# Intelligent Access Governance for New Joiners: A Spring AI and Azure OpenAI Solution with MCP Architecture

## Executive Summary

This report details a robust, Spring AI-driven solution designed to intelligently recommend group memberships for new employees within an organization. By leveraging Azure OpenAI and a Model Context Protocol (MCP) server/client architecture, the proposed system addresses the common challenges of manual access provisioning. The solution automates suggestions based on existing employee hierarchies and group data, aiming to streamline onboarding processes. Key architectural considerations include a modular design, secure data exposure through MCP tools, sophisticated prompt engineering for accurate recommendations, and a comprehensive mocking strategy to facilitate efficient hackathon development and testing. Adopting this approach can significantly streamline new employee onboarding, enhance security by adhering to the principle of least privilege, and improve overall operational efficiency.

## Understanding the Hackathon Project: AI for Access Governance

This section delves into the specific problem targeted by the hackathon project, detailing the current inefficiencies in access governance and illustrating how an AI-powered solution can deliver substantial value.

### Problem Statement: Streamlining New Joiner Onboarding

Manual access provisioning for new employees often presents significant challenges within organizations. This traditional approach is typically time-consuming, prone to human error, and can inadvertently lead to over-privileging employees with more access than necessary, or conversely, cause delays in productivity due to insufficient access. The core objective of this hackathon project is to automate the suggestion of appropriate group memberships for new joiners, thereby mitigating these issues.

The proposed AI-driven solution aims to transform the process of verifying and authenticating user identities and assigning designated roles during the onboarding phase.1 By dynamically assigning roles based on job functions, AI can make the onboarding process smoother and more efficient.2 This transition from manual to AI-driven access provisioning signifies a fundamental shift from reactive security—where access is granted only upon request—to a more proactive governance model. In this proactive model, the system anticipates and suggests necessary access from the outset, which inherently supports and improves compliance with the "least privilege" principle. This proactive stance significantly reduces the potential for human error and enhances the overall security posture of the organization.

### Core Use Case: Intelligent Group Membership Suggestions

The primary application of this system involves an AI model analyzing a new joiner's departmental affiliation, line manager, and potentially other relevant attributes. Based on this analysis, the system suggests appropriate security or distribution group memberships. This functionality represents a specific implementation of a recommendation system, tailored for access governance.

The efficacy of this recommendation system relies not merely on the Large Language Model's (LLM) general intelligence, but critically on its ability to effectively interpret and reason over structured, hierarchical organizational data.3 While prompts for recommendation systems are well-documented 5, the nuance here is the requirement for the LLM to understand and utilize the intricate relationships between employees, departments, and existing group memberships, rather than performing a simple data lookup. AI's role-mining capabilities, which analyze identity interaction patterns, are instrumental in enforcing the principle of least privilege in this context.2 This necessitates meticulous prompt engineering to ensure the LLM can accurately interpret and leverage the provided hierarchical context. The intelligence of the suggestion stems from the LLM's advanced reasoning capabilities applied to well-structured input, guided by precise prompt instructions, moving beyond basic retrieval to a more inferential and analytical task.

### Data Model: Representing Employee Hierarchy and Group Memberships

The foundational input data for this system will comprise employee records, including attributes such as employeeId, name, department, lineManagerId, and a mapping of groupId to groupName and members. A significant challenge lies in effectively presenting this inherently hierarchical and relational data to the LLM.

Various approaches exist for representing hierarchical structures for LLMs, such as using folder-like organizations, URL structures, or even mimicking document headings (H1/H2 tags).6 Advanced methods like Tree-of-Thoughts or Graph-of-Thoughts can also guide complex reasoning over such structures.9 For structured output, JSON is highly favored.10 The specific format chosen for the input data to the LLM—whether a deeply nested JSON representing the full organizational chart or a flattened list of "employee X reports to Y in department Z" combined with a separate list of "group A has members B, C, D"—will directly influence the prompt's complexity, the LLM's ability to accurately parse and reason, and token usage. A poorly selected input structure, despite well-crafted prompt instructions, could lead to misinterpretations or inaccuracies by the LLM. This underscores the necessity for careful data pre-processing and experimentation with different input formats to identify the most effective and token-efficient representation for the specific LLM and the access recommendation task.

