Federated Gateway Layer for Enterprise Conversational AI Agents

Supporting Agentic Enterprise AI Through a Unified Discovery and Messaging Infrastructure

Bachelor's thesis in Computer Engineering

Hussein Hafid & Yunhee Jung Larsson

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Thesis work 2026

Federated Gateway Layer for Enterprise Conversational AI Agents

Supporting Agentic Enterprise AI Through a Unified Discovery and Messaging Infrastructure

Hussein Hafid & Yunhee Jung Larsson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg 2026

Federated Gateway Layer for Enterprise Conversational AI Agents
Supporting Agentic Enterprise AI Through a Unified Discovery and Messaging Infrastructure
Hussein Hafid & Yunhee Jung Larsson

Supervisors:	Stas Invinski, Volvo Cars Mohammad Mustafa, Chalmers University of Technology
Examiner:	John Camilleri, Chalmers University of Technology

Thesis work 2026
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telefon +46 31 772 1000

Skriven i LaTeX
Gothenburg 2026

Federated Gateway Layer for Enterprise Conversational AI Agents
Supporting Agentic Enterprise AI Through a Unified Discovery and Messaging Infrastructure
Hussein Hafid & Yunhee Jung Larsson
Department of Computer Science and Engineering
Chalmers Tekniska Högskola

Abstract

This thesis presents a federated gateway layer designed to improve how users interact with distributed enterprise conversational AI agents. In large organizations, services are often fragmented, making it difficult to discover and access relevant information. The proposed system introduces a lightweight API gateway that enables unified service discovery and message routing without centralizing domain systems.

The gateway uses metadata-driven filtering and deterministic routing to identify suitable agents, ensuring predictable and explainable behavior. It is integrated with conversational interfaces through the Model Context Protocol (MCP), allowing AI assistants to discover and communicate with services via structured tool calls. A registry-based design supports flexible search, including fuzzy matching, while sensitive data is securely managed through encryption.

The implemented system demonstrates that a gateway-based approach can reduce complexity, improve discoverability, and provide a consistent interaction model in enterprise environments

Sammanfattning

Denna uppsats presenterar ett federerat gateway-lager som är utformat för att förbättra hur användare interagerar med distribuerade konversationsbaserade AI-agenter i företagsmiljöer. I stora organisationer är tjänster ofta fragmenterade, vilket gör det svårt att upptäcka och få tillgång till relevant information. Det föreslagna systemet introducerar en lättviktig API-gateway som möjliggör enhetlig tjänsteupptäckt och meddelanderouting utan att centralisera domänsystemen.

Gatewayen använder metadata-driven filtrering och deterministisk routing för att identifiera lämpliga agenter, vilket säkerställer förutsägbart och förklarbart beteende. Den är integrerad med konversationsgränssnitt genom Model Context Protocol (MCP), vilket gör det möjligt för AI-assistenter att upptäcka och kommunicera med tjänster via strukturerade verktygsanrop. En registerbaserad design möjliggör flexibel sökning, inklusive fuzzy matching, samtidigt som känslig data hanteras säkert genom kryptering.

Det implementerade systemet visar att en gateway-baserad metod kan minska komplexitet, förbättra upptäckbarhet och tillhandahålla en konsekvent interaktionsmodell i företagsmiljöer.

Acknowledgement

This project would not have been possible without the support and guidance we received throughout the process.

We would like to express our sincere gratitude to our examiner, John J. Camilleri, and our Chalmers supervisor, Muhammad Mustafa Hassan, for their valuable feedback and academic guidance throughout this project.

We also extend our sincere thanks to our supervisors at Volvo Cars, Erfan Makhsos and Stas Ivinski, for their continuous support, practical insights, and for providing us with the opportunity to carry out this project in an enterprise environment.

Finally, we would like to thank our family and friends for their encouragement and support throughout the project.

Hussein Hafid & Yunhee Jung Larsson, Gothenburg, June 2026

Contents

Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

Abbreviation	Meaning	Page
AI	Artificial intelligence	3, 8, 9, 15, 25, 27, 28, 35, 37, 39
API	Application Programming Interface	3, 5–7, 9, 15–19, 26–29, 35–37
CI/CD	Continuous Integration and Continuous Delivery/Deployment	38
CRUD	Create Read Update Delete	6, 25
HR	Human Resources	1
HTTP	Hypertext Transfer Protocol	5, 6, 9, 15, 16, 22–26, 36
HTTPS	Hypertext Transfer Protocol Secure	38
IT	Information Technology	1
JSON	JavaScript Object Notation	4, 22, 25
LLM	Large Language Model	4, 35
MCP	Model Context Protocol	5, 9, 15, 16, 22, 25, 26, 35–38
MVP	Minimum Viable Product	3, 27, 35, 36, 39
ORM	Object-Relational Mapping	5, 7
RBAC	Role-Based Access Control	36
SQL	Structured Query Language	7
STDIO	Standard Input and Output	16, 26
UI	User Interface	16, 22, 25
URL	Uniform Resource Locator	6

List of Figures

List of Tables

1

Introduction

This chapter introduces the context, motivation, and objectives of the project. It presents the challenges associated with accessing distributed enterprise systems and outlines the proposed approach to address them. The purpose, goals, and scope of the project are defined, followed by a description of the key problems that motivate the work.

1.1 Background

In large enterprises such as Volvo Cars , functions such as Information Technology (IT) support, Human Resources (HR) policy guidance, and data insights are often provided through separate domain agents or data services. Since these systems are typically developed and operated independently, users must manually navigate multiple platforms or repeat the same query across different systems to obtain the information they need.

Instead of building a centralized "all-knowing" system, Volvo Cars is exploring a federated architecture. In this model, each domain retains ownership of its own services and data. The goal is not to merge systems, but to enable coordinated access across them.

To support this, a gateway layer is introduced as an abstraction between users and domain agents. The gateway does not contain domain-specific logic or data. Instead, it acts as an orchestrator and policy enforcement point that routes requests based on metadata, applies access control, and standardizes interaction patterns. This allows the gateway to hide the internal complexity of domain agents while ensuring consistent and controlled access through a unified interface.

1.2 Purpose

The purpose of this project is to optimize information - intensive workflows in Volvo Cars by reducing the time, effort, and risk involved in collecting data from multiple dashboards, databases, and teams. Today, analysts and engineers must know where to find information, how to interpret differing semantics and aliases across sources, and how to avoid missing critical data points that could skew decisions.

The project aims to standardize request handling and information retrieval. This ensures that user requests are forwarded to the correct agent or data service quickly and reliably. This reduces decision risk from incomplete context and improves cross-domain discoverability, leading to more consistent and efficient access to relevant information.

1.3 Goal

The goal of this project is to develop a lightweight gateway designed to streamline the handling of user requests. The system will analyze each incoming request, direct it to the appropriate domain service, and ensure that the final response is consistent and clear, regardless of the data source. The project will include the following concrete tasks:

1. Design the interaction protocol or layer between the entry-point agent and the gateway.
2. Implement the gateway layer, responsible for:
    - Implementing a request-correlation mechanism to maintain context within a stateless architecture
    - Routing requests based on metadata **or** heuristic-based intent mapping
3. Build a management interface for registering chatbots and data services.
4. Implement a lightweight logging system to track activity and support efficient debugging.
5. Design the communication protocol or layer between the gateway and downstream domain chatbots and services.
6. Integrate both:
    - Conversational AI backends
    - Analytic service providers
7. Validate the system through automated testing.
8. Architect the system with built-in fault tolerance to ensure resilience and reliability.

1.4 Problem Definition

As the number of enterprise data services grow, the overall system architecture becomes increasingly complex and difficult to maintain. This results in several key problems:

- Users often do not know which agent, bot, or data service can answer their question or provide the information they need.
- Different agents have varying levels of functionality, data access, and interaction patterns, leading to an uneven and unpredictable user experience.

- Users must understand how to interact with multiple agents and services, each with its own commands, semantics, or interface.
- Information from different Artificial intelligence (AI) models or data sources is returned separately, requiring users to manually interpret and consolidate results.
- As more agents and services are added, the overall architecture becomes harder to scale, test, and extend without increasing system complexity.

The focus of this project is on improving the discovery and routing of requests. By identifying the appropriate service and forwarding requests correctly, the system reduces the manual effort required to locate relevant information sources. The consolidation of responses from multiple services is recognized as a related challenge, and is treated as an extension beyond the core scope.

1.5 Scope

To ensure timely delivery, functional correctness, and sufficient testing, the project scope is structured around two categories of functionality: the Minimum Viable Product (MVP) capabilities and a set of additional enhancements that can improve usability and overall user experience. The MVP defines the essential features required for a fully functioning system, while the additional enhancements represent optional improvements that may be included depending on available time and project priorities.

1.5.1 MVP Scope

The MVP encompasses the essential features necessary to deliver a functional, end-to-end system. Delivery of the MVP constitutes successful project completion. The scope includes:

1. Conversational agentic interface
    - Chat interaction
    - Submits user queries to the gateway
    - Presents responses to the user
2. Gateway Application Programming Interface (API)
    - Receives and forwards requests
    - Handles stateless request tracking with optional identifiers
    - Applies basic routing behaviours:
        - No matches: signal absence of suitable services
        - One or more matches: return a set of candidate services, deferring explicit confirmation request to the user
3. Routing Engine
    - Keyword-based intent classification
4. Service Registry
    - Lightweight service entries representing external domain agents or services

    - Administrative interface for registering and managing services
5. Turnaround-Time Logging
    - Measures request latency during development-time execution for architectural evaluation

1.5.2 Additional Enhancements

The features below represent improvement opportunities that extend the system's capabilities. These are not required for project success and may be implemented partially or fully depending on available time.

