POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Agent Engineering for the Enterprise: An MCP-Based Framework

Supervisor

Prof. Stefano QUER

Candidates

Alessio GIOÈ

Vincenzo CATALANO

A.A 2024/2025

Abstract

The rapid evolution of artificial intelligence has driven organizations across sectors to develop agents that genuinely augment human expertise and automate complex tasks. Retrieval-Augmented Generation (RAG) has proven to be a promising paradigm: it unites large language models with external knowledge retrieval to boost factual accuracy and domain relevance. However, the seamless integration of heterogeneous tools and context sources remains a thorny challenge.

The Model Context Protocol (MCP) is a lightweight and extensible framework designed to unify the exchange of structured context between RAG-powered agents and external services. MCP defines a clear JSON schema for context requests and responses, encompassing metadata, user session state, and tool interfaces. In particular, we found that standardizing these exchanges simplifies the orchestration of multi-modal capabilities, whether database queries, knowledge-base lookups, or custom computations.

MCP establishes a unified and extensible standard for context exchange, aimed at simplifying integration of AI agents with heterogeneous tools and services. By clearly defining request and response schemas, it reduces implementation ambiguity and promotes consistency between teams. The modular design of the protocol facilitates interoperability, accelerates development cycles, and reduces maintenance overhead, also helping enforce governance policies and scalable, stateful agent interactions. Ultimately, MCP provides a solid foundation for building reliable AI solutions in dynamic enterprise environments.

The project was developed collaboratively by the two candidates. Responsibilities were intentionally divided to exploit complementary expertise. This collaborative approach accelerated development, improved the robustness of the prototype through peer review, and ensured that both design and evaluation received balanced attention.

Acknowledgements

The path that has led us to the completion of this thesis has been long and challenging, marked by moments of difficulty but also by important growth, both personal and academic. We would like to express our sincere gratitude to all those who have supported us and contributed in various ways to the realization of this work.

First of all, we wish to thank our supervisor, Professor Stefano Quer, for his guidance, expertise and valuable advice, which have been crucial in shaping our research and improving the quality of our work. His insightful feedback has been a continuous source of motivation.

We are deeply grateful to AROL, whose collaboration has been essential to this project. In particular, we would like to thank Gianpaolo Macario, Alessio Chessa and Diego Bruno, who generously offered their time, technical expertise and strategic advice. Their guidance not only helped us to overcome practical challenges, but also provided valuable insights on how to direct the scope and objectives of our thesis towards meaningful and impactful outcomes.

We are also grateful to the researchers and developers whose contributions provided the foundations upon which our project was built. Without their prior work and shared knowledge, this thesis would not have been possible.

A heartfelt thanks goes to our families, who have always shown patience, understanding and support throughout these intense months. Their encouragement gave us the strength to keep going even in the most demanding times.

We would also like to thank our friends and colleagues for being present during this journey, for their willingness to listen, discuss ideas and provide suggestions, as well as for the moments of lightness that helped us maintain balance alongside our studies.

Finally, we reserve a special thought for those closest to us, who have always believed in our abilities and never stopped supporting us, even in moments of uncertainty. To all of them, we owe much of the achievement of this milestone.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

LLM
Large Language Models

RAG
Retrieval Augmented Generation

API
Application Programming Interface

MCP
Model Contex Protocol

QA
Quality Assurance

IT
Information Techonology

SDK
Software Development Kit

REST
Representational state transfer

VPN
Virtual Private Network

GPT

Generative Pre-trained Transformer

CoT

Chain-of-thought

Chapter 1

Introduction

In recent years, artificial intelligence has undergone rapid and transformative advances, driven in particular by the emergence of large language models (LLMs) capable of fluent natural language generation, contextual reasoning, and flexible interaction patterns. These capabilities have opened a new horizon for intelligent assistants within organizations: agents that do not merely answer questions but actively assist in workflows, orchestrate multi-step processes, and operate against heterogeneous enterprise systems. However, real enterprise adoption requires more than strong language capabilities; it demands reliable integration with existing tooling, auditable side effects, and governance mechanisms that respect authentication, authorization, and data residency constraints.

Early practical deployments often treated LLMs as monolithic problem solvers: developers relied on the models' generative abilities directly, prompting them to produce instructions, synthesize documents, or suggest actions. While this approach leverages the impressive generalist reasoning of modern models, it exposes several limitations when applied to operational settings. First, purely generative usage can lead to brittle and inconsistent behaviour: models may produce plausible but incorrect outputs (hallucinations), or they may overfit behavioural patterns seen in pretraining and fine-tuning data, yielding responses that are either too generic or excessively specialised for a particular prompt style. Second, LLMs operating without explicit tool interfaces lack the means to safely perform side effects: invoking an API, modifying a ticketing system, or scheduling a meeting requires precise authentication, error handling and auditable logs—features not natively provided by bare-text generation. Third, state management and long-term memory are difficult to guarantee through prompt engineering alone: maintaining coherent multi-turn interactions, respecting authorization scopes across conversations, and handling conversational context drift demand explicit mechanisms beyond single-turn generation.

These shortcomings motivated a transition from monolithic LLM usage to

agentic architectures. In agentic designs, the LLM is one component within a modular system where responsibilities are clearly separated: perception and retrieval modules ground the model in up-to-date and verifiable knowledge; a planning component decomposes complex requests into sub-tasks; specialised tooling components perform side-effecting operations under enforced contracts; and a memory or state store preserves long-term context and provenance. This decomposition brings several practical benefits. Grounding retrieval reduces hallucinations by providing explicit evidence for answers. Planners enable robust orchestration of multi-step flows, reasoning about dependencies and fallback strategies. Tooling modules encapsulate access controls and transactional guarantees, making it possible to invoke operations with explicit consent, logging, and rollback capabilities. Finally, modularity supports targeted testing, monitoring and incremental improvement, aligning system reliability with enterprise risk models.

Despite these advantages, moving to agentic architectures introduces new engineering challenges: the need for standardised interfaces between components, clear schemas for tool manifests, and runtime protocols that can mediate model requests while enforcing security and observability. Protocols such as the Model Context Protocol (MCP) aim to fill this gap by providing a structured, type-aware contract for how a model may request tools, how results are streamed back, and how side effects are recorded and audited. By separating the model's reasoning from execution semantics, MCP-style approaches enable developers to retain the flexibility of LLM reasoning while imposing the operational constraints required in production.

This thesis documents the design, implementation, and evaluation of a task-oriented conversational agent developed for AROL S.p.A. The project was driven by the concrete goal of supporting the company IT department in common operations such as ticket lifecycle management, document retrieval, meeting scheduling and basic diagnostics. From the outset we prioritized pragmatic constraints: minimal disruption to existing user workflows (hence Microsoft Teams as the primary entry point), strict conformance to corporate authentication flows, and clear provenance for any side-effecting operation. To satisfy these requirements we combined Microsoft Copilot Studio's RAG capabilities with a custom MCP server that exposes typed tools and controlled sampling of contextual data. The outcome is a layered architecture in which the LLM handles reasoning and natural language grounding, while MCP mediates discovery, sampling, and explicit tool invocation under schema-driven contracts.

On the implementation side, several concrete engineering decisions and contributions are documented. First, to provide an up-to-date retrieval backbone we developed an ingestion pipeline that exported historical Jira artifacts (issues, comments and attachments), normalised them, converted heterogeneous attachments to PDF, and indexed chunks and embeddings for SharePoint ingestion; this

allowed Copilot Studio's RAG layer to reference ticket-level provenance directly when answering user queries. The ingestion pipeline included practical heuristics for chunking, OCR of image-only PDFs, and metadata sidecars to preserve traceability to original Jira keys. Second, we implemented a custom MCP server and iterated on SDK-level behaviour: early prototyping used the MCP Python SDK for rapid iteration and we contributed fixes upstream to stabilise transport reconnection, manifest parsing and streaming tests. Third, because production-grade integration with Microsoft services required more reliable OAuth handling, we migrated critical parts of the server to a TypeScript stack based on the official reference, improving interoperability with Copilot Studio and reducing deployment friction.

The thesis also examines operational considerations that surfaced during the pilot deployment. We deployed the MCP server on Azure Web Services, connected Copilot Studio to SharePoint-backed RAG indices, and integrated the conversational front-end within Microsoft Teams to reduce end-user friction. A structured validation phase with company employees produced broadly positive feedback: users appreciated faster access to relevant documentation and the conversational interaction style, while QA and IT highlighted important non-functional areas to address, such as latency sensitivity, context-management across turns, and the need to prevent unintended disclosure of sensitive fields unless the requester is properly authorised. Moreover, the validation highlighted an architectural fragility: the hybrid dependence on our self-managed MCP server and Microsoft's managed services implies that outages or behavioural changes in any layer can interrupt the conversational flow; practical mitigations include fallback modes, improved observability and runbooks, and contract-driven monitoring.

Contributions of this work are threefold. Practically, we deliver a working prototype that integrates Copilot Studio, a SharePoint-backed RAG index sourced from Jira, and a schema-driven MCP server capable of authenticated tool invocations suitable for IT workflows. Technically, we report and upstream fixes to the MCP Python SDK, describe migration challenges and solutions to a TypeScript implementation, and present reproducible patterns for manifest validation, streaming testing (Inspector-driven fixtures) and CI contract tests. Finally, methodologically, we outline an evaluation strategy for enterprise pilots that combines qualitative user feedback, telemetry-informed analytics and a clear distinction between development flexibility and production governance.

The work presented in this thesis is the result of a research and development effort carried out collaboratively by the two candidates. To exploit complementary skills and optimize implementation time, responsibilities were deliberately divided: Vincenzo Catalano took charge of Chapters 3–6, focusing on the design and reasearch of the system, while Alessio Gioè developed Chapters 7–9, concentrating on integration, development, and practical evaluation. The remaining chapters were written jointly. Moreover, although primary responsibility for individual

chapters was assigned, both authors continuously assisted one another and carried out reciprocal peer review of drafts and implementations. This organization enabled parallel development, improved the quality of the prototype through continuous cross-checking, and contributed to making the description of the adopted methodologies clearer and more reproducible.

The remainder of this thesis is organised as follows. Chapter 2 formalises the problem statement and constraints. Chapter 3 summarises the requirements and the use-case analysis that guided the design. Chapter 4 discusses architectural alternatives and motivates the choice of Copilot Studio and MCP. Chapter 5 describes the RAG pipeline and the Jira-to-SharePoint ingestion process. Chapter 6 presents MCP concepts, SDK work and our specific server implementation. Chapter 7 documents the Azure deployment and operational practices. Chapter 8 reports the validation campaign, results and lessons learned. Finally, Chapter 9 concludes and outlines directions for future work.

Chapter 2

Problem Definition

In modern enterprises, the reliance on digital tools and software platforms is ubiquitous. Organizations use a variety of systems to manage operations, track data, and facilitate collaboration across teams and departments. These systems often range from legacy databases and custom applications to cloud-hosted services, REST APIs, and message queues. Each tool embodies its own design philosophy, interaction model, and security paradigm. Consequently, the integration of multiple systems into a coherent workflow poses significant challenges, as each system expects specific communication patterns, data formats, and authentication mechanisms. The lack of a unifying interface or standard makes it difficult to orchestrate these tools effectively, particularly when automation is desired at scale.

One of the central challenges arises from the heterogeneity of enterprise systems. Every platform may use distinct protocols, implement different data schemas, and expose unique operational semantics. For example, a ticketing system may provide a REST API with JSON responses, while an internal database exposes SQL queries with strict schema constraints. Similarly, message brokers or event-driven architectures may require asynchronous handling and acknowledgment semantics that differ from synchronous API calls. Bridging these diverse paradigms demands the development of translation layers or "glue code" that can harmonize interactions, convert data formats, and manage error propagation. Without such careful integration, even a simple multi-step operation spanning multiple systems can fail unpredictably or produce inconsistent results.

Beyond technical heterogeneity, authentication and authorization represent a further layer of complexity. Modern enterprise environments enforce strict access controls, often employing OAuth2, token-based authentication, or enterprise single sign-on mechanisms. An AI agent orchestrating multiple tools must navigate these flows correctly, ensuring that credentials are valid, scopes are respected, and sensitive data is never exposed improperly. Misconfigured authentication can result in operational failures or, worse, security breaches. Moreover, different

systems may impose additional constraints, such as rate limiting, multi-factor authentication, or token expiration, which the orchestrating agent must handle dynamically. Designing a solution that reliably manages these interactions requires careful attention to detail and robust error handling to maintain service continuity and security compliance.

Maintainability and resilience are equally critical in enterprise-scale automation. As the number of integrated systems grows, the orchestration layer often becomes increasingly complex. Tracking the source of failures—whether originating in a tool, an integration component, or the agent itself—can be extremely challenging. Without clear modularization, standardized interfaces, and comprehensive logging, diagnosing and mitigating issues may require significant manual effort. Furthermore, the dynamic nature of enterprise software means that tools evolve, APIs change, and internal processes are updated. A solution that is brittle or tightly coupled to specific versions of systems risks becoming obsolete rapidly, leading to downtime or operational inconsistencies. Therefore, long-term maintainability requires the adoption of structured design principles, rigorous testing pipelines, and observability mechanisms that allow teams to detect, diagnose, and resolve issues efficiently.

Building an AI agent capable of managing heterogeneous enterprise systems is more than a simple software engineering task; it is a multi-dimensional problem involving systems design, security, operational reliability, and user experience. It involves orchestrating interactions across multiple services while respecting authentication and access controls, preserving data integrity, and maintaining coherent operational workflows. Moreover, it requires the ability to handle unexpected failures gracefully, provide transparency and traceability for all operations, and scale to support multiple concurrent users or processes. In essence, the challenge is to design a layer that abstracts the inherent complexity of enterprise ecosystems, enabling AI agents to act reliably and predictably while minimizing manual oversight.

The need for robust agentic architectures emerges from these challenges. Traditional approaches in AI-driven automation often relied on monolithic systems or directly prompting models to perform actions. While large language models offer impressive capabilities in natural language understanding and generation, their use as standalone reasoning engines exposes significant limitations in operational environments. Generative outputs may be plausible but incorrect, contextual dependencies may be lost across multi-turn interactions, and side effects such as API calls or database updates require strict auditing and transactional guarantees that language models alone cannot enforce. These limitations underline the necessity of separating reasoning from execution and establishing well-defined interfaces between components.

In response, agentic systems introduce modularity and structured orchestration. In such designs, an AI model is one component among several, interacting with

planning, retrieval, and tooling modules. Retrieval modules provide grounding and evidence-based responses, planners break complex tasks into manageable sub-tasks, and specialized tooling modules perform actions under clearly defined contracts. State management components maintain context and provenance, enabling long-term memory and traceable operations. This decomposition addresses multiple enterprise concerns simultaneously: it reduces hallucinations by grounding decisions in verifiable data, ensures secure and auditable execution of side effects, and allows targeted monitoring and testing of each module independently. Ultimately, this approach enables AI agents to operate at scale in environments that demand reliability, accountability, and compliance.

Finally, the operationalization of such systems introduces additional engineering considerations. Standardized interfaces between modules, schema-driven contracts for tools, and runtime protocols that enforce security and observability are essential. Model Context Protocols (MCP) exemplify frameworks designed to address these challenges, providing structured contracts for tool invocation, streaming of results, and logging of side effects. By decoupling the reasoning performed by the model from the execution semantics of underlying tools, such frameworks allow developers to leverage the flexibility of AI while imposing the operational constraints required for enterprise deployment. Through careful design, testing, and validation, agentic architectures empowered by structured protocols can deliver the benefits of AI reasoning without compromising security, reliability, or maintainability.

In this work, we focus specifically on the development of an AI agent for the IT operations of a mid-sized enterprise. The agent is designed to integrate seamlessly with existing enterprise tools, respecting their diverse authentication mechanisms and operational semantics, while providing users with a conversational interface for common tasks such as ticket management, document retrieval, and workflow automation. By abstracting the heterogeneity of underlying systems and enforcing strict contracts for tool usage, the proposed framework demonstrates a practical approach to enabling AI agents to operate reliably in real-world enterprise contexts. The following chapters describe in detail the problem space, design requirements, architectural decisions, and implementation considerations that guided the development of this system.

Chapter 3

System Design

In order to design a reliable and enterprise-ready system, it is essential to begin with a structured definition of requirements. This phase provides the foundation on which all subsequent design and implementation activities are built, ensuring that the solution aligns with organizational objectives, user expectations, and technical constraints. The following sections present the methodology adopted for requirements gathering and the resulting functional and non-functional specifications.

3.1 Requirements Gathering

The process of requirements gathering represents one of the most critical phases in the development of complex software systems, especially when the target solution involves interaction with heterogeneous platforms and end-users with diverse needs. This activity goes beyond the mere collection of technical specifications; it requires a structured approach that combines observation, stakeholder engagement, and iterative validation. The objective is to capture not only what the system must do, but also the constraints, expectations, and contextual conditions under which it will operate.

A fundamental aspect of requirements gathering is the identification of stakeholders and the analysis of their needs. In the context of an AI-powered assistant, this involves employees at different organizational levels, IT administrators, and even external service providers who contribute to the company's digital ecosystem. Each of these actors has distinct expectations: employees demand a tool that simplifies daily operations and reduces friction in tasks such as support requests, knowledge retrieval, and communication management, while IT departments focus on security, compliance, and integration reliability. Balancing these perspectives ensures that the final system aligns with both usability goals and enterprise-grade robustness.

Another important dimension of this phase concerns the definition of the operational environment. The assistant must function within an enterprise ecosystem that includes ticketing systems, email platforms, scheduling tools, and knowledge repositories, each with its own access policies, APIs, and technical constraints. This requires the early identification of integration points, dependencies, and possible incompatibilities between systems. The analysis of authentication methods, data formats, and service limitations allows development teams to anticipate potential barriers and design appropriate mitigation strategies.

Finally, requirements gathering also includes the prioritization and validation of the features to be implemented. Not all requirements carry the same weight: some are essential for system viability, such as secure access and basic support ticket management, while others enhance usability or efficiency, like image-based diagnostics or advanced scheduling features. Through iterative validation with stakeholders, the development team can refine these priorities and ensure that the assistant evolves in response to real organizational needs rather than abstract assumptions.

3.2 Requirements

Based on the insights gathered during the requirements elicitation phase, a comprehensive set of functional and non-functional requirements was established to guide the design and development of the proposed system. The definition of these requirements plays a central role in bridging the gap between the expectations of stakeholders and the concrete technical solutions that can be implemented. While functional requirements describe the services, behaviors, and interactions that the assistant must support in order to provide tangible value to its users, non-functional requirements capture broader quality attributes such as performance, reliability, scalability, and security.

This dual perspective ensures that the resulting system is not only capable of carrying out specific tasks but also operates within the standards and constraints expected in an enterprise environment. For instance, functionality related to support ticket management, knowledge base access, and meeting scheduling must be complemented by guarantees of data protection, compliance with authentication protocols, and resilience under heavy usage. By explicitly distinguishing between what the system should do and how it should perform, the development team can create a roadmap that balances user-facing features with underlying technical robustness.

Furthermore, these requirements serve as a reference point throughout the development lifecycle. They provide measurable criteria against which prototypes can be evaluated, facilitate communication among stakeholders with different

technical backgrounds, and reduce ambiguity by formalizing expectations in advance. In the specific context of AROL, this structured set of requirements ensures that the proposed assistant integrates smoothly with the existing IT infrastructure while simultaneously addressing the daily operational challenges faced by employees. The combination of functional and non-functional specifications thus establishes a solid foundation for subsequent design choices, architectural decisions, and validation activities.

3.2.1 Functional Requirements

The functional requirements define the core capabilities expected from the system and the interactions it must support with users and external platforms. Central to its functionality is the management of support tickets: the assistant must be able to assist employees in creating, updating, and resolving tickets, including the automatic escalation of issues that cannot be addressed autonomously. In addition, the system is expected to provide efficient access to the company knowledge base, retrieving the most relevant and up-to-date documentation upon request. Scheduling and meeting management are also fundamental, requiring the assistant to interact seamlessly with Microsoft Teams to organize meetings, manage participants, and deliver timely notifications. Email handling represents another critical functionality, as the system must allow users to send, receive, and review Outlook messages directly through a conversational interface. Furthermore, the assistant is expected to support image-based diagnostics by processing screenshots of error messages, such as those generated by Outlook, and providing targeted, context-aware troubleshooting guidance.

3.2.2 Non-Functional Requirements

Non-functional requirements specify the quality attributes, operational constraints, and performance expectations of the system. High performance is essential, with responses delivered within a few seconds to ensure smooth usability during daily workflows. Scalability is another key consideration, as the system must accommodate an increasing number of users and requests without compromising service quality. Maintainability is addressed through modular design, enabling future extensions or integration of new tools with minimal modification. Security requirements mandate compliance with corporate authentication and authorization standards, such as OAuth2 and Microsoft Graph protocols, to guarantee confidentiality, integrity, and controlled access to all communications and data exchanges. Reliability is also critical, with the system designed to maintain high availability, handle errors gracefully, and recover from partial failures of external services. Finally, compliance with company policies and relevant data protection regulations is enforced for all

interactions with external APIs, ensuring that the system operates within legal and organizational frameworks.

3.3 Actors

The system involves a set of actors, both human and digital, that interact with the AI assistant to support daily operations within AROL. The primary actor is the employee, who represents the end user interacting with the AI assistant through Microsoft Teams. Employees rely on the assistant to submit requests, pose questions, and receive guidance in completing routine IT and administrative tasks, benefiting from a conversational interface that streamlines interaction with various company services.

Supporting actors include several digital systems and services that enable the assistant to fulfill its functions. The AI assistant itself acts as an intelligent agent, interpreting employee requests, orchestrating workflows, querying external services, and delivering results in natural language. Microsoft Teams serves as the main communication channel, providing both the conversational interface and the message delivery infrastructure. The Microsoft Graph API offers unified access to Outlook, Calendar, and Directory services, allowing the assistant to send and retrieve emails, manage calendar events, check participant availability, and query organizational data efficiently. SharePoint, accessed through Microsoft RAG, functions as the company's knowledge base, granting the assistant access to official documentation and guides. For project management and development tracking, the assistant integrates with Jira Software, enabling the creation, updating, and retrieval of tasks to support ongoing projects. Similarly, Jira Service Management is used for IT service requests and support tickets, allowing the assistant to open, escalate, and track incident reports. Finally, external services such as wttr.in provide weather information upon employee request, demonstrating the assistant's ability to access and present real-time data from outside the corporate ecosystem.

The system also indirectly benefits key stakeholders within the organization. IT support staff gain efficiency, as the assistant reduces the volume of repetitive requests and allows them to focus on more complex issues. Company management, on the other hand, has a vested interest in the efficiency, compliance, and reliability of the assistant, ensuring that it contributes positively to overall business workflows.

In summary, the actors involved in the system encompass both the direct users, represented by employees, and the digital services integrated via APIs, with the AI assistant serving as the central coordinator. This comprehensive classification facilitates the subsequent analysis of use cases by clearly identifying the roles and interactions within the system.

3.4 Use Case Analysis

The software system developed in this work is designed to act as an intelligent assistant within a corporate IT environment, enabling automation and streamlining of various daily tasks typically handled by support staff. Its capabilities span several key use cases that demonstrate the integration of natural language understanding, image recognition, and enterprise API interaction.

One fundamental use case is **support for company software usage**, such as configuring secure VPN access for remote work. In this scenario, the AI system interprets the user's request and provides guided, step-by-step instructions, including security best practices and troubleshooting suggestions. This reduces the need for human intervention in standard IT setup procedures, improving efficiency and minimizing delays.

Another important use case is **software issue resolution**, where employees can report problems by uploading screenshots of error messages (e.g., in Outlook). The AI system leverages image analysis to understand the issue and returns a tailored resolution process. This not only speeds up problem-solving but also empowers employees to resolve issues independently without relying on IT personnel.

