Comprehensive Guide to Building an Enterprise RAG System

This guide provides a detailed overview of the Retrieval-Augmented Generation (RAG) system architecture, key concepts, essential Python skills, and an implementation roadmap based on the provided diagrams. The goal is to equip you with the knowledge necessary to successfully build and deploy a robust enterprise-grade RAG solution.

Part 1: Understanding the RAG Architecture and Concepts

1.1 Document Ingestion & Preprocessing Pipeline

The first stage of any RAG system involves efficiently ingesting and preparing documents for retrieval. The provided `RAG(3).png` diagram outlines a clear process for this, focusing on transforming raw data into a structured format suitable for embedding and storage in a vector database.

1.1.1 Scheduled Document Crawler

Concept: The foundation of the ingestion pipeline is a document crawler responsible for discovering and downloading documents from specified sources. In an enterprise context, this often involves internal document management systems.

Feature: The diagram specifies a "Scheduled Document Crawler" that "Downloads docs from SharePoint". This highlights several important aspects:

- **SharePoint Integration:** A common requirement in enterprise environments, necessitating robust connectors to Microsoft SharePoint APIs.
- **Scheduled Execution:** Implies the need for a scheduling mechanism (e.g., cron jobs, task schedulers) to periodically check for new or updated documents, ensuring the RAG system's knowledge base remains current.
- **Incremental Updates:** An efficient crawler should ideally support incremental updates, processing only new or modified documents to conserve resources.

1.1.2 Preprocessing Pipeline

Concept: Raw documents often contain various formats, embedded content, and security features that need to be handled before they can be effectively processed for RAG. The preprocessing pipeline addresses these challenges.

Features: The diagram lists the following key preprocessing steps:

- **OCR for Images:** Optical Character Recognition (OCR) is crucial for extracting text from image-based documents (e.g., scanned PDFs, images embedded in other documents). This ensures that content within images is also retrievable.
- **Password Removal (if authorized):** Enterprise documents may be password-protected. This step implies the need for a secure and authorized mechanism to decrypt or remove passwords, allowing access to the document content. This requires careful consideration of security policies and access controls.
- **Format Parsing (PDF, DOCX, XLSX):** Documents come in various formats. The pipeline must be capable of parsing diverse file types to extract their textual content. This includes:
  - **PDF:** Extracting text, handling layouts, and potentially tables from Portable Document Format files.
  - **DOCX:** Parsing Microsoft Word documents, including text, headings, and potentially embedded objects.
  - **XLSX:** Extracting data from Microsoft Excel spreadsheets, often requiring intelligent handling of rows, columns, and cell relationships.

1.1.3 Chunking + Metadata Tagging

Concept: Large documents are typically broken down into smaller, manageable units called

chunks" or "passages". This is essential for efficient retrieval, as searching and embedding entire documents is computationally expensive and can dilute the relevance of retrieved information.

Features:

- **Semantic/Fixed Chunking:**
  - **Fixed Chunking:** Breaking documents into chunks of a predetermined size (e.g., 256 or 512 tokens) with a fixed overlap. This is simple to implement but may break semantic coherence.
  - **Semantic Chunking:** A more advanced approach that aims to keep semantically related sentences or paragraphs together. This can involve techniques like sentence-transformer-based chunking, where sentences are grouped based on their semantic similarity, or using document structure (headings, paragraphs) to define chunks. The diagram's mention of "Semantic/Fixed chunking" suggests flexibility and the importance of choosing the right strategy based on document type and retrieval needs.
- **Metadata Tagging:** Adding relevant metadata to each chunk is critical for filtering and improving retrieval accuracy. The diagram specifies:
  - **Site:** The SharePoint site from which the document originated. This is vital for implementing access controls and filtering based on user permissions.
  - **Path:** The file path or URL of the original document, useful for tracing back to the source.
  - **ACL Info:** Access Control List (ACL) information. This is paramount for ensuring that only authorized users can retrieve information from specific documents or sites, directly addressing the security requirements mentioned in the additional context.

