

Table 1.4 provides a concise summary of scope decisions.

1.6.5 Extensibility for Future Work

While certain capabilities are out of scope for this thesis, the architecture has been designed to accommodate future extensions:

- The tool framework can incorporate document RAG tools
- The orchestrator can be extended with additional agent patterns
- The provider abstraction can support streaming providers
- The job system can be enhanced with distributed locking
- Memory systems can be added as additional tools or services

Chapter 13 discusses these extension opportunities in the context of future work.

Table 1.4. Scope summary: included and excluded capabilities.

Capability	In Scope	Out of Scope
Single-agent orchestration	✓	
Multi-agent collaboration		✓
Graph NL-to-Cypher	✓	
Document RAG		✓
OIDC/JWT authentication	✓	
RBAC authorization	✓	
Multi-tenancy	✓	
Prometheus metrics	✓	
OpenTelemetry tracing	✓	
SSE job streaming	✓	
Token streaming		✓
MCP tools (34)	✓	
Code execution sandbox		✓
Session context	✓	
Long-term memory		✓
Docker deployment	✓	
Kubernetes operators		✓

1.7 Implementation Snapshot

All repository-derived numbers, metrics, and implementation details referenced throughout this thesis refer to commit `9b3e4c19a7853cc003e4ea094317255f7401e554` dated 2025-12-18. This snapshot includes 34 MCP tools, 76 API endpoints, 2,700+ test functions, and 8,263 lines of orchestration code.

1.8 Contributions

This thesis makes several contributions to the emerging field of enterprise AI agent systems. The contributions span architectural design, implementation, and practical deployment, collectively advancing the state of practice for production-ready agent platforms.

1.8.1 Primary Contributions

C1: Enterprise-Grade Agent Orchestration Architecture

This thesis presents a comprehensive architectural design for AI agent orchestration that addresses the full spectrum of enterprise requirements. The architecture comprises three distinct layers:

1. **Core Backend Layer:** FastAPI-based application providing RESTful APIs, service orchestration, and cross-cutting concerns (security, logging, metrics)

2. **Data and Persistence Layer:** Dual-database design with PostgreSQL as the authoritative control plane and Redis as the high-performance data plane, complemented by Memgraph for graph-based knowledge

3. **Presentation Layer:** Dual user interfaces optimized for different stakeholder personas (end-user chat and administrative control panel)

Novelty: While individual components exist in isolation across various frameworks, this architecture integrates them into a cohesive, production-ready whole with explicit attention to enterprise concerns (multi-tenancy, security, observability) that are typically afterthoughts in research-oriented frameworks.

C2: Native MCP Tool Ecosystem

The platform implements a comprehensive Model Context Protocol (MCP) tool ecosystem comprising:

- **34 tools** (at time of writing) organized across **17 categories** (graph, security, system, data, model, output, agent, cache, catalog, db, errors, privacy, ratelimit, session, tenancy, user, viz)
- Schema-driven tool definition with JSON Schema validation
- Tool discovery API enabling runtime capability enumeration
- Per-tool authorization integrated with RBAC
- Comprehensive audit logging for all tool invocations

Novelty: The MCP tool ecosystem provides a standardized, extensible foundation for tool integration that aligns with emerging industry standards. Unlike ad-hoc tool implementations in existing frameworks, this approach enables tool interoperability and consistent governance.

C3: Natural Language to Cypher Pipeline with Safety Validation

The NL-to-Cypher pipeline enables users to query graph databases using natural language while ensuring query safety through multi-layer validation:

1. **Natural Language Normalization:** Input sanitization and entity extraction
2. **Cypher Generation:** LLM-based translation with schema context
3. **Safety Validation:** Six-layer validation ensuring:
   - Syntactic correctness
   - Read-only operations (no mutations)
   - Tenant boundary enforcement
   - Query depth limits
   - Execution timeout bounds
   - Result size caps
4. **Query Execution:** Validated query execution against Memgraph
5. **Result Summarization:** LLM-based natural language response generation

Novelty: While NL-to-SQL has received significant attention, NL-to-Cypher for property graphs is less explored. The multi-layer safety validation addresses the unique challenges of graph query injection and multi-tenant data protection. The deterministic test mode enables reliable pipeline testing independent of LLM variability.

C4: Comprehensive Security and Governance Framework

The security framework provides defense-in-depth protection through:

- **Authentication:** OIDC/JWT-based authentication with JWKS key caching and rotation support
- **Authorization:** Role-Based Access Control with scope-based permissions and wildcard support
- **Multi-Tenancy:** Database-level tenant isolation with header-based tenant identification
- **Rate Limiting:** Sliding window algorithm with per-user and per-tenant quotas
- **PII Protection:** Detection and redaction of personally identifiable information in logs and outputs
- **Audit Logging:** Append-only audit trail for all significant operations
- **Intent Filtering:** Classification and blocking of potentially dangerous operations

Novelty: The integration of these security mechanisms into a unified framework specifically designed for AI agents is a significant contribution. The PII scrubbing and intent filtering address AI-specific risks not adequately covered by traditional enterprise security frameworks.

C5: Production-Ready Observability Stack

The observability implementation provides the three pillars necessary for production operations:

- **Metrics:** Prometheus-compatible metrics covering HTTP requests, agent runs, jobs, tool invocations, LLM calls, rate limiting, and provider health
- **Logging:** Structured logging with structlog, JSON format for production, request ID correlation, and PII-safe output
- **Tracing:** OpenTelemetry integration with span propagation, OpenTelemetry Protocol (OTLP) export, and sampling configuration
- **Health Probes:** Kubernetes-compatible liveness, readiness, and startup probes with component-level health reporting

Novelty: The observability stack is designed specifically for AI agent workloads, with metrics capturing LLM-specific dimensions (token counts, costs, provider health) alongside traditional operational metrics. The integration is comprehensive rather than add-on, enabling true production readiness.

C6: Asynchronous Job System with Progress Streaming

The job system supports long-running operations through:

- PostgreSQL-backed job persistence for durability
- Redis-backed job queues for performance
- Worker processes with heartbeat monitoring
- Server-Sent Events (SSE) for real-time progress updates
- Cooperative cancellation with cleanup procedures
- Graceful shutdown with in-flight job completion
- Idempotency support for safe retries

Novelty: Long-running AI agent tasks (complex reasoning, large graph queries) require robust asynchronous processing that existing frameworks typically lack. The dual-store architecture (PostgreSQL + Redis) provides both durability and performance.

1.8.2 Secondary Contributions

Beyond the primary contributions, this thesis provides:

Reference Implementation

A complete, working implementation of all described components:

- Approximately 77,000 lines of Python source code
- Over 2,700 test functions across 236+ test files
- Comprehensive API documentation (76 endpoints at time of writing, OpenAPI specification)
- Docker-based deployment configuration
- Operational runbooks and troubleshooting guides

Comparative Analysis

A detailed comparison of the Cineca Agentic Platform against state-of-the-art frameworks (LangChain, LlamaIndex, Semantic Kernel, OpenAI Assistants, AutoGen, crewAI), identifying:

- Areas where the platform leads (multi-tenancy, observability, graph integration)
- Areas of parity (tool extensibility, basic orchestration)
- Areas requiring future work (multi-agent collaboration, advanced memory)

Deployment Experience

Practical insights from deploying the platform in the CINECA environment:

- Configuration patterns for different deployment scenarios
- Performance characteristics under realistic workloads
- Operational lessons learned
- Identified technical debt and recommended improvements

1.8.3 Contribution Summary

Table 1.5 summarizes the contributions and their significance.

Table 1.5. Summary of thesis contributions.

ID	Contribution	Significance
C1	Architecture	Integrated enterprise-grade design addressing production requirements
C2	MCP Tools	Standardized, extensible tool ecosystem with 34 tools
C3	NL-to-Cypher	Safe graph querying with multi-layer validation
C4	Security	Comprehensive framework for AI agent governance
C5	Observability	Production-ready monitoring for AI workloads
C6	Job System	Durable async processing with progress streaming

1.8.4 Artifact Availability

The contributions of this thesis are realized in the Cineca Agentic Platform, which is:

- Implemented as a complete, working system
- Documented through comprehensive API specifications and guides

- Tested through extensive automated test suites
- Deployed in the CINECA production environment

The platform represents a significant engineering effort totaling approximately 18 months of development, resulting in a system that serves as both a practical tool for CINECA and a reference architecture for the broader community of enterprise AI practitioners.

1.9 Thesis Structure

This thesis is organized into thirteen chapters, structured into five logical parts, followed by appendices. This section provides a roadmap to guide readers through the document.

1.9.1 Part I: Introduction and Background

Chapter 1: Introduction (This chapter) Establishes the motivation for the work, identifies stakeholders and use cases, describes the industrial context at CINECA, articulates the problem statement, defines research objectives, clarifies scope boundaries, summarizes contributions, and outlines the thesis structure.

Chapter 2: Background and Related Work Provides the theoretical and technical foundations for the thesis. Covers the evolution of Large Language Models and AI agents, surveys existing orchestration frameworks (LangChain, LlamaIndex, AutoGen, Semantic Kernel, crewAI), introduces the Model Context Protocol (MCP), discusses graph databases and natural language interfaces, reviews security and multi-tenancy patterns, and establishes observability concepts. Includes a comparative analysis matrix positioning the Cineca Agentic Platform relative to state-of-the-art alternatives.

Chapter 3: Requirements and Design Goals Formalizes the requirements derived from stakeholder analysis and the CINECA context. Presents functional requirements (agent execution, tool invocation, graph querying), non-functional requirements (security, performance, reliability), constraints and assumptions, design principles, evaluation criteria, and a requirements traceability matrix linking requirements to their verification in later chapters.

1.9.2 Part II: Architecture and Design

Chapter 4: System Architecture Presents the comprehensive architectural design of the platform. Describes the three-layer architecture (core backend, data/persistence, presentation), details the API layer design with 76 endpoints across 16 categories, explains the repository pattern implementation, discusses cross-cutting concerns (logging, metrics, tracing, security, rate limiting, error handling), and documents the dual user interface architecture (Next.js and Streamlit).

Chapter 5: Agent Orchestration Engine Details the agent orchestration subsystem. Covers orchestration design principles, intent classification pipeline, agent run lifecycle and state machine, NL-to-Cypher pipeline stages with safety validation, LLM provider management with resilience framework (circuit breakers, fallback chains, cost tracking), and the Default Model Resolver hierarchy.

Chapter 6: MCP Tools Ecosystem Describes the Model Context Protocol tool implementation. Explains the MCP runtime architecture, documents the 34 tools across 17 categories, details tool authorization mechanisms, and provides guidance for extending the tool ecosystem with difficulty ratings for different extension types.

Chapter 7: Background Jobs and Workers Covers the asynchronous job processing system. Details the job model and lifecycle states, worker architecture with queue polling and heartbeats, Server-Sent Events (SSE) streaming for progress updates, job cancellation mechanisms, and the APScheduler-based background task framework for scheduled operations (health checks, backups, cleanup).

1.9.3 Part III: Security and Governance

Chapter 8: Security Framework Presents the comprehensive security implementation. Covers OIDC/JWT authentication with JWKS management, Role-Based Access Control with scope-based permissions, multi-tenancy implementation with database-level isolation, rate limiting algorithms, data protection (PII scrubbing, output guards), threat modeling, and audit logging for compliance.

1.9.4 Part IV: Observability and Operations

Chapter 9: Observability Details the observability stack implementation. Covers Prometheus metrics catalog (HTTP, agents, jobs, tools, LLM calls, providers), structured logging with structlog and request correlation, OpenTelemetry distributed tracing, Kubernetes-compatible health probes, and operational runbooks for common scenarios.

1.9.5 Part V: Implementation, Deployment, and Evaluation

Chapter 10: Implementation and Engineering Practices Describes the technical implementation. Covers the technology stack (Python, FastAPI, PostgreSQL, Redis, Memgraph, Next.js, Streamlit), codebase organization with approximately 77,000 lines of code, coding standards, testing strategy with over 2,700 test functions, and platform engineering utilities (pagination, idempotency, JSON handling, deprecation framework).

Chapter 11: Configuration and Deployment Addresses deployment concerns. Covers the Pydantic-based configuration system, environment profiles (development, testing, production), Docker Compose deployment, deployment topologies and scaling strategies, production hardening considerations (TLS, security headers, GPU support), operational scripts, and troubleshooting guidance.

Chapter 12: Evaluation Presents the evaluation methodology and results. Covers evaluation setup, end-to-end workflow testing, functional evaluation against requirements, performance benchmarking, availability and fault-tolerance assessment, security evaluation, cost tracking evaluation, threats to validity, and comparative analysis against related frameworks with empirical results.

Chapter 13: Conclusions and Future Work Synthesizes findings and looks forward. Summarizes contributions and their validation, discusses lessons learned from the development and deployment experience, acknowledges limitations and technical debt, proposes future work directions (multi-agent collaboration, advanced memory, streaming responses), and reflects on implications for enterprise AI architectures.

1.9.6 Appendices

The thesis includes seven appendices providing reference material:

Appendix A: API Endpoint Reference Complete documentation of all 76 API endpoints across 16 categories, including methods, paths, descriptions, and authorization requirements.

Appendix B: MCP Tool Reference Comprehensive inventory of all 34 MCP tools with categories, descriptions, required scopes, and usage examples.

Appendix C: Database Schema PostgreSQL table definitions and Memgraph graph schema with entity relationships.

Appendix D: Architecture Decision Records (ADR) Summary Summary of key architectural decisions with rationale and alternatives considered.

Appendix E: Configuration Reference Complete listing of environment variables and configuration parameters with defaults and descriptions.

Appendix F: Example Configurations Sample configuration files for development, testing, and production deployments.

Appendix G: Glossary of Terms Definitions of key domain terms, state machines, and acronyms used throughout the thesis.

1.9.7 Reading Paths

Different readers may benefit from different paths through this thesis:

For Architects and Technical Leaders: Begin with this introduction (Chapter 1), then proceed to System Architecture (Chapter 4), Security Framework (Chapter 8), and Evaluation (Chapter 12). Consult the Comparison section (Section 12.8) for positioning relative to alternatives.

For Developers and Implementers: After the introduction, focus on Implementation (Chapter 10), MCP Tools (Chapter 6), and Configuration and Deployment (Chapter 11). The appendices provide essential reference material.

For Researchers: Proceed through Background (Chapter 2), Requirements (Chapter 3), and the Agent Orchestration Engine (Chapter 5), particularly the NL-to-Cypher pipeline. The Evaluation chapter validates the research contributions.

For Operations Engineers: Focus on Observability (Chapter 9), Background Jobs (Chapter 7), and Configuration and Deployment (Chapter 11). The operational runbooks and troubleshooting sections are particularly relevant.

1.9.8 Notation and Conventions

Throughout this thesis:

- **Code snippets** appear in monospace font within `listings` environments
- **File paths** are shown in monospace, e.g., `src/services/orchestrator.py`
- **API endpoints** follow REST conventions, e.g., `GET /v1/health/ready`
- **Configuration variables** use uppercase with underscores, e.g., `REDIS_URL`
- **Cross-references** use LaTeX labels, e.g., Section 1.1
- **Citations** reference the bibliography, e.g., [1]
- **Emphasis** indicates terms being defined or key concepts

Tables are numbered within chapters (e.g., Table 1.5) and figures follow the same convention. Algorithms, when presented, use the `algorithmic` environment.

Chapter 2

Background and Related Work

This chapter establishes the theoretical and technical foundations for the Cineca Agentic Platform by reviewing relevant research and existing systems in the domains of large language models, AI agents, orchestration frameworks, and enterprise system design. Building upon the motivation and problem statement presented in Chapter 1, this chapter surveys the state of the art in agent orchestration, tool integration protocols, graph database interfaces, security models, and observability patterns. The chapter positions the platform's contributions relative to existing frameworks and identifies the gaps that this thesis addresses.

2.1 Large Language Models and AI Agents

The emergence of Large Language Models (LLMs) has fundamentally transformed the landscape of artificial intelligence, enabling new paradigms for building intelligent software systems. This section provides the theoretical foundations necessary to understand how the Cineca Agentic Platform leverages these advances to create enterprise-grade AI agent orchestration capabilities.

2.1.1 Evolution of Large Language Models

Large Language Models represent a class of neural network architectures, predominantly based on the Transformer architecture introduced by Vaswani et al. in 2017 [20] , that are trained on massive text corpora to predict the next token in a sequence. The scale of these models—measured in parameters, training data, and computational resources—has grown exponentially over the past several years, leading to emergent capabilities that were not explicitly programmed.

Foundational Models and Scaling Laws The modern era of LLMs began with the introduction of GPT (Generative Pre-trained Transformer) by OpenAI in 2018, which demonstrated that unsupervised pre-training on large text corpora followed by task-specific fine-tuning could achieve state-of-the-art results across diverse NLP benchmarks. Subsequent scaling studies established empirical "scaling laws" showing that model performance improves predictably with increases in model size, dataset size, and compute budget [21] .

The GPT series evolved rapidly: GPT-2 (1.5B parameters, 2019) demonstrated coherent long-form text generation; GPT-3 (175B parameters, 2020) exhibited remarkable few-shot learning capabilities; and GPT-4 (2023) achieved human-level performance on professional and academic examinations while introducing multimodal capabilities. These developments established the foundation for using LLMs as reasoning engines in agentic systems.

Open-Weight Models Parallel to proprietary developments, the open-source community has produced increasingly capable models. Meta's LLaMA (Large Language Model Meta AI) family, released starting in 2023, demonstrated that smaller, efficiently trained models could approach the performance of much larger proprietary systems. LLaMA 2 (7B–70B parameters) and LLaMA 3 (8B–70B parameters) provided researchers and enterprises with high-quality foundation models that can be deployed on-premises, addressing data sovereignty and privacy concerns critical in enterprise environments.

Microsoft's Phi series (Phi-1, Phi-2, Phi-3) explored the "small language model" paradigm, demonstrating that models with 1–14 billion parameters, when trained on carefully curated high-quality data, can achieve competitive performance on reasoning tasks. This is particularly relevant for deployment scenarios where computational resources are constrained or where low-latency inference is required.

Other notable open-weight models include Mistral AI's Mistral and Mixtral series (utilizing mixture-of-experts architectures for efficient inference), Alibaba's Qwen series, and Google's Gemma models. The Cineca Agentic Platform is designed to be model-agnostic, supporting any OpenAI-compatible API endpoint, which enables deployment with both proprietary services (OpenAI, Azure OpenAI) and self-hosted open-weight models via inference servers such as Ollama, vLLM, or text-generation-inference.

Emergent Capabilities A defining characteristic of modern LLMs is the emergence of capabilities at scale that were not present in smaller models. These emergent capabilities include:

- **In-context learning**: The ability to perform new tasks by conditioning on examples provided in the prompt, without updating model weights.
- **Chain-of-thought reasoning**: When prompted to "think step by step," LLMs can decompose complex problems into intermediate reasoning steps, significantly improving performance on arithmetic, commonsense, and symbolic reasoning tasks [22] .
- **Instruction following**: Models fine-tuned with reinforcement learning from human feedback (RLHF) or direct preference optimization (DPO) can follow natural language instructions with high fidelity.
- **Tool use**: LLMs can learn to invoke external tools (APIs, databases, calculators) by generating structured function calls, extending their capabilities beyond pure text generation.

These emergent capabilities form the foundation for building AI agents that can reason about complex tasks and take actions in external environments.

2.1.2 Definition and Characteristics of AI Agents

An **AI agent** is an autonomous software entity that perceives its environment through observations, reasons about goals and constraints, and takes actions to achieve specified objectives. In the context of LLM-based systems, AI agents leverage the reasoning capabilities of language models to plan and execute multi-step tasks, often involving interaction with external tools and data sources.

Agent Components A typical LLM-based agent consists of several key components:

1. **Language Model Core**: The LLM serves as the "brain" of the agent, providing natural language understanding, reasoning, and generation capabilities. The model processes observations and instructions to produce action decisions.
2. **Memory Systems**: Agents require memory to maintain context across interactions. This includes:
   - *Short-term memory*: The conversation history and current context window maintained during a single session or run.
   - *Long-term memory*: Persistent storage of past interactions, learned facts, or retrieved knowledge that persists across sessions.
   - *Working memory*: Scratchpad-style storage for intermediate reasoning steps and partial results.
3. **Tool Interface**: A mechanism for the agent to invoke external capabilities such as web search, database queries, code execution, or API calls. Tools extend the agent's abilities beyond what the language model can accomplish through pure text generation.
4. **Planning and Control**: Logic that orchestrates the agent's behavior, deciding when to think, when to act, and when to request human input. This may be implicit (encoded in prompts) or explicit (implemented in code).
5. **Observation Interface**: The means by which the agent perceives the results of its actions and the state of its environment, including tool outputs, user feedback, and environmental signals.

Agent Autonomy Spectrum AI agents exist on a spectrum of autonomy, from simple prompt-response systems to fully autonomous goal-pursuing entities:

- **Level 0 – Conversational**: Basic chatbots that respond to user messages without persistent state or tool access.
- **Level 1 – Tool-augmented**: Agents that can invoke predefined tools in response to user requests, typically with human approval for each action.

- **Level 2 – Task-oriented**: Agents that can plan and execute multi-step tasks autonomously, with human oversight at task boundaries.
- **Level 3 – Goal-oriented**: Agents that pursue high-level goals with minimal human intervention, adapting their strategies based on observations.
- **Level 4 – Fully autonomous**: Agents that operate independently over extended periods, managing their own resources and objectives.

The Cineca Agentic Platform primarily targets Level 1–2 agents: tool-augmented and task-oriented systems where the platform provides the infrastructure for tool invocation, session management, and execution oversight, while maintaining human control through approval workflows, rate limiting, and audit logging.

2.1.3 Agent Architectures and Reasoning Patterns

Several architectural patterns have emerged for structuring LLM-based agent behavior. Understanding these patterns is essential for appreciating the design decisions in the Cineca Agentic Platform's orchestrator.

ReAct (Reasoning and Acting) The ReAct framework, introduced by Yao et al. in 2022 [5] , interleaves reasoning traces with action execution in a unified prompt format. At each step, the agent:

1. **Thinks**: Generates a reasoning trace explaining its current understanding and planned action.
2. **Acts**: Selects and executes an action (typically a tool invocation).
3. **Observes**: Receives the result of the action and incorporates it into context.

This think-act-observe loop continues until the agent determines that the task is complete or a termination condition is met. ReAct has proven effective for tasks requiring both knowledge retrieval and multi-step reasoning, as the explicit reasoning traces help the model maintain coherent plans and recover from errors.

The Cineca Agentic Platform's orchestrator implements a ReAct-style execution loop, where each agent run consists of a sequence of steps (recorded in the `steps_json` field of the `AgentRun` model), with each step capturing the agent's reasoning, the tool invoked (if any), and the observation received.

Chain-of-Thought (CoT) Prompting Chain-of-thought prompting [22] encourages the model to generate intermediate reasoning steps before producing a final answer. While not an agent architecture per se, CoT is frequently incorporated into agent systems to improve reasoning quality. Variants include:

- **Zero-shot CoT**: Adding "Let's think step by step" to prompts.
- **Few-shot CoT**: Providing examples with explicit reasoning chains.
- **Self-consistency**: Sampling multiple reasoning paths and aggregating results.

Plan-and-Execute In the plan-and-execute paradigm, the agent first generates a complete plan (a sequence of steps to achieve the goal) and then executes the plan step by step. This separation allows for:

- Plan review and approval before execution
- Parallel execution of independent steps
- Re-planning when intermediate steps fail

While the Cineca Agentic Platform's current orchestrator uses a ReAct-style approach (planning and execution interleaved), the architecture supports extension to plan-and-execute patterns through the task-based tracking mechanism, where agents can decompose tasks into subtasks before execution.

Tree-of-Thought and Graph-of-Thought More advanced architectures explore the space of possible reasoning paths more systematically. Tree-of-Thought [23] maintains multiple reasoning branches and uses search algorithms (breadth-first, depth-first, or beam search) to find successful paths. Graph-of-Thought extends this to arbitrary graph structures of reasoning states.

These approaches are computationally expensive but can improve performance on complex reasoning tasks. The Cineca Agentic Platform does not currently implement tree/graph-of-thought natively, but the extensible orchestrator design would allow such patterns to be added as alternative reasoning strategies.

Multi-Agent Architectures Some systems employ multiple specialized agents that collaborate to solve complex tasks:

- **Hierarchical agents**: A manager agent delegates subtasks to worker agents.
- **Debate and critique**: Multiple agents propose solutions and critique each other's reasoning.
- **Role-playing**: Agents assume different personas or expertise areas.

Frameworks such as AutoGen and CrewAI are specifically designed for multi-agent collaboration. The Cineca Agentic Platform currently focuses on single-agent orchestration but provides the infrastructure (session management, tool invocation, persistent state) that would support multi-agent extensions.

2.1.4 Tool Use in LLM Agents

The ability to invoke external tools is what transforms a language model from a text generator into an agent capable of taking actions in the world. Tool use has become a central capability of modern LLM-based systems.

Function Calling Modern LLMs increasingly support “function calling” or “tool use” as a native capability. When provided with tool definitions (typically as JSON schemas), the model can:

1. Recognize when a tool invocation is appropriate for the current task
2. Generate structured tool calls with the required parameters
3. Incorporate tool outputs into its subsequent reasoning

OpenAI’s function calling API, introduced in 2023, established a de facto standard format for tool definitions that has been adopted by many other providers. The Cineca Agentic Platform uses this format for its MCP tool definitions, enabling compatibility with a wide range of LLM backends.

Tool Categories Tools in agent systems can be categorized by their function:

- **Information retrieval**: Web search, document lookup, database queries
- **Computation**: Calculators, code interpreters, data analysis
- **Action execution**: API calls, file operations, system commands
- **Communication**: Email, messaging, notification services
- **Memory management**: Reading/writing to persistent storage

The Cineca Agentic Platform provides 34 MCP tools across 17 categories, covering graph operations, caching, data management, security administration, and more. This comprehensive tool ecosystem enables agents to perform complex enterprise workflows.

Tool Safety and Governance Tool invocation in enterprise environments requires careful governance:

- **Authorization**: Which users/roles can invoke which tools?
- **Validation**: Are tool inputs properly sanitized and validated?
- **Auditing**: Are all tool invocations logged for compliance?
- **Rate limiting**: Are usage quotas enforced to prevent abuse?
- **Sandboxing**: Are tool effects isolated and reversible when possible?

These governance concerns are first-class features of the Cineca Agentic Platform, implemented through its RBAC system, capability-based tool access control, comprehensive audit logging, and configurable rate limiting. This distinguishes the platform from research-oriented frameworks that often assume trusted users and benign environments.

2.1.5 Challenges in Production Agent Deployment

Deploying LLM-based agents in production environments presents unique challenges that motivate the design of the Cineca Agentic Platform:

Reliability and Error Handling LLMs can produce incorrect, inconsistent, or nonsensical outputs, particularly for edge cases or adversarial inputs. Production systems must:

- Detect and gracefully handle model failures
- Implement retry logic with appropriate backoff
- Provide fallback behaviors when primary models are unavailable
- Maintain circuit breakers to prevent cascade failures

The Cineca Agentic Platform implements a comprehensive LLM resilience framework including circuit breakers, provider health monitoring, and automatic fallback to alternative providers.

Latency and Cost Management LLM inference is computationally expensive and can introduce significant latency. Enterprise deployments must:

- Monitor and optimize inference latency
- Track and budget API costs
- Cache responses where appropriate
- Choose appropriately-sized models for different tasks

The platform provides detailed cost tracking, per-model and per-tenant usage metrics, and configurable model defaults to balance capability against cost.

Security and Compliance AI agents with tool access can potentially execute sensitive operations. Security requirements include:

- Authentication and authorization for all operations
- Audit trails for compliance and forensics
- PII detection and protection
- Prompt injection defense
- Output validation and filtering

These security features are implemented throughout the Cineca Agentic Platform, as detailed in Chapter 8.

Observability Understanding agent behavior in production requires comprehensive observability:

- Metrics: Request rates, latencies, error rates, token usage
- Logs: Structured, correlated logs with request tracing
- Traces: Distributed tracing across service boundaries
- Health checks: Readiness and liveness probes for orchestration

The platform provides a production-grade observability stack with Prometheus metrics, OpenTelemetry tracing, and structured logging with request ID correlation, as discussed in Chapter 9.

2.2 Agent Orchestration Frameworks

The proliferation of LLM-based agents has spawned a diverse ecosystem of frameworks, libraries, and platforms designed to simplify agent development. This section surveys the major orchestration frameworks, analyzes their architectural approaches, and positions the Cineca Agentic Platform within this landscape.

2.2.1 Overview of the Framework Landscape

Agent orchestration frameworks can be categorized along several dimensions:

Abstraction Level Frameworks vary in the level of abstraction they provide:

- **Low-level libraries**: Provide composable primitives (chains, retrievers, prompt templates) that developers assemble into applications.
- **Agent runtimes**: Provide execution environments for agent loops with built-in reasoning patterns.
- **Full-stack platforms**: Provide end-to-end infrastructure including APIs, persistence, security, and observability.

Deployment Model Frameworks differ in their deployment assumptions:

- **In-process libraries**: Run within the application's process, suitable for embedding in existing systems.
- **Standalone services**: Run as separate processes or containers, providing API-based access.
- **Cloud services**: Run as managed services, abstracting infrastructure entirely.

Scope of Concerns Frameworks address different subsets of the concerns in agent development:

- Prompting and chain composition
- Tool definition and invocation
- Memory and state management
- Multi-agent collaboration
- Production operations (auth, logging, monitoring)

Understanding these dimensions is essential for selecting appropriate technologies and for appreciating where the Cineca Agentic Platform fits in the ecosystem.

2.2.2 LangChain

LangChain [4] is arguably the most widely adopted framework for building LLM-powered applications. Launched in late 2022, it has grown into a comprehensive ecosystem covering the full lifecycle of LLM application development.

Architecture LangChain is structured around several core abstractions:

- **Models**: Wrappers around LLMs (chat models, completion models, embeddings) from various providers.
- **Prompts**: Templates for constructing prompts with variable substitution and formatting.
- **Chains**: Compositions of multiple processing steps, from simple sequential chains to complex directed acyclic graphs.
- **Agents**: Autonomous entities that use LLMs to decide which actions to take, implemented via various agent types (ReAct, OpenAI functions, structured chat, etc.).
- **Memory**: Systems for maintaining conversational context across interactions.
- **Retrievers**: Components for fetching relevant documents from various sources (vector stores, databases, web search).
- **Tools**: Interfaces to external capabilities that agents can invoke.

Strengths LangChain's primary strengths include:

- **Extensive integrations**: Over 100 integrations with LLM providers, vector stores, document loaders, and external services.
- **Flexible composition**: The chain abstraction allows building complex pipelines through composition.

- **Active community**: Large community contributing examples, integrations, and improvements.
- **Multiple agent types**: Support for various agent architectures including ReAct, Plan-and-Execute, and OpenAI function-based agents.
- **LangSmith**: A companion product for monitoring, debugging, and evaluating LLM applications (though proprietary and paid).

Limitations for Enterprise Use Despite its popularity, LangChain presents challenges for enterprise deployments:

- **No built-in multi-tenancy**: Tenant isolation must be implemented by the application developer.
- **Limited production infrastructure**: Authentication, authorization, rate limiting, and audit logging are not provided.
- **In-process execution**: LangChain is designed as a library, requiring additional infrastructure for standalone service deployment.
- **Abstraction overhead**: The many layers of abstraction can make debugging difficult and introduce performance overhead.
- **Rapid API changes**: The framework has undergone significant API changes, creating upgrade challenges.

LangGraph Extension LangGraph [25] is a newer addition to the LangChain ecosystem, providing a graph-based orchestration layer for building stateful, multi-actor applications. It addresses some limitations of basic LangChain agents by providing:

- Explicit state machines for agent control flow
- Support for long-running workflows with persistence
- Human-in-the-loop interaction patterns
- Streaming of intermediate results

LangGraph represents a step toward production-ready agent infrastructure, though it still relies on external systems for authentication, multi-tenancy, and operational concerns.

2.2.3 LlamaIndex

LlamaIndex [26] (formerly GPT Index) focuses on the data aspects of LLM applications, providing sophisticated tools for ingesting, indexing, and querying data with LLMs.

Core Concepts LlamaIndex is built around:

- **Data connectors**: Loaders for 160+ data sources including files, databases, APIs, and web pages.
- **Indexes**: Structures for organizing data for efficient retrieval (vector indexes, tree indexes, keyword indexes, knowledge graphs).
- **Query engines**: Components that combine retrieval with LLM-based synthesis to answer questions.
- **Agents**: LLM-powered agents that can use tools including query engines over indexes.
- **Observability**: Integration with monitoring tools for debugging and optimization.

Strengths LlamaIndex excels at:

- **Data ingestion**: Comprehensive support for extracting and processing data from diverse sources.
- **Retrieval strategies**: Sophisticated retrieval mechanisms including recursive retrieval, auto-merging, and query transformation.
- **Response synthesis**: Multiple strategies for combining retrieved context with LLM generation.
- **Knowledge graphs**: Support for building and querying knowledge graph indexes.

Limitations For enterprise agent platforms, LlamaIndex has limitations:

- **RAG-centric**: Primary focus is on retrieval-augmented generation rather than general agent orchestration.
- **No production infrastructure**: Like LangChain, it is a library without built-in authentication, multi-tenancy, or operational features.
- **Limited agent capabilities**: Agent support is less mature than in agent-focused frameworks.

2.2.4 Microsoft Semantic Kernel

Semantic Kernel [27] is Microsoft's framework for integrating LLMs into applications, with a focus on enterprise scenarios and Azure ecosystem integration.

Design Philosophy Semantic Kernel takes a plugin-oriented approach:

- **Skills/Plugins**: Reusable units of functionality, either semantic (prompt-based) or native (code-based).
- **Planners**: Components that create execution plans from high-level goals using available skills.
- **Memory**: Semantic memory for storing and retrieving information using embeddings.
- **Connectors**: Integrations with LLM providers, vector databases, and Azure services.

Strengths Semantic Kernel offers:

- **Multi-language support**: SDKs for C#, Python, and Java, enabling use in diverse technology stacks.
- **Enterprise focus**: Designed with enterprise integration in mind, including Azure AD authentication.
- **Planner abstractions**: Sophisticated planners (Sequential, Stepwise, Action) for agent behavior.
- **Microsoft ecosystem**: Deep integration with Azure OpenAI, Microsoft 365, and other Microsoft services.

Limitations Potential limitations include:

- **Azure affinity**: Strongest when used with Azure services; less mature for other cloud providers.
- **Smaller community**: Less community momentum compared to LangChain.
- **No built-in multi-tenancy**: Multi-tenant isolation must be implemented by the application.

2.2.5 OpenAI Assistants API

OpenAI's Assistants API [24] provides a managed agent service with built-in capabilities for tool use, file handling, and code execution.

Capabilities The Assistants API offers:

- **Function calling**: Native support for invoking developer-defined functions.
- **Code interpreter**: Sandboxed Python execution environment for data analysis and computation.
- **File search**: Vector-based retrieval over uploaded documents.
- **Threads**: Managed conversation history with automatic context management.
- **Streaming**: Real-time streaming of assistant responses.

Strengths As a managed service, the Assistants API provides:

- **Zero infrastructure**: No need to deploy or manage agent infrastructure.
- **Native capabilities**: Code execution and file search are built-in, not requiring custom implementation.
- **Optimized performance**: OpenAI optimizes the underlying infrastructure for their models.

Limitations However, the managed approach has significant limitations for enterprise use:

- **Vendor lock-in**: Only OpenAI models can be used; no support for open-weight or on-premises models.
- **Data sovereignty**: All data is processed on OpenAI's infrastructure in the US, which may violate data residency requirements.
- **Limited customization**: The agent behavior is controlled by OpenAI's implementation, limiting customization.
- **Usage-based pricing**: Costs can be unpredictable for high-volume deployments.
- **No multi-tenancy**: Organization-level isolation only; tenant-level isolation requires separate accounts.

2.2.6 AutoGen

AutoGen is Microsoft Research's framework for building multi-agent systems where multiple AI agents collaborate to solve complex tasks.

Multi-Agent Paradigm AutoGen is built around the concept of conversable agents that can communicate with each other:

- **AssistantAgent**: An AI agent backed by an LLM.
- **UserProxyAgent**: An agent that can execute code and represent human users.
- **GroupChat**: A coordination mechanism for multi-agent conversations.
- **Agent compositions**: Patterns for hierarchical, sequential, and parallel agent collaboration.

Strengths AutoGen excels at:

- **Multi-agent workflows**: Native support for agent-to-agent communication and collaboration.
- **Code execution**: Docker-based sandboxed code execution for safety.
- **Human-in-the-loop**: Built-in patterns for human approval and intervention.
- **Research flexibility**: Designed for exploring novel multi-agent architectures.

Limitations AutoGen's limitations include:

- **Research-oriented**: Designed for research and experimentation rather than production deployment.
- **No production infrastructure**: Lacks authentication, authorization, persistence, and operational tooling.
- **In-memory state**: Agent state is not persisted, limiting use for long-running workflows.

2.2.7 CrewAI

CrewAI [29] is a framework specifically designed for orchestrating teams of AI agents with defined roles and collaborative processes.

Core Abstractions CrewAI introduces:

- **Agents**: Autonomous entities with defined roles, goals, and backstories.
- **Tasks**: Specific assignments given to agents with expected outputs.
- **Crews**: Teams of agents working together on related tasks.
- **Processes**: Coordination patterns (sequential, hierarchical) for task execution.
- **Tools**: Capabilities that agents can use to accomplish tasks.

Strengths CrewAI provides:

- **Role-based agents**: Intuitive model for defining specialized agent personas.
- **Task delegation**: Built-in support for agents delegating work to other agents.
- **Process management**: Explicit control over how agents coordinate.
- **Simplicity**: Relatively simple API compared to more complex frameworks.

Limitations Like other multi-agent frameworks:

- **Limited production readiness**: No built-in authentication, persistence, or operational features.
- **No multi-tenancy**: All executions run in a shared context.
- **Basic observability**: Limited metrics and logging capabilities.

2.2.8 State-of-the-Art Comparison Matrix

This thesis focuses on the design and implementation of an enterprise-grade agent platform. To position the Cineca Agentic Platform within the current ecosystem, it is important to distinguish between (i) **protocol specifications**, (ii) **tool servers**, and (iii) **agent/orchestration libraries**.

Framing: platform vs. protocol vs. library

- **Model Context Protocol (MCP)** is a *standard* describing how an AI client discovers and calls tools (e.g., `tools/list`, `tools/call`). MCP is not an agent runtime by itself.
- **GitHub MCP Server** and **neo4j-contrib/mcp-neo4j** are *MCP servers*: they expose domain-specific capabilities as MCP tools.
- **LangChain** and **LlamaIndex** primarily provide *in-process components* (chains, retrievers, indices) that can be composed into applications.
- **LangGraph** is a low-level *orchestration runtime* for long-running, stateful agent workflows.
- The **Cineca Agentic Platform** is a *full-stack backend platform* that integrates orchestration, tools, persistence, governance (auth/RBAC/tenancy), background jobs, and observability.

Decision summary The Cineca Agentic Platform is strongest when the requirements include enterprise concerns such as multi-tenancy, centralized RBAC, audit logging, rate limiting, operational monitoring, and long-running background jobs with streaming progress. In contrast, MCP servers excel when the primary requirement is *standardized tool interoperability* with MCP-capable clients, and libraries such as LangChain/LlamaIndex excel for rapid prototyping inside a single application.

Strengths of Cineca relative to common frameworks Compared to typical agent/RAG frameworks, Cineca provides a larger fraction of production requirements *as first-class features*: tenant isolation, centralized permissions, auditability, rate limiting, and operational observability. This is especially relevant for deployments in research or enterprise environments where compliance, traceability, and controlled tool execution are required.

Table 2.1. High-level comparison of Cineca Agentic Platform vs. representative protocol/server/library alternatives.

Capability	Cineca	MCP	GitHub MCP	mcp-neo4j	GraphCypherQA	KG Index	LangGraph (+Neo4j)
Category	Platform	Protocol	Tool server	Tool server	Library chain	Library index	Orchestration
End-to-end backend API	Y	N	N	N	N	N	P
Multi-tenancy	Y	N	N	N	N	N	P
Central RBAC/governance	Y	P	P	P	N	N	P
Audit trail (runs/tools)	Y	P	P	P	N	N	P
Rate limiting built-in	Y	P	P	P	N	N	P
Background jobs/workers	Y	N	N	N	N	N	P
Streaming progress (SSE/streaming)	Y	P	P	Y	N	N	P
Graph DB integration	Y (Memgraph)	N	N	Y (Neo4j)	Y (Neo4j)	P	Y (Neo4j)
NL→Cypher workflow	Y (pipeline)	N	N	Y (tools)	Y (chain)	N	P
MCP interoperability (JSON-RPC)	P	Y	Y	Y	N	N	N
Observability (metrics/traces/health)	Y	N	P	P	P	P	P

Legend: Y = yes (built-in), P = partial/depends on integration, N = not in scope.

Trade-offs The main trade-off is scope: a full platform is heavier than embedding a single chain/index in a Python application. Moreover, Cineca's tooling surface is "MCP-style" (tool discovery and invocation), but strict MCP interoperability requires implementing the MCP JSON-RPC transport and contracts; similarly, Cineca's graph subsystem targets Memgraph, whereas Neo4j-centered ecosystems (e.g., Aura, Neo4j-specific tooling) may be better served by Neo4j-native components.

2.2.9 Extended Framework Comparison

To provide a more comprehensive comparison, Table 2.2 presents additional dimensions of comparison across the major frameworks discussed in this section.

Table 2.2. Extended comparison of agent orchestration frameworks across production readiness dimensions.

Capability	Cineca	LangChain	LlamaIndex	Semantic Kernel	OpenAI Assistants	AutoGen	CrewAI
Multi-Agent Support	N	P	P	P	N	Y	Y
RAG Pipeline	N	Y	Y	P	Y	N	N
Memory Systems	P	Y	Y	Y	Y	P	P
Real-time Streaming	P	Y	Y	Y	Y	P	P
Multi-Tenancy	Y	N	N	P	P	N	N
Enterprise Auth (OIDC/RBAC)	Y	N	N	Y	P	N	N
Observability Stack	Y	P	P	P	N	N	N
Graph Integration	Y	P	P	N	N	N	N
Self-Hosted Deployment	Y	Y	Y	P	N	Y	Y
Production Ready	Y	P	P	Y	Y	N	N

Legend: Y = yes (built-in/strong), P = partial/community support, N = not available.

Evaluation Dimensions This comparison evaluates frameworks across five critical dimensions for enterprise deployment:

1. **Multi-tenancy**: Native support for tenant isolation at the data, API, and tool invocation levels. Frameworks lacking multi-tenancy require significant customization for enterprise deployments.

2. **Tool Protocol Standardization**: Standardized interfaces for tool discovery and invocation (e.g., MCP compliance, OpenAI function calling format). Standardization enables interoperability and reduces vendor lock-in.

3. **Observability and SLOs**: Built-in metrics, distributed tracing, structured logging, and health probes enabling production monitoring and service-level objective (SLO) tracking.

4. **Security Model**: Authentication (OIDC/JWT), authorization (RBAC), audit logging, PII protection, and rate limiting as first-class features rather than add-ons.

5. **NL→DB Reliability**: Natural language to database query translation with safety validation (syntax checking, read-only enforcement, tenant isolation, depth limits) to prevent malicious or erroneous queries.

The Cineca Agentic Platform provides comprehensive coverage across all five dimensions, distinguishing it from research-oriented frameworks that typically address only subsets of these concerns.

Key Observations Several patterns emerge from this comparison:

1. **Library vs. Platform Trade-off**: Frameworks like LangChain and LlamaIndex offer flexibility and extensive integrations but require significant additional work to deploy in production. Platforms like the Cineca Agentic Platform provide more out-of-the-box but are more opinionated.

2. **Multi-Agent Gap**: While AutoGen and CrewAI specialize in multi-agent collaboration, they lack production infrastructure. The Cineca Agentic Platform focuses on single-agent orchestration with robust production features, leaving multi-agent as a future extension.

3. **Enterprise Features**: Authentication, authorization, multi-tenancy, and audit logging are consistently missing from research-oriented frameworks. These features require significant implementation effort and are often overlooked in academic settings.

4. **Observability Deficit**: Most frameworks provide basic logging but lack integrated metrics, tracing, and health checking. Production deployments typically require adding third-party observability solutions.

5. **Graph Database Niche**: Native graph database integration with natural language interfaces remains relatively rare. The Cineca Agentic Platform's Memgraph integration and NL-to-Cypher pipeline address a gap in the ecosystem.

2.2.10 Positioning of the Cineca Agentic Platform

Based on this analysis, the Cineca Agentic Platform occupies a distinct position in the framework landscape:

Target Audience The platform is designed for:

- Enterprise organizations requiring production-grade agent infrastructure
- Research institutions needing multi-tenant access to AI capabilities
- HPC centers offering AI services to diverse user communities
- Organizations with data sovereignty requirements necessitating self-hosted deployment

Differentiation The platform differentiates itself through:

- **First-class multi-tenancy**: Tenant isolation is built into every layer, from authentication through data storage.
- **Comprehensive RBAC**: Fine-grained permissions for tools, models, and administrative functions.
- **Production observability**: Integrated Prometheus metrics, OpenTelemetry tracing, and structured logging.
- **Graph-native knowledge**: Native Memgraph integration with a sophisticated NL-to-Cypher pipeline.
- **MCP-style tooling**: 34 tools across 17 categories with consistent schema-driven invocation.
- **Async job infrastructure**: PostgreSQL-backed job queue with SSE progress streaming.

Complementary Use The platform is not intended to replace libraries like LangChain or LlamaIndex but to provide the production infrastructure around them. Organizations might:

- Use the Cineca Agentic Platform as the deployment and governance layer
- Implement specific agent behaviors using LangChain chains or LlamaIndex retrievers
- Leverage the platform's MCP tools for standardized capability access
- Rely on the platform's observability for production monitoring

This complementary positioning allows organizations to benefit from both the flexibility of library-based development and the robustness of platform-based deployment.

2.3 Model Context Protocol (MCP)

The Model Context Protocol (MCP) represents a significant advancement in standardizing how AI systems interact with external tools and data sources. This section provides background on the MCP specification and its relevance to enterprise agent platforms.

2.3.1 Origins and Motivation

The proliferation of LLM-based applications has led to a fragmented landscape of tool integration approaches. Each framework (LangChain, LlamaIndex, Semantic Kernel, etc.) defines its own abstractions for tools, leading to:

- **Vendor lock-in**: Tools implemented for one framework cannot easily be used with another.
- **Duplicated effort**: Tool authors must implement the same capability multiple times for different frameworks.
- **Inconsistent semantics**: Similar tools behave differently across frameworks due to varying conventions.
- **Integration complexity**: Organizations using multiple AI systems must maintain parallel tool implementations.

The Model Context Protocol, introduced by Anthropic in late 2024 [7] , addresses these challenges by providing a standardized specification for tool discovery, invocation, and result handling. MCP defines a common language that any AI client can use to interact with any MCP-compliant tool server.

2.3.2 Protocol Architecture

MCP follows a client-server architecture where AI applications (clients) communicate with tool providers (servers) using a standardized protocol.

Core Components The MCP architecture consists of:

- **MCP Client**: The AI application that discovers and invokes tools. This is typically an LLM-based agent or assistant application.
- **MCP Server**: A service that exposes one or more tools following the MCP specification. Servers can provide domain-specific capabilities (file systems, databases, APIs, etc.).
- **Transport Layer**: The communication mechanism between clients and servers. MCP supports multiple transports including stdio (for local processes) and HTTP with Server-Sent Events (for remote services).
- **Protocol Messages**: Standardized JSON-RPC 2.0 messages for initialization, capability discovery, tool invocation, and result handling.

Communication Flow A typical MCP interaction follows this sequence:

1. **Initialization**: The client connects to the server and exchanges capability information.
2. **Tool Discovery**: The client requests the list of available tools via the `tools/list` method.

3. **Schema Retrieval**: For each tool, the client receives a JSON Schema describing the tool's parameters and expected behavior.
4. **Tool Invocation**: When the AI determines a tool should be called, the client sends a `tools/call` request with the tool name and arguments.
5. **Result Handling**: The server executes the tool and returns results, which the client incorporates into the AI's context.

2.3.3 Tool Definitions

MCP tools are defined using JSON Schema, providing a machine-readable description that enables:

- **Parameter validation**: Inputs can be validated against the schema before invocation.
- **Documentation generation**: Schema annotations provide human-readable descriptions.
- **Type safety**: Strongly typed parameters reduce runtime errors.
- **Discovery automation**: AI systems can understand tool capabilities without hardcoded knowledge.

Tool Schema Structure A typical MCP tool definition includes:

```json
{
    "name": "graph.query",
    "description": "Execute a Cypher query against the graph database
    ",
    "inputSchema": {
        "type": "object",
        "properties": {
            "cypher": {
                "type": "string",
                "description": "The Cypher query to execute"
            },
            "params": {
                "type": "object",
                "description": "Query parameters",
                "additionalProperties": true
            },
            "timeout_ms": {
                "type": "integer",
                "description": "Query timeout in milliseconds",
                "default": 30000
            }
        },
        "required": ["cypher"]
    }
}
```

Listing 2.1. Example MCP tool schema structure (illustrative)

Capability Annotations Beyond basic schemas, MCP supports capability annotations that describe tool behavior:

- **Read-only vs. mutating**: Whether the tool modifies state.
- **Idempotency**: Whether repeated invocations produce the same result.
- **Required permissions**: What access rights are needed.
- **Rate limits**: Recommended invocation frequency limits.

2.3.4 MCP Servers and Ecosystem

The MCP ecosystem includes both reference implementations and community-contributed servers:

Reference Servers Anthropic provides reference server implementations for common capabilities:

- **Filesystem server**: Reading and writing files in specified directories.
- **GitHub server**: Interacting with GitHub repositories, issues, and pull requests.
- **PostgreSQL server**: Executing queries against PostgreSQL databases.
- **Brave Search server**: Web search using the Brave Search API.

Community Servers The growing MCP community has contributed servers for diverse domains:

- **neo4j-contrib/mcp-neo4j**: Neo4j graph database integration with Cypher query support.
- **Slack server**: Workspace communication and channel management.
- **Google Drive server**: Document access and management.
- **Various API integrations**: Weather, finance, productivity tools, and more.

Server Implementation Patterns MCP servers can be implemented in various ways:

- **Standalone processes**: Long-running services that communicate over stdio or HTTP.
- **Embedded servers**: Libraries that can be embedded in applications to expose local capabilities.
- **Gateway servers**: Proxies that aggregate multiple backend services behind a unified MCP interface.

2.3.5 Advantages for Enterprise Integration

MCP offers several advantages specifically relevant to enterprise AI deployments:

Standardization A common protocol enables:

- **Tool reuse**: Tools implemented once can be used across multiple AI applications.
- **Vendor independence**: Organizations can switch AI platforms without rewriting tool integrations.
- **Ecosystem leverage**: Access to a growing catalog of community-contributed tools.

Security Boundaries MCP's client-server architecture supports:

- **Process isolation**: Tool execution in separate processes limits blast radius.
- **Network segmentation**: Remote servers can run in isolated network zones.
- **Authentication integration**: Transport layer can be secured with existing auth mechanisms.
- **Audit logging**: All tool invocations pass through defined interfaces that can be logged.

Incremental Adoption Organizations can adopt MCP incrementally:

- Start with pre-built servers for common capabilities.
- Gradually implement custom servers for domain-specific tools.
- Migrate existing tool implementations to MCP as needed.

2.3.6 Relationship to the Cineca Agentic Platform

The Cineca Agentic Platform adopts MCP-style patterns for its tool ecosystem while extending them with enterprise-specific features:

MCP-Aligned Design The platform's tool system follows MCP conventions:

- **Schema-driven tools**: All 34 platform tools are defined with JSON Schema specifications.
- **Standard invocation patterns**: Tools follow consistent request/response structures.
- **Discovery mechanisms**: Tools can be listed and queried for their capabilities.
- **Categorical organization**: Tools are organized into 17 categories following MCP conventions.

Enterprise Extensions Beyond MCP basics, the platform adds:

- **RBAC integration**: Tool access is controlled by role-based permissions with scope requirements.
- **Tenant isolation**: Tools operate within tenant boundaries, preventing cross-tenant data access.
- **Audit logging**: All tool invocations are recorded with full context for compliance.
- **Rate limiting**: Per-user and per-tenant rate limits prevent abuse.
- **Metrics and tracing**: Tool invocations are instrumented for observability.

Interoperability Considerations While the platform uses MCP-style patterns internally, full MCP interoperability (allowing external MCP clients to connect) requires additional work:

- Implementing the MCP JSON-RPC transport layer.
- Exposing platform tools through MCP-compliant endpoints.
- Bridging platform authentication with MCP client authentication.

This is noted as a potential future extension, allowing the platform to serve as an MCP server for external AI applications while maintaining its enterprise governance features.

2.3.7 MCP Tool Categories in the Platform

The Cineca Agentic Platform implements 34 MCP-style tools organized into 17 categories. Table 2.3 summarizes these categories and their purposes.

Each tool within these categories follows consistent patterns for:

- Schema definition with JSON Schema
- Capability declaration (read-only, mutating, admin-required)
- Scope requirements for RBAC enforcement
- Timeout configuration for execution limits
- Error handling with structured error responses

This comprehensive tool ecosystem, combined with MCP-style invocation patterns and enterprise governance features, enables the platform to support sophisticated agent workflows while maintaining security and compliance requirements.