The software also includes a **document retrieval function**, allowing users to request specific versions of internal documentation. The system interfaces with a centralized knowledge base to identify and deliver the most recent version of the requested file, saving time and ensuring version consistency across teams.

In cases where the AI cannot fully resolve an issue, the system is capable of **issuing a support ticket**. By collecting user input (including descriptions and attachments), it automatically generates and submits a Jira ticket through API integration, ensuring smooth escalation and traceability of unresolved issues.

Furthermore, the solution enhances collaboration through features like **Teams meeting management**. Users can message the bot directly in Microsoft Teams to schedule meetings. The bot collects necessary details, participants, date, time, and uses the Microsoft Graph API to create and distribute the calendar invite, eliminating manual coordination.

Finally, the system supports **email dispatch via Teams**, enabling users to send Outlook emails simply by chatting with the bot. The assistant parses the content, identifies recipients, and sends the message using the same Microsoft Graph interface, allowing communication tasks to be completed quickly and conversationally.

Overall, these use cases illustrate the system's versatility and the practical value of AI-driven automation in enterprise workflows. By reducing the need for manual interventions in routine operations, the software improves response times, enhances user autonomy, and integrates seamlessly with existing company tools and platforms.

In order to provide a structured overview of the AI Assistant's functionalities,

the use cases can be grouped into four main categories: **Knowledge Access**, **Communication & Messaging**, **Calendar & Scheduling Management**, and **Task & Issue Management**. Each category highlights a specific aspect of the assistant's role within the corporate environment, illustrating how it supports employees in accessing information, coordinating activities, and managing tasks efficiently.

Knowledge Access It encompasses scenarios in which the assistant provides existing information or documentation without modifying external data sources. The first use case within this category involves the retrieval of documentation from SharePoint. In this context, the employee represents the primary actor, while the assistant and the SharePoint platform operate as supporting systems. The scenario assumes that the requested material is already available and correctly indexed within SharePoint. When the employee expresses a need in natural language, such as requesting a "VPN setup guide," the assistant formulates a query and searches the document repository via Microsoft RAG. The system then identifies the most relevant version of the document and delivers it seamlessly within the Teams chat environment. If multiple plausible results are found, the assistant actively engages with the employee by requesting further refinement, ensuring the accuracy and relevance of the delivered content.

Within the same category, the assistant also facilitates access to external information sources, such as weather data. Here, the employee specifies a valid city name, and the assistant forwards the request to the external service *wttr.in*, which responds with up-to-date meteorological information. The assistant then presents the results in Teams, making the information immediately accessible. In cases where the city name is incomplete or invalid, the assistant prompts the employee for clarification, thereby minimizing errors and ensuring the provision of contextually appropriate data.

By grouping these use cases, it becomes evident that the assistant primarily acts as an informational intermediary in the *Document Retrieval / Knowledge Access* category, seamlessly bridging the gap between the employee's natural language queries and structured data repositories, whether internal or external.

Communication & Messaging It encompasses scenarios in which the assistant facilitates corporate communications, whether through email or internal messaging, thereby reducing context switching and improving workflow efficiency. Within this category, the assistant supports sending emails directly from Teams. When the employee provides the necessary elements, such as recipient, subject, and message body, the assistant interprets the request, structures the email appropriately, and dispatches it through the Microsoft Graph API to Outlook. Following successful delivery, a confirmation is returned to the employee within Teams. If an invalid

recipient address is detected, the assistant requests clarification or correction before proceeding, ensuring reliability and minimizing communication errors.

In addition to sending messages, the assistant enables rapid inspection of the mailbox by listing recent emails. Upon the employee's request, the system interprets the number of messages to be displayed, applying a default of five if not specified, and retrieves the corresponding entries from the mailbox. Each message is presented in Teams with sender, subject, and date. To maintain system performance and user experience, the assistant enforces an upper limit of fifty messages and automatically adjusts the output when the requested number exceeds this threshold. This capability allows employees to remain informed without switching between applications, streamlining their workflow within the conversational context of Teams.

Complementing these email-focused operations, the assistant also addresses collaborative and directory-oriented needs. When the employee provides a locating cue, such as an office location or a department name, the assistant queries the corporate directory via the Microsoft Graph API and returns a structured roster of matching colleagues, including role, email address, and telephone number where permitted. The results are presented inline within Teams and can be used as actionable items: the employee can initiate a direct conversation, prefill meeting invitations, or select participants for availability checks, all without leaving the current conversational context. Collectively, these features highlight the assistant's role as an intermediary that streamlines communication and coordination across the organization.

Calendar & Scheduling Management It encompasses all operations related to planning, coordinating, and optimizing the availability of colleagues. Within this domain, the assistant interacts closely with the Microsoft Graph API to manage calendar events efficiently, serving as a coordinator that streamlines scheduling and reduces administrative overhead.

Calendar-related requests include the creation, modification, and deletion of meetings. When an employee initiates a request, the assistant collects the necessary parameters, such as the event title, date, time, participants, and, when applicable, an existing event identifier. For updates or cancellations, the assistant first validates the provided event ID to ensure that it corresponds to an existing entry. Once the parameters are verified, the assistant invokes the Calendar API to perform the requested operation and confirms the outcome by returning salient details, including the event identifier or a Teams meeting link. In the case of scheduling conflicts due to participant unavailability, the assistant actively contributes to resolution by suggesting alternative time slots, thereby minimizing negotiation overhead and preserving calendar consistency.

In addition to managing individual events, the assistant facilitates the planning

of group activities by checking team availability. When the employee provides a list of participants along with a preferred time interval, the system leverages the corporate directory to prefill participant information and queries the calendar service for availability data. The assistant then identifies all time slots in which every participant is free and returns a ranked set of candidate options. If no common availability exists, the assistant proposes the closest alternatives, ensuring that scheduling conflicts are efficiently mitigated without requiring extensive manual coordination.

The assistant also supports the refinement of existing events. Employees can request updates to an event's title, schedule, or list of participants, referencing an existing event ID. The assistant validates the identifier, applies the requested modifications via the Calendar API, and provides confirmation with relevant metadata. If the identifier is invalid, ambiguous, or no longer exists, the system prompts the employee for clarification to prevent unintended changes and maintain the integrity of the calendar.

By combining these capabilities, the assistant acts as a central facilitator for both individual and group scheduling, integrating event management with participant availability checks, updates, and cancellations. This comprehensive approach ensures that employees can efficiently organize their time and coordinate with colleagues, all within the conversational context of Teams.

Task & Issue Management The final category, **Task & Issue Management**, focuses on the assistant's capabilities to support employees in handling project tasks, issue tracking, and IT support requests directly within Teams. Central to this category is the management of Jira tickets, which allows employees to create, update, search, comment on, and close tickets without leaving the conversational interface.

When creating a new Jira ticket, the assistant collects all necessary information, including the ticket title, description, priority, and assignee, and uses the Jira API through MCP to generate the ticket within the appropriate project. The system returns the ticket identifier to the employee, and if any information, such as the project key, is missing or unclear, the assistant proactively requests clarification to ensure the ticket is correctly created and assigned.

For existing tickets, the assistant enables modifications to key attributes such as status, priority, or assignee. Prior to making any changes, the assistant validates the ticket identifier to ensure that the requested update applies to a valid entry. Once confirmed, the update is performed via the Jira API, and the employee receives a confirmation in Teams. In cases where the ticket ID is ambiguous, the assistant requests clarification, thereby preventing errors and maintaining accurate task tracking.

The assistant also streamlines the retrieval and oversight of Jira tickets. Employees can request lists of open tickets or apply filters based on assignee, status, or tags. The assistant queries Jira Software according to these criteria and presents a structured list of matching tickets directly in Teams. If no tickets meet the requested conditions, the system informs the employee and, when appropriate, suggests refining the search parameters to improve relevance. This capability facilitates efficient project management and reduces the need to switch between tools.

Collaboration is further supported by the assistant's ability to add comments to existing Jira tickets. Upon receiving a request to comment, the assistant first verifies that the ticket exists and that the employee has the necessary permissions. Once validated, the comment is appended in Jira Software, and confirmation is provided within Teams. In cases where permissions are insufficient, the assistant informs the employee, preserving transparency and adherence to access controls.

Closure of Jira tickets is similarly managed. The assistant allows employees to mark completed tickets as closed, updating the status in Jira Software and confirming the action within Teams. If the ticket is linked to unresolved subtasks, closure is blocked and the employee is notified, ensuring that task dependencies are respected and project tracking remains accurate.

Extending beyond project management, the assistant facilitates IT support operations via Jira Service Management. Employees can report technical issues, such as VPN problems or blocked access, and the assistant identifies the appropriate service desk to create a support ticket. The system provides the ticket ID and relevant details, ensuring proper tracking of the request. When the correct service desk cannot be determined automatically, the assistant prompts the employee for clarification, maintaining accuracy and streamlining resolution.

Finally, the assistant supports the closure of resolved IT tickets. Once an IT agent marks a ticket as resolved, the assistant notifies the employee in Teams, requesting confirmation of the resolution. Only upon receiving confirmation does the assistant close the ticket in Jira Service Management. If the employee does not confirm, the ticket remains in the "resolved" status, thereby preserving accountability, ensuring transparency, and maintaining control over IT support workflows.

Collectively, these capabilities illustrate how the assistant integrates task tracking, project management, and IT support into a unified conversational interface, enhancing efficiency, accuracy, and collaboration within the organization.

Chapter 4

Architecture Design

In recent years, the concept of *AI agents* has gained significant relevance in enterprise environments. An agent can be described as a software entity capable of autonomously perceiving inputs, reasoning about them, and producing an output tailored to the context. Unlike traditional applications, an agent is designed to interact continuously with users and systems, often relying on Large Language Models (LLMs) to interpret natural language queries and provide structured responses. The fundamental flow can therefore be summarized as: **user input** → **agent reasoning** → **agent output**.

In our project, the input and output channel is represented by **Microsoft Teams**. This choice was motivated by practical reasons: Teams is already widely adopted within the organization, and employees are familiar with its interface and usage patterns. Embedding the agent into Teams allows us to reduce the learning curve for end-users and seamlessly integrate the new functionality into their daily workflow.

The core of the architecture lies in the choice of the AI engine that powers the agent. Several alternatives were evaluated, with particular focus on three main candidates: **Azure OpenAI**, **Azure Agents in AI Foundry** and **Microsoft Copilot Studio**. Each solution provides specific advantages and limitations, which we discuss in the following sections.

4.1 Components

At its core, an AI agent can be decomposed into three fundamental components: the **input channel**, the **reasoning and knowledge layer**, and the **output channel**. In our architecture, these elements are combined to ensure seamless interaction between end-users and the underlying intelligence.

The **input and output** are both represented by **Microsoft Teams**. This

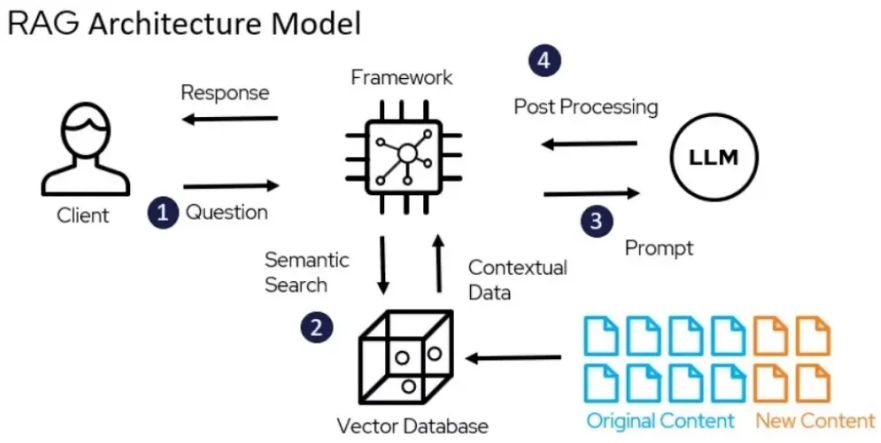

Figure 4.1: RAG (Source)

decision stems from practical considerations: Teams is already the central communication hub in the organization, and embedding the agent directly within it minimizes the learning curve for users. Requests are initiated as natural language messages within Teams, while the responses generated by the agent are returned in the same environment, ensuring a smooth and familiar workflow for employees.

The **reasoning layer** is the heart of the agent and is designed around the **Retrieval-Augmented Generation (RAG)** paradigm, as illustrated in Figure 4.6. RAG enriches the reasoning process of the Large Language Model (LLM) by integrating external domain-specific knowledge. Instead of relying solely on the general knowledge embedded in the model, the agent retrieves relevant documents or data from connected sources and augments its responses accordingly. This approach ensures that answers are not only linguistically coherent but also accurate and contextualized for the specific needs of the company.

Within this reasoning layer, the agent can also leverage additional **tools** that extend its capabilities beyond pure text generation. Examples include APIs for querying structured data, connectors to enterprise knowledge bases, or utilities for automating workflows. The orchestration of these components allows the agent to move closer to real problem-solving rather than simply producing text.

In summary, the architecture of our agent revolves around a clear flow: **user input in Teams → reasoning through RAG and tools → agent output in Teams**. This modular design not only simplifies the integration of different AI components but also guarantees that the solution remains adaptable to future extensions, such as new tools or alternative communication channels.

4.2 Azure OpenAI

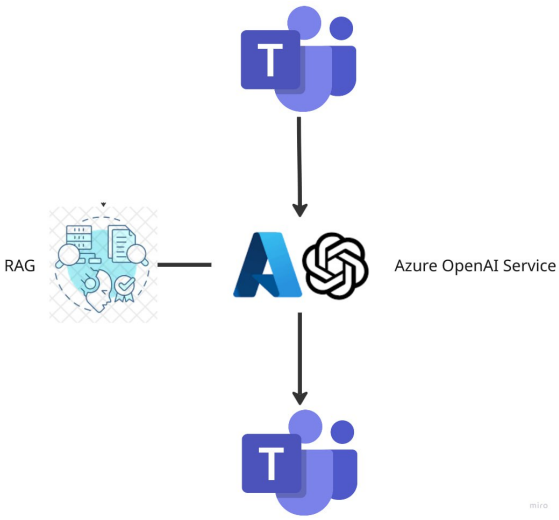

Figure 4.2: Azure OpenAI

The first option we analyzed is **Azure OpenAI**, which provides access to OpenAI's language models through the Azure ecosystem. This service enables seamless integration of GPT models with enterprise security, compliance, and scalability guarantees. In our case, the API plays a central role in handling multimodal inputs, such as text and images, as shown in Figure 4.3. Moreover, the integration with Copilot (Figure 6.11) demonstrates how Azure OpenAI can be extended to support productivity scenarios within Microsoft applications.

A further strength of Azure OpenAI lies in its native compatibility with Azure identity and access management, which allowed us to integrate authentication and authorization flows without the need for custom logic. This was particularly useful in ensuring that only authorized employees could interact with the agent, while keeping the deployment compliant with enterprise policies. Additionally, the service offers built-in monitoring and usage analytics, which proved valuable for tracking performance and anticipating scaling needs.

Another key aspect is the availability of fine-tuning and embedding capabilities directly within the platform. Although we primarily relied on base GPT models, the possibility of enriching them with domain-specific corpora or generating vector embeddings for semantic search makes Azure OpenAI especially attractive for enterprise applications where context-awareness is critical. In our project, this capability was considered as a potential extension to improve the retrieval of company documents and to align the agent more closely with internal knowledge bases.

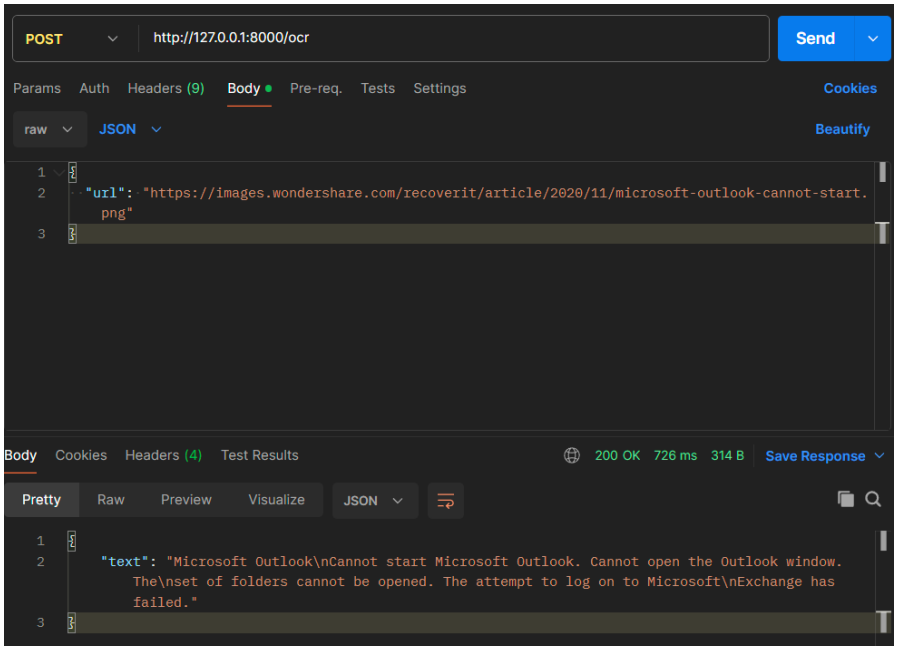

Figure 4.3: API dedicated to read images

From an operational perspective, Azure OpenAI also benefits from enterprise-grade deployment options, such as regional availability and data residency guarantees. This means that sensitive business data can be processed within predefined geographic boundaries, an aspect that is often crucial for regulatory compliance in corporate environments. Furthermore, the service supports network isolation through private endpoints, which allows traffic to remain inside the organization's virtual network, thus reducing exposure to external threats.

The main advantages of this option are the maturity of the models, their advanced natural language understanding, and the ease of integration into existing Azure services. However, some limitations were observed, particularly regarding customization: while powerful, the models tend to behave as black boxes, leaving limited flexibility in deeply adapting the reasoning pipeline to domain-specific workflows. This characteristic makes Azure OpenAI an excellent choice for general-purpose tasks and rapid integration, but potentially less suited for scenarios that demand highly specialized reasoning chains or granular control over intermediate steps.

4.3 Azure Agents in AI Foundry

The second candidate we considered is **Azure Agents in AI Foundry**. This framework is designed to create modular AI agents that can combine reasoning with task orchestration. Figure 7.9 illustrates the AI Foundry interface, which supports the configuration of different agents and their connections with enterprise data sources and APIs.

One of the main strengths of Azure Agents lies in their modularity. They allow the creation of workflows where the LLM reasoning can be combined with deterministic steps, external API calls, or the invocation of other agents. This makes the system highly adaptable to different business requirements and particularly suited for scenarios where the agent must integrate with heterogeneous services within the company.

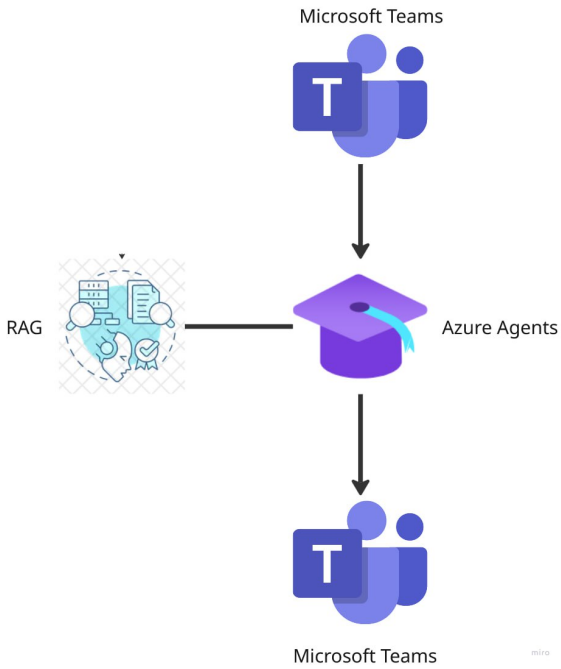

Figure 4.4: Azure Agents

For our project, this flexibility translated into the possibility of connecting the agent not only to knowledge bases but also to external systems, such as ticketing, monitoring, or workflow management tools, through pre-defined connectors. This enabled the design of end-to-end automation pipelines where the agent could interpret a user request, enrich it with contextual data, trigger backend processes, and return structured outputs, all in a single seamless interaction.

Another advantage is the support for conditional logic and branching workflows

directly within the agent orchestration. This allows the implementation of complex reasoning paths, prioritization of tasks, and handling of exceptional cases without hardcoding them into the LLM itself. Such capabilities are particularly valuable in enterprise contexts, where processes are often highly structured and need to comply with internal policies and governance standards.

However, this flexibility comes with operational considerations. The configuration complexity is higher compared to simpler LLM APIs, requiring careful design to avoid redundant or conflicting agent behaviors. Moreover, monitoring and maintaining multiple interconnected agents demands robust logging, telemetry, and alerting mechanisms to ensure that the system remains reliable and performant over time. Finally, deployment in production environments requires attention to security, network isolation, and identity management, which are all supported by Azure but must be explicitly configured to meet enterprise standards.

In summary, Azure Agents in AI Foundry provide a highly modular and extensible platform for building sophisticated agents, combining reasoning, orchestration, and integration with multiple services, but at the cost of increased configuration and operational complexity.

4.4 Azure Bot Framework

The **Azure Bot Framework** is an optional component that can be employed as a communication layer between user-facing platforms and AI services. Its primary function is to capture user inputs, manage basic dialog flow, and forward requests to the appropriate backend, effectively acting as a connector.

In our case, the framework was only considered at a high level, since Microsoft Teams already offered a direct interaction channel and Copilot Studio later became our preferred environment. However, the Bot Framework remains useful in scenarios where Teams is not the target interface or when a custom client needs to communicate with services such as **Azure OpenAI** or **Azure Agents in AI Foundry**.

Among its advantages are flexibility in integration, the possibility to standardize the communication layer, and built-in support for authentication and message routing. These aspects make it suitable for enterprise environments where an agent might need to serve multiple front ends. For our project, it served mainly as a reference option rather than a core component, but it highlights the broader ecosystem available within Azure for connecting conversational agents to different clients.

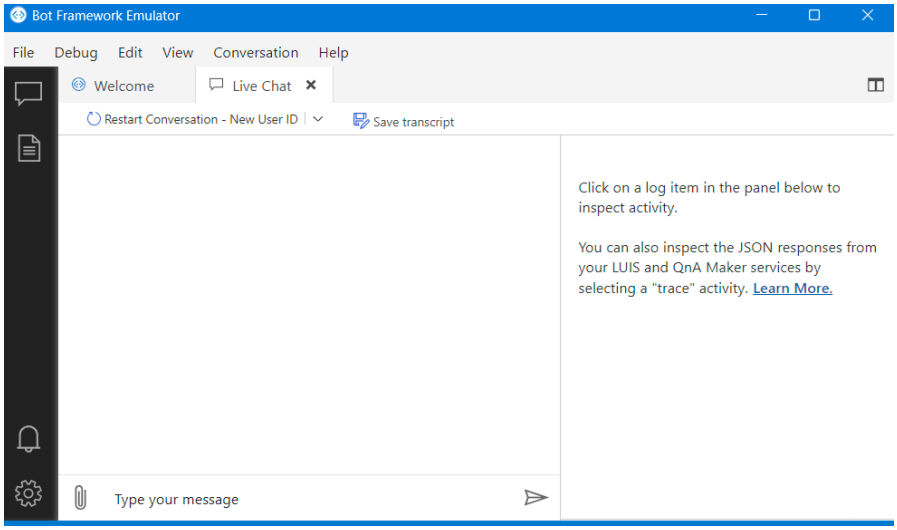

Figure 4.5: Azure Bot Framework

4.5 Final Choice

After conducting a detailed comparison of the three approaches, it became evident that while each of them provides strong features and enterprise-grade capabilities, the option that best aligned with our context was **Copilot Studio**. The evaluation process was not limited to a technical benchmark of functionalities, but rather considered a broader set of factors including ease of adoption, long-term sustainability, integration with existing platforms, and total cost of ownership. From this perspective, Copilot Studio emerged as the solution that provided the best balance between technical robustness and organizational feasibility.