1. User Interaction Improvements
    - Human-in-the-loop decision-making when multiple matches occur
    - Large Language Model (LLM)-guided follow-up questions to narrow search results
2. Smarter Search Logic
    - Synonym expansion for more flexible keyword matching
    - Support for negative keywords (e.g., excluding specific terms)
3. Observability and Quality Improvements
    - Advanced logging, including error categories, performance metrics, and deeper traceability
    - User feedback mechanisms for evaluating response quality
4. Support for Heterogeneous Systems
    - Integration with various analytic services or databases
    - Unified routing across mixed data types
    - Schema validation for structured outputs such as JavaScript Object Notation (JSON) objects or tabular data
5. Multi-Response Handling
    - Structured data aggregation
        - Combine responses from multiple services (e.g., merging JSON objects or tables)
        - Apply validation rules to ensure consistency
    - Chatbot response summarization
        - Generate a unified summary when multiple chatbots respond
        - Highlight common points, contradictions, or recommended actions
        - Present a single, coherent response to the user

Together, these features define a system that improves service discovery and access while preserving the independence of domain agents in a federated environment.

2

Technical Background

This chapter introduces the technical concepts required to understand the design choices and implementation details of the system. It covers the fundamentals of web frameworks, web APIs, the API gateway pattern, cryptographic protection of stored credentials, database abstraction through an Object-Relational Mapping (ORM), heuristic-based intent mapping, and the role of agent tooling through Model Context Protocol (MCP).

2.1 Web Frameworks

Web development frameworks are software tools that provide a structured foundation for building web applications. They offer reusable components and abstractions that simplify common development tasks, such as handling Hypertext Transfer Protocol (HTTP) requests, routing, and managing application logic [1]. By providing these capabilities, frameworks enable developers to focus on application-specific functionality rather than low-level implementation details.

Web frameworks are typically categorized into two main types: front-end frameworks and back-end frameworks. Back-end frameworks are used to develop server-side applications that run on remote machines, commonly referred to as servers. These applications are responsible for providing services such as data processing, authentication, and communication with databases. In contrast, front-end frameworks are used to develop client-side applications, which run on the user's device and provide graphical user interfaces that interact with back-end systems.

This project focuses on the development of a back-end application. The system is designed as a server-side component that exposes functionality through a set of defined interfaces, enabling interaction with external clients and services.

Among the available back-end frameworks, FastAPI [2] was selected for this project. It provides built-in support for handling HTTP requests, asynchronous processing, and automatic generation of API documentation. The selection of FastAPI was influenced by its compatibility with Python as well as its conceptual similarities to other established frameworks, such as Django [3], which reduced implementation risk during early development.

2.2 Application Programming Interfaces

Most web APIs follow a resource-based model. Resources (e.g., users, messages, or chatbots) are identified by Uniform Resource Locators (URLs), and clients interact with them using standard HTTP methods [4]:

- GET: Retrieve data
- POST: Create data or trigger an action
- PUT: Update existing data
- DELETE: Remove data

In this project, the API follows a mixed design. Standard resource-oriented endpoints are used for data retrieval, while certain operations are expressed as actions. These action endpoints are prefixed with an underscore (e.g., `_search`, `_message`) to distinguish them from resource-based endpoints. This approach allows the API to support operations that do not map cleanly to standard Create Read Update Delete (CRUD) interactions.

2.3 API Gateway

An API gateway is a server that acts as an entry point for client requests. Unlike a simple reverse proxy that mainly forwards traffic, a gateway can apply active logic such as authentication, request validation, and routing decisions before requests reach downstream services. This makes the gateway a policy enforcement point rather than a passive forwarder [5].

In this project, the gateway exposes a single HTTP API that provides access to discovery and interaction. Clients send requests to the gateway, which then processes them and returns a response. This design centralizes shared logic and ensures that all clients interact with the system in a consistent way.

The gateway manages two separate APIs with varying access requirements (see section 4.3). For example, the public API is accessible without authentication, allowing clients to perform discovery and interaction tasks. In contrast, the administrative API is restricted and uses session-based authentication to control access. This separation ensures that sensitive operations are protected while keeping public functionality simple to use.

In addition to routing, the gateway enforces system policies. In this project, enforced policies include protecting stored credentials through encryption and applying deterministic discovery rules when mapping a request to candidate domain agents. These policies ensure that security and routing behavior are applied consistently across all clients and interfaces.

2.4 Cryptographic Foundations

In an enterprise setting, a gateway often needs to store sensitive credentials such as API keys for downstream services. These credentials enable the gateway to act as a trusted intermediary when forwarding requests to domain agents. If such secrets are stored in plain text, an attacker who gains read access to the registry can misuse them to access downstream systems.

To reduce this risk, secrets should be encrypted at rest [6]. This project uses symmetric encryption to protect stored credentials. Symmetric encryption uses the same key for encryption and decryption [7], which fits the gateway setting where only the gateway needs to recover the plaintext secret.

A practical pattern for this is Fernet encryption in Python. Fernet provides confidentiality and tamper detection for encrypted values, meaning the stored secret cannot be read without the key and any modification of the ciphertext is detected during decryption [8]. Internally, Fernet uses AES-128 in CBC mode for encryption and an HMAC-SHA256 tag for integrity verification [8].

Key management is critical. The encryption key must not be stored next to the encrypted data. In practice, it should be supplied through environment variables or a dedicated secret manager [6]. If the key is lost, stored credentials cannot be recovered. If the key is leaked, encrypted credentials can be decrypted.

This security mechanism is treated as an enforced policy of the gateway. The gateway is not only a routing component, but also a controlled access layer that must handle credentials and enforce security decisions on behalf of clients.

2.5 Object-Relational Mapping

ORM is a technique that allows developers to interact with a database using programming language objects instead of writing raw Structured Query Language (SQL) queries [9]. An ORM maps database tables to classes and rows to objects, and translates application operations into database operations behind the scenes.

ORMs are also commonly used to define the database schema through code [9]. By defining models in the application, the database structure can be created and maintained from the same codebase, reducing the need for manual schema scripts.

In this project, all data access is handled through Python code using an ORM. No database-specific queries or SQL statements are written directly. This simplifies development and improves readability, as the data layer is expressed entirely in Python.

This approach also provides flexibility. Since the application logic does not depend on a specific database language, the underlying database system can be changed with minimal impact on the rest of the code.

2.6 Intent Mapping

Service discovery requires mapping user input to one or more domain agents. In practice, user input can be incomplete, inconsistent, or contain minor spelling errors. A strict exact-match approach can therefore lead to unnecessary failures.

Several matching techniques are commonly used:

- Exact matching: the input must match the stored value exactly (e.g., “Sweden” → “Sweden”).
- Partial matching: the input may match a substring of the stored value (e.g., “Swed” → “Sweden”).
- Fuzzy matching: the input may differ slightly from the stored value (e.g., “Swdeen” → “Sweden”).

In this project, intent mapping is implemented using metadata-driven discovery combined with tolerant string matching. The gateway compares user-provided filters against stored service metadata such as country, department, and capabilities. To make discovery tolerant to minor variations, the project uses string matching through the RapidFuzz library [10], which supports fault-tolerant matching during search.

This approach is deterministic. It does not rely on a language model to decide which service matches the input. Instead, it applies defined matching rules to map user input to candidate domain agents. In an enterprise setting, this improves reliability and auditability because the routing decision can be explained and reproduced using the same stored metadata and matching rules.

2.7 Agentic Artificial Intelligence

Agentic AI refers to systems that can select actions and use external tools to achieve a goal. In addition to generating responses, such systems can reason about tasks, plan a sequence of steps, and use information from previous interactions [11].

In this project, an AI agent is used as the interface to the system. The agent receives user input, interprets the intent, and decides which actions to perform. These actions include discovering available domain agents and sending messages to them.

The agent also maintains context across interactions, allowing it to use information from earlier exchanges when generating responses [11]. This enables consistent behavior across multiple turns.

However, the agent cannot directly communicate over HTTP. It does not call API endpoints by issuing HTTP requests. Instead, it requires a tool interface that can execute operations on its behalf. This motivates the use of a standardized tool-connection protocol.

2.8 Model Context Protocol

MCP is an open standard for connecting AI applications to external systems [12]. It allows an AI application to access data sources and invoke tools through a consistent interface, without requiring a custom integration for each external system. More formally, MCP is “an emerging open standard that defines a unified, bi-directional communication and dynamic discovery protocol between AI models and external tools or resources, aiming to enhance interoperability and reduce fragmentation across diverse systems” [13, p. 1].

In MCP, the server exposes a structured catalogue of available operations. Tools are defined by a name and a machine-readable schema that states what parameters are required and what results are returned [14]. This schema functions as a contract between the AI application and the external system, ensuring that tool calls are well-defined and unambiguous.

In this project, MCP is used as the tool interface for the agent. The MCP server exposes gateway functionality as tools, such as listing attributes, searching for domain agents, and sending messages. When the agent selects a tool, the MCP server validates the input against the tool schema and executes the operation on behalf of the agent.

The gateway itself remains an HTTP-based service. The MCP server therefore also acts as a translation layer: it converts tool invocations into HTTP requests to the Gateway API and returns the HTTP response back to the agent in a structured form. This design allows the agent to use the gateway without implementing HTTP communication directly, while keeping the tool contract separate from the API implementation.

3

Methodology

This chapter describes the planned approach for the project from a “before” perspective. It outlines how the work was intended to be organized, which tools were planned to be used, the planned development increments, and how results were planned to be validated. Any deviations from this plan and the reasons behind them are discussed later in the report.