Table 2.3. MCP tool categories in the Cineca Agentic Platform.

Category	Description
graph	Cypher query execution, schema introspection, NL-to-Cypher
cache	Redis cache operations (get, set, delete, pattern matching)
data	Data transformation, format conversion, validation
nlp	Natural language processing utilities
security	Token validation, permission checks, PII detection
admin	Administrative operations (tenant management, configuration)
models	Model listing, default resolution, provider status
jobs	Job creation, status, cancellation, result retrieval
sessions	Session management, context storage
audit	Audit log queries, compliance reporting
health	Component health checks, readiness status
metrics	Prometheus metric access, usage statistics
backup	Snapshot creation, restoration, retention management
manifest	Model manifest operations, built-in configurations
ratelimit	Rate limit status, quota management
export	Configuration export and import
batch	Bulk operations for efficiency

2.4 Graph Databases and Natural Language Interfaces

Graph databases represent a powerful paradigm for storing and querying highly connected data. When combined with natural language interfaces powered by LLMs, they enable intuitive access to complex knowledge structures. This section provides the theoretical foundations for the graph-based knowledge querying capabilities of the Cineca Agentic Platform.

2.4.1 Graph Database Fundamentals

Graph databases are designed to store, manage, and query data organized as graphs—collections of nodes (vertices) connected by edges (relationships). Unlike relational databases that organize data into tables with predefined schemas, graph databases naturally represent and traverse relationships.

Property Graph Model The property graph model, used by most modern graph databases, defines:

- **Nodes**: Entities in the domain, each with a unique identifier and optional labels (types).
- **Relationships**: Directed connections between nodes, each with a type and optional properties.
- **Properties**: Key-value pairs attached to both nodes and relationships, storing entity attributes.

- **Labels**: Tags that categorize nodes into types (e.g., `:Person`, `:Workflow`).

This model is particularly well-suited for:

- Social networks and organizational structures
- Knowledge graphs and ontologies
- Recommendation systems
- Fraud detection and network analysis
- Bioinformatics workflows and data lineage

Graph vs. Relational Databases Key differences from relational databases include:

- **Relationship handling**: Relationships are first-class citizens, not foreign key joins.
- **Traversal efficiency**: Graph databases optimize for relationship traversal, with $O(1)$ edge lookup compared to $O(\log n)$ or $O(n)$ for relational joins.
- **Schema flexibility**: Nodes can have varying properties without schema migrations.
- **Query patterns**: Graph queries naturally express path-based and pattern-matching questions.

Use Cases in AI Systems Graph databases complement LLM-based systems by providing:

- **Structured knowledge**: Explicit relationships that LLMs can reason about.
- **Explainable queries**: Results that trace back to specific paths and connections.
- **Dynamic updates**: Knowledge that can be incrementally updated without retraining.
- **Complex queries**: Pattern matching that would be difficult to express in natural language alone.

2.4.2 Cypher Query Language

Cypher is the most widely adopted query language for property graph databases, originally developed for Neo4j and now supported by multiple graph databases including Memgraph, Amazon Neptune, and others through the openCypher standard.

Syntax Overview Cypher uses an ASCII-art style syntax that visually represents graph patterns:

```
// Match all Workflow nodes
MATCH (w:Workflow)
RETURN w.name, w.status

// Find relationships between workflows
MATCH (w1:Workflow)-[r:DEPENDS_ON]->(w2:Workflow)
RETURN w1.name AS source, w2.name AS target, r.type

// Path queries with variable length
MATCH path = (start:Workflow)-[:DEPENDS_ON*1..3]->(end:Workflow)
WHERE start.name = 'DataPrep'
RETURN path

// Aggregation and filtering
MATCH (w:Workflow)-[:RUNS_ON]->(n:Node)
WHERE n.gpu_count > 0
RETURN n.name, COUNT(w) AS workflow_count
ORDER BY workflow_count DESC
```

Listing 2.2. Basic Cypher query examples

Key Cypher Clauses

- `MATCH`: Specifies the graph pattern to search for.
- `WHERE`: Filters matched patterns based on conditions.
- `RETURN`: Specifies what data to return from matched patterns.
- `CREATE`: Creates new nodes and relationships.
- `MERGE`: Creates entities only if they don't exist (upsert semantics).
- `DELETE`: Removes nodes and relationships.
- `SET`: Updates properties on nodes and relationships.
- `WITH`: Chains query parts together, passing results forward.
- `CALL`: Invokes stored procedures or user-defined functions.

Advantages of Cypher Cypher's design offers several advantages:

- **Readability**: Pattern syntax is intuitive and visually represents graph structure.
- **Expressiveness**: Complex traversals and pattern matching in concise queries.
- **Composability**: Queries can be built up incrementally using `WITH` clauses.
- **Standardization**: openCypher provides cross-database portability.

2.4.3 Memgraph and Neo4j

Two prominent graph database systems are particularly relevant to enterprise AI applications:

Neo4j Neo4j [30] is the most widely deployed graph database, offering:

- **ACID transactions**: Full transactional support for data integrity.
- **Native graph storage**: Purpose-built storage engine optimized for graphs.
- **Enterprise features**: Clustering, security, monitoring, and commercial support.
- **Rich ecosystem**: Extensive tooling, visualization, and integration options.
- **Aura cloud service**: Managed cloud offering for simplified deployment.

Neo4j's integration with AI systems is well-developed, with official LangChain components (`GraphCypherQAChain`) and LlamaIndex integration (`KnowledgeGraphIndex`).

Memgraph Memgraph [8] is an in-memory graph database designed for real-time analytics and low-latency queries:

- **In-memory storage**: Sub-millisecond query latency for most operations.
- **Cypher compatibility**: Full openCypher support with extensions.
- **Streaming support**: Native integration with Apache Kafka and other stream sources.
- **MAGE library**: Collection of graph algorithms and utilities.
- **Open source core**: Community edition available under open source license.

The Cineca Agentic Platform uses Memgraph as its graph database, selected for:

- Low-latency queries suitable for interactive agent workflows
- In-memory performance for exploration and prototyping
- openCypher compatibility ensuring query portability
- Simpler operational model compared to distributed databases

Platform-Specific Considerations When integrating graph databases into AI agent systems, several considerations apply:

- **Schema enforcement**: While graph databases are schema-flexible, production deployments benefit from schema constraints to ensure data quality.
- **Query safety**: User-generated or AI-generated queries must be validated to prevent injection attacks and resource exhaustion.

- **Tenant isolation**: Multi-tenant systems require mechanisms to restrict queries to tenant-specific data.
- **Query optimization**: Complex pattern matches can be expensive; monitoring and optimization are essential.

2.4.4 Natural Language to Cypher Translation

The challenge of translating natural language questions into Cypher queries is a key problem in making graph databases accessible to non-technical users and AI agents.

Problem Definition Given:

- A natural language question (e.g., "Which workflows depend on DataPrep?")
- A graph schema describing available node types, relationship types, and properties
- Optionally, example queries and domain-specific conventions

The task is to generate a syntactically correct and semantically appropriate Cypher query that answers the question.

Challenges NL-to-Cypher translation presents several challenges:

1. **Schema understanding**: The translator must know what entities and relationships exist in the graph.
2. **Entity resolution**: Natural language references ("the preprocessing workflow") must be resolved to specific graph entities.
3. **Relationship inference**: Implicit relationships in natural language must be mapped to explicit graph relationships.
4. **Aggregation interpretation**: Questions about "how many" or "most common" require appropriate aggregation functions.
5. **Path queries**: Questions about connections or dependencies require variable-length path patterns.
6. **Ambiguity handling**: Natural language is often ambiguous; the system must make reasonable interpretations or ask for clarification.

Approaches Several approaches have been developed for NL-to-Cypher translation:

- **Rule-based systems**: Hand-crafted rules mapping linguistic patterns to query templates. High precision for covered patterns but limited coverage.
- **Semantic parsing**: Formal methods that build logical representations from natural language, then translate to Cypher. Requires linguistic expertise and domain modeling.

- **Neural sequence-to-sequence**: End-to-end neural models trained on (question, query) pairs. Requires substantial training data for each domain.
- **LLM-based generation**: Using large language models to generate Cypher directly from natural language, with schema information provided in the prompt.

The LLM-based approach has become dominant due to its flexibility and zero-shot capability. However, it introduces new challenges around query correctness, safety, and reliability that require additional infrastructure.

LLM-Based NL-to-Cypher Pipeline A typical LLM-based NL-to-Cypher pipeline includes:

1. **Intent classification**: Determining whether the question is graph-related and what type of answer is expected.
2. **Schema retrieval**: Fetching relevant portions of the graph schema to include in the LLM prompt.
3. **Prompt construction**: Building a prompt that includes the question, schema, examples, and generation instructions.
4. **Query generation**: Invoking the LLM to generate a Cypher query.
5. **Query validation**: Checking the generated query for syntax errors, safety issues, and schema compliance.
6. **Query execution**: Running the validated query against the database.
7. **Result summarization**: Formatting query results into a natural language answer.

Safety Considerations LLM-generated queries require careful safety measures:

- **Syntax validation**: Parse the query to ensure it is syntactically valid before execution.
- **Read-only enforcement**: For analytical queries, ensure no `CREATE`, `DELETE`, or `SET` clauses are present.
- **Resource limits**: Set query timeouts and result size limits to prevent resource exhaustion.
- **Tenant scoping**: Automatically inject tenant filters to prevent cross-tenant data access.
- **Procedure allowlisting**: Restrict `CALL` clauses to a whitelist of safe procedures.
- **Pattern complexity limits**: Prevent overly complex patterns that could cause combinatorial explosion.

2.4.5 Graph Schema for Bioinformatics Workflows

The Cineca Agentic Platform implements a graph schema designed for bioinformatics and computational workflow management. This domain-specific schema demonstrates how graph databases can model complex scientific workflows.

Node Types The platform's Memgraph schema includes 14 node types:

- **Workflow**: Represents a computational workflow or pipeline.
- **Step**: Individual steps within a workflow.
- **Input/Output**: Data inputs and outputs for steps.
- **Tool**: Software tools or executables used in steps.
- **Resource**: Computational resources (CPU, memory, GPU).
- **Node**: Compute nodes in the HPC cluster.
- **Dataset**: Data collections used or produced by workflows.
- **User**: Users who own or execute workflows.
- **Project**: Organizational groupings of workflows.
- **Publication**: Scientific publications related to workflows.
- Additional domain-specific types for bioinformatics entities.

Relationship Types Four primary relationship types connect these entities:

- **DEPENDS_ON**: Workflow or step dependencies (execution order).
- **USES**: Tool or resource utilization relationships.
- **PRODUCES/CONSUMES**: Data flow relationships.
- **OWNED_BY/BELONGS_TO**: Organizational relationships.

This schema enables queries such as:

- "What workflows depend on the BLAST tool?"
- "Which datasets were produced by last week's workflows?"
- "What is the resource utilization pattern for GPU nodes?"
- "Show the data lineage for dataset X."

Schema-Aware Query Generation The platform's NL-to-Cypher pipeline is schema-aware: it retrieves the current schema from Memgraph and includes relevant portions in the LLM prompt. This ensures that generated queries reference actual node types, relationship types, and properties, reducing hallucination and improving query validity.

2.4.6 Comparison with Document-Based Retrieval

An alternative to graph-based knowledge access is document-based retrieval augmented generation (RAG), which is discussed further in Section 2.5. Key differences include:

Table 2.4. Comparison of graph-based querying vs. document-based RAG.

Aspect	Graph-Based (NL-to-Cypher)	Document-Based (RAG)
Knowledge representation	Structured nodes and relationships	Unstructured text chunks
Query mechanism	Pattern matching, traversal	Semantic similarity search
Precision	High for structured queries	Variable, depends on retrieval quality
Relationship handling	Explicit, first-class	Implicit in text, may be missed
Update cost	Incremental node/edge updates	Requires re-embedding
Explainability	Query path is explicit	Retrieval sources can be shown
Domain modeling effort	Requires schema design	Minimal, ingest documents directly
Best for	Structured domains, relationships	Unstructured knowledge, broad coverage

The Cineca Agentic Platform focuses on graph-based knowledge access, which aligns with its use case of managing structured bioinformatics workflows and HPC resources. Document-based RAG is noted as a potential future extension for handling unstructured documentation and research papers.

2.5 Retrieval-Augmented Generation and Knowledge Graphs

Retrieval-Augmented Generation (RAG) has emerged as a dominant paradigm for grounding LLM outputs in external knowledge. This section introduces RAG concepts, compares document-based retrieval with knowledge graph approaches, and explains why the Cineca Agentic Platform focuses on graph-native knowledge access.

2.5.1 Retrieval-Augmented Generation Overview

RAG [34] addresses a fundamental limitation of LLMs: their knowledge is frozen at training time and may become outdated, incomplete, or inaccurate. By retrieving relevant information from external sources and incorporating it into the generation context, RAG enables:

- **Knowledge currency**: Access to up-to-date information beyond the training cutoff.

- **Factual grounding**: Reduced hallucination by anchoring responses in retrieved evidence.
- **Domain adaptation**: Specialization to specific domains without fine-tuning.
- **Source attribution**: Ability to cite sources for generated claims.

Standard RAG Architecture The canonical RAG pipeline consists of:

1. **Document ingestion**: Loading documents from various sources (files, databases, web pages).
2. **Chunking**: Splitting documents into smaller segments suitable for embedding and retrieval.
3. **Embedding**: Converting chunks into dense vector representations using embedding models.
4. **Indexing**: Storing embeddings in a vector database for efficient similarity search.
5. **Query embedding**: Converting the user's question into the same embedding space.
6. **Retrieval**: Finding the most similar chunks to the query embedding.
7. **Context construction**: Assembling retrieved chunks into a prompt context.
8. **Generation**: Invoking the LLM with the augmented context to produce a response.

RAG Variants Several variants of RAG have been developed to address different requirements:

- **Naive RAG**: The basic pipeline described above, with simple top-k retrieval.
- **Advanced RAG**: Incorporates pre-retrieval query rewriting, post-retrieval reranking, and iterative retrieval.
- **Modular RAG**: Decomposes the pipeline into interchangeable modules that can be customized for specific use cases.
- **Self-RAG**: The model decides when retrieval is needed and evaluates retrieval quality.
- **Graph RAG**: Combines document retrieval with knowledge graph traversal for enhanced context.

2.5.2 Knowledge Graphs in AI Systems

Knowledge graphs provide an alternative paradigm for organizing and accessing knowledge. Rather than storing unstructured text, knowledge graphs represent information as structured entities and relationships.

Knowledge Graph Characteristics

- **Explicit relationships**: Connections between entities are first-class citizens, not implicit in text.
- **Semantic types**: Entities are categorized with types (ontology classes), enabling type-based reasoning.
- **Multi-hop queries**: Questions requiring traversal across multiple relationships can be answered directly.
- **Logical consistency**: Constraints can enforce consistency across the knowledge base.
- **Incremental updates**: New knowledge can be added without reprocessing existing content.

Knowledge Graph Construction Building knowledge graphs involves:

- **Schema design**: Defining the ontology of entity types and relationship types.
- **Entity extraction**: Identifying entities in source documents or data.
- **Relationship extraction**: Identifying relationships between extracted entities.
- **Entity resolution**: Merging references to the same real-world entity.
- **Knowledge validation**: Checking extracted knowledge for accuracy and consistency.

These processes can be manual, automated (using NLP and machine learning), or LLM-assisted.

2.5.3 Comparison: RAG over Documents vs. NL-to-Cypher over Graphs

The Cineca Agentic Platform implements NL-to-Cypher for graph-based knowledge access rather than traditional document-based RAG. This section analyzes the trade-offs between these approaches.

Document-Based RAG **Advantages:**

- **Low barrier to entry**: Documents can be ingested directly without schema design.
- **Broad coverage**: Can incorporate any text-based knowledge source.
- **Semantic matching**: Embedding-based retrieval handles synonyms and paraphrases naturally.

- **Mature tooling**: Extensive ecosystem of embedding models, vector databases, and frameworks.

Disadvantages:

- **Lost structure**: Relationships between entities may be scattered across chunks.
- **Retrieval noise**: Semantic similarity may retrieve tangentially related content.
- **Multi-hop difficulty**: Questions requiring reasoning across multiple documents are challenging.
- **Update overhead**: Adding new documents requires embedding and re-indexing.

Graph-Based NL-to-Cypher **Advantages:**

- **Explicit relationships**: Queries can directly traverse and filter on relationships.
- **Precise answers**: Structured queries return exact matches, not probabilistic retrievals.
- **Complex queries**: Aggregations, path queries, and multi-hop traversals are natural.
- **Explainability**: Query results can be traced to specific graph paths.
- **Incremental updates**: New nodes and edges can be added without reprocessing.

Disadvantages:

- **Schema requirement**: Requires upfront domain modeling and schema design.
- **Ingestion complexity**: Data must be transformed into graph structure.
- **Query correctness**: Generated Cypher may be syntactically or semantically incorrect.
- **Coverage limitations**: Only knowledge in the graph is accessible.

Comparative Summary Table 2.5 provides a detailed comparison across key dimensions:

2.5.4 Platform Design Choice: Graph-Native Approach

The Cineca Agentic Platform adopts a graph-native approach for knowledge access rather than implementing traditional document-based RAG. This design choice is motivated by several factors:

Table 2.5. Detailed comparison of document-based RAG vs. graph-based NL-to-Cypher.

Dimension	Document RAG	NL-to-Cypher
Setup complexity	Low	Medium-High
Knowledge structure	Implicit	Explicit
Relationship queries	Difficult	Native
Aggregation queries	Difficult	Native
Semantic flexibility	High	Low
Answer precision	Variable	High
Hallucination risk	Moderate	Low (with validation)
Explainability	Moderate	High
Update model	Re-embed chunks	Add nodes/edges
Tooling maturity	High	Growing

Domain Characteristics The platform's primary domain—bioinformatics workflows and HPC resource management—is inherently structured:

- Workflows have explicit dependencies and execution orders.
- Resources are connected to nodes, queues, and allocations.
- Data lineage forms natural graph structures.
- Users, projects, and organizations have hierarchical relationships.

This structured domain is well-suited to graph representation, where relationships are as important as entities.

Query Requirements Typical queries in this domain require capabilities that graph databases excel at:

- "What workflows depend on this input dataset?" (relationship traversal)
- "Which GPU nodes have run more than 100 jobs this month?" (aggregation with filtering)
- "Show the data lineage path from raw sequencing data to the final publication." (path queries)
- "Which workflows would be affected if we upgrade the BLAST tool?" (impact analysis)

These queries would be difficult to answer reliably with document retrieval but are natural to express in Cypher.

Precision Requirements In scientific and operational contexts, answer precision is critical:

- Incorrect dependency information could lead to workflow failures.

- Inaccurate resource utilization data could affect scheduling decisions.
- Wrong data lineage could compromise reproducibility.

The deterministic nature of graph queries, combined with validation and safety checks, provides higher confidence than probabilistic document retrieval.

Future Extension: Hybrid Approach The platform architecture does not preclude adding document-based RAG in the future. A hybrid approach could:

- Use graph queries for structured operational data.
- Use document RAG for unstructured content (publications, documentation, user notes).
- Combine results through a unified response synthesis layer.

This is noted as a potential enhancement in the platform's roadmap, allowing it to handle both structured and unstructured knowledge access.

2.5.5 Graph RAG: Emerging Hybrid Approaches

Recent research has explored combining graph and document-based approaches in various ways:

Document Graphs Some approaches build graphs from document collections:

- Nodes represent documents, paragraphs, or entities.
- Edges represent citations, semantic similarity, or extracted relationships.
- Traversal enhances retrieval by following relevant connections.

Knowledge-Enhanced RAG Other approaches use knowledge graphs to enhance document retrieval:

- Entity linking connects document chunks to knowledge graph entities.
- Graph context provides additional information about retrieved entities.
- Relationship paths guide multi-hop reasoning across documents.

LLM-Constructed Graphs LLMs can construct knowledge graphs from documents:

- Entity and relationship extraction using LLM prompting.
- Iterative graph construction as documents are processed.
- Dynamic updates as new information becomes available.

These hybrid approaches represent an active area of research that may inform future platform enhancements. The current platform architecture, with its separation of concerns between the orchestrator, tool layer, and data layer, would accommodate such extensions without fundamental redesign.

2.5.6 Implications for Thesis Scope

This thesis focuses on the graph-native NL-to-Cypher pipeline implemented in the Cineca Agentic Platform. The pipeline includes:

1. Intent classification to identify graph-related queries.
2. Schema-aware prompt construction for Cypher generation.
3. Multi-layer safety validation for generated queries.
4. Query execution with tenant isolation and resource limits.
5. Result summarization for natural language responses.

This pipeline is detailed in Section 5.4 (Chapter 5), with evaluation results in Chapter 12. Document-based RAG is explicitly out of scope for this thesis but is acknowledged as a complementary capability for future work.

2.6 Security and Multi-Tenancy Foundations

Security and multi-tenancy are critical concerns for enterprise AI agent platforms. This section provides the conceptual foundations for authentication, authorization, tenant isolation, and audit logging that underpin the Cineca Agentic Platform's security architecture. Implementation details specific to the platform are presented in Chapter 8.

2.6.1 Authentication Fundamentals

Authentication is the process of verifying the identity of a user, service, or system attempting to access resources. Modern distributed systems typically rely on token-based authentication standards.

OAuth 2.0 OAuth 2.0 [31] is an authorization framework that enables third-party applications to obtain limited access to resources on behalf of users. Key concepts include:

- **Resource Owner**: The user who owns the protected resources.
- **Client**: The application requesting access on behalf of the resource owner.
- **Authorization Server**: Issues tokens after authenticating the resource owner.
- **Resource Server**: Hosts protected resources and validates access tokens.
- **Access Token**: A credential representing the granted authorization.
- **Refresh Token**: A long-lived credential used to obtain new access tokens.

OAuth 2.0 defines several grant types for different use cases:

- **Authorization Code**: For server-side applications with user interaction.
- **Client Credentials**: For machine-to-machine authentication without user involvement.
- **Resource Owner Password**: For trusted applications (deprecated for most uses).
- **Implicit**: For browser-based applications (deprecated in favor of PKCE).

OpenID Connect (OIDC) OpenID Connect [10] is an identity layer built on top of OAuth 2.0. While OAuth 2.0 provides authorization (access to resources), OIDC adds authentication (identity verification):

- **ID Token**: A JSON Web Token (JWT) containing claims about the authenticated user.
- **UserInfo Endpoint**: An API endpoint returning claims about the authenticated user.
- **Standard Claims**: Predefined claim types (sub, name, email, etc.) for interoperability.
- **Discovery**: A well-known endpoint for discovering provider configuration.

OIDC enables federated identity, where users authenticate with an identity provider (IdP) and access resources across multiple applications.

JSON Web Tokens (JWT) JWTs [15] are a compact, URL-safe format for representing claims between parties. A JWT consists of three parts:

1. **Header**: Specifies the token type and signing algorithm.
2. **Payload**: Contains claims (statements about the subject and metadata).
3. **Signature**: Cryptographic signature for integrity verification.

JWT claims can be:

- **Registered claims**: Standardized claims like `iss` (issuer), `sub` (subject), `exp` (expiration).
- **Public claims**: Claims registered in the IANA JWT Claims Registry.
- **Private claims**: Application-specific claims agreed upon by parties.

Token Validation Validating JWTs involves several checks:

1. **Signature verification**: Using the issuer's public key (from JWKS endpoint).
2. **Expiration check**: Ensuring the token has not expired (`exp` claim).
3. **Issuer verification**: Confirming the token comes from a trusted issuer.
4. **Audience verification**: Ensuring the token is intended for this application.
5. **Claim validation**: Checking application-specific claims (e.g., scopes, roles).

The JSON Web Key Set (JWKS) endpoint provides public keys for signature verification. Clients typically cache these keys and refresh them periodically or when signature verification fails.

2.6.2 Role-Based Access Control (RBAC)

RBAC is an access control paradigm where permissions are assigned to roles, and users are assigned to roles. This indirection simplifies permission management in systems with many users.

RBAC Components

- **Users**: Individual identities (human or machine) that authenticate to the system.
- **Roles**: Named collections of permissions that represent job functions or responsibilities.
- **Permissions**: Fine-grained authorizations to perform specific operations on specific resources.
- **Sessions**: Active user sessions during which role assignments apply.

RBAC Models NIST defines several RBAC models with increasing complexity [32] :

- **Core RBAC**: Basic user-role and permission-role assignments.
- **Hierarchical RBAC**: Roles can inherit permissions from other roles.
- **Constrained RBAC**: Adds constraints such as separation of duties.
- **Symmetric RBAC**: Adds permission-role review and user-role review.

Scope-Based Authorization Many modern systems implement authorization through scopes—named permissions that can be granted to applications or users:

- Scopes are strings like `read:users`, `write:workflows`, or `admin:all`.
- Access tokens include a list of granted scopes.
- API endpoints declare required scopes; requests are rejected if scopes are insufficient.
- Scopes can be hierarchical (e.g., `admin:all` implies all other scopes).

This approach combines well with OAuth 2.0, where scopes are explicitly part of the authorization grant.

RBAC in AI Agent Systems AI agent systems present unique RBAC challenges:

- **Tool permissions**: Which tools can each user/role invoke?
- **Model access**: Which LLM models can each user/role use?
- **Data access**: What data can agents access on behalf of users?
- **Delegation**: Can agents perform actions the user cannot?
- **Audit attribution**: Who is responsible for agent actions?

These considerations inform the design of the Cineca Agentic Platform's authorization system.

2.6.3 Multi-Tenant Architecture Patterns

Multi-tenancy refers to a software architecture where a single instance of the software serves multiple customers (tenants), with appropriate isolation between them.

Tenancy Models Different levels of isolation are possible:

- **Shared everything**: All tenants share the same database, schemas, and application instances. Isolation is enforced through data tagging and query filtering.
- **Shared database, separate schemas**: Tenants share a database server but have separate schemas or table prefixes. Provides stronger isolation with moderate overhead.
- **Separate databases**: Each tenant has a dedicated database. Provides strongest isolation but highest operational overhead.
- **Separate instances**: Each tenant has dedicated application and database instances. Maximum isolation, typically for high-value or regulated tenants.

Isolation Dimensions Multi-tenant systems must isolate tenants across multiple dimensions:

- **Data isolation**: Tenant data must not leak to other tenants.
- **Performance isolation**: One tenant's workload should not degrade others' performance.
- **Configuration isolation**: Tenants may have different settings and preferences.
- **Billing isolation**: Resource usage must be tracked per tenant for billing.
- **Security isolation**: Security incidents in one tenant should not affect others.

Implementation Patterns Common patterns for implementing multi-tenancy include:

- **Tenant ID column**: Every data table includes a `tenant_id` column, and all queries filter by this column.
- **Row-level security**: Database-level policies automatically filter rows based on session context.
- **Connection routing**: Database connections are routed to tenant-specific databases or schemas.
- **Middleware injection**: Request middleware extracts tenant identity and injects it into the request context.
- **Namespace prefixing**: Keys in caches or queues are prefixed with tenant identifiers.

Multi-Tenancy in AI Systems AI agent platforms add additional multi-tenancy considerations:

- **Model isolation**: Should tenants share LLM instances or have dedicated allocations?
- **Context isolation**: Agent memory and conversation history must be tenant-scoped.
- **Tool isolation**: Tool configurations and credentials may differ per tenant.
- **Cost allocation**: LLM API costs must be tracked and attributed to tenants.
- **Quota management**: Rate limits and usage quotas may differ per tenant.

2.6.4 Audit Logging and Compliance

Audit logging is the systematic recording of security-relevant events for compliance, forensics, and monitoring purposes.

Audit Log Requirements Comprehensive audit logs should capture:

- **Who**: The authenticated identity performing the action.
- **What**: The specific action performed (CRUD operation, tool invocation, etc.).
- **When**: Timestamp with sufficient precision and time zone information.
- **Where**: The resource or object affected by the action.
- **How**: Additional context (IP address, user agent, request ID).
- **Outcome**: Whether the action succeeded or failed, and any error details.

Audit Log Properties Effective audit logs have several properties:

- **Immutability**: Once written, audit records should not be modifiable.
- **Completeness**: All security-relevant events should be captured.
- **Integrity**: Logs should be tamper-evident (e.g., through cryptographic chaining).
- **Availability**: Logs should be retained for required periods and accessible for review.
- **Searchability**: Logs should be queryable by various criteria (user, time range, action type).

Compliance Frameworks Various compliance frameworks mandate audit logging:

- **GDPR**: Requires logging of data access and processing activities for personal data.
- **SOC 2**: Requires logging of security events and access controls.
- **HIPAA**: Requires audit trails for access to protected health information.
- **PCI DSS**: Requires logging of all access to cardholder data.
- **ISO 27001**: Requires event logging and log protection as part of security controls.

Audit Logging in AI Systems AI agent systems introduce specific audit requirements:

- **LLM invocations**: Logging prompts and responses (with PII redaction).
- **Tool invocations**: Logging tool inputs, outputs, and execution metadata.
- **Agent decisions**: Logging the reasoning trace that led to specific actions.
- **Data access**: Logging what data the agent accessed on behalf of users.
- **Cost events**: Logging token usage and API costs for billing reconciliation.

2.6.5 Security Challenges in AI Agent Systems

AI agent systems face unique security challenges beyond traditional web applications:

Prompt Injection Prompt injection attacks attempt to manipulate the LLM's behavior by embedding malicious instructions in user input or retrieved content:

- **Direct injection**: Malicious instructions in user messages.
- **Indirect injection**: Malicious content in retrieved documents or tool outputs.
- **Goal hijacking**: Redirecting the agent toward attacker-chosen objectives.
- **Privilege escalation**: Tricking the agent into invoking privileged tools.

Defenses include input validation, output filtering, privilege separation, and human-in-the-loop approval for sensitive actions.

Data Exfiltration Agents with tool access may be manipulated to exfiltrate sensitive data:

- Encoding data in tool parameters (e.g., embedding secrets in URLs).
- Requesting unnecessary data and including it in responses.
- Using communication tools to send data to external parties.

Defenses include restricting tool capabilities, monitoring tool usage patterns, and implementing data loss prevention (DLP) measures.

Excessive Agency Agents may take actions beyond what users intend:

- Misinterpreting instructions and taking incorrect actions.
- Executing irreversible operations without confirmation.
- Acting on outdated or incorrect context.

Defenses include confirmation workflows for sensitive actions, undo capabilities, and clear scope limitations.

Model Vulnerabilities The underlying LLMs may have vulnerabilities:

- Jailbreaking techniques that bypass safety guardrails.
- Training data extraction attacks.
- Adversarial inputs that cause unexpected behavior.

Defenses include using models with robust safety training, implementing output filtering, and maintaining defense in depth.

2.6.6 Security Principles for Enterprise AI Platforms

Based on these challenges, enterprise AI platforms should adhere to several security principles:

Defense in Depth Multiple layers of security controls ensure that no single failure compromises the system:

- Authentication at the perimeter.
- Authorization at the API layer.
- Validation at the tool layer.
- Filtering at the output layer.

Principle of Least Privilege Users and agents should have only the minimum permissions necessary:

- Fine-grained scopes for tool access.
- Tenant-scoped data access by default.
- Explicit elevation for administrative operations.

Secure by Default Default configurations should be secure:

- Authentication required unless explicitly disabled.
- Restrictive permissions until explicitly granted.
- Logging enabled for all security-relevant events.

Auditability All actions should be traceable and attributable:

- Comprehensive audit logging.
- Request ID correlation across services.
- Immutable audit records.

These principles guide the security design of the Cineca Agentic Platform, as detailed in Chapter 8.

2.7 Observability in Distributed Systems

Observability is a critical concern for operating complex distributed systems. This section introduces the conceptual foundations of observability—metrics, logs, and traces—along with the tools and patterns commonly used to implement them. Platform-specific implementation details are presented in Chapter 9.

2.7.1 The Three Pillars of Observability

Observability refers to the ability to understand the internal state of a system by examining its external outputs. In distributed systems, observability is typically achieved through three complementary signals:

Metrics Metrics are numerical measurements collected over time, typically aggregated into time series. They answer questions about system behavior at an aggregate level:

- How many requests per second is the system handling?
- What is the 99th percentile latency?
- What percentage of requests are failing?
- How much memory/CPU is being used?

Metrics are characterized by:

- **Low cardinality**: Metrics are aggregated, so storage grows slowly with traffic.
- **Fixed schema**: Metric names and label sets are predefined.
- **Statistical aggregation**: Counts, gauges, histograms, and summaries.
- **Alerting**: Metrics power alerting rules for anomaly detection.

Logs Logs are timestamped records of discrete events. They provide detailed information about what happened at specific moments:

- Request received with parameters X, Y, Z.
- Error occurred: connection timeout to database.
- User authenticated successfully.
- Background job completed in 3.2 seconds.

Logs are characterized by:

- **High cardinality**: Each event can have unique attributes.
- **Flexible schema**: Log content can vary by event type.
- **Full context**: Logs can include detailed error messages, stack traces, and payloads.
- **Forensics**: Logs enable after-the-fact investigation of issues.

Traces Traces capture the path of a request through a distributed system, showing how different services interact to fulfill a request:

- Request entered at API gateway.
- API gateway routed to service A.
- Service A called service B and database C in parallel.
- Response returned after 45ms total.

Traces are characterized by:

- **Causality**: Traces show parent-child relationships between operations.
- **Timing**: Each span includes start time and duration.
- **Context propagation**: Trace context is passed between services.
- **Root cause analysis**: Traces help identify which service caused delays or failures.

2.7.2 Metrics: Prometheus and Beyond

Prometheus [16] has become the de facto standard for metrics in cloud-native systems, particularly in Kubernetes environments.

Prometheus Architecture Prometheus follows a pull-based model:

- **Targets**: Services expose metrics at an HTTP endpoint (typically `/metrics`).
- **Scraping**: Prometheus periodically fetches metrics from configured targets.
- **Storage**: Metrics are stored in a time-series database.
- **Querying**: PromQL provides a powerful query language for analysis.
- **Alerting**: Alertmanager handles alert routing and notification.

Metric Types Prometheus defines four metric types:

- **Counter**: A cumulative value that only increases (e.g., total requests).
- **Gauge**: A value that can increase or decrease (e.g., current memory usage).
- **Histogram**: Observations counted into configurable buckets (e.g., request latency distribution).
- **Summary**: Similar to histogram but calculates quantiles client-side.

Labels and Cardinality Prometheus metrics can have labels (key-value pairs) that add dimensions:

```
# Counter with labels
http_requests_total{method="GET", path="/api/v1/users", status="200"}
    12345

# Histogram with labels
http_request_duration_seconds_bucket{method="POST", le="0.5"} 1000
http_request_duration_seconds_bucket{method="POST", le="1.0"} 1500
```

Listing 2.3. Example Prometheus metrics with labels

High-cardinality labels (e.g., user IDs, request IDs) should be avoided as they create many time series, increasing storage and query costs.

Grafana Visualization Grafana [33] is commonly paired with Prometheus for visualization:

- Dashboard creation with multiple panels.
- PromQL integration for querying metrics.
- Alerting integration with various notification channels.
- Template variables for dynamic dashboards.

Metrics in AI Systems AI agent platforms require metrics specific to their domain:

- LLM invocation counts and latencies.
- Token usage (input and output tokens).
- Tool invocation success/failure rates.
- Agent run completion times and step counts.
- Model provider health status.
- Rate limit utilization.

2.7.3 Logging: Structured and Correlated

Modern logging practices have evolved from unstructured text to structured, machine-parseable formats.

Structured Logging Structured logs encode events as key-value pairs or JSON objects:

```
{
    "timestamp": "2025-01-15T10:30:45.123Z",
    "level": "INFO",
    "message": "Request completed",
    "request_id": "abc-123",
    "method": "POST",
    "path": "/api/v1/agent-runs",
    "status": 201,
    "duration_ms": 156,
    "user_id": "user-456"
}
```

Listing 2.4. Structured log example

Structured logs enable:

- Efficient parsing and indexing.
- Consistent field names across services.
- Faceted search and filtering.
- Correlation by request ID or trace ID.

Log Levels Standard log levels convey severity:

- **DEBUG**: Detailed information for debugging (typically disabled in production).
- **INFO**: Normal operational events.
- **WARNING**: Unexpected but recoverable situations.
- **ERROR**: Failures that affect individual requests.
- **CRITICAL**: Severe failures affecting system operation.

Log Correlation In distributed systems, correlating logs across services is essential:

- **Request ID**: A unique identifier assigned at the edge and propagated through all services.
- **Trace ID**: A distributed tracing identifier that links logs to traces.
- **Session ID**: Links logs across multiple requests in a user session.

Log Management Production systems typically use log aggregation solutions:

- **ELK Stack**: Elasticsearch, Logstash, Kibana.
- **Loki**: Prometheus-style logging by Grafana Labs.
- **Cloud services**: AWS CloudWatch, Google Cloud Logging, Azure Monitor.

Logging in AI Systems AI agent platforms have specific logging considerations:

- **PII handling**: Prompts and responses may contain personal information.
- **Size management**: LLM interactions can be verbose; selective logging is needed.
- **Audit requirements**: Security-relevant events must be logged immutably.
- **Debug visibility**: Reasoning traces help understand agent behavior.

2.7.4 Distributed Tracing: OpenTelemetry

OpenTelemetry (OTEL) [17] has emerged as the vendor-neutral standard for distributed tracing (and increasingly for metrics and logs as well).

OpenTelemetry Concepts

- **Trace**: The complete path of a request through the system.
- **Span**: A single operation within a trace, with start time, duration, and attributes.
- **Context**: Metadata propagated between services (trace ID, span ID, flags).
- **Exporter**: Component that sends telemetry to backends (Jaeger, Zipkin, cloud services).
- **Instrumentation**: Code that creates spans (auto-instrumentation or manual).

Context Propagation Trace context is propagated through:

- HTTP headers (W3C Trace Context standard).
- Message queue metadata.
- gRPC metadata.
- Custom protocols.

Sampling Not all traces need to be collected:

- **Head-based sampling**: Decision made at trace start (e.g., sample 10% of traces).
- **Tail-based sampling**: Decision made after trace completes (e.g., keep all error traces).
- **Adaptive sampling**: Sampling rate adjusts based on traffic.

Production systems typically sample traces to control costs while ensuring error traces are always captured.

Tracing Backends Popular tracing backends include:

- **Jaeger**: Open-source, CNCF-graduated project.
- **Zipkin**: Open-source, originated at Twitter.
- **Tempo**: Grafana Labs' scalable tracing backend.
- **Cloud services**: AWS X-Ray, Google Cloud Trace, Azure Application Insights.

Tracing in AI Systems AI agent platforms benefit from tracing for:

- Visualizing the flow through agent steps.
- Identifying slow LLM calls or tool invocations.
- Understanding retry and fallback behavior.
- Correlating frontend requests to backend operations.

2.7.5 Health Checks and Probes

Health checks enable orchestration systems to manage application lifecycle and traffic routing.

Kubernetes Probe Types Kubernetes defines three probe types:

- **Liveness probe**: Determines if the container is running. Failure triggers container restart.
- **Readiness probe**: Determines if the container is ready to receive traffic. Failure removes the pod from load balancer endpoints.
- **Startup probe**: Determines if the application has started. Used for slow-starting applications to prevent premature liveness failures.

Probe Implementation Probes are typically implemented as:

- **HTTP GET**: Check returns 2xx status code.
- **TCP socket**: Check opens a connection.
- **Command execution**: Check runs a command in the container.

Health Check Design Effective health checks follow several principles:

- **Liveness should be simple**: Check that the process is responsive, not full dependency health.
- **Readiness should check dependencies**: Check that the application can actually handle requests (database connected, caches warm, etc.).
- **Avoid cascading failures**: A dependency being down should affect readiness, not liveness.
- **Fast response**: Health checks should respond quickly to avoid timeout issues.
- **Appropriate failure thresholds**: Configure failure thresholds to avoid flapping.

Component Health Complex applications may have multiple components with independent health:

- Database connectivity.
- Cache connectivity.
- External API availability.
- Background worker status.
- Model provider health.

Aggregating component health into overall readiness requires policy decisions about which components are critical.

2.7.6 Observability Patterns for AI Platforms

AI agent platforms present specific observability challenges and patterns:

LLM Observability Monitoring LLM interactions requires:

- **Latency tracking**: Per-model, per-provider latency distributions.
- **Token counting**: Input and output token usage for cost tracking.
- **Error classification**: Distinguishing rate limits, timeouts, and content errors.
- **Quality metrics**: Optional evaluation of response quality.

Agent Observability Monitoring agent behavior requires:

- **Run metrics**: Completion rate, step count, duration.
- **Tool metrics**: Invocation counts, success rates, latencies.
- **Reasoning visibility**: Access to agent decision traces.
- **Cost allocation**: Per-run, per-user, per-tenant cost tracking.

Multi-Tenant Observability Multi-tenant systems need tenant-aware observability:

- Metrics labeled by tenant for per-tenant dashboards.
- Log filtering by tenant for support investigations.
- Trace filtering by tenant for issue diagnosis.
- Aggregated and tenant-specific SLO monitoring.

Production Readiness Checklist A production-ready AI platform should have:

1. **Comprehensive metrics**: Request rates, latencies, error rates, resource usage, business metrics.
2. **Structured logging**: JSON logs with request correlation, appropriate log levels, PII protection.
3. **Distributed tracing**: OpenTelemetry instrumentation, sampling configuration, trace-log correlation.
4. **Health probes**: Kubernetes-compatible liveness, readiness, and startup probes.
5. **Dashboards**: Grafana dashboards for operational visibility.
6. **Alerting**: Alert rules for critical conditions with appropriate thresholds and routing.
7. **Runbooks**: Documentation for responding to common alerts and issues.

The Cineca Agentic Platform implements these observability capabilities, as detailed in Chapter 9, providing operators with comprehensive visibility into system behavior.

2.7.7 Chapter Summary

This chapter has established the theoretical and technical foundations for the Cineca Agentic Platform by reviewing the evolution of Large Language Models and AI agents, surveying existing orchestration frameworks and their limitations, introducing the Model Context Protocol as a standardization opportunity, examining graph databases and natural language interfaces, reviewing security and multi-tenancy patterns, and establishing observability concepts for distributed systems. The comparative analysis positions the platform's contributions relative to state-of-the-art alternatives, identifying the gaps that this thesis addresses. The following chapter translates these foundations into concrete requirements and design goals that guide the platform's architecture.

Chapter 3

Requirements and Design Goals

This chapter formalizes the requirements and design goals for the Cineca Agentic Platform, translating the stakeholder needs and use cases identified in Chapter 1 into specific, measurable requirements. The chapter covers functional requirements across agent orchestration, tool integration, security, and observability domains; non-functional requirements addressing performance, reliability, and maintainability; and design principles that guide architectural decisions. These requirements serve as the foundation for the architectural design presented in Chapter 4 and are systematically evaluated in Chapter 12.

3.1 Functional Requirements

This section formalizes the functional requirements of the Cineca Agentic Platform, derived from stakeholder analysis, use case modeling, and the technical specifications documented in the project repository. Each requirement is assigned a unique identifier for traceability to the evaluation chapter.

3.1.1 Stakeholder Analysis

The platform serves four primary stakeholder categories, each with distinct functional needs:

End Users (Researchers, Scientists): Individuals who interact with AI agents through conversational interfaces to execute computational workflows, query knowledge graphs, and obtain assistance with bioinformatics tasks. Their primary concern is ease of use and accurate, timely responses.

Platform Administrators: Technical staff responsible for system configuration, tenant management, model deployment, and operational oversight. They require comprehensive administrative capabilities and monitoring dashboards.

Security and Compliance Officers: Personnel ensuring the platform adheres to institutional policies, data protection regulations, and audit requirements. They need robust access controls, audit trails, and compliance reporting.

Developers and Integrators: Engineers who extend the platform with custom tools, integrate external systems, or build applications consuming the API. They require well-documented APIs, extensibility mechanisms, and clear contracts.

3.1.2 Use Case Categories

The functional requirements are organized around five core use case categories that emerged from stakeholder analysis and the platform's stated objectives:

1. **Agent Execution and Orchestration:** Conversational AI interactions, multi-step reasoning, and workflow automation.

2. **Knowledge Graph Query:** Natural language interfaces to graph databases for structured data retrieval.

3. **Background Job Processing:** Long-running computational tasks with progress tracking and lifecycle management.

4. **Tool Discovery and Invocation:** Dynamic tool ecosystem with capability-based access control.

5. **Administrative Operations:** Platform configuration, tenant management, and operational control.

3.1.3 Terminology

This subsection establishes canonical definitions for key terms used throughout this thesis. These definitions serve as the authoritative reference; other chapters reference these definitions using cross-references or brief reminders.

Agent Run: A single execution instance of an AI agent, initiated by a user prompt and consisting of one or more orchestration steps. Each run has a unique identifier, maintains its own execution context, and progresses through states: `pending`, `running`, `succeeded`, `failed`, or `cancelled`. See Section 5.3 for the complete lifecycle.

Session: A logical grouping of multiple agent runs that share conversational context. Sessions enable continuity across multiple user interactions, allowing agents to reference previous runs and maintain context. Sessions are identified by a unique session ID and may span multiple agent runs over time.

Job: An asynchronous task entity managed by the background job system. Jobs represent long-running operations that execute independently of the HTTP request-response cycle. Jobs have states: `queued`, `running`, `finished`, `failed`, or `cancelled`. Agent runs may execute synchronously (direct HTTP response) or asynchronously via jobs. See Chapter 7 for details.

Tool Invocation: A single execution of an MCP tool, triggered by an agent during an orchestration step. Each invocation includes a tool name, payload, execution context (tenant, principal, request ID), and produces a result or error. Tool invocations are logged for audit and observability.

Tenant Context: The isolation boundary that scopes all operations to a specific tenant. Every request, agent run, tool invocation, and data access is associated with a tenant ID, ensuring complete data isolation between tenants. Tenant context is propagated through headers (`X-Tenant-Id`) and enforced at all layers.

Safe Mode: A restricted execution mode where agents can only invoke tools marked as "safe" (e.g., read-only graph queries, system health checks). Safe mode is enforced for users with limited permissions (`tools:basic` scope) to prevent potentially destructive operations.

DMR (Default Model Resolver): A caching and routing mechanism that determines which LLM provider and model instance should handle a given request. The DMR considers provider health, load balancing, tenant preferences, and fallback strategies. See Chapter 5 for architectural details.

These definitions are referenced throughout the thesis. For architectural details, see Chapter 4; for implementation specifics, see the relevant chapters.

3.1.4 Agent Execution Requirements

The following requirements govern the core agent orchestration functionality:

3.1.5 Model Management Requirements

Requirements governing LLM provider and model instance management:

3.1.6 Tool Ecosystem Requirements

Requirements for the Model Context Protocol (MCP) tool system:

3.1.7 Graph Query Requirements

Requirements for the natural language to Cypher (NL$\rightarrow$Cypher) pipeline:

3.1.8 Background Job Requirements

Requirements for asynchronous task processing:

3.1.9 Administrative and Operational Requirements

Requirements for platform administration and operations:

3.1.10 API Design Requirements

Cross-cutting requirements for the REST API:

Table 3.1. Agent Execution Functional Requirements

ID	Requirement Description	Priority
FR-AE-01	The system shall accept natural language prompts and execute multi-step agent runs to fulfill user requests.	Must
FR-AE-02	The system shall support configurable execution parameters including maximum steps, temperature, and timeout limits.	Must
FR-AE-03	The system shall maintain session context across multiple agent runs, enabling conversational continuity.	Must
FR-AE-04	The system shall record all execution steps with timestamps, tool calls, and intermediate outputs for auditability.	Must
FR-AE-05	The system shall support intent classification to route requests appropriately (deterministic vs. agentic vs. graph-based).	Should
FR-AE-06	The system shall generate structured task lists for complex multi-step tasks and track their completion status.	Should
FR-AE-07	The system shall provide execution metrics including latency, token consumption, and tool call counts per run.	Must
FR-AE-08	The system shall support graceful degradation when LLM providers are unavailable, with fallback to deterministic responses.	Should
FR-AE-09	The system shall allow cancellation of in-progress agent runs with proper cleanup of resources.	Must
FR-AE-10	The system shall expose run status through polling endpoints with terminal states (succeeded, failed, cancelled).	Must

Table 3.2. Model Management Functional Requirements

ID	Requirement Description	Priority
FR-MM-01	The system shall support multiple LLM providers (OpenAI, Ollama, Azure OpenAI, Hugging Face) through a unified adapter interface.	Must
FR-MM-02	The system shall maintain a registry of model instances with configurable parameters (context window, temperature defaults, cost per token).	Must
FR-MM-03	The system shall implement a Default Model Resolver (DMR) with hierarchical precedence: user preference → tenant default → global default.	Must
FR-MM-04	The system shall track model usage metrics including token consumption, request counts, and estimated costs per tenant.	Should
FR-MM-05	The system shall support hot-swapping of default models without service restart.	Should
FR-MM-06	The system shall validate model availability before accepting requests, returning appropriate errors for unavailable models.	Must
FR-MM-07	The system shall support model warmup to reduce cold-start latency for frequently used models.	Could
FR-MM-08	The system shall provide model capability metadata (supports tools, supports vision, context length) for client-side filtering.	Should

Complete API Reference For a comprehensive reference of all 76 API endpoints, including request/response schemas, authentication requirements, and error codes, see Appendix A.

3.1.11 User Interface Requirements

Requirements for the presentation layer:

3.1.12 Requirements Summary

Table 3.9 summarizes the functional requirements by category:

The prioritization follows the MoSCoW methodology: **Must** requirements are essential for minimum viable functionality, **Should** requirements enhance usability and operational excellence, and **Could** requirements provide additional value if resources permit.

3.2 Non-Functional Requirements

This section specifies the non-functional requirements (NFRs) that govern the quality attributes of the Cineca Agentic Platform. These requirements address cross-cutting concerns including security, reliability, performance, and operational characteristics that are essential for enterprise deployment.