A major reason behind this choice was the natural integration of Copilot Studio into the Microsoft 365 environment, and in particular Microsoft Teams, which was already central to the daily operations of the company. Unlike other solutions that required additional configuration layers or complex orchestration workflows, Copilot Studio offered a more direct path to adoption, reducing the time needed to deliver a working prototype and lowering the learning curve for employees. This characteristic translated into a smoother user experience, where employees could start interacting with the assistant in a familiar environment without needing extensive training or onboarding.

Beyond usability, commercial and strategic considerations also played a crucial role. By leveraging the existing Microsoft licensing model, the deployment of Copilot Studio introduced minimal overhead compared to adopting additional third-party solutions. This aspect reduced both the financial cost and the administrative burden of managing extra platforms, making it easier for the IT department to

support and govern the system. At the same time, aligning the solution with the company's broader Microsoft ecosystem strategy ensures long-term maintainability and compatibility with future updates or extensions introduced by the vendor.

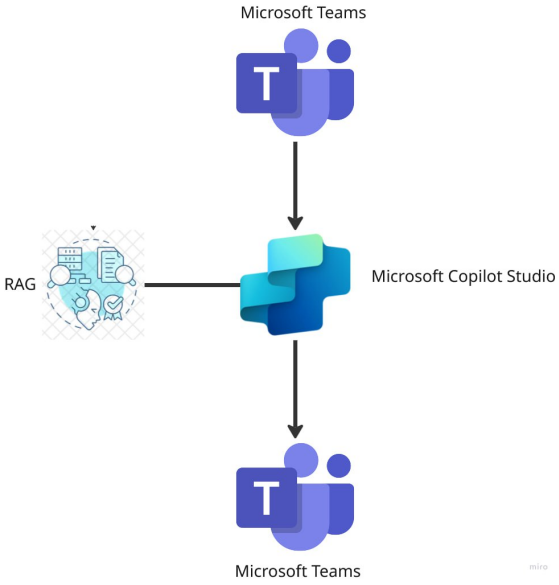

Figure 4.6: Copilot Studio

Another decisive element was the extensibility potential offered by Copilot Studio. Although not explored in detail at this stage, its architecture allows the gradual addition of connectors, workflows, and integrations with other enterprise services. This flexibility means that the assistant can start from a well-defined scope and progressively evolve in response to new requirements, without forcing disruptive redesigns of the overall system. In this sense, Copilot Studio not only addresses the immediate needs identified during the requirements gathering phase but also provides a forward-looking framework that can accommodate future developments.

In summary, Copilot Studio was selected as the most suitable option because it combines practical deployment advantages, cost-effectiveness, and strategic alignment with the organization's technology stack. While alternatives such as Azure OpenAI and Azure Agents in AI Foundry offered significant capabilities, they required either greater customization efforts or introduced complexity that was not justified by the expected use cases. By contrast, Copilot Studio provided a balanced solution capable of delivering value quickly while leaving room for gradual expansion.

The following chapter will examine this choice in greater detail, presenting the architectural principles, key features, and advantages of Copilot Studio, and showing how it became the cornerstone of our LLM-based agent implementation.

Chapter 5

Foundations of Artificial Intelligence

5.1 Historical Foundations

Artificial intelligence, as we know it today, is the result of decades of exploration, experimentation, and gradual evolution rather than a sudden breakthrough. Early efforts in AI focused on symbolic reasoning and explicit rule-based systems. These systems excelled in highly structured domains, such as theorem proving and expert systems, where rules could be fully specified. However, their rigidity made them ill-suited for tasks that required adaptability, generalisation, or understanding of naturalistic data.

As computational resources and data availability increased, statistical methods and early machine learning approaches gained prominence. Models could now learn patterns from examples rather than relying solely on handcrafted rules. This shift from knowledge engineering to data-driven techniques marked a crucial turning point in AI research, enabling systems to tackle more complex and varied tasks. In the domain of natural language processing, this evolution is reflected in the progression from bag-of-words models, which captured simple word frequencies, to word embeddings that represented semantic relationships in continuous space, followed by transformer architectures that modelled contextual dependencies across sequences, and ultimately to large language models (LLMs) capable of capturing intricate patterns, context, and reasoning in text.

The emergence of deep learning further accelerated these trends: multi-layer neural networks demonstrated unprecedented abilities in computer vision, speech recognition, and natural language processing, establishing new benchmarks for performance. This historical trajectory, from symbolic logic to statistical learning and deep representation learning, laid the foundation for contemporary AI systems.

In particular, it set the stage for transformer-based LLMs, which leverage massive amounts of data to learn nuanced patterns in text. These models now form the backbone of many modern AI applications, enabling sophisticated reasoning, fluent language generation, and integration into agentic architectures capable of operating in real-world enterprise environments.

5.2 From LLMs to Agentic Systems

Artificial intelligence in its modern form is dominated by data-driven, representation-learning approaches, among which large language models (LLMs) occupy a central role. These models, trained on massive corpora with self-supervised objectives, possess an exceptional ability to model natural language and produce fluent, contextually appropriate continuations. When applied to tasks such as summarisation, code generation or question answering, LLMs have demonstrated remarkable generalisation capabilities that make them powerful building blocks for conversational systems. However, the raw generative paradigm has intrinsic limitations when the objective is robust, auditable and safe operation within operational settings. Models that are relied upon solely as generators tend to produce outputs that prioritise linguistic plausibility over verifiable correctness. This gap manifests as hallucination, a failure mode in which models assert facts or relationships that are not supported by evidence. Hallucinations arise from a combination of factors, including biases in training data, the absence of explicit grounding mechanisms and the probabilistic nature of continuation models.

Prompt engineering Prompt engineering is a foundational technique to guide LLMs toward producing more reliable, context-aware outputs. At its core, it involves structuring inputs with clear instructions and providing illustrative examples through few-shot prompting, which helps the model recognise patterns and adapt its responses to a desired task. Zero-shot prompting, by contrast, challenges the model to generalise purely from the instructions without examples, testing its inherent ability to apply learned patterns to novel tasks. Role prompts can assign the model a specific persona or domain expertise, ensuring responses are framed appropriately for the intended context. Sampling parameters, such as temperature, top-k, and top-p, offer additional control over output diversity, while system-level instructions maintain consistency in style, tone, and behaviour. Together, these methods help constrain the model's generative space, improving both factual correctness and linguistic coherence, though they remain inherently limited by the model's architecture and the data on which it was trained.

Chain-of-thought and reasoning techniques To enhance reasoning capabilities, advanced prompting methods encourage LLMs to explicitly articulate intermediate steps before producing a final answer. Chain-of-thought (CoT) prompting instructs the model to decompose complex problems into sequential sub-steps, fostering logical reasoning and reducing errors that arise from skipped or implicit inference. CoT can be combined with few-shot examples to illustrate structured reasoning patterns, allowing the model to internalise the stepwise approach while solving similar problems. Role-based prompts can additionally frame the model as an expert or decision-maker in a given domain, further improving interpretability and confidence in outputs where correctness and traceability are critical. In enterprise contexts, reasoning-enhanced prompts are particularly valuable for tasks requiring multi-step problem solving, careful planning, or compliance with strict operational standards.

Multimodal models and expanded reasoning Recent advances in AI have introduced multimodal models that process and integrate multiple types of input—such as text, images, structured data, and audio—allowing LLMs to extend their reasoning capabilities far beyond traditional text-only models. By combining information from diverse sources, these models can perform more complex and contextually informed inferences, plan multi-step operations across heterogeneous data, and ground their outputs in real-world domains with greater fidelity. The integration of multimodal inputs not only enhances understanding and decision-making, but also amplifies the importance of structured retrieval, tool interfaces, and state management, laying the groundwork for agentic architectures where complex input types must be handled reliably and outputs need to remain verifiable and auditable.

Agentic architectures and MCP To address these shortcomings the community has increasingly adopted agentic architectures. In such designs the LLM functions as the reasoning and language interface within a modular system where retrieval modules, planners, tool interfaces and state managers implement specialised responsibilities. Retrieval components provide up-to-date, domain-specific evidence that can be surfaced to the model and to end users. Planning components decompose complex requests into explicit sub-tasks and determine execution order and fallback strategies. Tool interfaces encapsulate interactions with external systems and enforce input validation, authentication, and logging. State managers and memory stores preserve long-term context across sessions. This separation of concerns enables practical benefits: grounding via retrieval reduces the frequency of hallucinated claims by anchoring responses to verifiable documents; planners increase robustness for multi-step operations by making dependencies explicit; and tool interfaces allow side-effecting actions to be executed with transactional

semantics and audit trails.

Protocols such as the Model Context Protocol (MCP) instantiate these ideas by providing typed contracts that describe available tools, expected inputs and outputs, streaming behaviours and error semantics. MCP-style contracts decouple model reasoning from execution semantics and allow operators to impose governance policies while preserving the model's flexible reasoning capabilities. The result is a hybrid approach where the LLM remains the creative and reasoning engine, but actions that could affect production systems are mediated by deterministic, auditable software components. This architectural evolution is foundational to the system described in this thesis and motivates many of the engineering choices documented in later chapters.

ChatGPT-4 and enterprise agents An illustrative example of modern LLM deployment is ChatGPT-4, which underpins Microsoft's Copilot Studio. Here, the model serves as a natural language interface that helps users orchestrate workflows and interact with enterprise applications. Its role extends beyond text generation to mediating tool interactions, demonstrating the shift from stand-alone LLMs to embedded reasoning engines within larger ecosystems. The integration of ChatGPT-4 into Copilot Studio highlights the balance between generative flexibility and the need for oversight, grounding, and security mechanisms in enterprise settings.

5.3 Practical Considerations

Beyond model accuracy Bringing AI systems into production environments requires attention to issues that extend beyond model accuracy. While achieving high predictive performance is important, enterprise deployment demands consideration of reliability, interpretability, and operational robustness. Prompt engineering, together with advanced techniques such as chain-of-thought reasoning, few-shot and zero-shot examples, ensemble prompts, self-consistency, and temperature/top-k tuning, provides critical mechanisms to influence model behaviour in predictable ways. For instance, carefully designed prompts can guide an LLM to reason step by step, reduce hallucinations, or adopt a specific professional persona suited to the enterprise workflow. In practice, these techniques must be embedded within broader frameworks that include retrieval, validation, and governance, ensuring that outputs are not only contextually accurate but also auditable, reproducible, and aligned with organisational policies.

Retrieval-Augmented Generation and system design Retrieval-Augmented Generation (RAG) bridges LLMs with domain-specific knowledge by dynamically retrieving relevant information from curated indices to condition the model's

responses. This approach enhances transparency by making claims traceable to source documents, and decouples knowledge updates from expensive model retraining: updating the retrieval index is often sufficient to reflect changes in the underlying knowledge base. However, RAG effectiveness strongly depends on the quality of the retrieval pipeline, including index freshness, document chunking strategy, embedding model selection, and re-ranking methods. Rigorous engineering practices are essential to ensure reliable outputs: relevance scoring, entailment verification, and provenance-aware presentation of retrieved content help mitigate the risk of misinformation or contextually inappropriate answers. Thoughtful integration of RAG into agentic systems also allows for access controls, content redaction, and compliance with organisational policies, making it an indispensable component of enterprise-ready AI architectures.

Security, fairness, and observability Safety, privacy, and fairness are central concerns when deploying AI systems in production. Broadly trained models may inadvertently reproduce harmful stereotypes, amplify bias, or expose sensitive information if retrieval indices are poorly curated or access permissions are misconfigured. Operational safeguards must therefore include sensitive content filters, role-based access controls, and human-in-the-loop approvals for high-risk actions. Observability and auditing are critical for maintaining accountability: detailed logging of tool invocations, contextual inputs, user identity, model outputs, and system responses enables forensic investigation, supports supervised fine-tuning, and facilitates continuous monitoring for anomalous or unsafe behaviour. Moreover, these practices allow organisations to track performance metrics over time, identify potential vulnerabilities, and iteratively refine system policies to maintain both ethical and operational standards.

Evaluation strategies Evaluation of agentic AI systems requires a multipronged approach that combines automated testing, synthetic benchmarks, and human-centered assessment. Unit and integration tests validate tool manifests, schema conformance, and authorization flows, while telemetry and synthetic workloads provide quantitative measures of latency, error rates, throughput, and resource utilization under realistic operating conditions. Human evaluation is equally important: pilot deployments and controlled studies assess user experience, clarity, explainability, trustworthiness, and the appropriateness of fallback strategies and escalations. Collectively, these evaluation practices ensure that AI agents meet functional requirements while also supporting softer dimensions such as user confidence, transparency, and organisational fit. Continuous monitoring and iterative refinement of both model behaviour and system interfaces are necessary to adapt to evolving operational demands and maintain high standards of performance and reliability.

Regulatory landscape The European Union's AI Act represents a milestone in codifying responsibilities and safeguards for high-risk AI systems, influencing the design, deployment, and governance of enterprise AI agents. By mandating transparency, audit trails, human oversight, and documentation of risk management processes, the Act imposes clear obligations on both developers and operators. Enterprise AI agents, particularly those interacting with sensitive data or performing actions with potential legal or safety implications, must incorporate comprehensive logging, strict access controls, verifiable execution paths, and mechanisms for human intervention. Compliance with such regulations reinforces the need to integrate governance and accountability directly into system architectures, ensuring that AI foundations encompass not only technical performance but also ethical, legal, and organisational considerations from the outset.

Chapter 6

The Conversational Layer: Implementation with Microsoft Copilot Studio

Copilot Studio represents a natural evolution in the use of artificial intelligence within the modern workplace. It does not present itself as yet another static tool, but rather as a system that adapts to the user's needs. Through a conversational interface, it enables the creation, management, and customization of virtual assistants in an intuitive way, without requiring complex coding skills. Embedded within the Microsoft ecosystem, Copilot Studio integrates seamlessly with Teams, Outlook, and other enterprise applications, allowing users to automate tasks, retrieve information, or orchestrate workflows with ease.

Among its key capabilities, Copilot Studio offers integration with knowledge sources through retrieval-augmented generation (RAG), allowing assistants to access and reference content from SharePoint, which is capable of automatically indexing the file you give to it. It also provides built-in analytics to monitor usage and performance, offering insights into user interactions. Additionally, it supports custom actions, enabling bots to trigger workflows or external services as part of a conversation.

6.1 Retrieval-Augmented Generation

One of the most impactful capabilities introduced in Copilot Studio is the use of Retrieval-Augmented Generation (RAG), a technique that significantly enhances the assistant's ability to provide accurate and context-aware responses. At its core, RAG allows the virtual agent to access external knowledge sources in real

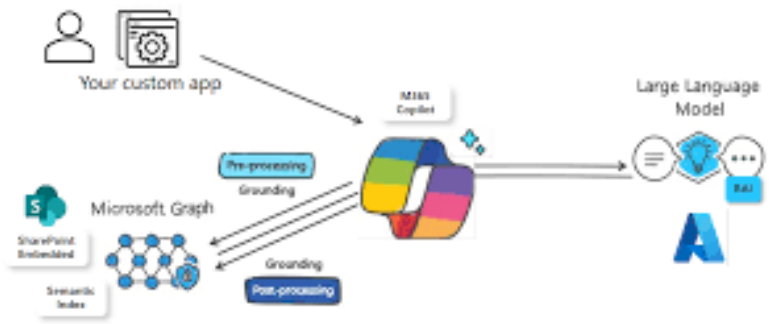

Figure 6.1: SharePoint Rag

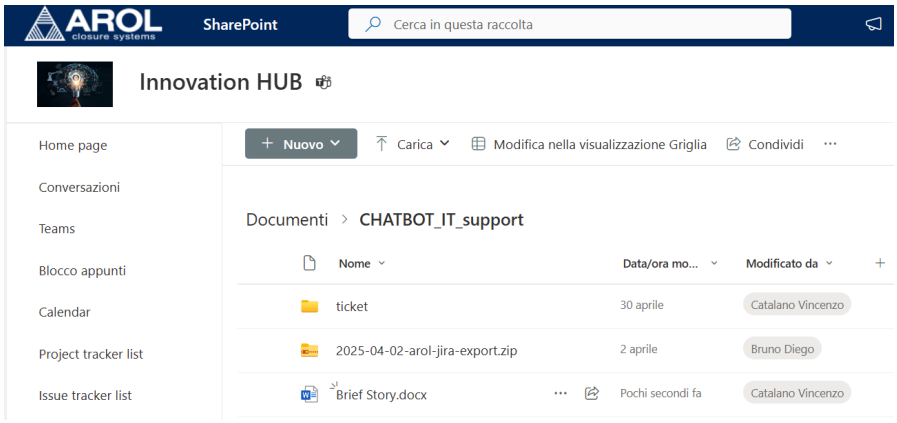

Figure 6.2: SharePoint folder

time, retrieve relevant content, and use it to generate more informed and grounded answers.

In the context of Copilot Studio, RAG is primarily used to extend the assistant's knowledge beyond pre-defined prompts or static data. Instead of relying solely on what is built into the assistant at design time, RAG enables dynamic access to enterprise documents, guidelines, FAQs, and other internal resources. This proves especially useful in large organizations, where institutional knowledge is often scattered across platforms and formats.

The integration with SharePoint plays a central role in this process. SharePoint document libraries and pages can be connected as data sources, allowing Copilot Studio to retrieve content directly from these repositories when answering user queries. This means that a user can, for example, ask a question about a company policy or procedure, and the assistant will respond based on the most recent version of the document stored on SharePoint without the need for manual updates or

hardcoded answers.

To mitigate hallucinations, the assistant relies on strict retrieval scoring and ranking to surface high-quality documents and always displays provenance (source snippets and links) alongside generated answers. A conservative answer policy is enforced: when confidence is low the assistant replies with an explicit uncertainty message (e.g., "I'm not sure") and offers to escalate or open a ticket for human follow-up. These measures reduce incorrect assertions and provide traceability for auditing and manual verification.

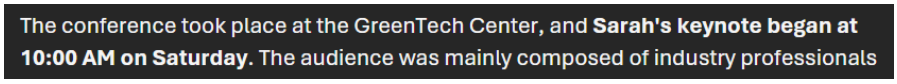

Figure 6.3: The piece of text inside the file in SharePoint

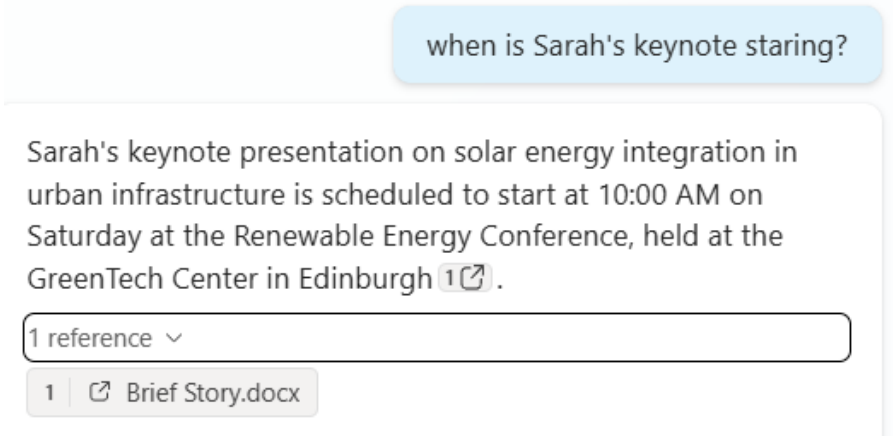

Figure 6.4: The actual Copilot response

6.1.1 Extraction from Jira

To populate the RAG index, we began with a raw export of the Jira instance for our project and implemented an exploratory extraction and conversion pipeline to produce PDFs suitable for SharePoint ingestion. This approach was motivated by the practical need to consolidate institutional knowledge, which primarily resided in Jira tickets, including descriptions, comments, and attachments, without any unified documentation. The pipeline provides a reproducible method to collect, normalize, and surface this content for retrieval within the AI assistant.

High-level workflow The pipeline was organized in several sequential stages. Initially, the Jira export, available in JSON or CSV format, was loaded into a data

analysis environment, such as a Jupyter notebook, to inspect projects, issue types, fields, and attachments. Given the absence of formal documentation regarding Jira organization, an iterative exploration of projects, components, labels, and custom fields was performed to identify relevant sections and subsections, for instance project keys, epic links, or subsets of components containing design notes.

For each relevant issue, canonical text fields, including summary, description, comments, and changelog entries, were extracted and normalized by stripping HTML or converting markdown-like markup to plain text or HTML. Attachments referenced by issues, such as images, documents, and archives, were downloaded using authenticated HTTP requests against Jira endpoints. Non-PDF attachments, including Markdown, HTML, and Word documents, were converted to PDF using utilities such as `pandoc`, `wkhtmltopdf`, or headless LibreOffice to ensure uniform storage in SharePoint; native PDFs were retained unchanged. Each PDF was accompanied by a metadata sidecar in JSON format containing provenance information, such as the original issue key, attachment ID, author, creation and modification dates, source URL, and a short extracted snippet. Finally, the processed PDFs and metadata were uploaded into a dedicated SharePoint document library, organized according to a predictable folder structure, and indexed either via the organization's ingestion connector or directly by a dedicated script to produce text chunks, compute embeddings, and populate the vectorstore used during query time.

Practical notes from the notebook The pipeline was implemented in a Jupyter notebook (`DumpJIRA.ipynb`), where several practical heuristics proved valuable. Exploratory filtering involved computing value counts for fields such as `project`, `component`, `labels`, and `issuetype` to locate dense regions of useful textual content. The attachment downloader incorporated retry mechanisms with exponential backoff to handle transient HTTP errors, logging any failures for later reattempt. Conversion rules were standardized: `pandoc` was preferred for Markdown to PDF, `wkhtmltopdf` for HTML pages to preserve layout, and headless LibreOffice for office formats. Filenames were normalized, and problematic characters removed. Metadata was carefully designed to include original Jira identifiers, ensuring traceability for audit purposes and enabling user-facing provenance display in Copilot responses. For RAG preparation, text was split into overlapping chunks (e.g., 800-token segments with 150-token overlap) to maintain context continuity across retrieval boundaries.

SharePoint ingestion and RAG indexing Once uploaded, the documents were surfaced via the organization's SharePoint connector for Copilot Studio. Additionally, a dedicated ingestion step was performed, in which documents were downloaded from the target SharePoint folder, textual content was extracted

(including OCR for image-only PDFs when necessary), and chunking was applied at the sentence or paragraph level with overlap. Embeddings were then computed using the selected model, and vectors and metadata were stored in the chosen vectorstore, such as FAISS, Milvus, or Azure Cognitive Search with vector indexing. Metadata fields, including issue key, attachment ID, and original URL, were preserved alongside each vector to guarantee provenance and provide direct links to the original Jira artifacts in Copilot responses.

Limitations and future work The heterogeneous content and inconsistent use of fields within the Jira dump necessitated some manual curation, for example to exclude ephemeral or test issues. Automated heuristics for filtering low-quality pages, such as very short issues or test tickets, improved retrieval precision. Future iterations should consider scheduling incremental crawls to ingest newly created Jira tickets and attachments, implementing a light quality assurance pass combining automated filters with brief manual review before promoting documents to the production RAG index, and maintaining a mapping registry linking SharePoint paths to Jira keys to facilitate traceability and potential rollback.