3.1 Planned Development Approach

The project was planned as an iterative development effort with two-week development cycles. The intention was to complete one planned increment per cycle. Each cycle was planned to end with a functional and well-documented system state, supported by testing and basic verification, before moving to the next increment.

The project was planned to be carried out jointly by the two authors. The intended strategy was to develop the system architecture and the technical report in parallel. This was planned to ensure that implementation details and design decisions were documented while they were still fresh, and that the report remained consistent with the evolving codebase.

Each two-week cycle was planned to follow the same high-level structure:

1. Planning: select the next increment, define its scope, and clarify expected inputs and outputs.
2. Implementation: implement the increment and incorporate it into the codebase.
3. Testing & verification: update the automated test suite and verify that the system remains functional and stable, while updating the report to reflect the implemented behavior.

3.1.1 Supervision and External Review

Supervision was planned as part of the quality assurance process. Regular meetings with the university supervisor were planned to support academic writing quality and research framing. These meetings were intended to review the report structure, clarity of argumentation, use of references, and alignment with thesis requirements. Project status updates were also included to ensure that the written work accurately reflected ongoing progress.

Technical questions and implementation-related decisions were planned to be handled through the company supervisor. This included guidance on practical constraints and company-specific context required to make correct technical choices. Consultations with the company supervisor were planned when technical uncertainty arose or when design decisions required confirmation against enterprise requirements.

3.2 Work Planning and Tracking

Work was planned and tracked using GitHub Projects [15] with a Kanban board. The board was intended to support prioritization and provide a shared overview of the current state of the work. The Kanban board columns were:

- Todo
- In progress
- Done

Each board item was intended to represent a development increment rather than a single feature or change. A single increment could therefore include multiple subtasks such as implementation, configuration, integration, and documentation updates.

3.3 Planned Development Increments

The planned work was structured as a sequence of development increments. The intent was to complete one increment every two weeks and keep the system functional, stable, and well documented at the end of each cycle. Integration details were expected to be clarified progressively as development advanced. Table 3.1 summarizes the planned increments.

Table 3.1: Planned development increments (two-week cycles).

Increment	**Planned outcome**
System Inception & Scaffolding	Repository setup, environment management via `uv`, interface-first design using stubbed modules and endpoints, and setup of the LaTeX report environment to document decisions and progress throughout development.
Service Discovery Framework	Implementation of the core logic required to identify relevant domain agents based on registry metadata.
Storage & Registry Management	Evaluation of persistence requirements and implementation of a registry layer for service records if required.

Increment	**Planned outcome** (continued)
Registry Management Interface	Development of a restricted interface for managing registry records with access limited to authorized users.
Forwarding & Orchestration	Implementation of the request/response loop between the gateway and downstream domain agents.
MCP Translation Layer	Development of a tool-calling interface that exposes gateway capabilities to AI agents via the Model Context Protocol.

The increments were planned to be adjusted if early integration revealed missing prerequisites or unnecessary components.

3.4 Planned Tools and Environment

The system was planned to be implemented in Python, using a lightweight development setup to support rapid iteration. Dependency and environment management was planned through `uv` [16]. Version control and task tracking were planned through GitHub [15], using issues and the Kanban board to track progress. Developing the software was through VS Code [17].

No machine learning technologies were planned for routing or discovery. The discovery logic was planned to be deterministic and based on stored metadata and matching rules, so that routing decisions could be explained and reproduced.

3.5 Planned Validation

Validation was planned as a combination of manual testing and automated tests. Manual testing was intended as an initial verification activity during development, particularly when introducing new increments or integrating external dependencies.

Automated testing was planned as a cross-cutting activity rather than a final task. The intention was to extend the test suite during each two-week cycle so that new functionality was verified and previously implemented behavior remained stable.

Since the gateway was planned to behave deterministically, validation was also framed as deterministic output verification. The intention was to verify that the gateway produces consistent and predictable discovery and routing results for the same inputs, based on the defined matching rules and stored metadata.

In addition, each cycle was planned to include a documentation update performed by the authors. The purpose was to ensure that the report reflected the implemented system behavior, interfaces, and limitations as the project evolved.

4

Implementation

This chapter describes how the gateway system was implemented and how its main components interact. The gateway is implemented as an HTTP service using FastAPI [2], where the API layer defines endpoints and request/response contracts. Data access is implemented using SQLModel [18], and administrative database access is provided through SQLAdmin [19]. Agent integration is implemented through an MCP server, which exposes gateway functionality as tools and bridges between the Gateway API and an AI assistant that supports MCP.

The chapter focuses on the purpose of each component, the interfaces between them, and the resulting system behavior. The implementation is described in terms of responsibilities and data flow, covering the Gateway API, the service registry, the administrative interface, and the MCP server used for agent integration. Figure 4.1 provides an overview of the implemented components and their interactions.

4.1 System Architecture

This section presents the macro view of the implemented system and how its main components interact. The system is divided into local components within the project scope and external cloud components. The local components are designed and implemented as part of this project, while the cloud components represent where downstream domain agents are hosted and executed.

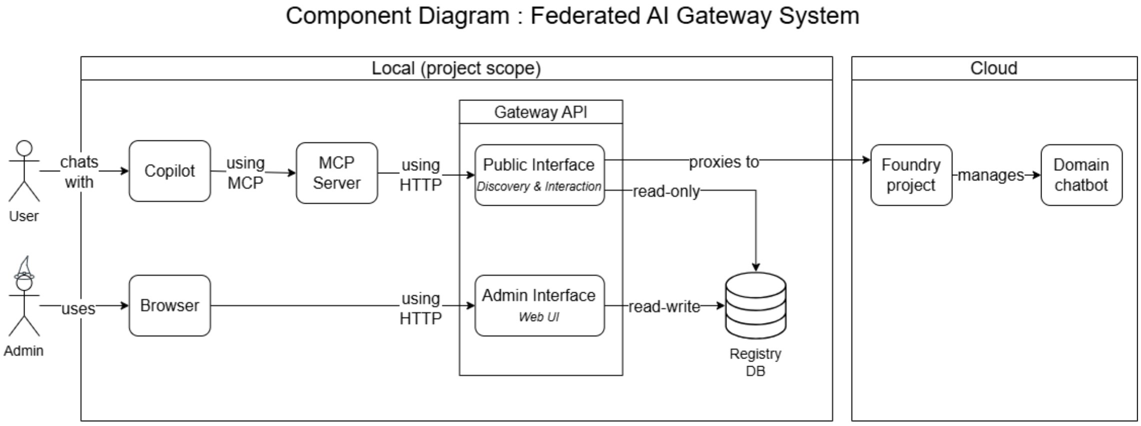


Figure 4.1: System component overview.

Figure 4.1 shows the separation between the MCP tool layer, the HTTP gateway interfaces, the registry database, and the external domain agents. In this report, the term *domain agent* refers to downstream chatbots or agents that receive forwarded requests from the gateway.

Copilot is used as the entry-point conversational interface. In this context, Copilot acts as an MCP-capable client and is distinct from the downstream domain agents that the gateway routes to. The user interacts with Copilot, and Copilot invokes gateway functionality through MCP tool calls.

The MCP server and the API gateway run as separate local processes. The MCP server communicates with Copilot over Standard Input and Output (STDIO) and translates tool invocations into HTTP requests to the gateway over localhost. This separation keeps the agent-facing tool interface independent from the HTTP API implementation.

The API gateway exposes two interfaces. The public interface is used for discovery and messaging, and it has read-only access to the registry data. The administrative interface provides restricted read-write access to manage registry records through a web User Interface (UI).

Domain agents are hosted externally. In the current implementation, domain agents are managed within a Microsoft Foundry project [20]. For the remainder of this report, the term *domain agent* is used independent of hosting provider, since the same interaction pattern can be applied to other hosting environments.

4.1.1 Layered Architecture

The Gateway API is implemented using a layered design to separate responsibilities and reduce coupling between concerns. At a high level, the API layer handles HTTP routing and request/response validation, the logic layer implements discovery and messaging behavior, and the data access layer encapsulates interaction with the registry database.

Figure 4.2 illustrates the layering and the two entry paths exposed by the gateway. The discovery and messaging path is implemented in the project codebase and accesses registry data in a read-only manner. The administrative interface is provided through SQLAdmin [19] as a restricted web UI for managing registry records, and it is the only path that performs read-write operations on registry data. Both paths share the same data access layer implemented through SQLModel [18] and the underlying database.