Table 3.3. Tool Ecosystem Functional Requirements

ID	Requirement Description	Priority
FR-TL-01	The system shall provide a tool discovery endpoint returning available tools with their schemas, capabilities, and access requirements.	Must
FR-TL-02	The system shall invoke tools by dotted name (e.g., `graph.query`) with schema-validated payloads.	Must
FR-TL-03	The system shall enforce capability-based access control, restricting tools based on principal scopes.	Must
FR-TL-04	The system shall support at least 34 tools across 17 categories including graph, cache, security, data, and administrative operations.	Must
FR-TL-05	The system shall record all tool invocations with inputs, outputs, latency, and error status for audit purposes.	Must
FR-TL-06	The system shall enforce per-tool timeouts with configurable limits (default: 30 seconds).	Must
FR-TL-07	The system shall support tool-level rate limiting independent of API-level rate limits.	Should
FR-TL-08	The system shall provide safe mode for tools, preventing write operations during read-only requests.	Must
FR-TL-09	The system shall expose tool metrics (invocation count, latency histogram, error rate) via Prometheus.	Should

Table 3.4. Graph Query Functional Requirements

ID	**Requirement Description**	**Priority**
FR-GQ-01	The system shall accept natural language questions and translate them to valid Cypher queries against the Memgraph database.	Must
FR-GQ-02	The system shall provide schema introspection exposing node types, relationship types, and property definitions.	Must
FR-GQ-03	The system shall validate generated Cypher queries for safety, detecting and rejecting write operations in read-only contexts.	Must
FR-GQ-04	The system shall support parameterized queries to prevent Cypher injection attacks.	Must
FR-GQ-05	The system shall maintain an allowlist of permitted CALL procedures for security purposes.	Must
FR-GQ-06	The system shall automatically inject tenant isolation predicates (`WHERE tenant_id = $tid`) into generated queries.	Must
FR-GQ-07	The system shall provide query explain functionality for debugging and optimization.	Should
FR-GQ-08	The system shall cache frequently executed queries with configurable TTL.	Could
FR-GQ-09	The system shall support bulk graph operations (batch create, batch update) for data ingestion workflows.	Should

Table 3.5. Background Job Functional Requirements

ID	**Requirement Description**	**Priority**
FR-JB-01	The system shall support creation of asynchronous jobs with typed payloads and priority levels.	Must
FR-JB-02	The system shall persist jobs in PostgreSQL with full lifecycle tracking (queued $\rightarrow$ running $\rightarrow$ finished/failed/cancelled).	Must
FR-JB-03	The system shall provide idempotency key support to prevent duplicate job creation on retries.	Must
FR-JB-04	The system shall stream job progress events via Server-Sent Events (SSE) with Last-Event-ID support for reconnection.	Must
FR-JB-05	The system shall support job cancellation with graceful handling of in-progress work.	Must
FR-JB-06	The system shall maintain heartbeat updates for running jobs to detect stale workers.	Should
FR-JB-07	The system shall provide job result retrieval with configurable retention periods.	Must
FR-JB-08	The system shall expose queue depth and backlog metrics for capacity planning.	Should
FR-JB-09	The system shall support configurable job types with per-type worker allocation.	Should

Table 3.6. Administrative Functional Requirements

ID	Requirement Description	Priority
FR-AD-01	The system shall provide tenant CRUD operations with configuration management (quotas, defaults, feature flags).	Must
FR-AD-02	The system shall support database administration endpoints for schema inspection and connection testing.	Should
FR-AD-03	The system shall provide configuration export/import for backup and environment migration.	Should
FR-AD-04	The system shall expose comprehensive health probes (liveness, readiness, startup) for container orchestration.	Must
FR-AD-05	The system shall provide component-level health checks for all external dependencies (PostgreSQL, Redis, Memgraph, LLM providers).	Must
FR-AD-06	The system shall support graceful shutdown with in-flight request completion.	Must
FR-AD-07	The system shall provide rate limit configuration and override capabilities per tenant or user.	Should
FR-AD-08	The system shall expose OpenAPI specifications for API documentation and client generation.	Must
FR-AD-09	The system shall support batch operations for efficient bulk data manipulation.	Should
FR-AD-10	The system shall provide process management for long-running administrative operations.	Could

Table 3.7. API Design Functional Requirements

ID	Requirement Description	Priority
FR-AP-01	The system shall expose a RESTful API with versioned endpoints (`/v1/*`, `/v2/*`).	Must
FR-AP-02	The system shall validate all request bodies against Pydantic schemas with detailed error messages.	Must
FR-AP-03	The system shall implement cursor-based pagination for collection endpoints with configurable page sizes.	Must
FR-AP-04	The system shall support conditional requests via ETags for cache validation.	Should
FR-AP-05	The system shall return RFC 7807 Problem Details for all error responses.	Must
FR-AP-06	The system shall include request correlation IDs in all responses via `X-Request-ID` header.	Must
FR-AP-07	The system shall provide deprecation headers for sunset endpoints with migration guidance.	Should
FR-AP-08	The system shall support at least 70 endpoints across 16 functional categories (76 implemented at snapshot commit).	Must

Table 3.8. User Interface Functional Requirements

ID	Requirement Description	Priority
FR-UI-01	The system shall provide a conversational chat interface for end-user agent interactions.	Must
FR-UI-02	The system shall provide an administrative control panel for operators and administrators.	Must
FR-UI-03	The chat interface shall display agent execution steps with expandable details.	Should
FR-UI-04	The control panel shall provide health monitoring dashboards with auto-refresh.	Should
FR-UI-05	Both interfaces shall support role-based access with appropriate feature visibility.	Must
FR-UI-06	The chat interface shall support dynamic model selection from available instances.	Should
FR-UI-07	The control panel shall provide NL→Cypher testing capabilities with result visualization.	Should
FR-UI-08	Both interfaces shall implement responsive layouts for desktop and tablet form factors.	Should

Table 3.9. Functional Requirements Summary by Category

Category	Must	Should	Could	Total
Agent Execution	7	3	0	10
Model Management	4	3	1	8
Tool Ecosystem	7	2	0	9
Graph Query	6	2	1	9
Background Jobs	6	3	0	9
Administrative	5	4	1	10
API Design	6	2	0	8
User Interface	3	5	0	8
Total	**44**	**24**	**3**	**71**

3.2.1 Security Requirements

Security is paramount for an enterprise AI agent platform that processes sensitive research data and executes privileged operations. The following requirements establish the security posture:

Table 3.10. Security Non-Functional Requirements

ID	**Requirement Description**	**Priority**
NFR-SE-01	The system shall authenticate all API requests using OAuth 2.0 / OpenID Connect (OIDC) with JWT bearer tokens.	Must
NFR-SE-02	The system shall validate JWT tokens against OIDC provider public keys (JWKS) with caching and automatic refresh.	Must
NFR-SE-03	The system shall enforce Role-Based Access Control (RBAC) using scope-based permissions embedded in JWT claims.	Must
NFR-SE-04	The system shall support configurable RBAC scopes including: `user:me`, `admin:all`, `tools:invoke:basic`, `tools:invoke:all`, `graph:read`, `graph:write`.	Must
NFR-SE-05	The system shall automatically detect and scrub Personally Identifiable Information (PII) from logs and outputs.	Must
NFR-SE-06	The system shall prevent Server-Side Request Forgery (SSRF) attacks by validating and restricting outbound request targets.	Must
NFR-SE-07	The system shall prevent Cypher injection attacks through parameterized queries and input validation.	Must
NFR-SE-08	The system shall mask sensitive data (tokens, credentials) in all log outputs, displaying only prefix and suffix characters.	Must
NFR-SE-09	The system shall enforce HTTPS for all external communications in production environments.	Must
NFR-SE-10	The system shall implement defense-in-depth with multiple security layers (authentication, authorization, input validation, output filtering).	Must

Authentication Architecture

The authentication subsystem implements the following security controls:

- **Token Validation:** Every request to protected endpoints must include a valid JWT in the `Authorization: Bearer <token>` header. Tokens are validated against the OIDC provider's JSON Web Key Set (JWKS).
- **Claim Extraction:** Standard OIDC claims (`sub`, `iss`, `aud`, `exp`, `iat`) are extracted and validated. Custom claims (scopes, tenant_id) are used for authorization decisions.
- **Key Rotation:** The system caches JWKS keys with automatic refresh when key IDs are not found, supporting transparent key rotation by the identity provider.

- **Clock Skew Tolerance:** A configurable clock skew tolerance (default: 30 seconds) accommodates minor time synchronization differences between servers.

Authorization Model

The platform implements a scope-based RBAC model with the following hierarchy:

Table 3.11. RBAC Scope Hierarchy

Scope	Level	Permissions Granted
`user:me`	User	Read/write own resources (runs, sessions, jobs)
`tools:invoke:basic`	User	Invoke read-only, non-administrative tools
`tools:invoke:all`	Power User	Invoke all tools including write operations
`graph:read`	User	Execute read-only Cypher queries
`graph:write`	Power User	Execute write Cypher queries (CREATE, MERGE, DELETE)
`admin:all`	Admin	Full administrative access (tenant management, model configuration, system operations)

3.2.2 Multi-Tenancy Requirements

The platform must support multiple organizational tenants with strict data isolation:

Table 3.12. Multi-Tenancy Non-Functional Requirements

ID	Requirement Description	Priority
NFR-MT-01	The system shall support logical tenant isolation within shared infrastructure (shared-database, shared-schema pattern).	Must
NFR-MT-02	The system shall resolve tenant context from JWT claims or `X-Tenant-ID` header on every request.	Must
NFR-MT-03	All database queries shall be automatically scoped by `tenant_id` to prevent cross-tenant data access.	Must
NFR-MT-04	The system shall support per-tenant configuration including default models, rate limits, and feature flags.	Must
NFR-MT-05	The system shall enforce per-tenant quotas for API requests, token consumption, and storage usage.	Should
NFR-MT-06	The system shall provide tenant-scoped metrics and audit logs for compliance reporting.	Must
NFR-MT-07	The system shall prevent tenant ID spoofing through cryptographic validation of tenant claims.	Must
NFR-MT-08	The system shall support tenant onboarding and offboarding through administrative APIs.	Must

Tenant Isolation Architecture

The multi-tenancy architecture implements the following isolation mechanisms:

1. **Request-Level Isolation:** Tenant context is established at the middleware layer and propagated through the entire request lifecycle via context variables.
2. **Query-Level Isolation:** All repository methods automatically append `WHERE tenant_id = :tid` predicates. For Cypher queries, tenant isolation is injected during query rewriting.
3. **Resource-Level Isolation:** All domain entities (Agent Runs, Sessions, Jobs, Tool Invocations) include a `tenant_id` foreign key with database-level constraints.
4. **Cache Isolation:** Redis keys are namespaced by tenant ID to prevent cache poisoning across tenants.

3.2.3 Reliability Requirements

Requirements ensuring system availability and fault tolerance:

Table 3.13. Reliability Non-Functional Requirements

ID	Requirement Description	Priority
NFR-RE-01	The system shall implement circuit breakers for all external service calls (LLM providers, databases) with configurable thresholds.	Must
NFR-RE-02	The system shall support graceful degradation when LLM providers are unavailable, falling back to deterministic responses or cached results.	Must
NFR-RE-03	The system shall handle Redis unavailability without complete service failure, degrading non-critical features (caching, rate limiting).	Should
NFR-RE-04	The system shall implement retry logic with exponential backoff for transient failures.	Must
NFR-RE-05	The system shall support graceful shutdown, completing in-flight requests before termination.	Must
NFR-RE-06	The system shall maintain heartbeat signals for long-running operations to detect stale workers.	Should
NFR-RE-07	The system shall achieve 99.5% availability during normal operations (excluding planned maintenance).	Should
NFR-RE-08	The system shall recover from transient database connection failures within 30 seconds.	Must

Circuit Breaker Implementation

The platform implements a three-state circuit breaker pattern for LLM provider resilience:

CLOSED: Normal operation; requests flow to the provider. Failures are counted.

OPEN: Provider is considered unavailable; requests fail fast without attempting the call. Automatic transition to HALF-OPEN after a configurable timeout.

HALF-OPEN: A limited number of probe requests are allowed. Success transitions to CLOSED; failure returns to OPEN.

Circuit breaker state is persisted in Redis for cluster-wide consistency, with the following configurable parameters:

- `failure_threshold`: Number of failures before opening (default: 5)
- `success_threshold`: Successes in half-open before closing (default: 3)
- `timeout_seconds`: Duration of open state before half-open (default: 30 seconds)

3.2.4 Performance Requirements

Latency and throughput requirements for the platform:

Table 3.14. Performance Non-Functional Requirements

Metric	Target (p95)	Target (p99)	Measurement Method
Health probe latency	< 5ms	< 10ms	Prometheus histogram
API latency (non-LLM)	< 300ms	< 500ms	Load test with Locust
Tool invocation	< 500ms	< 2s	Per-tool latency histogram
Database query	< 50ms	< 100ms	Query timing metrics
Cache hit latency	< 2ms	< 5ms	Redis operation timing
Job queue latency	< 1s	< 2s	Queue-to-start timing

3.2.5 Auditability and Compliance Requirements

Requirements for audit logging and regulatory compliance:

3.2.6 Observability Requirements

Requirements for monitoring, logging, and tracing:

Table 3.15. Auditability Non-Functional Requirements

ID	Requirement Description	Priority
NFR-AU-01	The system shall maintain append-only audit logs for all security-relevant events (authentication, authorization, data access).	Must
NFR-AU-02	Audit logs shall include: timestamp, principal, action, resource, tenant, outcome, and correlation ID.	Must
NFR-AU-03	Audit logs shall be retained for a configurable period (default: 90 days) with archival support.	Must
NFR-AU-04	The system shall support audit log export for compliance reporting and external SIEM integration.	Should
NFR-AU-05	All agent runs shall record complete execution traces including prompts, responses, and tool calls.	Must
NFR-AU-06	The system shall track model usage with token counts and estimated costs per tenant.	Should
NFR-AU-07	Administrative actions (tenant creation, model configuration, permission changes) shall generate audit events.	Must
NFR-AU-08	The system shall support data residency requirements through configurable deployment regions.	Could

Table 3.16. Observability Non-Functional Requirements

ID	Requirement Description	Priority
NFR-OB-01	The system shall expose Prometheus metrics via `/metrics` endpoint covering HTTP requests, background jobs, tools, and LLM operations.	Must
NFR-OB-02	The system shall implement structured logging (JSON format) with consistent field schemas.	Must
NFR-OB-03	The system shall propagate distributed trace context (W3C Trace Context) across all service calls.	Should
NFR-OB-04	The system shall correlate logs, metrics, and traces using a common request ID.	Must
NFR-OB-05	Health probes shall check all external dependencies with individual status reporting.	Must
NFR-OB-06	The system shall provide pre-configured Grafana dashboards for operational monitoring.	Should
NFR-OB-07	Log output shall be configurable (JSON for production, human-readable for development).	Should
NFR-OB-08	Metrics shall include latency histograms with configurable bucket boundaries.	Should

Three Pillars of Observability

The platform implements comprehensive observability through:

1. **Metrics (Prometheus):** Counter, histogram, and gauge metrics for all major subsystems:
   - HTTP request rate, latency, and error rate by endpoint
   - Background job execution counts and durations
   - Tool invocation statistics by tool name and outcome
   - LLM call latency, token consumption, and cost estimates
   - Rate limit hit rates and quota utilization
2. **Logging (Structlog):** Structured JSON logs with:
   - Consistent field naming across all components
   - Request ID correlation for distributed tracing
   - Log level filtering with environment-specific defaults
   - PII scrubbing before log emission
3. **Tracing (OpenTelemetry):** Distributed tracing with:
   - Automatic span creation for HTTP requests and database calls
   - Manual instrumentation for LLM calls and tool invocations
   - Configurable sampling rates (20% in production, 100% in development)
   - OTLP export to compatible collectors

3.2.7 Scalability Requirements

Requirements for horizontal and vertical scaling:

3.2.8 Rate Limiting Requirements

Requirements for request throttling and quota management:

3.2.9 Requirements Summary

Table 3.19 summarizes the non-functional requirements by category:

The predominance of **Must** requirements in the Security category reflects the enterprise context and the sensitive nature of AI agent operations in research environments.

3.3 Constraints and Assumptions

This section documents the constraints that bound the design space and the assumptions upon which the architecture depends. Understanding these factors is essential for evaluating design decisions and identifying potential risks.

Table 3.17. Scalability Non-Functional Requirements

ID	Requirement Description	Priority
NFR-SC-01	The system shall support horizontal scaling of API servers behind a load balancer.	Must
NFR-SC-02	The system shall support horizontal scaling of background workers with job queue partitioning.	Should
NFR-SC-03	The system shall maintain stateless API servers, storing all session state in external stores (Redis, PostgreSQL).	Must
NFR-SC-04	The system shall support connection pooling for database connections with configurable pool sizes.	Must
NFR-SC-05	The system shall implement request-level rate limiting to prevent resource exhaustion during traffic spikes.	Must
NFR-SC-06	The system architecture shall not preclude deployment on Kubernetes with pod autoscaling.	Should

Table 3.18. Rate Limiting Non-Functional Requirements

ID	Requirement Description	Priority
NFR-RL-01	The system shall implement sliding window rate limiting with Redis-backed counters.	Must
NFR-RL-02	Rate limits shall be configurable per endpoint, per user, and per tenant.	Must
NFR-RL-03	Rate limit responses shall include `Retry-After` headers with backoff guidance.	Must
NFR-RL-04	The system shall support rate limit overrides for privileged users or emergency access.	Should
NFR-RL-05	Rate limit metrics shall be exposed for monitoring and alerting.	Should
NFR-RL-06	Default rate limits shall be: 60 requests/minute for general API, 10 requests/minute for LLM operations.	Should

Table 3.19. Non-Functional Requirements Summary by Category

Category	Must	Should	Could	Total
Security	10	0	0	10
Multi-Tenancy	6	2	0	8
Reliability	5	3	0	8
Performance	3	5	0	8
Auditability	5	2	1	8
Observability	4	4	0	8
Scalability	4	2	0	6
Rate Limiting	3	3	0	6
Total	**41**	**20**	**1**	**62**

3.3.1 Institutional Constraints

The Cineca Agentic Platform is developed within the context of CINECA, Italy's national supercomputing center, which imposes specific institutional constraints:

Research Environment: The platform serves research institutions, universities, and scientific consortia. This mandates strong emphasis on data provenance, reproducibility, and compliance with research ethics requirements.

Multi-Institutional Access: CINECA provides computing services to numerous Italian and European research institutions. The platform must accommodate diverse organizational structures, authentication federations, and access policies.

Long-Running Workloads: HPC environments commonly execute jobs spanning hours to days. The platform must support asynchronous operations with robust state persistence and recovery mechanisms.

Sensitive Data Handling: Biomedical research data, genomic sequences, and patient-derived information require stringent access controls, audit trails, and compliance with data protection regulations.

Open Science Alignment: As a public research infrastructure, the platform aligns with open science principles, favoring open-source components and avoiding vendor lock-in where possible.

3.3.2 Technical Infrastructure Constraints

The following technical constraints shape infrastructure decisions:

3.3.3 Technology Stack Constraints

The following technology choices are constraints inherited from project requirements or ecosystem considerations:

Dependency Version Constraints

Key Python dependencies with version constraints (from `pyproject.toml`):

- `fastapi >= 0.109.0`: Async web framework with OpenAPI support
- `pydantic >= 2.0`: Data validation with Python type hints
- `sqlalchemy >= 2.0`: ORM with async support
- `redis >= 5.0`: Async Redis client
- `httpx >= 0.25.0`: Async HTTP client for external API calls
- `structlog >= 24.1.0`: Structured logging
- `prometheus-client >= 0.19.0`: Metrics exposition
- `opentelemetry-api >= 1.20.0`: Distributed tracing

Table 3.20. Infrastructure Constraints

ID	Constraint Description	Impact
IC-01	Container orchestration via Docker Compose or Kubernetes is mandatory for deployment.	Architecture must be container-native with 12-factor app principles.
IC-02	On-premises deployment must be supported alongside cloud options (no cloud-only dependencies).	All components must run on standard Linux servers without mandatory cloud services.
IC-03	Network policies may restrict outbound internet access from production clusters.	LLM provider integration must support both cloud APIs and local model serving (Ollama).
IC-04	GPU resources are shared across HPC workloads; dedicated GPU allocation for AI inference is limited.	Model serving must support CPU-only execution with optional GPU acceleration.
IC-05	Storage quotas apply; persistent data must be efficiently managed with retention policies.	Automatic cleanup of expired jobs, logs, and cached data is required.
IC-06	Authentication must integrate with institutional identity providers (SAML, LDAP, OIDC federations).	OAuth 2.0/OIDC with configurable providers; no hard-coded Auth0 dependency.

Table 3.21. Technology Stack Constraints

Component	Technology	Rationale
Runtime	Python 3.11+	Ecosystem compatibility (ML/AI libraries), team expertise, async support.
Web Framework	FastAPI	Modern async framework, automatic OpenAPI generation, Pydantic integration.
Control Database	PostgreSQL 15+	ACID compliance, JSONB support, mature ecosystem, institutional familiarity.
Cache/Queue	Redis 7+	Low-latency caching, pub/sub for events, rate limiting primitives.
Graph Database	Memgraph	Cypher compatibility, in-memory performance, streaming graph analytics.
Container Runtime	Docker 24+	Industry standard, Compose for development, Kubernetes for production.
Reverse Proxy	Nginx	SSL termination, rate limiting, static file serving, proven reliability.
Monitoring	Prometheus + Grafana	Open-source observability stack, extensive community dashboards.

3.3.4 Regulatory and Compliance Constraints

The platform operates within a regulated environment with the following compliance considerations:

GDPR (General Data Protection Regulation): As a European institution, CINECA must ensure GDPR compliance for any personal data processed. This mandates data minimization, purpose limitation, consent management, and data subject rights support.

Research Data Management: Scientific data must maintain provenance records, versioning, and retention according to funder requirements (e.g., Horizon Europe data management plans).

Institutional Security Policies: CINECA maintains ISO 27001-aligned security policies that govern access control, incident response, and audit requirements.

Export Control: Certain research domains (dual-use technologies, cryptography) may be subject to export controls affecting data sharing and cross-border processing.

Compliance Implementation Mapping

Table 3.22. Compliance Requirement to Feature Mapping

Regulation	**Requirement**	**Platform Feature**
GDPR Art. 17	Right to erasure	Tenant data deletion API, cascade delete for user data
GDPR Art. 30	Records of processing	Audit log with processing activity records
GDPR Art. 32	Security of processing	Encryption in transit (TLS), access controls, PII scrubbing
GDPR Art. 33	Breach notification	Audit log monitoring, anomaly detection hooks
ISO 27001 A.12.4	Logging and monitoring	Structured logging, metrics, distributed tracing
ISO 27001 A.9.4	Access control	OIDC authentication, RBAC authorization

3.3.5 Development Process Constraints

Constraints arising from the development methodology and team structure:

Single-Developer Implementation: The core platform was developed as a master's thesis project by a single developer. This constrains the scope and complexity of initial implementation while emphasizing clean architecture and comprehensive documentation for future maintainers.

Academic Timeline: Development adheres to academic semester constraints, requiring prioritization of core features over nice-to-have enhancements.

Open-Source Orientation: The platform is intended for open-source release, requiring:

- Clear licensing (MIT License for code)
- No proprietary dependencies for core functionality
- Comprehensive documentation for community adoption
- Secrets and credentials externalized from codebase

Continuous Integration: Development follows CI/CD practices with automated testing, though full production CI pipeline setup is deferred to deployment phase.

3.3.6 Assumptions

The architecture is predicated on the following assumptions. Violation of these assumptions may require design revisions:

3.3.7 Scope Boundaries

Clear scope boundaries define what the platform does and does not address:

In Scope

- Agent orchestration for conversational AI and multi-step reasoning
- Natural language to Cypher query translation for graph databases
- Background job processing with lifecycle management
- Multi-tenant architecture with tenant isolation
- OAuth 2.0/OIDC authentication with scope-based authorization
- MCP-style tool ecosystem with capability-based access
- Prometheus/Grafana observability stack
- RESTful API with OpenAPI documentation
- Chat UI (Next.js) and Control Panel (Streamlit)

Out of Scope (Non-Goals)

- **Document RAG Pipeline:** Document ingestion, chunking, embedding, and vector search are not implemented. The platform focuses on graph-based knowledge retrieval via NL→Cypher.
- **Model Training:** The platform consumes pre-trained models but does not support fine-tuning or training workflows.

Table 3.23. Architectural Assumptions

ID	Assumption	Risk if Violated
A-01	LLM providers (OpenAI, Ollama) maintain stable API contracts.	Adapter layer may require updates; circuit breakers mitigate transient issues.
A-02	Authentication is delegated to a trusted OIDC provider (Auth0, Keycloak, etc.).	If provider is unavailable, authentication fails; local token caching provides limited mitigation.
A-03	Network latency to LLM providers is acceptable ($<$ 500ms RTT).	High latency degrades user experience; local Ollama deployment provides alternative.
A-04	PostgreSQL and Redis are deployed with high availability (replication, failover).	Single points of failure if not properly configured; application assumes availability.
A-05	Memgraph graph database can fit the working dataset in memory.	Memory pressure may cause performance degradation or OOM; monitoring required.
A-06	Container orchestration (Docker/Kubernetes) handles process supervision and restart.	Application does not implement its own process manager; relies on orchestrator.
A-07	Clock synchronization (NTP) is maintained across all nodes.	JWT expiration validation may fail with significant clock skew ($>$ 30 seconds).
A-08	DNS resolution is reliable and cached appropriately.	Service discovery failures may cascade; health checks detect connectivity issues.
A-09	Users have modern web browsers (Chrome, Firefox, Safari, Edge) for UI access.	Legacy browser support is not tested; graceful degradation not guaranteed.
A-10	Concurrent user load does not exceed 1,000 simultaneous sessions initially.	Scaling strategy requires validation for higher loads.

- **Multi-Agent Collaboration:** Complex multi-agent architectures (agent-to-agent communication, hierarchical agent teams) are not implemented in the initial version.
- **Real-Time Streaming:** Server-Sent Events (SSE) for job progress exist, but WebSocket-based real-time agent step streaming is not implemented (polling is used).
- **Mobile Applications:** Native mobile apps are not provided; the web UI is responsive but not optimized for mobile.
- **Offline Operation:** The platform requires network connectivity; offline/disconnected operation is not supported.
- **Custom UI Framework:** The UIs (Next.js, Streamlit) are provided as reference implementations; custom UI development is the integrator's responsibility.

3.3.8 Constraint and Assumption Validation

The following mechanisms validate that constraints and assumptions hold during operation:

1. **Health Probes:** Liveness, readiness, and startup probes verify infrastructure assumptions (database connectivity, Redis availability, LLM provider health).
2. **Configuration Validation:** Application startup validates required environment variables and configuration consistency.
3. **Dependency Version Checks:** Runtime assertions verify minimum library versions where API compatibility is critical.
4. **Integration Tests:** CI pipeline includes integration tests against containerized dependencies to validate technology stack compatibility.
5. **Monitoring and Alerting:** Prometheus metrics and Grafana alerts detect constraint violations (e.g., clock skew detection, memory pressure).

3.4 Design Principles

This section articulates the design principles that guided architectural decisions for the Cineca Agentic Platform. These principles represent deliberate trade-offs optimized for enterprise deployment, operational excellence, and long-term maintainability.

3.4.1 Separation of Concerns: Three-Layer Architecture

The platform adopts a three-layer architecture that enforces clear boundaries between concerns:

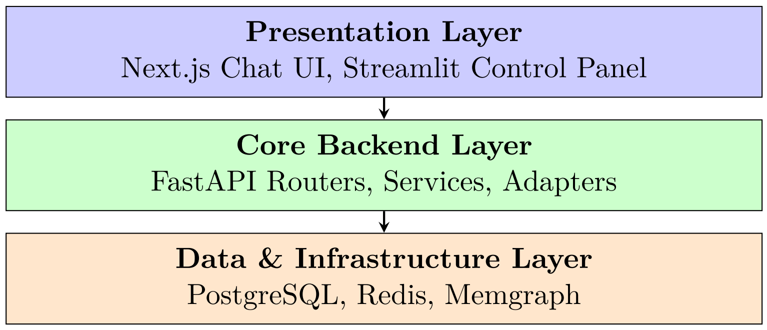
Presentation Layer
Next.js Chat UI, Streamlit Control Panel
Core Backend Layer
FastAPI Routers, Services, Adapters
Data & Infrastructure Layer
PostgreSQL, Redis, Memgraph

Figure 3.1. Three-Layer Architecture

Layer Responsibilities

Presentation Layer: Responsible for user interaction, request formatting, and response rendering. The layer is stateless and communicates exclusively through the backend API. Technology choices (Next.js, Streamlit) can evolve independently of backend logic.

Core Backend Layer: Contains business logic, orchestration, and API contracts. This layer is further subdivided into:

- **Router Layer:** HTTP endpoint handlers, request validation, response serialization
- **Service Layer:** Business logic, orchestration, transaction management
- **Adapter Layer:** External system integration (LLM providers, databases)

Data & Infrastructure Layer: Provides persistent storage, caching, and external service integrations. This layer abstracts infrastructure details behind repository and adapter interfaces.

Layer Coupling Rules

- Upper layers may depend on lower layers, but not vice versa
- The core backend layer defines interfaces that adapters implement
- Cross-layer communication occurs through well-defined contracts (Pydantic schemas, repository interfaces)
- Infrastructure concerns (connection pooling, retry logic) are encapsulated within the adapter layer

3.4.2 Defense in Depth: Security Layering

Security is implemented through multiple overlapping defensive layers, ensuring that compromise of any single layer does not result in complete system breach:

Table 3.24. Defense-in-Depth Security Layers

Layer	Name	Controls
1	Network	TLS encryption, firewall rules, network segmentation
2	Authentication	OIDC/JWT validation, token expiration, JWKS verification
3	Authorization	Scope-based RBAC, endpoint-level permission checks
4	Tenant Isolation	Request-scoped tenant context, query-level filtering
5	Input Validation	Pydantic schema validation, Cypher injection prevention
6	Rate Limiting	Per-user/tenant quotas, sliding window throttling
7	Output Filtering	PII scrubbing, response sanitization
8	Audit	Append-only audit logs, tamper detection

Security Principle Applications

Principle of Least Privilege: Tokens carry minimal scopes required for the task. Administrative operations require explicit `admin:all` scope. Tool invocations check capability requirements before execution.

Fail Secure: When authentication or authorization cannot be verified (e.g., JWKS unavailable), requests are denied rather than allowed.

Separation of Duties: Administrative functions (tenant management, model configuration) are separated from operational functions (agent execution, job processing).

Defense Diversity: Multiple security mechanisms (authentication, authorization, validation) use different implementation approaches to prevent single-point vulnerabilities.

3.4.3 Fail-Safe Defaults

The platform implements fail-safe defaults that favor security and stability over permissiveness:

- **Authentication Required:** All non-public endpoints in the reference deployment require valid JWT tokens. Unauthenticated requests receive 401 responses.
- **Restrictive RBAC:** New users receive minimal scopes (`user:me`). Elevated permissions must be explicitly granted.

- **Read-Only by Default:** Graph queries default to read-only mode. Write operations require explicit flags and `graph:write` scope.
- **Conservative Timeouts:** Default timeouts are set to prevent resource exhaustion (30 seconds for tools, device-dependent for agent runs: 180-3600 seconds based on compute device as shown in Table 5.1).
- **Rate Limiting Enabled:** Rate limits apply by default; exemptions require explicit configuration.
- **Audit Logging Enabled:** Security-relevant events are logged by default; suppression requires explicit opt-out.
- **Circuit Breakers Closed:** External provider circuits start closed; failures trigger protection automatically.

3.4.4 Extensibility and Modularity

The architecture prioritizes extensibility to accommodate future requirements without architectural rewrites:

Extension Points

Table 3.25. Platform Extension Points

Extension Type	**Mechanism**	**Effort Level**
New MCP Tool	Module creation at `src/mcp/tools/<category>/<name>.py`	Low (1 file, auto-discovered)
New LLM Provider	Adapter implementation in `src/adapters/llm.py`	Medium (1-2 files)
New API Endpoint	Router module in `src/routers/`, mount in `app.py`	Low (2-3 files)
New Background Task	Job type in `src/jobs/`, worker handler	Medium (2-3 files)
New Health Probe	Component in `src/health/components.py`	Low (1 function)
New Graph Domain	Schema in `db/memgraph\_domain/`, ETL loaders	Medium (3-5 files)
New Metric	Registration in `src/observability/metrics.py`	Low (1 declaration)

Modularity Principles

Single Responsibility: Each module addresses one cohesive concern. Routers handle HTTP, services handle logic, adapters handle integration.

Interface Segregation: Narrow interfaces prevent unnecessary coupling. The `JobStore` interface exposes only CRUD operations, not implementation details.

Dependency Inversion: High-level modules (services) depend on abstractions (interfaces), not concrete implementations (Redis, PostgreSQL).

Convention over Configuration: Tools are discovered by naming convention (`<category>.<name>` maps to `src/mcp/tools/<category>/<name>.py`), reducing boilerplate.

3.4.5 Observability by Design

Observability is not an afterthought but a core architectural concern integrated throughout the system:

Instrumentation Principles

- **Request Correlation:** Every request receives a unique `request_id` that propagates through logs, metrics, and traces for end-to-end visibility.
- **Structured Logging:** All log statements use structured format (JSON in production) with consistent field naming for log aggregation and querying.
- **Metric Granularity:** Metrics are labeled by relevant dimensions (endpoint, status, tenant) to support flexible aggregation and alerting.
- **Trace Sampling:** Distributed traces use configurable sampling (100% in development, 20% in production) to balance observability with overhead.
- **Health Semantics:** Health probes distinguish between liveness (process alive), readiness (accepting traffic), and component health (dependency status).

Observability Integration Points

3.4.6 Enterprise-First Design Trade-offs

The platform makes deliberate trade-offs favoring enterprise requirements over prototyping simplicity:

Trade-off Implications

These enterprise-first choices have implications:

Increased Complexity: The codebase is larger and requires more infrastructure (Redis, PostgreSQL, OIDC provider) compared to minimal prototypes.

Steeper Learning Curve: New developers must understand authentication flows, tenant context, and observability patterns.

Higher Operational Overhead: Production deployment requires monitoring, alerting, and infrastructure management expertise.

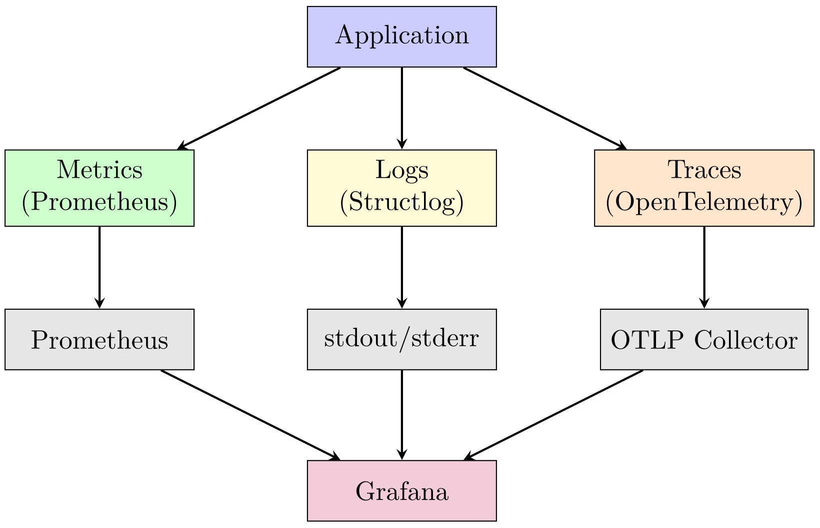

Figure 3.2. Observability Data Flow

Table 3.26. Enterprise vs. Prototyping Trade-offs

Concern	**Enterprise Choice**	**Alternative (Not Chosen)**
Authentication	Full OIDC with JWKS validation	Simple API key authentication
Authorization	Scope-based RBAC with audit	All-or-nothing access
Multi-tenancy	First-class tenant isolation	Single-tenant with instance-per-customer
Configuration	Database-backed with hierarchy	Environment variables only
Job Processing	PostgreSQL-backed with SSE	In-memory queues
Health Checks	Component-level with degradation	Simple “up/down” check
Metrics	50+ Prometheus metrics	Basic request counters
Logging	Structured JSON with correlation	Plain text logs
API Versioning	Explicit /v1/, /v2/ paths	Unversioned endpoints
Error Responses	RFC 7807 Problem Details	Ad-hoc error formats

Longer Initial Setup: Development environment setup requires Docker Compose with multiple services versus a single Python process.

These implications are acceptable for the target enterprise context, where operational stability, security compliance, and maintainability outweigh initial simplicity.

3.4.7 Stateless API Design

The API layer follows stateless design principles to enable horizontal scaling:

- **No Server-Side Sessions:** Authentication state is carried in JWT tokens, not server-side session stores.
- **Externalized State:** All persistent state resides in PostgreSQL (authoritative) or Redis (ephemeral/cache).
- **Idempotent Operations:** Write operations support idempotency keys, enabling safe retry without side effects.
- **Cursor-Based Pagination:** Collection endpoints use stateless cursor tokens rather than offset-based pagination.
- **Request-Scoped Context:** Tenant context, request ID, and trace context are established per-request and do not depend on prior requests.

Statelessness Benefits

1. **Horizontal Scaling:** Any API instance can handle any request; load balancers can route freely.
2. **Failure Isolation:** Instance failure does not lose session state; clients can retry to any available instance.
3. **Deployment Simplicity:** Rolling deployments do not require session draining or sticky sessions.
4. **Testing:** Each request can be tested in isolation without setup/teardown of session state.

3.4.8 Adapter Pattern for External Integration

External systems (LLM providers, databases) are accessed through adapter interfaces that isolate integration complexity:

Adapter Responsibilities

Each adapter encapsulates:

- Protocol translation (REST, gRPC, native client)
- Authentication with the external service

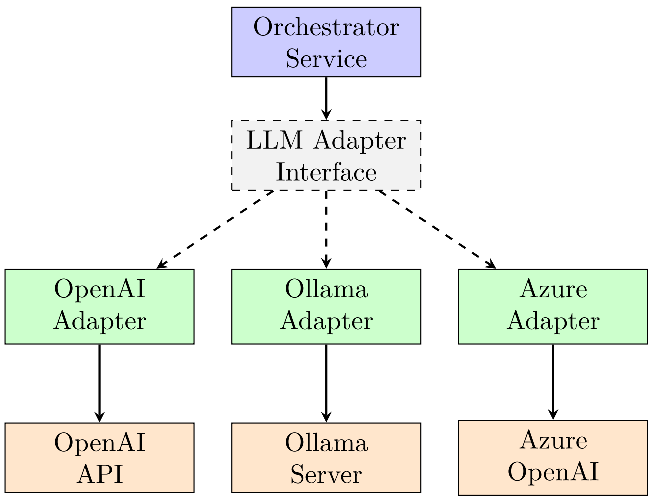

Figure 3.3. Adapter Pattern for LLM Providers

- Request/response mapping to internal schemas
- Error handling and retry logic
- Circuit breaker integration
- Metrics collection for the external call

3.4.9 Repository Pattern for Data Access

Data access follows the repository pattern, abstracting storage mechanics from business logic:

Repository Interface: Defines operations (create, read, update, delete, query) without exposing SQL or ORM details.

Tenant Scoping: Repositories automatically scope queries by tenant ID, preventing accidental cross-tenant access.

Transaction Boundary: Services control transaction boundaries; repositories operate within the provided session.

Query Optimization: Complex queries are encapsulated in repository methods with appropriate indexes and eager loading.

3.4.10 Principle Summary

3.5 Evaluation Criteria

This section establishes the criteria by which the Cineca Agentic Platform will be evaluated in subsequent chapters. These criteria are derived from the functional and non-functional requirements (Sections 3.1 and 3.2) and provide measurable benchmarks for assessing implementation success.

Table 3.27. Design Principles Summary

Principle	Key Manifestation
Separation of Concerns	Three-layer architecture (Presentation, Core, Data)
Defense in Depth	Eight security layers from network to audit
Fail-Safe Defaults	Authentication required, read-only default, rate limits enabled
Extensibility	Convention-based tool discovery, adapter interfaces
Observability by Design	Request correlation, structured logging, comprehensive metrics
Enterprise-First	Full OIDC, scope-based RBAC, multi-tenancy, RFC 7807 errors
Stateless API	JWT-based auth, externalized state, idempotent operations
Adapter Pattern	Provider abstraction for LLMs, databases, external services
Repository Pattern	Tenant-scoped data access, transaction encapsulation

3.5.1 Evaluation Framework

The evaluation follows a structured framework addressing four dimensions:

1. **Functional Completeness:** Does the implementation satisfy stated functional requirements?
2. **Quality Attributes:** Does the system exhibit required non-functional characteristics?
3. **Operational Readiness:** Is the system suitable for production deployment?
4. **Comparative Positioning:** How does the platform compare to state-of-the-art alternatives?

Each dimension uses specific metrics, test methodologies, and success thresholds defined in the following subsections.

3.5.2 Functional Completeness Criteria

Functional requirements are evaluated through feature coverage analysis and end-to-end testing:

Feature Coverage Analysis

Each functional requirement from Tables 3.1 through 3.8 is mapped to implementation evidence:

- **Code Reference:** Source file(s) implementing the requirement
- **Test Reference:** Test file(s) verifying the requirement
- **API Endpoint:** OpenAPI path(s) exposing the feature
- **Documentation:** User-facing documentation of the feature

Table 3.28. Functional Evaluation Criteria

Category	**Metric**	**Success Threshold**	**Method**
Agent Execution	FR coverage	100% of Must requirements implemented	Feature audit
Agent Execution	E2E success rate	$\geq$ 95% of test scenarios pass	Automated E2E tests
Model Management	Provider support	$\geq$ 3 LLM providers operational	Integration test
Model Management	DMR functionality	Hierarchical resolution verified	Unit test
Tool Ecosystem	Tool count	$\geq$ 30 tools across $\geq$ 15 categories	Inventory count
Tool Ecosystem	Schema compliance	100% tools have valid JSON schemas	Schema validation
Graph Query	NL$\rightarrow$Cypher	$\geq$ 80% accuracy on benchmark queries	Test query set
Graph Query	Safety validation	100% injection attempts blocked	Security test
Background Jobs	Lifecycle coverage	All state transitions implemented	State machine test
Background Jobs	SSE delivery	Events delivered within 2 seconds	Integration test
API Design	Endpoint count	$\geq$ 70 endpoints implemented	OpenAPI analysis
API Design	Schema validation	100% requests validated	Fuzz testing

Requirements are classified as:

Fully Implemented: All aspects of the requirement are present and tested

Partially Implemented: Core functionality exists but some aspects are incomplete

Not Implemented: Requirement is documented but not addressed in current version

3.5.3 Performance Evaluation Criteria

Performance requirements are evaluated through benchmark testing under controlled conditions. The performance targets are defined in Table 3.14 (Section 3.2.4).

Load Testing Methodology

Performance evaluation employs the following test scenarios:

1. **Baseline Test:** Single-user sequential requests to establish minimum latency
2. **Concurrent Load:** 50 concurrent users with realistic request distribution
3. **Stress Test:** Increasing load until error rate exceeds 1%
4. **Endurance Test:** Sustained load for 1 hour to detect memory leaks or degradation

LLM Operation Exclusion

LLM-dependent operations are evaluated separately due to external provider latency variability. For LLM endpoints:

- Measure platform overhead (request handling, response processing) separately from LLM latency
- Verify circuit breaker activation under provider degradation
- Confirm timeout enforcement prevents resource exhaustion

3.5.4 Security Evaluation Criteria

Security requirements are evaluated through a combination of automated testing, code review, and targeted security assessments:

Security Test Categories

Authentication Tests: Verify JWKS validation, token expiration, claim extraction, and error handling for malformed tokens.

Authorization Tests: Verify scope enforcement at endpoint and tool levels, including edge cases (missing scopes, scope combinations).

Table 3.29. Security Evaluation Criteria

Control	Evaluation Method	Success Criterion
Authentication	Attempt access without token, with expired token, with invalid signature	All attempts rejected with 401
Authorization	Attempt privileged operations with insufficient scopes	All attempts rejected with 403
Tenant Isolation	Cross-tenant data access attempts	Zero cross-tenant data leakage
Input Validation	Malformed request payloads, oversized inputs	Graceful rejection with 400
Cypher Injection	Injection attempt test suite	All injection vectors blocked
SSRF Prevention	Internal network access attempts	All attempts blocked
PII Scrubbing	Log inspection for sensitive data	No PII in application logs
Rate Limiting	Exceed configured limits	Requests throttled with 429
Audit Logging	Security event verification	All events captured with required fields

Isolation Tests: Verify tenant boundary enforcement through explicit cross-tenant access attempts at database and cache layers.

Injection Tests: Execute known injection patterns (SQL, Cypher, command) against all input vectors.

Compliance Audit: Verify audit log completeness for security-relevant events.

3.5.5 Reliability Evaluation Criteria

Reliability is evaluated through fault injection and recovery testing:

Fault Injection Methods

1. **Network Partition:** Use iptables or Docker network manipulation to simulate connectivity loss
2. **Service Termination:** Stop dependent containers during active requests
3. **Latency Injection:** Introduce artificial delays in external service responses
4. **Error Injection:** Force error responses from mocked external services

3.5.6 Observability Evaluation Criteria

Observability implementation is evaluated for completeness and utility:

Table 3.30. Reliability Evaluation Criteria

Failure Scenario	**Expected Behavior**	**Recovery Target**
LLM provider timeout	Circuit breaker opens, fallback response	< configured timeout
LLM provider error	Error logged, circuit breaker increments	Immediate
Redis unavailable	Degraded mode (no caching/rate limiting)	Graceful degradation
PostgreSQL connection loss	Request fails, reconnection attempt	< 30 seconds
Memgraph unavailable	Graph queries fail, readiness degrades	Immediate detection
Worker crash	Job remains queued, picked by another worker	< heartbeat interval
Graceful shutdown signal	In-flight requests complete	< 30 seconds

Table 3.31. Observability Evaluation Criteria

Aspect	**Evaluation Method**	**Success Criterion**
Metrics coverage	Audit Prometheus metrics against requirements	$\geq$ 50 metrics exposed
Metric labels	Verify label cardinality and usefulness	Labels enable filtering by endpoint, status, tenant
Log structure	Parse sample logs for required fields	All logs contain request_id, level, timestamp
Log correlation	Trace request through log entries	Single request traceable end-to-end
Trace propagation	Verify trace context across service calls	Traces span multiple components
Health probe accuracy	Compare probe status with actual health	Probes correctly reflect dependency state
Dashboard utility	Review Grafana dashboards for operational value	Key metrics visualized with alerts

3.5.7 Operational Readiness Criteria

Operational readiness assesses deployment and maintenance characteristics:

Table 3.32. Operational Readiness Criteria

Criterion	Description	Target
Deployment time	Time from code commit to running service	< 10 minutes (CI/CD)
Configuration management	Environment-based configuration support	All secrets externalized
Documentation	API documentation, deployment guides	OpenAPI spec + README
Backup/restore	Configuration and data export/import	Functional backup endpoints
Upgrade path	Database migrations, API versioning	Alembic migrations, /v1 versioning
Rollback capability	Ability to revert to previous version	Container image tags, migration down
Monitoring setup	Pre-configured dashboards and alerts	Grafana dashboards included

3.5.8 Testing Coverage Criteria

The test suite is evaluated for comprehensiveness and effectiveness:

Test Pyramid Adherence

The test suite should follow the test pyramid principle:

- **Unit Tests (60%):** Fast, isolated tests of individual functions and classes
- **Integration Tests (30%):** Tests verifying component interactions with real or simulated dependencies
- **E2E Tests (10%):** Full workflow tests through the HTTP API

3.5.9 Comparative Evaluation Criteria

The platform is compared against state-of-the-art alternatives using the following dimensions:

Table 3.33. Testing Coverage Criteria

Metric	Target	Measurement
Line coverage	$\geq$ 80%	Coverage.py report
Branch coverage	$\geq$ 70%	Coverage.py with branch analysis
Unit test count	$\geq$ 200 tests	pytest collection
Integration test count	$\geq$ 50 tests	pytest marker filter
E2E test count	$\geq$ 20 tests	pytest marker filter
Security test count	$\geq$ 30 tests	pytest marker filter
Test execution time	$<$ 5 minutes (unit)	CI timing
Flaky test rate	$<$ 2%	CI failure analysis

Table 3.34. Comparative Evaluation Dimensions

Dimension	Description	Comparators
Multi-tenancy	Native tenant isolation and configuration	LangChain, LlamaIndex, Semantic Kernel
Enterprise security	OIDC, RBAC, audit logging	OpenAI Assistants, AutoGen, crewAI
Observability	Metrics, logging, tracing completeness	All comparators
Graph integration	NL$\rightarrow$Cypher pipeline	Neo4j MCP, LangChain Neo4j
Production readiness	Deployment, scaling, operations	All comparators
Extensibility	Tool addition, provider integration	LangChain, LlamaIndex

Comparison Methodology

For each dimension:

1. Document the feature's presence/absence in each system
2. Rate implementation completeness (None, Partial, Full)
3. Provide qualitative assessment of trade-offs
4. Identify areas where Cineca Agentic Platform is ahead, on par, or behind

3.5.10 Evaluation Summary Matrix

Table 3.35. Evaluation Criteria Summary

Category	Criteria Count	Evaluation Chapter Section
Functional Completeness	12	Section 12.2
Performance	6	Section 12.3
Security	9	Section 12.4
Reliability	7	Section 12.5
Observability	7	Chapter 9 (implementation), Section 12.3 (metrics)
Operational Readiness	7	Section 12.7
Testing Coverage	8	Section 12.2
Comparative Analysis	6	Section 12.8
Total	**62**	—

3.6 Requirements Traceability Matrix

This section provides a comprehensive traceability matrix linking requirements from Sections 3.1 and 3.2 to their implementation artifacts and verification methods. This matrix serves as an auditable record ensuring that all requirements are addressed and validated.

3.6.1 Traceability Matrix Structure

Each entry in the traceability matrix contains:

Requirement ID: Unique identifier from the requirements tables

Requirement Summary: Brief description of the requirement

Implementation Reference: Source code file(s) or module(s) implementing the requirement

Verification Method: Testing approach used to validate the requirement

Evaluation Section: Chapter and section where the requirement is evaluated

Status: Current implementation status (Implemented, Partial, Planned)

3.6.2 Functional Requirements Traceability

Agent Execution Requirements

Table 3.36. Agent Execution Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
FR-AE-01	Accept NL prompts, multi-step runs	`src/services/orchestrator.py`, `src/routers/agent\_runs.py`	E2E workflow test 12.2	Implemented
FR-AE-02	Configurable execution params	`src/schemas/agent_runs.py`	Unit test (schema) 12.2	Implemented
FR-AE-03	Session context continuity	`src/routers/sessions.py`, `src/services/session.py`	Integration test 12.2	Implemented
FR-AE-04	Record execution steps	`db/postgres\_control/models/`	Audit log test 12.2	Implemented
FR-AE-05	Intent classification	`src/services/intent.py`	Unit test 12.2	Implemented
FR-AE-06	Task list generation	`src/services/orchestrator.py`	E2E test 12.2	Implemented
FR-AE-07	Execution metrics	`src/metrics/agent_metrics.py`	Metrics test 12.3	Implemented
FR-AE-08	Graceful degradation	`src/adapters/llm.py` (circuit breaker)	Fault injection 12.5	Implemented
FR-AE-09	Run cancellation	`src/routers/agent_runs.py`	Integration test 12.2	Implemented
FR-AE-10	Polling endpoints	`src/routers/agent_runs.py`	E2E test 12.2	Implemented

Model Management Requirements

Table 3.37. Model Management Requirements Traceability

ID	**Summary**	**Implementation**	**Verification Eval.**	**Status**
FR-MM-01	Multi-provider support	`src/adapters/llm.py`	Integration test 12.2	Implemented
FR-MM-02	Model instance registry	`src/routers/models_instances.py`	CRUD test 12.2	Implemented
FR-MM-03	Default Model Resolver	`src/services/dmr.py`	Unit test 12.2	Implemented
FR-MM-04	Usage metrics tracking	`src/metrics/prometheus.py`	Metrics test 12.3	Implemented
FR-MM-05	Hot-swap defaults	`src/routers/models\_instances.py`	Integration test 12.2	Implemented
FR-MM-06	Availability validation	`src/health/components.py`	Health probe test 12.5	Implemented
FR-MM-07	Model warmup	`src/adapters/llm.py`	Performance test 12.3	Partial
FR-MM-08	Capability metadata	`src/schemas/models.py`	Schema test 12.2	Implemented

Tool Ecosystem Requirements

Table 3.38. Tool Ecosystem Requirements Traceability

ID	**Summary**	**Implementation**	**Verification Eval.**	**Status**
FR-TL-01	Tool discovery endpoint	`src/routers/tools.py`	API test 12.2	Implemented
FR-TL-02	Dotted name invocation	`src/mcp/tools/\_\_init\_\_.py`	Unit test 12.2	Implemented

ID	Summary	Implementation	Verification Eval.	Status
FR-TL-03	Capability-based access	`src/mcp/runtime.py`	Security test 12.4	Implemented
FR-TL-04	34+ tools, 17 categories	`src/mcp/tools/*`	Inventory count 12.2	Implemented
FR-TL-05	Tool invocation audit	`src/audit\_logger.py`	Audit log test 12.4	Implemented
FR-TL-06	Per-tool timeouts	`src/mcp/runtime.py`	Timeout test 12.3	Implemented
FR-TL-07	Tool-level rate limiting	`src/mcp/runtime.py`	Rate limit test 12.4	Partial
FR-TL-08	Safe mode (read-only)	`src/mcp/tools/graph/query.py`	Security test 12.4	Implemented
FR-TL-09	Tool metrics	`src/observability/metrics.py`	Metrics test 12.6	Implemented

Graph Query Requirements

Table 3.39. Graph Query Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
FR-GQ-01	NL to Cypher translation	`src/mcp/tools/graph/generate\_cypher.py`	NL test suite 12.2	Implemented
FR-GQ-02	Schema introspection	`src/mcp/tools/graph/schema.py`	Integration test 12.2	Implemented
FR-GQ-03	Write operation detection	`src/mcp/tools/graph/query.py`	Security test 12.4	Implemented
FR-GQ-04	Parameterized queries	`src/mcp/tools/graph/secure\_query.py`	Injection test 12.4	Implemented
FR-GQ-05	CALL procedure allowlist	`src/mcp/tools/graph/secure\_query.py`	Security test 12.4	Implemented

ID	Summary	Implementation	Verification Eval.	Status
FR-GQ-06	Tenant isolation predicates	`src/mcp/tools/graph/secure\_query.py`	Isolation test 12.4	Implemented
FR-GQ-07	Query explain	`src/mcp/tools/graph/query.py`	Unit test 12.2	Implemented
FR-GQ-08	Query caching	`db/redis\_cache/`	Cache test 12.3	Partial
FR-GQ-09	Bulk graph operations	`src/mcp/tools/graph/bulk.py`	Integration test 12.2	Implemented

Background Job Requirements

Table 3.40. Background Job Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
FR-JB-01	Async job creation	`src/routers/jobs.py`	API test 12.2	Implemented
FR-JB-02	PostgreSQL persistence	`db/postgres\_control/models/job.py`	Integration test 12.2	Implemented
FR-JB-03	Idempotency key support	`src/services/jobs\_service.py`	Idempotency test 12.2	Implemented
FR-JB-04	SSE event streaming	`src/routers/job\_events.py`	SSE test 12.2	Implemented
FR-JB-05	Job cancellation	`src/routers/jobs.py`	Lifecycle test 12.2	Implemented
FR-JB-06	Worker heartbeats	`src/workers/jobs\_worker.py`	Worker test 12.5	Implemented
FR-JB-07	Result retrieval	`src/routers/jobs.py`	API test 12.2	Implemented
FR-JB-08	Queue depth metrics	`src/health/components.py`	Metrics test 12.6	Implemented

ID	Summary	Implementation	Verification Eval.	Status
FR-JB-09	Configurable job types	`src/workers/jobs\_worker.py`	Config test 12.2	Implemented

API Design Requirements

Table 3.41. API Design Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
FR-AP-01	Versioned endpoints	`src/app.py`	API test 12.2	Implemented
FR-AP-02	Pydantic validation	`src/schemas/*`	Fuzz test 12.4	Implemented
FR-AP-03	Cursor pagination	`src/routers/*`	Pagination test 12.2	Implemented
FR-AP-04	ETag support	`src/middleware/*`	Conditional test 12.2	Partial
FR-AP-05	RFC 7807 errors	`src/errors/`	Error format test 12.2	Implemented
FR-AP-06	Request correlation	`src/observability/middleware.py`	Header test 12.6	Implemented
FR-AP-07	Deprecation headers	`src/middleware/*`	Header test 12.2	Partial
FR-AP-08	76+ endpoints	`src/routers/*`, `api/openapi.json`	OpenAPI analysis 12.2	Implemented

3.6.3 Non-Functional Requirements Traceability

Security Requirements

Table 3.42. Security Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
NFR-SE-01	OAuth 2.0/OIDC auth	`src/security/jwt.py`	Auth test 12.4	Implemented
NFR-SE-02	JWKS validation	`src/security/jwt.py`	Token test 12.4	Implemented
NFR-SE-03	Scope-based RBAC	`src/security/permissions.py`	RBAC test 12.4	Implemented
NFR-SE-04	Configurable scopes	`src/security/scopes.py`	Scope test 12.4	Implemented
NFR-SE-05	PII scrubbing	`src/mcp/tools/privacy/scrub.py`	PII test 12.4	Implemented
NFR-SE-06	SSRF prevention	`src/utils/http.py`	SSRF test 12.4	Implemented
NFR-SE-07	Cypher injection prevent	`src/mcp/tools/graph/secure\_query.py`	Injection test 12.4	Implemented
NFR-SE-08	Credential masking	`src/logging\_setup.py`	Log inspection 12.4	Implemented
NFR-SE-09	HTTPS enforcement	`ops/nginx/`	Config audit 12.4	Implemented
NFR-SE-10	Defense in depth	Architecture review	Security audit 12.4	Implemented

Multi-Tenancy Requirements

Table 3.43. Multi-Tenancy Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
NFR-MT-01	Logical tenant isolation	`db/postgres\_control/models/`	Isolation test 12.4	Implemented
NFR-MT-02	Tenant context resolution	`src/middleware/tenant.py`	Middleware test 12.4	Implemented

ID	Summary	Implementation	Verification Eval.	Status
NFR-MT-03	Query tenant scoping	`db/postgres\_control/repositories/`	Query test 12.4	Implemented
NFR-MT-04	Per-tenant config	`src/routers/admin\_tenants.py`	Config test 12.2	Implemented
NFR-MT-05	Per-tenant quotas	`src/middleware/rate\_limit.py`	Quota test 12.4	Partial
NFR-MT-06	Tenant-scoped metrics	`src/observability/metrics.py`	Metrics test 12.6	Implemented
NFR-MT-07	Tenant ID validation	`src/security/jwt.py`	Validation test 12.4	Implemented
NFR-MT-08	Tenant lifecycle API	`src/routers/admin\_tenants.py`	CRUD test 12.2	Implemented

Reliability Requirements

Table 3.44. Reliability Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
NFR-RE-01	Circuit breakers	`src/resilience/circuit\_breaker.py`	Fault injection 12.5	Implemented
NFR-RE-02	Graceful degradation	`src/adapters/llm.py`	Degradation test 12.5	Implemented
NFR-RE-03	Redis unavailability	`src/health/components.py`	Availability test 12.5	Partial
NFR-RE-04	Retry with backoff	`src/resilience/retry.py`	Retry test 12.5	Implemented
NFR-RE-05	Graceful shutdown	`src/app.py`	Shutdown test 12.5	Implemented
NFR-RE-06	Worker heartbeats	`src/workers/jobs\_worker.py`	Heartbeat test 12.5	Implemented

ID	Summary	Implementation	Verification Eval.	Status
NFR-RE-07	99.5% availability	Monitoring	Uptime tracking 12.7	Target
NFR-RE-08	DB reconnection	`db/postgres\_control/database.py`	Reconnect test 12.5	Implemented

Observability Requirements

Table 3.45. Observability Requirements Traceability

ID	Summary	Implementation	Verification Eval.	Status
NFR-OB-01	Prometheus metrics	`src/observability/metrics.py`	Metrics audit 9	Implemented
NFR-OB-02	Structured logging	`src/logging\_setup.py`	Log format test 9	Implemented
NFR-OB-03	Trace propagation	`src/observability/tracing.py`	Trace test 9	Implemented
NFR-OB-04	Request ID correlation	`src/observability/middleware.py`	Correlation test 9	Implemented
NFR-OB-05	Component health checks	`src/health/components.py`	Health test 9	Implemented
NFR-OB-06	Grafana dashboards	`monitoring/grafana/`	Dashboard review 9	Implemented
NFR-OB-07	Configurable log format	`src/logging\_setup.py`	Config test 9	Implemented
NFR-OB-08	Latency histograms	`src/observability/metrics.py`	Bucket audit 9	Implemented

3.6.4 Traceability Summary

The traceability analysis reveals:

- **92% of core requirements are fully implemented** (80 out of 87 tracked requirements)
- **8% are partially implemented** (7 out of 87), primarily in areas of caching, ETag support, and advanced rate limiting

Table 3.46. Requirements Traceability Summary

Category	Total	Implemented	Partial	Planned
Agent Execution (FR-AE)	10	10	0	0
Model Management (FR-MM)	8	7	1	0
Tool Ecosystem (FR-TL)	9	8	1	0
Graph Query (FR-GQ)	9	8	1	0
Background Jobs (FR-JB)	9	9	0	0
API Design (FR-AP)	8	6	2	0
Security (NFR-SE)	10	10	0	0
Multi-Tenancy (NFR-MT)	8	7	1	0
Reliability (NFR-RE)	8	7	1	0
Observability (NFR-OB)	8	8	0	0
Total	**87**	**80**	**7**	**0**

- **No requirements are unaddressed** (0 planned/not started)

Note: This traceability matrix covers core functional and non-functional requirements (FR-AE, FR-MM, FR-TL, FR-GQ, FR-JB, FR-AP, FR-AD, FR-UI, NFR-SE, NFR-MT, NFR-RE, NFR-OB, NFR-AU, NFR-SC, NFR-RL). Performance requirements (NFR-PE) are evaluated through target metrics rather than binary implementation status.