6.2 Analytics

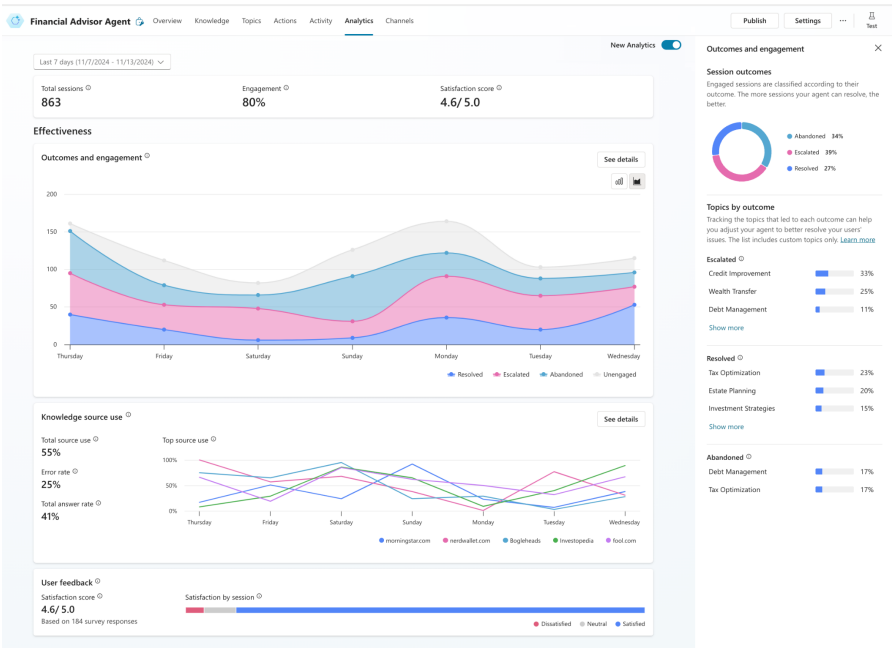

Figure 6.5: Analytics

Copilot Studio provides a comprehensive analytics dashboard that allows creators and administrators to monitor the performance, usage, and overall health of their virtual assistants. This functionality is particularly valuable in enterprise contexts, where understanding user interactions and identifying potential issues is critical to ensure adoption and effectiveness. Among the key metrics available are the total number of sessions initiated, the volume and type of user messages, the frequency with which specific topics or intents are triggered, and conversation abandonment rates. By analyzing these indicators, developers can gain insights into how users engage with the assistant, identify recurring patterns, and detect points where the interaction flow may break down or require refinement.

In addition to quantitative metrics, Copilot Studio offers qualitative tools for assessing response quality and user satisfaction. For instance, unhandled questions can be automatically grouped into thematic clusters, allowing developers to recognize gaps in the assistant's knowledge base or reasoning capabilities. Moreover, user feedback can be collected directly within conversations through embedded rating prompts or satisfaction surveys. These mechanisms can be orchestrated via Power Automate flows, enabling the automated collection, storage, and aggregation of feedback data, which is crucial for continuous improvement and iterative development cycles.

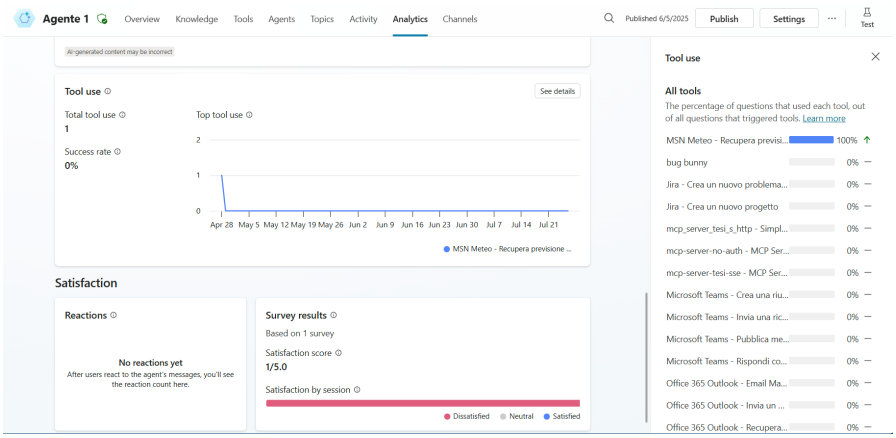

Figure 6.6: Analytics

From an enterprise perspective, such analytics not only support the enhancement of user experience but also facilitate compliance, monitoring, and reporting requirements. Advanced users can export the analytics data and integrate it with business intelligence platforms such as Power BI, creating customized dashboards, trend analyses, and alerts for anomalous behavior. This enables both technical teams and management to track adoption, evaluate the impact of the assistant, and make data-driven decisions regarding future enhancements or deployment strategies.

Furthermore, analytics insights play a pivotal role in maintaining conversational quality over time. By monitoring session duration, user engagement, and the distribution of topics, teams can optimize training datasets, adjust intent recognition models, and refine response generation strategies. Such continuous monitoring ensures that the virtual assistant remains aligned with organizational objectives, adapts to evolving user needs, and contributes positively to workflow efficiency.

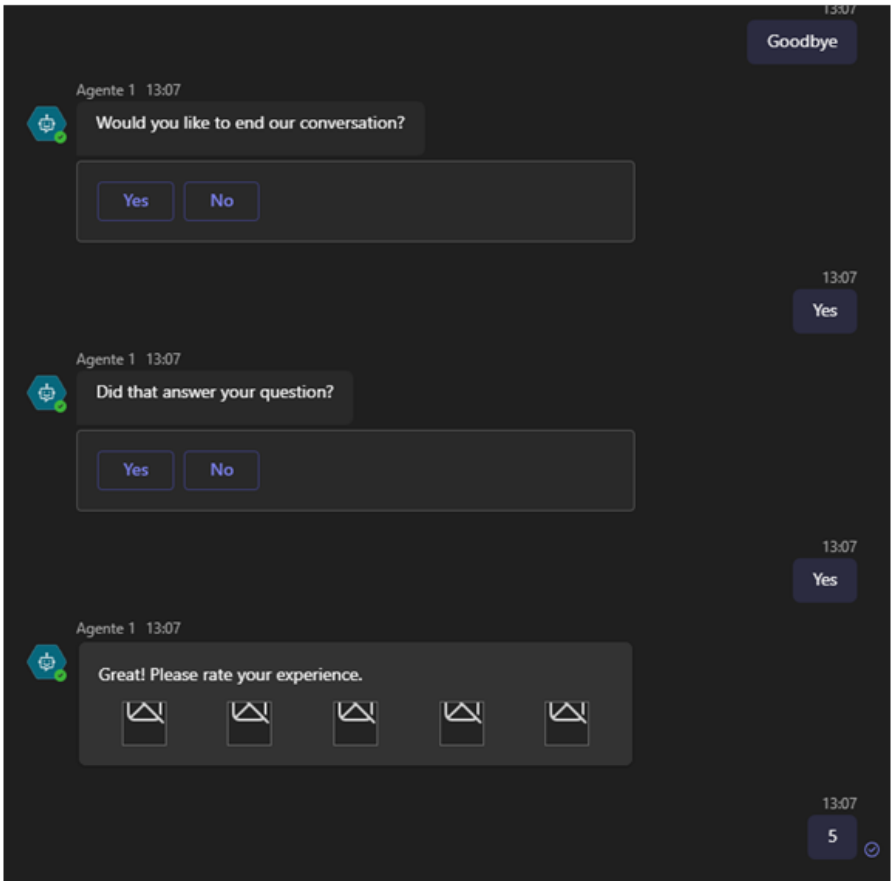

Figure 6.7: Rating promt

Overall, the analytics framework within Copilot Studio provides a rich and flexible set of tools that bridge the gap between user behavior, operational metrics, and actionable insights. It empowers organizations to systematically evaluate the performance of AI agents, drive iterative improvements, and maintain a high standard of user-aligned conversational experiences in production environments.

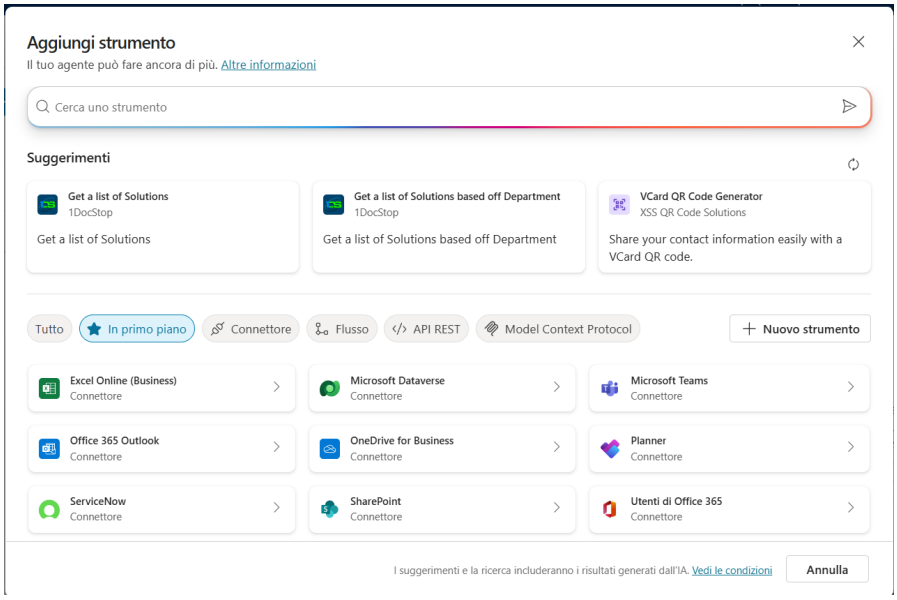

Figure 6.8: Tools

6.3 Agentic Tools

Within Copilot Studio, multiple integration options are available to extend the assistant's functionality, enabling the system to interact with external services, internal databases, and custom workflows. These tools are critical in connecting conversational intelligence with actionable enterprise processes, allowing virtual assistants to perform operations beyond mere question-and-answer interactions. The choice of tools directly impacts the agent's capabilities, maintainability, and scalability, and therefore constitutes a central design consideration.

Broadly, these integrations can be divided into two main categories: prebuilt low-code tools provided by Microsoft or third-party vendors, and fully custom tools developed through APIs. Prebuilt tools offer a convenient starting point, as they provide ready-made connectors and actions for common tasks such as sending emails, scheduling Teams meetings, querying external services, or interacting with internal enterprise systems. These solutions are typically accessible via Power Automate flows, which simplify configuration and reduce the need for extensive programming expertise. However, while initially straightforward, prebuilt tools can become limiting as business requirements evolve. Adjusting workflows, handling edge cases, or troubleshooting unexpected behavior often requires deeper intervention, and the visibility into internal logic is limited, which can complicate maintenance and debugging.

Custom tools, on the other hand, provide greater flexibility and control. By

exposing proprietary APIs and integrating them directly into Copilot Studio, developers can design precise business logic, access specialized data sources, and orchestrate complex workflows tailored to the organization's needs. This approach supports more sophisticated automations, enabling the assistant to act as an intermediary between various enterprise systems. The trade-off lies in the management overhead: each API must be represented as a separate tool, requiring ongoing monitoring, versioning, and documentation. As the number of APIs grows, the complexity of maintaining and debugging the assistant can increase substantially, necessitating careful governance and development practices.

Figure 6.9 illustrates an example of how Copilot handles standard prebuilt tools, showing the interface for configuring and managing actions. This highlights the intuitive visual environment provided by the platform, which supports rapid prototyping and iterative testing, while also demonstrating the limits of low-code solutions when enterprise-specific customization is needed.

Overall, the tools ecosystem in Copilot Studio offers a spectrum of options, ranging from ease-of-use with low-code connectors to full programmability with custom APIs. Design decisions must balance flexibility, maintainability, and the operational context of the assistant, particularly when aiming to integrate AI agents into real-world enterprise workflows.

6.3.1 Low-code tools (Microsoft and third-party)

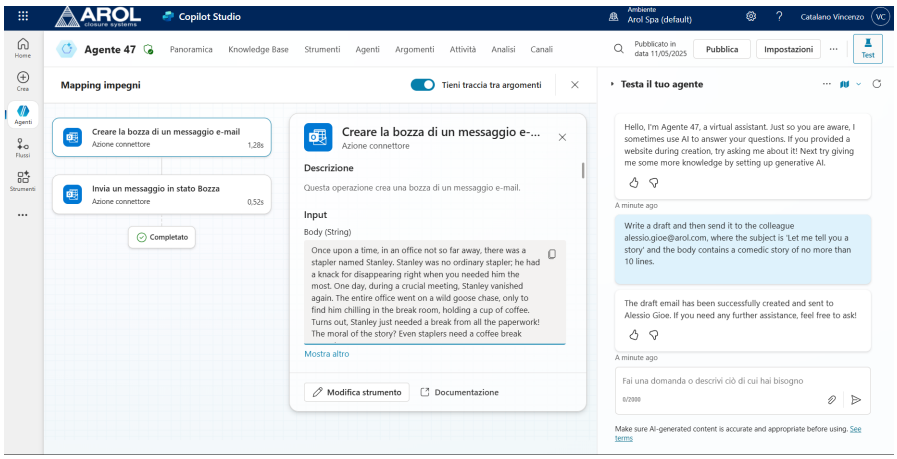

Figure 6.9: This shows how Copilot handles standard pre-built tools

The Microsoft ecosystem and its certified partners provide a rich set of low-code connectors and actions, often accessible directly via **Power Automate** flows. These tools enable rapid integration of standard enterprise operations such as sending emails, scheduling meetings in Teams, or querying commonly used

services like weather, stock, or database APIs. For teams with limited development resources, low-code tools offer an immediate way to make the assistant operational, significantly shortening time-to-value.

However, despite their convenience, low-code solutions have intrinsic limitations. Once initial configurations are in place, even minor changes in business logic or workflows can become cumbersome. The underlying implementation is largely abstracted, which reduces transparency and makes debugging unexpected behavior challenging. Moreover, low-code tools often struggle with complex orchestration scenarios, where multiple dependent actions need to be coordinated reliably across different systems. In practice, this can hinder the assistant's scalability and restrict its ability to adapt to evolving enterprise needs.

6.3.2 Custom tools via APIs

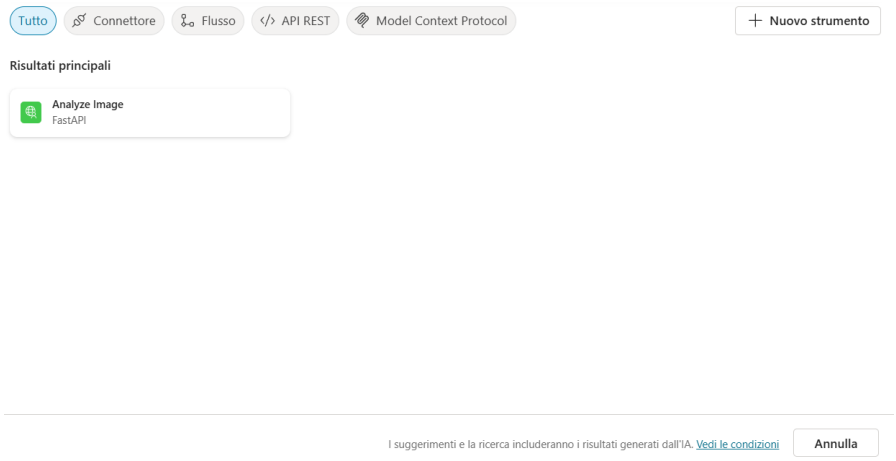

Figure 6.10: Custom API REST tool

Custom tools represent the most versatile option for extending Copilot Studio's functionality, as they are developed by exposing proprietary APIs and integrating them directly within the platform. Unlike low-code connectors, which rely on predefined actions and workflows, custom APIs provide full control over the assistant's operations. Every component of the interaction pipeline—from input validation and authentication to the execution of business logic—can be explicitly modeled, ensuring that the assistant adheres precisely to enterprise requirements. This approach is particularly beneficial when dealing with proprietary systems, legacy applications, or sensitive data sources that are not supported by standard connectors. Through API integration, the assistant can operate as an orchestrator, mediating between heterogeneous enterprise services and enabling advanced workflows that would otherwise be impossible to implement.

The advantages of this strategy are strongly tied to its flexibility. Custom APIs allow for the implementation of domain-specific logic, the extension of existing enterprise applications, and the creation of new services tailored to organizational needs.

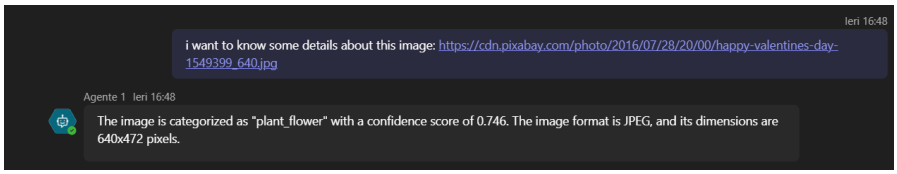

Figure 6.11: How the API works with Copilot

Furthermore, they enable fine-grained access to internal databases and processes, ensuring that the assistant can provide targeted, context-aware functionality. Figure 7.9 illustrates an example of a custom REST API configuration, while Figure 6.11 highlights how APIs are orchestrated within Copilot Studio, showcasing the system's ability to handle sophisticated enterprise operations. These visual examples demonstrate not only the technical feasibility of API-based tools but also their central role in bridging conversational intelligence with enterprise IT infrastructure.

However, the flexibility of custom APIs comes with significant management overhead. Each API endpoint must be carefully represented within the Power Platform, with detailed configurations for authentication, error handling, and data mapping. As the number of APIs integrated into the system increases, the complexity of managing them also grows. This includes tasks such as monitoring endpoints, ensuring compatibility across different versions, and documenting the interfaces to support maintainability. Additionally, coordination across development, DevOps, and support teams is essential, as errors or version mismatches can directly disrupt the assistant's behavior in production environments.

The trade-off, therefore, lies between flexibility and operational complexity. While custom APIs make it possible to design highly specialized tools capable of addressing unique enterprise needs, they also demand a robust governance framework to ensure stability and scalability. Without proper lifecycle management, including regular updates, logging, and systematic debugging procedures, the assistant may become difficult to maintain over time. In highly dynamic business contexts, this burden can reduce agility, slowing down iteration cycles and complicating the deployment of new features. For these reasons, organizations adopting custom APIs must complement their technical implementation with structured governance models and best practices for API management.

Table 6.1: Comparison between Low-Code Tools and Custom APIs in Copilot Studio

Aspect	Low-Code Tools	Custom APIs
Ease of Development	Intuitive drag-and-drop configuration, no coding required. Suitable for rapid prototyping.	Requires programming knowledge and integration effort. Slower to implement initially.
Flexibility	Limited to predefined connectors and templates provided by Copilot Studio.	Fully customizable, capable of handling complex and domain-specific workflows.
Maintenance	Low maintenance; updates are handled by the platform.	Requires continuous maintenance and version management.
Scalability	Well-suited for simple use cases with small to medium workloads.	Can scale to enterprise-grade solutions and integrate with heterogeneous systems.
Learning Curve	Very low, accessible to non-technical staff.	High, developers need expertise in APIs, authentication, and security.
Limitations	Restricted extensibility; cannot cover all business requirements.	Higher cost of development and risk of integration complexity.

6.3.3 Towards a new approach

To address the limitations of both low-code and custom API approaches, we explored a more scalable and developer-friendly paradigm: the **Model Context Protocol (MCP)**. Unlike traditional integration methods, MCP introduces a structured protocol for defining, exposing, and consuming external capabilities within conversational systems. It allows AI assistants to access a wide range of functionalities, whether internal services, data stores, or third-party APIs, without being tightly coupled to the low-code framework or requiring individual tool definitions for every endpoint.

The MCP approach provides several practical benefits. It decouples the assistant's reasoning logic from external services, simplifying maintenance and enhancing modularity. It also enables more dynamic workflows, where tools can be invoked based on context, user intent, or agent reasoning, rather than being hardcoded into fixed sequences. From a project perspective, adopting MCP allowed us to build a more flexible, testable, and scalable system, while retaining the possibility of

connecting both prebuilt and custom functionality. Overall, this paradigm aligns well with enterprise needs for maintainable, extensible, and context-aware AI agents, particularly in environments where multiple teams or services must coexist and evolve independently.

Chapter 7

Implementing the Orchestration Layer: A Custom Model Context Protocol Server

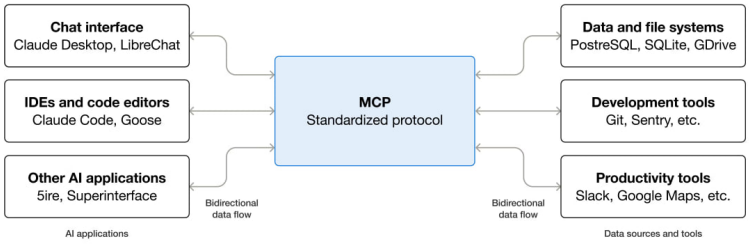

Figure 7.1: MCP

The Model Context Protocol (MCP) is an open specification designed to connect AI applications with the systems that hold relevant contextual data and tooling. Rather than relying on bespoke or ad-hoc integrations for every new assistant or environment, MCP defines a small set of interoperable primitives (servers, tools, resources, prompts and discovery mechanisms) that allow models and clients to find and consume functionality and data in a predictable, secure way. By providing a standardized interface for tool discovery, invocation and context sampling, MCP reduces integration overhead and enables the same external capabilities to be reused

across different assistants and platforms.

7.1 Origins

MCP emerged from community efforts to make AI assistants practical for real-world workflows by giving them safe, auditable access to the user's environment (files, APIs, task trackers, etc.). The driving motivation was to avoid brittle, single-vendor integrations and instead define a transport-agnostic protocol that any client or server can implement. Early adopters and implementors include a mix of open-source projects, desktop clients, IDEs and commercial platforms; this ecosystem-driven growth has allowed MCP to evolve rapidly while keeping implementation complexity low.

Historical context and goals. Beyond the immediate pragmatic need for interoperability, MCP was shaped by three complementary goals: (1) *portability* — make tools usable by different assistant implementations without per-client rewrites; (2) *auditable automation* — provide clear provenance for model-driven actions; and (3) *incremental adoption* — allow systems to implement a minimal subset of the protocol and progressively expose more capabilities. These goals explain design choices such as typed tool schemas, lightweight discovery mechanisms and optional streaming transports.

Adoption dynamics. Because MCP is specification-first and transport-agnostic, adoption tends to follow a multi-stage path: local developer experimentation (stdio/CLI servers), integrations into developer tools (IDEs, local assistants), and finally cloud-hosted registries and managed MCP servers for enterprise scenarios. This staged adoption lowers the bar for experimentation while enabling more mature deployments to adopt additional governance and access-control features.

7.2 Core concepts and architecture

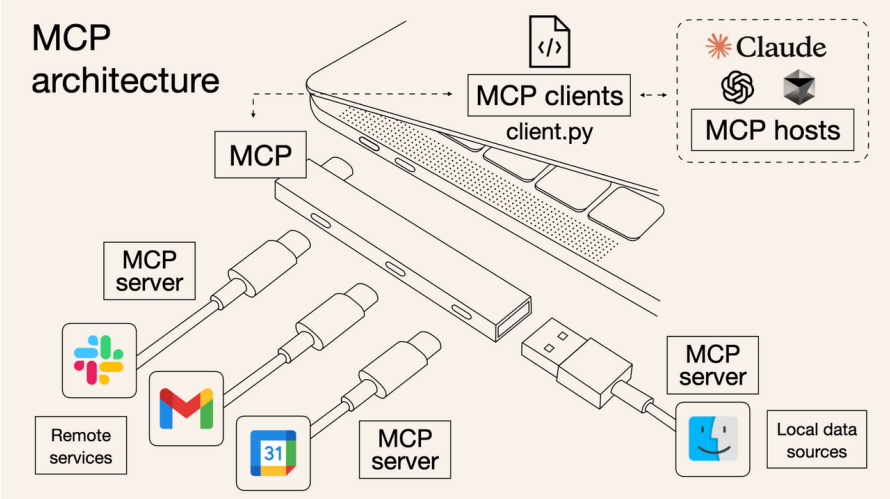

Figure 7.2: Architecture

Core concepts. The Model Context Protocol (MCP) is built around a set of well-defined core concepts that work together to separate concerns in a modular and extensible way. At its foundation, MCP introduces the notion of *servers*, which are long-running services responsible for exposing tools, resources, and prompt templates. These servers may operate locally, for instance as processes communicating through standard input and output, or they may be deployed remotely and accessed over HTTP or Server-Sent Events (SSE). In both cases, their role is to act as providers of functionality and context that can later be discovered and consumed by clients.