## Spring AI and Azure OpenAI: Foundational Setup

This section provides a guide for setting up the development environment and establishing connectivity to Azure OpenAI using Spring AI.

### Prerequisites and Initial Spring Boot Project Setup

To begin, developers should ensure they have Java Development Kit (JDK) 17 or later installed, with JDK 21 being recommended, along with a build automation tool like Maven or Gradle, and an integrated development environment (IDE) such as IntelliJ IDEA.16 The initial Spring Boot project can be quickly bootstrapped using Spring Initializr (start.spring.io).18

For the project dependencies, it is essential to include Spring Web (or Spring Reactive Web if a reactive programming model is preferred for handling streaming responses) and the Spring AI OpenAI Starter.18 The choice between

spring-boot-starter-web (which uses the Servlet API) and spring-boot-starter-webflux (which uses the Reactive API) carries implications for how streaming responses are managed. While WebMVC is simpler for basic REST endpoints, WebFlux is often the preferred choice for LLM streaming responses due to its non-blocking nature, which aligns well with the incremental, real-time generation of AI outputs.17 This architectural decision impacts not only the chosen dependency but also the overall programming model for handling AI responses in interactive applications.

### Provisioning and Configuring Azure OpenAI Resources

A critical step involves provisioning an Azure OpenAI resource. This can be accomplished through the Azure CLI or the Azure portal.20 During this process, a suitable language model, such as

gpt-4o-mini, should be deployed. gpt-4o-mini is often recommended for its cost-efficiency and regional availability, for example, in eastus2.3 It is crucial to meticulously record the generated endpoint, API key, and deployment name. These credentials are then configured in the application's

application.properties or application.yml file, typically under spring.ai.openai.api-key and spring.ai.openai.chat.options.model.14

The selection of the Azure region and the specific LLM model, such as gpt-4o-mini versus gpt-4o, represents a critical decision driven by both model availability and cost optimization. While gpt-4o-mini is cost-effective, it has demonstrated lower performance in complex prompt generation tasks compared to larger models like gpt-4o, with a reported 15% drop in quality for such tasks.3 For the access recommendation use case, which involves reasoning over hierarchical data to generate nuanced suggestions, this performance difference could impact the quality of the recommendations. This highlights a fundamental trade-off that the hackathon team must consider: prioritizing cost-efficiency and rapid deployment versus achieving the highest possible quality in AI-driven recommendations.

### Integrating Spring AI with Azure OpenAI in the Client Application

Spring AI simplifies the integration with Azure OpenAI by allowing developers to use dependency injection to autowire ChatClient or ChatClient.Builder instances.21 A basic

ChatClient call can then be performed to verify successful connectivity and interaction with the deployed LLM.18

Spring AI's ChatClient provides a portable API that abstracts away the specifics of various AI providers, including Azure OpenAI, OpenAI, Ollama, and Google, among others.18 This portability offers a significant architectural advantage. For a hackathon, this means the development team can seamlessly switch between a mocked environment, a local LLM (like Ollama for offline development), and the actual Azure OpenAI service without requiring substantial code modifications. This capability is invaluable for rapid prototyping and ensures future flexibility, significantly de-risking the project by decoupling the application logic from a specific LLM vendor.

## Implementing the Model Context Protocol (MCP) Architecture

This section is central to establishing the communication framework for the access governance solution, detailing the construction of the MCP server and client to facilitate secure and structured interactions between the access governance data and the AI model.