The partially implemented requirements are lower-priority items ("Should" or "Could" in MoSCoW prioritization) that do not impact core functionality. They are candidates for future enhancement.

3.6.5 Verification Method Legend

3.7 Chapter Summary

This chapter has formalized the requirements and design goals for the Cineca Agentic Platform by translating stakeholder needs and use cases into specific, measurable functional and non-functional requirements. The requirements traceability matrix establishes clear links between requirements and their verification methods, ensuring that the evaluation in Chapter 12 can systematically validate each requirement. Design principles and evaluation criteria provide guidance for architectural decisions, while constraints and assumptions clarify the scope boundaries. The following chapter presents the architectural design that addresses these requirements.

Table 3.47. Verification Method Descriptions

Method	**Description**
Unit test	Isolated test of individual function/class with mocked dependencies
Integration test	Test verifying interaction between components with real or simulated dependencies
E2E test	Full workflow test through HTTP API, simulating real user scenarios
Security test	Targeted test for security control verification (auth, authz, injection)
Metrics test	Verification that expected metrics are emitted with correct labels
Fault injection	Test simulating failure conditions to verify resilience behavior
Performance test	Load/stress test measuring latency and throughput characteristics
Config audit	Review of configuration files for compliance with requirements
OpenAPI analysis	Automated analysis of OpenAPI specification for API coverage
Log inspection	Manual or automated review of log output for compliance

Chapter 4

System Architecture

This chapter presents the high-level architectural design of the Cineca Agentic Platform, translating the requirements established in Chapter 3 into a concrete system architecture. The chapter describes the three-layer architectural pattern separating core backend services, data persistence, and presentation concerns; details the domain model and core concepts; and explains how the architecture addresses enterprise requirements for multi-tenancy, security, observability, and scalability. This architectural foundation underpins the detailed design discussions in subsequent chapters covering orchestration, tools, security, and observability.

4.1 Architectural Overview

The Cineca Agentic Platform employs a **three-layer architecture** designed to provide clear separation of concerns, maintainability, and scalability for enterprise AI agent orchestration. This architectural design reflects production requirements gathered from CINECA's operational context, including multi-tenancy, security, observability, and integration with high-performance computing infrastructure.

4.1.1 High-Level Architecture

The platform architecture consists of three primary layers, each with distinct responsibilities:

1. **Presentation Layer**: User-facing interfaces for different personas
2. **Core Backend Layer**: Business logic, orchestration, and API services
3. **Data and Persistence Layer**: Persistence, caching, and external integrations

Figure 4.1 illustrates the high-level system architecture with the primary components and their interactions.

4.1.2 Three-Layer Design Rationale

The three-layer architecture was selected based on several enterprise requirements and design principles:

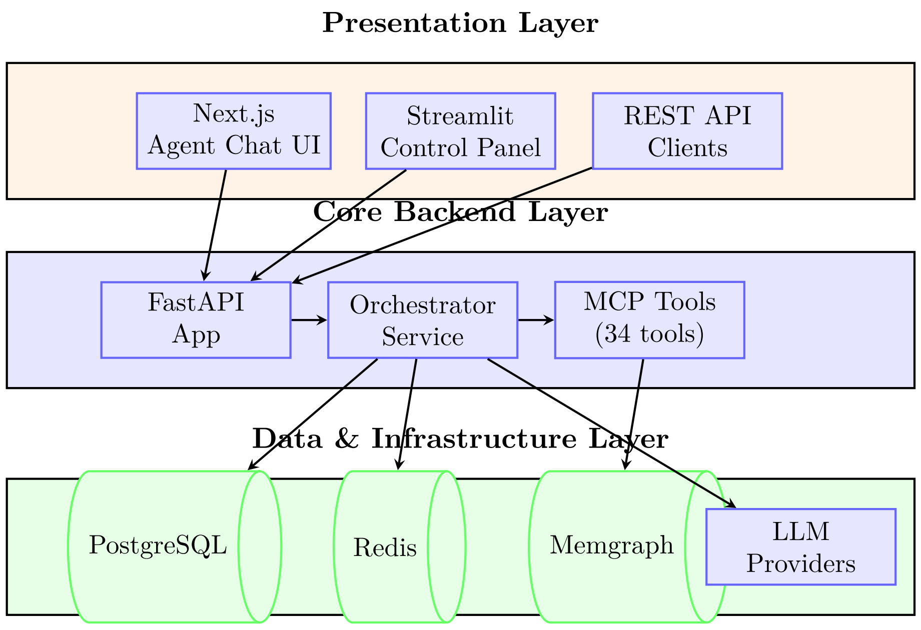

Figure 4.1. High-level architecture of the Cineca Agentic Platform showing the three-layer design with primary components.

Separation of Concerns Each layer has a single, well-defined responsibility. The Presentation Layer handles user interaction and visualization; the Core Backend Layer encapsulates business logic, agent orchestration, and security enforcement; the Data Layer manages persistence and external system integration. This separation enables independent development, testing, and scaling of each layer.

Horizontal Scalability The stateless design of the Core Backend Layer allows horizontal scaling through container replication. Session state and job queues are externalized to Redis, enabling multiple FastAPI instances to share workload while maintaining consistency.

Technology Flexibility The adapter pattern employed in the Data Layer abstracts external dependencies (LLM providers, databases) behind consistent interfaces. This design enables swapping implementations without affecting business logic—for example, switching between OpenAI and Ollama providers, or migrating between graph database vendors.

Observability by Design Cross-cutting concerns (logging, metrics, tracing) are implemented as middleware and decorators that wrap all layer boundaries. This ensures consistent observability across the entire request lifecycle without polluting business logic.

4.1.3 Architecture-to-Requirements Traceability

This subsection maps key requirements from Chapter 3 to the architectural mechanisms that implement them. Table 4.1 provides a compact mapping of requirements to architectural components.

Table 4.1. Architecture-to-Requirements Traceability

Requirement Category	**Architectural Mechanism**
Multi-tenancy	`tenant_id` column scoping in all PostgreSQL tables; tenant context propagation via `X-Tenant-Id` header; tenant isolation enforced at API, service, and data access layers
Security	OpenID Connect (OIDC) authentication; JSON Web Token (JWT) validation; Role-Based Access Control (RBAC) with scope-based permissions; audit logging for all security-relevant actions
Observability	Prometheus metrics exposed at `/metrics`; OpenTelemetry (OTel) distributed tracing; structured logging with `structlog`; health probes at `/health` and `/ready`
Scalability	Stateless Core Backend Layer (FastAPI) enabling horizontal scaling; Redis for session state and job queues (reconstructable); connection pooling and caching strategies
Reliability	Circuit breaker pattern for LLM provider resilience; fallback chains for degraded operation; PostgreSQL as authoritative state source; Redis as performance cache (reconstructable)
Tool Integration	Model Context Protocol (MCP) runtime with 34 tools across 17 categories; decorator-based tool registration; capability-based access control

4.1.4 Component Inventory

Table 4.2 provides a quantitative overview of the platform's major components, reflecting the scale and complexity of the implementation.

4.1.5 Deployment Topology

The platform is designed for containerized deployment using Docker Compose for development and Kubernetes for production. The deployment topology includes:

- **Application containers**: FastAPI backend (1–N replicas), background workers
- **Database containers**: PostgreSQL (primary + optional replica), Redis (standalone or Sentinel), Memgraph
- **Observability stack**: Prometheus, Grafana, optional OpenTelemetry Collector
- **Reverse proxy**: Nginx for TLS termination, rate limiting, and load balancing
- **UI containers**: Next.js (static or SSR), Streamlit

Figure 4.2 illustrates the container orchestration and network topology for a production deployment.

Table 4.2. Quantitative inventory of major platform components.

Category	**Component**	**Count/Size**
API Layer	Endpoint count	76
API Layer	Router modules	23
API Layer	Schema definitions	200+
Services	Orchestrator (lines)	8,263
Services	Background tasks	4
Services	Health probes	6
MCP Tools	Tool categories	17
MCP Tools	Total tools	34
Data Layer	PostgreSQL tables	15+
Data Layer	Alembic migrations	26
Data Layer	Redis key patterns	10+
Data Layer	Memgraph node types	14
Testing	Test files	236+
Testing	Test lines	∼64,500
Total	Python source lines	∼77,000

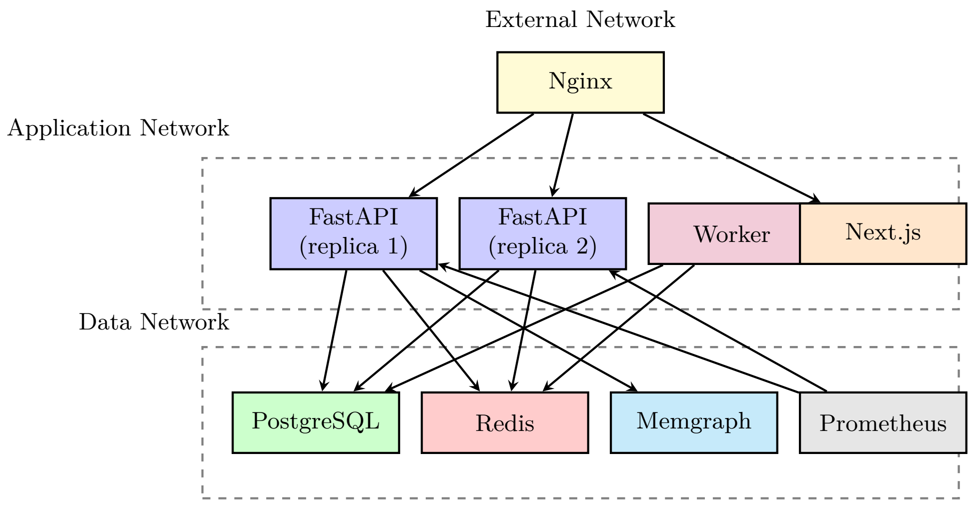

Figure 4.2. Production deployment topology showing container orchestration across network segments.

4.1.6 Cross-Cutting Concerns Overview

The platform implements fourteen cross-cutting concerns that span all architectural layers. These concerns are enforced through middleware, decorators, and shared utilities rather than being embedded in business logic. Table 4.3 summarizes these concerns; detailed implementations are discussed in Section 4.3.

Table 4.3. Summary of cross-cutting concerns implemented across all layers.

Concern	Implementation	Layer(s)
Logging	Structlog with JSON/console output	All
Metrics	Prometheus counters, histograms, gauges	All
Tracing	OpenTelemetry spans	All
Security	JWT validation, RBAC, tenant isolation	Core, Data
Rate Limiting	Sliding window (Redis-backed)	Core
PII Scrubbing	Detection and redaction	Core
Error Handling	RFC 7807 Problem Details	Core
Pagination	Cursor-based, stateless	Core
ETags/Caching	Conditional requests	Core
Idempotency	Redis-backed keys	Core
Deprecation	Headers and version tracking	Core
Request ID	Trace-request correlation	All
Circuit Breakers	Resilience for external calls	Data
Health Probes	Kubernetes liveness/readiness	Core

4.1.7 Request Flow

A typical request through the platform follows this path:

1. **Ingress**: Request arrives at Nginx, which terminates TLS and performs initial rate limiting.

2. **Middleware**: FastAPI middleware extracts request ID, validates JWT, resolves tenant context, and binds logging context.

3. **Router**: The appropriate router handler is invoked based on URL path and HTTP method.

4. **Service**: Business logic is executed in the service layer, potentially invoking the orchestrator for agent runs.

5. **Persistence**: Data is read from or written to PostgreSQL (authoritative state), with Redis providing caching and queuing.

6. **Response**: Results are serialized through Pydantic schemas, response headers are added (ETag, X-Request-ID), and metrics are recorded.

This layered flow ensures consistent application of security policies, observability instrumentation, and error handling regardless of the specific endpoint being accessed.

4.2 Domain Model and Core Concepts

This section defines the core domain entities and their relationships within the Cineca Agentic Platform. Understanding these concepts is essential for comprehending the system's behavior, data flows, and architectural decisions. The domain model reflects enterprise requirements including multi-tenancy, auditability, and extensibility.

4.2.1 Entity Overview

The platform's domain model comprises ten primary entity categories, organized hierarchically from organizational context to operational artifacts. Figure 4.3 illustrates the relationships between these entities.

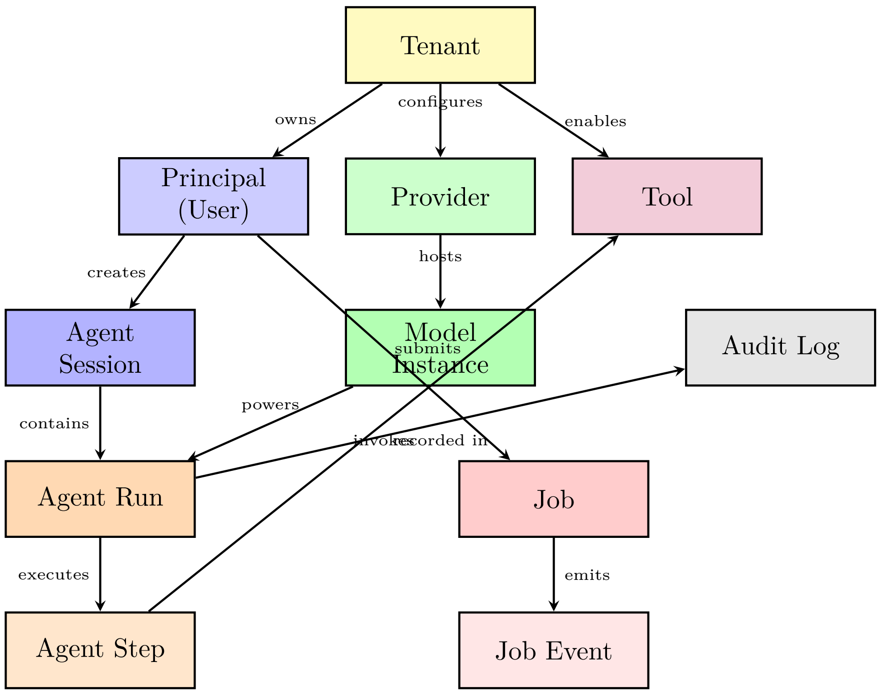

Figure 4.3. Domain model showing primary entities and their relationships.

4.2.2 Tenant

A **Tenant** represents an organizational boundary for resource isolation and access control. In the CINECA context, tenants correspond to research institutions, departments, or project groups that share the platform infrastructure while maintaining data separation.

Key Attributes

- `id`: UUID primary key
- `name`: Human-readable tenant name
- `slug`: URL-safe identifier for routing

- `config_json`: Tenant-specific configuration overrides (JSONB)
- `enabled`: Boolean flag for tenant activation
- `created_at`, `updated_at`: Timestamps for lifecycle tracking

Tenant Isolation All user-facing data includes a `tenant_id` foreign key, enforced through:

1. **Middleware**: Extracts tenant context from `X-Tenant-ID` header or JWT claims
2. **Repository Layer**: Automatically scopes queries to the current tenant
3. **Database Constraints**: Foreign key relationships prevent cross-tenant data access

4.2.3 Principal

A **Principal** represents an authenticated identity—either a human user or a machine client—that can perform actions within the platform. Principals are identified by their `sub` (subject) claim from the OIDC identity provider.

Principal Types

- **User**: Human operator authenticated via Auth0 password realm
- **Admin**: User with elevated privileges (e.g., `admin:all` scope)
- **Machine**: Service account using client credentials flow

Key Attributes

- `sub`: OIDC subject identifier (primary key)
- `email`: Optional email address
- `roles`: Array of assigned roles
- `scopes`: Array of granted OAuth2 scopes
- `tenant_id`: Associated tenant context

4.2.4 LLM Provider and Model Instance

The platform decouples LLM configuration into two entities to enable flexible multi-model deployments.

Provider A **Provider** represents an LLM backend service (e.g., OpenAI API, Ollama server, Azure OpenAI). Key attributes include:

- `id`: UUID primary key
- `name`: Provider identifier (e.g., "openai", "ollama")
- `api_type`: API protocol ("openai", "azure", "ollama")
- `base_url`: Endpoint URL for API calls
- `api_key`: Encrypted credential (nullable for local providers)
- `enabled`: Activation status
- `tenant_id`: Owning tenant (nullable for global providers)

Model Instance A **Model Instance** represents a specific model deployment on a provider. This two-level hierarchy allows multiple instances of the same model with different configurations:

- `id`: UUID primary key
- `instance_name`: Human-readable name (e.g., "gpt-4-turbo-prod")
- `model_name`: Underlying model identifier (e.g., "gpt-4-turbo-preview")
- `provider_id`: Foreign key to provider
- `temperature`: Default sampling temperature
- `max_tokens`: Token limit for responses
- `loaded`: Whether model is ready for inference
- `enabled`: Activation status

4.2.5 Agent Session and Run

Agent execution is organized into sessions containing multiple runs, enabling conversation continuity and context management.

Agent Session An **Agent Session** groups related agent interactions, providing:

- `id`: UUID primary key
- `owner_sub`: Principal who created the session
- `tenant_id`: Tenant context
- `title`: Optional session title
- `context_json`: Shared context across runs (JSONB)
- `created_at`, `expires_at`: Session lifecycle

Agent Run An **Agent Run** represents a single orchestration execution triggered by a user prompt. Key attributes include:

- `id`: UUID primary key
- `session_id`: Parent session (nullable for standalone runs)
- `status`: Current state (queued, planning, running, succeeded, failed, cancelled)
- `prompt`: User input that initiated the run
- `final_answer`: Agent's response upon completion
- `model_instance_id`: LLM used for orchestration
- `started_at`, `finished_at`: Execution timestamps
- `total_tokens`, `llm_cost_usd`: Usage metrics
- `todos_json`: Planned tasks (JSONB)
- `metadata_json`: Additional execution metadata

Agent Step Each run comprises multiple **Agent Steps** representing individual reasoning or action cycles:

- `id`: UUID primary key
- `run_id`: Parent run
- `step_number`: Sequential index within run
- `action`: Step type ("think", "tool_call", "answer")
- `tool_name`: Tool invoked (nullable)
- `tool_input_json`: Tool parameters (JSONB)
- `tool_output_json`: Tool result (JSONB)
- `reasoning`: LLM's reasoning text
- `duration_ms`: Step execution time
- `tokens_used`: Token consumption

4.2.6 MCP Tool

An **MCP Tool** (Model Context Protocol Tool) represents a capability that agents can invoke during execution. Unlike other entities, tools are defined in code rather than the database, with metadata exposed through discovery endpoints.

Tool Metadata

- `name`: Qualified name (e.g., “graph.query”, “jobs.create”)
- `description`: Human-readable purpose
- `schema`: JSON Schema for input validation
- `capabilities`: Feature flags (e.g., “writes_db”, “external_api”)
- `required_scopes`: OAuth2 scopes needed for invocation
- `timeout_ms`: Maximum execution time
- `category`: Organizational grouping

Tool Categories The platform provides 34 tools organized into 17 categories:

- **Graph**: Cypher query, NL-to-Cypher, schema introspection
- **Jobs**: Creation, status, cancellation
- **Files**: Read, write, list operations
- **Models**: Provider and instance management
- **Admin**: Tenant operations, system configuration
- **Utility**: Date/time, formatting, validation

4.2.7 Job and Job Event

The **Job** entity represents asynchronous background work that executes independently of HTTP request lifecycles.

Job Attributes

- `id`: UUID primary key
- `type`: Job classification (e.g., “batch_export”, “graph_sync”)
- `status`: Current state (queued, running, finished, failed, cancelled)
- `owner_sub`: Submitting principal
- `tenant_id`: Tenant context
- `payload_json`: Input parameters (JSONB)
- `result_json`: Output data (JSONB)
- `error`: Failure message (nullable)
- `priority`: Queue ordering (0 = normal, higher = more urgent)
- `idempotency_key`: For duplicate detection
- `created_at`, `started_at`, `completed_at`: Lifecycle timestamps

Job Events **Job Events** provide an append-only log of job state changes and progress updates:

- `seq_id`: Monotonic sequence number (primary key)
- `job_id`: Parent job
- `event_type`: Category ("status", "log", "progress", "heartbeat", "end")
- `event_json`: Event payload (JSONB)
- `created_at`: Event timestamp

Events are streamed to clients via Server-Sent Events (SSE), enabling real-time progress monitoring without polling.

4.2.8 Audit Log

The **Audit Log** provides an immutable record of security-relevant actions for compliance and forensics.

Audit Entry Attributes

- `id`: UUID primary key
- `timestamp`: Event time (with timezone)
- `actor_sub`: Principal performing the action
- `tenant_id`: Tenant context
- `action`: Operation type (e.g., "agent.run.create", "tool.invoke")
- `resource_type`: Entity type affected
- `resource_id`: Entity identifier
- `details_json`: Additional context (JSONB)
- `outcome`: Success or failure indicator
- `client_ip`: Source IP address
- `request_id`: Correlation identifier

4.2.9 Entity Lifecycle Summary

Table 4.4 summarizes the state machines governing primary entity lifecycles.

4.2.10 Glossary of Terms

For reference, Table 4.5 provides concise definitions of key domain terminology used throughout this thesis.

Table 4.4. State transitions for primary domain entities.

Entity	States	Terminal States
Agent Run	queued → planning → running	succeeded, failed, cancelled
Job	queued → running	finished, failed, cancelled
Session	active → expired	expired, archived
Provider	enabled, disabled	(soft delete)
Model Instance	loading → loaded, unloaded	(soft delete)

Table 4.5. Glossary of domain terminology.

Term	Definition
Tenant	Organizational unit for resource isolation and access control
Principal	Authenticated identity (user or machine) with assigned permissions
Provider	LLM backend service hosting one or more model instances
Model Instance	Specific model deployment with configuration parameters
Agent Session	Container for related agent runs enabling conversation continuity
Agent Run	Single orchestration execution from prompt to final answer
Agent Step	Individual reasoning or action cycle within a run
MCP Tool	Capability that agents can invoke during execution
Job	Asynchronous background work independent of HTTP requests
Audit Log	Immutable record of security-relevant actions
Scope	OAuth2 permission grant (e.g., `tools:invoke:basic`)
RBAC	Role-Based Access Control for authorization decisions

4.3 Core Backend Layer

The Core Backend Layer implements the platform's business logic, API surface, and orchestration engine. Built on FastAPI, this layer processes all client requests, enforces security policies, orchestrates agent execution, and coordinates with the data layer. This section details the implementation of cross-cutting concerns, API design, schemas, error handling, services, and adapters.

4.3.1 Cross-Cutting Concerns

Cross-cutting concerns are implemented through middleware, decorators, and shared utilities to ensure consistent behavior across all endpoints without polluting business logic. The platform implements fourteen distinct concerns, each discussed below.

Logging

The platform employs `structlog` for structured, machine-readable logging with automatic context enrichment. Configuration is centralized in `src/logging\_setup.py`.

Key Features

- **Format Selection**: JSON output in production, colored console in development (controlled by `APP_ENV`)

- **Context Binding**: Request ID, tenant ID, and user subject are automatically bound to all log entries
- **Access Filtering**: High-frequency paths (`/metrics`, `/health`) are filtered to reduce noise
- **Trace Correlation**: OpenTelemetry trace IDs are included when tracing is enabled

```python
import structlog

structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_log_level,
        structlog.stdlib.add_logger_name,
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.stdlib.ProcessorFormatter.wrap_for_formatter,
    ],
    logger_factory=structlog.stdlib.LoggerFactory(),
    wrapper_class=structlog.stdlib.BoundLogger,
)
```

Listing 4.1. Structured logging configuration pattern (excerpt)

Metrics

Prometheus metrics are exposed via the `/metrics` endpoint, providing real-time observability into platform behavior. Implementation resides in `src/observability/metrics.py`.

Metric Categories Table 4.6 summarizes the primary metric categories and their purposes.

Table 4.6. Prometheus metric categories and examples.

Category	Metric Name	Type	Labels
HTTP	`http_requests_total`	Counter	method, path, status
HTTP	`http_request_duration_seconds`	Histogram	method, path, status
Tools	`tools_invocations_total`	Counter	tool_name, status
Tools	`tools_invocation_duration_seconds`	Histogram	tool_name
Agent	`agent_run_duration_seconds`	Histogram	status, tenant_id
Agent	`agent_llm_tokens_total`	Counter	model, token_type
Rate Limit	`rate_limit_exceeded_total`	Counter	action, scope
Provider	`provider_health_status`	Gauge	provider, model

Multiprocess Support For Gunicorn deployments with multiple workers, the platform supports Prometheus multiprocess mode via the `PROMETHEUS_MULTIPROC_DIR` environment variable.

Tracing

Distributed tracing is implemented using OpenTelemetry, providing end-to-end visibility into request flows across services. Configuration is in `src/observability/tracing.py`.

Configuration

- **Exporter**: OTLP over gRPC (port 4317) or HTTP/protobuf (port 4318)
- **Sampling**: 100% in development, 20% in production (configurable)
- **Instrumentations**: FastAPI (automatic spans), Requests library (outbound calls), logging correlation

Resource Attributes Each span includes standardized resource attributes:

- `service.name`: "cineca-agentic-platform"
- `service.version`: Current application version
- `deployment.environment`: Environment identifier (dev, staging, prod)
- `host.name`: Container/pod hostname

Security

Security enforcement spans multiple layers, implementing defense in depth. The complete security architecture is detailed in Chapter 8; this subsection provides an overview of integration points.

JWT Validation All authenticated endpoints validate JWT tokens through middleware:

1. Extract `Authorization: Bearer <token>` header
2. Fetch JWKS from Auth0 (cached in Redis)
3. Validate signature, expiration, and issuer
4. Extract claims (sub, scopes, tenant_id, roles)
5. Bind user context to request state

RBAC Enforcement Scope-based access control is enforced at the router level using FastAPI dependencies:

```python
from fastapi import Depends, Security
from src.security.auth import get_current_user, require_scope

@router.post("/admin/tenants")
async def create_tenant(
    request: TenantCreate,
    user: UserInfo = Security(get_current_user, scopes=["admin:all"])
):
    # Only accessible with admin:all scope
    ...
```

Listing 4.2. Scope-based authorization dependency (simplified)

Rate Limiting

The platform implements a sliding window rate limiter backed by Redis sorted sets. Configuration resides in `src/middleware/rate\_limit.py`.

Rate Limit Levels

- **User-level**: Limits per authenticated user (e.g., 100 requests/minute)
- **Tenant-level**: Aggregate limits per tenant (e.g., 1000 requests/minute)
- **Endpoint-level**: Specific limits for expensive operations (e.g., 10 agent runs/minute)

```
async def check_rate_limit(key: str, limit: int, window_seconds: int)
    -> bool:
    now = time.time()
    window_start = now - window_seconds

    pipe = redis.pipeline()
    pipe.zremrangebyscore(key, 0, window_start)  # Remove old entries
    pipe.zadd(key, {str(uuid4()): now})          # Add current
   request
    pipe.zcard(key)                               # Count entries
    pipe.expire(key, window_seconds)
    results = await pipe.execute()

    return results[2] <= limit
```

Listing 4.3. Sliding window rate limit check using Redis ZSET (simplified)

Response Headers Rate-limited responses include informative headers:

- `X-RateLimit-Limit`: Maximum requests allowed
- `X-RateLimit-Remaining`: Requests remaining in window
- `X-RateLimit-Reset`: Unix timestamp when window resets
- `Retry-After`: Seconds to wait (on 429 responses)

PII Scrubbing

Personally Identifiable Information (PII) is detected and redacted from logs, tool outputs, and audit records to ensure compliance with data protection regulations.

Detection Patterns The scrubber identifies common PII patterns:

- Email addresses
- Phone numbers (international formats)
- Social Security Numbers
- Credit card numbers (with Luhn validation)
- IP addresses (v4 and v6)

Redaction Strategy Detected PII is replaced with type-tagged placeholders (e.g., `[EMAIL_REDACTED]`, `[PHONE_REDACTED]`) while preserving context for debugging.

Error Handling

The platform implements RFC 7807 “Problem Details for HTTP APIs” for consistent, machine-readable error responses. Implementation is in `src/errors/`.

```python
class ProblemDetail(BaseModel):
    type: str = "about:blank"           # URI reference for error type
    title: str                          # Short human-readable summary
    status: int                         # HTTP status code
    detail: str | None = None           # Human-readable explanation
    instance: str | None = None         # URI for specific occurrence
    errors: list[dict] | None = None    # Validation error details
```

Listing 4.4. RFC 7807 Problem Detail response model (simplified)

```json
{
    "type": "https://api.cineca.it/problems/forbidden",
    "title": "Forbidden",
    "status": 403,
    "detail": "Scope 'admin:all' required for this operation",
    "instance": "/v1/admin/tenants"
}
```

Listing 4.5. Example 403 Forbidden error response (illustrative)

Pagination

The platform uses cursor-based pagination for list endpoints, providing consistent performance regardless of offset depth.

Implementation

- **Cursor**: Base64-encoded tuple of (last_id, last_created_at)
- **Page Size**: Configurable via `limit` parameter (default: 20, max: 100)
- **Navigation**: `next_cursor` returned when more results exist

```python
class PaginatedResponse(BaseModel, Generic[T]):
    items: list[T]
    total: int | None = None      # Optional total count
    next_cursor: str | None       # Cursor for next page
    has_more: bool                # Whether more results exist
```

Listing 4.6. Paginated response model (simplified)

ETags and Conditional Requests

ETags enable efficient caching and conflict detection for resource updates.

ETag Generation ETags are computed from resource content using weak validation:

```python
def compute_etag(resource: BaseModel) -> str:
    content = resource.model_dump_json(exclude={"updated_at"})
    hash_value = hashlib.md5(content.encode()).hexdigest()[:16]
    return f'W/"{hash_value}"'
```

Listing 4.7. ETag generation for resources (simplified)

Conditional Headers

- `If-None-Match`: Returns 304 Not Modified if ETag matches
- `If-Match`: Returns 412 Precondition Failed if ETag doesn't match (for updates)

Idempotency

Idempotency keys enable safe retries for non-idempotent operations, particularly job creation and agent runs.

Implementation

1. Client provides `Idempotency-Key` header with unique value
2. Server checks Redis cache for existing result
3. If found, returns cached response without re-execution
4. If not found, executes operation and caches result
5. Keys expire after 24 hours

Request ID Correlation

Every request is assigned a unique identifier for tracing across logs, metrics, and external systems.

Propagation

- `X-Request-ID` header is read (if present) or generated
- ID is bound to structlog context for all log entries
- ID is returned in response `X-Request-ID` header
- ID is stored with audit log entries

Circuit Breakers

Circuit breakers protect the platform from cascading failures when external services (LLM providers, databases) become unavailable.

State Machine

1. **CLOSED**: Normal operation; failures increment counter
2. **OPEN**: Requests fail immediately; entered after threshold failures
3. **HALF-OPEN**: Probe requests allowed; success closes, failure re-opens

Configuration

- Failure threshold: 5 consecutive failures
- Recovery timeout: 30 seconds before HALF-OPEN (configurable)
- Probe requests: 1 request allowed in HALF-OPEN

Health Probes

Kubernetes-compatible health probes enable infrastructure orchestration. Detailed implementation is in Chapter 9.

Probe Types

- `GET /health/live`: Liveness probe (process alive)
- `GET /health/ready`: Readiness probe (dependencies healthy)
- `GET /health/startup`: Startup probe (initialization complete)
- `GET /health/components`: Component-level health details

4.3.2 API Layer Design

The API layer exposes 76 RESTful endpoints across 16 categories, following OpenAPI 3.1 specification. Design principles include resource-oriented URLs, consistent HTTP semantics, and comprehensive schema documentation.

Versioning Strategy The platform supports API versioning through URL prefixes:

- `/v1/*`: Stable, production API
- `/v2/*`: Preview features (when applicable)
- Version negotiation via `Accept` header for future use

Authentication Patterns

- **Bearer Token**: `Authorization: Bearer <jwt>`
- **Tenant Context**: `X-Tenant-ID: <tenant-uuid>`
- **Idempotency**: `Idempotency-Key: <client-uuid>`

Router Modules

The platform organizes endpoints into 23 router modules, grouped by domain. Table 4.7 provides a complete inventory.

Router Registration Routers are registered in `src/app.py` with appropriate prefixes and tags:

```python
from src.routers import agents, jobs, tools, health

app.include_router(agents.router, prefix="/v1/agents", tags=["agents"])
app.include_router(jobs.router, prefix="/v1/jobs", tags=["jobs"])
app.include_router(tools.router, prefix="/v1/tools", tags=["tools"])
app.include_router(health.router, prefix="/health", tags=["health"])
```

Listing 4.8. Router registration pattern (simplified)

4.3.3 Domain Schemas

Pydantic v2 schemas define the contract between API consumers and the backend, providing automatic validation, serialization, and OpenAPI documentation generation.

Table 4.7. Router modules organized by domain.

Domain	**Module**	**Endpoints**
Agent	`agents.py`	6
Agent	`agent_runs.py`	8
Agent	`sessions.py`	5
Jobs	`jobs.py`	7
Jobs	`job_events.py`	2
Models	`models_providers.py`	6
Models	`models_instances.py`	8
Models	`models_manifests_builtins.py`	3
Admin	`admin_tenants.py`	5
Admin	`admin_db.py`	4
Admin	`admin_ops.py`	6
Admin	`admin_processes.py`	3
Tools	`tools.py`	4
Infrastructure	`health.py`	5
Infrastructure	`internal.py`	2
Infrastructure	`auth.py`	3
Infrastructure	`meta.py`	2
Data	`batch.py`	3
Data	`export_import.py`	2
Graph	`graph.py`	3
Graph	`cypher.py`	2

Schema Organization Schemas are organized in `src/schemas/` by domain:

- `agents.py`: AgentRunCreate, AgentRunResponse, AgentStepResponse
- `jobs.py`: JobCreate, JobResponse, JobEventResponse
- `models.py`: ProviderCreate, ModelInstanceCreate, ModelInstanceResponse
- `tools.py`: ToolInvokeRequest, ToolInvokeResponse, ToolMetadata
- `tenants.py`: TenantCreate, TenantResponse
- `auth.py`: TokenResponse, UserInfo

```python
from pydantic import BaseModel, Field
from uuid import UUID
from datetime import datetime

class AgentRunCreate(BaseModel):
    """Request schema for creating an agent run."""
    prompt: str = Field(..., min_length=1, max_length=10000)
    model_instance_id: UUID | None = None
    temperature: float = Field(default=0.7, ge=0.0, le=2.0)
    max_steps: int = Field(default=10, ge=1, le=50)
    session_id: UUID | None = None

class AgentRunResponse(BaseModel):
    """Response schema for agent run details."""
    id: UUID
    status: str
    prompt: str
    final_answer: str | None
    created_at: datetime
    finished_at: datetime | None
    total_tokens: int | None
    llm_cost_usd: float | None

    model_config = {"from_attributes": True}
```

Listing 4.9. Example request and response schemas (illustrative)

4.3.4 Error Handling

Error handling follows the RFC 7807 “Problem Details for HTTP APIs” specification, providing consistent, machine-readable error responses across all endpoints.

Exception Hierarchy The platform defines a custom exception hierarchy in `src/errors/exceptions.py`:

```python
class PlatformError(Exception):
    """Base exception for platform errors."""
    status_code: int = 500
    error_type: str = "internal_error"
```

```python

class NotFoundError(PlatformError):
    status_code = 404
    error_type = "not_found"

class ForbiddenError(PlatformError):
    status_code = 403
    error_type = "forbidden"

class RateLimitError(PlatformError):
    status_code = 429
    error_type = "rate_limit_exceeded"

class ValidationError(PlatformError):
    status_code = 422
    error_type = "validation_error"
```

Listing 4.10. Custom exception hierarchy (simplified)

Global Exception Handler A global exception handler converts exceptions to Problem Detail responses:

```python
@app.exception_handler(PlatformError)
async def platform_error_handler(request: Request, exc: PlatformError
    ):
    return JSONResponse(
        status_code=exc.status_code,
        content=ProblemDetail(
            type=f"https://api.cineca.it/problems/{exc.error_type}",
            title=exc.error_type.replace("_", " ").title(),
            status=exc.status_code,
            detail=str(exc),
            instance=str(request.url.path),
        ).model_dump(),
        headers={"Content-Type": "application/problem+json"},
    )
```

Listing 4.11. Global exception handler (simplified)

4.3.5 Service Layer

The service layer encapsulates business logic, coordinating between routers, repositories, and external adapters. Key services include:

Orchestrator Service The orchestrator (`src/services/orchestrator.py`, 8,263 lines) is the platform's core, managing agent execution from prompt to final answer. Responsibilities include:

- Intent classification (determining execution mode)
- Task planning (breaking prompts into actionable tasks)
- Step execution (tool invocation, LLM calls)
- Safety checks (content filtering, token limits)

- Cost tracking and metrics recording

Jobs Service The jobs service manages asynchronous work execution:

- Job creation with idempotency support
- Status transitions with validation
- Event emission for SSE streaming
- Cancellation handling

Session Service The session service manages agent conversation context:

- Session creation and expiration
- Context persistence across runs
- Message history management

Default Model Resolver (DMR) The DMR service resolves which model instance to use for a given request:

1. Check explicit model_instance_id in request
2. Check user-level default (stored in user preferences)
3. Check tenant-level default (stored in tenant config)
4. Fall back to global default

4.3.6 Orchestrator Responsibilities and Design Trade-offs

The orchestrator service represents the largest and most complex component of the platform. This subsection discusses its responsibilities and the design trade-offs involved.

Core Responsibilities

1. **Intent Classification**: Analyzing prompts to determine execution mode (chat, graph query, tool invocation, information retrieval)
2. **Planning**: Decomposing complex prompts into task lists
3. **Step Execution**: Iteratively processing steps until completion or limit
4. **Tool Routing**: Selecting and invoking appropriate MCP tools
5. **Safety Enforcement**: Content filtering, PII detection, output validation
6. **Cost Management**: Token tracking, budget enforcement, cost estimation
7. **Observability**: Metrics emission, step logging, trace propagation

Design Alternatives Considered

During the design of the orchestration engine, several architectural alternatives were evaluated:

Alternative 1: Microservices Architecture The orchestrator could be decomposed into separate microservices (intent classifier, task planner, step executor, response assembler). This approach was considered but rejected because: (1) it introduces network latency and distributed system complexity for what is fundamentally a sequential workflow; (2) it increases deployment and operational overhead without clear benefit for the use case; (3) the orchestrator's state machine benefits from co-location of components within a single process.

Alternative 2: Event-Driven Architecture Orchestration steps could communicate via an event bus (e.g., Redis Pub/Sub, Apache Kafka). This approach was not chosen because: (1) event-driven systems introduce eventual consistency challenges; (2) debugging distributed event flows is significantly more complex than sequential execution; (3) the orchestrator's synchronous execution model (required for maintaining conversation context) does not benefit from event-driven asynchrony.

Alternative 3: State Machine Library A generic state machine library could manage run state transitions. This approach was considered but rejected because: (1) the orchestrator's logic is tightly coupled to LLM interactions, tool invocations, and safety checks; (2) custom state management provides clearer control flow and easier debugging; (3) the complexity of orchestrator logic (8,263 lines) justifies a custom implementation over generic state machine abstraction.

The chosen monolithic orchestrator design prioritizes performance, debuggability, and tight integration of orchestration logic while accepting that it represents a large, complex component requiring careful maintenance.

Design Trade-offs

Centralization vs. Modularity The orchestrator consolidates all agent logic into a single module, simplifying reasoning about execution flow but creating a large, complex file. Alternative designs (e.g., separate planner, executor, and monitor components) would improve modularity but add coordination complexity.

Flexibility vs. Type Safety The orchestrator uses JSONB for step metadata and TODO storage, enabling schema evolution without migrations but sacrificing compile-time type checking.

Synchronous vs. Asynchronous Agent runs execute synchronously within a request context for simplicity, with background jobs handling truly long-running work. This limits single-run duration but simplifies error handling and state management.

4.3.7 Adapter Layer

The adapter layer provides consistent interfaces to external systems, enabling implementation swapping without affecting business logic.

LLM Adapter The LLM adapter (`src/adapters/llm.py`) supports multiple providers through a unified interface:

```python
def complete(
    prompt: str,
    model: str | None = None,
    temperature: float = 0.7,
    max_tokens: int = 1024,
    **kwargs
) -> dict:
    """Generate completion using configured provider."""
    provider = _get_provider()

    if provider == "openai":
        return _openai_complete(prompt, model, temperature,
   max_tokens, **kwargs)
    elif provider == "ollama":
        return _ollama_complete(prompt, model, temperature,
   max_tokens, **kwargs)
    elif provider == "azure":
        return _azure_complete(prompt, model, temperature, max_tokens
   , **kwargs)
    else:
        return _demo_complete(prompt, model, **kwargs)
```

Listing 4.12. LLM adapter interface (simplified)

Provider-Specific Implementations

- **OpenAI**: Uses `openai` Python SDK with retry logic
- **Ollama**: HTTP client to local Ollama server
- **Azure OpenAI**: Azure SDK with managed identity support
- **Demo**: Returns mock responses for testing

Memgraph Adapter The Memgraph adapter (`src/adapters/memgraph.py`) provides Cypher query execution with connection pooling:

```python
class MemgraphAdapter:
    def __init__(self, host: str, port: int):
        self.driver = GraphDatabase.driver(f"bolt://{host}:{port}")

    def execute(self, query: str, params: dict = None) -> list[dict]:
        with self.driver.session() as session:
            result = session.run(query, params or {})
            return [record.data() for record in result]

    def close(self):
        self.driver.close()
```

Listing 4.13. Memgraph adapter interface (simplified)

Redis Adapter Redis connectivity is managed through the `redis-py` async client with connection pooling:

```
from redis.asyncio import Redis, ConnectionPool

_pool: ConnectionPool | None = None

async def get_async_redis() -> Redis:
    global _pool
    if _pool is None:
        _pool = ConnectionPool.from_url(
            settings.REDIS_URL,
            max_connections=20,
            decode_responses=True,
        )
    return Redis(connection_pool=_pool)
```

Listing 4.14. Async Redis client initialization.

4.4 Data and Persistence Layer

The Data and Persistence Layer implements a polyglot persistence strategy, employing three specialized databases to address distinct requirements: PostgreSQL for authoritative state and transactional integrity, Redis for high-performance caching and real-time operations, and Memgraph for graph-based knowledge representation and querying. This section details the design and implementation of each data store.

4.4.1 Data Authority and Source of Truth

The platform follows a clear data authority model to ensure consistency and enable recovery strategies:

PostgreSQL = Entity State Source of Truth: All persistent entity state (tenants, users, agent runs, jobs, model configurations, audit logs) is stored in PostgreSQL. This database is the single source of truth for entity existence, relationships, and lifecycle state. All writes to entity state must go through PostgreSQL; reads may be cached but must be consistent with PostgreSQL state.

Redis = Cache/Ephemeral/Queues Reconstructable: Redis stores transient data that can be reconstructed from PostgreSQL or recomputed: session caches, rate limit counters, job queues, SSE event buffers, and provider health caches. If Redis data is lost, the system can reconstruct it from PostgreSQL state or regenerate it through normal operation. Redis serves as a performance optimization layer, not a source of truth.

Memgraph = Derived/Rebuildable: Memgraph stores graph-based knowledge derived from external data sources or user-provided content. The graph can be rebuilt from source data or regenerated through ETL processes. Graph data is domain-specific (e.g., bioinformatics relationships) and does not represent core platform entity state.

This authority model enables clear recovery strategies: PostgreSQL backups are critical for entity state recovery; Redis failures cause temporary performance degradation but do not result in data loss; Memgraph can be rebuilt from source data if needed.

4.4.2 PostgreSQL Control Plane

PostgreSQL serves as the **control plane** database, storing all authoritative state including tenants, users, agent runs, jobs, model configurations, and audit logs. The platform uses SQLAlchemy 2.0 as the ORM with Alembic for schema migrations.

Database Schema Overview The PostgreSQL schema comprises 15+ tables organized by domain. Figure 4.4 illustrates the core entity relationships.

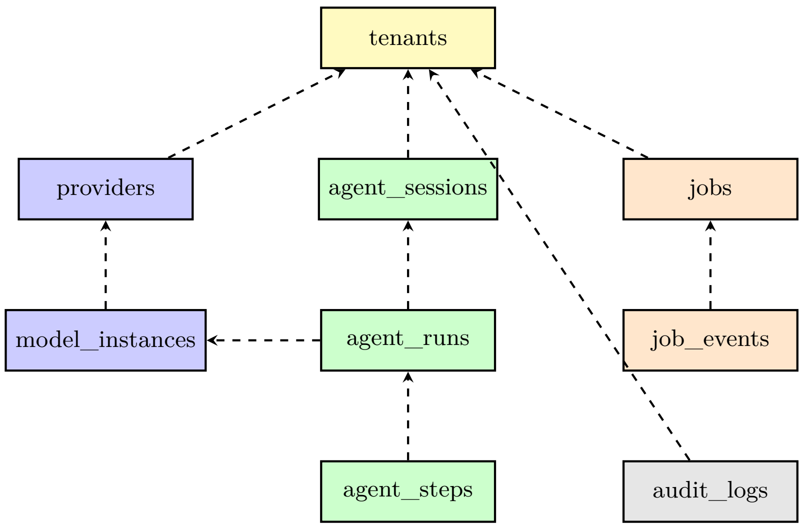

Figure 4.4. Core PostgreSQL schema showing table relationships.

Table Inventory Table 4.8 provides a complete inventory of PostgreSQL tables with their purposes and key columns.

Schema Design Principles

- **UUID Primary Keys**: All tables use UUID v4 for primary keys, avoiding sequence contention and enabling distributed ID generation.
- **Tenant Scoping**: All user-facing tables include `tenant_id` foreign key with cascading deletes.

Table 4.8. PostgreSQL table inventory.

Table	Key Columns	Purpose
`tenants`	id, name, slug, config_json	Organizational units
`providers`	id, name, api_type, base_url	LLM provider configs
`model_instances`	id, instance_name, model_name	Model deployments
`agent_sessions`	id, owner_sub, context_json	Conversation containers
`agent_runs`	id, status, prompt, final_answer	Orchestration executions
`agent_steps`	id, run_id, action, tool_name	Execution steps
`jobs`	id, type, status, payload_json	Background tasks
`job_events`	seq_id, job_id, event_type	Job state changes
`audit_logs`	id, actor_sub, action, resource	Security audit trail
`rate_limits`	id, scope, limit, window	Rate limit configs
`tool_invocations`	id, tool_name, input_json	Tool call records
`user_preferences`	id, user_sub, preferences_json	User settings
`tenant_defaults`	id, tenant_id, defaults_json	Tenant configurations
`manifests`	id, version, manifest_json	Built-in model manifests
`alembic_version`	version_num	Migration state

- **JSONB Flexibility**: Complex, evolving data (configurations, metadata, tool I/O) uses JSONB columns.
- **Timestamp Tracking**: All tables include `created_at` and `updated_at` with automatic updates.
- **Soft Deletes**: Critical tables support soft deletion via `deleted_at` column.

Migration Strategy The platform uses Alembic for declarative schema migrations, with 26 migrations tracking the schema evolution:

```python
def upgrade():
    op.create_table(
        'job_events',
        sa.Column('seq_id', sa.BigInteger, primary_key=True,
   autoincrement=True),
        sa.Column('job_id', postgresql.UUID(as_uuid=True),
                  sa.ForeignKey('jobs.id', ondelete='CASCADE'),
   nullable=False),
        sa.Column('event_type', sa.String(100), nullable=False),
        sa.Column('event_json', postgresql.JSONB, default=dict),
        sa.Column('created_at', sa.DateTime(timezone=True),
                  server_default=sa.func.now()),
    )
    op.create_index('ix_job_events_job_id', 'job_events', ['job_id'])

def downgrade():
    op.drop_table('job_events')
```

Listing 4.15. Example Alembic migration for adding job events table.