On the other side of the architecture stand the *clients*. These are applications that connect to servers in order to discover the capabilities they expose. Clients can then present these capabilities either to users or to language models, enabling them to invoke the tools offered by the servers. The relationship between servers and clients is intentionally flexible: servers evolve independently as long as they continue to honour the primitives defined by the protocol, while clients act as orchestrators that decide when and how to make use of the exposed capabilities.

Among the key elements provided by servers are the *tools*, which represent discrete operations that can be performed. A tool might allow searching within a document store, running a database query, or creating an issue in a tracking system. Each tool is accompanied by a schema and metadata describing its expected inputs and outputs, so that clients and models can reason about when and how to invoke it. Alongside tools, servers can also expose *resources* or *prompts*. These may

consist of static documents or parameterized prompt templates that can be used as contextual grounding, or as part of elicitation strategies when interacting with a model.

A final architectural component concerns the mechanisms of *discovery and sampling*. Discovery allows a client to enumerate all available tools and resources, while sampling introduces the possibility of retrieving a focused subset of context that is most relevant to the current query. Sampling often provides only partial or non-sensitive previews, enabling a conservative approach where decisions about whether to proceed with side-effecting operations can be deferred until explicit confirmation is obtained. This separation between preview and invocation is central to MCP's design philosophy: it reduces the risk of unintended operations, supports human-in-the-loop workflows, and simplifies failure recovery because actions are explicit and auditable.

Decision boundaries. The protocol therefore encourages explicit decision points in the overall orchestration: first discover what exists; then sample lightweight contextual material (metadata, short excerpts, schema-driven examples) to form a preliminary judgement; and finally perform an invocation only when sufficient confidence or authorization is present. In practice this pattern supports a variety of enterprise workflows. For example, when a user asks the agent to locate a document, the client can request sampled excerpts of candidate files, present these excerpts either to the model or to the user for confirmation, and only after a confirmation step perform a deeper retrieval or a side-effecting action such as sharing a document link or creating a ticket. Such staged interaction reduces the surface area of sensitive data exposure and makes the execution of side effects predictable and controllable.

Schema-driven interactions are another founding principle with strong UX and safety implications. Tools declare input and output schemas using typed, JSON-like descriptions. Clients can leverage these schemas to render structured input forms for users, validate inputs generated by models, and provide UI affordances such as required-field indicators, type hints, and constrained value pickers. From a developer's perspective, schema-first design dramatically reduces integration friction: a well-formed schema makes a new tool immediately usable across different client UIs without bespoke interface code. From an operational standpoint, schemas enable automatic validation of model outputs before they are translated into actual API calls, thereby catching malformed requests early and preventing many classes of runtime errors.

Operational aspects. Security and governance concerns are woven throughout the MCP model. Authentication and authorization are enforced at the level of server endpoints and client connections, and the protocol design encourages least-privilege exposure of capabilities. Sampling mechanisms are intentionally conservative:

previews are often truncated, redacted, or metadata-only so that sensitive payloads are not inadvertently transmitted to models or clients without explicit consent. Additionally, manifest and tool metadata can include policy annotations that clients must interpret: for example, which roles are allowed to invoke a tool, whether invocation requires multi-factor confirmation, or if an operation should be logged at higher fidelity for audit purposes. These features make MCP suitable for enterprise deployments where compliance, data residency and role-based access control are first-class requirements.

Operational aspects such as transport selection, streaming behaviour, and long-running operations are also important design considerations. MCP supports multiple transport adapters: stdio is convenient for local development and debugging, while HTTP/SSE or WebSocket transports are preferable for remote, production-grade deployments where streaming partial results and progress updates are required. Long-running tools can emit incremental outputs or progress events; clients can display these streaming updates to users, provide the option to cancel operations, and manage reconnection semantics. This streaming-first mindset helps implement responsive UIs for potentially slow tasks such as large-scale document indexing, batch processing, or complex external API orchestration.

Versioning, compatibility and testing are critical to maintain a healthy ecosystem of servers and clients. Manifests and tool schemas should be treated as contracts: changes to schemas must be coordinated, documented, and properly versioned to avoid breaking consumers. The recommended practice is to include contract tests in continuous integration pipelines, spawn ephemeral servers during automated tests, and employ mock servers where necessary to validate client behaviour. Such testing practices ensure that evolution of server capabilities does not lead to silent regressions in production client UIs or automated flows.

Observability rounds out the architecture. Servers and clients should emit structured logs, metrics, and tracing information that capture discovery events, sampling requests, invocation attempts, authorization checks, and the lifecycle of streaming responses. Provenance metadata—caller identifiers, timestamps, tool identifiers and request IDs—should be attached to outputs to enable post-hoc analysis, auditing and debugging. Centralized dashboards and alerting systems help teams detect anomalies such as sudden increases in failed invocations, repeated schema validation errors, or suspiciously frequent sampling of sensitive documents.

Finally, the layered MCP design naturally supports incremental adoption and future extensibility. Teams can start by exposing a handful of read-only tools and sampling-enabled resources to demonstrate value and evaluate governance implications. Over time they can add side-effecting tools, implement richer orchestration, and introduce more advanced prompts and template resources, all while preserving the client-side abstractions that make the system predictable to end users. A typical interaction under MCP therefore looks like a composed flow:

the user asks a question, the client discovers relevant tools, the client samples contextual material, a model (or the client) reasons about whether a tool invocation is warranted, and, subject to policy and confirmation, an invocation is executed and its results incorporated into a final, user-facing answer. This architecture balances flexibility with control, enabling powerful agent behaviours while respecting the operational constraints of enterprise environments.

7.3 Transports, formats and typical deployments

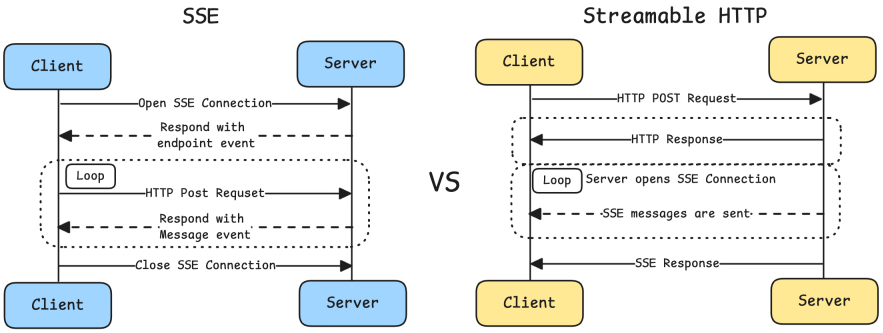

Figure 7.3: Context and Goals

The Model Context Protocol (MCP) has been conceived to operate across very different deployment environments, from lightweight local prototypes to large-scale enterprise services hosted in the cloud. This versatility is reflected in the variety of transports it supports, each with its own advantages and trade-offs. The most basic transport relies on *STDIO*, where a local process communicates over its standard input and output streams. This approach is extremely convenient for development, debugging, and privacy-preserving use cases, since all the data remains confined to the local machine. It is also appealing for scenarios where users or organizations want to experiment with new tools or workflows without setting up complex infrastructure. At the same time, STDIO communication is inherently limited to single-machine contexts, meaning it cannot easily be scaled to multi-user or distributed environments. For those larger-scale scenarios, MCP servers are typically deployed as remote services accessible over *streamable HTTP endpoints* or through *Server-Sent Events* (SSE). These protocols are particularly suited to long-lived connections, where results can be streamed incrementally and clients are kept updated about the state of ongoing operations in near real time. This is especially valuable in workflows where tasks may take minutes or even hours to complete, such as indexing a large corpus of documents or performing multi-step reasoning across multiple external APIs. Finally, for situations in which

low latency interaction is not critical, or where the goal is simply to invoke a single operation and retrieve its result, more traditional *simple HTTP APIs* can be employed. This option sacrifices streaming capabilities but offers a lightweight and widely compatible solution that can be integrated into existing web services with minimal friction.

Performance and reliability considerations. The choice of transport mechanism has direct implications for performance, reliability, and operational resilience. Local STDIO transports are attractive because they eliminate network hops, thus reducing latency and minimizing the attack surface for sensitive data. They are particularly effective in privacy-conscious environments, where keeping data on-device is a hard requirement. However, their utility is confined to scenarios where a single machine can handle the workload, limiting scalability and collaboration across teams. In contrast, remote deployments using HTTP or SSE enable multi-user access, distributed operation, and cloud-scale architectures, but they also introduce complexity. Engineers must account for connection drops, network partitions, and transient failures. Long-running tasks may require timeout management, periodic heartbeat messages, or progress updates so that clients do not assume the connection has stalled. Robust implementations therefore make use of mechanisms such as exponential backoff for retries, operation identifiers for ensuring idempotency, and health-check endpoints to provide external monitoring systems with reliable indicators of system status. In practice, these considerations are not merely theoretical: in enterprise environments, failure to address them can quickly lead to cascading issues where stalled requests, duplicate invocations, or unmonitored crashes compromise the stability of the whole system. MCP's design therefore encourages a systematic approach to reliability that combines protocol-level guarantees with best practices in distributed system engineering.

Interoperability and content negotiation. Beyond performance, another cornerstone of MCP's design is interoperability. Because MCP is intended to be transport-agnostic, it cannot assume that all clients and servers will share the same technical environment or evolve at the same pace. To deal with this, the protocol relies on explicit content negotiation mechanisms that allow participants to agree on schema versions, supported features, and encoding formats before any substantive interaction takes place. Metadata and schemas are always described using structured JSON, but binary payloads can be optionally transmitted through mechanisms such as multipart messages or base64 encoding when large attachments are involved. This design choice ensures that even complex workflows, such as retrieving a large document or passing along a media file for analysis, can be handled in a standardized way. Equally important is the strategy for schema evolution. In real-world deployments, servers inevitably add new capabilities, refine

existing tools, or deprecate older ones. Without careful planning, such changes can break existing clients, leading to downtime or unpredictable behaviour. MCP addresses this risk by encouraging explicit version fields, clear deprecation warnings, and soft-compatibility modes that allow clients to gracefully handle unfamiliar schema elements. This makes it possible for organizations to upgrade incrementally, introducing new tools or formats without forcing an immediate migration across all consumers. The result is a protocol that is not only technically robust but also operationally sustainable, providing a path for long-term evolution while protecting stability in production environments where reliability, compliance, and backward compatibility are essential.

7.4 Security, privacy and permissions

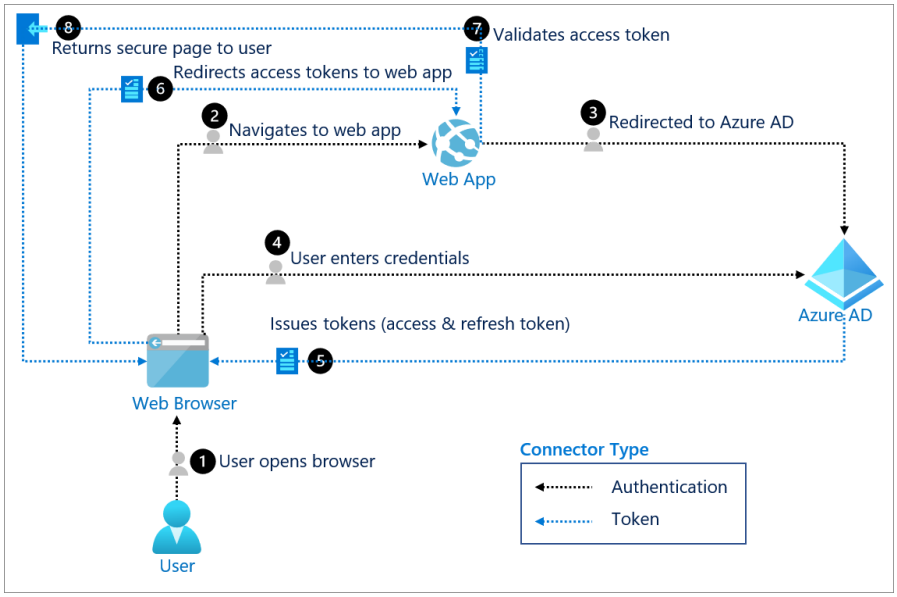

Figure 7.4: OAUTH 2.0

MCP explicitly recognises the sensitivity of enabling models to act on user systems and treats authorization as an important cross-cutting concern. When HTTP-based authorization is enabled, MCP relies on OAuth (the specification refers to OAuth 2.1 / OAuth 2.0 families and related RFCs) and classifies MCP servers as OAuth resource servers and MCP clients as OAuth clients. In practice this means the protocol defines how servers advertise their authorization servers (Protected Resource Metadata) and how clients discover and interact with those authorization endpoints.Token handling and validation are central requirements. MCP servers

acting as resource servers must validate access tokens according to OAuth rules (for example OAuth 2.1 §5.2) and reject invalid or expired tokens with appropriate HTTP responses, such as 401 for unauthorized, 403 for insufficient scopes, or 400 for malformed requests. Access tokens must never be sent in query strings; they must be presented in the Authorization header and stored and rotated securely. These requirements reduce token-theft risk and make it possible to reliably attribute actions to authenticated principals.

Audience binding and resource indicators are mandatory safeguards. MCP requires clients to include the ‘resource‘ parameter (RFC 8707) when requesting tokens and requires servers to verify that received tokens were issued specifically for that server (audience validation). Accepting tokens intended for other resources or forwarding unvalidated tokens to downstream APIs (token passthrough) creates "confused deputy" vulnerabilities and is explicitly forbidden by the MCP guidance. Implementations must therefore validate audience claims and avoid passing client-supplied tokens to other services without fresh, server-obtained credentials. OAuth operational safeguards are required in authorization flows. All authorization endpoints must be served over HTTPS, redirect URIs must be pre-registered and either localhost or HTTPS, and clients should use PKCE to protect the authorization-code flow. Public clients should rotate refresh tokens and authorization servers are encouraged to issue short-lived access tokens to reduce the blast radius of leaked credentials. Additionally, clients and servers must use standard metadata discovery (RFC8414 / RFC9728) and ‘WWW-Authenticate‘ headers to guide clients through error and discovery flows.

Consent, scoping and least privilege are fundamental principles. Beyond simple interactive confirmations in the UI, production deployments should apply fine-grained permissioning, including per-tool scopes, role-based access control, temporal tokens, and per-user or per-client quotas. UI surfaces must clearly present the trust boundary, indicating what data will be shared with a remote server and which tools are local-only, and request explicit consent before any side-effectful operation. Audit trails and provenance metadata should be recorded for every tool invocation so that actions can be traced and, where feasible, reversed. Local-first and transport-specific guidance is also provided: MCP treats authorization as transport-dependent, so HTTP(S)-based transports should follow the OAuth-based authorization specification, whereas STDIO or local transports should not implement the HTTP authorization flow and instead obtain credentials from the environment, for example using local secrets, OS key stores, or host-managed credentials. This allows private or local deployments to avoid sending sensitive data to external authorization servers while still enabling secure cloud-hosted scenarios.

Operational recommendations, which in the original text were presented as a checklist, can be described discursively as follows. All authentication endpoints should use HTTPS and have redirect URIs precisely registered. Authorization

code flows should implement PKCE, and public clients should rotate refresh tokens regularly. Tokens must be stored securely, with preference for short-lived access tokens, and incident response playbooks should include procedures for token revocation and rotation. Clients and servers should expose clear UI consent prompts, maintain detailed provenance logs for auditing, and provide users with mechanisms to review and revoke permissions.

Granular permissioning and trust boundaries are further emphasized beyond binary allow/deny confirmations. Production deployments benefit from more nuanced policies, including per-tool scopes, role-based access control, temporal tokens with short lifespans, and throttling quotas per client or per-user. Clients should present clear indications of trust boundaries, showing what data will be sent to remote servers and which tools are local-only, and offer mechanisms to audit past actions explicitly. Defensive programming is required because MCP allows models to request or manipulate external state: servers and clients should validate all inputs against declared schemas, enforce rate limits, and implement circuit-breakers for repeatedly failing tools. Clients should maintain a secure, append-only audit trail containing tool identifiers, inputs, user confirmations, timestamps, and returned outputs, enabling post-hoc review and rollback when needed.

Finally, data minimisation and telemetry principles should be applied whenever observability is collected. Metrics on error rates, latencies, and feature usage should avoid logging full user payloads, sensitive fields should be masked, and clients should provide opt-in or opt-out controls. Whenever possible, aggregated metrics should be preferred over per-request payload retention, balancing operational needs with the protection of user privacy.

7.5 SDKs

A variety of SDKs exist to help developers both implement MCP servers and connect clients to remote MCP servers. Commonly maintained SDKs include Python and TypeScript/Node libraries, and other language bindings may be available or community-contributed. These SDKs encapsulate recurring concerns such as discovery, transport handling, schema validation, streaming, authentication, and provenance, so developers can focus primarily on implementing tool logic rather than low-level protocol plumbing. A typical language-agnostic developer workflow involves defining a server manifest (for instance, `mcp.json`) listing tools, resource endpoints, and authentication requirements, then implementing tool handlers in a language-idiomatic way and registering them with a local server or a remote HTTP/SSE endpoint using SDK helper functions. On the client side, developers use the SDK to discover available tools, present them to the model or end user, and orchestrate tool calls while performing validation and handling user-consent flows.

SDKs, whether Python, TypeScript, or others, provide utilities that cover manifest parsing, transport adapters for stdio, SSE, and HTTP, schema-driven request/response validation, streaming helpers for long-running operations, and small UI integration helpers such as form rendering hints or JSON-schema → form metadata mappings that simplify client implementations. Python SDKs are often chosen for server-side tooling, rapid prototyping, and integrations that run on desktops or developer machines, frequently including simple stdio server scaffolds and ephemeral testing helpers suitable for CI pipelines. TypeScript/Node SDKs are popular for web and IDE integrations where strong typing and close integration with front-end stacks are important. TypeScript SDKs can generate typed clients directly from manifests and fit naturally into browser, extension, and VS Code ecosystems. Community-contributed bindings in other languages broaden adoption in environments such as JVM-based backends or systems programming contexts. Modern SDKs also include code-generation tools that produce typed client libraries or handler stubs from manifests, reducing boilerplate, preventing client-server contract mismatches, and improving developer ergonomics through auto-complete, compile-time checks, and type hints in IDEs.

Testing, CI, and contract validation are critical in SDK-driven development. Since MCP tools behave like RPC endpoints, standard software engineering practices apply: unit tests for handler logic, contract tests to validate manifest compatibility, integration tests for discovery and streaming flows, and end-to-end tests simulating client interactions including user-consent flows. SDKs provide utilities to spin up ephemeral stdio or HTTP servers in CI pipelines. Interactive inspection and replay tools such as MCP Inspector allow developers to debug and reproduce complex edge cases by connecting to local or remote MCP servers, visualizing discovered tools and resources, monitoring streaming events, and recording request/response sessions that can be exported as fixtures. This enables deterministic replay tests, validation of user-consent and provenance flows, and demonstration of issues in bug reports. Best practices combine unit and contract tests with Inspector-driven exploratory testing: developers use Inspector during development to triage streaming and reconnection behavior, generate small recorded fixtures for CI contract tests, and add automated checks such as manifest linting, schema validation, and replay-based integration tests. These combined practices ensure contract compatibility between client and server, enhance observability of long-running operations, and reduce runtime surprises in production deployments.

7.6 MCP Inspector.

The MCP Inspector is an interactive developer tool specifically designed for testing and debugging Model Context Protocol servers. It complements programmatic test

suites by providing a rapid, hands-on environment to inspect capabilities, exercise tools and record realistic sessions for later automated replay. The Inspector can be invoked directly via `npx` and does not require a prior global installation, which simplifies adoption and makes it convenient to include in developer documentation or troubleshooting guides.

Getting started and basic usage. The Inspector is typically launched with `npx` as follows:

```
npx @modelcontextprotocol/inspector <command>
npx @modelcontextprotocol/inspector <command> <arg1> <arg2>
```

To inspect published server packages from NPM or PyPI you can combine the Inspector invocation with the package runner, for example:

```
npx -y @modelcontextprotocol/inspector npx <package-name> <args>
# example:
npx -y @modelcontextprotocol/inspector npx
↪  @modelcontextprotocol/server-filesystem /path/to/dir
```

To inspect a locally developed server (repository checkout), run the Inspector and point it to the local entrypoint:

```
npx @modelcontextprotocol/inspector node path/to/server/index.js args...
```

Always consult the target server's README for any package-specific invocation details.

What the Inspector shows and why it matters. The Inspector UI is divided into panes that collectively expose the runtime surface of an MCP server. The *Server connection* pane allows selecting transport (stdio, HTTP/SSE, etc.) and tuning command-line arguments or environment variables for local servers. The *Resources* tab lists available resources with MIME types and metadata, and supports content inspection and subscription testing. The *Prompts* tab displays prompt templates, argument schemas and enables on-the-fly prompt testing with arbitrary arguments, while the *Tools* tab lists all declared tools, shows their JSON schemas and descriptions, and permits executing tools with custom inputs. The *Notifications* pane aggregates logs, server notifications and streamed events so developers can follow SSE or chunked outputs in real time.

Recording, replay and fixture generation. One of the most valuable Inspector features is session recording: the Inspector can save full request/response traces, including streamed updates, into fixtures that are exportable. These recorded sessions are ideal for creating deterministic integration tests in CI: the same sequence of interactions that reproduced an issue in the Inspector can be replayed automatically by a test harness, enabling regression coverage for complex streaming and reconnection scenarios.

Inspecting different server sources. The Inspector supports inspecting servers launched from NPM packages, PyPI packages or directly from a local checkout. This makes it straightforward to combine the Inspector with continuous integration: for example, CI can run the same server entrypoint used in development and then exercise a suite of recorded Inspector fixtures to validate fresh builds.

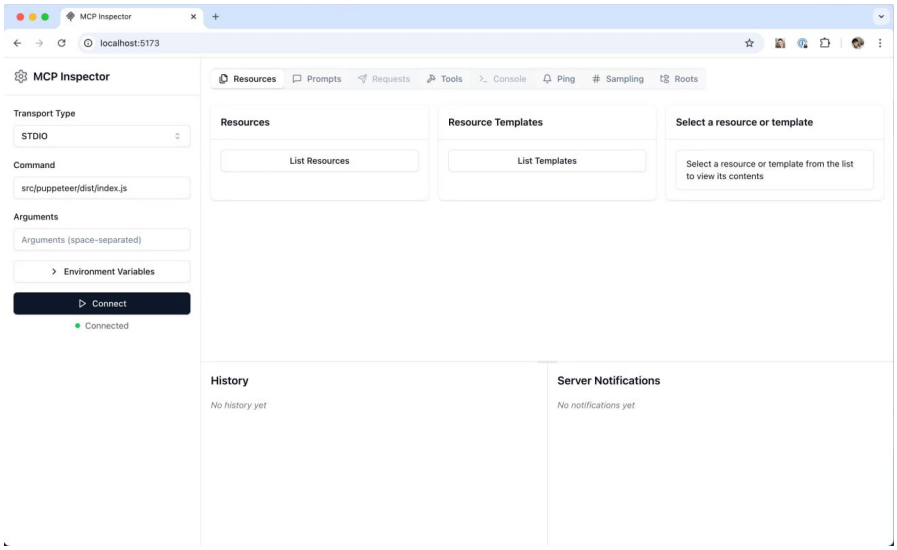

Figure 7.5: MCP Inspector

Best practices when using the Inspector. Use the Inspector as part of an iterative development loop: start it during development to verify connectivity and capability negotiation, exercise tools and prompts after each change, and use it to test edge cases such as invalid inputs, missing prompt arguments and concurrent invocations. When streaming behaviour is involved, use the Inspector to replicate reconnection and backpressure scenarios; record the sequence and add the fixture to the CI suite. Finally, combine Inspector-driven exploratory testing with automated checks (manifest linting, JSON-schema validation, and replay-based integration tests) to catch regressions early and improve the overall robustness of the client/server contract.