The separation between read-only and read-write access is intentional. Public requests can use registry metadata to perform discovery and route messages, but cannot change registry state. Registry modifications are performed only through the

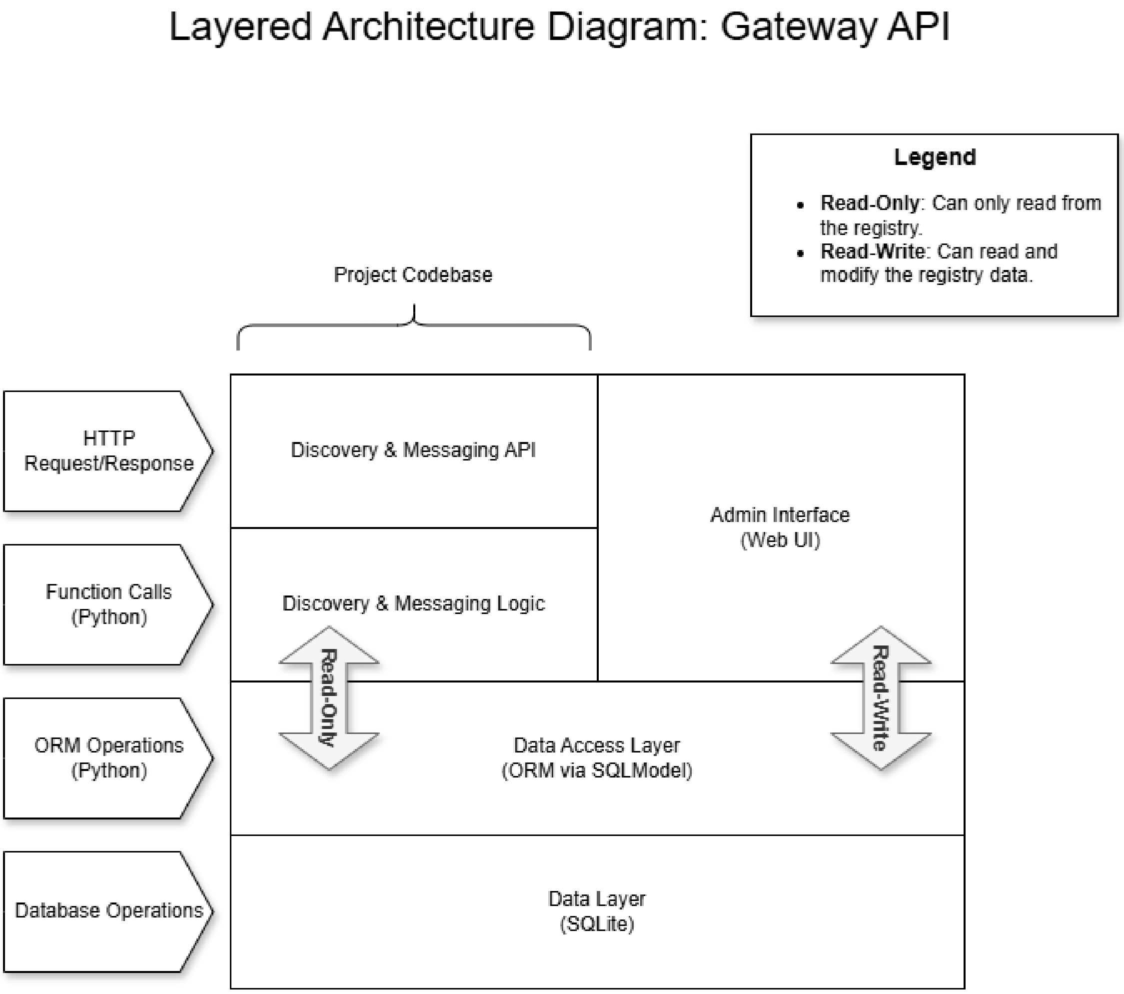


Figure 4.2: Layered architecture of the Gateway API, highlighting read-only registry access for discovery and messaging and read-write registry access through the admin interface.

restricted administrative interface. This keeps the public API simple and reduces the risk of accidental or unauthorized registry changes.

4.1.2 Configuration Fail-safe

Configuration is centralized in a `Settings` class that inherits from Pydantic `BaseSettings` [21]. This class loads configuration fields from the environment and applies validation rules to ensure that required values are present and meet defined constraints. It also defines derived values that are computed from these configuration fields during initialization.

A simplified example is shown below:

```
class Settings(BaseSettings):
    ADMIN_USERNAME: str
    ADMIN_PASSWORD: SecretStr
```

```

    DB_ENCRYPTION_KEY: str
    DB_ENCRYPTION_SALT: str
    DB_ENCRYPTION_ITER: int

    @cached_property
    def SYMMETRIC_ENCRYPT_UTIL(self) -> SymmetricEncryptUtil:
        return SymmetricEncryptUtil(
            encrypt_key=self.DB_ENCRYPTION_KEY,
            encrypt_salt=self.DB_ENCRYPTION_SALT,
            encrypt_iter=int(self.DB_ENCRYPTION_ITER)
        )

    class Config:
        env_file = ENV_FILE_PATH
        env_file_encoding = "utf-8"
```

In this example, configuration values are loaded from a dedicated `.env` file located in the project codebase and are used both directly and to construct derived runtime components. For instance, the symmetric encryption utility is initialized from multiple configuration fields and reused throughout the application lifecycle.

This approach also ensures that configuration values are consistently loaded from the intended environment file, independent of the directory from which the application is executed.

If any required configuration field is missing, invalid, or cannot be used to construct derived components, the settings initialization fails and the program terminates before the API server is started.

4.2 Metadata Driven Discovery

The service registry is the central data component of the gateway and acts as the source of truth for all domain agents and their associated metadata. It stores the minimum set of information required to support discovery and interaction. This includes domain agent records as well as supporting metadata such as countries, departments, capabilities, and connection details for downstream systems.

In this project, the terms *chatbot* and *agent* are used interchangeably. The downstream agents referenced in the registry are externally hosted text-based conversational systems. The registry stores both metadata and connection details required for interaction, but only a subset of this information is exposed through the Gateway API. The external interface is therefore limited to metadata required for discovery and routing, while implementation and connection details remain internal to the system.

The registry is not connected to any external data source and is instead populated manually. This design keeps the system lightweight and ensures that only relevant and curated data is included. In addition to descriptive metadata, the registry stores connection information such as API endpoints and keys required for forwarding requests to downstream domain agents.

Discovery is implemented as a metadata-driven process on top of the registry. The gateway supports listing available metadata entries and searching for relevant domain agents using optional filters. These filters are used to narrow down candidates based on attributes such as country, department, or capability.

Pagination is applied uniformly to all discovery results. When no filters are provided, the gateway returns a paginated subset of all registered domain agents. When filters are applied, pagination is performed on the filtered result set.

To improve usability, discovery applies tolerant matching within each filter condition. Instead of requiring exact matches, filter values are compared using fuzzy matching, allowing minor differences in user input to produce relevant results. For example, a filter such as `capabilities=["tech"]` may match agents with capabilities such as "technical advice".

This approach keeps the discovery process lightweight while providing flexible and user-friendly search behavior.

The discovery logic is intentionally deterministic. For a given set of inputs and registry state, the same results will always be returned. This improves predictability, makes the system easier to reason about, and avoids incorrect routing decisions that could occur if routing logic were delegated to probabilistic models.

4.2.1 Data Models

The service registry is implemented using SQLModel [18] to define the database schema and manage data persistence through an object-oriented interface. The data model is centered around the *chatbot* entity, which represents a domain agent, and is associated with various metadata used for discovery.

The schema is organized into separate tables for metadata categories, including countries, departments, and capabilities. These tables are linked to chatbots to enable filtering based on multiple attributes. In particular, the relationship between chatbots and capabilities is modeled as a many-to-many association, allowing a single chatbot to support multiple capabilities and each capability to be associated with multiple chatbots.

Chatbots are also associated with Foundry [20] project records that store connection details required for forwarding requests. This relationship is defined such that each chatbot is linked to a single project, while a project may be associated with multiple chatbots.

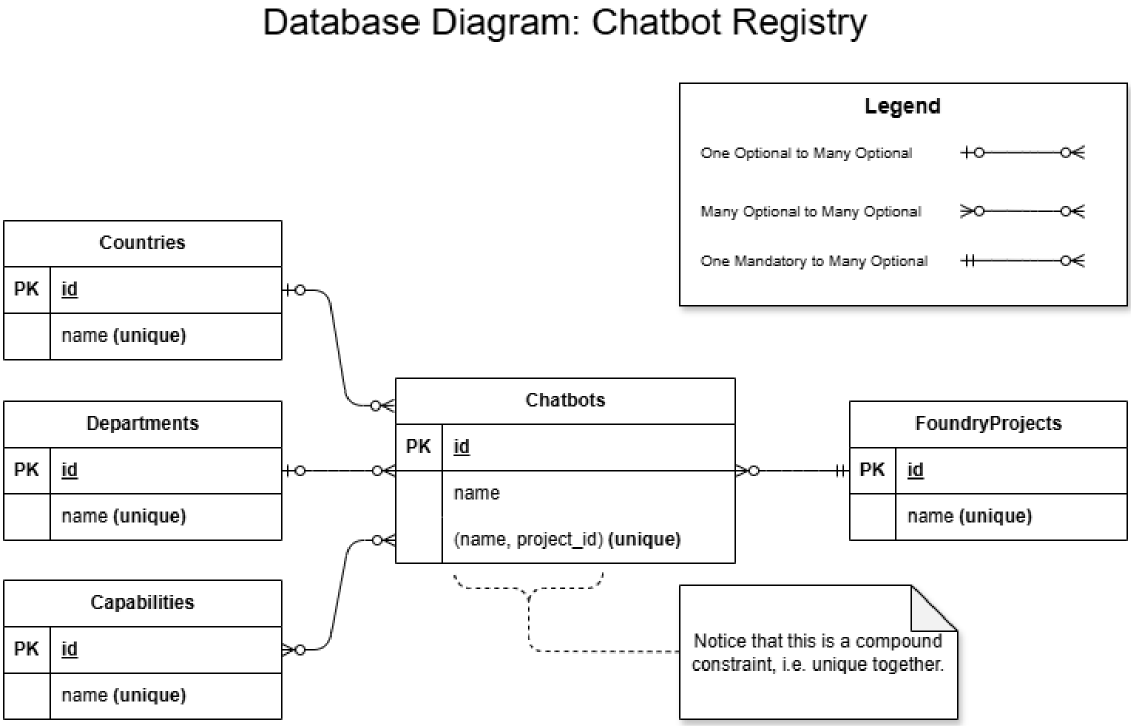


Figure 4.3: Database schema of the chatbot registry. The chatbot entity acts as the central component and is linked to metadata tables such as countries, departments, and capabilities. Each chatbot belongs to a single Foundry project, and a composite uniqueness constraint is enforced on the combination of chatbot name and project.