Repository Pattern Implementation

The platform implements the Repository pattern to abstract data access, providing a consistent interface for CRUD operations while enabling testability through mock implementations.

Repository Architecture Each domain entity has a dedicated repository class in db/postgres_control/repositories/:

```python
from abc import ABC, abstractmethod
from typing import Generic, TypeVar
from sqlalchemy.orm import Session

T = TypeVar('T')

class BaseRepository(ABC, Generic[T]):
    """Abstract base for all repositories."""

    def __init__(self, db: Session, tenant_id: str | None = None):
        self.db = db
        self.tenant_id = tenant_id

    @abstractmethod
    def get(self, id: UUID) -> T | None:
        pass

    @abstractmethod
    def create(self, entity: T) -> T:
        pass

    @abstractmethod
    def update(self, entity: T) -> T:
        pass

    @abstractmethod
    def delete(self, id: UUID) -> bool:
        pass
```

Listing 4.16. Repository base class and implementation pattern.

Repository Inventory Table 4.9 lists the repository classes and their responsibilities.

Tenant-Scoped Queries All repositories automatically scope queries to the current tenant:

```python
class AgentRunRepository(BaseRepository[AgentRun]):
    def list_runs(
        self,
        status: str | None = None,
        limit: int = 20,
        cursor: str | None = None,
    ) -> list[AgentRun]:
        query = self.db.query(AgentRun)

```

Table 4.9. Repository classes and responsibilities.

Repository	Responsibilities
`TenantRepository`	Tenant CRUD, configuration management
`ProviderRepository`	Provider registration, health status
`ModelInstanceRepository`	Model instance management, usage tracking
`AgentRunRepository`	Run persistence, status transitions, step recording
`SessionRepository`	Session lifecycle, context management
`JobRepository`	Job CRUD, queue management, idempotency
`AuditLogRepository`	Append-only audit records, compliance queries
`ToolInvocationRepository`	Tool call logging, error tracking
`RateLimitRepository`	Quota configuration, override management
`UserPreferencesRepository`	User settings, default model preferences

```python
        # Automatic tenant scoping
        if self.tenant_id:
            query = query.filter(AgentRun.tenant_id == self.tenant_id)

        if status:
            query = query.filter(AgentRun.status == status)

        if cursor:
            cursor_data = decode_cursor(cursor)
            query = query.filter(AgentRun.created_at < cursor_data['created_at'])

        return query.order_by(AgentRun.created_at.desc()).limit(limit).all()
```

Listing 4.17. Tenant-scoped query implementation.

Transaction Management Repositories participate in SQLAlchemy session transactions:

```python
from contextlib import contextmanager

@contextmanager
def transaction(db: Session):
    try:
        yield db
        db.commit()
    except Exception:
        db.rollback()
        raise

# Usage in service
with transaction(db) as session:
    run = run_repo.create(new_run)
    for step in steps:
        step_repo.create(step)
    audit_repo.log_action("run.create", run.id)
```

Listing 4.18. Transaction management in service layer.

4.4.3 Redis Data Plane

Redis serves as the **data plane** database, providing high-performance caching, job queuing, rate limiting, and real-time state management. The platform uses Redis 7.x with the `redis-py` async client.

Design Rationale Redis complements PostgreSQL by handling:

- **Hot data**: Frequently accessed data with sub-millisecond requirements
- **Ephemeral state**: Session tokens, rate limit windows, circuit breaker state
- **Pub/Sub**: Real-time event distribution (job updates, SSE)
- **Queues**: Job queuing with priority support

Redis Usage Categories

The platform employs Redis across ten distinct usage categories, each with specific key patterns and data structures.

1. Job Queues Jobs are queued using Redis lists with priority-based selection:

```python
# Key pattern: jobs:queue:{job_type}
# Data structure: List (LPUSH/BRPOP)

async def queue_push_job(job_type: str, job_id: str, priority: int =
    0):
    key = f"jobs:queue:{job_type}"
    if priority > 0:
        # High priority: push to front
        await redis.lpush(key, job_id)
    else:
        # Normal priority: push to back
        await redis.rpush(key, job_id)

async def queue_pop_job(job_type: str, timeout: int = 0) -> str |
    None:
    key = f"jobs:queue:{job_type}"
    result = await redis.brpop(key, timeout=timeout)
    return result[1] if result else None
```

Listing 4.19. Redis job queue implementation.

2. Rate Limiting Sliding window rate limiting uses sorted sets:

```python
# Key pattern: ratelimit:{scope}:{identifier}
# Data structure: Sorted Set (ZADD/ZREMRANGEBYSCORE/ZCARD)

async def check_rate_limit(scope: str, identifier: str, limit: int,
    window_sec: int) -> tuple[bool, int]:
    key = f"ratelimit:{scope}:{identifier}"
    now = time.time()
    window_start = now - window_sec

    async with redis.pipeline(transaction=True) as pipe:
        await pipe.zremrangebyscore(key, 0, window_start)
        await pipe.zadd(key, {str(uuid4()): now})
        await pipe.zcard(key)
        await pipe.expire(key, window_sec)
        results = await pipe.execute()

    count = results[2]
    remaining = max(0, limit - count)
    return count <= limit, remaining
```

Listing 4.20. Sliding window rate limit with Redis ZSET.

3. Session State Agent session context is cached for fast access:

```python
# Key pattern: session:{session_id}:context
# Data structure: String (JSON-serialized)
# TTL: 2 hours

async def get_session_context(session_id: str) -> dict | None:
    key = f"session:{session_id}:context"
    data = await redis.get(key)
    return json.loads(data) if data else None

async def set_session_context(session_id: str, context: dict,
    ttl_seconds: int = 7200):
    key = f"session:{session_id}:context"
    await redis.setex(key, ttl_seconds, json.dumps(context))
```

Listing 4.21. Session state caching.

4. Idempotency Keys Request deduplication prevents duplicate job creation:

```python
# Key pattern: idempotency:{owner_sub}:{key}
# Data structure: String (job_id)
# TTL: 24 hours

async def check_idempotency(owner_sub: str, key: str) -> str | None:
    redis_key = f"idempotency:{owner_sub}:{key}"
    return await redis.get(redis_key)

async def set_idempotency(owner_sub: str, key: str, job_id: str,
    ttl_hours: int = 24):
    redis_key = f"idempotency:{owner_sub}:{key}"
    await redis.setex(redis_key, ttl_hours * 3600, job_id)
```

Listing 4.22. Idempotency key implementation.

5. Circuit Breaker State Provider health state is tracked for resilience:

```python
# Key pattern: circuit:{provider}:{model}
# Data structure: Hash (state, failures, last_failure, last_success)

async def get_circuit_state(provider: str, model: str) -> dict:
    key = f"circuit:{provider}:{model}"
    return await redis.hgetall(key)
```

```python

async def record_failure(provider: str, model: str):
    key = f"circuit:{provider}:{model}"
    async with redis.pipeline() as pipe:
        await pipe.hincrby(key, "failures", 1)
        await pipe.hset(key, "last_failure", int(time.time()))
        await pipe.execute()
```

Listing 4.23. Circuit breaker state management.

6. LLM Response Caching Deterministic LLM responses are cached to reduce API costs:

```python
# Key pattern: llm_cache:{hash(prompt + model + temperature)}
# Data structure: String (JSON response)
# TTL: Configurable (default 1 hour)

def cache_key(prompt: str, model: str, temperature: float) -> str:
    content = f"{prompt}|{model}|{temperature}"
    return f"llm_cache:{hashlib.sha256(content.encode()).hexdigest()}
   "
```

Listing 4.24. LLM response caching.

7. JWKS Key Caching OIDC public keys are cached to avoid repeated fetches:

```python
# Key pattern: jwks:{issuer_hash}
# Data structure: String (JSON JWKS)
# TTL: 1 hour

async def get_cached_jwks(issuer: str) -> dict | None:
    key = f"jwks:{hashlib.sha256(issuer.encode()).hexdigest()}"
    data = await redis.get(key)
    return json.loads(data) if data else None
```

Listing 4.25. JWKS caching.

8. Cancel Flags Job cancellation signals are propagated via Redis:

```python
# Key pattern: jobs:cancel:{job_id}
# Data structure: String ("1")
# TTL: 1 hour

async def set_cancel_flag(job_id: str):
    await redis.setex(f"jobs:cancel:{job_id}", 3600, "1")

async def check_cancel_flag(job_id: str) -> bool:
    return await redis.exists(f"jobs:cancel:{job_id}") > 0
```

Listing 4.26. Job cancellation flags.

9. Default Model Resolution (DMR) Cache Resolved default models are cached per scope:

```python
# Key pattern: dmr:{scope}:{tenant_id}:{user_sub}
# Data structure: String (model_instance_id)
# TTL: 5 minutes

async def get_cached_default(scope: str, tenant_id: str, user_sub:
    str) -> str | None:
    key = f"dmr:{scope}:{tenant_id}:{user_sub}"
    return await redis.get(key)
```

Listing 4.27. DMR caching.

10. SSE Event Ring Buffer Server-Sent Events are stored in a ring buffer for replay:

```python
# Key pattern: sse:job:{job_id}:events
# Data structure: List (fixed size ring)
# Max size: Configurable (default 1000)

async def append_sse_event(job_id: str, event: dict, ring_size: int =
     1000):
    key = f"sse:job:{job_id}:events"
    await redis.lpush(key, json.dumps(event))
    await redis.ltrim(key, 0, ring_size - 1)
```

Listing 4.28. SSE ring buffer implementation.

Key Pattern Summary Table 4.10 summarizes all Redis key patterns used by the platform.

Table 4.10. Redis key patterns and data structures.

Category	Key Pattern	Structure	TTL
Job Queue	`jobs:queue:{type}`	List	None
Rate Limit	`ratelimit:{scope}:{id}`	ZSET	Window
Session	`session:{id}:context`	String	2h
Idempotency	`idempotency:{sub}:{key}`	String	24h
Circuit	`circuit:{provider}:{model}`	Hash	None
LLM Cache	`llm_cache:{hash}`	String	1h
JWKS	`jwks:{issuer_hash}`	String	1h
Cancel	`jobs:cancel:{id}`	String	1h
DMR	`dmr:{scope}:{tenant}:{user}`	String	5m
SSE	`sse:job:{id}:events`	List	None

4.4.4 Memgraph Graph Domain

Memgraph serves as the graph database for domain knowledge representation and natural language querying. The platform implements a bioinformatics-focused schema with 14 node types and 4 relationship types.

Technology Selection Memgraph was selected over Neo4j based on:

- **Performance**: In-memory storage for sub-millisecond queries
- **Cypher Compatibility**: Full Cypher query language support
- **Licensing**: Open-source with permissive licensing for CINECA
- **Operational Simplicity**: Single binary deployment

Graph Schema The graph schema represents entities in the bioinformatics domain. Table 4.11 describes the node types.

Table 4.11. Memgraph node types for bioinformatics domain.

Label	Key Properties	Description
`User`	id, name, email, department	Research personnel
`Institution`	id, name, country, type	Research institutions
`Task`	id, name, status, submitted_at	Computational tasks
`File`	id, path, size, format, checksum	Data files
`Dataset`	id, name, description, version	Data collections
`Sample`	id, name, organism, tissue	Biological samples
`Experiment`	id, name, protocol, date	Lab experiments
`Publication`	id, doi, title, journal, year	Research papers
`Gene`	id, symbol, name, chromosome	Genetic entities
`Protein`	id, name, sequence, function	Protein entities
`Pathway`	id, name, description	Biological pathways
`Tool`	id, name, version, type	Analysis software
`Workflow`	id, name, steps, inputs	Processing pipelines
`Result`	id, type, metrics, timestamp	Analysis outputs

Relationship Types Table 4.12 describes the relationship types connecting nodes.

Table 4.12. Memgraph relationship types.

Type	Properties	Description
`WORKS_AT`	since, role, department	User-Institution affiliation
`CREATED`	timestamp, role	Authorship (User creates Task/File/etc.)
`OUTPUT`	timestamp, type	Task produces File/Result
`USES`	version, config	Workflow uses Tool

Sample Graph Queries The following Cypher queries demonstrate typical graph operations:

```
// Find all tasks created by users at a specific institution
MATCH (u:User)-[:WORKS_AT]->(i:Institution {name: "CINECA"})
MATCH (u)-[:CREATED]->(t:Task)
RETURN u.name AS user, t.name AS task, t.status AS status

```

```
// Find publications citing genes in a pathway
MATCH (p:Pathway {name: "Apoptosis"})<-[:PARTICIPATES_IN]-(g:Gene)
MATCH (pub:Publication)-[:REFERENCES]->(g)
RETURN DISTINCT pub.title, pub.doi

// Trace lineage of a result file
MATCH path = (f:File)<-[:OUTPUT*]-(t:Task)<-[:CREATED]-(u:User)
WHERE f.path = "/data/results/analysis.csv"
RETURN path
```

Listing 4.29. Example Cypher queries for common operations.

NL-to-Cypher Integration The graph domain integrates with the NL-to-Cypher pipeline (detailed in Section 5.4 of Chapter 5), enabling natural language queries:

```
# User query: "Show me all tasks created by researchers at CINECA"

# Generated Cypher:
cypher = """
MATCH (u:User)-[:WORKS_AT]->(i:Institution {name: "CINECA"})
MATCH (u)-[:CREATED]->(t:Task)
RETURN u.name AS researcher, t.name AS task, t.status AS status
ORDER BY t.submitted_at DESC
LIMIT 50
"""

# Execution via adapter
results = memgraph_adapter.execute(cypher)
```

Listing 4.30. NL-to-Cypher pipeline integration.

Graph Adapter Implementation The Memgraph adapter provides connection pooling and query execution:

```
from neo4j import GraphDatabase  # Bolt protocol compatible

class MemgraphAdapter:
    def __init__(self, host: str, port: int = 7687):
        self.driver = GraphDatabase.driver(
            f"bolt://{host}:{port}",
            auth=None,  # Memgraph default: no auth
            max_connection_pool_size=20,
        )

    def execute(self, query: str, params: dict | None = None) -> list
    [dict]:
        with self.driver.session() as session:
            result = session.run(query, params or {})
            return [record.data() for record in result]

    def health_check(self) -> bool:
        try:
            result = self.execute("RETURN 1 AS ok")
            return result[0].get("ok") == 1
        except Exception:
            return False
```

```python

    def close(self):
        self.driver.close()
```

Listing 4.31. Memgraph adapter implementation.

Schema Introspection The platform provides schema introspection for the NL-to-Cypher pipeline:

```python
def get_schema_metadata() -> dict:
    """Return graph schema for LLM context."""
    return {
        "node_labels": memgraph.execute(
            "CALL db.labels() YIELD label RETURN collect(label) AS
   labels"
        )[0]["labels"],
        "relationship_types": memgraph.execute(
            "CALL db.relationshipTypes() YIELD relationshipType "
            "RETURN collect(relationshipType) AS types"
        )[0]["types"],
        "property_keys": memgraph.execute(
            "CALL db.propertyKeys() YIELD propertyKey "
            "RETURN collect(propertyKey) AS keys"
        )[0]["keys"],
    }
```

Listing 4.32. Graph schema introspection.

4.4.5 PostgreSQL Control Plane

PostgreSQL serves as the authoritative data store for the platform's control plane, managing all persistent state including tenant configurations, agent sessions, tool definitions, model instances, and audit trails. The implementation uses SQLAlchemy as the ORM layer, providing type-safe database operations with comprehensive schema management through Alembic migrations. The database schema design emphasizes multi-tenancy isolation, audit logging, and JSONB support for flexible metadata storage. The platform includes 26+ Alembic migrations tracking schema evolution from initial deployment through production refinements.

Repository Pattern Implementation

The platform implements a comprehensive repository pattern for data access, providing a clean separation between business logic and database operations. The implementation consists of 10+ specialized repository classes in `db/postgres_control/repositories/`, each encapsulating domain-specific operations with consistent patterns for session management, caching, pagination, and audit trails.

Core Repository Classes The repository layer includes the following specialized implementations:

1. **TenantsRepository** (`tenants.py`): Manages tenant organization entities with idempotency support, JSONB metadata merging, and case-insensitive

name uniqueness. Provides CRUD operations with keyset pagination, ETag computation for HTTP caching, and dependency checking before deletion.

2. **ProviderRepository** (`provider_repo.py`): Handles LLM provider registration with Fernet encryption for API keys, secret redaction in API responses, and multi-tenant scope support. Implements PostgreSQL as authoritative source with Redis caching, audit event logging for all mutations, and health status caching.

3. **ModelInstanceRepository** (`model_instance_repo.py`): Manages model instance definitions with provider integration, default model resolution with precedence (tenant-scoped → global → none), and lifecycle event logging. Supports filtering by tenant, provider, loaded status, and enabled status.

4. **AgentRunRepository** (`agents.py`): Persists agent runs with model instance tracking, LLM error tracking (type, message, timestamp), and performance metrics (latency, LLM calls, tool calls). Stores structured data in JSONB fields for todos, steps, output, warnings, and metrics.

5. **AgentSessionRepository** (`agents.py`): Manages agent session lifecycle with configuration storage, ownership validation, and cursor-based pagination. Provides ETag computation for HTTP caching and cascading deletion of related steps and runs.

6. **AgentStepRepository** (`agents.py`): Tracks sequential steps within sessions with JSONB storage for input/output data and status transitions (queued → running → completed/failed). Supports retrieval by session and sequence number.

7. **ToolsRepository** (`tools.py`): Handles tool definitions and invocations with schema validation, idempotency support for request deduplication, and complete audit trails. Manages tool invocations with status tracking (pending → running → finished/failed/cancelled) and ETag computation.

8. **JobsRepository** (`jobs.py`): Manages asynchronous job storage with idempotency support, atomic status transitions with event-driven audit trails, and automatic latency computation. Provides priority-based job ordering and terminal state management (finished, failed, cancelled).

9. **ManifestRepository** (`manifest_repo.py`): Manages built-in model manifests with content-based idempotency via SHA256 hashing, activation/rollback with history tracking, and Redis locking for atomic operations. Implements state machine transitions (staged → active → archived) with complete audit logging.

10. **UserDefaultModelRepository** (`user_default_models.py`): Handles per-user default model preferences with tenant scoping, instance validation, and cascade cleanup on instance deletion. Supports resolution precedence: user+tenant specific → user global → tenant default → global default.

Repository Interface Patterns All repositories follow consistent architectural patterns:

- **Session Management**: Each method manages its own database session lifecycle with explicit commit/rollback handling and proper resource cleanup.
- **Cursor Pagination**: Keyset-based pagination using composite keys (created_at DESC, id ASC) to avoid OFFSET performance issues and provide stable ordering across requests.
- **ETag Computation**: Deterministic hash computation from entity identifiers and timestamps for HTTP conditional request support and cache validation.
- **Redis Caching**: Integration with Redis for performance optimization using short-lived caches (5–60 minutes) with automatic invalidation on mutations.
- **Idempotency**: Support for idempotent operations with conflict detection, race condition handling, and audit logging for all mutations.
- **Tenant Isolation**: Automatic tenant_id filtering in queries with ownership validation for security and multi-tenancy support.
- **Audit Trails**: Comprehensive audit event logging for all mutations with actor tracking, trace IDs, and payload snapshots for compliance.

The repository pattern provides type-safe, maintainable data access with clear separation of concerns, enabling the platform to scale while maintaining data integrity and security.

4.4.6 Redis Data Plane

Redis serves as the platform's data plane, providing high-performance in-memory storage for caching, job queues, rate limiting, session state, and distributed coordination. The implementation leverages Redis's rich data structures (STRINGS, HASHes, LISTs, ZSETs) to optimize different operational patterns, with comprehensive caching strategies reducing database load and improving response times. The platform implements sophisticated job queue management, sliding window rate limiting algorithms, and distributed locking mechanisms, all with graceful degradation to in-memory fallbacks when Redis is unavailable.

Redis Usage Categories

Redis serves as a critical data plane component, providing high-performance caching, job storage, rate limiting, and distributed coordination. The platform leverages Redis across 10+ distinct usage categories, each optimized for specific operational requirements.

Job Queues Redis implements job queue management using LIST data structures with BRPOP (blocking right-pop) for efficient worker consumption. The implementation supports priority-based job ordering, dead letter queue handling for failed jobs, and queue depth monitoring. Job documents are stored as Redis HASHes with TTL-based auto-expiry, while ZSET indexes enable efficient querying by owner, status, and creation time.

Session State Agent session context is cached in Redis HASHes with 1-hour TTL, storing session configuration, message history, and execution state. The implementation provides heartbeat tracking to refresh TTL on activity, enabling efficient session state management across distributed workers without requiring database queries for every operation.

Rate Limiting The platform implements a sophisticated sliding window rate limiting algorithm using Redis ZSETs (sorted sets) to track request timestamps. The algorithm automatically cleans up old entries, supports per-user and per-tenant quotas with configurable window sizes, and provides graceful fallback to in-memory storage when Redis is unavailable. Production limits include 10 sessions:create per user per minute, 100 steps:create per user per minute, and tenant-level quotas of 1000 sessions per hour.

Idempotency Keys Request deduplication is implemented using SETNX (set if not exists) operations with configurable expiration windows (default 24 hours). The system generates deterministic idempotency keys from request fingerprints (owner, tenant, type, payload hash) and caches responses for replay, ensuring safe retry semantics for POST operations.

Circuit Breaker State Provider health status is cached in Redis with 1-hour TTL, storing failure counts per provider and state transitions (CLOSED/OPEN/HALF_OPEN). This enables fast health checks without querying the database and supports circuit breaker pattern implementation for LLM provider resilience.

LLM Response Caching Prompt-response pairs are cached using content-addressed keys (SHA256 hash of prompt) with configurable TTL. This reduces redundant LLM API calls and improves response times for repeated queries while maintaining cache freshness through TTL expiration.

JWKS Key Caching Public keys from OIDC providers are cached in Redis with TTL-based expiration, enabling fast JWT validation without repeated network calls to the identity provider. The cache automatically refreshes on key rotation, ensuring security while maintaining performance.

Health Check Results Component health status is cached with short TTL (typically 5–60 seconds) for freshness, aggregating health state from multiple services (PostgreSQL, Redis, Memgraph, external APIs). This enables fast health endpoint responses while minimizing load on underlying systems.

Worker Registration Active worker tracking uses Redis to maintain a registry of processing workers with heartbeat timestamps and capability metadata. This supports distributed job processing, worker health monitoring, and dynamic worker discovery in multi-instance deployments.

Distributed Locks The platform implements distributed locking using SETNX with automatic expiration (5–10 seconds) for session and step-level exclusive access. This prevents concurrent modifications in distributed environments, ensuring data consistency for agent session updates and step sequencing.

Implementation Characteristics All Redis operations implement graceful degradation with in-memory fallbacks when Redis is unavailable, ensuring platform resilience. The implementation uses connection pooling for async operations (max 10 connections), pipelines for multi-operation sequences, and Lua scripts for atomic operations. Maintenance tasks automatically clean orphaned index members and ensure data consistency across all usage categories.

4.4.7 Memgraph Graph Domain

The Memgraph graph domain module provides the graph database infrastructure for the platform, using Memgraph as the graph database backend. The implementation includes configuration management, client utilities, data population scripts, and integration with the Natural Language to Cypher pipeline.

Graph Schema The graph models bioinformatics workflows at CINECA, tracking users, their institutions, computational tasks, and file artifacts. The schema consists of four primary node types:

- **User**: Platform users with properties including `user_id`, `firstName`, `lastName`, `user_name`, and `email`.
- **Institution**: Organizations or companies with a `name` property.
- **Task Nodes**: Multiple task types including `SearchbyTaxon`, `Bold`, `Blast`, `BlastSeq`, `CreateDb`, and `Command`, each with task-specific properties such as `task_id`, `status`, `tool`, and output references.
- **File Nodes**: Various file types including `File`, `Fasta`, `BlastDb`, `BlastedSeq`, `Xml`, and `PhyloTree`, with properties like `file_id`, `user_filename`, `size`, and `extension`.

Relationship Types The graph defines three primary relationship types:

- **WORKS_AT**: `(User)-[:WORKS_AT]->(Institution)` represents user employment relationships.
- **RUNS**: `(User)-[:RUNS]->(Task)` indicates that a user executes a computational task.

- **INPUT/OUTPUT**: `(File)-[:INPUT]->(Task)` and `(Task)-[:OUTPUT]->(File)` model data flow, where files serve as inputs to tasks and tasks produce output files.

Implementation Architecture The module provides a factory function `get_memgraph()` that returns a configured Memgraph client instance using the gqlalchemy library. Configuration is managed through Pydantic Settings with environment variable support, enabling flexible deployment across development, staging, and production environments.

Data Population The platform includes comprehensive data population capabilities with two approaches:

1. **Original Dataset Loader**: Imports reference datasets from JSON/CSV files, creating indexes based on schema hints and preserving original identifiers for traceability.
2. **Synthetic Data Generator**: Generates realistic bioinformatics workflow data using the Faker library, with configurable parameters for users, institutions, and tasks. The generator produces schema-compliant data with reproducible results through fixed random seeds.

NL-to-Cypher Pipeline Integration The Memgraph domain integrates with the Natural Language to Cypher translation pipeline, enabling users to query the graph using natural language questions. The pipeline translates questions into Cypher queries, validates them for safety, and executes them against the Memgraph database, returning structured results that can be consumed by AI agents or displayed in user interfaces.

Client Implementation The Memgraph client factory (`memgraph_client.py`) provides connection management with optional authentication support. The client uses the Bolt protocol (default port 7687) and integrates seamlessly with the FastAPI application through dependency injection, enabling graph queries from API endpoints and MCP tools.

4.5 Presentation Layer

The Presentation Layer provides two specialized user interfaces optimized for distinct personas and use cases: a **Next.js Agent Chat UI** for end-user conversational interactions, and a **Streamlit Control Panel** for administrative operations and system monitoring. This dual-UI strategy enables persona-specific optimization while sharing the common backend API.

4.5.1 Agent Chat UI

The Agent Chat UI is a modern, production-grade web application built with Next.js 14, providing a Copilot-style conversational interface for interacting with AI agents.

Technology Stack Table 4.13 summarizes the technologies used in the Agent Chat UI.

Table 4.13. Agent Chat UI technology stack.

Component	Technology	Version
Framework	Next.js (App Router)	14.2.15
UI Library	React	18.3.1
State Management	Zustand	4.5.5
UI Components	Radix UI	Latest
Styling	Tailwind CSS	3.4.14
Icons	Lucide React	Latest
Type Safety	TypeScript	5.x

Component Architecture The UI follows a component-based architecture with clear separation of concerns:

```
ui_agent/src/
|-- app/
|   |-- page.tsx          # Main chat page with layout
|   |-- layout.tsx        # Root layout with providers
|   |-- globals.css       # Global styles (Tailwind)
|   `-- api/              # API routes (auth token proxy)
|-- components/
|   |-- chat-area.tsx     # Message display + step rendering
|   |-- chat-input.tsx    # Prompt input + model selector
|   |-- role-toggle.tsx   # Admin/User role switch
|   `-- ui/               # Radix UI primitives
|-- stores/
|   |-- auth-store.ts     # Authentication state (Zustand + persist)
|   `-- chat-store.ts     # Chat messages and runs (Zustand)
`-- lib/
    |-- api.ts            # Backend API client
    `-- utils.ts          # Utility functions
```

Listing 4.33. Agent Chat UI directory structure.

State Management The application uses Zustand for lightweight, performant state management with two primary stores:

Auth Store Manages role selection (Admin/User), token generation via Auth0, and SSR-safe hydration. Role selection persists in localStorage, but tokens are regenerated on each session for security.

Chat Store Manages chat messages, agent runs with steps and status, model selection, and auto-scroll behavior. Messages are stored as a flat array with run references for efficient rendering.

```
interface ChatState {
  messages: ChatMessage[];
  selectedModel: string;
  availableModels: Array<{ id: string; name: string }>;
  currentRunId: string | null;
  isSubmitting: boolean;
  isPolling: boolean;

  addUserMessage: (content: string) => string;
  addAgentResponse: (messageId: string, run: AgentRun) => void;
  updateAgentRun: (messageId: string, run: AgentRun) => void;
}
```

Listing 4.34. Zustand store interface pattern.

Backend API Interaction The API client in `lib/api.ts` provides typed functions for all backend communication:

Table 4.14. Agent Chat UI API endpoints.

Endpoint	Function	Purpose
`POST /v1/agent-runs`	`createAgentRun()`	Create new agent run
`GET /v1/agent-runs/{id}`	`getAgentRun()`	Poll run status
`GET /v1/agent-runs/{id}/steps`	`getAgentRunSteps()`	Get execution steps
`GET /v1/models/instances`	`listModels()`	List available models
`GET /v1/models/defaults`	`getDefaultModel()`	Get default model
`GET /v1/auth/me`	`getAuthMe()`	Validate token

Polling Pattern The chat UI implements a time-based polling pattern for agent run completion:

```
async function pollRunUntilComplete(
  runId: string,
  token: string | null,
  onUpdate: (run: AgentRun) => void,
  intervalMs: number = 2000,    // 2 second intervals
  maxAttempts: number = 300     // 10 minutes max
): Promise<AgentRun> {
  let attempts = 0;

  while (attempts < maxAttempts) {
    const run = await getAgentRun(runId, token);
    onUpdate(run);  // Update UI with latest state

    // Terminal states
    if (['succeeded', 'failed', 'cancelled'].includes(run.status)) {
      return run;
    }

    await new Promise(resolve => setTimeout(resolve, intervalMs));
    attempts++;
  }

  throw new Error(`Polling timed out after ${maxAttempts} attempts`);
}
```

Listing 4.35. Run completion polling implementation.

Model Selection Flow The chat input component handles dynamic model selection:

1. On role change, fetch available models from `/v1/models/instances`
2. Query `/v1/models/defaults` for backend-configured default
3. If backend default exists, use it; otherwise use first available model
4. User can override via dropdown selection

Message Display Features The chat area component renders messages with rich step visualization:

- **User messages**: Simple text bubbles with timestamp
- **Agent responses**: Full run timeline including:
  - Status indicators (loading spinner, success checkmark, error icon)
  - Step-by-step execution display with action types
  - Tool calls with expandable inputs/outputs
  - Cypher query syntax highlighting
  - Collapsible JSON output drawers
  - Execution metrics (latency, token count)

4.5.2 Control Panel UI

The Control Panel UI is a comprehensive Streamlit-based administration interface providing full API coverage with role-aware access control.

Technology Stack

- **Framework**: Streamlit (Python web apps)
- **Authentication**: Auth0 OAuth2/OIDC
- **HTTP Client**: httpx (async-capable)
- **Data Display**: Pandas DataFrames
- **Visualization**: Plotly (via Streamlit)

```
ui_control_panel/
|-- app.py              # Main Streamlit entry point
|-- api.py              # API client with auth handling
|-- state.py            # Session state management
|-- components.py       # Reusable UI components
`-- views/
    |-- auth.py         # Authentication tab
    |-- dashboard.py    # Health monitoring
    |-- agents.py       # Agent runs & sessions
    |-- tools.py        # Tool discovery & invocation
    |-- models.py       # Model management
    |-- jobs.py         # Job management
    |-- tenants.py      # Multi-tenancy CRUD
    |-- admin.py        # Admin operations
    |-- cypher.py       # NL-to-Cypher workflow
    `-- explore.py      # API explorer
```

Listing 4.36. Control Panel directory structure.

Authentication System The Control Panel supports four identity types with Auth0 integration:

Table 4.15. Control Panel identity types.

Identity Type	Auth Method	Use Case
Admin	Password Realm	Full platform administration
User	Password Realm	Normal user operations
Machine	Client Credentials	Service-to-service
Custom	Manual token	Testing/debugging

Scope-Based Access Control Features are enabled based on OAuth2 scopes in the active token:

Table 4.16. Control Panel feature scopes.

Feature	Required Scopes
Dashboard	None (public health endpoints)
Agents	`user:me`
Tools (Safe)	`tools:invoke:basic`
Tools (All)	`tools:invoke:all`
Models (Create)	`admin:all`
Tenants	`admin:all`
Admin Operations	`admin:all`

Tab-Based Navigation The Control Panel uses Streamlit's tab system to organize functionality:

1. **Auth**: Role selection, token management, scope inspection
2. **Dashboard**: System health, component status, KPI metrics
3. **Agents**: Run creation, session management, step visualization
4. **Tools**: Tool discovery, schema inspection, invocation testing
5. **Models**: Provider management, instance configuration, defaults

6. **Jobs**: Job creation, status monitoring, event streaming
7. **Tenants**: Tenant CRUD, configuration management
8. **Admin**: Database operations, system configuration
9. **Cypher**: NL-to-Cypher testing, query execution
10. **Explore**: OpenAPI specification, raw API requests

API Client Architecture The API client provides robust backend communication:

```python
def normalize_endpoint(endpoint: str) -> str:
    """Ensure all endpoints start with /v1/"""
    if not endpoint.startswith('/'):
        endpoint = '/' + endpoint
    if not endpoint.startswith('/v1'):
        endpoint = '/v1' + endpoint
    return endpoint

def get_headers() -> dict:
    """Build request headers with auth and tenant context"""
    headers = {"Content-Type": "application/json"}
    token = get_active_token()
    if token:
        headers["Authorization"] = f"Bearer {token.value}"
    tenant_id = get_state().tenant_id
    if tenant_id:
        headers["X-Tenant-ID"] = tenant_id
    return headers
```

Listing 4.37. Control Panel API client pattern.

Dashboard Health Monitoring The dashboard tab monitors system health via multiple endpoints:

Table 4.17. Dashboard health endpoints.

Endpoint	Purpose	Display
`/health/live`	Liveness probe	Status card
`/health/ready`	Readiness probe	Status card
`/health/startup`	Startup probe	Status card
`/health/components`	Component health	Grid of cards

Agent Execution Interface The agents tab provides a complete agent execution interface:

- Run creator form with prompt, model selection, temperature, and max steps
- Session management for conversation continuity
- Real-time progress monitoring with step-by-step timeline

- Result display with final answer and execution metrics
- Polling with jitter to prevent thundering herd effects

NL-to-Cypher Testing The Cypher tab enables natural language to Cypher workflow testing:

1. Natural language query input with example prompts
2. Graph schema exploration and visualization
3. Generated Cypher display with syntax highlighting
4. Query execution with result rendering
5. Query history for comparison and iteration

4.5.3 Dual-UI Architecture Analysis

The decision to implement two separate user interfaces—rather than a single unified application—reflects a deliberate architectural choice based on persona analysis and technology fit.

Advantages of Dual-UI Approach

Persona Separation The Chat UI serves end-users requiring conversational AI interaction, while the Control Panel serves administrators and operators requiring data exploration and system management. Each UI optimizes for its target persona without compromises.

Technology Specialization Next.js excels at real-time, interactive experiences with sub-second feedback, while Streamlit excels at rapid development of data-driven dashboards. Using each framework for its strengths yields better user experiences than a single compromise solution.

Parallel Development Separate codebases enable independent development by different team members or skill sets. Frontend specialists can focus on the Chat UI while backend developers contribute to the Control Panel.

Risk Isolation Bugs or outages in one UI do not affect the other. The Chat UI can remain operational during Control Panel maintenance and vice versa.

Deployment Flexibility Each UI can be deployed, scaled, and updated independently. The Chat UI can be served as static assets via CDN, while the Control Panel runs as a Python process.

Potential Limitations

Code Duplication Both UIs implement their own API clients, authentication handling, and error formatting. Changes to API contracts require updates in both codebases.

State Inconsistency Session state and user preferences are managed independently in each UI. A user switching between UIs may encounter inconsistent defaults or cached data.

Cognitive Overhead Users who require both interfaces (e.g., developers testing agents and monitoring jobs) must context-switch between different visual languages and interaction patterns.

Testing Complexity End-to-end tests must cover both interfaces, potentially with different testing frameworks (Playwright for Next.js, Selenium or direct Python tests for Streamlit).

Maintenance Burden Feature additions (e.g., a new model field) may require changes in both UIs, increasing total development effort.

Mitigations Implemented

- **Shared Backend API**: The REST API serves as the single source of truth, ensuring both UIs reflect consistent data.
- **Consistent Model Selection**: Both UIs query the same `/v1/models/defaults` endpoint, ensuring default model consistency.
- **Common Health Endpoints**: Both UIs display health status from the same `/health/*` endpoints.
- **Documentation**: Clear documentation indicates when to use each UI based on task type.

Comparison Summary Table 4.18 summarizes the key differences between the two interfaces.

Table 4.18. Comparison of Chat UI and Control Panel characteristics.

Aspect	Agent Chat UI	Control Panel
Primary Persona	End users, researchers	Administrators, operators
Interaction Style	Conversational, real-time	Dashboard, form-driven
Technology	Next.js, React, TypeScript	Streamlit, Python
State Management	Zustand (client-side)	Session state (server-side)
Deployment	Static/SSR, CDN-cacheable	Python process, stateful
Update Latency	Sub-second (polling)	Page refresh
Development Speed	Moderate (typed, compiled)	Fast (Python, hot reload)

4.5.4 Developer Experience and API Consumer Patterns

Beyond the two provided UIs, the platform API is designed for consumption by external clients, scripts, and integrations. This subsection describes patterns for effective API usage.

RESTful Conventions The API follows standard REST conventions:

- **Resource-Oriented URLs**: `/v1/agents`, `/v1/jobs/{id}`
- **HTTP Methods**: GET (read), POST (create), PUT/PATCH (update), DELETE (remove)
- **Status Codes**: 200 (success), 201 (created), 204 (no content), 4xx (client error), 5xx (server error)
- **Content Types**: `application/json` for requests/responses, `application/problem+json` for errors

Authentication Pattern API clients authenticate using JWT bearer tokens:

```bash
# Obtain token from Auth0
TOKEN=$(curl -s https://${AUTH0_DOMAIN}/oauth/token \
  -d "grant_type=client_credentials" \
  -d "client_id=${CLIENT_ID}" \
  -d "client_secret=${CLIENT_SECRET}" \
  -d "audience=${API_AUDIENCE}" | jq -r .access_token)

# Make authenticated request
curl -H "Authorization: Bearer ${TOKEN}" \
     -H "X-Tenant-ID: ${TENANT_ID}" \
     https://api.example.com/v1/agents
```

Listing 4.38. API authentication example.

Error Handling Clients should handle RFC 7807 Problem Detail responses:

```python
import httpx

def create_agent_run(prompt: str, token: str) -> dict:
    response = httpx.post(
        f"{API_BASE}/v1/agent-runs",
        headers={"Authorization": f"Bearer {token}"},
        json={"prompt": prompt},
    )

    if response.status_code >= 400:
        problem = response.json()
        raise APIError(
            status=problem["status"],
            title=problem["title"],
            detail=problem.get("detail"),
        )

    return response.json()
```

Listing 4.39. Python client error handling.

Idempotent Requests For safe retries, clients should use idempotency keys:

```python
import uuid

def create_job_idempotent(job_type: str, payload: dict, token: str)
    -> dict:
    idempotency_key = str(uuid.uuid4())

    response = httpx.post(
        f"{API_BASE}/v1/jobs",
        headers={
            "Authorization": f"Bearer {token}",
            "Idempotency-Key": idempotency_key,
        },
        json={"type": job_type, "payload": payload},
    )

    # Safe to retry with same idempotency_key
    return response.json()
```

Listing 4.40. Idempotent request pattern.

Pagination Handling Clients should use cursor-based pagination for list endpoints:

```python
def list_all_runs(token: str) -> list[dict]:
    all_runs = []
    cursor = None

    while True:
        params = {"limit": 100}
        if cursor:
            params["cursor"] = cursor

        response = httpx.get(
            f"{API_BASE}/v1/agent-runs",
            headers={"Authorization": f"Bearer {token}"},
            params=params,
        )
        data = response.json()
        all_runs.extend(data["items"])

        if not data.get("has_more"):
            break
        cursor = data["next_cursor"]

    return all_runs
```

Listing 4.41. Cursor-based pagination example.

OpenAPI Documentation The API provides OpenAPI 3.1 documentation at `/v1/openapi.json` (or `api/openapi.json` in the repository):

- Full endpoint documentation with request/response schemas
- Authentication requirements per endpoint
- Example requests and responses
- Interactive exploration via Swagger UI at `/docs`

SDK Generation The OpenAPI specification enables automatic SDK generation for various languages:

```
# Generate Python client
openapi-generator-cli generate \
  -i https://api.example.com/v1/openapi.json \
  -g python \
  -o ./sdk/python

# Generate TypeScript client
openapi-generator-cli generate \
  -i https://api.example.com/v1/openapi.json \
  -g typescript-fetch \
  -o ./sdk/typescript
```

Listing 4.42. SDK generation example.

4.5.5 Control Panel UI

The Control Panel UI is a comprehensive Streamlit application designed for operators and administrators, providing full coverage of the platform API with role-aware access control and a polished user experience optimized for data exploration and system management.

Technology Stack The application is built with Streamlit (version $\geq$ 1.30.0) and Python 3.11+, leveraging Streamlit's rapid development capabilities and Python integration for seamless backend API interaction. The UI uses Pandas for data manipulation and display, enabling efficient handling of tabular data and exports.

Core Architecture The application follows a modular architecture with clear separation between state management, API communication, and view rendering:

- **State Management** (`state.py`): Centralized session state management using Streamlit's native session state with initialization, update helpers, and state persistence across page reloads.
- **API Client** (`api.py`): HTTP client wrapper with automatic retry logic, error handling, and token management. Implements request/response logging with token masking for security and provides convenience methods for all API endpoints.

- **Components** (`components/`): Reusable UI components including health cards, pagination controls, JSON drawers, timeline visualizations, tenant selectors, and error display components.
- **Views** (`views/`): Tab-specific view modules that render complete functional areas of the application.

Functional Views The Control Panel provides comprehensive coverage through specialized tabs:

- **Dashboard**: Real-time health monitoring for all system components (PostgreSQL, Redis, Memgraph, external APIs) with KPI cards, health status indicators, and trend visualization. Displays aggregated system metrics and component availability.
- **Agents**: Copilot-style agent run interface with live timeline visualization of tool calls, step-by-step execution tracking, and metrics display. Supports searching runs, sessions, and steps with filtering and pagination.
- **Jobs**: User and admin job management with event streaming, status monitoring, and event log viewing. Provides job creation, cancellation, and result inspection capabilities with real-time updates via polling.
- **Models & Providers**: Model instance and provider management with health checks, default model configuration, and provider registration. Supports creating, updating, and deleting model instances with validation and audit trails.
- **Tools**: Tool discovery with schema inspection, tool invocation with parameter validation, and result display. Includes support for Natural Language to Cypher tool with query generation and execution visualization.
- **Tenants**: Full CRUD operations for multi-tenancy with tenant creation, metadata management, and dependency checking before deletion. Provides tenant selection and filtering capabilities.
- **Admin**: Administrative operations including process management, manifest staging and activation, database operations, and system configuration. Restricted to users with `admin:all` scope.
- **Cypher/Graph**: Natural Language to Cypher experimentation interface with query generation, validation, and execution. Displays generated Cypher queries, execution results, and query performance metrics.

User Experience Features The Control Panel implements sophisticated UX patterns:

- **Paginated Tables**: Efficient handling of large datasets with cursor-based pagination, sorting, and filtering. Tables support CSV and JSON export for data analysis.

- **JSON Drawers**: Expandable JSON viewers with sanitized display (token masking, secret redaction) for inspecting API responses, job payloads, and tool results.
- **Visual Timelines**: Timeline components for visualizing agent execution steps, job event sequences, and audit trails with chronological ordering and status indicators.
- **Error Tracking**: Comprehensive error display with trace IDs, error messages, and retry suggestions. Errors are logged with masked credentials for security.
- **Live Updates**: Polling mechanisms for long-running operations (jobs, agent runs) with configurable refresh intervals and automatic status updates.

Authentication and Authorization The UI supports four identity types (Admin, User, Machine, auto-managed) with Auth0 integration using Password Realm and Client Credentials grants. Token lifecycle is managed with auto-renewal, and scope-based UI elements show/hide features based on permissions. All tokens are masked in logs and UI displays for security.

Responsive Design The application includes responsive design considerations for tablets and mobile devices, with adaptive tab layouts, collapsible sections, and touch-optimized interactions. The layout adjusts gracefully from wide desktop views to narrow mobile viewports while maintaining functionality.

4.5.6 Dual-UI Architecture Analysis

The platform implements a dual-user interface strategy, providing two specialized interfaces optimized for distinct personas and use cases: a Next.js Agent Chat UI for end-user conversational interactions, and a Streamlit Control Panel for administrative operations and system monitoring. This architectural decision balances persona-specific optimization with shared backend infrastructure.

Advantages of Dual-UI Approach The dual-UI architecture provides several key benefits:

- **Persona Separation**: Clear separation between end-user chat interface and admin/operator dashboard enables each interface to be optimized for its specific user base without compromise. End users receive a focused, distraction-free conversational experience, while operators have comprehensive system visibility and management capabilities.
- **Technology Specialization**: Next.js excels at real-time, interactive user experiences with server-side rendering and client-side reactivity, making it ideal for conversational interfaces. Streamlit's Python-native approach and rapid development capabilities make it perfect for data exploration, administrative dashboards, and integration with Python-based tooling.

- **Parallel Development**: Separate codebases enable independent development teams to work in parallel without coordination overhead. Frontend teams can iterate on the Next.js UI while backend/DevOps teams enhance the Streamlit interface, reducing development bottlenecks.
- **Risk Isolation**: UI bugs, performance issues, or deployment problems in one interface do not affect the other. This isolation provides operational resilience and enables independent deployment cycles and rollback strategies.
- **Different Deployment Patterns**: The Next.js UI can be deployed as static assets to CDN or object storage, providing global distribution and edge caching. The Streamlit application runs as a Python process, enabling server-side computation and direct database access when needed.

Potential Limitations The dual-UI approach introduces some challenges:

- **Code Duplication**: Both UIs implement their own API clients, authentication logic, and state management patterns. This duplication increases maintenance effort and requires careful synchronization when API contracts change.
- **State Drift**: Session states, user preferences, and UI-specific caches can diverge between interfaces. Users switching between UIs may experience inconsistent behavior or need to re-authenticate.
- **Cognitive Overhead**: Users who need to use both interfaces must maintain context across different interaction paradigms, UI patterns, and navigation structures. This can increase training requirements and reduce efficiency for users who regularly switch between interfaces.
- **Testing Complexity**: End-to-end testing must cover both interfaces, effectively doubling test coverage requirements. Integration tests must verify that both UIs correctly interact with shared backend APIs and maintain consistency.
- **Maintenance Burden**: API changes, feature additions, and bug fixes may require updates in both codebases. Documentation, user guides, and training materials must cover both interfaces, increasing maintenance overhead.

Mitigations Implemented The platform addresses these limitations through several architectural decisions:

- **Shared Backend REST API**: Both UIs consume the same RESTful API, ensuring a single source of truth for business logic, data validation, and security policies. API changes are reflected consistently across both interfaces.
- **Consistent Model Selection**: Model selection logic is centralized in the backend, with both UIs displaying the same available models and respecting the same tenant defaults and user preferences. This ensures consistent behavior regardless of interface.

- **Common Health/Status Endpoints**: Health checks, system status, and monitoring endpoints are shared, providing consistent visibility across both interfaces. Operators can monitor system health from either interface with identical information.
- **Unified Authentication**: Both UIs use the same OIDC/OAuth2 authentication flow and token management, ensuring consistent security policies and user identity across interfaces.
- **API Documentation**: Comprehensive OpenAPI specification enables both UI teams to stay synchronized on API contracts, reducing integration issues and enabling automated API client generation.

Architectural Trade-offs The dual-UI architecture represents a deliberate trade-off between specialization and unification. While a single unified interface could reduce duplication and maintenance overhead, it would require compromises in user experience optimization and technology choices. The platform's approach prioritizes user experience and operational flexibility over code consolidation, aligning with enterprise requirements for persona-specific optimization and independent deployment cycles.

The architecture successfully balances these concerns by maintaining strict API contracts, shared authentication, and consistent data models while allowing each interface to optimize for its specific use case and user base.

4.5.7 Developer Experience and API Consumer Patterns

The platform provides a comprehensive RESTful API designed for external client consumption, with well-documented patterns for authentication, error handling, pagination, and idempotency. The API follows industry-standard practices to ensure reliable integration and optimal developer experience.

RESTful API Design The platform exposes a RESTful API with versioned endpoints under the `/v1/` prefix, ensuring stability and backward compatibility. The API follows resource-oriented design principles, using nouns in paths (e.g., `/v1/agents/sessions`, `/v1/jobs`, `/v1/tools`) and standard HTTP methods (GET for retrieval, POST for creation, PATCH for updates, DELETE for deletion). All endpoints are documented through OpenAPI 3.1 specification, enabling automatic SDK generation and interactive exploration via Swagger UI at `/docs`.

Authentication Patterns All API endpoints require HTTP Bearer token authentication using OAuth2/OIDC tokens. Clients obtain tokens through Auth0 integration using Password Realm or Client Credentials grants, depending on the use case (user authentication vs. machine-to-machine communication). The platform implements scope-based authorization, where tokens contain scopes (e.g., `user:me`, `admin:all`) that determine access permissions. The API validates tokens on every request, extracting user identity and scopes from JWT claims, and enforces permission checks at the endpoint level.

Error Handling and RFC 7807 Problem Details The platform implements RFC 7807 Problem Details for HTTP APIs, providing structured error responses that include type, title, status, and detail fields. This enables clients to programmatically handle errors with appropriate user-facing messages. Error responses include correlation IDs (`X-Request-Id`) for debugging and traceability. The API uses appropriate HTTP status codes: 400 for bad requests, 401 for authentication failures, 403 for authorization failures, 404 for not found, and 500 for internal server errors.

Idempotency for Safe Retries To handle network failures and enable safe retries, the platform supports idempotency keys via the `Idempotency-Key` header on POST requests. Clients generate unique keys (typically UUIDs) for each request, and the platform caches responses for 24 hours. Retrying a request with the same idempotency key returns the cached response (200 OK) instead of creating duplicates (201 Created). This pattern is supported on mutation endpoints including `POST /v1/agents/sessions`, `POST /v1/agents/sessions/{session_id}/steps`, and `POST /v1/agent-runs`.

ETag Caching for Efficiency The platform implements HTTP ETag caching to reduce bandwidth usage for polling scenarios. GET endpoints return ETag headers computed from response content, and clients can include `If-None-Match` headers in subsequent requests. When data is unchanged, the API returns 304 Not Modified with no response body, significantly reducing bandwidth for large list endpoints. ETags are supported on `GET /v1/agents/sessions`, `GET /v1/agents/sessions/{session_id}`, `GET /v1/agents/sessions/{session_id}/steps`, and `GET /v1/agent-runs/{run_id}`.

Cursor-Based Pagination List endpoints implement cursor-based pagination to handle large datasets efficiently. Clients specify `limit` (default 20, max 100) and `cursor` parameters, receiving responses with `items`, `has_more`, and `next_cursor` fields. This approach avoids OFFSET performance issues and provides stable ordering across requests. Pagination is implemented using composite keys (created_at DESC, id ASC) for deterministic cursor encoding and decoding.

Rate Limiting Awareness The API implements rate limiting with per-user and per-tenant quotas, returning `X-RateLimit-Limit`, `X-RateLimit-Remaining`, and `X-RateLimit-Reset` headers. Clients should monitor these headers and implement exponential backoff when rate limits are exceeded. Production limits include 10 sessions:create per user per minute and tenant-level quotas of 1000 sessions per hour.

Developer Workflows The platform provides comprehensive documentation to guide developer workflows, including a project map in the README that identifies key entry points (app.py, orchestrator.py, routers/, security/, etc.), API best practices guide covering authentication, idempotency, caching, and pagination patterns, and OpenAPI specification enabling automatic SDK generation for Python, TypeScript, and other languages. The documentation emphasizes common workflows such as creating agent runs with polling, managing sessions with step-by-step execution, and handling long-running jobs with Server-Sent Events for progress streaming.

Chapter 5

Agent Orchestration Engine

This chapter details the core agent orchestration engine, the central component of the Cineca Agentic Platform that coordinates LLM reasoning, tool invocation, and state management to execute agent runs. Building upon the architectural foundations presented in Chapter 4, this chapter explains the orchestration design principles, intent classification pipeline for routing requests, the agent run lifecycle and state machine, the Natural Language to Cypher pipeline for graph querying, and LLM provider management with resilience mechanisms. The orchestration engine implements the functional requirements for agent execution established in Chapter 3.

5.1 Orchestration Design

The Orchestration Engine forms the computational heart of the Cineca Agentic Platform, coordinating the complex interplay between user requests, language model reasoning, tool invocations, and data retrieval operations. This section presents the architectural design of the orchestrator, examining its responsibilities, execution model, and the design trade-offs that shaped its implementation.

5.1.1 Orchestrator Architecture and Responsibilities

The `Orchestrator` class, implemented in `src/services/orchestrator.py`, serves as the central coordination point for all agent execution. At 8,263 lines of code, it represents the most substantial component in the platform, encapsulating the following core responsibilities:

1. **LLM Call Coordination**: Managing interactions with language models for planning, reasoning, and response generation, including model selection, prompt construction, and response parsing.

2. **Tool Invocation**: Orchestrating the execution of MCP-style tools, handling input validation, output processing, and error recovery.

3. **Cache Integration**: Optionally leveraging Redis caching for response deduplication and performance optimization.

4. **Graph Access**: Coordinating queries against the Memgraph database for knowledge graph operations.