Development workflow recommendation. A recommended workflow is to begin with the Inspector to triage and understand any runtime or streaming issues, produce a minimal reproducible fixture, add a unit or integration test that replays that fixture in CI, and then iterate on fixes in the server code. This pattern shortens the feedback loop, provides concrete reproduction steps for issue reports and ensures that complex behaviours—especially around streaming and reconnection—are covered by automated tests rather than relying solely on manual verification.

Where to find more information. The Inspector repository and the MCP documentation include an in-depth guide covering installation, feature descriptions and example workflows. Consult the official MCP Inspector documentation for the latest commands, platform-specific notes and troubleshooting tips.

Best practice is to combine unit/contract tests with Inspector-driven exploratory testing: use Inspector during development to triage streaming and reconnection behaviour, generate small recorded fixtures for CI contract tests, and add automated checks (linting of manifests, schema validation, and replay-based integration tests) into the CI pipeline. Together these techniques help ensure contract compatibility between client and server, improve observability of long-running operations, and reduce runtime surprises in production deployments.

7.7 Python SDK

Figure 7.6: MCP Python Framework

For our implementation, we adopted the Python ecosystem as the foundation for the MCP server. Python remains the lingua franca in applied machine learning and research-oriented engineering due to its extensive scientific libraries, large and active community, and capabilities for rapid prototyping. These characteristics make it a natural choice when the project demands close interaction with model tooling, data pipelines, and experimental code, allowing developers to iterate quickly and integrate seamlessly with existing ML workflows.

Python offers several practical advantages for MCP development. Its mature ecosystem provides libraries for JSON schema validation, asynchronous I/O, HTTP/SSE transport handling, testing frameworks, and CI integration tools, all of which reduce boilerplate and simplify the creation of a robust server. The language's concise syntax and dynamic typing facilitate rapid iteration on tool handlers and manifests, which is especially valuable for specification-driven development like MCP. Additionally, Python's interoperability with common ML frameworks and data-processing tools makes it straightforward to construct tools that rely on model outputs, perform file processing, or extract features from data. Finally, many researchers and engineers on our team were already proficient in Python, which reduced the ramp-up time and allowed faster debugging and testing cycles.

The Python SDK itself provides a rich set of capabilities that are particularly useful for MCP servers. It includes utilities to read, validate, and serve `mcp.json`-style manifests, ensuring that server capabilities are discoverable by clients. Built-in transport adapters support both local (stdio) and remote (HTTP/SSE) servers, while schema-driven request and response validation guarantees adherence to declared contracts. Streaming helpers allow for partial outputs and progress updates during long-running operations, and provenance and metadata conventions enable auditing and rollback. Testing utilities facilitate the creation of ephemeral stdio or HTTP servers in CI pipelines, verifying contract compatibility and end-to-end flows. Optional code-generation or handler-stub features further reduce boilerplate and provide typed client stubs, improving developer ergonomics.

For our server-side implementation, we based our work on the MCP Python SDK repository published by Anthropic on GitHub. This repository offered immediate scaffolding for stdio and HTTP transports and included the testing utilities required to iterate quickly on manifests and handlers. The client-side integration for our project was Copilot Studio, so our focus was on producing a server that was easily discoverable and consumable, with stable streaming behavior, clear provenance metadata, and robust manifest semantics. During development, we also identified areas where improvements could be made to enhance reliability and interoperability, which led us to contribute fixes upstream (see next section).

Despite the Python SDK's strong support for local development and testing with `mcp-inspector`, we encountered limitations when deploying the server on external hosts, particularly in scenarios involving OAuth-based authentication for Microsoft services. Tools that did not require authentication functioned correctly, confirming that the core server logic was sound, but tools relying on OAuth consistently failed due to the SDK's handling of Microsoft OAuth flows in remote environments. This ultimately motivated a transition to the TypeScript SDK, which provided a reliable foundation for authenticated MCP tools in production.

7.8 Contributions to the MCP Python SDK

During the development and deployment of the MCP-based server described in this thesis, our team encountered a reproducible stability problem affecting the streamable-http transport when the server was deployed in a public hosting environment. The problem manifested as a failure after the OAuth authorization phase: while local testing with MCP Inspector worked without issue, the same flow deployed to Azure repeatedly failed at a subsequent POST request which was observed to return an unauthorized response. This behaviour produced an authorization loop that prevented the client from completing the MCP handshake. The anomaly did not appear when using the SSE transport, which initially suggested a transport-specific interaction between the HTTP stack, reverse proxying and the authorization middleware.

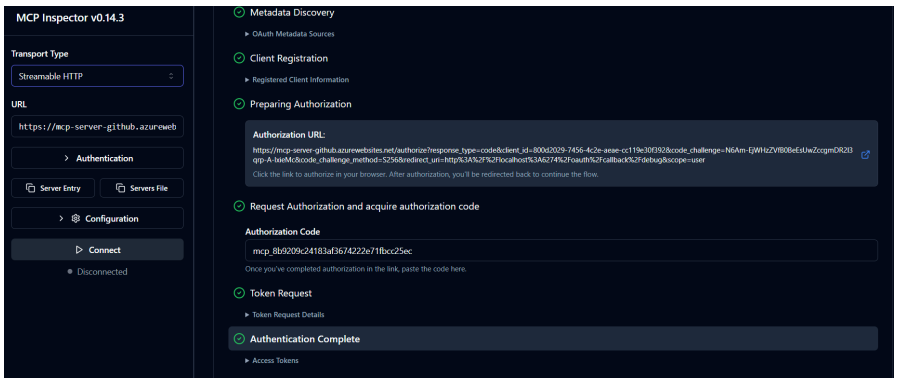

Figure 7.7: MCP inspector successfully authenticate the user

Figure 7.8: Error on the browser when usign a tool

A systematic diagnosis, performed in collaboration with another contributor, revealed the root cause: during an internal redirect from the route path "/mcp" to "/mcp/" the Authorization header was being dropped. The header loss turned an otherwise valid token exchange into an unauthorized POST that restarted the redirect sequence. This subtle interplay did not surface in local tests because certain local stacks (for example Uvicorn combined with HTTPX in development configurations) re-attached or preserved the Authorization header, masking the underlying defect. Once exercised under a public deployment model the discrepancy became evident.

To address the issue we modified the streamable-http route registration so that the handler would be attached directly as a Route rather than mounted with a

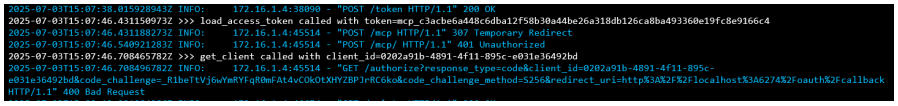

Figure 7.9: The error logged on azure console

component that provoked a redirect. Concretely, the fix replaces a mount-based registration with a direct Route mapping that passes requests to the handler wrapped by the authorization middleware when token verification is enabled. The key change can be expressed succinctly in code form; the critical fragment of the server routing logic was adjusted as follows:

```
- from starlette.routing import Mount
+ from starlette.routing import Route

- routes: list[Route | Mount] = []
+ routes: list[Route] = []

- routes.append(
-     Mount(
-         self.settings.streamable_http_path,
-         app=RequireAuthMiddleware(handle_streamable_http, required_scopes, resource_metadata_url),
-     )
- )
+ routes.append(
+     Route(
+         self.settings.streamable_http_path,
+         endpoint=RequireAuthMiddleware(handle_streamable_http, required_scopes, resource_metadata_url),
+         methods=["POST", "GET", "DELETE"],
+     )
+ )

- routes.append(
-     Mount(
-         self.settings.streamable_http_path,
-         app=handle_streamable_http,
-     )
- )
+ routes.append(
+     Route(
+         self.settings.streamable_http_path,
+         endpoint=handle_streamable_http,
```

```
+          methods=["POST", "GET", "DELETE"],
+      )
+ )
```

This change avoids an implicit redirect that was responsible for dropping headers during the transition between mounted subpaths. After applying the fix the server accepted the POST requests in the previously failing deployment scenarios and the authorization loop disappeared. The implemented solution was verified against the original failing Azure deployment and with the MCP Inspector to ensure parity between local and public behaviours.

Beyond the immediate code modification, we extended the project work to include a reproducible diagnosis and additional tests that exercise the streamable-http transport path under authenticated conditions. The tests were designed to simulate the redirect behaviour and to assert that authorization headers are preserved across the relevant request lifecycles. These tests increase confidence that regressions of the same class will be detected during continuous integration, and they document the behavioural contract expected of the HTTP transport in hosted environments.

Following standard open-source practice, the diagnosis, patch and tests were prepared as a set of changes and submitted as a pull request to the upstream MCP Python SDK repository. The contribution aimed not only to stabilise our deployment but also to improve interoperability of the SDK for other users who might encounter similar hosting nuances. Although the issue was resolved in the upstream project, the eventual merge was applied from an alternative contribution. Our patch preceded the accepted change in time and correctly addressed the issue, but the maintainers elected to merge another submission that solved the same problem. This outcome illustrates several aspects of collaborative open-source development: the importance of clear reproduction steps and tests in accelerating triage, the timing sensitivity of pull requests in active repositories, and the sometimes independent evolution of parallel fixes. Even when a particular PR is not merged, documenting the issue and demonstrating a robust fix with tests contributes to collective knowledge and strengthens the case for integrating similar safeguards in the codebase.

The technical and social outcomes of this episode were meaningful for the project described in this thesis. Technically, fixing the transport header drop increased runtime stability and reduced a class of environment-specific failures that would have undermined operational reliability. The addition of targeted tests improved the SDK's resilience to routing and hosting idiosyncrasies and provided a basis for future regression detection. Socially, the engagement with the upstream repository reinforced good open-source hygiene: constructing minimal reproductions, providing thorough explanations of environment-dependent behaviour, collaborating with other contributors during debugging, and submitting tests alongside code

changes. From a governance perspective, the experience highlighted the utility of contributing operational fixes upstream rather than maintaining long-term forks, because upstream fixes benefit the entire ecosystem and reduce duplication of maintenance effort.

Based on this experience, recommended follow-ups include expanding the integration test matrix in continuous integration environments to explicitly cover authenticated transports under common hosting configurations, documenting known hosting pitfalls in the SDK's contributor guide, and asserting header-preservation properties in end-to-end tests. These measures would help prevent regressions and shorten the feedback loop for future contributors who encounter similar interactions between routing primitives and authentication middleware.

7.9 TypeScript SDK

Following early experiments with the Python SDK, we decided to move parts of the implementation to a TypeScript-based stack. The primary motivation for this change was compatibility: our target client for the project was Copilot Studio, and the Python server variant exhibited integration frictions that prevented reliable operation with that client. To maximize interoperability, we adopted a TypeScript reference implementation published by Microsoft as the starting point for the server-side work. This reference provided a working example of the full authentication flow, including OAuth for personal and organizational accounts, along with practical scaffolding that mapped naturally to web and IDE client patterns.

The Microsoft TypeScript example was chosen because it implemented OAuth flows and credential handling aligned with Copilot Studio expectations, followed idioms common in web and extension ecosystems such as async/await and promise-based handlers, and included clear examples for HTTP/SSE transports and manifest exposure. These characteristics significantly reduced the effort required to produce a server that Copilot Studio could reliably discover and interact with.

Migrating from Python to TypeScript revealed more than syntactic differences. The two SDKs embody distinct architectural conventions for request/response handling, stream semantics, and error propagation. Patterns that were straightforward in Python, such as sync-style handlers, blocking I/O assumptions, or certain reconnection heuristics, required redesign to fit TypeScript/Node idioms, including non-blocking I/O, explicit Promise lifecycles, and careful management of stream backpressure. Long-running tools emitting incremental results needed adaptation from Python generator or async-iterator models into Node streams or SSE chunking, with attention to client reconnection behavior.

One immediate challenge was that `mcp-inspector`, previously used for Python development, did not interoperate out-of-the-box with TypeScript due to differences

in transport framing and manifest shapes. We addressed this by implementing small compatibility adapters to normalize manifest and stream behavior, recording request/response fixtures from manual Inspector sessions for deterministic replay tests, and adding contract, manifest validation, and end-to-end streaming tests in the TypeScript CI pipeline. These measures restored a productive development loop while keeping Inspector useful for interactive debugging.

Despite the migration cost, the TypeScript stack delivered practical advantages. OAuth integration became reliable, resolving authorization issues observed with Python. TypeScript's alignment with browser and IDE ecosystems simplified client deployment and consumption, while static typing and IDE tooling reduced runtime contract errors and improved ergonomics when generating typed clients from manifests. Generated TypeScript client stubs could be used directly by Copilot Studio plugins or web front-ends with minimal translation, further streamlining integration.

Recommended practices for TypeScript-based MCP servers include enabling strict TypeScript options, generating types from manifests to check client/server contracts at compile time, incorporating schema validation and manifest linting into CI pipelines, adding replay fixtures from Inspector or recorded sessions to validate streaming deterministically, and providing a small compatibility proxy or adapter layer when development tools assume different framing. Collectively, these practices ensure reliable production deployments and maintain compatibility with interactive development tools.

In conclusion, the TypeScript SDK, anchored on Microsoft's reference implementation, proved to be a practical, production-ready foundation for the MCP server designed to interoperate with Copilot Studio. The migration required careful adaptation of streaming and reconnection semantics and modest investments in test and compatibility infrastructure, but delivered robust authentication, smoother integration with IDE and web clients, and a cleaner contract surface for the remainder of the project.

7.9.1 APIs

The following sub-sections document the RESTful endpoints and tool interfaces exposed by our MCP server implementation for integration with Microsoft Copilot agents. Each described endpoint includes a functional description, expected authorization requirements, canonical request and response shapes, and brief operational notes regarding idempotency, time zone handling and error semantics. All Microsoft Graph–backed endpoints require an OAuth2 access token valid for the server resource; the only exception is the weather endpoint which queries a public service and does not require authentication.

Weather: `get_weather`

The weather tool provides a text description of current weather conditions for a requested city by querying the public `wttr.in` service. The only input is the city name; when the city is ambiguous the tool returns a short disambiguation hint. Because the endpoint is unauthenticated, it is suitable for lightweight contextual answers embedded in conversational replies.

```
GET https://{host}/mcp/tools/weather/get_weather?city=Rome
Authorization: none
```

```
{
  "city": "Rome",
  "summary": "Partly cloudy, temperature 18°C, wind 10 km/h from NW",
  "raw": "...wttr.in response..."
}
```

Calendar: `list_events`

The `list_events` tool retrieves calendar events in a given UTC interval for the authenticated user; callers must supply `start` and `end` in ISO 8601 UTC. The server returns event summaries, start and end times (in UTC) and identifiers suitable for subsequent update or deletion operations. Clients and models are expected to present returned times to users in the Italian timezone.

```
POST https://{host}/mcp/tools/calendar/list_events
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{"start":"2025-09-20T08:00:00Z","end":"2025-09-20T18:00:00Z"}
```

```
{
  "events": [
    {"id":"evt_1","subject":"Standup","start":"2025-09-20T08:30:00Z","e⌋
    ↪  nd":"2025-09-20T09:00:00Z"},
    {"id":"evt_2","subject":"Project meeting","start":"2025-09-20T10:00⌋
    ↪  :00Z","end":"2025-09-20T11:00:00Z"}
  ]
}
```

Calendar: `create_meeting`

`create_meeting` constructs a Teams meeting and registers it on the user's calendar. Inputs include subject, attendee emails and a date with local start/end times in Italian locale; the server converts these to UTC for Graph API calls, creates

the event and returns the event identifier together with the Teams meeting join URL. The operation is side-effecting and requires explicit user consent via the authorization flow.

```
POST https://{host}/mcp/tools/calendar/create_meeting
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{
  "subject":"Riunione con il team",
  "attendees":["alice@example.com","bob@example.com"],
  "date":"2025-09-25",
  "startTime":"14:00",
  "endTime":"15:00"
}
```

```
{
  "eventId":"evt_abc123",
  "teamsJoinUrl":"https://teams.microsoft.com/l/meetup-join/..."
}
```

Calendar: `get_team_availability`

This tool computes common free slots for a list of participants inside a date range, using the specified slot granularity. Inputs are attendee emails and date bounds in Italian local date format; outputs list only those slots where all participants are free within working hours (08:00–19:00 CET/CEST) and present results in the Italian timezone. The server enforces sensible defaults for the availability interval and bounds the returned set to avoid overwhelming responses.

```
POST https://{host}/mcp/tools/calendar/get_team_availability
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{"attendees":["alice@example.com","bob@example.com"],"start":"2025-09-2
↪  5","end":"2025-09-27","availabilityViewInterval":60}
```

```
{
  "availableSlots":[
    {"start":"2025-09-25T09:00:00+02:00","end":"2025-09-25T10:00:00+02:
    ↪  00"},
    {"start":"2025-09-26T11:00:00+02:00","end":"2025-09-26T12:00:00+02:
    ↪  00"}
  ]
}
```

Calendar: `update_event` and `delete_event`

The `update_event` endpoint accepts an event identifier and a partial update object and applies the changes via Microsoft Graph after converting local times to UTC; the `delete_event` endpoint removes the event and returns a confirmation message. Both endpoints require authorization and perform schema validation on inputs to prevent malformed requests. Clients should treat update and delete as non-idempotent operations unless they supply an idempotency key at the transport layer.

```
PATCH https://{host}/mcp/tools/calendar/update_event
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{"eventId":"evt_abc123","updates":{"subject":"Updated subject","start":
↪  "2025-09-25T13:00:00+02:00","end":"2025-09-25T14:00:00+02:00"}}
```

```
{"status":"ok","eventId":"evt_abc123","message":"Event updated"}
```

Mail: `list_emails` and `send_email`

`list_emails` fetches the most recent messages from the authenticated user's inbox constrained by a `count` parameter; the server returns a compact listing containing date, sender and subject suitable for conversational summaries. `send_email` composes and dispatches a plain-text message on behalf of the user, returning a delivery confirmation. Both tools require valid Microsoft Graph scopes and include clear error messages when the mailbox is unavailable or the token is insufficient.

```
POST https://{host}/mcp/tools/mail/list_emails
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{"count":5}
```

```
{
  "emails":[
    {"id":"m1","from":"ceo@example.com","subject":"Quarterly
    ↪  update","date":"2025-09-01T09:00:00Z"},
    {"id":"m2","from":"hr@example.com","subject":"Policy
    ↪  update","date":"2025-09-02T11:00:00Z"}
  ]
}
```

```
POST https://{host}/mcp/tools/mail/send_email
Authorization: Bearer $ACCESS_TOKEN
```

```
Content-Type: application/json

{"to":"colleague@example.com","subject":"Meeting notes","body":"Please
↪  find attached the notes from today's meeting."}
```

```
{"status":"sent","messageId":"msg_98765"}
```

Colleagues: `find_colleagues_by_department`

The colleagues tool searches the directory using the `officeLocation` attribute to return personnel records; the `mode` parameter toggles between partial and exact matching and a `maxResults` limit constrains response size. Returned records include name, role, officeLocation, email and telephone numbers, and a structured JSON representation suitable for programmatic consumption by front-ends.

```
POST https://{host}/mcp/tools/colleagues/find_colleagues_by_department
Authorization: Bearer $ACCESS_TOKEN
Content-Type: application/json

{"office":"R&D","mode":"partial","maxResults":25}
```

```
{
  "results":[
    {"name":"Giulia Rossi","role":"Researcher","officeLocation":"R&D
    ↪  Rome","email":"giulia.rossi@example.com","phone":"+39061234567"}
  ]
}
```

Error handling and operational notes

When tokens are missing or expired, endpoints return clear, human-readable diagnostics with standard HTTP codes (401 for unauthorized, 403 for forbidden). API errors from Microsoft Graph are surfaced with status and body to facilitate debugging while the server masks sensitive internal traces. Date and time inputs must be normalised by the caller to the expected format: the LLM or client should convert local Italian times to UTC before invoking Graph-backed tools, and servers perform server-side validation and conversion to ensure correctness. For streaming or long-running operations (for example large mailbox fetches or complex availability computation), the server may emit incremental progress events over SSE or streamable HTTP and provide an operation identifier for reconciliation and retries.

Operational environment variables and troubleshooting

The server relies on a small set of environment variables for OAuth configuration and hostname resolution; operators should ensure the correct `CLIENT_ID`, `CLIENT_SECRET`, tenant identifiers and callback paths are configured and that registered redirect URIs exactly match deployment endpoints. In hosted deployments, care must be taken to select the appropriate transport (SSE or streamable HTTP) and to verify that reverse proxies preserve Authorization headers, as described earlier in the contributions section.

7.10 Copilot Integration

To make our MCP server fully usable within a Microsoft environment, we undertook a direct integration with **Microsoft Copilot Studio**. This involved creating a custom connector that allowed the server to communicate with Copilot agents while ensuring that all calls were authenticated securely via Azure App Registration. The integration was designed so that tools exposed by our MCP server would be automatically discoverable by Copilot, callable through the Streamable HTTP transport, and protected using OAuth2. This setup provided both seamless functionality for the end user and adherence to enterprise security standards.

Creating an Agent

The first stage in the integration workflow was the creation of a new agent inside Copilot Studio. This required logging into `https://copilotstudio.microsoft.com`, navigating to the *Agents* tab, and selecting *New Agent*. Once the agent was created, its language and configuration parameters were set to match the requirements of the target environment. Opening the *Settings* panel, we ensured that the *Generative AI orchestration* feature was enabled. This setting is essential because it allows Copilot agents to communicate with external MCP servers and orchestrate tool invocations, forming the core of the integration pipeline. By following these steps, we established the foundational agent that could later be associated with custom connectors and OAuth credentials.

Creating a Custom Connector

Following agent creation, the next step was to enable communication between the MCP server and Copilot through a custom connector. From Copilot Studio, we accessed the *Tools* section, selected *Add a Tool*, and chose *Custom Connector*. The connector was then imported from GitHub using the official *MCP-Streamable-HTTP* template provided by Microsoft. Configuration required specifying the *Host*,

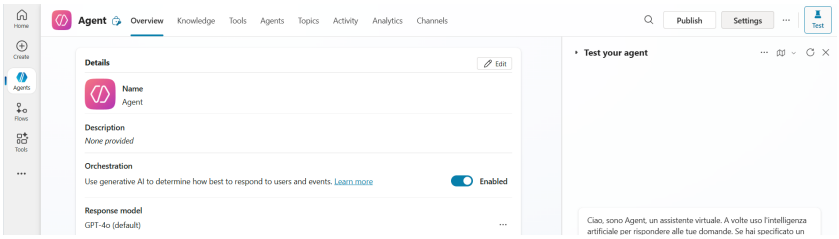

Figure 7.10: Creation of a Copilot agent.

which pointed to the domain where our MCP server was hosted (omitting the `https://` prefix), while the *Base URL* remained fixed as `/mcp`. This step allowed the connector to serve as a bridge between the agent and the MCP server, mapping requests to the correct endpoints and enabling real-time streaming of tool outputs. The connector also standardized authentication and ensured that calls could be routed securely to the appropriate server resources.