### Why MCP?

The Model Context Protocol (MCP) serves as a crucial bridge, enabling AI models to securely access external tools and data sources.16 This protocol facilitates a unified abstraction layer for integrating diverse AI models and services into Spring applications, promoting an extensible architecture through its function calling capabilities.16 MCP effectively separates business logic, which resides in the server, from the AI orchestration, handled by the client. This separation of concerns is vital for building robust and maintainable AI applications. MCP transforms traditional Spring Boot services into "AI-callable tools".16 This paradigm shift allows domain experts, who possess deep understanding of the access governance data, to expose their business logic as accessible tools without requiring extensive AI expertise. Simultaneously, AI developers can consume these tools without needing to delve into the intricacies of underlying data persistence or complex business rules. This fosters a more collaborative and efficient development workflow, which is particularly advantageous in a time-constrained hackathon setting.

### Designing the MCP Server: Exposing Access Governance Data as Tools

To implement the MCP Server, a separate Spring Boot application is created. Within this application, domain models, such as Employee, Department, Group, and Membership, are defined. These models can be backed by Spring Data JPA repositories for persistent storage, or, for the rapid development pace of a hackathon, simple in-memory data structures can be utilized.16

The core of the MCP Server functionality lies in implementing service methods that encapsulate the business logic, such as getEmployeeDetails(employeeId), getDirectReports(employeeId), getDepartmentGroups(departmentId), and getGroupMemberships(groupId). Crucially, these methods are annotated with @Tool(description = "...").16 The

description attribute is paramount, as it serves as the LLM's primary interface to understand the semantic meaning and purpose of the tool, enabling the AI to decide when and how to invoke it.16 A clear, concise, and accurate description is essential for effective function calling and for minimizing LLM "hallucinations" regarding tool usage. The

description acts as a form of "meta-prompting," where the application itself provides the LLM with a structured understanding of its capabilities.

Configuration for the MCP Server involves setting spring.ai.mcp.server.enabled=true and specifying a distinct server.port (e.g., 8081) in application.properties or application.yml.16 Finally,

ToolCallback beans are registered in a configuration class, which converts the tool classes into instances that the MCP Server can manage and expose.16

### Building the MCP Client: Consuming Tools and Orchestrating AI Interactions

The MCP Client is implemented as another distinct Spring Boot application. Its pom.xml or build.gradle.kts file must include the spring-ai-openai-spring-boot-starter (for OpenAI integration) and spring-ai-starter-mcp-client dependencies.16 The client's

application.properties or application.yml is configured with spring.ai.mcp.client.server.url to point to the address of the running MCP Server.16

Within the MCP Client application, ChatClient and McpClient instances are injected into an AI service. The mcpClient.getAvailableTools() method is then used to dynamically fetch the tools exposed by the MCP Server.16 This dynamic discovery of tools enables a flexible and evolvable architecture. The MCP Client does not require recompilation or redeployment if new tools are added or existing tools are modified on the MCP Server, provided their interfaces remain compatible. This significantly accelerates iteration and allows for independent deployment of different components within the hackathon project. Once the tools are available, the AI service constructs prompts that can strategically leverage these tools to generate accurate access recommendations.

### Configuring Secure Communication between MCP Server and Client

For a hackathon environment, simple HTTP communication between the MCP Server and Client might be sufficient for rapid prototyping. However, for production deployments, robust security mechanisms are imperative, especially given the sensitive nature of access governance data. Options for secure service-to-service authentication include managed identities (as supported by Azure App Service) 20, OAuth2, or API keys.