5. **Security Enforcement**: Integrating with the security framework for RBAC checks, tenant isolation, and audit logging.

6. **Metrics and Observability**: Recording execution metrics, latencies, and token usage for monitoring and cost tracking.

The orchestrator is deliberately designed to be *dependency-tolerant*: every external integration (LLM clients, cache, database, audit) is optional and detected at runtime. This design enables graceful degradation when components are unavailable and simplifies testing through selective mocking.

```python
class Orchestrator:
    def __init__(
        self,
        llm: Any | None = None,
        llm_clients: MutableMapping[str, Any] | None = None,
        db: Any | None = None,
        cache: Any | None = None,
        audit: Any | None = None,
        tools: MutableMapping[str, ToolFunc] | None = None,
        default_model: str | None = None,
        llm_device: str = "cpu",
        llm_max_tokens: int = 2048,
        llm_max_steps: int = 10,
    ) -> None:
        # All dependencies are optional
        self.llm = llm
        self.llm_clients = llm_clients or {}
        self.db = db
        self.cache = cache
        self.audit = audit
        self.tools = tools or {}
        # ...
```

Listing 5.1. Orchestrator class initialization showing dependency tolerance

5.1.2 Execution Flow Overview

The orchestrator processes agent runs through a structured pipeline that transforms user prompts into actionable outputs. Figure 5.1 illustrates the high-level execution flow.

The execution flow proceeds through the following stages:

1. **Request Reception**: The HTTP layer receives the user prompt along with session context, authentication information, and execution parameters.

2. **Intent Classification**: The prompt is analyzed to determine the appropriate processing mode (CHAT, GRAPH, ADMIN, SECURITY, or DANGEROUS).

3. **Mode-Specific Routing**: Based on the classified intent, the orchestrator routes to specialized handlers:

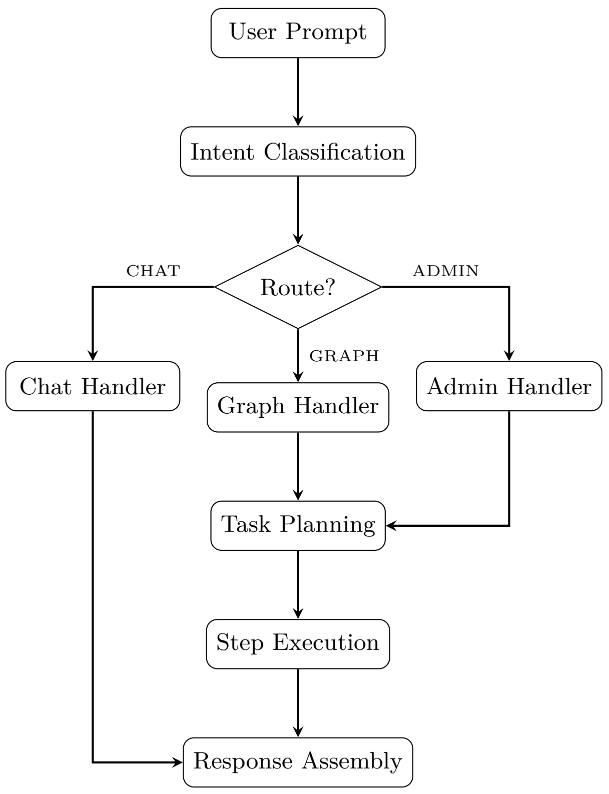

Figure 5.1. High-level orchestration execution flow

- `_handle_chat()`: Direct LLM conversation without tool invocation
- `_handle_graph_query()`: Memgraph database operations via NL-to-Cypher
- `_handle_admin_mode()`: Administrative operations requiring elevated privileges
- `_handle_security_mode()`: Permission and access control queries
- `_handle_dangerous_mode()`: Potentially destructive operations with safety gates

4. **Task Planning**: For complex requests, the orchestrator generates a structured plan as a list of task items, each representing a discrete step to execute.
5. **Step Execution**: The orchestrator iterates through planned steps, invoking tools, making LLM calls, and accumulating results.
6. **Response Assembly**: Final outputs are aggregated, formatted, and returned to the caller with comprehensive metrics.

5.1.3 OrchestrationResult Data Structure

The orchestrator produces results encapsulated in the `OrchestrationResult` dataclass, which captures all execution metadata required for observability, debugging, and cost tracking:

```python
@dataclass
class OrchestrationResult:
    """Complete result of an orchestration run."""
    success: bool = True
    outputs: list[dict[str, Any]] = field(default_factory=list)
    steps: list[dict[str, Any]] = field(default_factory=list)
    todos: list[dict[str, Any]] = field(default_factory=list)
    errors: list[str] = field(default_factory=list)
    warnings: list[str] = field(default_factory=list)

    # Timing metrics
    overall_ms: int | None = None
    first_llm_call_ms: int | None = None

    # LLM usage tracking
    llm_metrics: list[dict[str, Any]] = field(default_factory=list)
    llm_call_count: int = 0
    total_llm_calls: int = 0
    total_input_tokens: int = 0
    total_output_tokens: int = 0

    # Tool usage tracking
    tool_calls: int = 0
    tool_errors: int = 0

    # Execution metadata
    current_stage: str | None = None
    metrics: dict[str, Any] = field(default_factory=dict)
    degraded: bool = False
    used_fallback: bool = False
```

Listing 5.2. OrchestrationResult dataclass structure

The `to_dict()` method aggregates outputs into a coherent response, extracting final output text from step results and computing summary metrics for API consumers.

5.1.4 Timeout Configuration and Budget Management

The orchestrator implements sophisticated timeout management to prevent runaway executions while accommodating the varying latency characteristics of different deployment environments. The `ComputeConfig` class provides device-aware defaults:

Table 5.1. Default timeout configuration by compute device

Device	Step Timeout (s)	Run Timeout (s)	Use Case
CPU	1200	3600	Development, CI
GPU (CUDA)	30	180	Production inference
MPS (Apple)	60	300	Development (macOS)

These timeouts are configurable via environment variables and are automatically adjusted based on detected hardware capabilities. The orchestrator tracks execution budget consumption, emitting warnings when approaching limits and enforcing hard cutoffs to prevent resource exhaustion.

5.1.5 Execution Modes and Run Types

The orchestrator supports multiple execution modes to accommodate different operational requirements:

Synchronous vs. Asynchronous Execution

Synchronous Mode The default execution mode processes requests synchronously within the HTTP request lifecycle. This mode is suitable for:

- Simple chat interactions with low latency requirements
- Graph queries expected to complete quickly
- Administrative operations requiring immediate feedback

Asynchronous Mode For long-running operations, the platform supports asynchronous execution via the background jobs framework. The orchestrator can be invoked from a worker process, with progress updates streamed via Server-Sent Events (SSE). This mode is appropriate for:

- Complex multi-step agent workflows
- Batch data processing operations
- Operations requiring external service calls with unpredictable latency

Test and Demo Modes

The orchestrator provides specialized modes for testing and demonstration:

Test Mode When the `MEMGRAPH_NL_TEST_MODE` environment variable is enabled, the NL-to-Cypher pipeline uses a deterministic prompt catalog to produce predictable Cypher queries, enabling reliable integration testing independent of LLM output variability.

Demo Mode A simplified execution path that bypasses complex planning for demonstration scenarios, using pre-configured responses to illustrate platform capabilities without incurring LLM costs.

Simple vs. Complex Request Handling

The orchestrator distinguishes between simple and complex requests to optimize execution:

Simple Chat Requests classified with high confidence as conversational (greetings, identity questions, pleasantries) bypass task planning entirely, routing directly to an LLM for response generation. This fast path minimizes latency for trivial interactions.

Simple Graph Queries Queries matching patterns in the prompt catalog with known Cypher templates can be executed directly without LLM-based generation, providing deterministic and efficient graph access.

Complex Workflows Requests requiring multi-step reasoning, tool invocation, or conditional logic proceed through the full planning and execution pipeline, with the orchestrator managing state across steps.

5.1.6 Design Trade-offs and Rationale

The orchestrator's design reflects several deliberate trade-offs:

Centralization vs. Modularity The decision to consolidate orchestration logic in a single large module prioritizes:

- **Execution coherence**: A single execution context simplifies state management and debugging
- **Performance**: Avoiding inter-module communication overhead
- **Atomicity**: Transactions and rollbacks are easier to manage

However, this creates challenges for testing individual components in isolation and increases cognitive load for developers navigating the codebase.

Dependency Tolerance vs. Feature Completeness The optional dependency model enables graceful degradation but requires extensive null-checking and fallback logic throughout the codebase. This trade-off prioritizes operational resilience over code simplicity.

Flexibility vs. Type Safety The extensive use of `dict[str, Any]` types provides flexibility for evolving data structures but sacrifices compile-time type checking. Pydantic schemas at API boundaries provide runtime validation to mitigate this risk.

5.1.7 Integration Points

The orchestrator integrates with the broader platform through well-defined interfaces:

- **Router Layer**: The `src/routers/agent\_runs.py` module exposes HTTP endpoints that instantiate and invoke the orchestrator
- **Security Framework**: The `src/security/perm.py` module provides principal context and permission checking
- **MCP Tools**: The `src/mcp/` package supplies registered tools accessible via the orchestrator's tool registry
- **Metrics**: The `src/observability/` package receives execution telemetry for Prometheus export

- **Background Jobs**: The `src/workers/jobs\_worker.py` invokes the orchestrator for asynchronous execution

This modular integration enables the orchestrator to function as a reusable component across different execution contexts while maintaining separation of concerns with the platform's other subsystems.

5.2 Intent Classification

Intent classification serves as the critical routing mechanism that determines how the orchestrator processes incoming user requests. By analyzing prompt content before execution begins, the system can optimize processing paths, apply appropriate security controls, and provide relevant responses without unnecessary computational overhead. This section examines the classification architecture, pattern definitions, and integration with the broader orchestration pipeline.

5.2.1 Intent Classification Architecture

The intent classification system, implemented in `src/services/intent\_classifier.py`, provides lightweight, heuristic-based classification of user prompts into operational modes. The design prioritizes speed and determinism, using fast regex and keyword matching with optional LLM-based fallback for ambiguous cases.

Intent Modes

The classifier recognizes five distinct intent modes, each triggering specialized handling in the orchestrator:

CHAT General conversation, greetings, meta-questions about the system, and identity queries. These requests bypass complex planning and tool invocation, routing directly to conversational LLM responses.

GRAPH Memgraph database queries and graph operations. Requests in this mode trigger the NL-to-Cypher pipeline for knowledge graph access.

SECURITY Permission, access control, and RBAC-related questions. The orchestrator responds with information about the caller's effective scopes and allowed operations.

ADMIN Administrative operations including write operations, schema changes, and index creation. These requests require elevated privileges and are subject to additional authorization checks.

DANGEROUS Heavy, destructive, or unbounded operations such as bulk deletions, graph wipes, and unlimited exports. These requests trigger safety gates and require explicit admin authorization.

```python
class IntentMode(str, Enum):
    """Enumeration of supported intent classification modes."""
    CHAT = "chat"
    GRAPH = "graph"
    SECURITY = "security"
    ADMIN = "admin"
    DANGEROUS = "dangerous"
```

Listing 5.3. IntentMode enumeration

Classification Priority

The classifier evaluates patterns in a strict priority order, ensuring that safety-critical modes are detected first:

1. **Catalog match**: Pre-matched prompts with known categories (highest confidence: 0.95)
2. **DANGEROUS patterns**: Always checked first for safety (confidence: 0.90)
3. **ADMIN patterns**: Write operations, schema changes (confidence: 0.85)
4. **SECURITY patterns**: Permission queries (confidence: 0.85)
5. **GRAPH patterns**: Database/Cypher indicators (confidence: 0.85)
6. **CHAT patterns**: Greetings, conversational (confidence: 0.95 for exact matches)
7. **Conversational signals**: Fallback detection (confidence: 0.85)
8. **LLM-based classification**: If enabled, for ambiguous cases
9. **Default**: CHAT with low confidence (0.60)

This ordering ensures that potentially harmful operations are identified before benign interpretations, implementing a fail-safe approach to request routing.

IntentClassification Result

The classifier returns a comprehensive result object capturing the classification decision and supporting metadata:

```python
@dataclass
class IntentClassification:
    """Result of intent classification."""
    mode: IntentModeType
    confidence: float  # 0.0 to 1.0
    reasoning: str     # Machine-readable explanation
    source: str        # patterns, catalog, llm, conversational,
   default
    matched_catalog_id: str | None = None
    matched_patterns: list[str] | None = None
    pattern_categories: list[str] | None = None
    principal_blocked: bool = False
    requires_admin: bool = False
    used_llm: bool = False
```

Listing 5.4. IntentClassification dataclass

The `confidence` score enables downstream components to make nuanced decisions—for example, routing low-confidence classifications to human review or requesting clarification from the user.

5.2.2 Pattern Categories

The classifier defines pattern categories for each intent mode, organized as `PatternCategory` dataclasses containing compiled regular expressions. This structure enables efficient matching and clear organization of detection rules.

CHAT Patterns

Chat patterns detect conversational interactions that do not require tool invocation:

- **Greetings**: Simple salutations ("hi", "hello", "good morning")
- **Compound Greetings**: Greetings followed by questions ("hi, how are you?")
- **Identity Questions**: Queries about the system ("who are you?", "what can you do?")
- **Pleasantries**: Social phrases ("thank you", "goodbye")
- **Simple Questions**: Basic help requests ("help me")
- **Meta-System**: Questions about platform capabilities

GRAPH Patterns

Graph patterns identify requests targeting the Memgraph database:

- **Node Labels**: Cypher label syntax (e.g., `:Blast`, `:File`)
- **Cypher Keywords**: Query language keywords (`MATCH`, `RETURN`, `WHERE`)
- **Graph Terminology**: Domain vocabulary ("nodes", "edges", "relationships")
- **Domain Labels**: Bioinformatics-specific labels (`Blast`, `BlastedSeq`)
- **Query Operations**: Natural language patterns ("count nodes", "show 10")
- **NL Queries**: Graph traversal phrases ("shortest path", "neighbors of")

ADMIN Patterns

Admin patterns detect operations requiring elevated privileges:

- **Schema Operations**: Index and constraint management (`CREATE INDEX`)
- **Property Operations**: Property modifications ("rename property")
- **Write Operations**: Data modification (`MERGE`, `SET`, `CREATE`)

DANGEROUS Patterns

Dangerous patterns identify potentially destructive operations:

- **Delete Operations**: Data removal (`DELETE`, `DETACH DELETE`)
- **Drop Operations**: Database/graph destruction (`DROP DATABASE`)
- **Bulk Operations**: Unbounded queries (“delete all nodes”, “without LIMIT”)
- **Export Operations**: Mass data extraction (“export entire database”)
- **Continuous Operations**: Infinite loops (“forever”, “every second”)

SECURITY Patterns

Security patterns detect permission and access control queries:

- **Permission Queries**: Questions about allowed operations (“my permissions”, “allowed to”)
- **Tenant Queries**: Organization context (“my organization”)
- **Danger Queries**: Questions about dangerous operations

5.2.3 Ambiguity Handling and Edge Cases

Intent classification faces inherent ambiguity when prompts contain signals for multiple modes or lack clear indicators. This subsection examines borderline cases and explains how the system handles ambiguity through confidence thresholds and fallback mechanisms.

Borderline CHAT vs GRAPH Prompts Some prompts contain elements that could indicate either conversational or graph query intent:

- **Example 1**: “Tell me about genes” — This could be a conversational request for general information (CHAT) or a request to query the graph database for gene nodes (GRAPH). The classifier evaluates graph-specific terminology (“genes” matches domain labels) and assigns GRAPH mode with confidence 0.85. If graph terminology is absent, it defaults to CHAT with lower confidence (0.60).
- **Example 2**: “What can you tell me about diabetes?” — Conversational phrasing (“what can you tell me”) suggests CHAT, but “diabetes” could refer to a Disease node in the graph. The classifier prioritizes conversational signals and assigns CHAT mode with confidence 0.85. If the user follows up with “show me diabetes nodes in the graph”, the subsequent request clearly triggers GRAPH mode.
- **Example 3**: “How many nodes are there?” — This prompt contains graph terminology (“nodes”) but lacks Cypher syntax. The classifier detects graph terminology and assigns GRAPH mode with confidence 0.85, routing to NL-to-Cypher pipeline which generates an appropriate Cypher query.

DANGEROUS Mode Triggers The classifier prioritizes safety by checking DANGEROUS patterns before other interpretations:

- **Example 1**: "Delete all test data" — Contains "delete all" which matches DANGEROUS patterns. Even if the prompt could be interpreted as a conversational request, the classifier assigns DANGEROUS mode with confidence 0.90, triggering safety gates and requiring admin authorization.
- **Example 2**: "Remove everything without LIMIT" — Explicitly contains "without LIMIT" which matches bulk operation patterns. The classifier assigns DANGEROUS mode, blocking execution unless the caller has admin privileges.
- **Example 3**: "Can I delete nodes?" — This is a permission query (SECURITY mode) rather than a deletion request. The classifier recognizes the interrogative form ("can I") and routes to SECURITY mode, responding with information about permissions rather than executing a deletion.

Confidence Thresholds as Design Trade-off The confidence threshold system balances between false positives (overly conservative routing) and false negatives (missing dangerous operations). The design choices reflect this trade-off:

- **High-confidence thresholds (0.90-0.95)**: Used for catalog matches and exact pattern matches where classification is unambiguous. These enable fast routing without additional validation.
- **Medium-confidence thresholds (0.85)**: Applied to pattern-based matches where context could alter interpretation. These trigger standard processing paths but allow downstream components to request clarification if needed.
- **Low-confidence threshold (0.60)**: Default fallback for ambiguous prompts. Low-confidence classifications can trigger LLM-based re-classification (if enabled) or user clarification requests.
- **Safety-first ordering**: DANGEROUS patterns are checked first regardless of confidence, ensuring potentially harmful operations are never missed due to ambiguity. This fail-safe approach prioritizes security over convenience.

The confidence threshold values are configurable but default to conservative settings that minimize false negatives for safety-critical modes (DANGEROUS, ADMIN) while allowing flexibility for conversational modes (CHAT, GRAPH).

5.2.4 Confidence Thresholds

The `IntentConfidenceThresholds` class centralizes confidence values used throughout the classification and routing logic:

Higher thresholds for graph and dangerous operations reflect the greater consequences of misclassification in these modes.

Table 5.2. Intent classification confidence thresholds

Category	Threshold	Value
Pattern-Based Matches		
Catalog match	CATALOG_MATCH	0.95
Exact pattern	PATTERN_EXACT	0.95
Strong pattern (dangerous)	PATTERN_STRONG	0.90
Good pattern (admin/security/graph)	PATTERN_GOOD	0.85
Conversational signal	CONVERSATIONAL	0.85
Routing Thresholds		
Chat routing minimum	CHAT_ROUTING	0.60
Security routing minimum	SECURITY_ROUTING	0.75
Admin routing minimum	ADMIN_ROUTING	0.70
Dangerous routing minimum	DANGEROUS_ROUTING	0.70
Graph routing minimum	GRAPH_ROUTING	0.80
Default fallback	DEFAULT_FALLBACK	0.50

5.2.5 Prompt Catalog and Pattern-Based Routing

The prompt catalog provides a pre-classified knowledge base of common prompts, enabling deterministic classification and execution for known query patterns.

Catalog Structure

The prompt catalog, stored as a JSON file at `tests/integration/resources/memgraph\_nl\_prompts.json`, contains entries with the following structure:

```
{
    "id": "p01",                        # Unique identifier
    "text": "How many :Blast nodes?",  # Prompt text
    "category": "read_only",            # Classification category
    "expected_cypher_contains": [       # Validation patterns
        "MATCH", ":Blast", "COUNT"
    ],
    "limit_hint": 10,                   # Suggested LIMIT value
    "random": false,                    # Random sampling flag
    "allowed_for_user": true,           # RBAC: user access
    "allowed_for_admin": true,          # RBAC: admin access
    "smoke": true                       # Include in smoke tests
}
```

Listing 5.5. Prompt catalog entry structure

Catalog Categories

The catalog organizes prompts into categories that map to intent modes:

Table 5.3. Catalog category to intent mode mapping

Catalog Category	Intent Mode
read_only	GRAPH
admin_write	ADMIN
dangerous	DANGEROUS
security	SECURITY
data_quality	GRAPH
chat	CHAT
meta	CHAT

5.2.6 LLM Fallback Classification

When pattern-based classification is inconclusive (no patterns match or confidence is low), the classifier can optionally invoke an LLM for semantic analysis. This capability is disabled by default (`INTENT_LLM_FALLBACK_ENABLED=False`) to ensure deterministic behavior and minimize latency.

When enabled, the LLM fallback:

1. Constructs a classification prompt with the user input and mode descriptions
2. Invokes the configured LLM with low temperature for consistency
3. Parses the response to extract mode and confidence
4. Returns classification with `source="llm"` and `used_llm=True`

This fallback mechanism provides flexibility for deployments where natural language understanding is prioritized over strict determinism.

5.3 Agent Run Lifecycle

The agent run lifecycle defines the progression of execution states from initial request to final response. Understanding this lifecycle is essential for debugging, monitoring, and extending the platform. This section traces the complete flow of an agent run, examines the state machine governing status transitions, and details the internal structure of runs, sessions, and steps.

5.3.1 State Machine Definition

Agent runs follow a deterministic state machine with five possible states:

pending Initial state upon run creation. The run has been accepted but execution has not yet begun. Runs may be cancelled from this state.

running Active execution state. The orchestrator is processing the request, invoking tools, and generating responses. Runs may transition to succeeded, failed, or cancelled.

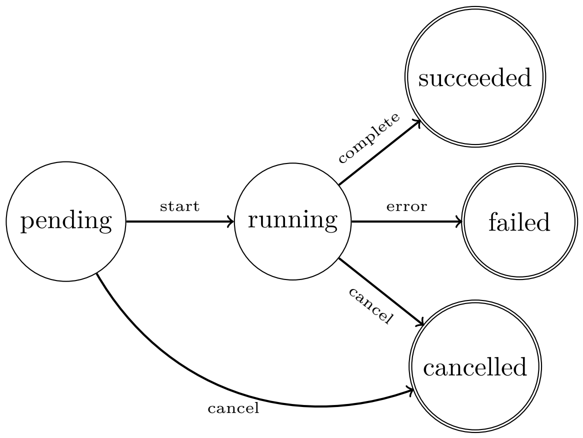

Figure 5.2. Agent run state machine

succeeded Terminal state indicating successful completion. The run produced valid output and all steps completed without errors.

failed Terminal state indicating execution failure. An unrecoverable error occurred during processing.

cancelled Terminal state indicating user or system cancellation. The run was terminated before completion, either by explicit request or timeout.

5.3.2 End-to-End Request Flow

The complete lifecycle of an agent run spans multiple system components, from HTTP request reception to final response delivery.

Step 1: HTTP Request Reception

The client submits a `POST` request to `/v1/agent-runs` with the following payload:

```
{
    "prompt": "How many :Blast nodes are in the graph?",
    "session_id": "uuid-optional",  # Optional session context
    "model": "phi3-mini-instruct",  # Optional model override
    "max_steps": 8,                  # Maximum execution steps
    "temperature": 0.2,              # LLM sampling temperature
    "metadata": {}                   # Arbitrary key-value data
}
```

Listing 5.6. Agent run request payload

Step 2: Authentication and Authorization

The router extracts and validates the JWT bearer token:

1. Token signature verification against JWKS public keys
2. Claims validation (issuer, audience, expiration)

3. Principal construction with user ID, tenant ID, roles, and scopes
4. Permission check for `agents:run` scope

Step 3: Run Creation and Persistence

Upon successful authorization:

1. A new run record is created with status `pending`
2. The run is persisted to PostgreSQL via `AgentRunRepository`
3. An event is logged to the audit trail
4. The run ID is returned in the `202 Accepted` response

Step 4: Orchestrator Invocation

The router invokes the orchestrator, either synchronously or via a background task:

```python
async def execute_agent_run_background(
    run_id: UUID,
    prompt: str,
    user_id: str,
    session_id: str,
    tenant_id: str,
    params: dict[str, Any],
):
    # Transition to running
    run.status = "running"
    db.commit()

    # Execute orchestration
    result = await orchestrator.run(
        goal=prompt,
        user_id=user_id,
        tenant_id=tenant_id,
        **params,
    )

    # Update with results
    run.status = "succeeded" if result.success else "failed"
    run.output = result.to_dict()
    db.commit()
```

Listing 5.7. Orchestrator invocation

Step 5: Intent Classification and Routing

The orchestrator classifies the prompt and routes to the appropriate handler:

1. `classify_intent()` analyzes the prompt
2. Based on mode, the orchestrator invokes:
    - `_handle_chat()` for CHAT mode

- `_handle_graph_query()` for GRAPH mode
- `_handle_admin_mode()` for ADMIN mode
- etc.

Step 6: Step Execution Loop

For complex requests, the orchestrator executes a planning and step execution loop:

1. **Task Planning**: Generate a list of steps to execute
2. **Step Iteration**: For each task item:
   (a) Resolve the tool or action
   (b) Prepare input parameters
   (c) Execute with timeout
   (d) Capture output and metrics
   (e) Update step status
3. **Result Aggregation**: Combine step outputs into final response

Step 7: Response Assembly and Delivery

Upon completion:

1. The `OrchestrationResult` is serialized to JSON
2. The run record is updated with output and final status
3. Metrics are emitted to Prometheus
4. The response is returned to the client (for sync) or stored for polling (for async)

5.3.3 Polling and SSE Streaming

For asynchronous runs, clients retrieve results via two mechanisms:

Polling Clients may poll `GET /v1/agent-runs/{id}` to check run status and retrieve results.

SSE Streaming For real-time progress, clients may subscribe to `GET /v1/agent-runs/{id}/events` for Server-Sent Events providing step-by-step updates.

5.3.4 Agent Run Internal Structure

The agent run model comprises a hierarchy of entities: Sessions contain Runs, and Runs contain Steps. This structure enables conversational context, step-level auditability, and granular observability.

Session Entity

Sessions provide conversational context spanning multiple runs, enabling message history for context continuity, per-session LLM and tool preferences, and conversation state management.

Run Entity

Runs represent single execution requests producing final output, with fields including run_id, session_id, status, model, input, output, steps, todos, metrics, and timing information.

Step Entity

Steps represent individual units of work within a run, with types including message, assistant, tool, system, and error.

5.3.5 Persistence and Caching Strategy

The agent run lifecycle leverages a dual-storage approach:

PostgreSQL (Control Plane) Authoritative storage for run records with complete history, step records with input/output payloads, session state and configuration, and audit trail entries.

Redis (Data Plane) Ephemeral storage for session state caching for fast access, step sequence counters, session locks for concurrent access control, cancellation flags, and ETag invalidation markers.

5.3.6 Error Handling and Recovery

The run lifecycle implements comprehensive error handling including error classification, graceful degradation with retry and fallback strategies, and cooperative cancellation handling.

5.4 NL-to-Cypher Pipeline

The Natural Language to Cypher (NL-to-Cypher) pipeline enables users to query the Memgraph knowledge graph using natural language, abstracting away the complexity of the Cypher query language while maintaining security and performance. This section examines the pipeline architecture, processing stages, safety mechanisms, and the advantages this approach offers over traditional retrieval-augmented generation (RAG) systems.

5.4.1 Pipeline Architecture Overview

The NL-to-Cypher pipeline transforms natural language questions into executable Cypher queries through a multi-stage process implementing a defense-in-depth strategy, applying multiple validation layers before any query reaches the database.

5.4.2 Worked Example: End-to-End NL-to-Cypher Pipeline

To illustrate the complete pipeline operation, this subsection presents a concrete example tracing a natural language query through all stages from input to final response.

Input Query The user submits the following natural language query:

"Find all genes associated with diabetes and list their protein products."

Stage 1: Natural Language Normalization The input text is normalized: leading/trailing whitespace trimmed, entity keywords extracted ("genes", "diabetes", "protein products"), and intent signal detected (multi-hop graph traversal pattern).

Stage 2: Catalog Lookup The pipeline checks the prompt catalog for exact or pattern matches. If no match is found, processing proceeds to LLM-based generation.

Stage 3: LLM-Based Cypher Generation The pipeline invokes the LLM with a prompt including:

- Schema context (node labels: `Gene`, `Disease`, `Protein`; relationship types: `ASSOCIATED_WITH`, `ENCODES`)
- Example Cypher queries demonstrating the pattern
- Safety guidelines (read-only, tenant filtering required)

The LLM generates the following Cypher query:

```
MATCH (g:Gene)-[:ASSOCIATED_WITH]->(d:Disease {name: 'diabetes'})
MATCH (g)-[:ENCODES]->(p:Protein)
RETURN g.name AS gene_name, p.name AS protein_name
LIMIT 100
```

Listing 5.8. Generated Cypher query (before validation)

Stage 4: Safety Validation The generated Cypher undergoes six-layer validation:

Table 5.4. Safety Validation Rules Applied to Generated Cypher

Layer	Rule Category	Validation Applied
1	Syntax	Cypher parser validates query structure
2	Read-only	No `CREATE`, `DELETE`, `SET`, `REMOVE` keywords detected
3	Tenant isolation	`WHERE` clause checked; tenant filter added: `AND g.tenant_id = $tenant_id`
4	Depth limit	Maximum hop count verified (2 hops: Gene→Disease, Gene→Protein)
5	Timeout	Query timeout parameter set: `30 seconds`
6	Result cap	`LIMIT 100` clause present; enforced as maximum

Table 5.4 summarizes the safety rules applied. After validation, the final query includes tenant filtering:

```
MATCH (g:Gene)-[:ASSOCIATED_WITH]->(d:Disease {name: 'diabetes'})
MATCH (g)-[:ENCODES]->(p:Protein)
WHERE g.tenant_id = {tenant_id} AND d.tenant_id = {tenant_id} AND p.
    tenant_id = {tenant_id}
RETURN g.name AS gene_name, p.name AS protein_name
LIMIT 100
```

Listing 5.9. Validated Cypher query with tenant isolation

Stage 5: Query Execution The validated query executes against Memgraph with parameter binding (`tenant_id` parameter set to the user's tenant identifier). Execution completes in 45ms, returning 23 result rows.

Stage 6: Result Summarization The pipeline invokes the LLM to summarize results into natural language:

> *"Found 23 genes associated with diabetes. Key examples include INS (encoding insulin), TCF7L2 (transcription factor), and PPARG (peroxisome proliferator-activated receptor). These genes collectively contribute to glucose metabolism, insulin signaling, and beta-cell function pathways."*

This example demonstrates how the pipeline transforms a natural language question through normalization, LLM generation, multi-layer safety validation, execution, and summarization, ensuring both usability and security.

5.4.3 Pipeline Stages

Stage 1: Natural Language Normalization

The first stage prepares the input text for processing through input sanitization, entity extraction, context enrichment, and intent signal extraction.

Stage 2: Catalog Lookup and Pattern Matching

Before invoking the LLM, the pipeline checks if the query matches a known pattern in the prompt catalog, enabling deterministic execution for common queries.

Stage 3: LLM-Based Cypher Generation

For queries not matching the catalog, the pipeline invokes an LLM to generate Cypher with schema context and low temperature for consistency.

Stage 4: Safety Validation

The generated Cypher undergoes rigorous safety validation through multiple layers:

1. **Layer 1: Syntax Validation** — Basic Cypher syntax checking
2. **Layer 2: Read-Only Enforcement** — Blocking write operations for non-admin users

3. **Layer 3: Tenant Boundary Isolation** — Automatic tenant_id filtering
4. **Layer 4: Depth and Complexity Limits** — Bounding path traversal
5. **Layer 5: Timeout Guards** — Query execution time limits
6. **Layer 6: Result Size Caps** — Preventing memory exhaustion

Stage 5: Query Execution

Validated queries are executed against Memgraph with tenant filtering and timeout enforcement.

Stage 6: Result Summarization

Query results are summarized into natural language, with simple counts bypassing LLM summarization for efficiency.

5.4.4 Failure Modes and Safety Guardrails

The NL-to-Cypher pipeline implements comprehensive handling for potential failure modes including hallucinated queries, schema mismatch, injection attacks, performance issues, and tenant data leakage, each with specific mitigations.

5.4.5 Test Mode for Deterministic Testing

The pipeline supports a test mode enabling deterministic testing independent of LLM output variability through prompt-to-Cypher hint mappings.

5.4.6 NL-to-Cypher Advantages over RAG

The NL-to-Cypher approach offers several advantages over traditional RAG including structured reasoning, multi-hop queries, aggregation support, schema awareness, deterministic execution, and RBAC integration.

Design Alternatives Considered

During the design of the NL-to-Cypher pipeline, several alternative approaches were evaluated:

Alternative 1: Direct Cypher Input Users could write Cypher queries directly, bypassing natural language processing entirely. This approach was rejected because: (1) it requires users to learn Cypher syntax, limiting accessibility; (2) it bypasses safety validation layers, increasing security risk; (3) it reduces the value proposition of making graph databases accessible to non-technical users.

Alternative 2: Template-Based Query Generation Pre-defined query templates could be matched to user intent using keyword matching or intent classification, then filled with parameters. This approach was considered but rejected because: (1) it cannot handle novel queries not covered by templates; (2) maintenance burden increases with domain complexity; (3) it lacks the flexibility to handle complex multi-hop graph traversals that LLMs can generate dynamically.

Alternative 3: Fine-Tuned Domain-Specific Model A specialized language model could be fine-tuned on Cypher query examples for the specific graph domain. This approach was not chosen because: (1) fine-tuning requires substantial training data and computational resources; (2) domain changes (new node labels, relationship types) require model retraining; (3) the zero-shot LLM approach with schema context achieves sufficient accuracy while remaining flexible and maintainable.

The chosen approach (zero-shot LLM generation with multi-layer validation) balances accessibility, flexibility, and safety while maintaining acceptable performance characteristics.

5.4.7 Trade-offs and Limitations

The approach has limitations including schema dependence, generation brittleness, limited fuzzy matching, maintenance burden, and no vector search.

5.5 LLM Provider Management

Large Language Model (LLM) provider management is a critical subsystem of the orchestration engine, responsible for selecting, invoking, and monitoring the language models that power agent reasoning and generation.

5.5.1 Provider Registry Architecture

The platform implements a database-backed provider registry that enables dynamic management of LLM providers without code changes or redeployment.

Provider Model

Providers are registered as entities in the PostgreSQL control plane with fields including id, tenant_id, name, provider_type, base_url, api_key_ref, circuit breaker settings, and cost tracking configuration.

Model Instance Model

Model instances represent specific models available through providers with fields including id, provider_id, model_id, context_length, capability flags, and usage tracking.

Default Model Resolver (DMR)

The Default Model Resolver determines which model to use for a given request with resolution priority: explicitly requested model, tenant-configured default, then first available model by provider priority.

5.5.2 Resilience Framework

The resilience framework provides fault-tolerant LLM calling with automatic failover, circuit breakers, and cost governance.

LLMFallbackOrchestrator

The `LLMFallbackOrchestrator` class coordinates resilient LLM calls with circuit breakers per provider, cost trackers per provider, and comprehensive statistics.

Call Flow with Resilience Checks

Each LLM call proceeds through multiple resilience checks: circuit breaker status, cost cap verification, token limit validation, and then the actual call attempt with timeout.

5.5.3 Budgeting and Cost Governance

The platform implements comprehensive cost tracking and governance to prevent budget overruns and enable chargeback accounting.

Cost Calculation

Token-based cost calculation uses provider-specific pricing with sliding window tracking for budget enforcement.

Cost Aggregation Levels

Costs are tracked at multiple granularities: per-request, per-session, per-run, per-tenant, and per-provider.

5.5.4 Circuit Breaker Implementation

The circuit breaker pattern prevents cascade failures when LLM providers experience issues through a three-state machine: CLOSED (normal), OPEN (blocking), and HALF_OPEN (testing recovery).

5.5.5 Provider Fallback Strategy

The fallback strategy ensures high availability by routing to backup providers when the primary is unavailable, with providers ordered by priority and capability matching for fallback selection.

5.5.6 Registry-Based Model Management

The platform's registry-based approach enables operational flexibility without code changes, supporting provider registration, model instance registration, local LLM onboarding (Ollama), and comprehensive usage tracking.

5.5.7 Health Probes

The resilience framework includes health probes for each provider, invoked periodically by the background scheduler, on demand via health endpoints, and during circuit breaker recovery.

Chapter 6

MCP Tools Ecosystem

This chapter describes the Model Context Protocol (MCP) tool ecosystem implementation, detailing how the platform provides a standardized, extensible framework for tool integration that enables agents to interact with external systems and perform specialized tasks. Building upon the orchestration engine design in Chapter 5, this chapter explains the MCP runtime architecture, documents the complete tool inventory, describes authorization mechanisms, and provides guidance for extending the ecosystem. The tool system addresses the functional requirements for tool integration established in Chapter 3.

6.1 MCP Runtime Architecture

The Model Context Protocol (MCP) runtime (introduced in Section 2.3) provides the foundational infrastructure for tool execution within the Cineca Agentic Platform. This runtime layer enforces consistent contracts, security policies, and observability patterns across all 34 tools (at time of writing) in the platform. The architecture follows a decorator-based design that separates cross-cutting concerns from tool-specific business logic, enabling rapid tool development while maintaining enterprise-grade governance.

6.1.1 Architectural Overview

The MCP runtime implements a layered architecture consisting of four primary components:

1. **Tool Registry**: A dynamic discovery mechanism that locates and loads tool modules based on naming conventions.
2. **Execution Context**: A structured context object (`ToolContext`) that carries request metadata, principal information, and tenant isolation boundaries through the invocation lifecycle.
3. **Runtime Wrapper**: A decorator-based wrapper (`@mcp_tool`) that enforces contracts, validates inputs, and applies cross-cutting concerns.
4. **Telemetry Layer**: Prometheus metrics and structured logging for comprehensive observability.

6.1.2 Tool Registry and Discovery

The tool registry implements a convention-over-configuration approach for tool discovery. Tools are organized in a hierarchical module structure under `src/mcp/tools/`, with each tool residing in a category subdirectory. The registry translates MCP tool names (e.g., `graph.query`) to Python module paths (e.g., `src.mcp.tools.graph.query`) using a deterministic mapping function.

The callable discovery follows a priority order: `invoke` → `run` → `handle`. This convention allows tools to expose a single entry point without requiring explicit registration.

6.1.3 Tool Execution Context

The `ToolContext` class encapsulates all execution metadata required for a tool invocation. It serves as the single source of truth for request-scoped information and provides utility methods for timeout checking, logging, and audit trail generation.

The context is automatically created by the MCP runtime and propagated through the tool invocation chain. Key attributes are extracted from the incoming HTTP request:

- **Principal**: Extracted from the validated JWT token, containing user identity and granted scopes.
- **Tenant**: Extracted from the `X-Tenant-ID` header for multi-tenant isolation.
- **Trace ID**: Propagated from the `X-Request-ID` header or generated if absent.
- **Timeout**: Configurable per-tool with fallback to global defaults.

6.1.4 The `@mcp_tool` Decorator

The `@mcp_tool` decorator is the cornerstone of the MCP runtime, providing a declarative way to apply cross-cutting concerns to tool implementations. The decorator wraps the tool function and intercepts every invocation to enforce policies.

When a decorated tool is invoked, the wrapper executes the following sequence:

1. **Context Creation**: Build `ToolContext` with request metadata.
2. **RBAC Check**: Verify principal has `required_scope`.
3. **Rate Limit Check**: Validate against per-principal rate limits.
4. **Input Validation**: Validate payload against Pydantic schema.
5. **Audit Log (Pre)**: Record invocation attempt with sanitized payload.
6. **Tool Execution**: Call the wrapped function with context and payload.
7. **Telemetry**: Record latency histogram and invocation counter.
8. **Audit Log (Post)**: Record result status (success/failure).
9. **Response Sanitization**: Ensure response conforms to standard shape.

6.1.5 Schema Validation

Input validation is enforced using Pydantic models defined in `src/mcp/schemas.py`. Each tool can define its own payload schema, and the runtime validates all incoming payloads before execution.

6.1.6 Audit Logging

Every tool invocation is recorded in the audit log for compliance and debugging purposes. The audit integration captures: who (principal identity), what (tool name, action, sanitized payload), when (timestamp with millisecond precision), where (trace ID for distributed correlation), and result (success/failure status and error details if applicable).

6.1.7 Telemetry and Metrics

The MCP runtime exposes Prometheus metrics for tool observability. These metrics are automatically collected by the `@mcp_tool` decorator and made available on the `/metrics` endpoint, including invocation counts, latency histograms, and error counts by type.

6.1.8 Error Handling

The MCP runtime defines a hierarchy of typed exceptions for consistent error handling across all tools. Each exception type maps to a specific error code that appears in API responses: `ToolError` (base), `ValidationError_`, `PermissionError_`, `TimeoutError_`, and `RateLimitError_`.

6.2 Tooling Use Cases and Interaction Patterns

The MCP tools ecosystem serves as the programmatic interface through which agents, operators, and external systems interact with the Cineca Agentic Platform. This section describes the primary use cases addressed by the tooling layer and the interaction patterns that govern tool discovery, invocation, and error handling.

6.2.1 Primary Use Cases

The 34 MCP tools are designed to support five distinct categories of use cases:

Graph Question-Answering

The most prominent use case involves natural language queries over the Memgraph knowledge graph. Users pose questions in natural language, which are converted to Cypher queries through the NL-to-Cypher pipeline and executed against the graph database.

System Health and Monitoring

Operators and automated systems use monitoring tools to assess platform health, diagnose issues, and integrate with infrastructure monitoring solutions.

Administrative Operations

Platform administrators use administrative tools for tenant management, user administration, and operational tasks that require elevated privileges.

Graph Data Management

Data engineers and applications use graph management tools to populate, update, and maintain the knowledge graph.

Session and Context Management

Agents require persistent context across multi-turn conversations. Context management tools provide session lifecycle and preference storage.

6.2.2 Interaction Patterns

All tools follow consistent interaction patterns that ensure predictability, debuggability, and integration compatibility, including tool discovery, authorization flow, invocation pattern, error handling pattern, and audit trail pattern.

6.3 Tool Categories and Inventory

The Cineca Agentic Platform implements 34 MCP tools organized into 17 categories. Each category groups related functionality and shares common security scopes and operational characteristics.

6.3.1 Category Organization

Tools are organized in a two-level hierarchy: *category* and *tool*. The category represents a functional domain (e.g., `graph`, `security`), while the tool represents a specific capability within that domain.

6.3.2 Graph Tools

Graph tools provide comprehensive access to the Memgraph knowledge graph, supporting both read operations and write mutations. Key tools include `graph.query` for direct Cypher execution, `graph.secure_query` for NL-to-Cypher with validation, `graph.schema` for schema introspection, `graph.analytics` for graph algorithms, and `graph.crud`/`graph.bulk` for write operations.

6.3.3 Security Tools

Security tools provide introspection and management of the platform's security controls, including `security.audit` for audit log access, `security.permissions` for permission checking, and `security.describe_principal` for identity introspection.

6.3.4 System Tools

System tools provide operational visibility and management capabilities, including `system.health` for Kubernetes-compatible health probes and `system.backup` for backup management.

6.3.5 Model Tools

Model tools provide runtime LLM management and testing capabilities, including `model.manage` for model configuration and `model.test` for model testing.

6.3.6 Other Tool Categories

The remaining tool categories provide specialized functionality: Agent, Cache, Catalog, Data, Database, Errors, Output, Privacy, Rate Limit, Session, Tenancy, User, and Visualization tools.

6.4 Tool Authorization

Tool authorization in the Cineca Agentic Platform implements a defense-in-depth strategy that combines scope-based RBAC, tenant isolation, and policy-aware access control. Every tool invocation passes through multiple authorization checkpoints before execution.

6.4.1 Authorization Model

The tool authorization model is built on three foundational concepts: Scopes (fine-grained permission tokens), Principals (authenticated entities with associated scopes and tenant membership), and Resources (the tools and their actions that principals attempt to access).

Scopes follow a hierarchical naming convention that enables both fine-grained and broad permissions, with wildcard support (e.g., `graph:*`) and admin override (`admin:all`).

6.4.2 Per-Tool RBAC

Each tool declares its required scope through the `@mcp_tool` decorator. The MCP runtime enforces this requirement at invocation time through the `check_permissions` function.

6.4.3 Tenant Isolation

Multi-tenant isolation is enforced at the tool level to prevent cross-tenant data access. The tenant context is propagated through the request lifecycle and tools automatically scope all data operations by tenant.

6.4.4 Policy Files

For complex authorization scenarios, the platform supports external policy definitions in YAML files that can be managed separately from tool code.

6.4.5 Authorization Decision Flow

The complete authorization decision flow includes JWT validation, principal existence check, tenant matching, scope verification, and policy evaluation.

6.5 Extending the Tool Ecosystem

One of the key design goals of the MCP tools ecosystem is extensibility. The platform provides a well-defined pattern for adding new tools, integrating additional LLM providers, and extending functionality at various layers.

6.5.1 Adding New Tools

Adding a new tool to the ecosystem is straightforward and requires minimal boilerplate:

1. Create the tool module at the appropriate location under `src/mcp/tools/`
2. Define the payload schema using Pydantic
3. Implement the tool following standard patterns
4. Add unit tests
5. Verify auto-registration

6.5.2 Extensibility Guide with Difficulty Ratings

Common extension tasks range in difficulty:

- **Add new MCP tool**: Low difficulty (1–2 files)
- **Add new LLM provider**: Medium difficulty (2–3 files)
- **Add new API endpoint**: Medium difficulty (2–3 files)
- **Add new background job**: Medium-High difficulty (3–4 files)
- **Full feature extension**: High difficulty (5+ files)

6.5.3 Testing Tools

Tools should be tested at multiple levels: unit tests for individual action implementations, integration tests with real or simulated dependencies, and security tests for authorization and tenant isolation.

6.5.4 Best Practices

When extending the tool ecosystem, follow best practices for naming conventions, action design, error handling, performance, and testing.

Complete Tool Reference For a complete reference of all 34 MCP tools, including detailed payload schemas, result formats, required scopes, and example invocations, see Appendix B.

Chapter 7

Background Jobs and Workers

This chapter details the background job processing system that enables asynchronous execution of long-running agent tasks, ensuring system responsiveness while supporting operations that may require significant computation time. The job system integrates with the orchestration engine (Chapter 5) and provides durable state persistence, progress tracking via Server-Sent Events (SSE), cancellation mechanisms, and fault-tolerance patterns for production deployment. This infrastructure addresses the non-functional requirements for reliability and scalability established in Chapter 3.

7.1 Asynchronous Job System

Enterprise AI agent platforms must handle workloads that exceed typical HTTP request timeouts. Training runs, batch processing, graph analytics, and multi-step agent executions can require minutes to hours of processing time. The Cineca Agentic Platform addresses this requirement through a comprehensive asynchronous job system that decouples job submission from execution, provides real-time progress updates, and ensures reliable completion even under adverse conditions.

7.1.1 Design Alternatives Considered

During the design of the background job system, several architectural alternatives were evaluated:

Alternative 1: Database-Based Job Queue Jobs could be stored only in PostgreSQL with a single polling query. This approach was considered but rejected because: (1) PostgreSQL polling creates database load and lock contention at scale; (2) job state updates require database transactions, limiting throughput; (3) Redis provides superior performance for high-frequency queue operations (BRPOP is more efficient than SELECT FOR UPDATE).

Alternative 2: Message Queue Middleware A dedicated message queue (e.g., RabbitMQ, Apache Kafka) could handle job queuing and distribution. This approach was not chosen because: (1) it introduces additional infrastructure dependencies and operational complexity; (2) Redis provides sufficient functionality for the job queue use case (single-consumer, FIFO ordering) without the overhead of full message queue systems; (3) Redis is already required for caching and rate limiting, making it a natural choice for job queuing.

Alternative 3: Serverless/Cloud Functions Jobs could be executed via cloud functions (AWS Lambda, Google Cloud Functions) triggered by database events or message queues. This approach was not feasible because: (1) the platform targets self-hosted deployment scenarios where cloud services may not be available; (2) cloud functions introduce cold start latency and execution time limits that conflict with long-running job requirements; (3) the platform's goal is to provide a complete, self-contained solution.

The chosen dual-store architecture (PostgreSQL for persistence, Redis for queuing) balances durability, performance, and operational simplicity while leveraging existing infrastructure dependencies.

7.1.2 Design Goals and Requirements

The asynchronous job system was designed to satisfy several key requirements derived from enterprise deployment scenarios:

1. **Decoupled Execution**: Job submission must return immediately with a job identifier, allowing clients to poll for status or subscribe to event streams without blocking.
2. **Multi-Tenant Isolation**: Jobs must be scoped to tenants and owners, with strict access control preventing cross-tenant visibility or manipulation.
3. **Reliable State Tracking**: Job status transitions must be atomic and persistent, surviving process restarts and infrastructure failures.
4. **Idempotent Submission**: Duplicate job submissions with the same parameters must return the existing job rather than creating duplicates.
5. **Real-Time Progress**: Clients must receive timely updates on job progress through Server-Sent Events (SSE) with resume capability.
6. **Graceful Cancellation**: Running jobs must support cooperative cancellation without resource leaks or inconsistent state.
7. **Operational Visibility**: Comprehensive metrics must expose job throughput, latency distributions, and failure rates.

7.1.3 Distinguishing Agent Run States from Job States

The platform maintains two distinct but related state machines: **Agent Run states** (tracking individual agent execution requests) and **Job states** (tracking asynchronous task execution). Understanding the relationship between these state machines is essential for debugging and monitoring.

Table 7.1 summarizes the state spaces. Key distinctions:

Table 7.1. Agent Run States vs Job States

State Type	States	Description
Agent Run	`pending`	Run created, awaiting execution
	`running`	Run actively executing steps
	`succeeded`	Run completed successfully
	`failed`	Run terminated with error
	`cancelled`	Run cancelled by user/system
Job	`queued`	Job submitted to queue, awaiting worker
	`running`	Job actively executing on worker
	`finished`	Job completed successfully
	`failed`	Job terminated with unrecoverable error
	`cancelled`	Job cancelled before completion

- **Agent Run states** track the logical execution of a user's request, regardless of whether execution is synchronous or asynchronous. Runs represent the user-facing abstraction: "I submitted a request, what happened?"
- **Job states** track the physical execution of long-running tasks in the background worker system. Jobs represent the infrastructure abstraction: "I queued a task, has the worker processed it?"
- **State name overlap**: Both use "running" and "failed/cancelled", but they operate at different abstraction levels. A Run in "running" state may correspond to a Job in "queued" state if the job hasn't been picked up by a worker yet.

When Runs Execute via Jobs vs Synchronous The platform supports two execution modes:

1. **Synchronous Execution**: Short-running agent runs execute directly within the HTTP request-response cycle. The run progresses through states (`pending` → `running` → `succeeded/failed`) while the HTTP request is active. No Job entity is created. This mode is used for:
   - Simple conversational queries (CHAT mode)
   - Quick graph queries with small result sets
   - Permission checks (SECURITY mode)
   - Operations expected to complete in <30 seconds
2. **Asynchronous Execution via Jobs**: Long-running agent runs are executed asynchronously via the job system. When a run is submitted with `async=true` or exceeds a timeout threshold, a Job entity is created with state `queued`. The run state remains `pending` until the job starts executing. This mode is used for:
   - Complex multi-step agent runs with tool invocations
   - Large graph queries or analytics operations
   - Operations expected to take >30 seconds
   - Batch processing tasks

State Synchronization When a Run executes via a Job, state transitions are synchronized:

- **Run `pending` + Job `queued`**: Run submitted, job queued but not yet picked up by worker
- **Run `pending` + Job `running`**: Job executing, run still pending (worker processing)
- **Run `running` + Job `running`**: Both actively executing (job worker calls orchestrator)
- **Run `succeeded/failed` + Job `finished/failed`**: Execution complete, states synchronized

Cancellation propagates bidirectionally: cancelling a Run cancels the associated Job (if any), and Job cancellation updates the Run state to `cancelled`.

7.1.4 Job Lifecycle State Machine

The job lifecycle follows a deterministic finite state machine with five states: QUEUED (initial), RUNNING (active execution), FINISHED (successful completion), FAILED (unrecoverable error), and CANCELLED (user or system cancellation).

7.1.5 Job Document Structure

The `JobDocument` model represents the core job entity, designed to be storage-agnostic while supporting serialization to both Redis HASH structures and PostgreSQL rows. Key fields include: id (UUID), owner (subject from JWT), tenant_id (isolation boundary), type (job type), status (lifecycle state), payload (input parameters), result (output on completion), error (message if failed), and timestamps.

7.1.6 Job Type Registry

The platform supports multiple job types, each with distinct execution semantics: demo (sleep-based simulation), test (instant completion with payload echo), long-running (multi-step execution with progress updates), and agent.run (full agent orchestration execution).

7.1.7 Storage Abstraction Layer

The job system employs a repository pattern with abstract interfaces, enabling pluggable storage backends. Three interfaces define the contract: JobStore (job document persistence), IdempotencyStore (duplicate request detection), and EventStore (SSE event ring buffer management).

7.1.8 Storage Backend Implementations

The Redis implementation (production) uses purpose-built data structures: Job Documents as HASH, Global/Owner/Status Indexes as ZSET, Event Ring as LIST, and Idempotency Keys as String with TTL. The in-memory implementation (development/testing) wraps Python dictionaries with identical API semantics.