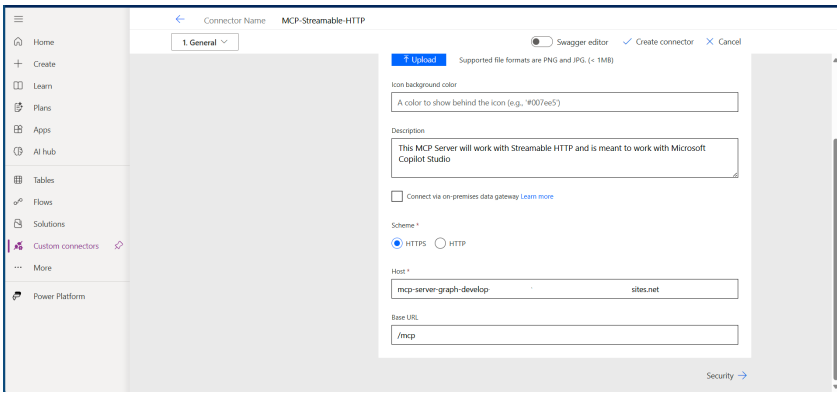

Figure 7.11: Configuration of MCP server host and base URL in Power Apps custom connector.

OAuth2 Authentication Setup

Authentication for the connector was handled via OAuth2 using Azure App Registration. Client credentials, including the **Client ID** and **Client Secret**, were retrieved from the Azure portal. Additionally, the **Authorization URL** and **Token URL** were obtained from Azure Endpoints, and the required **Scope** values were copied from the *API Permissions* page. This configuration ensured that the connector could authenticate on behalf of the agent, acquiring tokens that allowed secure and auditable access to the MCP server. By relying on OAuth2, we guaranteed that all interactions between Copilot and the MCP server adhered to

Microsoft's recommended security and identity management practices.

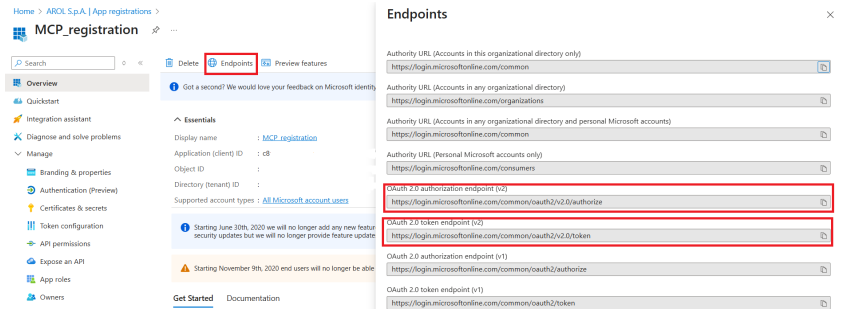

Figure 7.12: OAuth2 endpoint configuration in Power Apps form.

Redirect URI Configuration

During the OAuth2 setup, Copilot Studio generates a *Redirect URI*, which is critical for completing the authorization flow. Initially, this URI triggers errors if it is not explicitly registered in Azure. To address this, the URI provided by Copilot Studio was copied and registered in the Azure App Registration portal under *Authentication* as a *Web* redirect. This step ensured that token redirection would be recognized as valid by the Azure authentication server, allowing seamless token exchanges between Copilot and the MCP server. After registration, repeating the connection process in Copilot Studio succeeded without errors, and the server tools became visible and callable through the agent interface.

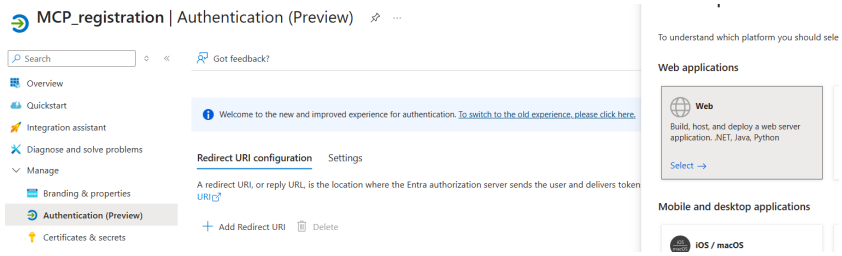

Figure 7.13: Adding the Redirect URI in Azure App Registration.

Outcome

Once fully configured, our MCP server was successfully integrated as a first-class tool provider within Microsoft Copilot Studio. The integration relied on three core components: the official MCP-Streamable-HTTP connector template, OAuth2 authentication via Azure App Registration, and proper registration of the Redirect

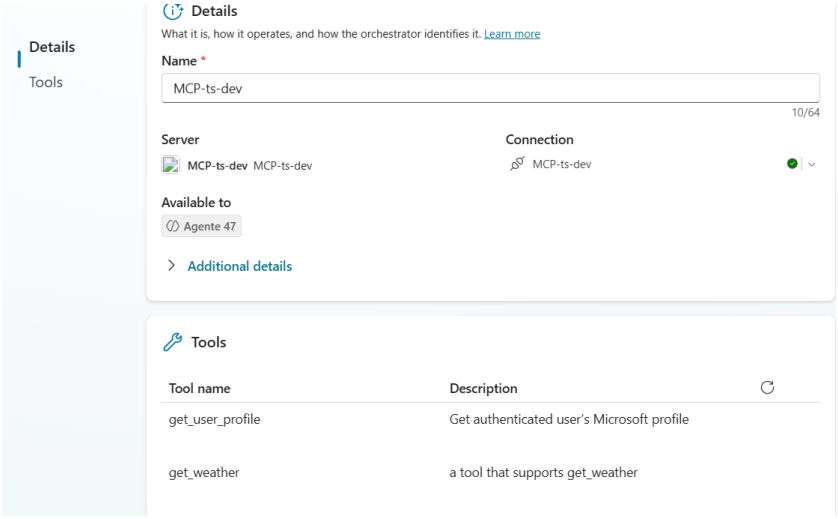

Figure 7.14: Successful integration: MCP server tools exposed inside Copilot Studio.

URI. With this setup, all server tools were discoverable, callable, and secured according to enterprise-grade standards. This approach demonstrates that MCP servers can be exposed in a highly controlled manner while remaining fully accessible to conversational AI workflows, providing a robust bridge between external services and Copilot agents.

Chapter 8

MCP Server Deployment on the Azure Web Services on Cloud

Cloud platforms have become the de-facto infrastructure for modern applications, providing elasticity, managed services and global availability. Among major providers, Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure stand out for the breadth of their offerings and their enterprise-oriented features. Although the MCP server could in principle be deployed on any of these clouds — for instance using container orchestration or serverless products — we selected Microsoft Azure because its ecosystem and native integrations aligned directly with the goals of this project. Azure couples a wide range of platform services with first-class identity and governance tooling, and it offers an AI-focused toolchain and strong interoperability with Microsoft productivity software; these attributes made it the most natural choice for an MCP deployment intended to interoperate with Copilot Studio and Microsoft 365.

Azure's global footprint and region model allowed us to position the service close to end users and to respect potential data-residency constraints. From a security and governance perspective, the tight integration with Entra ID / Azure AD simplified the implementation of authentication and authorization flows, while policy and resource management features helped enforcing compliance during rollout. In practice, the platform's integration with development toolchains — GitHub Actions, Azure DevOps and IDEs like Visual Studio — accelerated our CI/CD pipelines and shortened the feedback loop for iterative changes. These factors are particularly relevant when integrating with Microsoft clients, because they reduce the operational complexity required to obtain a stable and repeatable connection between cloud services and enterprise-grade assistants.

We opted for a Platform-as-a-Service approach by hosting the MCP front-end on Azure Web Service (part of the App Service family) in order to minimise operational overhead. Using App Service removes the need to manage operating system patches or runtime updates: the platform handles the lifecycle of the runtime, supports slot-based deployments for low-downtime releases, and exposes autoscaling primitives to adapt to variable workloads. This allowed the team to concentrate on application logic — MCP manifests, transport handling, and streaming semantics — while relying on Azure for runtime, resilience and monitoring capabilities. In addition, App Service integrates seamlessly with Key Vault for secure secret management and with Application Insights for rich telemetry, which significantly sped up debugging and the analysis of live behaviour.

Azure's broader capabilities were also important to the project beyond simple hosting. The platform provides managed services for data ingestion, OCR, indexing and vector search that are useful when preparing and serving RAG corpora at scale. It also supports hybrid deployment models (for example via Azure Arc or on-premises gateways) which enable architectures where sensitive data remains on-premise while a cloud-hosted MCP front-end handles client interactions. Finally, App Service supports multiple hosting models and runtimes — from direct code deployments to container images and Kubernetes-backed topologies — which eased the migration from a local prototype to a production-ready instance by allowing us to choose the most appropriate deployment model without being constrained by the platform.

8.1 Azure Web Service

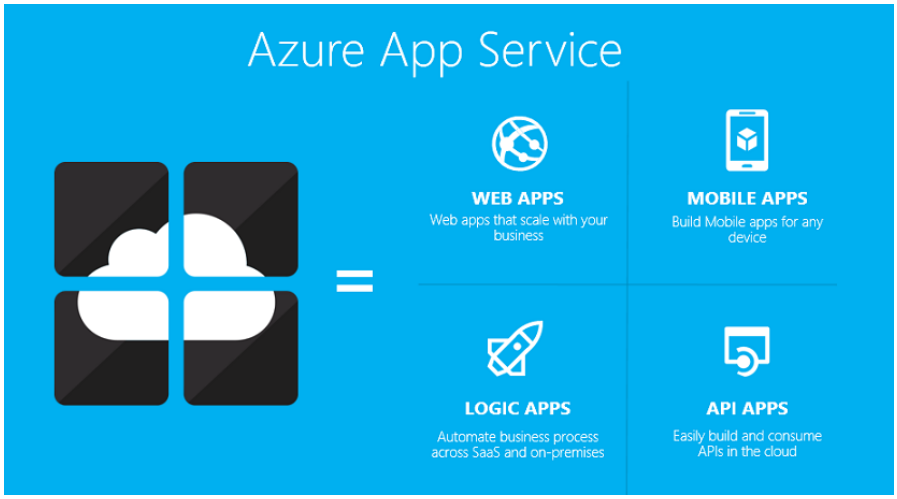

Figure 8.1: Azure Web Servic capabilities

Azure Web Service is designed to host web applications, APIs and microservices in a managed PaaS environment that abstracts many of the operational details of traditional infrastructure. The service manages runtime patching and platform updates, simplifies deployments through slot-based release strategies and offers autoscaling to cope with changing traffic patterns. Crucially for this project, App Service provides native bindings to Entra ID / Azure AD for OAuth/OIDC workflows, managed certificate handling and seamless integration with Key Vault, all of which are essential when the service must interact with authenticated clients such as Copilot Studio.

Operationally, App Service exposes diagnostic consoles and integrated logging that proved essential during validation: real-time logs and application telemetry allowed us to confirm that the server accepted requests, processed authentication tokens correctly and emitted metrics useful for monitoring. For heavier workloads or large-scale data processing, the front-end hosted on App Service can be paired with specialised managed services — such as Blob Storage for document assets, Azure SQL or Cosmos DB for structured state, and managed indexing or vector search services for retrieval workloads — thereby separating the front-end responsibilities from compute-intensive pipelines.

In short, choosing Azure Web Service provided a low-maintenance cloud presence for the MCP server while preserving direct access to Azure's identity, observability and AI ecosystems. This allowed us to deliver a production-ready endpoint that integrates securely with Copilot Studio and can evolve to use additional managed services as the project's data and AI needs grow.

8.2 MCP Server Integration

The integration of the MCP server into Azure marked a fundamental step in our project, since it allowed us to move from a local prototype to a fully deployable service accessible in the cloud. This transition required several configuration steps, but also highlighted the advantages of adopting Azure Web Service as the hosting environment.

The first step was the setup of the Azure environment, starting from the landing page (Figure 8.2), which acts as the main hub for all resource management. From here, we were able to create and configure the service instance that would host our MCP server. Azure provides a wide range of deployment options, but for our purposes we focused on building a lightweight web service that could handle requests from the client and maintain integration with Microsoft's authentication flow.

Once the resource was created, attention shifted to the definition of the server distribution (Figure 8.3). This step is crucial because it specifies how the server

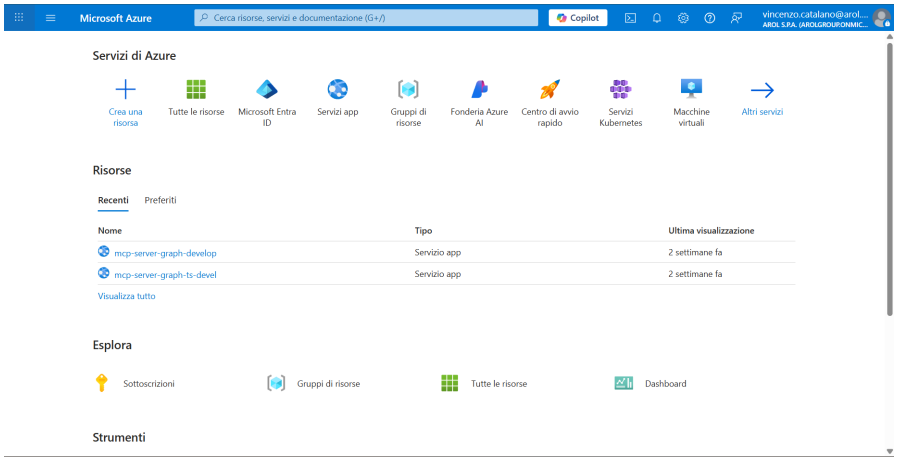

Figure 8.2: Azure landing page

code is packaged and deployed in the cloud. The configuration includes the choice of runtime, the definition of scaling parameters, and the allocation of computing resources. At this stage, we also encountered some differences compared to our Python implementation: while the TypeScript SDK required slightly different logic in the server setup, Azure provided a unified deployment framework that made the migration smoother.

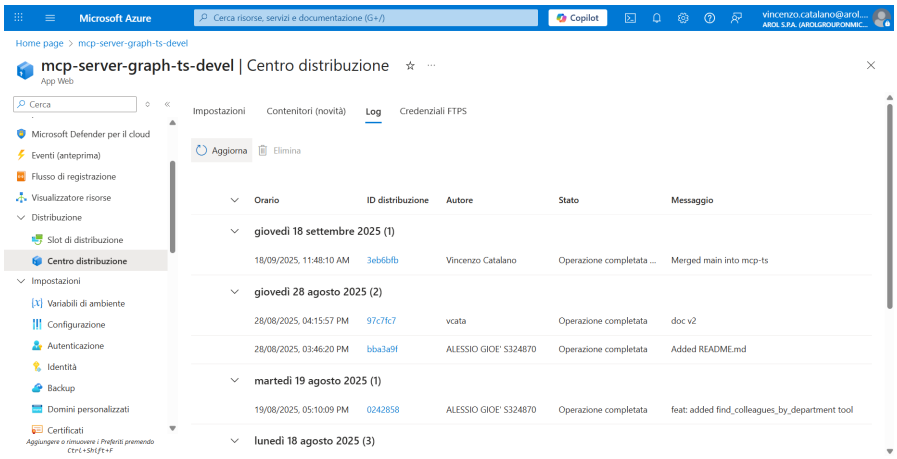

Figure 8.3: Server distributions

Another key aspect of the integration process involved the definition of environment variables (Figure 8.4).

These variables allowed us to externalize sensitive data (such as API keys, project identifiers, and Microsoft account configurations) without hardcoding them in the application. This approach not only improves security, but also makes it

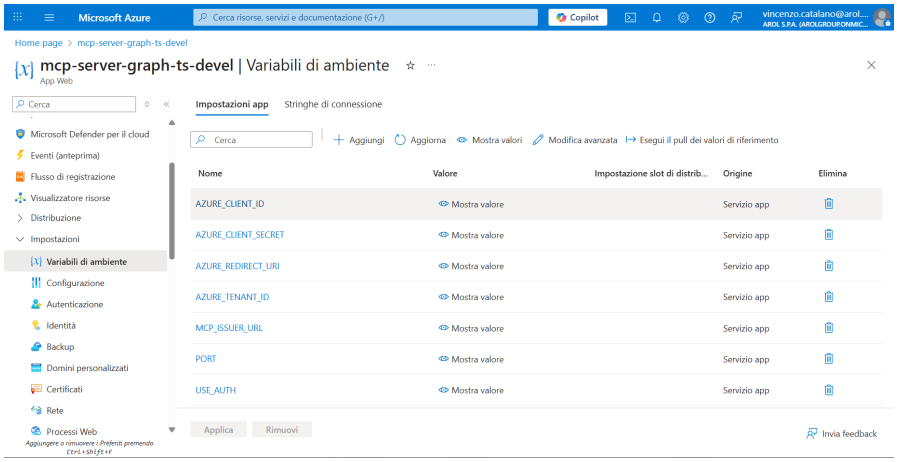

Figure 8.4: Server environment variables

easier to manage different environments (development, staging, production) without modifying the source code.

Finally, after the service was deployed, we verified its functionality directly from the Azure portal by inspecting the server logs (Figure 8.6).

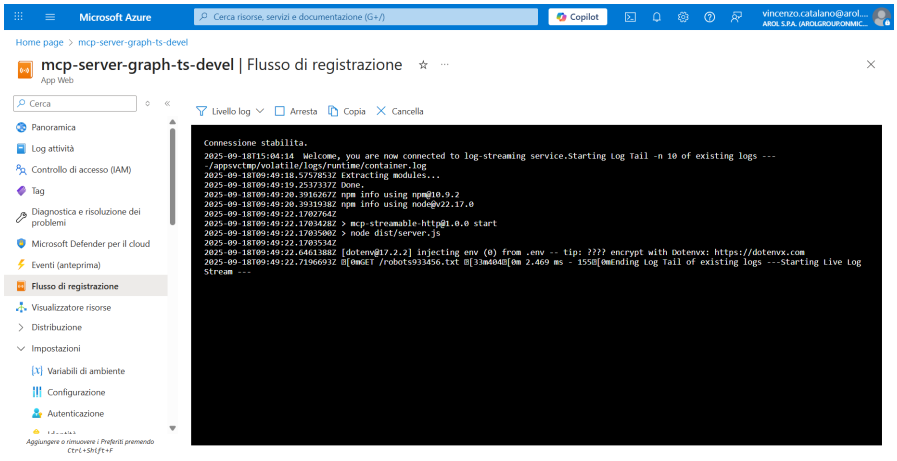

Figure 8.5: Server log terminal

This step was essential for debugging and validation: through the real-time output, we confirmed that the server was running correctly, responding to client requests, and properly handling authentication. In some cases, the logs also highlighted configuration errors or misalignments in environment variables, which we could quickly fix thanks to the integrated monitoring tools.

Overall, the integration of the MCP server into Azure demonstrated how a cloud environment can simplify deployment and maintenance. The platform not only

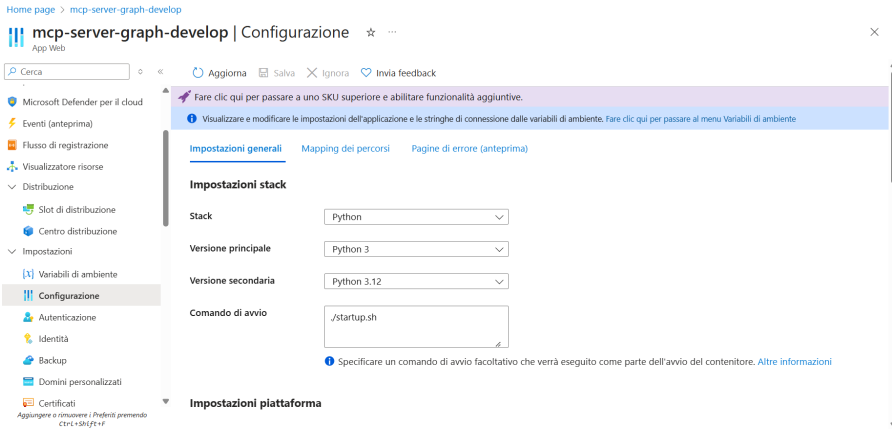

Figure 8.6: Server running script

provided us with a reliable hosting infrastructure, but also with integrated tools for monitoring, authentication, and scaling. This reinforced our decision to adopt Azure Web Service over alternative providers and underlined the importance of cloud-native development in modern software projects.

Chapter 9

End User Validation

A fundamental step in the development of enterprise AI systems is the validation phase, where the solution is tested in a realistic business context. Designing and implementing an intelligent agent in a controlled, laboratory-like environment can provide useful insights on architecture, performance, and feature completeness, but it is only by exposing the system to real users and real data that its true value emerges. For this reason, once the main components of our solution had been integrated, we carried out a structured testing activity within the company.

The validation process was not limited to verifying whether the bot could technically answer queries, but aimed at assessing a broader set of dimensions. These included: the **usability** of the interface, the **effectiveness** in solving concrete business problems, the **integration** with existing workflows, and the **stability** of the infrastructure under realistic conditions. In line with common practices in enterprise IT projects, our approach can be described as a *pilot test*, where a selected group of employees is involved in an exploratory deployment to provide feedback before scaling the solution more widely. This methodology offers the dual advantage of gathering valuable qualitative feedback from early adopters and identifying potential issues in a safe, low-risk setting.

The chosen entry point for the validation was **Microsoft Teams**, a platform already heavily adopted in the organization. Integrating the agent directly into Teams provided several benefits: users were not required to learn a new tool, the chatbot was seamlessly embedded into their daily collaboration environment, and the barrier to adoption was significantly reduced. From a system perspective, Teams acted as the front-end layer, routing all interactions to the MCP server, which orchestrated reasoning and tool usage, and then returning the output back to Teams in a natural conversation flow. This setup created an experience consistent with common chat applications, but backed by a powerful AI engine capable of addressing enterprise-specific needs.

9.1 Test Scenario

For the testing phase, we defined a practical scenario based on an issue frequently encountered by employees: the retrieval and management of **company documents**. This choice was deliberate, since document-related queries are among the most recurrent in corporate environments and provide a concrete test of the system's reasoning and retrieval capabilities.

The deployed bot was presented as a conversational assistant. Colleagues were invited to formulate their requests in natural language, exactly as they would do with a human colleague, without adapting their queries to rigid command structures. The agent, leveraging its RAG-based reasoning pipeline, was expected to interpret the request, retrieve relevant documentation, and provide structured answers directly in the chat.

The following screenshots illustrate the sequence of interactions: from the user's initial request, through the system's intermediate processing, to the final structured response.

9.2 Results and Observations

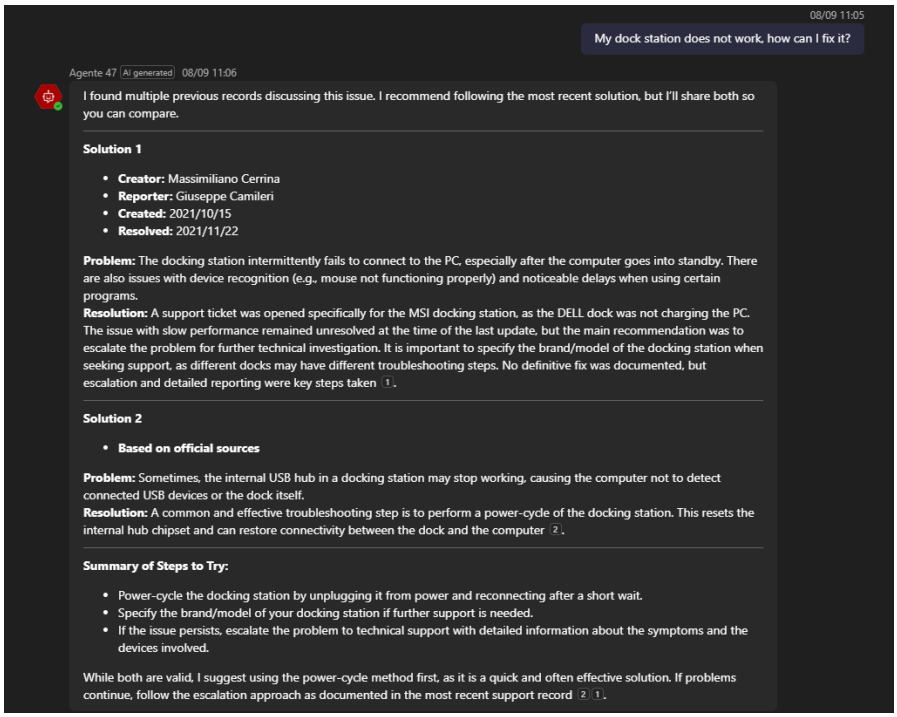

Figure 9.1: Bot interaction – user request about company documents

The internal feedback was broadly positive. Employees highlighted the bot's ability to reduce search times and its usefulness in structuring responses in a clear and accessible way directly within Teams. A key added value was the naturalness of the interaction: users did not need to adapt to a rigid syntax but could rely on everyday language. This aspect is particularly relevant in enterprise adoption, where user acceptance is often tied less to technical sophistication than to the perceived ease of use.