Figure 4.3 illustrates the structure of the service registry. The chatbot entity serves as the central component and is connected to metadata tables that support flexible filtering. The separation of metadata into distinct tables enables both many-to-many and many-to-one relationships while maintaining a normalized schema. The diagram also highlights the composite constraint that enforces uniqueness of chatbot names within each Foundry project, reflecting constraints imposed by the downstream system.

To maintain data integrity, several constraints are enforced at the schema level. These include unique naming constraints for metadata entities such as countries, departments, and capabilities, as well as a compound uniqueness constraint ensuring that chatbot names are unique within a given Foundry project. This ensures that registry data remains consistent with the requirements of the downstream environment.

The data model is designed to support efficient discovery and filtering. By normalizing metadata into separate tables and explicitly modeling relationships, the schema enables flexible queries while maintaining a clear structure. At the same time, it remains lightweight by storing only the information required to represent domain agents and support gateway functionality.

4.2.2 Search and Filtering

Discovery in the gateway is implemented as a deterministic filtering process over registry data. Rather than relying on probabilistic models to interpret user intent, the gateway applies explicit filters to domain agent records using their stored metadata.

The primary mechanism for discovery is the search functionality. It accepts a set of optional filters corresponding to metadata fields such as countries, departments, and capabilities. Each filter may contain multiple values (e.g., `countries=["Sweden", "Denmark", "Norway"]`). A record satisfies such a filter if any of the provided values matches the corresponding attribute, which corresponds to a logical `OR` within the filter.

When multiple different filters are provided, a record is included only if it satisfies all filter conditions. This corresponds to a logical `AND` across filters.

A central element of the discovery logic is fuzzy matching within each filter condition. Instead of requiring exact matches, filter values are compared using string similarity scoring provided by the RapidFuzz library [22]. Similarity scoring methods are commonly based on the idea that two strings are more similar when fewer small changes are needed to transform one into the other.

One way to understand tolerant matching is through edit distance. Edit distance counts the minimum number of simple operations required to transform one word into another, where an operation is either inserting a character, removing a character, or replacing one character with another.

For example, the edit distance between “rabbit” and “rapid” is 3: `rabbit` → `rabit` (remove `b`) → `rapit` (replace `b` with `p`) → `rapid` (replace `t` with `d`). A smaller number of required edits indicates that the strings are more similar.

The exact scoring functions and optimizations used by RapidFuzz are outside the scope of this report. In this project, the main point is that similarity-based comparison enables robust filtering even when provided filter terms are incomplete or slightly misspelled.

Pagination is applied uniformly to all discovery results. The gateway returns a subset of results defined by a limit and offset, regardless of whether filters are applied. When filters are present, pagination is applied to the filtered result set; otherwise, it is applied to the full set of registered domain agents.

4.3 The Gateway Interface

The gateway exposes an HTTP interface for discovery and interaction. The public interface is designed to be simple and stateless, enabling both traditional clients and MCP-based agents to discover candidate domain agents and send messages without maintaining server-side sessions. Administrative data management is provided through a restricted web UI mounted on the same application.

4.3.1 Public API

The public interface groups endpoints into two categories: *discovery* and *interaction*. Discovery endpoints provide access to registry metadata and search functionality, while the interaction endpoint forwards a message to a selected domain agent. Table 4.1 summarizes the public endpoints.

Table 4.1: Public gateway endpoints for discovery and interaction.

Method	Endpoint	Purpose
GET	`/chatbots/countries`	List available country values registered in the catalogue (results are paginated using `limit` and `offset`).
GET	`/chatbots/departments`	List available department values registered in the catalogue (results are paginated using `limit` and `offset`).
GET	`/chatbots/capabilities`	List available capability values registered in the catalogue (results are paginated using `limit` and `offset`).
POST	`/chatbots/_search`	Search for candidate domain agents using optional metadata filters; results are always paginated.
POST	`/chatbots/_message`	Forward a message to a selected domain agent.

Endpoints prefixed with an underscore (`_`) denote action-style operations, where the path name is intended to be read as a verb (e.g., `_search`, `_message`). This avoids ambiguity that can arise from noun-like endpoint names (for example, `message` could be interpreted as either a resource or an operation). These endpoints use `POST` to support structured JSON request bodies, which are better suited for operations that require complex or optional input fields.

The search functionality accepts three optional array fields: `countries`, `departments`, and `capabilities`. Any field may be omitted, and an empty request body represents an unfiltered query. All results are paginated using `limit` and `offset`, defaulting to `limit=100` and `offset=0` when not explicitly provided.

Messaging is implemented as a stateless operation. The request contains `chatbot_id` and `message`, and the gateway forwards the message to the selected domain agent and returns the response. No session state is maintained between requests. If the referenced chatbot does not exist, the gateway returns a structured error with HTTP status code 404.

The search functionality returns all matching candidate records rather than enforcing a single selection. This delegates decision-making to the entry-point conversational interface, which can either select a candidate directly or prompt the user to refine the query. The gateway thus provides a deterministic interface, while the agent orchestrates interaction with the user. An example of the end-to-end message flow through the gateway is shown in Figure 4.4.

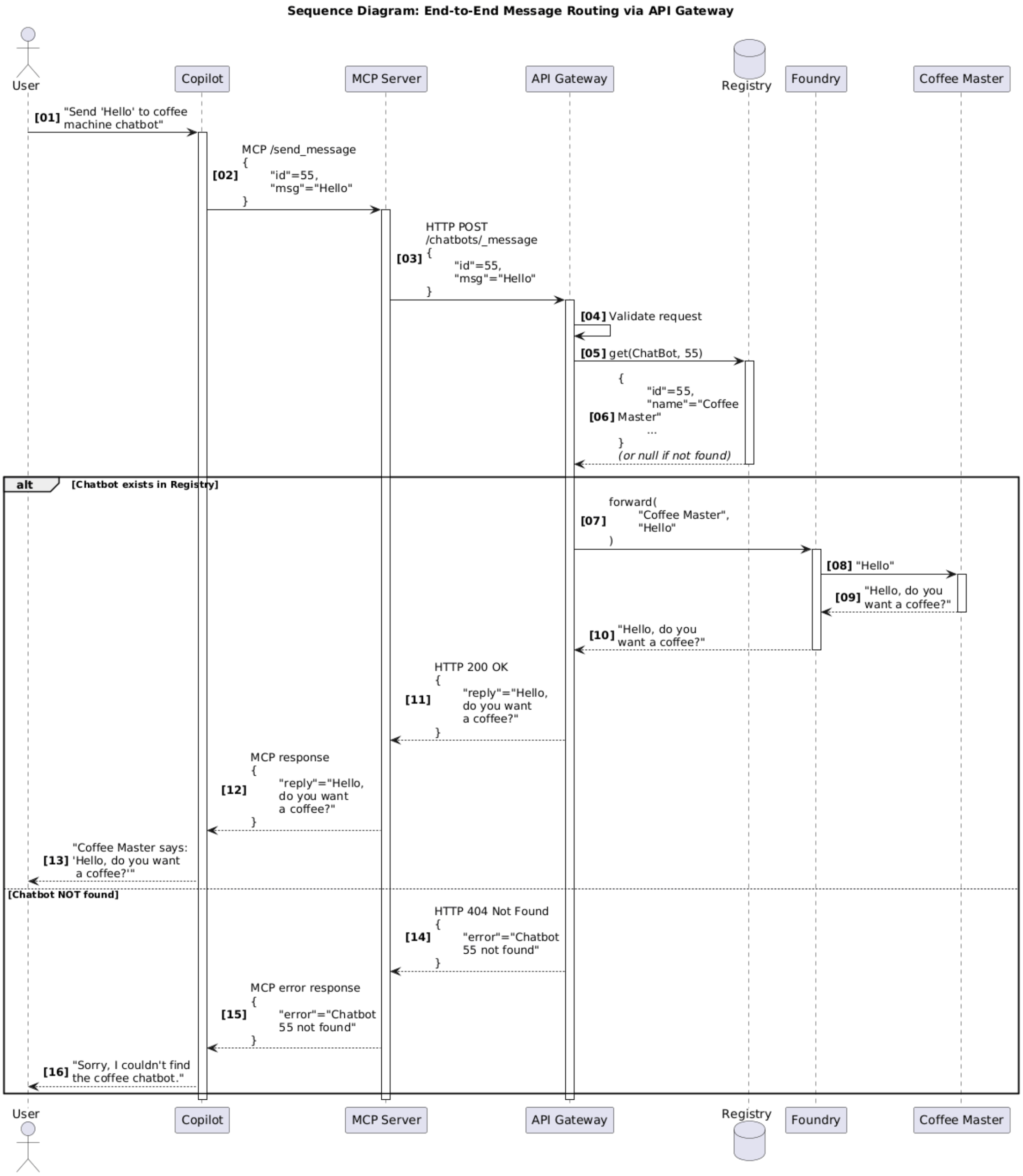


Figure 4.4: End-to-end message flow through the gateway, showing interaction between the user, MCP server, API gateway, registry, and downstream domain agent.

Figure 4.4 illustrates the full execution path of a messaging request. The sequence highlights how user input is translated into a tool call, forwarded as an HTTP request, validated by the gateway, and routed to a downstream domain agent. The diagram also shows how responses and errors are propagated back through the system.