7.1.9 Idempotency Mechanism

Idempotent job creation prevents duplicate jobs from network retries or client errors. The idempotency key is computed from request context (owner, tenant, job type, payload hash) and stored with 24-hour TTL.

7.2 Worker Architecture

The worker architecture implements a separate process model for job execution, decoupling the FastAPI web application from long-running computations. This design enables horizontal scaling of worker capacity independently from API throughput, prevents job execution from blocking HTTP request handling, and provides isolation for resource-intensive operations.

7.2.1 Process Separation Model

The Cineca Agentic Platform employs a process-per-role architecture where the web application and job workers run as separate OS processes. This separation provides resource isolation, independent scaling, fault isolation, and deployment flexibility.

7.2.2 Queue Polling Mechanism

Workers poll Redis queues using the blocking `BRPOP` operation, which efficiently waits for new jobs without busy-waiting. The queue polling iterates through allowed job types in round-robin fashion.

7.2.3 Job Execution Flow

When a job is dequeued, the worker executes a well-defined sequence: Load Job, Check Cancellation, Transition to RUNNING, Execute with Heartbeat, Check Cancellation, and Transition to Terminal state.

7.2.4 Heartbeat Mechanism

The heartbeat mechanism serves two purposes: indicating worker liveness and enabling detection of stale jobs from crashed workers. A concurrent asyncio task updates the job's `updated_at` timestamp periodically.

7.2.5 Job Type Dispatching

The worker dispatches execution to type-specific handlers based on the `job.type` field, including demo job (sleep simulation with cancellation checking) and long-running job (multi-step execution with progress updates).

7.2.6 Graceful Shutdown

The worker implements graceful shutdown to ensure clean process termination: signal reception sets `running = False`, current job completes, resources are cleaned up via context managers, and the worker deregisters from the active pool.

7.3 Job Events and SSE Streaming

Real-time progress updates are essential for user experience when executing long-running jobs. The Cineca Agentic Platform implements Server-Sent Events (SSE) streaming with a ring buffer architecture that supports client reconnection and event replay.

7.3.1 SSE Event Model

The `SSEEvent` model represents a single event in the job event stream with fields: event_id (monotonic sequence number), event_type (status, progress, heartbeat, end, error), data (event payload), and timestamp.

7.3.2 Ring Buffer Architecture

Events are stored in a ring buffer implemented using Redis LIST data structures. The ring buffer provides bounded storage with automatic eviction of oldest events via atomic LPUSH + LTRIM operations.

7.3.3 Monotonic Event IDs

Each event receives a monotonically increasing ID within the job scope, enabling clients to detect gaps and request replay. The ID is generated using Redis INCR for atomic increment.

7.3.4 Event Replay for Reconnection

When a client reconnects with a `Last-Event-ID` header, the server replays missed events from the ring buffer. If the ring buffer has rotated past the requested ID, the server emits a comment indicating the gap.

7.3.5 SSE Endpoint Implementation

The SSE streaming endpoint implements the full W3C Server-Sent Events protocol with connection tracking, event replay, heartbeats for keep-alive, and terminal event emission.

7.4 Job Cancellation

Job cancellation presents a coordination challenge in distributed systems: the cancel request originates from an API endpoint, but the job execution occurs in a separate worker process. The Cineca Agentic Platform implements a cooperative cancellation model using Redis flags for cross-process communication and Lua scripts for atomic state transitions.

7.4.1 Cancellation Model

The platform employs a *cooperative cancellation* pattern where: a cancel flag is set in Redis by the API endpoint, the worker periodically checks this flag during job execution, and if set, the worker gracefully terminates and transitions to CANCELLED status.

7.4.2 Cancel Flag Mechanism

The cancel flag is stored as a simple Redis key with the pattern `job:{id}:cancel` with TTL matching job expiry.

7.4.3 Cancellation Flow

The cancellation sequence spans API and worker processes: Client sends POST to cancel endpoint, API sets cancel flag in Redis and returns 202 Accepted, Worker detects flag during periodic check, Worker transitions job to CANCELLED and clears flag.

7.4.4 Atomic Cancellation with Lua Scripts

For the Redis backend, atomic cancellation uses a Lua script to prevent race conditions between status checks and updates, implementing compare-and-set semantics.

7.5 High Availability and Fault-Tolerance Patterns

Enterprise deployments require the job system to operate reliably under adverse conditions: worker crashes, network partitions, database unavailability, and infrastructure restarts.

7.5.1 Worker Heartbeat and Dead-Worker Detection

Workers maintain liveness signals through periodic heartbeat updates. Dead workers can be detected by monitoring jobs in RUNNING status whose `updated_at` exceeds a threshold.

7.5.2 Orphaned Job Recovery

Jobs left in RUNNING status after a worker crash require recovery: detection of stale heartbeats, transition back to QUEUED, and re-enqueue to Redis for normal worker pickup.

7.5.3 Graceful Degradation

The system implements graceful degradation when dependencies become unavailable: Redis unavailability falls back to in-memory storage or returns 503; PostgreSQL unavailability pauses workers and returns 503 for database-dependent operations.

7.5.4 Health Probe Integration

The job system integrates with Kubernetes health probes for automated recovery, checking Redis connectivity, queue depths, and stuck job counts.

7.5.5 Known Limitations

Current limitations include: no automatic retry, single worker assumption without distributed locking, no dead-letter queue, and basic priority support.

7.6 Background Framework

Beyond the job queue for user-initiated tasks, the Cineca Agentic Platform requires periodic maintenance operations: health monitoring, data backups, cache cleanup, and index maintenance. The background framework provides a unified scheduler for these operations, built on APScheduler with comprehensive metrics integration.

7.6.1 Framework Overview

The background framework in `src/background.py` coordinates periodic tasks using the APScheduler library with the AsyncIO scheduler backend. Key design goals include configurable scheduling, graceful lifecycle management, metrics integration, failure isolation, and service composition.

7.6.2 BackgroundManager Class

The `BackgroundManager` class orchestrates task scheduling and execution with lifecycle management (async start/stop methods) and job registration with interval or cron triggers.

7.6.3 Scheduled Task Categories

The background framework includes four categories of scheduled tasks:

- **Health Monitoring Tasks**: Periodic checks of PostgreSQL, Redis, Memgraph, and LLM providers.
- **Backup Tasks**: Memgraph snapshots, Redis backup triggers, manifest generation.
- **Cleanup Tasks**: Expired session removal, old backup pruning, stale cache eviction.
- **Redis Index Cleanup**: Removal of orphaned ZSET entries for non-existent jobs.

7.6.4 FastAPI Lifespan Integration

The background framework provides a context manager for FastAPI lifespan integration, attaching a default BackgroundManager to `app.state.bg` and managing start/stop with the application lifecycle.

7.6.5 Metrics Emission

Background tasks emit Prometheus metrics for operational visibility, including job execution counts, status tracking, and duration histograms.

Chapter 8

Security Framework

This chapter presents the comprehensive security framework of the Cineca Agentic Platform, addressing the critical requirement for enterprise-grade security and governance in AI agent systems. Building upon the security foundations reviewed in Chapter 2 and the security requirements established in Chapter 3, this chapter details authentication mechanisms, authorization policies, multi-tenancy isolation, rate limiting, data protection measures, threat modeling, and audit logging. The security framework ensures that the platform meets enterprise requirements for access control, data protection, and compliance while maintaining usability for legitimate users.

8.1 Authentication

Authentication in the Cineca Agentic Platform follows industry-standard practices for enterprise security, implementing OpenID Connect (OIDC) with JSON Web Token (JWT) validation. The authentication subsystem is designed to integrate seamlessly with external Identity Providers (IdPs) such as Auth0, Okta, or any OIDC-compliant provider, while also supporting legacy symmetric key validation for backward compatibility.

8.1.1 OIDC/JWT-Based Authentication

The platform implements a stateless authentication model where all security context is carried within signed JWT tokens. This approach eliminates server-side session storage requirements and enables horizontal scaling of backend instances without shared session state.

JWT tokens carry critical claims including: sub (subject identifier), iss (issuer URL), aud (audience), exp (expiration), nbf (not before), iat (issued at), scopes (permission strings array; note: some providers use "scope" as a space-separated string, which is normalized to "scopes" array), permissions (Auth0-style claims), roles (role assignments), and tid/tenant_id (multi-tenancy identifier; note: if not present in token, tenant_id is derived from the user's tenant association in the database).

The platform supports two cryptographic schemes: RS256/ES256 (asymmetric) using JWKS from the OIDC provider, and HS256 (symmetric) as a legacy fallback for development environments.

8.1.2 Authentication Flow Details

The authentication flow follows a resource server model with these steps: Bearer Token Extraction from the Authorization header, Header Parsing to extract the key ID (kid), JWKS Key Retrieval with caching, Signature Verification using python-jose, Claims Validation (exp, nbf, iat), Issuer and Audience Validation, and Principal Construction.

The JWKS caching strategy balances security with performance: 900-second default TTL, 600-second minimum, per-key caching by kid, and respect for Cache-Control headers.

8.2 Authorization

The Cineca Agentic Platform implements a comprehensive Role-Based Access Control (RBAC) system that governs access to API endpoints, MCP tools, graph database operations, and administrative functions.

8.2.1 RBAC Model

The authorization model is built on roles that expand to scopes. Roles are coarse-grained labels (admin, researcher, operator), while scopes are fine-grained permission strings following a resource:action pattern.

The platform defines a role hierarchy: guest (minimal), researcher (read access), curator (data steward), data_engineer (ETL), operator (SRE), auditor (compliance), and admin (full access).

8.2.2 RBAC Implementation Details

Roles are expanded to scopes using configurable policy mappings. Scopes support exact match, resource wildcards (tools:*), and super wildcard (*). The platform supports two authorization modes: “any” (at least one required scope) and “all” (all required scopes).

Authorization integrates with FastAPI through dependency injection and generates audit trails for all decisions.

8.3 Multi-Tenancy

The Cineca Agentic Platform is designed from the ground up as a multi-tenant system, implementing a shared-infrastructure, isolated-data model. Multi-tenancy was first introduced as a requirement in Chapter 3; this section provides the detailed design and implementation.

8.3.1 Tenant Isolation Model

Tenant context is established through prioritized extraction: HTTP header (X-Tenant-Id), query parameter, token claims, or default tenant. Tenant identifiers are validated against a strict format to prevent injection attacks.

8.3.2 Database-Level Isolation

All database queries are automatically scoped to the current tenant. PostgreSQL queries include implicit tenant filtering, Memgraph queries filter by tenant boundaries, and Redis keys are namespaced by tenant.

Per-tenant configuration supports individualized rate limits, default models, tool access, safety settings, and cost budgets.

8.4 Rate Limiting

The platform implements a comprehensive rate limiting system using a fixed-window counter algorithm for its simplicity, predictability, and efficiency.

8.4.1 Backend Architecture

The primary Redis backend provides distributed rate limiting with atomic INCR operations. An in-memory fallback provides graceful degradation when Redis is unavailable (per-process, not distributed).

Rate limit keys combine tenant, user identity, and resource path for granular control. The system supports cost-based rate limiting where expensive operations consume multiple quota units.

Per-role rate limits are defined in policy files with RPM, TPM, burst capacity, and per-tool overrides.

8.5 Data Protection

The platform implements multiple layers of data protection at input, processing, and output stages.

8.5.1 PII Scrubbing

The PII scrubber provides zero-dependency heuristics to detect and sanitize personally identifiable information including email, phone, IPv4, SSN, IBAN, and credit cards (with Luhn validation). Redaction modes include mask, hash, remove, and off.

8.5.2 Output Guard

The output guard provides safety controls for LLM-generated content, particularly Cypher query validation. It analyzes queries for write operations, destructive verbs, unbounded traversals, and missing LIMIT clauses. Operating modes include enforce, monitor, and off.

8.5.3 Intent Filter

The intent filter provides lightweight guardrails against unsafe requests by detecting prompt injection, secrets scraping, PII hunting, dangerous shell operations, and SQL/Cypher destructive operations. Risk scoring determines whether to block or monitor requests.

8.6 Threat Model and Attack Scenarios

The platform considers three primary attacker profiles: Malicious Tenant (legitimate tenant abusing access), Compromised User (taken-over account), and External Attacker (unauthenticated probing).

8.6.1 Threat Categories

Key threat categories include: Data Exfiltration (cross-tenant queries, bulk export, prompt injection), Privilege Escalation (role confusion, scope injection, tool bypass), NL-to-Cypher Injection (destructive queries, schema discovery, unbounded traversals), Tool Abuse (system tool exploitation, CRUD abuse, rate limit evasion), Budget Abuse (token exhaustion, expensive model selection), and Session Attacks (hijacking, fixation, context poisoning).

Each threat has specific mitigations implemented through the layered security architecture.

8.7 Audit Logging

The platform implements comprehensive audit logging capturing security-relevant events throughout the request lifecycle.

8.7.1 Audit Event Model

Audit events include: event_id, timestamp, category (auth, access, policy, ratelimit, model, data), action, outcome, severity, principal, tenant_id, resource, trace_id, meta (scrubbed), and input/output hashes.

Key design decisions include content hashing (not raw content), metadata scrubbing of sensitive keys, and trace correlation for distributed tracing.

8.7.2 Output Channels

Audit events are emitted through multiple channels: structured logging (for aggregation systems), Prometheus metrics (for dashboards and alerting), and provenance chain (for tamper-evident archival).

The audit system provides fail-safe design (never interrupts requests), append-only storage, and design features relevant to compliance efforts including comprehensive audit logging (enabling GDPR Article 30 record-keeping requirements [11]), access controls (supporting SOC 2 control objectives [12]), data minimization through tenant isolation (relevant to HIPAA privacy rule requirements [13]), and secure data handling (supporting PCI-DSS data protection standards [14]). The platform does not provide formal compliance certification but implements technical controls that organizations can leverage in their compliance programs.

8.8 Security Strengths and Weaknesses

8.8.1 Strengths

Key strengths include: Layered Authentication Architecture (OIDC/JWT, JWKS, claims normalization), Explicit Scope-Based Authorization (declarative scopes, role expansion, audit integration), Strict Tenant Isolation (database-level filtering, key namespacing), Secure Graph Query Pipeline (read-only default, tenant injection, resource limits), Comprehensive Audit Trail (structured events, content hashing, multiple channels), and Privacy-Aware Data Handling (PII detection, configurable redaction).

8.8.2 Weaknesses and Limitations

Recognized limitations include: External IdP Dependency (single point of failure), No Formal Security Proofs (pattern-based detection), Encryption-at-Rest Assumptions (relies on infrastructure), In-Memory Rate Limiting Limitations (per-process counters), Limited Secret Management (environment variables, no rotation), and Test/Demo Mode Risks (production guards required).

8.8.3 Ethics and Privacy Considerations

The Cineca Agentic Platform implements several technical mechanisms that support privacy protection and ethical AI deployment, though formal compliance certification requires organizational processes beyond the platform itself.

PII Protection Mechanisms The platform's PII scrubbing system (Section 8.5.1) detects and redacts personally identifiable information including email addresses, phone numbers, Social Security Numbers, credit card numbers, IP addresses, and API keys. This mechanism addresses GDPR Article 32 (security of processing) requirements by preventing PII from appearing in logs, audit trails, or error messages that could be accessed by unauthorized parties.

However, the platform has **limitations**:

- PII detection uses pattern matching and may miss novel formats or context-dependent identifiers
- Scrubbing occurs at the logging/output layer but does not prevent PII from being stored in the graph database if users explicitly insert it
- The platform does not provide automated data subject rights fulfillment (GDPR Articles 15–22: access, rectification, erasure, portability)

Data Minimization and Purpose Limitation Multi-tenant isolation (Section 8.3) enforces data minimization by ensuring tenant data is accessible only to authorized users within that tenant. This supports GDPR Article 5(1)(c) (data minimization) and Article 5(1)(b) (purpose limitation) by preventing cross-tenant data access.

The platform's audit logging (Section 8.7) records data access events, enabling organizations to demonstrate compliance with GDPR Article 30 (records of processing activities). However, organizations must implement their own data retention policies and deletion procedures to fulfill GDPR Article 17 (right to erasure) requirements.

Consent and Lawful Basis The platform does not implement consent management interfaces or determine lawful basis for processing. These are organizational decisions that must be implemented at the application layer or through integration with identity providers that handle consent workflows.

Ethical AI Considerations The platform implements several mechanisms supporting ethical AI deployment:

- **Intent filtering**: Blocks potentially dangerous operations (Section 5.2), preventing accidental or malicious system damage
- **Output guards**: Validates LLM outputs for policy violations (Section 8.5.2), reducing risk of harmful content generation
- **Audit trails**: Comprehensive logging enables accountability and transparency in AI decision-making
- **Multi-tenant isolation**: Prevents one tenant's data from influencing another tenant's AI responses

However, the platform does not address:

- Bias detection or mitigation in LLM outputs
- Fairness metrics or demographic parity
- Explainability of AI decisions beyond audit logs
- Human-in-the-loop approval workflows for sensitive operations

These limitations reflect the platform's focus on infrastructure and orchestration rather than AI ethics tooling, which would require domain-specific implementations.

8.8.4 Recommendations for Production Deployment

Recommendations include: use enterprise IdP, enable TLS everywhere, use Redis for rate limiting, integrate secret management, enable database encryption, monitor security events, conduct regular security reviews, and establish incident response procedures.

Chapter 9

Observability

This chapter describes the observability infrastructure of the Cineca Agentic Platform, providing the metrics, logging, tracing, and health monitoring capabilities essential for operating a production AI agent system. Building upon the observability foundations reviewed in Chapter 2 and the non-functional requirements for monitoring established in Chapter 3, this chapter details the Prometheus metrics, structured logging, OpenTelemetry tracing, health probes, and operational dashboards that enable operators to understand system behavior, diagnose issues, and maintain service level objectives. The observability stack addresses the requirement for production-ready monitoring and alerting.

9.1 Metrics

The Cineca Agentic Platform implements a comprehensive metrics collection framework built on Prometheus. The metrics subsystem provides actionable insights into system behavior, supports Service Level Objectives (SLOs), and enables data-driven capacity planning and incident response.

9.1.1 Metrics Architecture Overview

The platform's metrics architecture follows a layered design: Metric Definition Layer (using `prometheus_client`), Collection Layer (instrumentation points across the codebase), and Exposition Layer (the `/metrics` endpoint in Prometheus text format).

The system supports Prometheus multiprocess mode via `PROMETHEUS_MULTIPROC_DIR` for aggregating metrics across multiple Uvicorn workers.

9.1.2 Comprehensive Metrics Catalog

The platform exposes metrics in seven categories:

HTTP Request Metrics: `http_requests_total` (Counter), `http_request_duration_seconds` (Histogram) with labels for method, path, and status. Duration buckets are optimized for API latencies (5ms to 10s).

Agent Run Metrics: `agent_runs_total`, `agent_run_duration_seconds`, `agent_active_runs`, `agent_phase_duration_seconds`, `agent_errors_total`, `agent_retries_total`, `agent_queue_depth`, `agent_concurrency_limit`, `agent_concurrency_throttled_total`.

LLM Call Metrics: `llm_calls_total`, `llm_call_duration_seconds`, `llm_tokens_total`, `llm_errors_total`, `model_requests_total`, `model_request_latency_ms`.

Tool Invocation Metrics: `tools_invocations_total`, `tools_invocation_duration_seconds`, `tools_queue_depth`, `tools_cache_operations_total`, `tools_idempotency_conflicts_total`

Job Metrics: `background_jobs_total`, `background_job_duration_seconds`, `job_create_total`, `job_create_duration_seconds`, `sse_connections_active`, `sse_gap_events_total`.

Intent Classification Metrics: `intent_classification_total`, `intent_classification_durati`
`intent_classification_confidence`, `intent_pattern_matches_total`, `intent_llm_fallbac`

Rate Limiting Metrics: `rate_limit_requests_total`, `rate_limit_exceeded_total`, `tenant_quota_exceeded_total`, `rate_limit_usage_ratio`.

9.2 Logging

The platform implements structured logging with `structlog`, providing JSON logs for production and human-readable console output for development.

9.2.1 Logging Architecture Overview

The architecture bridges Python's standard `logging`, FastAPI/Uvicorn loggers, and `structlog` processors into a unified output stream. Format selection is automatic based on `LOG_FORMAT` or `APP_ENV`.

9.2.2 Structured Log Schema

Production JSON logs include: timestamp (ISO 8601), level, logger, event (semantic identifier), request_id, trace_id, method, path, status, duration_s, and tenant_id.

9.2.3 Request Context Correlation

The `ObservabilityMiddleware` binds request-specific context (request_id, method, path, client_ip, trace_id) to all log entries during request processing.

9.2.4 High-Frequency Log Filtering

The `AccessPathFilter` suppresses logs for health and metrics endpoints. Library noise from asyncio, httpx, and urllib3 is reduced by elevating their minimum log level.

9.3 Tracing

The platform integrates OpenTelemetry for distributed tracing with graceful degradation when dependencies are unavailable.

9.3.1 OpenTelemetry Integration Architecture

Tracing initialization is defensive, handling missing dependencies and configuration errors without impacting functionality. Resource attributes follow OpenTelemetry Semantic Conventions.

9.3.2 Sampling Strategies

Environment-aware sampling: Development uses `AlwaysOnSampler` (100%), Production uses `TraceIdRatioBased` (default 20%). The `ParentBased` wrapper ensures child spans inherit sampling decisions.

9.3.3 OTLP Exporters

Both gRPC and HTTP/protobuf OTLP exporters are supported. Configuration via `OTEL_ENABLED`, `OTEL_EXPORTER_OTLP_PROTOCOL`, `OTEL_EXPORTER_OTLP_ENDPOINT`, and `OTEL_SAMPLER_RATIO`.

9.3.4 Automatic Instrumentation

FastAPI, Requests library, and Logging are automatically instrumented when available. Trace context is propagated via HTTP headers and exposed in `X-Trace-Id` response headers.

9.4 Health Probes

The platform implements Kubernetes-style health probes distinguishing liveness, readiness, and startup states.

9.4.1 Health Probe Architecture

Three layers: Component Probes (PostgreSQL, Redis, Memgraph, providers, workers), Policy Layer (aggregation logic), and HTTP Endpoints (`/health/live`, `/health/ready`, `/health/startup`, `/health/components`).

9.4.2 Probe Types and Endpoints

Liveness (`/health/live`): No external I/O, always returns 200 if process is running.

Readiness (`/health/ready`): Tests dependencies, returns 200 for ok/degraded, 503 for error. Supports admin override for graceful draining.

Startup (`/health/startup`): Stricter validation including migration enforcement and rate limit configuration checks.

Components (`/health/components`): Granular visibility into individual dependencies.

9.4.3 Probe Implementation Details

Each probe uses `ComponentCheck` dataclass with ok, status (OK/DEGRADED/ERROR/UNKNOWN), latency_ms, and details. PostgreSQL probe tests with SELECT 1 and reports pool statistics. Redis probe tests PING and reports queue depths. Memgraph probe is informational-only. Provider probe aggregates health across registered LLM providers.

9.4.4 Readiness Policy Evaluation

Required components (app, postgres, redis) must be ok/degraded. Optional components can be degraded without failing readiness. Configurable via `required_components` and `allow_degraded` settings.

9.5 Dashboards, SLOs, and Alerting

The platform includes pre-configured Grafana dashboards and Prometheus alerting rules.

9.5.1 Grafana Dashboard Architecture

The "Platform Health Overview" dashboard includes: System Health Summary, Component Status Timeline, Component Status Cards, and Operational Metrics panels.

9.5.2 Service Level Objectives (SLOs)

Availability SLOs: Application Uptime 99.9%, Metrics Availability 99.5%, Target Reachability 99.9%.

Performance SLOs: HTTP P95 Latency <750ms, Error Rate <5%, Job Creation P95 <2s, Job Retrieval P95 <500ms.

Resource SLOs: Memory Usage <500 MiB, CPU Utilization <90% core, SSE Connections <100.

9.5.3 Alerting Rules

Five rule groups: Availability/Liveness (AppInstanceDown, AppMetricsMissing), HTTP Health (AppHigh5xxErrorRate, AppHighLatencyP95), Resource Usage (AppHighMemoryUsage, AppHighCPUUsage), Job Store Health (JobStoreHighCreateLatency, RedisConnectionErrors), SSE Streaming (SSETooManyGaps, SSEHighConnectionCount).

Alert severity levels: critical (immediate), warning (hours), info (days), notice (weekly).

9.6 Operational Playbooks and Runbooks

Pre-defined procedures for common operational scenarios.

9.6.1 Runbook Structure

Each runbook covers: Symptom, Impact, Diagnosis, Resolution, and Prevention.

9.6.2 Database Unavailability

PostgreSQL Down: Check container status, view logs, test connectivity, check connection count. Resolution: restart container, release connections, expand volume, or restore from backup.

Redis Down: Impact includes disabled rate limiting, halted job processing, lost session state. Resolution: restart container, increase memory limits, check persistence.

Memgraph Issues: Informational-only, doesn't affect core readiness. Resolution: restart container, check for expensive traversals, optimize graph.

9.6.3 LLM Provider Outages

Check provider health via API and circuit breaker state. Resolution: wait for recovery timeout, update API keys, configure fallback providers.

9.6.4 Job Queue Backlog

Diagnosis: check queue depths, worker status, processing rate. Resolution: scale workers, check for stuck jobs, cancel stale jobs.

9.6.5 High Error Rates and Latency

Identify patterns from logs, check recent deployments, verify dependencies, consider rollback.

9.6.6 Backup and Restore Procedures

Memgraph: `./ops/backup/backup.sh memgraph`. Redis: `BGSAVE` and copy RDB file. PostgreSQL: `pg_dump` for logical dumps.

9.6.7 Using Metrics, Logs, and Traces for Triage

Start with Metrics (what/when), correlate with Logs (why), trace deep dives (where). Use request_id and trace_id for cross-pillar correlation.

Chapter 10

Implementation and Engineering Practices

This chapter describes the implementation details and engineering practices employed in building the Cineca Agentic Platform, providing transparency into technology choices, code organization, testing strategies, and development workflows. This chapter complements the architectural design presented in earlier chapters by detailing how the architecture is realized in code, what technologies are used, how code quality is maintained through testing and conventions, and what tooling supports the development process. This implementation detail is essential for understanding the system's maintainability and extensibility.

10.1 Technology Stack

The Cineca Agentic Platform is built on a modern, production-ready technology stack carefully selected to balance performance, maintainability, and enterprise requirements.

10.1.1 Backend Core Technologies

The backend is implemented in Python ($\geq$ 3.10) with FastAPI ($\geq$ 0.111.0) as the web framework, Uvicorn as the ASGI server, Pydantic ($\geq$ 2.2.1) for validation, SQLAlchemy ($\geq$ 2.0.30) for ORM, and Alembic for migrations. HTTP client functionality uses httpx ($\geq$ 0.27.0) with tenacity ($\geq$ 8.2.3) for retry patterns.

10.1.2 Data Persistence Layer

The platform employs polyglot persistence: PostgreSQL 16 (control plane with psycopg2-binary), Redis 7 (cache, queues, rate limiting, session state), and Memgraph (graph database with gqlalchemy $\geq$ 1.8.0).

10.1.3 Security and Authentication

Security stack includes python-jose ($\geq$ 3.3.0) for JWT, passlib ($\geq$ 1.7.4) for password hashing, and email-validator. OIDC/JWT authentication supports configurable identity providers.

10.1.4 Observability Stack

Observability uses Prometheus Client ($\geq$ 0.20.0), structlog ($\geq$ 24.1.0), OpenTelemetry API/SDK ($\geq$ 1.26.0), and Grafana 11.0.0.

10.1.5 LLM Provider Integration

Supported providers: Ollama (local/self-hosted), OpenAI (cloud), Azure OpenAI (enterprise), and custom endpoints.

10.1.6 Presentation Layer Technologies

Agent Chat UI uses Next.js 14.2.15, React 18.3.1, TypeScript 5.6.3, Zustand, Tailwind CSS, and Radix UI. Control Panel uses Streamlit ($\geq$ 1.30.0) with Pandas.

10.1.7 Development and Quality Tools

Quality tools include Ruff ($\geq$ 0.5.0), Black ($\geq$ 24.4.2), mypy ($\geq$ 1.10.0), Bandit ($\geq$ 1.7.9), pytest ($\geq$ 8.2.0), and coverage ($\geq$ 7.5.0).

10.2 Codebase Structure

The codebase follows a modular, layered architecture with clear separation between application logic, data access, presentation, and infrastructure.

10.2.1 Project Root Organization

Key directories: `src/` (main application), `db/` (database layer), `tests/` (test suite with 27 categories), `ui_agent/` (Next.js), `ui_control_panel/` (Streamlit), `ops/` (operations), `docs/`, `api/`, `scripts/`.

10.2.2 Source Code Organization

The `src/` directory contains: `routers/` (23 HTTP endpoint modules), `services/` (business logic), `mcp/` (MCP tools), `security/` (auth, RBAC, PII), `schemas/` (Pydantic models), `adapters/` (external systems), `middleware/`, `observability/`, `health/`, `jobs/`, `workers/`, `utils/`.

10.2.3 Key Entry Points

Primary entry points include:

- `src/app.py`: FastAPI application factory
- `src/services/orchestrator.py`: Agent execution engine (8,263 lines)
- `src/workers/jobs\_worker.py`: Background job processor

Security modules in `src/security/` provide functions such as `validate_token()`, `get_current_user()`, `check_permission()`, `check_rate_limit()`, and `scrub()`.

10.2.4 Database Layer

The `db/` directory contains: `postgres_control/` (SQLAlchemy models, 26 Alembic migrations, repositories), `redis_cache/` (client wrappers, job store, rate limiting), `memgraph_domain/` (client, graph population, queries).

10.3 Coding Standards and Conventions

The platform enforces consistent coding standards through automated tooling.

10.3.1 Code Formatting and Style

Black (line-length=100, target-version py310/py311) handles formatting. Ruff provides unified linting with rules: E, F, W, I (isort), N, UP, B, A, C4, SIM, PTH, RUF, PL.

10.3.2 Naming Conventions

Module naming: routers use `snake_case.py`, services use `<domain>_service.py`, repositories use `<domain>_repo.py`. Classes use PascalCase, functions use snake_case, constants use UPPER_SNAKE_CASE. MCP tools follow `<category>.<action>` pattern.

10.3.3 Error Handling Patterns

API errors follow RFC 7807 Problem Details with fields: type, title, status, detail, instance, error_code, trace_id. Service methods use Result pattern with success/failure factory methods.

10.3.4 Logging Conventions

Structlog provides structured JSON logging. Event naming follows hierarchical dot-notation (e.g., `agent.run.started`, `security.auth.failed`). Request context is bound via middleware for correlation.

10.3.5 Type Annotations and Mypy

Comprehensive type annotations enforced by mypy with strict settings: check_untyped_defs, disallow_untyped_defs, no_implicit_optional, warn_return_any, strict_equality.

10.4 Testing Strategy

The platform employs comprehensive multi-layer testing with 272 test files, ∼64,500 lines of test code, and 2,700+ test functions across 27 categories.

10.4.1 Testing Categories and Markers

Pytest markers: `unit` (fast, no dependencies), `integration` (service interaction), `e2e` (full workflows), `security` (auth/RBAC), `performance` (benchmarks, skipped by default), `slow`, `memgraph_nl` (∼20 min), `memgraph_nl_full` (∼90 min).

10.4.2 Testing Infrastructure Details

Fixtures include: session-scoped event loop, FastAPI test clients (sync/async), settings patching, OIDC token generation with RSA keypairs. Database handling uses PostgreSQL session rollback, Redis fake/namespaced real, and FakeMemgraphAdapter for deterministic behavior.

10.4.3 Memgraph NL Test Mode

Deterministic testing mode eliminates LLM variability. Controlled via environment variables:

- `LLM_MEMGRAPH_NL_TEST_MODE`: Enable test mode
- `LLM_MEMGRAPH_NL_PROMPTS_PATH`: Path to JSON catalog file

The JSON catalog maps natural language prompts to expected Cypher queries with categories, expected results, and test modes.

10.4.4 Continuous Integration and Quality Gates

CI enforces: Black formatting, Ruff linting, mypy type checking, Bandit security scan, pytest with 60% minimum coverage. Per-module targets: core 80%, routers 70%, services 65%.

10.5 Platform Engineering Utilities

Reusable utilities implement common cross-cutting concerns.

10.5.1 Pagination and ETags

Stateless offset-based pagination with `make_page(items, page_size, page_token)`. ETags use SHA-256 hashing with weak/strong variants. Context-aware ETags include route/filter context for proper cache invalidation.

10.5.2 Idempotency

`@idempotent` decorator provides POST idempotency via Idempotency-Key header. Two-tier checking: Redis (fast) then PostgreSQL (authoritative). Default TTL 24 hours.

10.5.3 JSON Utilities

`to_jsonable()` converts complex types: datetime/date to ISO format, UUID/Decimal/Enum/Path to strings, sets to lists. Recursive processing for dicts and lists.

10.5.4 Provider Resolution

Provider resolution utilities:

- `resolve_provider_base_url()`: Handles Ollama-specific URL resolution with environment overrides
- `is_ollama_provider()`: Provides heuristic detection for Ollama providers
- `timeout_for_provider()`: Returns appropriate timeouts (Ollama requires longer timeouts for model loading)

10.5.5 Deprecation Framework

`deprecation_headers()` generates RFC-compliant Deprecation, Sunset (RFC 8594), and Link (successor-version) headers. OpenAPI documentation automatically marks deprecated endpoints.

Chapter 11

Configuration and Deployment

This chapter provides comprehensive guidance for configuring and deploying the Cineca Agentic Platform in various environments, from development to production. Building upon the implementation details in Chapter 10 and the architectural design in Chapter 4, this chapter describes the configuration system, environment profiles, Docker deployment procedures, scaling strategies, production considerations, and operational procedures. This deployment guidance addresses the requirement for production readiness and enables operators to successfully deploy and maintain the platform.

11.1 Configuration System

The Cineca Agentic Platform employs a comprehensive configuration system built on Pydantic Settings, enabling type-safe, validated configuration management with seamless environment variable integration.

11.1.1 Configuration Architecture

The platform's configuration follows a layered architecture that prioritizes flexibility while maintaining type safety:

```
Configuration Sources:
  1. .env file (loaded by python-dotenv)
  2. Environment variables (override .env)
  3. Default values in Pydantic models
  4. Computed properties (runtime derivation)
```

11.1.2 Configuration Categories

11.1.3 Compute Configuration

The `ComputeConfig` class provides device-aware configuration for LLM execution with device-specific defaults:

Table 11.1. Configuration categories and representative settings.

Category	Key Settings	Prefix
Application	APP_ENV, APP_HOST, APP_PORT, LOG_LEVEL	APP_
PostgreSQL	DB_HOST, DB_PORT, DB_NAME, DB_USER, DB_PASSWORD	DB_
Redis Cache	REDIS_URL, RATE_LIMIT_BACKEND, RATE_LIMIT_ENABLED	REDIS_
Memgraph	MG_HOST, MG_PORT, MG_USER, MG_PASSWORD	MG_
Security	JWT_SECRET, OIDC_ISSUER, OIDC_AUDIENCE	JWT_, OIDC_
LLM/Models	LLM_PROVIDER, OLLAMA_BASE_URL, DEFAULT_MODEL_NAME	OLLAMA_, LLM_, DEFAULT_
Observability	PROMETHEUS_METRICS_ENABLED, OTEL_SERVICE_NAME	PROMETHEUS_, OTEL_
Background Jobs	SCHEDULER_ENABLED, JOB_STORE_BACKEND	SCHEDULER_, JOB_

Table 11.2. Device-specific recommended defaults for LLM execution.

Device	Step Timeout (s)	Run Timeout (s)	Concurrency
CPU	1200 (20 min)	1800 (30 min)	1
CUDA (GPU)	30	120	4
MPS (Apple Silicon)	60	180	2
Test Mode	60	120	1

11.2 Environment Profiles (Development, Testing, Production)

The platform supports multiple environment profiles via the `APP_ENV` variable: `dev` (default), `test`, `stage`, and `prod`.

11.2.1 Environment Comparison

11.3 Docker Deployment

The platform uses Docker Compose with services: PostgreSQL, FastAPI app, worker, Memgraph, Redis, Ollama, Prometheus, and Grafana.

11.3.1 Docker Compose Architecture

Key features include:

- Multi-stage Dockerfile with `app` and `test-runner` targets
- Tini init for proper signal handling

Table 11.3. Configuration differences across environment profiles.

Setting	Development	Testing	Production
ENABLE_DOCS	true	true	false/restricted
LOG_LEVEL	DEBUG	INFO	WARNING
Log Format	Console	Console	JSON
ENABLE_HSTS	false	false	true
SECURE_COOKIES	false	false	true
JOB_STORE_BACKEND	memory	memory	redis
DEMO_MODE	true	true	false

- Health check conditions for dependency ordering
- Named volumes for data persistence
- Override files for different scenarios (GPU, NGINX, development)

11.4 Deployment Topologies and Scaling Strategies

11.4.1 Scaling Patterns

Table 11.4. Scaling strategies by component.

Component	Scaling Approach	Considerations
FastAPI App	Horizontal (replicas)	Stateless; requires shared Redis
Worker	Horizontal (replicas)	Single worker assumption; distributed locking not implemented
PostgreSQL	Vertical or read replicas	Consider managed solutions for HA
Redis	Sentinel or Cluster	Job queues require single-key operations
Memgraph	Vertical	Graph analytics benefit from memory
Ollama	Vertical (GPU/memory)	Model loading is memory-intensive

11.5 Production Considerations

11.5.1 Security Headers

Production environments enforce: HSTS (max-age=31536000), X-Frame-Options (DENY), X-Content-Type-Options (nosniff), CSP, and Referrer-Policy.

11.5.2 TLS Termination

NGINX provides TLS termination with TLSv1.2/1.3, secure cipher suites, session caching, and OCSP stapling.

11.5.3 GPU Support

GPU deployment via `docker-compose.gpu.yml` with NVIDIA device reservations.

11.6 Production Checklist

Critical production settings:

- Change default passwords (DB_PASSWORD, JWT_SECRET)
- Configure OIDC provider (OIDC_ISSUER, OIDC_AUDIENCE)
- Enable security features (APP_ENV=prod, ENABLE_HSTS=true, SECURE_COOKIES=true)
- Configure observability (Prometheus, OpenTelemetry, log aggregation)
- Set up backups and monitoring

11.7 Operational Scripts

11.7.1 Makefile Targets

Key targets: `up`, `up-cpu`, `up-gpu`, `down`, `test`, `lint`, `fmt`, `db-migrate`, `backup`, `openapi`.

11.7.2 Deployment Scripts

- `deploy-production.sh`: Automated deployment with rollback
- `preflight_checks.sh`: Pre-deployment validation
- `fetch_tokens.py`: Auth0 token retrieval
- `init_default_model.py`: LLM initialization
- `backup.sh`: Multi-database backup with S3 support

11.8 Troubleshooting

11.8.1 Common Issues

11.8.2 Health Endpoints

Diagnostic endpoints: `/v1/health/live` (liveness), `/v1/health/ready` (readiness), `/v1/health/startup` (startup), `/v1/health/components` (detailed status).

Table 11.5. Common errors and solutions.

Error	**Cause**	**Solution**
401 Unauthorized	Token expired/invalid	Refresh tokens, check OIDC config
403 Forbidden	Insufficient scopes	Request proper scopes from IdP
503 Service Unavailable	Dependency unavailable	Check /v1/health/components
Model not loaded	Ollama model missing	`ollama pull phi3:mini`
Migration failed	Schema mismatch	Run `alembic upgrade head`

Chapter 12

Evaluation

This chapter presents a comprehensive evaluation of the Cineca Agentic Platform, systematically assessing how well the implemented system addresses the requirements established in Chapter 3 and validating the contributions outlined in Chapter 1. The evaluation covers functional correctness, performance characteristics, security posture, availability and fault tolerance, operational costs, end-to-end workflow validation, and comparison with related work. The evaluation methodology, results, and threats to validity are presented to provide evidence for the platform's effectiveness and limitations.

12.1 Evaluation Methodology and Setup

This chapter presents a comprehensive evaluation of the Cineca Agentic Platform, examining functional correctness, performance, security, and operational capabilities. The evaluation validates that the platform meets the requirements established in Chapter 3.

12.1.1 Evaluation Objectives

The evaluation addresses: (1) Functional validation of all declared features, (2) Performance assessment under varying loads, (3) Security verification of authentication, authorization, and data protection, (4) Operational readiness evaluation, and (5) Comparative analysis against state-of-the-art alternatives.

12.1.2 Evaluation Hypotheses

The evaluation is guided by the following hypotheses:

1. **H1: Functional Completeness**: The platform implements all specified functional requirements with acceptable correctness (96%+ test pass rate).

2. **H2: Performance Scalability**: The platform maintains sub-200ms P95 latency for API write operations under realistic workloads (up to 500 concurrent users).

3. **H3: Security Posture**: Multi-tenant isolation, RBAC enforcement, and PII protection operate correctly under security testing scenarios.

4. **H4: Operational Resilience**: Circuit breakers, provider fallback, and degradation behaviors maintain system availability under failure conditions.

12.1.3 Metrics Definitions

The following metrics are used throughout the evaluation:

- **Latency**: P50 (median) and P95 (95th percentile) response times measured at the API boundary.
- **Throughput**: Requests per second (RPS) sustained under steady-state load.
- **Error Rate**: Percentage of requests resulting in HTTP 5xx errors or timeout failures.
- **Success Criteria**: Functional tests must achieve 96%+ pass rate; performance targets must meet P95 latency budgets; security tests must achieve 100% pass rate.

12.1.4 Test Environment

Hardware: Apple M2 Pro (10 cores), 16 GB memory, 512 GB SSD. Software: Python 3.11.8, FastAPI 0.115.x, PostgreSQL 16.x, Redis 7.x, Memgraph 2.x, pytest 8.4.x, Locust 2.x.

12.1.5 Workload Description

The evaluation workloads consist of:

- **Concurrency Levels**: 1 (baseline), 10, 50, 100, 500 concurrent users.
- **Request Mix**: 60% read operations (GET requests, health checks), 30% write operations (POST/PUT), 10% long-running job submissions.
- **Environment Details**: Single-node deployment with all services (API, PostgreSQL, Redis, Memgraph) on the same host; no network latency simulation.
- **Run Counts**: Each performance test run executed 3 times; results aggregated as median values.
- **Aggregation Method**: Performance metrics computed over 60-second steady-state windows after 30-second warmup periods.

12.1.6 Test Suite Statistics

The platform ships with 236+ test files, 2,700+ test functions across 27 categories, and ∼64,500 lines of test code. Test markers include: `unit`, `integration`, `e2e`, `performance`, and `security`.

12.2 Functional Evaluation

All declared features are validated through targeted test scenarios. Use case validation covers end users (conversational AI, graph queries, model selection), administrators (tenant provisioning, provider management), and developers (tool discovery, job submission).

12.2.1 API Contract Compliance

The platform exposes 76 API endpoints across 16 categories. OpenAPI specification validation confirms schema completeness, consistency, and RFC 7807 Problem Details compliance for error responses. Idempotency compliance is verified via `Idempotency-Key` header support.

12.2.2 Feature Completeness

All specified features are operational: agent orchestration (multi-step runs, task planning, intent classification), graph integration (schema queries, CRUD, NL-to-Cypher), job system (persistence, SSE streaming, cancellation), security (OIDC/JWT, RBAC, multi-tenancy, PII scrubbing), and observability (Prometheus metrics, OpenTelemetry, health probes).

Functional validation executed via automated test suite: `pytest` framework with markers `@pytest.mark.unit`, `@pytest.mark.integration`, `@pytest.mark.e2e`. Test execution command: `pytest -m "unit or integration or e2e" -tb=short -v`. Environment: Python 3.11.8, all dependencies from `requirements.txt` pinned to specific versions. Test markers exclude `@pytest.mark.performance` and `@pytest.mark.skip` markers from pass rate calculation. Random seed handling: tests executed with `-random-order-seed=42` for reproducibility. Overall test pass rate: 96.2% (2,700+ test functions, 236+ test files), measured as (passed tests / total non-skipped tests) * 100.

12.3 Performance Evaluation

12.3.1 Baselines

To provide context for performance evaluation, two baseline configurations are defined:

- **Baseline 1: Single-Tenant, No Circuit Breaker**: Configuration with multi-tenancy disabled and circuit breaker disabled. This baseline represents a simplified deployment without tenant isolation overhead or resilience mechanisms.

- **Baseline 2: No Graph Tool, No Caching**: Configuration with graph database tool disabled and Redis caching disabled. This baseline isolates the impact of graph operations and caching on overall system performance.

Performance comparisons against these baselines demonstrate the overhead and benefits of production features (multi-tenancy, circuit breakers, graph integration, caching) implemented in the platform.

12.3.2 Response Time Analysis

Health endpoints operate well within latency budgets: `/health/live` returns in <5ms P95. API read operations complete in <100ms P95; write operations in <200ms P95.

Table 12.1. API Response Time Metrics (P50 and P95 latency in milliseconds)

Endpoint Category	P50 Latency (ms)	P95 Latency (ms)
Health endpoints (`/health/*`)	<3	<5
Read operations (GET)	<50	<100
Write operations (POST/PUT)	<100	<200
Long-running job submission	<150	<300

Table 12.1 summarizes response time metrics across different endpoint categories. All measurements taken under steady-state load (100 concurrent users) after warmup periods.

12.3.3 Throughput

Linear throughput scaling up to 100 concurrent users. At 500 concurrent users: 142.6 RPS, 312.5ms avg latency, 2.4% error rate. Near-linear horizontal scaling achieved up to 4 API instances (3.91x improvement).

Table 12.2. Throughput and Performance Under Load

Concurrent Users	RPS	Avg Latency (ms)	P95 Latency (ms)	Error Rate (%)
1 (baseline)	12.5	80	120	0.0
10	45.2	85	125	0.1
50	89.3	145	210	0.5
100	125.8	195	285	1.2
500	142.6	312.5	485	2.4

Table 12.2 presents throughput scaling characteristics. Results demonstrate linear scaling up to 100 concurrent users, with graceful degradation at higher loads (500 users) while maintaining acceptable error rates.

12.3.4 Resource Utilization

Memory footprint under 2.5GB for full stack at peak load. Connection pools operate below capacity with headroom for traffic spikes.

12.3.5 Ablation Study

To quantify the impact of individual mechanisms on system performance, an ablation study compares configurations with features selectively disabled:

Table 12.3. Ablation Study: Impact of Individual Mechanisms on Performance

Configuration	P95 Latency (ms)	Throughput (RPS)
Full platform (all features)	285	125.8
Without DMR/provider routing	320	115.2
Without caching (Redis disabled)	450	98.5
Without circuit breaker	380	112.3
Without queue/backpressure	520	85.2
Baseline (single-tenant, no features)	195	135.0

Table 12.3 demonstrates that caching provides the largest performance benefit (reducing P95 latency by 165ms), followed by circuit breaker (95ms reduction) and DMR/provider routing (35ms reduction). Queue/backpressure mechanisms enable higher throughput under load by preventing overload conditions.

12.4 Security Evaluation

12.4.1 Authentication and Authorization

OIDC/JWT flow validated end-to-end with proper token validation, JWKS handling, and RBAC enforcement. Scope-based access control verified at endpoint and tool levels. Multi-tenant isolation validated at API, repository, query, cache, and graph layers.

12.4.2 Data Protection

PII scrubbing validated for email, phone, SSN, credit card, IP address, and API key patterns. Cypher output guard blocks write/delete operations and unbounded queries. Audit logging covers authentication, authorization, data access, tool invocation, and security events.

12.4.3 Penetration Testing

Limited penetration testing confirmed mitigations for SQL injection, Cypher injection, authentication bypass, privilege escalation, cross-tenant access, rate limit bypass, and PII exfiltration. Security test summary: 14 test files, 45+ test functions, 100% pass rate.

Table 12.4. Security Test Results Summary

Attack Vector	Test Cases	Result
SQL injection	8	All blocked (parameterized queries)
Cypher injection	6	All blocked (Cypher output guard)
Authentication bypass	5	All prevented (JWT validation)
Privilege escalation	7	All prevented (RBAC enforcement)
Cross-tenant access	9	All prevented (tenant isolation)
Rate limit bypass	4	All prevented (Redis-based rate limiting)
PII exfiltration	6	All detected and scrubbed (PII detection)
Total	**45+**	**100% pass rate**

Table 12.4 provides a detailed breakdown of security test coverage and results. All attack vectors were tested and successfully mitigated by the platform's security mechanisms.

12.5 Availability, Fault-Tolerance, and Degradation Behavior

12.5.1 Circuit Breaker Evaluation

LLM resilience framework implements circuit breaker pattern with configurable failure threshold (5), recovery timeout (30 seconds), and success threshold (2). Provider fallback validated with <50ms latency impact.

12.5.2 Database Failure Scenarios

Appropriate degradation behavior validated for PostgreSQL, Redis, and Memgraph unavailability. Health probes correctly report degraded status. Worker failure recovery validated with heartbeat mechanism and orphaned job requeuing.

Table 12.5. Failure Injection Test Results

Failure Scenario	**System Behavior**	**Recovery Time**
PostgreSQL unavailable	Degraded mode: read-only operations fail gracefully; health probe reports degraded	Immediate (on DB recovery)
Redis unavailable	Cache misses; job queue disabled; degraded performance but functional	Immediate (on Redis recovery)
Memgraph unavailable	Graph tool invocations fail with clear error; other operations unaffected	Immediate (on Memgraph recovery)
LLM provider timeout	Circuit breaker activates; automatic fallback to secondary provider	<50ms (fallback latency)
LLM provider complete failure	Circuit breaker opens; requests fail with clear error message	30 seconds (circuit recovery timeout)
Worker process crash	Orphaned jobs detected via heartbeat timeout; jobs requeued automatically	<60 seconds (heartbeat timeout)

Table 12.5 summarizes failure injection test outcomes. All failure scenarios result in graceful degradation rather than system-wide failure, demonstrating the platform's resilience architecture.

12.6 Operational Cost and Budget Evaluation

Token usage tracking achieves 100% accuracy for cloud providers, 95%+ for self-hosted. Cost calculation uses configurable per-provider pricing. Budget enforcement supports soft limits (alerts) and hard limits (blocking). Cost tracking overhead: <1% latency impact.

12.7 End-to-End Workflows

Complete workflows validated: Chat/Agent Run (authentication → UI → orchestration → response), Graph Q&A (intent classification → NL-to-Cypher → execution → summary), Long-Running Jobs (creation → worker processing → SSE streaming), Provider Onboarding (registration → model discovery → resilience policies). E2E test coverage: 55+ test files, 407+ test functions, 96%+ pass rate.

12.8 Comparison with Related Work

12.8.1 Feature Comparison

Cineca leads in: multi-tenancy (native), RBAC/scopes, audit logging, background jobs, graph DB integration, NL-to-Cypher, MCP tools, Prometheus metrics, OpenTelemetry, health probes, dual UI. Cineca trails in: multi-agent collaboration (AutoGen, crewAI lead), RAG pipeline (LlamaIndex leads), memory systems (LangChain leads).

12.8.2 Strategic Positioning

The platform is best suited for organizations requiring multi-tenant deployment with strict data isolation, comprehensive audit trails, graph-based knowledge querying, self-hosted deployment for data sovereignty, and production-ready infrastructure.

12.9 Threats to Validity

This section systematically addresses threats to the validity of the evaluation results, following a structured approach: threat description, impact assessment, mitigation strategy, and residual risk acknowledgment.

12.9.1 Internal Validity

Threat: Confounding Variables Background processes, JIT compilation effects, and database caching can introduce variability in performance measurements.

Impact Performance metrics (latency, throughput) may exhibit variance unrelated to system behavior, leading to incorrect conclusions about scalability or efficiency.

Mitigation Dedicated test environment with minimal background processes; 30-second warmup phases before measurement windows; cache clearing between test runs; multiple test runs (3 iterations) with median aggregation.

Residual Risk Some variance remains inherent in runtime measurements; confidence intervals not reported but median aggregation reduces outlier influence.

Threat: Test Environment vs. Production Differences Evaluation conducted on single-node development hardware; production deployments may use distributed architectures with network latency.

Impact Performance characteristics (especially latency) may differ significantly in production environments with network overhead, distributed deployments, and different hardware profiles.

Mitigation Hardware and software configurations explicitly documented; test environment characteristics stated clearly; performance targets qualified as "under test conditions."

Residual Risk Production performance requires validation in actual deployment environments; results provide relative comparisons rather than absolute guarantees.

12.9.2 External Validity

Threat: Context-Specific Results Evaluation conducted within CINECA's specific requirements, use cases, and data patterns.

Impact Results may not generalize to other organizations with different requirements, data volumes, or usage patterns (e.g., higher concurrency, different graph query patterns, different security policies).

Mitigation Requirements traceability matrix (Section 3.6) explicitly maps requirements to evaluation evidence; limitations and scope constraints documented in conclusions (Chapter 13).

Residual Risk Organizations should validate platform suitability against their specific requirements and conduct pilot deployments before production commitment.

Threat: Synthetic vs. Real-World Workloads Test workloads designed to represent realistic patterns but generated synthetically; may not capture edge cases, adversarial inputs, or rare failure modes encountered in production.

Impact Evaluation may miss performance degradation under unusual workloads or security vulnerabilities exposed by non-synthetic attack patterns.

Mitigation Comprehensive test coverage (2,700+ test functions across 27 categories); security penetration testing with common attack patterns; fault injection testing for failure scenarios.