At the same time, the test offered insights into aspects that go beyond functionality. For example, participants pointed out the relevance of response time: even small latencies can affect the perception of the tool's usefulness, especially in fast-paced working contexts. Similarly, the importance of context management emerged: the agent's ability to retain information across turns and provide coherent follow-ups proved critical to sustaining a smooth user experience.

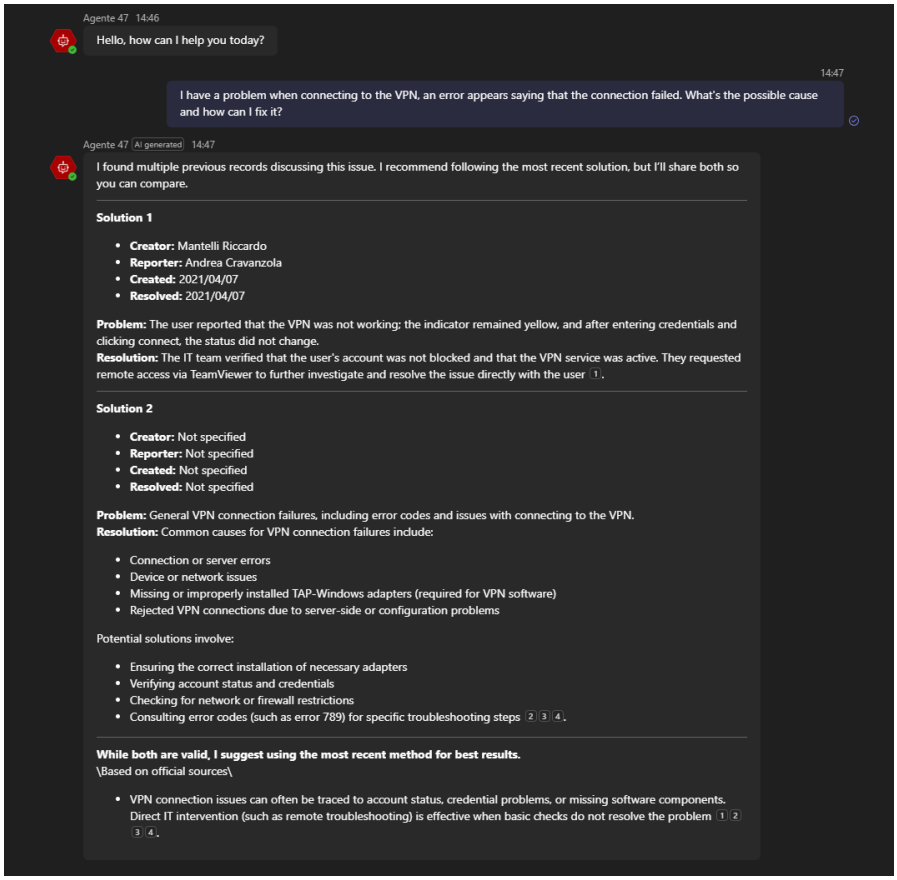

Figure 9.2: Bot processing the request and retrieving information

From a methodological standpoint, the testing activity also showed the importance of structured monitoring. In enterprise environments, collecting logs,

telemetry, and user interaction data is essential not only for debugging but also for ensuring compliance with audit and security requirements. Although in our prototype these aspects were handled at a minimal level, their role becomes decisive when moving from an experimental deployment to production.

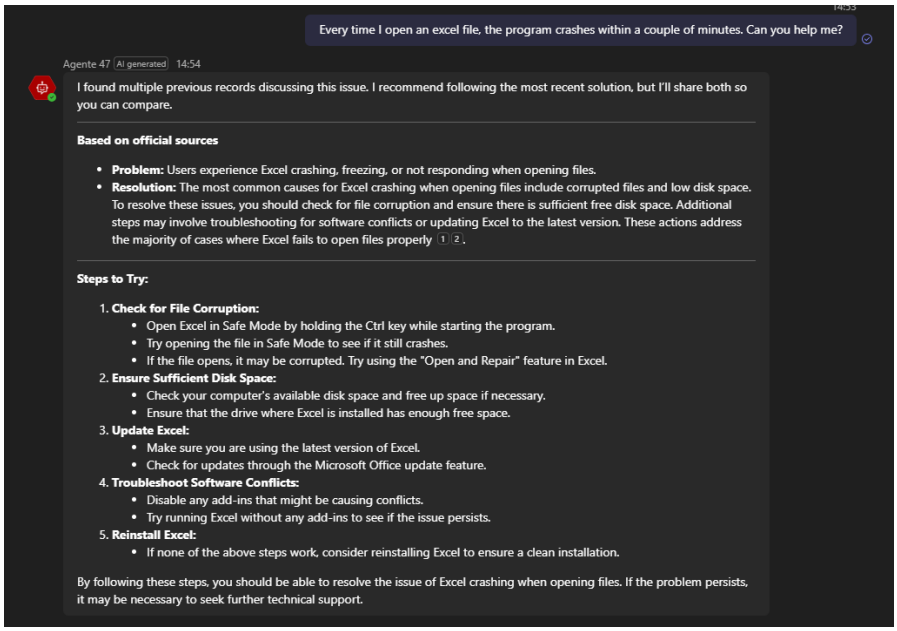

Figure 9.3: Bot providing a structured solution in Teams chat

9.3 Role of Feedback in Enterprise Testing

In enterprise contexts, the introduction of new AI-driven solutions requires not only technical validation but also systematic collection of user feedback. This process involves both **final users**, who act as early adopters and provide impressions of usability and effectiveness, and **quality assurance (QA) staff**, who contribute more structured evaluations with an eye on consistency, compliance, and long-term maintainability.

The participation of colleagues as testers is particularly valuable in this phase. Unlike formal laboratory testing, in which conditions are tightly controlled, in-company trials expose the agent to realistic working dynamics, where expectations, workflows, and habits influence the perception of utility. End users often focus on immediacy: whether the system saves them time, whether the answers are understandable, and whether the interaction feels natural. QA staff, on the other hand, tend to emphasize robustness, reliability, and edge cases that could emerge in broader adoption.

In our case, QA observations provided concrete improvement directions. One remark concerned the management of sensitive information: the agent should not expose personal details, such as the name of the creator or reporter of a ticket, when the requester is a standard user. This type of information, however, could be legitimately included if the query comes from an IT operator, who is expected to require such data for operational purposes.

Another suggested refinement involved the breadth of diagnostic insights. Rather than limiting responses to detailed information about one or two specific tickets, the agent could generate a broader list of potential causes, each accompanied by a brief description. Such an approach would enrich the user's perspective, reducing the risk of bias from too small a sample and providing a more comprehensive basis for troubleshooting. These contributions illustrate how QA feedback acts as a bridge between technical feasibility and organizational adoption. By combining experiential feedback from end users with methodological input from QA staff, the evaluation process supports not only immediate usability but also the long-term alignment of the system with enterprise standards.

9.4 From Development to Production

A further element that emerged during validation relates to the lifecycle management of the solution. As is standard in enterprise IT, the entire system was first deployed in a **development environment**, conceived as a safe playground for iterative refinement and testing. This environment reproduced the architecture described in previous chapters, including the MCP server, the Azure services, and the Teams integration, but was isolated from the production environment to avoid any interference with day-to-day business operations.

Once the development pipeline yielded stable results, the same infrastructure can be replicated in a **production environment**. This migration is far from trivial: while development emphasizes flexibility, experimentation, and rapid iteration, production requires robustness, high availability, and strict governance. In particular, in production the solution must comply with enterprise policies for data retention, identity management, and access control. Moreover, observability mechanisms such as advanced logging, performance dashboards, and automated alerting need to be in place to ensure operational continuity.

In our case, this distinction was crucial. In the development setup, we could run tests with partial datasets, simulate failures, and directly inspect logs for debugging purposes. In production, however, such liberties are not allowed: stability and predictability take precedence over flexibility. Therefore, the adoption of **Infrastructure-as-Code** and pipeline-based deployment strategies becomes essential. By defining the infrastructure declaratively, the same architecture can

be consistently reproduced across environments, minimizing human errors and ensuring reproducibility.

9.5 Infrastructure Limitations

Despite the encouraging results, the validation process also highlighted some structural limitations of the architecture. Our solution relies on two distinct layers:

- An **MCP server**, developed and managed by us, responsible for reasoning and orchestration of tools.
- A **Copilot service**, hosted and managed by Microsoft, over which we have no direct control.

This dual dependency introduces a critical fragility. If the MCP server encounters issues, we are in a position to intervene, diagnose, and fix the problem. However, if the Copilot service suffers a malfunction, the entire conversational flow may collapse without any possibility for us to mitigate the impact. During the test, such a situation occurred: as shown in Figure 9.4, the bot session was abruptly interrupted due to a failure in the Copilot layer.

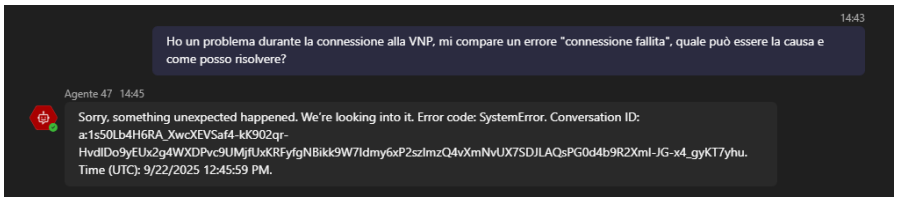

Figure 9.4: Failure in Copilot service causing bot interruption

This observation underscores a broader point about enterprise AI adoption: hybrid architectures that combine self-managed components with third-party managed services inevitably entail trade-offs between control and convenience. While relying on external services can drastically reduce development time and integration effort, it also introduces dependencies that must be managed carefully. Future evolutions of the system should thus include resilience strategies, such as fallback mechanisms, redundancy, or at least monitoring dashboards that can promptly detect and signal failures in external components.

In conclusion, the validation phase confirmed both the potential and the challenges of our approach. The prototype demonstrated its ability to support real business tasks effectively and integrate smoothly into established workflows, but at the same time revealed infrastructural dependencies and lifecycle management requirements that must be addressed in order to ensure long-term sustainability in production settings.

Chapter 10

Conclusion

This thesis has addressed the engineering challenge of designing and implementing an intelligent conversational agent that is both operationally reliable and suitable for enterprise contexts, with a specific focus on supporting the IT department of AROL S.p.A. The research and development process highlighted that integrating Large Language Models (LLMs) into real-world operational environments requires an architecture that moves beyond the "monolithic" generative paradigm, embracing instead a modular, secure, and governable agentic approach.

The primary contribution of this work lies in the conception, implementation, and validation of a framework based on the **Model Context Protocol (MCP)**, which serves as a critical mediation layer between the natural language reasoning capability of an LLM and the controlled execution of operations on external systems. The proposed architecture combines the **Retrieval-Augmented Generation (RAG)** capabilities , with a custom MCP server that exposes a set of tools for interaction with services such as Microsoft Graph (Calendar, Mail, Directory).

The results from the end-user validation phase conducted within the company confirmed the validity of the approach. The agent proved effective in handling real-world use cases, such as ticket management, document retrieval, and meeting scheduling, significantly reducing search times and simplifying otherwise fragmented workflows. The native integration into Microsoft Teams ensured immediate adoption and a fluid user experience, which are fundamental to the success of any tool introduced into a productive environment.

However, the project also revealed significant challenges. The transition from a functioning prototype in a development environment to a production-ready system required meticulous attention to critical non-functional aspects. Security and authentication emerged as a central concern: robust integration with Azure Entra ID OAuth2 flows was indispensable, requiring iterative refinements at the SDK level, first in Python and later in TypeScript, to guarantee both stability and secure communications. Operational reliability represented another crucial issue, given

the hybrid nature of the architecture. Since it depends on both a self-managed component (the MCP server) and third-party managed services (Copilot Studio), the system introduced a single point of fragility: as observed during testing, an outage in Copilot could compromise the entire conversational flow, a vulnerability beyond the control of the development team. Furthermore, questions of governance and control arose, particularly the need to enforce granular policies for context management and data access, ensuring that the agent's operations conformed to the principles of least privilege and information confidentiality.

In conclusion, this work demonstrates that modern LLMs, when encapsulated within a well-designed, standardized agent architecture like the one enabled by MCP, can effectively transition from promising prototypes into reliable components of enterprise IT ecosystems. The presented framework provides a blueprint for building intelligent assistants that not only understand natural language but also act securely, auditably, and in an integrated manner within the complex technological fabric of modern organizations.

Future Works Although the developed system has achieved its primary objectives, the research path in this field remains open and full of promising directions. A first avenue concerns the development of dynamic, role-based privacy and access policies. At present, data access is mainly regulated at the initial authentication stage. Future iterations should integrate more advanced authorization mechanisms at the MCP tool level, enabling differentiated response schemas that adapt to the role of the interlocutor. For example, while personal details such as the name of a ticket's creator should remain hidden from standard users, they may be legitimately disclosed to authorized IT operators, thus ensuring a balance between operational utility and confidentiality.

The resilience of the hybrid architecture also warrants further exploration. To mitigate the strong dependency on external services such as Copilot Studio, future work could introduce redundant LLM, to make sure that there is an agent always available.

Moreover, the presence of better conversational memory and reasoning would represent a significant improvement. While more advanced models exist that could better handle multi-turn interactions and long-term user context, current integration with Copilot Studio shows limitations in maintaining state reliably over time. Addressing these limitations could enable the agent to provide a more coherent experience.

Finally, improving the quality of the files, both in terms of structure and semantic content, would lead to a more effective RAG. This enhancement would facilitate the agent's ability to accurately retrieve and interpret relevant information.

Overall, these directions point to a natural evolution of the presented system, reinforcing its potential to act not only as a conversational interface but also

as a trusted, resilient, and intelligent partner within enterprise IT ecosystems. Importantly, the system remains fully manageable by the company, and many of these improvements could be implemented internally by their own development team.

Appendix A

MCP Typescript Server

In this appendix, we present selected excerpts of the MCP TypeScript server implementation. The code snippets illustrate key components, including server initialization, OAuth authentication, middleware for token propagation, and the definition of MCP tools. These examples aim to provide a concise technical overview while highlighting the architectural decisions and security considerations applied in the development of the server.

```

const {
    AZURE_CLIENT_ID,
    AZURE_CLIENT_SECRET,
    AZURE_TENANT_ID,
    AZURE_REDIRECT_URI,
    MCP_ISSUER_URL,
    PORT = 3000,
} = process.env;

// 1. Create the MCP Server istance
const server = new McpServer({name: "mcp-server", version: "1.0.0"});

```

Here we can see the initialization of the environment variables for Azure authentication and create the MCP server instance. Since this thesis is produced in collaboration with a company, secure handling of credentials and strong authentication/authorization are of fundamental importance.

```

const getWeather = server.tool(
    "get_weather", "A tool to get the weather of a city. Does not need
    ↪  any authentication.",
```

```
    {
        city: z.string().describe("Name of the city to get the weather for"),
    },
    async ({city}) => {
        //console.log(`>>> get_weather called for city: ${city}`);

        const response = await fetch(`https://wttr.in/${encodeURIComponent(city)}?format=3`);
        if (!response.ok) {
            const text = await response.text();
            throw new Error(`Weather API error: ${response.status} - ${text}`);
        }

        const weatherText = await response.text();

        return {
            content: [
                {type: "text", text: `Weather for ${city}: ${weatherText}`}
            ]
        };
    }
);

const createMeeting = server.tool(
    "create_meeting","Create a new Microsoft Teams meeting in the authenticated user's calendar. The user provides a meeting title, a list of attendees (emails), the meeting date, start time, and end time. The user will always provide times in local Italian time (CET/UTC+1 or CEST/UTC+2 depending on the date). The LLM is responsible for converting these times into ISO 8601 UTC format before calling this tool. The tool schedules the event and returns the meeting ID and Teams join link.",
    {
        subject: z.string().default("Riunione con il team").describe("Meeting title"),
        attendees: z.array(z.string().email()).default([]).describe("List of e-mails to invite"),
        date: z.string().describe("Meeting date in YYYY-MM-DD format"),
        startTime: z.string().describe("Start time in HH:mm (24h) format, local time"),
```

```
        endTime: z.string().describe("End time in HH:mm (24h) format, local time"),
    },
    async (
        { subject, attendees, date, startTime, endTime }:
        { subject: string; attendees: string[]; date: string; startTime: string; endTime: string },
        context: any
    ) => {
        // same mechanism pf send_mail
        const msToken =
            (context?.authInfo?.token as string | undefined) ??
            ((server as any)?._authProviderLastToken as string | undefined) ??
            "";

        if (!msToken || msToken.split(".").length !== 3) {
            return {
                content: [{ type: "text", text: " Token Microsoft mancante o non valido in create_meeting." }],
            };
        }

        const start = new Date(`${date}T${startTime}:00`);
        const end = new Date(`${date}T${endTime}:00`);
        const payload = {
            subject,
            start: { dateTime: start.toISOString().slice(0, 19), timeZone: "UTC" },
            end: { dateTime: end.toISOString().slice(0, 19), timeZone: "UTC" },
            location: { displayName: "Online" },
            attendees: attendees.map((email) => ({
                emailAddress: { address: email, name: email.split("@")[0] },
                type: "required",
            })),
            isOnlineMeeting: true,
            onlineMeetingProvider: "teamsForBusiness",
        };
        const res = await fetch("https://graph.microsoft.com/v1.0/me/events", {
            method: "POST",
            headers: {
                Authorization: `Bearer ${msToken}`,
```

```
                "Content-Type": "application/json",
                Accept: "application/json",
            },
            body: JSON.stringify(payload),
        });
        const bodyText = await res.text();
        if (!res.ok) {
            throw new Error(`Graph error ${res.status} - ${bodyText}`);
        }
        const data = JSON.parse(bodyText);
        return {
            content: [
                {
                    type: "text",
                    text: ` Evento creato\nID: ${data.id}\nTeams link:
                    ↪  ${data.onlineMeeting?.joinUrl ?? "N/A"}`
                },
            ],
        };
    }
);

```

Here we show two example MCP tools: a simple, unauthenticated weather lookup and a more complex Teams meeting creator that integrates with Microsoft Graph and requires valid user authentication. The MCP architecture grants the industrial partner full freedom to select, integrate, and extend external tools and APIs while retaining complete operational control.

```

// 3. Configure the OAuth Provider as a "proxy" to Azure AD
const authority =
↪  `https://login.microsoftonline.com/${AZURE_TENANT_ID}`;
const authProvider = new ProxyOAuthServerProvider({
    endpoints: {
        // Azure AD OAuth2.0 endpoints (v2.0)
        authorizationUrl: `${authority}/oauth2/v2.0/authorize` +
            `?scope=openid profile offline_access` +

            ↪  `+Mail.ReadWrite+Mail.Send+Calendars.ReadWrite+Calendars.ReadWrite.Sh
        tokenUrl: `${authority}/oauth2/v2.0/token`,
        revocationUrl: `${authority}/oauth2/v2.0/logout`,
    },
    // This method is called by the middleware to validate the incoming
    ↪  Bearer token
```

```
    verifyAccessToken: async (token: string) => {
        // Decode only for logging purposes
        const [, p] = token.split('.');
        const claims = JSON.parse(Buffer.from(p,
        ↪  'base64url').toString());
        //console.log('>>> claims', claims);

        // Always return OK - only scope validation is handled here
        return {
            token,
            clientId: claims.aud,
            scopes: claims.scp?.split(' ') ?? [],
            extra: claims
        };
    },
    // The MCP client (Copilot) is pre-registered: associate the
    ↪  redirect URI
    getClient: async (client_id: string) => ({
        client_id,
        redirect_uris: [AZURE_REDIRECT_URI!],
    }),
});
```

Here the OAuth provider is configured to act as a proxy to Azure AD: authorization, token and revocation endpoints are declared and a verifyAccessToken routine decodes JWT claims (used here for logging and to extract clientId and scopes). The getClient method binds pre-registered redirect URIs for MCP clients. While this implementation relies on Azure AD for authentication, the MCP architecture is fully provider-agnostic: organizations can integrate any OAuth 2.0 compliant identity service

```

// 4. Middleware to protect /mcp with Bearer token and "user" scope
const tokenMiddleware = (req: Request, res: Response, next: Function) =>
↪  {
    const auth = req.headers.authorization;
    if (!auth?.startsWith("Bearer ")) return res.status(401).end();

    const raw = auth.slice(7);
    const token = raw.replace(/^["']|["']$/g, '').replace(/\s+/g, '');
    ↪  // ← cleaned
    const [, payload] = token.split(".");
    let claims;
    try {
```

```
        claims = JSON.parse(Buffer.from(payload,
        ↪  "base64url").toString());
    } catch {
        return res.status(401).end();
    }
    const scopes = (claims.scp as string)?.split(" ") ?? [];
    if (!scopes.includes("User.Read")) return res.status(403).end();
    // Store the context (authInfo) in the req object
    (req as any).authInfo = {token, claims, scopes};
    next();
};
```

This middleware protects the /mcp endpoint by verifying the presence of a valid Bearer token and ensuring that it includes the required User.Read scope. It decodes the JWT payload, validates the structure, and attaches authentication information to the request context. Such middleware ensures that only properly authenticated users can access critical routes, enforcing basic authorization and context propagation for subsequent operations.

```

// 5. Connects the server and defines the /mcp endpoint
const transport = new StreamableHTTPServerTransport({sessionIdGenerator:
↪  undefined});
server.connect(transport).then(() => {
});
app.use("/mcp", (req, res, next) => {
    try {

        // If an Authorization Bearer token is present, extract it,
        ↪  decode the claims, and propagate it
        const authHeader = (req.headers.authorization ?? "") as string;
        if (authHeader.startsWith("Bearer ")) {
            const token = authHeader.slice(7).replace(/^["']|["']$/g,
            ↪  '').trim();

            // Save it in req for later use
            (req as any).authInfo = (req as any).authInfo || {};
            (req as any).authInfo.token = token;

            // Also store it in the server so that the SDK/transport can
            ↪  access it
            (server as any)._authProviderLastToken = token;

            // Try to decode the payload for logging (do not verify the
            ↪  signature here)
```

```
            try {
                const payload = token.split(".")[1] ?? "";
                const claims = JSON.parse(Buffer.from(payload,
                ↪  "base64url").toString() || "{}");
            } catch (e) {
                console.warn(">>> Failed to decode token payload:", e);
            }

        } else {
            console.log(">>> No Bearer token found in header");
        }
    } catch (err) {
        console.warn(">>> Error in /mcp logging middleware:", err);
        // Do not block the request: continue anyway
    }
    next();
});

```

This section connects the MCP server to the HTTP transport layer and defines the /mcp endpoint. It also implements middleware logic to propagate the Bearer token from incoming requests to the server context, allowing subsequent tools or SDK components to access the authentication information. Although this step does not perform cryptographic verification, it ensures token continuity across requests, which is crucial for maintaining user context and enabling authenticated interactions between the MCP client and server.

After this we defines the main /mcp endpoint that connects incoming HTTP requests to the MCP server. We introduce a configurable flag to allow the system to run either in authenticated or open mode. When authentication is enabled, requests are validated through token-based middleware before being processed, while discovery calls remain accessible without authentication. This flexible design ensures a clear separation between development and production environments.