4.3.2 Admin Interface

Administrative access is provided through a restricted web UI mounted on the same FastAPI application. The interface, implemented using SQLAdmin, provides read-write access to the registry for managing catalogue content and chatbot records. Link tables (also known as join tables, which are used to implement many-to-many relationships) are not exposed, as they are not intended for direct administrative interaction.

Authentication is handled through session-based login using credentials defined in the environment configuration. The interface currently supports CRUD operations on the exposed entity tables. Certain sensitive fields are restricted after creation to prevent accidental modification. For example, connection details and keys are encrypted when stored and are not intended to be edited through the UI.

4.3.3 Error Handling

Error handling is centralized to ensure that failures return consistent and actionable feedback. The gateway uses a custom exception type, `AIGatewayException`, which encapsulates an HTTP status code, an error code, a message, and a hint. A global handler converts these exceptions into a uniform JSON response format.

A simplified example from the implementation is shown below:

```
if not chatbot_record:
    raise AIGatewayException(
        status_code=404,
        code=ErrorCode.CHATBOT_NOT_FOUND,
        message="No chatbot with given ID is registered.",
        hint="Please check available chatbots via search."
    )
```

In this example, an invalid reference to a chatbot results in a structured exception. Each error contains a machine-readable code, a descriptive message, and an actionable hint, making it easier for both users and AI agents to understand the failure and determine appropriate next steps.

Similar patterns are used for other failure scenarios, such as connection errors to downstream services and access-related errors, ensuring consistent and informative error responses across the system.

4.4 MCP Integration

Agent integration is implemented using MCP. A separate MCP server process provides a tool interface for an MCP-capable assistant (Copilot). The MCP server exposes gateway operations as tools and translates tool invocations into HTTP requests to the Gateway API. This avoids requiring the assistant to construct HTTP requests directly and reduces the risk of hallucinated routes.

The MCP server is implemented in Python and runs as a separate process from the Gateway API. It exposes a fixed set of tools that correspond to supported gateway operations, which constrains the assistant to valid actions supported by the gateway.

4.4.1 Transport

The MCP server communicates with the MCP client over STDIO, which requires the MCP client and MCP server to run on the same machine. While the MCP client and MCP server must be co-located, the Gateway API can be deployed elsewhere as long as it is reachable over the network.

4.4.2 Tool Mapping

The MCP server exposes gateway capabilities as tools with structured input parameters and descriptions. This tool interface provides the assistant with the context needed to select an appropriate operation. In contrast to raw HTTP endpoints, tools explicitly describe purpose and expected arguments, which helps the assistant make informed choices.

With the exception of `describe_gateway`, each tool maps one-to-one to a Gateway API endpoint. The `describe_gateway` tool does not require access to the Gateway API and instead provides guidance about gateway purpose, available operations, and limitations when the assistant is uncertain.

The translation process is lightweight. The MCP server receives a tool invocation, constructs the corresponding HTTP request, and returns the gateway response as the tool result. If the Gateway API returns a structured error response, the MCP server forwards that error content back to the assistant, keeping failure handling consistent across both interfaces. Table 4.2 summarizes the mapping between MCP tools and Gateway API endpoints.

Table 4.2: Mapping between MCP tools and Gateway API endpoints.

MCP tool	HTTP method	Gateway endpoint
`describe_gateway`	—	—
`list_countries`	GET	`/chatbots/countries`
`list_departments`	GET	`/chatbots/departments`
`list_capabilities`	GET	`/chatbots/capabilities`
`search_chatbots`	POST	`/chatbots/_search`
`message_chatbot`	POST	`/chatbots/_message`

5

Results

This chapter presents the implemented system and demonstrates that the defined MVP has been successfully achieved. The results are presented through a series of verified system behaviors, supported by screenshots and execution evidence.

Each section focuses on demonstrating a specific aspect of the system rather than discussing the results, ensuring that the implemented functionality is clearly observable and verifiable.

5.1 Overview of Implemented MVP

To validate the outcome of the project, all components defined in the MVP scope were implemented and integrated into a functional system.

Component	**Status**
Conversational Interface	Implemented
Gateway API	Implemented
Routing Engine	Implemented
Service Registry	Implemented
Administrative Interface	Implemented
Messaging Functionality	Implemented
Logging	Implemented

Table 5.1: Overview of implemented MVP components

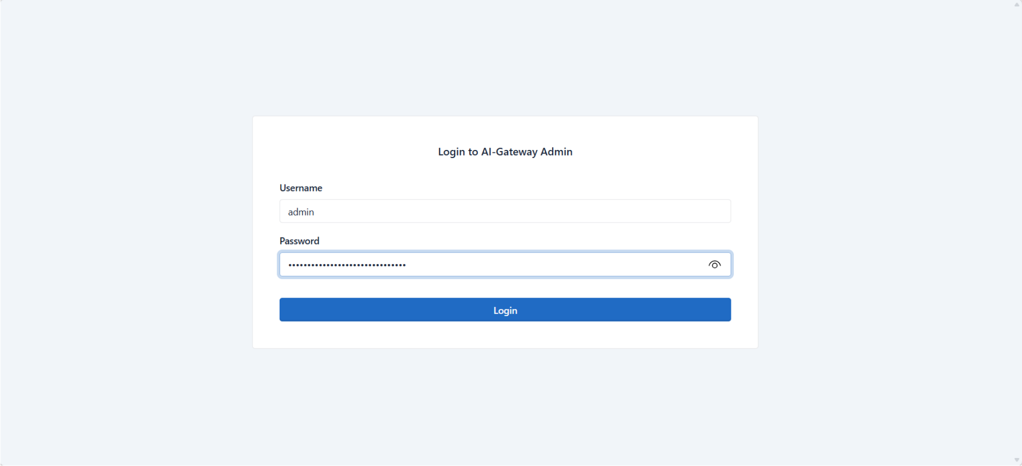
The table shows that all core system components defined in the MVP scope were implemented and integrated into the final prototype.

5.2 Administrative Interface

This section demonstrates how administrators interact with the system to configure and manage external AI service integrations through the administrative interface. These configurations enable the gateway to connect to external AI services and route requests to appropriate endpoints.

5.2.1 Login

Figure 5.1 shows the login interface of the AI-Gateway Admin panel.



Username
admin

Password
••••••••••••••••••••••••••••••

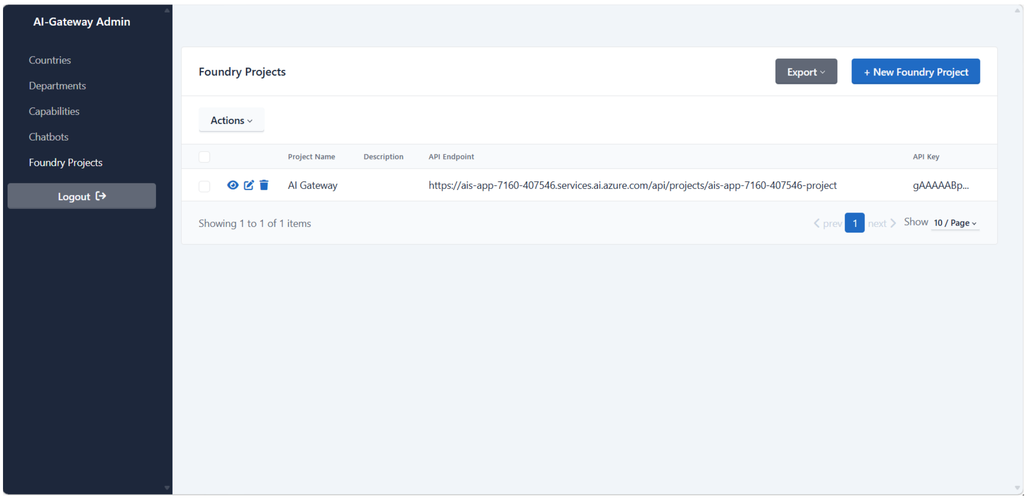
Figure 5.1: Administrator login interface

The login functionality was validated by attempting authentication using valid administrator credentials. After successful authentication, access to the administrative interface was granted, allowing management of external AI service configurations.

5.2.2 Data Management

This example demonstrates how a record can be added to a specific database table. The same principles apply to other data management operations, such as editing and deleting records across other tables.

The admin interface provides an overview of existing Foundry projects, including the project name, description, API endpoint, and encrypted API key. Figure 5.2 shows the admin interface with the “Foundry Projects” view selected and a single project record listed.

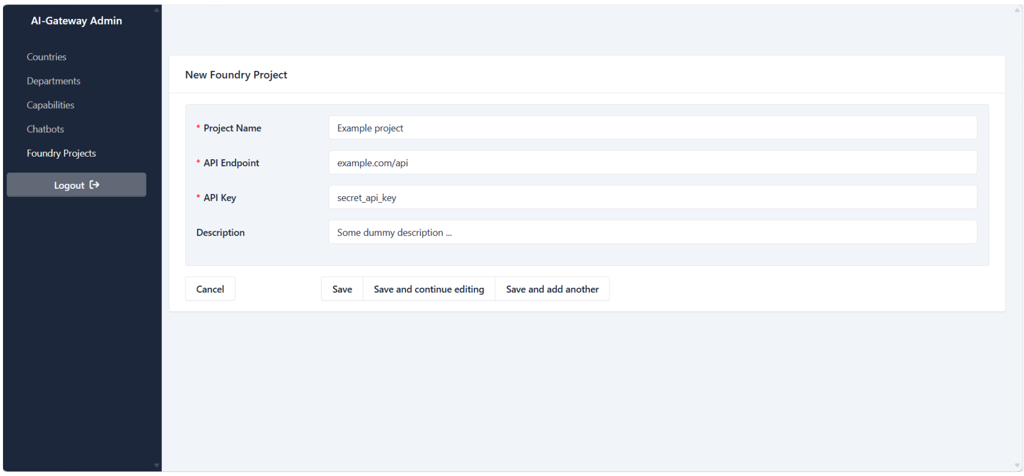


☐	Project Name	Description	API Endpoint	API Key
☐	AI Gateway		https://ais-app-7160-407546.services.ai.azure.com/api/projects/ais-app-7160-407546-project	gAAAABp...