1.1.4 Embedding Generation (GPT)

Concept: Embeddings are numerical representations of text that capture its semantic meaning. These embeddings are crucial for enabling semantic search, where the system can find chunks of text that are conceptually similar to a user's query, even if they don't share exact keywords.

Feature: The diagram specifies "Embedding Generation (GPT)". This implies the use of a pre-trained language model, likely from the GPT series (e.g., `text-embedding-ada-002` from OpenAI, or similar models from Azure OpenAI), to convert text chunks into high-dimensional vectors. The quality of these embeddings directly impacts the relevance of retrieved results.

1.1.5 Store Embeddings in Vector DB (AU only)

Concept: A vector database is a specialized database optimized for storing and querying high-dimensional vectors (embeddings). It allows for efficient similarity search, quickly finding the most relevant chunks to a given query embedding.

Feature: The requirement for a "Vector DB (AU only)" is a critical constraint, emphasizing data residency and security. The additional context further clarifies this, mentioning "Azure Cosmos or PineCone?" and the need for the vector DB to be on an Australian server presence. This points to the need for a vector database solution that can be deployed within Azure's Australian regions and meets stringent security standards.

1.2 Retrieval and Generation Pipeline

The `Pipeline(1).jpg` diagram illustrates the user-facing part of the RAG system, focusing on how user queries are processed, relevant information is retrieved, and a final answer is generated.

1.2.1 User Interface (Microsoft Teams / Web Chat UI)

Concept: The user interface is the primary point of interaction for end-users, allowing them to submit queries and receive answers. In an enterprise setting, integration with existing communication platforms is often preferred.

Feature: The diagram specifies "Microsoft Teams / Web Chat UI (User Interface - Secured via Azure Entra ID)". This indicates:

- **Microsoft Teams Integration:** A common enterprise communication platform, suggesting the development of a Teams bot or application for seamless user interaction.
- **Web Chat UI:** A standard web-based chat interface, providing flexibility for access outside of Teams.
- **Azure Entra ID (formerly Azure Active Directory) Security:** Crucial for authentication and authorization, ensuring that only authenticated users can access the RAG system and that their permissions are respected.

1.2.2 User Authorization via Entra ID & SharePoint Access Filter

Concept: Security and access control are paramount in an enterprise RAG system, especially when dealing with sensitive documents. The system must ensure that users can only access information they are authorized to view.

Features:

- **User Authorization via Entra ID:** This step maps the user's identity to their access permissions within SharePoint. It's the gateway for enforcing document-level security.
- **SharePoint Access Filter:** This filter ensures that the RAG system "only fetches documents from sites user has access to". This is a critical security measure, preventing unauthorized retrieval of information. The additional context explicitly highlights this: "Sharepoint is secured at user level to sharepoint sites as some sharepoint sites hold confidential data that should not be shared with the rest of the staff team. This security aspect needs to carry through to the Sharepoint RAG with access restricted to sites where the user is not authorised for that site."

1.2.3 Vector DB Lookup

Concept: Once a user's query is received, it needs to be converted into an embedding, and then this embedding is used to search the vector database for semantically similar document chunks.

Feature: The diagram mentions "Azure AI Search / Pinecone / Cosmos DB - AU-hosted only)" for "Retrieve top-k relevant chunks". This provides options for the vector database and retrieval mechanism:

- **Azure AI Search:** A comprehensive search service that can be used for vector search, keyword search, and hybrid search. It's a strong candidate given the Azure ecosystem.
- **Pinecone:** A popular specialized vector database known for its performance and scalability.
- **Cosmos DB:** Azure's globally distributed, multi-model database service. While not a native vector database, it can be used to store and query vectors, especially with recent vector search capabilities.
- **AU-hosted only:** Reiteration of the data residency requirement.

- **Retrieve top-k relevant chunks:** The system retrieves the 'k' most similar chunks to the user's query. The value of 'k' is a tunable parameter that balances relevance and the amount of context provided to the LLM.