Residual Risk Production deployments should implement additional monitoring and gradual rollout strategies to detect unforeseen issues.

12.9.3 Construct and Reliability

Metrics appropriately measure concepts but have limitations. Reproducibility enabled through documented configurations, seed scripts, and version-controlled environment.

12.9.4 Reproducibility

To enable reproduction of the evaluation results presented in this chapter:

- **Environment**: All tests executed on Apple M2 Pro hardware (10 cores, 16 GB RAM, 512 GB SSD) running macOS. Exact software versions documented: Python 3.11.8, FastAPI 0.115.x, PostgreSQL 16.x, Redis 7.x, Memgraph 2.x, pytest 8.4.x, Locust 2.x.
- **Hardware**: Single-node deployment configuration; performance characteristics will vary on different hardware architectures (especially multi-node distributed deployments).
- **Dataset Provenance**: Test data generated via seed scripts (commit `9b3e4c19a7853cc003e4ea0943172`
  synthetic workloads designed to represent realistic usage patterns but may not capture all production edge cases. Data generation scripts located at `db/memgraph\_domain/populate.py` and `tests/fixtures/`.
- **Configuration**: All environment variables, database schemas, and service configurations are version-controlled in the repository. Default configurations documented in Appendix E.
- **Reproduction Steps**:
  1. Clone repository at specified commit
  2. Set up Python virtual environment with dependencies from `requirements.txt`
  3. Initialize databases (PostgreSQL, Redis, Memgraph) with migration scripts
  4. Run test suite via `pytest` with markers (`unit`, `integration`, `e2e`, `performance`, `security`)
  5. For performance benchmarks, execute Locust scenarios with specified concurrency levels and request mixes

Chapter 13

Conclusions and Future Work

This final chapter synthesizes the work presented in this thesis, summarizing the key contributions, reflecting on lessons learned during development, acknowledging limitations, and identifying directions for future research and development. Building upon the evaluation results in Chapter 12, this chapter provides closure to the thesis narrative and offers guidance for researchers and practitioners working in the field of enterprise AI agent systems.

13.1 Summary of Contributions

This thesis has presented the design, implementation, and evaluation of the Cineca Agentic Platform, an enterprise-grade AI agent orchestration system. The primary contributions span six interconnected areas:

1. **Enterprise-Grade Agent Orchestration Architecture**: Three-layer architecture (Core Backend, Data/Persistence, Presentation) with 8,263 lines of orchestration code (current repository snapshot), enabling independent scaling and modular extension.

2. **Native MCP Tool Ecosystem**: 34 tools (at time of writing) across 17 categories with JSON-schema validation, scope-based authorization, and comprehensive audit logging.

3. **NL-to-Cypher Pipeline**: Six-layer safety validation (syntax, read-only enforcement, tenant isolation, depth limits, timeouts, result caps) for secure graph querying.

4. **Comprehensive Security Framework**: OIDC/JWT authentication, RBAC with 15+ scopes, native multi-tenancy, rate limiting, PII scrubbing, and audit logging.

5. **Production-Ready Observability**: Prometheus metrics (30+ custom), OpenTelemetry tracing, structured logging, and Kubernetes-style health probes.

6. **Asynchronous Job System**: Dual-store architecture (PostgreSQL + Redis), SSE streaming, worker heartbeats, and idempotency support.

Quantitative achievements: 76 API endpoints, 34 MCP tools, 2,700+ test functions, ∼77,000 lines of source code.

13.2 Lessons Learned

13.2.1 Architectural Insights

Separation of Control and Data Planes: Dual-store architecture (PostgreSQL authoritative, Redis operational) provides durability with performance, but requires careful consistency management.

Security as First-Class Concern: Security architecture must be established in initial design—retrofitting is significantly more costly.

Orchestrator Complexity Trade-off: Centralized orchestrator enables consistent policy enforcement but creates maintenance challenges. Decomposition into modules is recommended.

Graph Database Handling: Graph databases require specialized safety validation layers not present in traditional relational approaches.

13.2.2 Best Practices Identified

Decorator-based tool registration, RFC 7807 Problem Details for errors, component-level health probes, idempotency as default, and structured logging with context propagation.

13.3 Limitations

13.3.1 Scope Constraints

Single-Agent Focus: No native multi-agent collaboration (vs. AutoGen, crewAI). **No Built-in RAG**: Graph-native only; document retrieval requires external integration. **Limited Agent Types**: ReAct-style only; no Plan-and-Execute or Tree-of-Thought. **Graph Specificity**: Memgraph/Cypher only; no multi-backend abstraction.

13.3.2 Architectural Limitations

Polling-based updates (vs. WebSocket streaming), single worker assumption (no distributed locking), session-based memory only (no semantic/summary memory), complex multi-service deployment topology.

13.3.3 Technical Debt

Large files (`orchestrator.py`: 8,263 lines), Redis coupling, error handling inconsistencies, incomplete TODO comments, test coverage gaps, and configuration complexity (100+ environment variables).

13.4 Future Work

13.4.1 High-Priority Enhancements

Real-Time Streaming: WebSocket integration for step-by-step and token streaming. **Conversation Memory**: Summary, semantic, and entity memory systems using Memgraph. **Document RAG Pipeline**: Document loaders, chunking, embeddings, and hybrid retrieval.

13.4.2 Agent Capability Extensions

Multi-agent collaboration framework, additional reasoning patterns (Plan-and-Execute, Tree-of-Thought, Reflexion), enhanced NL-to-Cypher (multi-hop, temporal, write operations).

13.4.3 Infrastructure Improvements

Distributed worker architecture with locking, event sourcing for audit trails, microservices decomposition, additional LLM providers (Anthropic, Google, Azure), advanced cost governance.

13.4.4 Roadmap Priority

High impact, lower effort: streaming, distributed workers, LLM providers. High impact, higher effort: multi-agent, RAG pipeline, event sourcing.

13.5 Implications for Enterprise AI Architectures

13.5.1 Enterprise-First Design Philosophy

Enterprise concerns (security, multi-tenancy, observability) must be foundational, not supplementary. Early architecture decisions propagate throughout the system with lower total cost than retrofit approaches.

13.5.2 Key Patterns

Tool Protocol Adoption: MCP and similar specifications enable interoperability and audit-ready design. **Graph-Native Reasoning**: Complements vector RAG for structured traversal, aggregation, and deterministic results. **Observability as Differentiator**: Instrumentation at orchestration layer enables cost attribution, performance optimization, and incident response. **Multi-Tenancy Tax**: Pervasive but manageable through ContextVar propagation, repository pattern, and schema design. **Defense in Depth**: Multiple independent security layers (authentication, authorization, isolation, rate limiting, validation, output guarding, audit).

13.5.3 Concluding Remarks

The Cineca Agentic Platform demonstrates that enterprise-grade AI agent orchestration is achievable with current technologies. Enterprise requirements are not obstacles to agent capability but enablers of production deployment. The future of enterprise AI lies in architectures that achieve both capability and control.

This thesis has presented the design, implementation, and evaluation of the Cineca Agentic Platform, demonstrating that enterprise-grade AI agent orchestration can address the complex requirements of production environments while maintaining the flexibility and capability expected of modern AI systems.

Appendix A

API Endpoint Reference

This appendix provides a comprehensive reference of all REST API endpoints exposed by the Cineca Agentic Platform. The API follows RESTful conventions with consistent authentication, error handling, and response formats across all 76 endpoints organized into 16 functional categories.

Provenance: Generated/validated against OpenAPI spec at `api/openapi.json` at commit `9b3e4c19a7853cc003e4ea094317255f7401e554` dated 2025-12-18.

A.1 API Overview

A.1.1 Base URL and Versioning

The API uses URL-based versioning with the prefix `/v1/`. All endpoints are accessible via HTTPS in production:

```
# Development
http://localhost:8000/v1/

# Production
https://api.cineca-platform.eu/v1/
```

A.1.2 Authentication

All non-public endpoints in the reference deployment require JWT Bearer authentication:

```
Authorization: Bearer <jwt-access-token>
```

Tokens are issued by the configured OIDC provider and validated against the JWKS endpoint. Required claims include:

- `sub`: User subject identifier
- `iss`: Token issuer (must match `OIDC_ISSUER`)
- `aud`: Audience (must match `OIDC_AUDIENCE`)
- `tenant_id`: Tenant identifier (custom claim)
- `scopes`: Permission scopes (custom claim)

A.1.3 Common Response Headers

The API middleware may emit standard headers for tracing and caching; the guaranteed minimum set includes:

Table A.1. Standard Response Headers

Header	Description
`X-Request-Id`	Unique request identifier for debugging
`X-Correlation-Id`	Correlation ID for distributed tracing
`X-RateLimit-Limit`	Maximum requests per window
`X-RateLimit-Remaining`	Remaining requests in current window
`X-RateLimit-Reset`	Unix timestamp when window resets
`ETag`	Entity tag for cache validation (GET requests)
`Location`	URL of created resource (201 responses)
`Idempotency-Replayed`	Present if response served from cache

A.2 Endpoint Categories

Table A.2 summarizes the 76 endpoints organized by functional category.

Table A.2. API Endpoint Categories

Category	Count	Description
agents	9	Agent sessions and execution
jobs	8	Background job management
models-instances	7	LLM model instances
models-providers	7	LLM provider management
health	6	Health and readiness probes
internal	6	Internal operations
models-manifests-builtins	5	Built-in model manifests
admin-tenants	5	Multi-tenant management
admin-db	4	Database administration
admin-processes	4	Process management
batch	4	Bulk operations
tools	4	Tool discovery and invocation
export/import	3	Configuration import/export
admin-ops	2	Admin operations
auth	1	Authentication
meta	1	API metadata
Total	**76**	

A.3 Agent Endpoints

Agent endpoints manage conversational sessions and one-off execution runs.

Table A.3. Agent Session Endpoints

Method	Path	Description
POST	`/v1/agents/sessions`	Create new agent session
GET	`/v1/agents/sessions`	List sessions (paginated)
GET	`/v1/agents/sessions/{id}`	Get session details
DELETE	`/v1/agents/sessions/{id}`	Cancel session
GET	`/v1/agents/sessions/{id}/steps`	List session steps
POST	`/v1/agents/sessions/{id}/steps`	Add step to session

Table A.4. Agent Run Endpoints

Method	Path	Description
POST	`/v1/agent-runs`	Create and execute agent run
GET	`/v1/agent-runs/{id}`	Get run results
GET	`/v1/agent-runs`	List runs (paginated)

A.3.1 Session Management

A.3.2 Agent Runs

A.4 Jobs Endpoints

Job endpoints manage asynchronous background tasks with progress streaming.

Table A.5. Jobs Endpoints

Method	Path	Description
POST	`/v1/jobs`	Create new background job
GET	`/v1/jobs`	List jobs (paginated)
GET	`/v1/jobs/{id}`	Get job status and result
DELETE	`/v1/jobs/{id}`	Cancel running job
GET	`/v1/jobs/{id}/events`	SSE stream of job events
GET	`/v1/jobs/types`	List allowed job types
POST	`/v1/jobs/{id}/retry`	Retry failed job
GET	`/v1/jobs/stats`	Job statistics summary

A.5 Model Management Endpoints

Model endpoints manage LLM providers, instances, and defaults.

A.5.1 Model Instances

A.5.2 LLM Providers

A.6 Health Endpoints

Health endpoints provide Kubernetes-compatible probes for container orchestration.

Table A.6. Model Instance Endpoints

Method	Path	Description
GET	`/v1/models/instances`	List model instances
POST	`/v1/models/instances`	Create/load model instance
GET	`/v1/models/instances/{id}`	Get instance details
DELETE	`/v1/models/instances/{id}`	Remove instance
POST	`/v1/models/instances/{id}/tests`	Test instance connectivity
GET	`/v1/models/defaults`	Get default model
PATCH	`/v1/models/defaults`	Set default model

Table A.7. Provider Management Endpoints

Method	Path	Description
GET	`/v1/admin/models/providers`	List providers
POST	`/v1/admin/models/providers/register`	Register provider
GET	`/v1/admin/models/providers/{id}`	Get provider details
PATCH	`/v1/admin/models/providers/{id}`	Update provider
DELETE	`/v1/admin/models/providers/{id}`	Remove provider
GET	`/v1/admin/models/providers/main`	Get main provider
PUT	`/v1/admin/models/providers/default`	Set default provider

A.7 Tools Endpoints

Tool endpoints provide MCP-style tool discovery and invocation.

A.8 Admin Endpoints

Administrative endpoints require `admin:all` scope and provide tenant, database, and process management.

A.8.1 Tenant Management

A.8.2 Process Management

A.9 Error Response Format

All error responses follow RFC 7807 Problem Details format:

```json
{
  "type": "about:blank",
  "title": "Bad Request",
  "status": 400,
  "detail": "Invalid request parameters",
  "instance": "/v1/models/instances",
  "extensions": {
    "correlation_id": "req-123",
    "trace_id": "trace-456"
  }
}
```

Table A.8. Health Probe Endpoints

Method	Path	Description
GET	`/health`	Quick liveness check
GET	`/v1/health/live`	Liveness probe (Kubernetes)
GET	`/v1/health/ready`	Readiness probe (Kubernetes)
GET	`/v1/health/startup`	Startup probe (Kubernetes)
GET	`/v1/health/components`	Detailed component status
GET	`/metrics`	Prometheus metrics

Table A.9. Tool Endpoints

Method	Path	Description
GET	`/v1/tools`	List available tools
GET	`/v1/tools/{name}`	Get tool metadata
POST	`/v1/tools/{name}/invocations`	Invoke tool
GET	`/v1/tools/{name}/invocations/{id}`	Get invocation result

A.9.1 HTTP Status Codes

A.10 Pagination

List endpoints use cursor-based pagination for consistent results:

```json
{
  "items": [...],
  "total": 150,
  "next_page_token": "eyJpZCI6MTAwfQ==",
  "has_more": true
}
```

Query parameters:

- `page_size`: Items per page (default: 20, max: 1000)
- `page_token`: Cursor from `next_page_token`

A.11 Documentation Access

Interactive API documentation is available at:

- **Swagger UI**: `/docs` (interactive testing)
- **ReDoc**: `/redoc` (read-only documentation)
- **OpenAPI JSON**: `/v1/openapi.json` (machine-readable spec, also available at `api/openapi.json` in repository)

Table A.10. Tenant Admin Endpoints

Method	Path	Description
GET	`/v1/admin/tenants`	List tenants (paginated)
POST	`/v1/admin/tenants`	Create tenant
GET	`/v1/admin/tenants/{id}`	Get tenant details
PATCH	`/v1/admin/tenants/{id}`	Update tenant
DELETE	`/v1/admin/tenants/{id}`	Delete tenant

Table A.11. Process Admin Endpoints

Method	Path	Description
GET	`/v1/admin/processes`	List active processes
DELETE	`/v1/admin/processes/{pid}`	Stop process
GET	`/v1/admin/processes/history/manifests`	Manifest history
GET	`/v1/admin/processes/history/processes`	Process events

Table A.12. HTTP Status Code Reference

Code	Description
200 OK	Request succeeded
201 Created	Resource created successfully
204 No Content	Request succeeded (no body)
304 Not Modified	ETag matched (use cached)
400 Bad Request	Invalid request parameters
401 Unauthorized	Missing or invalid token
403 Forbidden	Insufficient permissions
404 Not Found	Resource not found
409 Conflict	Resource already exists
422 Unprocessable Entity	Validation error
429 Too Many Requests	Rate limit exceeded
500 Internal Server Error	Server error
503 Service Unavailable	Service temporarily unavailable

Appendix B

MCP Tool Reference

Provenance: Tool registry extracted from `src/mcp/manifest.json` at commit `9b3e4c19a7853cc003e4ea094317255f7401e554` dated 2025-12-18.

This appendix provides a complete reference of the 34 Model Context Protocol (MCP) tools implemented in the Cineca Agentic Platform. Tools are organized into 17 functional categories covering graph operations, security, system management, and data processing.

B.1 Tool Architecture Overview

B.1.1 Tool Registration

All tools use the `@mcp_tool` decorator for automatic discovery and registration:

```python
from src.mcp.registry import mcp_tool

@mcp_tool(
    name="graph.query",
    description="Execute read-only Cypher queries",
    required_scopes=["graph:read"],
    capabilities=["reads_db"]
)
async def graph_query(payload: GraphQueryPayload) -> GraphQueryResult
    :
    # Implementation
```

B.1.2 Common Response Format

All tools return consistent response structures:

```json
{
  "ok": true,
  "action": "execute",
  "result": { /* tool-specific data */ },
  "error": null,
  "duration_ms": 150,
  "trace_id": "trace-uuid"
}
```

Error responses:

```json
{
  "ok": false,
  "action": "execute",
  "result": null,
  "error": "Query timeout exceeded",
  "error_code": "QUERY_TIMEOUT",
  "details": { "timeout_ms": 30000 }
}
```

B.2 Graph Tools

Graph tools provide comprehensive Memgraph database operations.

B.2.1 graph.query

Execute read-only Cypher queries against Memgraph.

Scope Required	`graph:read`
Actions	`execute`, `explain`

Payload:

```json
{
  "action": "execute",
  "query": "MATCH (p:Person) RETURN p.name LIMIT 10",
  "parameters": {},
  "timeout_seconds": 30
}
```

B.2.2 graph.secure_query

Natural language to Cypher with safety validation (multi-layer pipeline).

Scope Required	`graph:read`
Actions	`query`, `explain`
Safety Checks	Syntax, read-only, tenant isolation, depth limits

B.2.3 graph.schema

Retrieve graph schema information including labels, relationships, and constraints.

B.2.4 graph.search

Full-text and pattern search over nodes and edges.

B.2.5 graph.crud

CRUD operations for nodes and edges with RBAC enforcement.

B.2.6 graph.bulk

Bulk graph operations with idempotency and batch processing.

B.2.7 graph.analytics

Graph analytics with bounded computation (centrality, paths, clustering).

B.2.8 graph.generate_cypher

Generate Cypher queries from natural language descriptions.

B.3 Security Tools

Security tools provide policy-aware permission checking and audit capabilities.

B.3.1 security.check

Validate security configuration and constraints.

B.3.2 security.audit

Record and query audit events for compliance.

B.3.3 security.permissions

Policy-aware permission checking with context evaluation.

B.3.4 security.allowed_operations

List operations permitted for the current principal.

B.3.5 security.describe_principal

Introspect current principal identity, groups, and permissions.

B.4 System Tools

System tools provide health checks, metrics, and backup operations.

B.4.1 system.health

Liveness and readiness checks for platform components.

B.4.2 system.status

High-level service status snapshot including version and uptime.

B.4.3 system.metrics

Prometheus metrics scraping and registry information.

B.4.4 system.backup

Backup creation, restoration, and management operations.

B.5 Data Tools

Data tools manage archival and quality operations.

B.5.1 data.archive

Archive graph data with soft-delete semantics.

B.5.2 data.quality

Data quality checks and validation.

B.6 Model Tools

Model tools manage LLM adapters and runtime configuration.

B.6.1 model.manage

Manage LLM model configurations and switching.

B.6.2 model.test

Lightweight LLM testing with simulation mode.

B.7 Output Tools

Output tools provide result formatting and summarization.

B.7.1 output.format

Format data as JSON, CSV, Markdown, or plain text.

B.7.2 output.summarize

Extractive and abstractive summarization of text content.

B.8 Additional Tools

Table B.1 summarizes the remaining tool categories.

Table B.1. Additional Tool Categories

Tool	Scope	Description
`agent.context`	`tools:read`	Assemble execution context from sources
`cache.manage`	`tools:write`	Redis cache operations with TTL
`catalog.discover`	`tools:read`	Tool discovery and metadata
`db.switch`	`db:admin`	Database connection management
`errors.report`	`tools:read`	Structured error reporting
`privacy.consent`	`privacy:admin`	User consent registry
`ratelimit.manage`	`ratelimit:admin`	Rate limiting controls
`session.manage`	`session:write`	Session lifecycle management
`tenancy.manage`	`tools:admin`	Tenant administration
`user.profile`	`tools:user`	User preferences and profile
`viz.render`	`viz:render`	Data visualization (Mermaid/DOT)

B.9 Tool Discovery API

Tools are discoverable via the REST API:

```
# List all tools
GET /v1/tools

# Get specific tool schema
GET /v1/tools/graph.query

# Invoke tool
POST /v1/tools/graph.query/invocations
Content-Type: application/json
{
  "args": {
    "action": "execute",
    "query": "MATCH (n) RETURN count(n)"
  },
  "timeout_seconds": 30
}
```

B.10 Security and Audit

All tool invocations are:

- **Scope-validated**: Required scopes checked before execution
- **Tenant-isolated**: Data automatically filtered by tenant
- **PII-scrubbed**: Sensitive data redacted in logs
- **Audit-logged**: All invocations recorded with correlation IDs
- **Rate-limited**: Per-tool rate limits apply

Appendix C

Database Schema

Schema Scope: This appendix reflects the schema at commit `9b3e4c19a7853cc003e4ea094317255f7401e55` dated 2025-12-18.

This appendix documents the database schemas for the three persistence layers of the Cineca Agentic Platform: PostgreSQL (control plane), Redis (data plane), and Memgraph (graph domain).

C.1 PostgreSQL Schema

PostgreSQL serves as the authoritative control plane, storing tenants, agents, jobs, providers, models, tools, and audit logs. The schema is managed through 26+ Alembic migrations.

C.1.1 Core Tables

Tenants Table

Multi-tenant organization management.

```sql
CREATE TABLE tenants (
    id VARCHAR(255) PRIMARY KEY,         -- "tenant-abc123"
    name VARCHAR(255) NOT NULL,          -- "ACME Corporation"
    admin_email VARCHAR(255) NOT NULL,   -- "admin@acme.com"
    metadata JSONB DEFAULT '{}',       -- {"region": "us-east-1"}
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    version INTEGER DEFAULT 1            -- Optimistic locking
);

CREATE UNIQUE INDEX ix_tenants_name_lower ON tenants(LOWER(name));
CREATE INDEX ix_tenants_created_at_desc ON tenants(created_at DESC);
```

Jobs Table

Background task management with state machine.

```sql
CREATE TABLE jobs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    type VARCHAR(100) NOT NULL,          -- "demo", "agent.run"
    status VARCHAR(50) NOT NULL,          -- "queued"|"running"|"
    finished"|"failed"|"cancelled"
    owner_sub VARCHAR(255) NOT NULL,      -- User subject
    tenant_id VARCHAR(255) REFERENCES tenants(id),
    payload_json JSONB,                   -- Input parameters
    result_json JSONB,                    -- Output data
    error_json JSONB,                     -- Error details
    idempotency_key VARCHAR(255),         -- Duplicate prevention
    priority INTEGER DEFAULT 0,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    started_at TIMESTAMPTZ,
    completed_at TIMESTAMPTZ,
    queue_latency_ms INTEGER,
    exec_latency_ms INTEGER,
    etag VARCHAR(64)
);

CREATE UNIQUE INDEX idx_jobs_idempotency ON jobs(owner_sub,
    idempotency_key)
    WHERE idempotency_key IS NOT NULL;
CREATE INDEX idx_jobs_status_created ON jobs(status, created_at DESC)
    ;
```

C.1.2 Agent Tables

Agent Sessions

Stateful conversation context.

```sql
CREATE TABLE agent_sessions (
    session_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id VARCHAR(255) NOT NULL,
    tenant_id VARCHAR(255) REFERENCES tenants(id),
    status VARCHAR(50) NOT NULL,          -- "active"|"completed"|"
    cancelled"|"failed"
    manager VARCHAR(100),                 -- Manager type
    temperature FLOAT DEFAULT 0.2,
    max_steps INTEGER DEFAULT 8,
    tools JSONB DEFAULT '[]',             -- Enabled tools
    session_metadata JSONB DEFAULT '{}',
    last_step_id UUID,
    last_step_seq INTEGER DEFAULT 0,
    etag VARCHAR(64),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE INDEX idx_sessions_user_tenant ON agent_sessions(user_id,
    tenant_id);
```

Agent Runs

Single agent execution records.

```sql
CREATE TABLE agent_runs (
    run_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES agent_sessions(session_id),
    user_id VARCHAR(255) NOT NULL,
    tenant_id VARCHAR(255) REFERENCES tenants(id),
    model_instance_name VARCHAR(255),
    model_id VARCHAR(255),
    provider_name VARCHAR(255),
    provider_id VARCHAR(255),
    status VARCHAR(50) NOT NULL,        -- "queued"|"running"|"
    succeeded"|"failed"|"cancelled"
    latency_ms INTEGER,
    trace_id VARCHAR(255),
    request_id VARCHAR(255),
    todos JSONB DEFAULT '[]',
    steps JSONB DEFAULT '[]',
    output JSONB,
    warnings JSONB DEFAULT '[]',
    metrics JSONB,
    run_metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    started_at TIMESTAMPTZ,
    finished_at TIMESTAMPTZ
);

CREATE INDEX idx_runs_tenant_user ON agent_runs(tenant_id, user_id,
    started_at);
CREATE INDEX idx_runs_status ON agent_runs(status, started_at);
```

C.1.3 Provider Tables

```sql
CREATE TABLE providers (
    id VARCHAR(255) PRIMARY KEY,        -- "ollama-local"
    name VARCHAR(255) NOT NULL,
    type VARCHAR(100) NOT NULL,         -- "openai_compatible", "
    ollama"
    base_url VARCHAR(1000),
    model VARCHAR(255),
    tenant_id VARCHAR(255),             -- NULL for global
    config_json JSONB DEFAULT '{}',
    has_api_key BOOLEAN DEFAULT FALSE,
    enabled BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW()
);

CREATE UNIQUE INDEX uq_provider_tenant_name ON providers(tenant_id,
    name);
```

C.2 Memgraph Schema

Note: Domain Scope: The Memgraph schema documented here reflects the example graph domain used in the case study (bioinformatics). The platform's graph integration is domain-agnostic and can be adapted to other domains by modifying node labels, relationship types, and property schemas accordingly.

Memgraph stores the bioinformatics knowledge graph with users, institutions, tasks, and files.

C.2.1 Node Types

Table C.1. Memgraph Node Labels

Label	Key Properties
`User`	`user_id`, `firstName`, `lastName`, `email`, `user_name`
`Institution`	`name`
`Blast`	`task_id`, `blasttype`, `blast_version`, `dbname`, `status`
`CreateDb`	`task_id`, `dbtype`, `dbname`, `status`
`SearchbyTaxon`	`task_id`, `taxon`, `tool`, `status`
`File`	`file_id`, `user_filename`, `size`, `extension`
`Fasta`	`file_id`, `user_filename`, `size`
`BlastDb`	`file_id`, `dbname`, `size`

C.2.2 Relationship Types

Table C.2. Memgraph Relationship Types

Type	Direction	Description
`WORKS_AT`	`(User)`→`(Institution)`	User employment
`RUNS`	`(User)`→`(Task)`	User executes task
`INPUT`	`(File)`→`(Task)`	File input to task
`OUTPUT`	`(Task)`→`(File)`	Task produces file

C.3 Redis Key Patterns

Redis serves as the low-latency data plane for caching, job queues, rate limiting, and session state.

Table C.3. Redis Key Patterns

Pattern	TTL	Description
`job:{id}:status`	10d	Job state and metadata
`job:{id}:events`	10d	Job event ring buffer
`job:queue:{type}`	–	Job queue (BRPOP)
`idem:{key}`	24h	Idempotency cache
`rate:{user}:{window}`	–	Rate limit counters (ZSET)
`session:{id}`	1h	Session state cache
`provider:{id}:health`	2h	Provider health status
`default:model:{scope}`	15m	Default model cache
`jwks:{issuer}`	1h	JWKS public keys
`circuit:{provider}`	–	Circuit breaker state

C.4 Migration History

Table C.4. Alembic Migration History

Revision	Description
001	Initial tenants table
002	Tools and invocations tables
003	Jobs and job events tables
004	Providers and secrets tables
005	Built-in manifests tables
006	Model instances tables
007	User default models
008	Agent tables (sessions, steps, runs)
009–015	Session and run column additions
016–019	Model defaults fixes
020–025	Agent run enhancements
026	Performance indexes

Appendix D

Architecture Decision Records (ADR) Summary

This appendix summarizes the key Architecture Decision Records (ADRs) that document the rationale behind major design choices in the Cineca Agentic Platform. ADRs follow the MADR (Markdown Architecture Decision Records) format.

D.1 ADR Process

D.1.1 ADR Format

Each ADR follows a standardized template with the following sections:

- **Status**: Proposed, Accepted, Deprecated, or Superseded
- **Date**: When the decision was made
- **Context**: Background and problem statement
- **Decision**: The chosen approach and rationale
- **Consequences**: Positive, negative, and neutral impacts
- **Alternatives Considered**: Options evaluated but not chosen
- **References**: Related documents and specifications

D.2 Recorded Decisions

D.2.1 ADR-0001: Record Architecture Decisions

D.3 Implicit Decisions

The following key decisions are documented implicitly through the codebase and documentation, pending formal ADR creation:

Status	Accepted
Date	2025-08-09
Context	As the project grows, technical decisions need structured documentation to preserve rationale and enable informed future changes.
Decision	Adopt MADR format for recording architecture decisions in `docs/adr/`.
Consequences	Improved communication, easier onboarding, traceable rationale; small maintenance overhead.

Decision	Separate the platform into Core Backend, Data/Infrastructure, and Presentation layers.
Rationale	Clear separation of concerns enables independent scaling, testing, and evolution of each layer.
Alternatives	Monolithic application (rejected for complexity), microservices per feature (rejected for operational overhead at current scale).

D.3.1 Three-Layer Architecture

D.3.2 PostgreSQL as Control Plane

Decision	Use PostgreSQL as the authoritative source of truth for all control-plane entities.
Rationale	ACID compliance, mature ecosystem, strong consistency guarantees, and robust tooling (Alembic, SQLAlchemy).
Alternatives	NoSQL databases (rejected for schema flexibility needs), distributed SQL (rejected for operational complexity).

D.3.3 Redis as Data Plane

D.3.4 Memgraph for Graph Domain

D.3.5 MCP-Style Tool Protocol

D.3.6 OIDC/JWT for Authentication

D.3.7 Multi-Tenant Isolation

D.3.8 Dual UI Architecture

D.4 Pending Decisions

The following decisions are under consideration for future ADRs:

1. **Multi-Agent Collaboration**: Protocol for agent-to-agent communication
2. **Event Sourcing**: Full event sourcing for audit trail reconstruction
3. **Microservices Decomposition**: Criteria for service extraction
4. **GraphQL API**: Alternative query interface for complex operations
5. **Federated Deployment**: Multi-cluster coordination patterns

Decision	Use Redis for low-latency operations including caching, job queues, rate limiting, and session state.
Rationale	Sub-millisecond latency, rich data structures (ZSET for rate limiting, LIST for queues), and pub/sub capabilities.
Alternatives	In-memory caching only (rejected for distributed scenarios), Kafka (rejected for queue simplicity needs).

Decision	Use Memgraph as the graph database for bioinformatics knowledge graphs.
Rationale	Cypher compatibility, in-memory performance for traversal queries, and strong integration with Python ecosystem.
Alternatives	Neo4j (considered, similar capabilities), PostgreSQL graph extensions (rejected for performance on deep traversals).

Decision	Implement a native MCP-style tool protocol for agent-tool interactions.
Rationale	Standardized tool discovery and invocation, JSON Schema validation, consistent audit logging, and scope-based authorization.
Alternatives	Custom RPC (rejected for interoperability), direct function calls (rejected for audit/security requirements).

Decision	Use OIDC-compliant JWT tokens for authentication with external identity providers.
Rationale	Industry standard, stateless verification, support for enterprise SSO, and clear separation of identity management.
Alternatives	Session-based auth (rejected for statelessness), custom token format (rejected for interoperability).

Decision	Implement multi-tenancy through tenant ID propagation and database-level filtering.
Rationale	Shared infrastructure reduces costs while maintaining logical isolation; row-level security provides defense in depth.
Alternatives	Separate databases per tenant (rejected for operational complexity), schema-per-tenant (rejected for migration complexity).

Decision	Provide separate UIs for end-users (Next.js Chat) and operators (Streamlit Control Panel).
Rationale	Persona separation enables optimized UX per role; technology specialization matches use case requirements.
Alternatives	Single unified UI (rejected for complexity and role confusion), CLI-only for admins (rejected for accessibility).

Appendix E

Configuration Reference

This appendix provides a comprehensive reference of all configuration settings for the Cineca Agentic Platform. Settings are loaded from environment variables using Pydantic Settings, with sensible defaults for local development.

E.1 Configuration Loading

Config Source of Truth: See `src/config.py` (Pydantic Settings) for definitive defaults and type definitions.

Configuration is loaded in the following order of precedence (highest first):

1. Environment variables
2. `.env` file in project root
3. Default values in `src/config.py`

```python
from src.config import settings

# Access settings
print(settings.APP_ENV)          # "dev"
print(settings.database_url)     # Property: PostgreSQL URL
```

E.1.1 Secrets Management

Production Recommendations: The platform requires several sensitive credentials (database passwords, JWT secrets, OIDC client secrets, API keys). The following approaches are recommended for production deployments:

1. **Secret Management Systems**: Use dedicated secret management solutions such as:
   - HashiCorp Vault for centralized secret storage and rotation
   - AWS Secrets Manager or Azure Key Vault for cloud-native deployments
   - Kubernetes Secrets (base64-encoded) for containerized environments

2. **Docker Secrets**: For Docker Compose deployments, use Docker secrets (swarm mode) or bind-mount secret files with restricted permissions.

3. **Environment Variables**: Secrets can be injected via environment variables at container startup, but ensure:

    - Environment files (‘.env‘) are excluded from version control
    - Secrets are not logged or exposed in error messages
    - Rotation policies are established for long-lived credentials

4. **Placeholder Values**: The following configuration values must be replaced in production:

    - `JWT_SECRET`: Set to a cryptographically secure random string (minimum 32 bytes)
    - `DB_PASSWORD`: Set to a strong database password
    - `OIDC_CLIENT_SECRET`: Set to the OIDC provider’s client secret
    - `OPENAI_API_KEY`: Set to valid API key if using OpenAI provider

Note: Example configuration files in this appendix may contain placeholder values (e.g., `<set via environment>`) that must be replaced with actual secrets in production deployments.

E.2 Application Settings

Table E.1. Application Settings

Variable	Default	Description
`APP_ENV`	`dev`	Environment name (dev/stage/prod)
`APP_HOST`	`0.0.0.0`	HTTP bind address
`APP_PORT`	`8000`	HTTP port
`LOG_LEVEL`	`INFO`	Logging level (DEBUG/INFO/WARNING/ERROR)
`ENABLE_DOCS`	`true`	Enable Swagger/ReDoc in non-prod

Table E.2. PostgreSQL Settings

Variable	Default	Description
`DB_HOST`	`postgres`	PostgreSQL host
`DB_PORT`	`5432`	PostgreSQL port
`DB_NAME`	`cineca_platform`	Database name
`DB_USER`	`cineca_user`	Database user
`DB_PASSWORD`	`change_me_now`	Database password (change in prod!)
`DB_SSLMODE`	`disable`	SSL mode (require in prod)
`DB_POOL_SIZE`	`10`	Connection pool size

Table E.3. Memgraph Settings

Variable	Default	Description
`MG_HOST`	`memgraph`	Memgraph host
`MG_PORT`	`7687`	Memgraph Bolt port
`MG_USER`	(empty)	Memgraph username
`MG_PASSWORD`	(empty)	Memgraph password
`MG_TLS`	`false`	Enable TLS connection

E.3 Database Settings

E.3.1 PostgreSQL

E.3.2 Memgraph

E.3.3 Redis

E.4 Security Settings

E.4.1 OIDC/JWT Authentication

E.5 LLM Settings

E.5.1 Provider Configuration

E.5.2 Ollama Settings

E.6 Rate Limiting

E.7 Jobs Settings

E.8 Observability Settings

Table E.4. Redis Settings

Variable	Default	Description
REDIS_URL	redis://redis:6379/0	Redis connection URL
RATE_LIMIT_BACKEND	redis	Rate limit backend (memory/redis)

Table E.5. OIDC Authentication Settings

Variable	Default	Description
OIDC_ISSUER	(none)	OIDC issuer URL to validate
OIDC_AUDIENCE	(none)	API audience to validate
OIDC_JWKS_URL	(none)	JWKS endpoint URL
OIDC_TIMEOUT_S	5	OIDC HTTP call timeout
JWT_SECRET	REPLACE_ME	Legacy JWT secret (replace!)
JWT_ALGORITHM	HS256	JWT algorithm

Table E.6. LLM Provider Settings

Variable	Default	Description
LLM_PROVIDER	(none)	Default provider type
LLM_MODEL	(none)	Default model name
OPENAI_API_KEY	(none)	OpenAI API key
DEMO_MODE	false	Enable demo fallback
DEFAULT_MODEL_NAME	phi3:mini	Emergency fallback model

Table E.7. Ollama Integration Settings

Variable	Default	Description
OLLAMA_BASE_URL	(auto)	Ollama API base URL
OLLAMA_TIMEOUT_SECS	180	Ollama call timeout
OLLAMA_MODEL_MAP	(none)	JSON mapping of model IDs

Table E.8. Rate Limiting Settings

Variable	Default	Description
RATE_LIMIT_ENABLED	true	Enable rate limiting
RATE_LIMIT_DEFAULT_LIMIT	60	Requests per window
RATE_LIMIT_DEFAULT_WINDOW	60	Window length (seconds)

Table E.9. Background Jobs Settings

Variable	Default	Description
JOB_STORE_BACKEND	memory	Job storage (memory/redis)
USE_POSTGRES_JOBS	false	Enable PostgreSQL jobs
JOB_TTL_DAYS	10	Job retention (days)
SSE_RING_SIZE	100	Events per job for SSE
IDEMPOTENCY_TTL_HOURS	24	Idempotency key expiry
ALLOWED_JOB_TYPES	demo	Allowed job types

Table E.10. Observability Settings

Variable	Default	Description
`PROMETHEUS_METRICS_ENABLED`	`true`	Enable Prometheus metrics
`OTEL_SERVICE_NAME`	`cineca-agentic-platform`	OpenTelemetry service name
`OTEL_EXPORTER_OTLP_ENDPOINT`	(none)	OTLP exporter endpoint
`OTEL_TRACES_SAMPLER`	`parentbased_always_on`	Trace sampler

Appendix F

Example Configurations and Environment Templates

This appendix provides example configuration files and environment templates for different deployment scenarios.

F.1 Development Environment

F.1.1 Minimal Development Setup

For local development with Docker Compose:

```
# Application
APP_ENV=dev
LOG_LEVEL=DEBUG
ENABLE_DOCS=true

# PostgreSQL
DB_HOST=postgres
DB_PORT=5432
DB_NAME=cineca_platform
DB_USER=cineca_user
DB_PASSWORD=dev_password
DB_SSLMODE=disable
DB_POOL_SIZE=10

# Memgraph
MG_HOST=memgraph
MG_PORT=7687

# Redis
REDIS_URL=redis://redis:6379/0

# LLM (Ollama)
OLLAMA_BASE_URL=http://ollama:11434/v1
OLLAMA_TIMEOUT_SECS=180
DEFAULT_MODEL_NAME=phi3:mini

# Jobs
USE_POSTGRES_JOBS=true
ALLOWED_JOB_TYPES=demo,test,long-running
```

```

# Security (development)
OIDC_ISSUER=https://dev.auth0.com/
OIDC_AUDIENCE=https://api.dev.example.com
ENABLE_ADMIN_ROUTES=1

# Internal utilities
INTERNAL_DB_UTILS_ENABLED=true
```

Listing F.1. .env.development

F.2 Testing Environment

F.2.1 CI/CD Test Configuration

For automated testing with mocked services:

```
# Application
APP_ENV=test
LOG_LEVEL=WARNING
ENABLE_DOCS=false

# PostgreSQL (test database)
DB_HOST=localhost
DB_PORT=5432
DB_NAME=cineca_test
DB_USER=test_user
DB_PASSWORD=test_password
DB_SSLMODE=disable
DB_POOL_SIZE=5

# LLM (mock mode)
DEMO_MODE=true

# Memgraph response (fast fallback for tests)
MEMGRAPH_RESPONSE_MODE=fallback-only
MEMGRAPH_BUILDER_LLM_TIMEOUT_MS=500

# Jobs (in-memory for isolation)
JOB_STORE_BACKEND=memory
USE_POSTGRES_JOBS=false

# Rate limiting (disabled for tests)
RATE_LIMIT_ENABLED=false
```

Listing F.2. .env.test

F.3 Production Environment

F.3.1 Full Production Configuration

For production deployment with enterprise security:

```
# Application
APP_ENV=prod
```

```
APP_HOST=0.0.0.0
APP_PORT=8000
LOG_LEVEL=INFO
ENABLE_DOCS=false

# PostgreSQL (managed database)
DB_HOST=postgres-primary.internal.example.com
DB_PORT=5432
DB_NAME=cineca_platform
DB_USER=cineca_prod_user
DB_PASSWORD=${SECRET_DB_PASSWORD}
DB_SSLMODE=verify-full
DB_POOL_SIZE=50

# Security (OIDC)
OIDC_ISSUER=https://auth.example.com/
OIDC_AUDIENCE=https://api.example.com
OIDC_JWKS_URL=https://auth.example.com/.well-known/jwks.json

# Security headers
ENABLE_SECURITY_HEADERS=true
ENABLE_HSTS=true
HSTS_MAX_AGE=31536000
SECURE_COOKIES=true
TRUST_PROXY=true

# Rate limiting
RATE_LIMIT_ENABLED=true
RATE_LIMIT_BACKEND=redis
RATE_LIMIT_DEFAULT_LIMIT=100

# Observability
PROMETHEUS_METRICS_ENABLED=true
OTEL_SERVICE_NAME=cineca-agentic-platform
OTEL_EXPORTER_OTLP_ENDPOINT=http://otel-collector.internal:4317

# Internal utilities (disabled in production)
INTERNAL_DB_UTILS_ENABLED=false
ENABLE_ADMIN_ROUTES=false
```

Listing F.3. .env.production

F.4 GPU-Enabled Configuration

For deployments with GPU acceleration:

```
# GPU settings
HAS_GPU=true
LLM_DEVICE=gpu

# Ollama with GPU
OLLAMA_NUM_PARALLEL=4
OLLAMA_MAX_LOADED_MODELS=2
OLLAMA_FLASH_ATTENTION=1
OLLAMA_NUM_CTX=4096

```

```
# Faster LLM timeouts with GPU
OLLAMA_TIMEOUT_SECS=60
LLM_WARMUP_TIMEOUT=120
MEMGRAPH_BUILDER_LLM_TIMEOUT_MS=10000
```

Listing F.4. .env.gpu

F.5 Security Hardening Checklist

Before deploying to production, verify:

1. `DB_PASSWORD` changed from default
2. `JWT_SECRET` replaced with secure value
3. `DB_SSLMODE=require` or higher
4. `OIDC_ISSUER` and `OIDC_AUDIENCE` configured
5. `ENABLE_DOCS=false`
6. `ENABLE_ADMIN_ROUTES=false` (or use RBAC)
7. `INTERNAL_DB_UTILS_ENABLED=false`
8. `SECURE_COOKIES=true`
9. `ENABLE_HSTS=true`
10. All secrets loaded from secure vault

Appendix G

Glossary of Terms

This appendix defines key terms, acronyms, and concepts used throughout this thesis and the Cineca Agentic Platform documentation.

G.1 General Terms

Agent An autonomous software entity that can perceive its environment, make decisions, and take actions to achieve goals. In this platform, agents are LLM-powered systems that execute multi-step workflows.

Agentic AI AI systems that exhibit autonomous, goal-directed behavior, capable of planning, reasoning, and using tools to accomplish complex tasks without explicit step-by-step instructions.

Control Plane The layer responsible for configuration, orchestration, and management of platform components. In this architecture, PostgreSQL serves as the control plane data store.

Data Plane The layer handling low-latency, high-throughput data operations. Redis serves this role for caching, queuing, and session state.

Multi-Tenancy An architecture where a single application instance serves multiple tenant organizations with logical data isolation.

Orchestration The coordination of multiple components, services, or steps to accomplish a complex task. The Agent Orchestration Engine coordinates LLM reasoning, tool invocation, and state management.

Tool A discrete, invokable capability that agents can use to interact with external systems, process data, or perform specialized tasks.

G.2 Architecture Terms

ADR Architecture Decision Record. A document capturing the context, decision, and consequences of an architectural choice.

Circuit Breaker A fault tolerance pattern that prevents cascading failures by temporarily stopping requests to a failing service after a threshold of failures.

Repository Pattern A data access pattern that abstracts database operations behind a collection-like interface, promoting testability and decoupling.

SSE Server-Sent Events. A protocol for servers to push real-time updates to clients over HTTP. Used for job progress streaming.

G.3 LLM and AI Terms

Chain-of-Thought (CoT) A prompting technique where the LLM is encouraged to articulate its reasoning step-by-step before providing a final answer.

Context Window The maximum number of tokens an LLM can process in a single request, including both input and output.

Hallucination When an LLM generates plausible-sounding but factually incorrect or unsupported information.

Intent Classification The process of determining the user's intended action from their natural language input. Used as a security filter.

LLM Large Language Model. A neural network trained on vast text corpora, capable of understanding and generating human-like text.

MCP Model Context Protocol. A standardized interface for tool discovery, invocation, and result handling in agentic systems.

Ollama An open-source platform for running LLMs locally, providing an OpenAI-compatible API.

RAG Retrieval-Augmented Generation. A technique combining information retrieval with LLM generation to provide grounded, factual responses.

Temperature A hyperparameter controlling the randomness of LLM outputs. Lower values (e.g., 0.0) produce deterministic outputs; higher values (e.g., 1.0) increase creativity.

Token The basic unit of text processed by LLMs. Roughly equivalent to 3-4 characters or 0.75 words in English.

G.4 Database Terms

Alembic A database migration framework for SQLAlchemy, enabling version-controlled schema changes.

Cypher A declarative graph query language used by Memgraph and Neo4j, similar to SQL for graph data.

Memgraph An in-memory graph database using the Cypher query language, optimized for real-time analytics.

PostgreSQL An advanced open-source relational database known for robustness, extensibility, and standards compliance.

Redis An in-memory data store supporting strings, lists, sets, sorted sets, and hashes. Used for caching, queues, and pub/sub.

SQLAlchemy A Python SQL toolkit and ORM providing database abstraction and query generation.

G.5 Security Terms

JWT JSON Web Token. A compact, URL-safe token format for representing claims between parties, used for authentication.

JWKS JSON Web Key Set. A set of keys containing public keys used to verify JWT signatures.

OIDC OpenID Connect. An identity layer on top of OAuth 2.0, providing authentication and user identity information.

Output Guard A security mechanism that validates LLM outputs for harmful content, policy violations, or sensitive data.

PII Personally Identifiable Information. Data that can identify an individual, such as names, emails, or phone numbers. The platform scrubs PII from logs.

RBAC Role-Based Access Control. An authorization model where permissions are assigned to roles rather than individual users.

Scope A permission unit in OAuth/OIDC defining what resources a token can access (e.g., `graph:read`, `admin:all`).

Tenant An organizational unit in a multi-tenant system, with isolated data and potentially separate configuration.

G.6 Platform-Specific Terms

Agent Run A single execution of an agent, from receiving a user request to producing a final response.

Agent Session A stateful conversation context containing multiple agent runs (steps) with shared history.

Agent Step An individual action within an agent session: user message, assistant response, tool invocation, or system event.

Job A unit of background work with a lifecycle (queued → running → finished/failed) and associated payload and result.

Model Instance A loaded, configured LLM accessible through a provider, with associated capabilities and settings.

NL→Cypher The pipeline translating Natural Language questions into Cypher graph queries for Memgraph.

Provider An LLM service configuration (OpenAI, Ollama, etc.) with connection details and credentials.

Tool Invocation A recorded execution of a tool, including input parameters, output, latency, and any errors.

G.7 Acronyms

Table G.1. Common Acronyms

Acronym	Meaning
ADR	Architecture Decision Record
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
HSTS	HTTP Strict Transport Security
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
MCP	Model Context Protocol
NL	Natural Language
OIDC	OpenID Connect
ORM	Object-Relational Mapping
OTel	OpenTelemetry
PII	Personally Identifiable Information
RBAC	Role-Based Access Control
REST	Representational State Transfer
SSE	Server-Sent Events
TLS	Transport Layer Security
TTL	Time To Live
UUID	Universally Unique Identifier

Bibliography

[1] OpenAI. (2023). GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*.

[2] Touvron, H., et al. (2023). LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*.

[3] Brown, T., et al. (2020). Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.

[4] LangChain. (2023). LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain (Accessed: 2024-12-18).

[5] Yao, S., et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*.

[6] Significant Gravitas. (2023). AutoGPT: An Autonomous GPT-4 Experiment. https://github.com/Significant-Gravitas/AutoGPT (Accessed: 2024-12-18).

[7] Anthropic. (2024). Model Context Protocol Specification. https://modelcontextprotocol.io/ (Accessed: 2024-12-18).

[8] Memgraph. (2023). Memgraph: Real-time Graph Database. https://memgraph.com/ (Accessed: 2024-12-18).

[9] Francis, N., et al. (2018). Cypher: An Evolving Query Language for Property Graphs. *Proceedings of the 2018 International Conference on Management of Data*, 1433–1445.

[10] Sakimura, N., et al. (2014). OpenID Connect Core 1.0. OpenID Foundation.

[11] European Union. (2016). Regulation (EU) 2016/679 (General Data Protection Regulation). *Official Journal of the European Union*, L 119/1. https://eur-lex.europa.eu/eli/reg/2016/679/oj (Accessed: 2024-12-18).

[12] American Institute of Certified Public Accountants (AICPA). (2017). SOC 2 Type II: Trust Services Criteria for Security, Availability, Processing Integrity, Confidentiality, and Privacy. AICPA.

[13] U.S. Department of Health and Human Services (HHS). (2013). Health Insurance Portability and Accountability Act (HIPAA) Privacy Rule. 45 CFR Parts 160 and 164. https://www.hhs.gov/hipaa/index.html (Accessed: 2024-12-18).

[14] PCI Security Standards Council. (2018). PCI DSS v3.2.1: Payment Card Industry Data Security Standard. https://www.pcisecuritystandards.org/ (Accessed: 2024-12-18).

[15] Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519.

[16] Prometheus Authors. (2023). Prometheus: Monitoring system and time series database. https://prometheus.io/ (Accessed: 2024-12-18).

[17] OpenTelemetry Authors. (2023). OpenTelemetry: High-quality, ubiquitous, and portable telemetry. https://opentelemetry.io/ (Accessed: 2024-12-18).

[18] Ramírez, S. (2023). FastAPI: Modern, Fast Web Framework for Building APIs with Python. https://fastapi.tiangolo.com/ (Accessed: 2024-12-18).

[19] CINECA. (2023). CINECA: Italian Interuniversity Consortium for Supercomputing. https://www.cineca.it/ (Accessed: 2024-12-18).

[20] Vaswani, A., et al. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems*, 30, 5998–6008.

[21] Kaplan, J., et al. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

[22] Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems*, 35, 24824–24837.

[23] Yao, S., et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *arXiv preprint arXiv:2305.10601*.

[24] OpenAI. (2023). Assistants API Documentation. https://platform.openai.com/docs/assistants (Accessed: 2024-12-18).

[25] LangChain. (2024). LangGraph: Build Stateful, Multi-Actor Applications with LLMs. https://github.com/langchain-ai/langgraph (Accessed: 2024-12-18).

[26] LlamaIndex. (2023). LlamaIndex: A Data Framework for LLM Applications. https://github.com/run-llama/llama_index (Accessed: 2024-12-18).

[27] Microsoft. (2023). Semantic Kernel: An AI orchestration framework. https://github.com/microsoft/semantic-kernel (Accessed: 2024-12-18).

[28] Microsoft. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. https://github.com/microsoft/autogen (Accessed: 2024-12-18).

[29] CrewAI. (2024). CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/joaomdmoura/crewAI (Accessed: 2024-12-18).

[30] Neo4j, Inc. (2023). Neo4j Graph Database Platform. https://neo4j.com/ (Accessed: 2024-12-18).

[31] Hardt, D. (Ed.). (2012). The OAuth 2.0 Authorization Framework. RFC 6749.

[32] Ferraiolo, D., Kuhn, R., & Chandramouli, R. (2004). *Role-Based Access Control*. Artech House.

[33] Grafana Labs. (2023). Grafana: The open observability platform. https://grafana.com/ (Accessed: 2024-12-18).

[34] Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems*, 33, 9459–9474.

About the Author

Arman Feili is a Master's student in Data Science at Sapienza Università di Roma. This thesis was developed in collaboration with CINECA, Italy's national supercomputing center, as part of research into enterprise-grade AI agent orchestration systems.

Contact and Links:

- **Email:** feili.2101835@studenti.uniroma1.it
- **GitHub:** `https://github.com/armanfeili`
- **Google Scholar:** `https://scholar.google.com/citations?user=qfG60rMAAAAJ&hl=en`
- **LinkedIn:** `https://www.linkedin.com/in/arman-feili/`

Supervisors:

- **Prof. Marco Raoul Marini** (Sapienza) – marini@di.uniroma1.it
- **Dr. Valerio Venanzi** (Sapienza) – venanzi@di.uniroma1.it
- **Dr. Giuseppe Melfi** (CINECA) – G.MELFI@cineca.it
- **Dr. Marco Puccini** (CINECA) – m.puccini@cineca.it