Showing 1 to 1 of 1 items

Figure 5.2: Foundry project list before creating a new project

To create a new project, the administrator selects the "+ New Foundry Project" button, which opens the project creation form. Figure 5.3 shows the form used to define a new Foundry project configuration. The form requires the administrator to provide a project name, API endpoint, and API key.

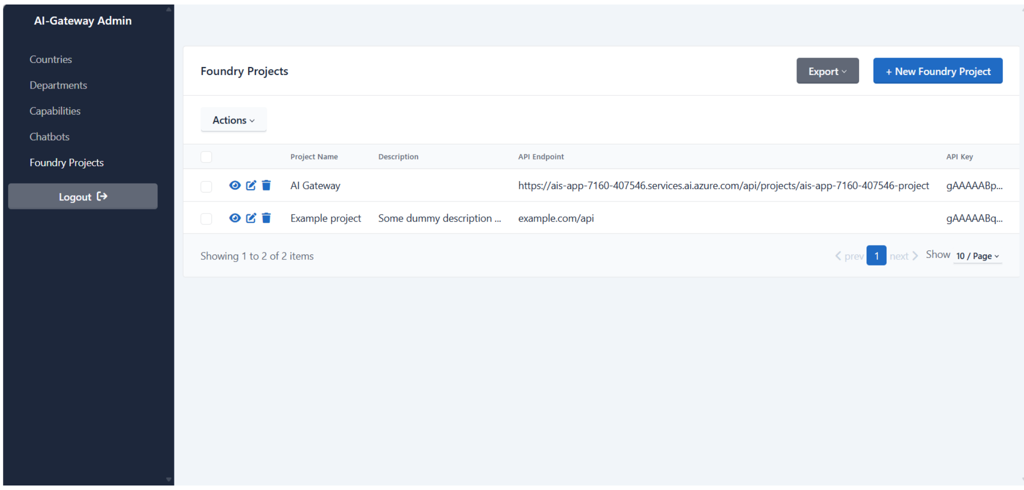


New Foundry Project

* Project Name: Example project
* API Endpoint: example.com/api
* API Key: secret_api_key
Description: Some dummy description ...

Figure 5.3: Form for creating a new Foundry project

To validate this functionality, a new project, named "Example project", was created by entering valid configuration data and submitting the form using the save button. Figure 5.4 shows the project list after the new project has been successfully created.



Foundry Projects

☐	Project Name	Description	API Endpoint	API Key
☐	AI Gateway		https://ais-app-7160-407546.services.ai.azure.com/api/projects/ais-app-7160-407546-project	gAAAAABp...
☐	Example project	Some dummy description ...	example.com/api	gAAAAABq...

Showing 1 to 2 of 2 items

Figure 5.4: Foundry project list after creating a new project

After submission, the newly created project appears alongside the existing project in the list view. The interface now displays two entries, confirming that the system successfully stored the configuration and updated the project registry accordingly.

5.3 Chat Interaction and Message Routing

The user-agent interaction in this experiment is conducted through GitHub Copilot in VS Code [23], which serves as the conversational interface between the user and the gateway. The underlying language model used by the agent is Claude Haiku 4.5 [24], responsible for interpreting user intent, orchestrating tool calls, and interacting with the gateway.

This section demonstrates the complete request-response cycle from user input to chatbot response through the gateway. The interaction below represents an execution trace of the system, demonstrating how the gateway performs chatbot discovery and message routing.

Input: User

Find the chatbot named "The-Technical-Coach" and verify that it successfully returns a response. Send a simple greeting such as "Hello world" to it.

Response: Copilot

I'll help you find and test "The-Technical-Coach" chatbot. Let me start by understanding the gateway structure, then search for and contact the chatbot.

Tool call: `describe_gateway` `mcp:1`

```
{}
```

Now let me search for "The-Technical-Coach" chatbot. Since the search tool filters by countries, departments, and capabilities, I'll do a broad search:

Tool call: `search_chatbots` `mcp:2`

```
{
  "limit": 100
}
```

Great! Found it. Now let me send the greeting message to verify it responds:

Tool call: `message_chatbot` `mcp:3`

```
{
  "chatbot_id": 1,
  "message": "Hello world"
}
```

Perfect! I found and tested "The-Technical-Coach" successfully. Here's the exchange:

Chatbot Details:

Result `rs`

Name: The-Technical-Coach
ID: 1
Department: Engineering
Capability: Technical advice (Handles technical advice and software engineering related queries)

Message Exchange:

Sent to chatbot: `io`

Hello world

Response received: `io`

Hello! How can I assist you today with technical documentation, onboarding, or upskilling in Product and Enablement?

Verification complete — The chatbot is online and responding successfully.

Gateway System Logs `sys`

```
POST /chatbots/_search?offset=0&limit=100 HTTP/1.1 200

SELECT department.id AS department_id, department.name AS department_name
FROM department
WHERE department.id IN (?)

ROLLBACK
BEGIN (implicit)

SELECT chatbot.id AS chatbot_id, chatbot.name AS chatbot_name,
       chatbot.country_id AS chatbot_country_id,
       chatbot.department_id AS chatbot_department_id,
       chatbot.project_id AS chatbot_project_id
FROM chatbot
WHERE chatbot.id = ?

```

```
SELECT foundryproject.id AS foundryproject_id,
       foundryproject.name AS foundryproject_name,
       foundryproject.endpoint AS foundryproject_endpoint,
       foundryproject.encrypted_api_key AS
          foundryproject_encrypted_api_key,
       foundryproject.description AS foundryproject_description
FROM foundryproject
WHERE foundryproject.id = ?

POST /chatbots/_message HTTP/1.1 200

ROLLBACK
```

5.4 Error Handling and Fault Tolerance

This section demonstrates system behavior under failure conditions. The interaction below represents an execution trace where the gateway attempts to communicate with a chatbot service but encounters a network-related error.

Input: User

Search for a chatbot related to “techncal advice” and verify that it returns a response.

Response: Copilot

I’ll search for chatbots with the “technical advice” capability and verify one responds.

Tool call: `search_chatbots` `mcp:1`

```
{
  "capabilities": [
    "technical advice"
  ]
}
```

Now let me send a test message to verify it responds:

Tool call: `message_chatbot` `mcp:2`

```
{
  "chatbot_id": 1,
  "message": "Can you provide some technical advice?"
}
```

Search Results:

Result `rs`

Name: The-Technical-Coach
ID: 1
Department: Engineering
Capability: Technical advice

Verification Attempt:

Sent to chatbot: `io`

```
Can you provide some technical advice?
```

Error Response: `io`

Access Denied - Connection Failure

You are likely trying to connect from outside the Volvo network. Please ensure your VPN is active. If you are on the VPN, ask your administrator to verify if your IP is allowlisted in the Azure AI Project.

5.5 Fuzzy Search in Service Discovery

This section demonstrates the system's ability to handle partial input through fuzzy search. The following interaction shows how the system retrieves relevant chatbot matches based on an incomplete query.

Input: User

Search for a chatbot with "tech" capability.

Response: VS Code Copilot

I'll help you find chatbots with "tech" capability. Let me first analyze the gateway and then search for matching chatbots.

Tool call: `describe_gateway` `mcp:1`

```
{}
```

Now let me search for chatbots with "tech" capability:

Tool call: `search_chatbots` `mcp:2`

```
{
    "capabilities": [
        "tech"
    ]
}
```

Found 1 chatbot with tech capability:

ID	Name	Department	Capabilities
1	The-Technical-Coach	Engineering	technical advice

Table 5.2: Search results for "tech" capability

The The-Technical-Coach chatbot specializes in technical advice and software engineering queries from the engineering department. Would you like me to send a message to this chatbot?

The result demonstrates that the fuzzy-search mechanism can retrieve relevant chatbot entries from partial user input, enabling more flexible service discovery.

6

Discussion

While the MVP was successfully delivered and the system meets its intended functionality, the development process made it clear that these results came with several compromises. In particular, working within an enterprise environment introduced constraints that had a noticeable impact on both design and implementation decisions. Time limitations, onboarding delays, and restricted access to development tools meant that there was less opportunity than expected to properly explore different architectural options.

Because of this, the implemented system should not be seen as a fully optimized solution, but rather as a pragmatic result shaped by real-world limitations. Some design choices worked well within the MVP scope, but others introduced weaknesses that would likely become more serious in a larger-scale system.

This chapter therefore aims to give a more balanced and critical evaluation of the system. It looks not only at what worked, but also at what did not, as well as how the project differed from the original plan and what practical lessons can be taken from the experience.

6.1 Project Evaluation

The main goal of this project was to design and implement a gateway that allows conversational AI agents to interact with distributed domain services in a structured and consistent way. Within the scope of the MVP, this goal was achieved.

One of the key outcomes is the MCP-based translation layer, which enables the agent to interact with backend services through structured tool calls. This worked well in practice and shows a clear way of connecting agent-based systems to traditional APIs. Instead of relying on the AI model to decide everything, the system separates decision logic from execution, which makes the overall behaviour easier to understand and control.