The explicit consideration of secure communication in the context of access governance is crucial.2 Exposing internal organizational data, even through well-defined tools, necessitates strong authentication and authorization protocols. This highlights the importance of designing security into the architecture from the outset, rather than treating it as an afterthought, particularly when dealing with sensitive HR information. Key security practices include validating all tool inputs, implementing rate limiting to prevent abuse, and ensuring that sensitive data is never exposed in raw form within tool responses.16

### Table 1: Key Spring AI Dependencies for MCP Server and Client

The following table outlines the essential Spring AI dependencies required for setting up both the MCP Server and Client applications, providing a clear reference for project initialization.

| Component | Dependency GroupId | Dependency ArtifactId | Notes |
| --- | --- | --- | --- |
| **MCP Server** | org.springframework.boot | spring-boot-starter-web (or -webflux) | Provides web capabilities. webflux for reactive/streaming. |
|  | org.springframework.ai | spring-ai-starter-mcp-server-webmvc (or -webflux) | Enables MCP Server functionality. |
|  | org.springframework.boot | spring-boot-starter-data-jpa | Optional, for database interaction if needed. |
|  | org.springframework.ai | spring-ai-bom | Recommended for consistent Spring AI dependency versions. |
| **MCP Client** | org.springframework.boot | spring-boot-starter-web | Provides web capabilities for client API. |
|  | org.springframework.ai | spring-ai-openai-spring-boot-starter | Integrates with OpenAI (including Azure OpenAI). |
|  | org.springframework.ai | spring-ai-starter-mcp-client | Enables MCP Client functionality. |
|  | org.springframework.ai | spring-ai-bom | Recommended for consistent Spring AI dependency versions. |

## Crafting Intelligent Prompts for Access Recommendations

This section focuses on the art and science of prompt engineering, specifically tailored to the access governance use case, ensuring the LLM receives structured input and provides actionable, structured output.

### Strategies for Structuring Hierarchical Employee Data for LLMs

To enable the LLM to provide accurate access recommendations, the hierarchical employee data and group memberships must be presented in an interpretable format. This includes representing employee and line manager hierarchy (e.g., Employee ID, Name, Department, Line Manager ID, Line Manager Name) and group memberships (e.g., Group Name, Description, Members).

Various options exist for structuring this input data within the prompt. A flattened JSON structure, where each employee record directly includes department and line manager details, and group memberships are provided as a separate list of objects, can be effective. Alternatively, delimited text, utilizing clear separators like ---EMPLOYEE--- or ---GROUP---, can delineate different data sections within the prompt. System messages can also be leveraged to provide the overall schema or context, with specific new joiner data and relevant existing data fed in user messages.13

The concept of "Hierarchy-of-Thoughts" (HoT) suggests that LLMs perform more effectively with complex reasoning tasks when guided through a structured thought process.9 For hierarchical data, this implies that simply providing raw data might be less effective than explicitly instructing the LLM to "think step-by-step" about the organizational structure. For instance, a prompt could guide the LLM to: "First, identify the new joiner's department. Second, find other employees in that department and their roles. Third, identify common groups for those roles." This approach, a form of zero-shot Chain-of-Thought (CoT) prompting 28, applied to data interpretation, can significantly improve the accuracy of recommendations by ensuring the LLM systematically processes the hierarchical context. This moves beyond mere data input to explicit reasoning guidance within the prompt.

### Developing Effective Prompts for Group Membership Suggestions

Crafting effective prompts involves defining clear roles and providing comprehensive context for the LLM. The **System Role** should instruct the AI to act as an "Access Governance Advisor" or "IT Onboarding Specialist," emphasizing accuracy, adherence to security best practices (such as least privilege), and a clear output format (e.g., JSON).13 The

**User Role** provides the new joiner's specific details (name, department, line manager) and the relevant contextual data, such as a JSON representation of existing employees in the department, their current groups, and the line manager's group memberships.

The **Tool/Function Role** is implicitly handled as the LLM, guided by the prompt, will strategically utilize the MCP tools to fetch specific data (e.g., getEmployeesInDepartment, getGroupMembers(groupId)). Furthermore, incorporating **Few-Shot Learning** by providing one or two examples of a new joiner's profile alongside the ideal group memberships and the reasoning behind them can significantly enhance the LLM's ability to generate desired outputs.3 This demonstrates the expected output format and the logical steps for arriving at the recommendation.

The application of "Role-based prompting" 28 is more than a stylistic choice; it is a powerful mechanism to align the LLM's behavior with the specific security and compliance objectives of access governance. Instructing the LLM to act as an "Access Governance Advisor" implicitly biases its responses towards principles like "least privilege" and "separation of duties," which are critical for this use case. This role-based context can subtly but effectively steer the LLM away from over-provisioning access and towards recommending only essential groups, thereby reinforcing security principles without requiring explicit, verbose instructions in every prompt. This represents a form of implicit policy enforcement through prompt design.

### Leveraging Spring AI's Structured Output for Actionable Recommendations

To ensure the LLM's recommendations are directly actionable by the application, it is crucial to instruct the model to produce output in a specific JSON format, such as a list of group names accompanied by a brief justification for each. While asking an LLM to produce valid JSON can sometimes be unreliable and lead to "hallucinations" or malformed output 12, Spring AI provides robust mechanisms to mitigate this.

Spring AI's BeanOutputConverter, MapOutputConverter, or ListOutputConverter can be utilized to parse the LLM's JSON response directly into Java objects (POJOs or Records).14 To further increase the reliability of JSON output, especially with OpenAI models, it is recommended to enable JSON mode in

application.properties by setting spring.ai.openai.chat.options.responseFormat.type=json\_object.14

These structured output converters, combined with model-specific JSON modes, provide a robust mechanism to transform free-form AI text into actionable, programmatic data.10 This capability is crucial for integrating LLM recommendations directly into automated workflows, moving beyond mere conversational AI. The converters do not merely parse; they enforce the expected structure, allowing the application to confidently consume the LLM's recommendations as programmatic objects. This enables further automation, such as automatically generating a request ticket for the suggested groups, rather than requiring human interpretation of free-form text, making the AI's output truly actionable.

### Table 2: Example Prompt Structure for Group Membership Recommendation

The following table illustrates a comprehensive prompt structure designed to guide the LLM in generating accurate and actionable group membership recommendations for a new joiner.

| Prompt Component | Description | Example Content / Role |
| --- | --- | --- |
| **System Message** | Defines the AI's persona, overall instructions, and output format constraints. | "You are an Access Governance Advisor. Your primary goal is to recommend the minimum necessary group memberships for new employees based on their role, department, and reporting structure, adhering strictly to the principle of least privilege. Provide your recommendations in JSON format only, without additional conversational text." |
| **User Message (New Joiner Info)** | Provides the specific details of the new employee for whom recommendations are needed. | "New Employee: {new\_joiner\_name}, Department: {new\_joiner\_department}, Line Manager ID: {new\_joiner\_manager\_id}, Line Manager Name: {new\_joiner\_manager\_name}" |
| **User Message (Contextual Data)** | Supplies relevant organizational context, typically fetched via MCP tools. | "Existing Department Employees and their Groups (JSON/Delimited): {department\_employees\_data}. Line Manager's Group Memberships (JSON/Delimited): {manager\_groups\_data}. Available Groups and Descriptions (JSON/Delimited): {all\_groups\_data}" |
| **Tool Calls (Implicit)** | Represents the LLM's internal calls to MCP tools to retrieve the contextual data. | (LLM internally calls getDepartmentEmployees({new\_joiner\_department}), getGroupMemberships({new\_joiner\_manager\_id}), etc., based on tool descriptions.) |
| **Output Format (JSON Schema)** | Specifies the exact JSON structure expected for the recommendation. | json<br/>[<br/> {<br/> "groupName": "string",<br/> "justification": "string"<br/> }<br/>]<br/> |
| **Few-Shot Example (Optional)** | A concrete example of input and the desired output, demonstrating the expected reasoning. | **Input:** New Employee: Jane Doe, Department: Marketing, Line Manager: John Smith. Existing Marketing Employees:},...]. John Smith's Groups:.**Output:** json<br/><br/> |