1.2.4 Prompt Composer

Concept: After retrieving relevant chunks, these chunks need to be integrated into a prompt that is then sent to the Large Language Model (LLM). The prompt composer is responsible for constructing this effective prompt.

Feature: The prompt composer's role is to "Inject retrieved context into GPT prompt". This involves:

- **Contextualization:** Placing the retrieved chunks strategically within the prompt to provide the LLM with the necessary information to answer the user's query.
- **Instruction Tuning:** Adding clear instructions to the LLM on how to use the provided context to generate an answer, including constraints (e.g.,

not to hallucinate, to answer only from provided context, etc.).

1.2.5 Azure OpenAI (GPT-4 API - AU)

Concept: The Large Language Model (LLM) is the core component responsible for generating human-like text responses. In a RAG system, the LLM receives the user's query augmented with retrieved context and generates an informed answer.

Feature: The diagram specifies "Azure OpenAI (GPT-4 API - AU)". This is a crucial detail:

- **Azure OpenAI:** Leveraging Microsoft Azure's managed service for OpenAI models ensures enterprise-grade security, compliance, and scalability. It also aligns with the existing Azure tenant mentioned in the additional context.
- **GPT-4 API:** Utilizing a powerful and advanced LLM like GPT-4 provides high-quality, coherent, and contextually relevant responses.
- **AU Region:** Reinforces the data residency requirement, ensuring that the LLM inference also occurs within Australia.

1.2.6 Final Answer to User

Concept: The ultimate output of the RAG system is a concise and accurate answer delivered to the user.

Feature: This final step represents the culmination of the entire RAG pipeline, providing the user with the synthesized information based on their query and the retrieved context.

Part 2: Essential Python Concepts and Technologies

Building a robust enterprise RAG system as described will require proficiency in several key Python concepts, libraries, and frameworks. Given the Azure-centric environment and the specific requirements, here's a breakdown:

2.1 Core Python Concepts

- **Object-Oriented Programming (OOP):** Essential for structuring a large project, creating reusable components (e.g., document parsers, chunkers, API clients), and managing complexity. Concepts like classes, objects, inheritance, and polymorphism will be heavily utilized.
- **Asynchronous Programming (asyncio, aiohttp):** Many operations in a RAG pipeline (e.g., crawling documents, making API calls to LLMs or vector databases) are I/O-bound. Asynchronous programming will be crucial for improving performance and responsiveness, especially when dealing with large volumes of documents or concurrent user requests.
- **Error Handling and Logging:** Robust `try-except` blocks and comprehensive logging (using Python's `logging` module) are vital for building a production-ready system that can gracefully handle failures, debug issues, and monitor performance.
- **Concurrency and Parallelism (threading, multiprocessing):** For CPU-bound tasks (e.g., complex text processing, OCR), understanding how to leverage multiple CPU cores or threads can significantly speed up the ingestion pipeline.
- **Data Structures and Algorithms:** Efficient use of data structures (lists, dictionaries, sets, queues) and algorithms will be important for optimizing text processing, chunking, and metadata management.

- **Virtual Environments (venv, conda):** Best practice for managing project dependencies and avoiding conflicts between different projects.
- **Packaging and Distribution:** Understanding how to package your Python code (e.g., using `setuptools` or `poetry`) will be important for deploying components as services or functions within Azure.

2.2 Key Python Libraries and Frameworks

2.2.1 Document Ingestion & Preprocessing

- **SharePoint Integration:**
  - `Office365-REST-Python-Client` [1]: A comprehensive library for interacting with SharePoint and Office 365 APIs. This will be critical for the scheduled document crawler to download documents securely from SharePoint sites.
  - `Microsoft Graph SDK for Python` [2]: For broader Microsoft 365 integration, including SharePoint, user management (for ACLs), and potentially Teams integration.
- **OCR for Images:**
  - `pytesseract` [3]: A Python wrapper for Google's Tesseract-OCR Engine. Suitable for extracting text from images.
  - `Azure Computer Vision SDK` [4]: For more advanced and scalable OCR capabilities, especially for cloud-based deployments, leveraging Azure's AI services.
- **Password Removal:**
  - This is highly sensitive and might involve integrating with enterprise key management systems or secure credential stores. Python libraries like `cryptography` could be used for decryption if authorized, but the primary mechanism would likely be an authorized internal service or API.
- **Format Parsing:**
  - **PDF:**
    - `PyPDF2` [5] or `pypdf` (its successor): For basic PDF text extraction.

    - `pdfminer.six` [6]: For more advanced PDF parsing, including layout analysis and handling of complex structures.
    - `fitz` (PyMuPDF) [7]: A fast and robust library for PDF processing, including text extraction, image extraction, and rendering.
- **DOCX:**
    - `python-docx` [8]: For reading and writing Microsoft Word .docx files, extracting text, and handling document structure.
- **XLSX:**
    - `openpyxl` [9]: For reading and writing Excel .xlsx files, accessing cell data, worksheets, and formulas.
    - `pandas` [10]: While primarily a data analysis library, `pandas` can efficiently read and process data from Excel files (`pd.read_excel`).
- **Chunking and Text Processing:**
    - `NLTK` (Natural Language Toolkit) [11] or `spaCy` [12]: For sentence tokenization, word tokenization, and other basic NLP tasks that can aid in chunking.
    - `LangChain` [13] or `LlamaIndex` [14]: These frameworks provide high-level abstractions and utilities for various chunking strategies (recursive character text splitter, semantic chunking), metadata management, and integration with embedding models and vector databases. They are highly recommended for building RAG pipelines.
- **Embedding Generation:**
    - `openai` [15] or `azure-openai` (if using Azure OpenAI Service directly): For interacting with OpenAI's embedding models (e.g., `text-embedding-ada-002`).
    - `sentence-transformers` [16]: For using open-source embedding models that can be run locally or on private infrastructure, offering more control over data privacy if needed.

2.2.2 Retrieval and Generation Pipeline

- **Web Framework (for Web Chat UI and Teams Bot Backend):**
  - `Flask` [17] or `FastAPI` [18]: Lightweight and efficient web frameworks for building RESTful APIs. `FastAPI` is particularly well-suited for asynchronous operations.
  - `Bot Framework SDK for Python` [19]: For building conversational AI experiences and integrating with Microsoft Teams. This would be essential for the Teams bot front end.
- **Azure Entra ID (Authentication and Authorization):**
  - `msal` (Microsoft Authentication Library) [20]: For handling authentication with Azure Entra ID, managing tokens, and securing API access.
- **Vector Database Interaction:**
  - **Azure AI Search:** Use the `azure-search-documents` SDK [21] for interacting with Azure AI Search, including vector search capabilities.
  - **Pinecone:** Use the `pinecone-client` library [22] for interacting with Pinecone.
  - **Azure Cosmos DB:** Use the `azure-cosmos` SDK [23] for Cosmos DB. If using its vector search capabilities, ensure the appropriate version and features are enabled.
  - `LangChain` [13] or `LlamaIndex` [14]: These frameworks provide connectors and abstractions for various vector databases, simplifying the integration.
- **LLM Interaction (Azure OpenAI):**
  - `openai` [15] or `azure-openai` (if using Azure OpenAI Service directly): For making API calls to GPT-4 via Azure OpenAI.
  - `LangChain` [13] or `LlamaIndex` [14]: These frameworks offer robust integrations with LLMs, including prompt templating, chain management, and output parsing, which are crucial for the prompt composer and overall RAG orchestration.

2.3 Data Storage and Management

- **Azure SQL DB:** For structured data storage, as mentioned in the additional context. Python libraries like `pyodbc` [24] or `SQLAlchemy` [25] (an ORM) would be used for

interacting with Azure SQL databases.

- **Azure Blob Storage:** For storing raw documents, processed documents, or intermediate files during the ingestion pipeline. The `azure-storage-blob` SDK [26] would be used for this.

2.4 Development and Deployment Tools

- **Git:** For version control.
- **Docker:** For containerizing applications, ensuring consistent environments across development, testing, and deployment. This is highly recommended for deploying to Azure App Services or Azure Kubernetes Service.
- **Azure CLI / Azure SDK for Python:** For managing Azure resources, deploying applications, and configuring services programmatically.
- **CI/CD Pipelines (Azure DevOps, GitHub Actions):** For automating testing, building, and deployment processes.

2.5 Security Considerations

- **Azure Key Vault:** For securely storing secrets, API keys, and connection strings. Accessing these secrets from Python applications would involve the `azure-keyvault-secrets` SDK [27].
- **Managed Identities:** Leveraging Azure Managed Identities for authenticating to Azure services (e.g., Azure OpenAI, Azure SQL DB, Azure Storage) without managing credentials in code.
- **Role-Based Access Control (RBAC):** Properly configuring RBAC in Azure to control who has access to what resources and services.

Part 3: Implementation Roadmap and Best Practices

This section outlines a phased approach to building the RAG system and highlights best practices for each stage.

3.1 Phase 1: Foundation and Core Components

Objective: Establish the basic infrastructure and develop core functionalities for document ingestion and retrieval.

Steps:

1. **Set up Azure Environment:**
    - Create an Azure tenant if not already available.
    - Set up resource groups, virtual networks, and necessary security configurations (e.g., network security groups).
    - Provision Azure OpenAI Service (GPT-4 API) in an AU region.
    - Provision a Vector Database (Azure AI Search, Pinecone, or Cosmos DB) in an AU region.
    - Set up Azure Blob Storage for raw document storage.
    - Configure Azure Entra ID for application registration and user synchronization.
2. **Develop Document Crawler:**
    - Implement the SharePoint document crawler using `Office365-REST-Python-Client` or `Microsoft Graph SDK`.
    - Focus on secure authentication to SharePoint and handling different site access levels.
    - Implement scheduling mechanism (e.g., Azure Functions with a timer trigger or Azure Logic Apps).
3. **Build Preprocessing Pipeline:**
    - Develop modules for parsing different document formats (PDF, DOCX, XLSX) using the recommended Python libraries.
    - Integrate OCR for image-based content.

- Implement robust error handling for corrupted or unreadable documents.
- Address password-protected documents with authorized mechanisms.

4. **Implement Chunking and Embedding Generation:**
   - Choose and implement a chunking strategy (semantic and/or fixed) using `LangChain` or `LlamaIndex`.
   - Integrate with Azure OpenAI for embedding generation.
   - Develop metadata tagging logic, especially for `site`, `path`, and `ACL info`.
5. **Store Embeddings in Vector DB:**
   - Develop code to ingest the generated embeddings and metadata into the chosen vector database.
   - Ensure efficient indexing for fast retrieval.

Best Practices for Phase 1:

- **Modular Design:** Break down the ingestion pipeline into small, testable modules (e.g., `sharepoint_connector.py`, `pdf_parser.py`, `chunker.py`, `embedding_generator.py`).
- **Incremental Development:** Start with a single document type and gradually add support for others.
- **Data Validation:** Implement rigorous validation at each step of the pipeline to ensure data quality.
- **Monitoring and Logging:** Set up comprehensive logging for the ingestion process to track progress, identify errors, and monitor performance.
- **Security First:** Prioritize secure access to SharePoint and sensitive document handling from the outset.

3.2 Phase 2: Retrieval and LLM Integration

Objective: Develop the user-facing interface, implement the retrieval mechanism, and integrate with the LLM to generate answers.

Steps:

1. **Develop User Interface:**
    - Build the Microsoft Teams bot or Web Chat UI using `Flask` / `FastAPI` and `Bot Framework SDK`.
    - Implement Azure Entra ID authentication for users.
2. **Implement User Authorization and SharePoint Access Filter:**
    - Integrate the user's Entra ID with SharePoint access permissions.
    - Develop the SharePoint Access Filter to restrict document retrieval based on user authorization.
3. **Integrate with Vector DB for Lookup:**
    - Develop the query embedding generation using the same model as for document embeddings.
    - Implement the vector search logic to retrieve top-k relevant chunks from the vector database.
    - Ensure the access filter is applied during retrieval.
4. **Develop Prompt Composer:**
    - Create logic to construct the LLM prompt by injecting the retrieved context and user query.
    - Experiment with different prompt engineering techniques to optimize answer quality.
5. **Integrate with Azure OpenAI:**
    - Make API calls to the GPT-4 model via Azure OpenAI.
    - Implement retry mechanisms and error handling for LLM calls.
6. **Present Final Answer to User:**

- Format the LLM's response for clear and concise presentation in the chat interface.

Best Practices for Phase 2:

- **Iterative Prompt Engineering:** Continuously refine prompts to improve LLM response quality and adherence to instructions.
- **Relevance Tuning:** Experiment with different `k` values for top-k retrieval and explore re-ranking techniques to improve the relevance of retrieved chunks.
- **User Feedback Loop:** Implement a mechanism for users to provide feedback on answer quality, which can be used to further refine the system.
- **Performance Optimization:** Monitor the latency of retrieval and generation steps and optimize where necessary.
- **Security Audits:** Regularly audit the access control mechanisms to ensure they are functioning as intended.

3.3 Phase 3: Advanced Features and SQL GPT Agent

Objective: Enhance the RAG system with advanced features and develop the SQL GPT Agent.

Steps:

1. **Advanced RAG Features:**
    - **Query Expansion/Rewriting:** Use an LLM to rephrase or expand user queries to improve retrieval effectiveness.
    - **Hybrid Search:** Combine keyword search (e.g., using Azure AI Search's full-text capabilities) with vector search for more comprehensive retrieval.
    - **Re-ranking:** Implement re-ranking algorithms (e.g., using cross-encoders) to further refine the order of retrieved chunks before sending them to the LLM.
    - **Conversational Memory:** Implement short-term and long-term memory for the RAG agent to maintain context across multiple turns in a conversation.

- **Source Attribution:** Clearly indicate the source document(s) for the generated answers.

2. **Develop SQL GPT Agent:**
    - **Database Schema Understanding:** The agent needs to understand the schema of `Prime`, `Endata`, and `in4mo` databases.
    - **Natural Language to SQL:** Leverage LLMs (potentially fine-tuned or with advanced prompt engineering) to convert natural language questions into SQL queries. This is where the "SQL + Vector DB" concept comes into play, where examples of natural language questions and their corresponding SQL queries can be embedded and retrieved to guide the LLM.
    - **Secure Database Interaction:** Use `pyodbc` or `SQLAlchemy` to execute generated SQL queries against Azure SQL DBs.
    - **Result Interpretation:** The agent needs to interpret the SQL query results and present them in a user-friendly natural language format.
    - **Complex Query Handling:** Address the complex query examples provided (e.g., "How many claims in in4mo have unanswered chats", "Show me the claims in construction where the prime scopes do not agree with the endata portal scopes"). This will likely involve multi-step reasoning and potentially chaining LLM calls.
    - **Integration with SharePoint RAG:** For queries like "Show me all estimates pending review where qbcc has been left off the estimate lines or has been incorrectly calculated (qbcc insurable works / premium pdf guides will be on sharepoint)", the SQL GPT agent might need to leverage the SharePoint RAG to retrieve relevant documents (e.g., QBCC guides) to inform its SQL query generation or result interpretation.
3. **Unified Teams Bot Front End:**
    - Integrate the SharePoint RAG and SQL GPT Agent into a single Teams bot interface, allowing users to seamlessly interact with both functionalities.
    - Implement logic to route user queries to the appropriate agent (SharePoint RAG or SQL GPT).

Best Practices for Phase 3:

- **Iterative Development for SQL Agent:** Start with simpler SQL queries and gradually increase complexity.
- **Data Security for SQL Agent:** Ensure that the SQL agent only generates and executes authorized queries and does not expose sensitive data.
- **Performance Tuning:** Optimize SQL query generation and execution for large datasets.
- **User Training and Documentation:** Provide clear documentation and training for users on how to effectively use both the SharePoint RAG and SQL GPT agents.

Part 4: General Best Practices and Considerations

- **Data Residency and Security:** Continuously ensure all data (documents, embeddings, LLM inference) remains within Australia and adheres to insurance industry security standards. This includes:
  - **Azure Region Selection:** Always deploy resources to Australian Azure regions (e.g., Australia East, Australia Southeast).
  - **Network Security:** Implement strict network security groups, private endpoints, and virtual network integration to isolate resources.
  - **Encryption:** Ensure data at rest and in transit is encrypted.
  - **Compliance:** Adhere to relevant industry regulations and certifications.
- **Scalability:** Design the system for scalability from the outset, anticipating growth in document volume and user queries. Utilize Azure services that offer auto-scaling capabilities.
- **Observability:** Implement comprehensive monitoring, logging, and alerting for all components of the RAG system. Use Azure Monitor, Application Insights, and custom logging to gain insights into performance, errors, and usage.
- **Cost Management:** Monitor Azure resource consumption and optimize costs by choosing appropriate SKUs, scaling strategies, and resource utilization.

- **Testing:** Implement unit tests, integration tests, and end-to-end tests for all components. For RAG, this includes evaluating retrieval accuracy and LLM response quality.
- **Version Control:** Use Git for all code and configuration management.
- **Documentation:** Maintain clear and up-to-date documentation for the architecture, code, deployment procedures, and user guides.
- **Feedback Loop:** Establish a continuous feedback loop with end-users to identify areas for improvement and new features.

References

[1] Office365-REST-Python-Client: https://pypi.org/project/Office365-REST-Python-Client/
[2] Microsoft Graph SDK for Python: https://learn.microsoft.com/en-us/graph/sdks/sdk-installation?tabs=python
[3] pytesseract: https://pypi.org/project/pytesseract/
[4] Azure Computer Vision SDK: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/overview
[5] PyPDF2: https://pypi.org/project/PyPDF2/
[6] pdfminer.six: https://pypi.org/project/pdfminer.six/
[7] fitz (PyMuPDF): https://pypi.org/project/PyMuPDF/
[8] python-docx: https://python-docx.readthedocs.io/en/latest/
[9] openpyxl: https://openpyxl.readthedocs.io/en/stable/
[10] pandas: https://pandas.pydata.org/
[11] NLTK: https://www.nltk.org/
[12] spaCy: https://spacy.io/
[13] LangChain: https://www.langchain.com/
[14] LlamaIndex: https://www.llamaindex.ai/
[15] OpenAI Python Library: https://pypi.org/project/openai/
[16] sentence-transformers: https://www.sbert.net/
[17] Flask: https://flask.palletsprojects.com/
[18] FastAPI: https://fastapi.tiangolo.com/
[19] Bot Framework SDK for Python: https://github.com/microsoft/botbuilder-python

[20] MSAL Python: https://github.com/AzureAD/microsoft-authentication-library-for-python
[21] Azure Search Documents SDK: https://learn.microsoft.com/en-us/python/api/azure-search-documents/azure.search.documents?view=azure-python
[22] Pinecone Client: https://docs.pinecone.io/docs/python-client
[23] Azure Cosmos DB SDK for Python: https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/sdk-python
[24] pyodbc: https://pypi.org/project/pyodbc/
[25] SQLAlchemy: https://www.sqlalchemy.org/
[26] Azure Storage Blob SDK for Python: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python?tabs=managed-identity%2Cblob-storage-security%2Croles-azure-portal%2Csign-in-azure-cli
[27] Azure Key Vault Secrets SDK for Python: https://learn.microsoft.com/en-us/python/api/azure-keyvault-secrets/azure.keyvault.secrets?view=azure-python