In addition, the deterministic, metadata-based discovery approach proved to be effective in this context. The results show that relevant services can be found without relying on LLM-based routing, which can sometimes be unpredictable. This is especially important in enterprise environments where consistency and traceability are required.

However, this effectiveness is strongly dependent on well-structured and well-maintained metadata. As the system scales and metadata becomes less controlled, this dependency may significantly impact both accuracy and reliability, particularly when handling more complex or loosely defined user queries.

6.2 Architectural Flaws

Even though the system works as intended, several architectural issues became visible during development.

One of the most important problems is the API design. The system currently uses hardcoded endpoints for metadata, such as `/chatbots/countries` and `/chatbots/departments`. While this approach was a simple and pragmatic choice during early development, it introduces clear limitations. In particular, due to limited development time and lack of sufficient architectural exploration, a more flexible design was not fully considered, and a solution that could be implemented quickly was prioritised.

Whenever a new metadata field is added, the API must be updated and new endpoints and MCP tools need to be created. This means the API and the MCP server are tightly coupled to the data model, which makes it difficult to extend.

A better approach would be to use a dynamic schema-based API. For example, endpoints like `/chatbots/schema` and `/chatbots/schema/list?field=...` would make it possible to handle new metadata fields without changing the codebase. This would significantly improve flexibility.

Another issue appears in the search mechanism. RapidFuzz allows for flexible matching, but at the same time it is difficult to control its behaviour. In particular, shorter strings tend to get higher similarity scores, therefore the similarity threshold should be set to a suitable value.

More importantly, the current discovery approach has limited ability to handle complex or loosely defined user queries, which reduces its effectiveness beyond simple matching scenarios.

There are also clear limitations in the administrative interface. SQLAdmin was a reasonable choice for quickly building the MVP, but it lacks basic features such as search and filtering. As the number of services increases, the interface becomes difficult to use and is not suitable for real enterprise scenarios.

Finally, the system lacks important functionality in terms of security and observability. It currently relies on HTTP and does not implement authentication or Role-Based Access Control (RBAC). This makes it impossible to support features such as rate limiting, request tracking, and auditing, which are essential in production environments.

6.3 Methodology Evaluation

The project was originally planned as an iterative development process, with continuous improvements and validation. While this approach provided a useful structure, the actual development did not follow the plan as closely as expected.

The main reason for this was the constraints imposed by the enterprise environment, including delays in onboarding processes, licensing approvals, and limited access to development tools. Onboarding procedures and licensing approvals took much longer than anticipated, which meant that a significant portion of the project timeline was lost before development could properly begin.

As a result, there was limited time left for exploring different design options or improving the architecture. Many decisions had to be made under time pressure, which led to a focus on getting the system working rather than making it scalable.

As a result, validation was primarily conducted through manual testing.
Automated testing was not implemented due to time constraints, which represents a limitation of the system and may affect its reliability and scalability.

More generally, this shows that in real-world projects, external constraints such as organizational processes can have a strong influence on technical outcomes.

6.4 Practical Insights

The project led to several practical insights that can be useful beyond this specific implementation.

First, it shows how agent-based systems can be built in practice by combining structured APIs with a translation layer like MCP. This highlights that, within the scope of this project, enabling agents to perform tasks depends not only on the AI model itself, but also on the design of the surrounding system.

Second, the project highlights some of the challenges of using metadata-based discovery. While the approach works well in simple cases, it becomes more difficult to manage as the system grows. Careful design is needed to maintain both scalability and usefulness.

Third, the project clearly shows the importance of preparation when working in an enterprise environment. Delays in getting access to tools and systems had a direct impact on both the timeline and the quality of the design.

These lessons can help guide future projects that aim to combine AI agents with distributed services.

6.5 Future Work

There are several ways in which the system could be improved in the future.

From an infrastructure perspective, deploying the system in a cloud environment such as Microsoft Azure would make it more scalable and reliable. Introducing a Continuous Integration and Continuous Delivery/Deployment (CI/CD) pipeline would also help automate testing and deployment, making the system easier to maintain.

While SQLite [25] is sufficient for the current size of the system, it may not handle concurrency and scalability well in a larger setting. For this reason, upgrading to a more robust database system such as PostgreSQL [26] would be a logical next step.

From a system design perspective, the discovery mechanism could be improved to better handle complex or loosely defined user queries. In addition, introducing automated registry management could reduce reliance on manually maintained metadata and improve scalability as the system grows.

Security and observability should also be improved. While Hypertext Transfer Protocol Secure (HTTPS) would improve basic security, it is not enough on its own. Without proper authentication and access control, the system cannot support essential features like rate limiting or auditing. Adding authentication, logging, and monitoring would therefore be necessary for production use.

Finally, the system could be extended to support aggregation of responses from multiple services. This would allow more complex use cases to be handled and improve the overall usefulness of the system. Additionally, developing a dedicated frontend with integrated MCP support could improve the user experience and reduce dependence on tools like VS Code.

7

Conclusion

This project set out to design and implement a lightweight gateway for coordinating access to distributed domain agents in an enterprise environment. The goal was not to centralize services, but to improve how users discover and interact with them through a unified interface.

The results show that this objective has been successfully achieved within the scope of the MVP. The implemented gateway provides a consistent interaction model that supports service discovery, selection, and messaging. By using metadata-driven filtering, users are able to identify relevant domain agents without prior knowledge of the system, addressing one of the main challenges outlined in Chapter 1.

The system also demonstrates that deterministic discovery, combined with a conversational interface, can provide a reliable and transparent alternative to AI-driven routing. While the gateway does not interpret user intent in a semantic sense, it offers predictable and explainable behaviour, which is particularly valuable in enterprise settings.

At the same time, the project highlights important limitations. The effectiveness of the system depends on well-maintained metadata, and the current approach has limited flexibility when handling more complex or loosely defined user queries. Additionally, the reliance on manual registry management may present scalability challenges as the system grows.

Despite these limitations, the project shows that even a relatively simple architectural approach can significantly improve how distributed services are accessed and coordinated. The combination of a gateway layer, metadata-based discovery, and agent interaction provides a solid foundation for further development.

Future work can build on this foundation by improving discovery mechanisms to better handle complex or loosely defined user queries, introducing automated registry management to reduce reliance on manually maintained metadata, and supporting aggregation of responses from multiple services. These extensions would further enhance both usability and scalability.

Overall, the project demonstrates a practical and scalable architectural approach that can support enterprise adoption of conversational AI systems in distributed environments.

Bibliography

[1] Mozilla. Server-side web frameworks. MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Web_frameworks

[2] Sebastián Ramírez. FastAPI. Official Website. [Online]. Available: https://fastapi.tiangolo.com/

[3] Django Software Foundation. Django. Official Website. [Online]. Available: https://www.djangoproject.com/

[4] Mozilla. HTTP request methods. MDN Web Docs. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods

[5] Microsoft. (2026) Use API gateways in microservices. Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/gateway?source=docs

[6] ———. (2026) Best practices for protecting secrets. Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/fundamentals/secrets-best-practices

[7] C. Michael Chernick, "Federal S/MIME V3 Client Profile," National Institute of Standards and Technology, NIST Special Publication 800-49, Nov. 2002. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-49.pdf

[8] Cryptography.io. (2026) Fernet (symmetric encryption). [Online]. Available: https://cryptography.io/en/latest/fernet/

[9] Amazon. (2026) What is Object-Relational Mapping (ORM)? Amazon Web Services. [Online]. Available: https://aws.amazon.com/what-is/object-relational-mapping/

[10] Python Software Foundation. RapidFuzz. Python Package Index. [Online]. Available: https://pypi.org/project/RapidFuzz/

[11] Microsoft. Introduction to agents for startups. Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/startups/build/ai/agents/intro-agents

[12] LF Projects. What is the Model Context Protocol (MCP)? Model Context Protocol. [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro

[13] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," *ACM Trans. Softw. Eng. Methodol.*, Feb. 2026, just Accepted. [Online]. Available: https://doi.org/10.1145/3796519

[14] LF Projects. Tools. Model Context Protocol. [Online]. Available: https://modelcontextprotocol.io/specification/draft/server/tools#tools

[15] Microsoft. (2026) GitHub. Official Documentation. [Online]. Available: https://docs.github.com/en/get-started

[16] Astral. uv. Official Documentation. [Online]. Available: https://docs.astral.sh/uv/

[17] Microsoft. VS Code. Official Website. [Online]. Available: https://code.visualstudio.com/

[18] Sebastián Ramírez. SQLModel. Official Website. [Online]. Available: https://sqlmodel.tiangolo.com/

[19] Amin Alaee. SQLAlchemy Admin for Starlette/FastAPI. GitHub Repository. [Online]. Available: https://github.com/smithyhq/sqladmin/tree/main

[20] Microsoft. Microsoft Foundry. Official Website. [Online]. Available: https://azure.microsoft.com/en-us/products/ai-foundry

[21] Samuel Colvin. Pydantic. Official Website. [Online]. Available: https://pydantic.dev/docs/validation/latest/concepts/pydantic_settings/

[22] Max Bachmann. RapidFuzz. GitHub Repository. [Online]. Available: https://github.com/rapidfuzz/RapidFuzz

[23] Microsoft. GitHub Copilot in VS Code. Official Website. [Online]. Available: https://code.visualstudio.com/docs/copilot/overview

[24] Anthropic. Claude Haiku 4.5. Official Website. [Online]. Available: https://www.anthropic.com/claude/haiku

[25] Hwaci. SQLite. Official Website. [Online]. Available: https://sqlite.org/

[26] The PostgreSQL Global Development Group. PostgreSQL. Official Website. [Online]. Available: https://www.postgresql.org/

INSTITUTIONEN FÖR något ämne
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS