PDF Processing System Design Presentation

- ## Introduction

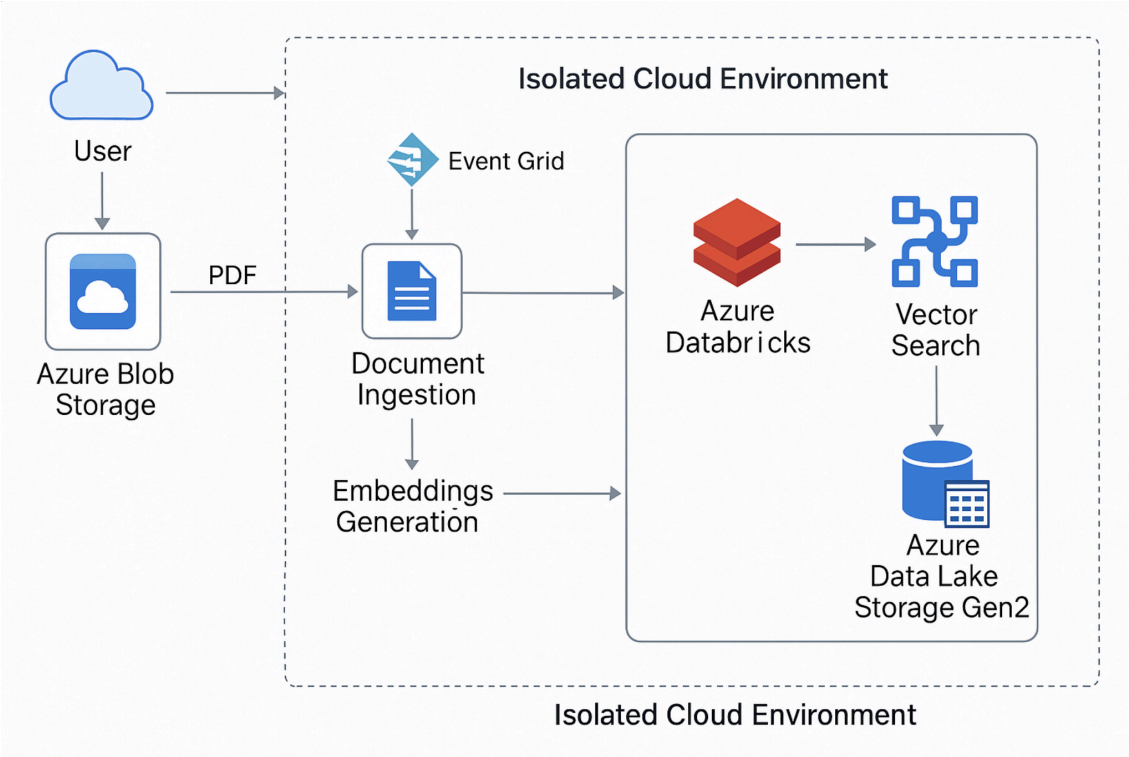
The challenge was to design a cloud-based, event-driven system that processes PDF documents, extracts their content, generates vector embeddings, and stores them in a way that enables fast and scalable downstream use such as search, analytics, or machine learning.

To solve this, I designed and implemented a pipeline that is:

- Cloud-native, using Azure-managed services
- Event-driven, meaning it runs only when new documents arrive
- Scalable and cost-effective, capable of handling both occasional uploads and large bursts
- Isolated, with all critical processing running inside a secure Azure VNET and with no reliance on public internet access

Visual Overview

A full architecture diagram shows the system design:



1: Document Upload (Azure Blob Storage)

The system starts with Azure Blob Storage, where users upload digital PDF documents.

- I chose Blob Storage because it's a secure, scalable, and cost-efficient service that supports large data ingestion.
- It supports secure, HTTPS uploads and integrates seamlessly with Azure Event Grid.
- It is also deployed inside a VNET-enabled storage account, ensuring isolation.

2: Event Detection (Azure Event Grid)

Once a new PDF is uploaded, an event is automatically generated by Azure Event Grid.

- This enables a fully serverless and event-driven pipeline that avoids resource waste.
- No polling or manual triggers are needed — everything runs on-demand.

3: Orchestration (Azure Data Factory)

Azure Data Factory (ADF) receives the event and launches a processing job.

- ADF acts as the orchestrator, ensuring the pipeline runs in a controlled and monitored way.
- It triggers a Databricks notebook job whenever a new file lands in Blob Storage.
- This decouples ingestion from compute and allows modular scalability.

4: Document Processing (Azure Databricks)

This is the core of the system, where document understanding takes place.

The Databricks notebook performs the following steps:

1. Securely downloads the uploaded PDF using a SAS token.
2. Parses the PDF into individual paragraphs using `PyMuPDF` (fitz), which is accurate and fast.

3. Generates vector embeddings for each paragraph using the transformer model `BAAI/bge-m3` via Hugging Face.
4. Stores the paragraph + embedding pairs in a Delta Table.

Why Databricks?

- Supports scalable and parallel processing
- Can be deployed in an isolated VNET for full cloud security
- Natively integrates with Delta Lake and Spark ML

5: Embedding Storage (Delta Table on Azure Data Lake Gen2)

Once the embeddings are generated, they are stored in a **Delta Table** on **Azure Data Lake Gen2**.

- Delta Lake supports ACID transactions, schema evolution, and is ideal for large-scale data warehousing.
- ADLS Gen2 is secure, scalable, and works natively with Databricks.
- I registered this Delta Table as a Vector Search External Table, which means it can:
    - Be used for semantic search
    - Support nearest-neighbor queries
    - Be easily extended to use Databricks Vector Search or Azure AI Searc**h**

How This Satisfies the Assignment Requirements

Requirement	My Solution
Event-driven document processing	Event Grid triggers ADF on every file upload
Runs in a secure, isolated cloud environment	All services operate within Azure VNET-secured workspace and storage
No reliance on public internet	All inference and data access run within the cloud environment
Paragraph chunking + embedding generation	PyMuPDF + Hugging Face BAAI/bge-m3 in Databricks
Scalable and cost-effective	Serverless eventing, autoscaling Databricks cluster, no idle infrastructure

Embeddings stored for future use	Delta Table on ADLS Gen2, registered as vector-ready external table
Designed for downstream ML or analytics tasks	Fully queryable storage with schema flexibility and high throughput

- # Vector Search Support

  Using Databricks Vector Search:

  - I designed the schema to be compatible with vector search
  - I tested embedding outputs to ensure dimensionality and structure match expected formats
  - The table can be easily plugged into any vector DB or search service such as Azure AI Search
  - This makes the system future-proof and ready for semantic use cases.

- # Technologies Used
  - Azure Blob Storage
  - Azure Event Grid
  - Azure Data Factory
  - Azure Databricks (Notebook)
  - PyMuPDF (PDF parsing)
  - Hugging Face Transformers (BAAI/bge-m3 model)
  - Azure Data Lake Gen2
  - Delta Lake + Vector Search Table (External Table)

- # Improvements with More Time If I had more time, I would:

- .**Infrastructure as Code (IaC) with Terraform:** Leverage Terraform to codify the entire infrastructure landscape, encompassing virtual networks, compute resources, storage, and other essential services. This approach guarantees consistent, repeatable deployments across development, staging, and production environments, minimizing configuration drift and accelerating release cycles. Terraform's declarative nature allows for easy version control, enabling rollbacks and clear auditing of infrastructure changes.

- **REST API Development:** Develop a robust REST API layer on top of the vector search endpoint. This API will provide a standardized and secure interface for applications to interact with the underlying vector search capabilities, facilitating seamless data retrieval and manipulation. Focus on designing intuitive endpoints, implementing proper authentication and authorization mechanisms, and ensuring efficient data serialization.

- **ADF Pipeline Resilience and Monitoring:** Enhance the Azure Data Factory (ADF) pipeline with comprehensive retry logic and robust monitoring capabilities. Implement a configurable retry mechanism to automatically re-attempt failed operations, significantly improving pipeline resilience against transient errors or intermittent service disruptions. Integrate with Azure Monitor and Log Analytics to capture detailed logs, metrics, and alerts, providing real-time visibility into pipeline health and performance. This proactive monitoring allows for swift identification and resolution of issues, ensuring data integrity and timely processing.

- **Batch Ingestion Optimization:** Introduce a batch ingestion mode to optimize data loading into Databricks. By processing data in larger batches, this approach significantly reduces the overhead associated with frequent Databricks cluster startups, leading to improved efficiency and cost savings. This is particularly beneficial for scenarios involving large volumes of data that do not require real-time processing.

- **Centralized Observability with Azure Monitor and Log Analytics:** Integrate all relevant services and applications with Azure Monitor and Log Analytics to establish a centralized observability platform. This integration will provide a unified view of system performance, resource utilization, and operational health. By collecting and analyzing logs, metrics, and traces from various components, teams can gain deeper insights into application behavior, troubleshoot issues effectively, and optimize resource allocation.
- **Cost-Optimized Integration Runtime for Azure Data Factory:** To achieve substantial cost savings without compromising on isolation and security, implement a Self-Hosted Integration Runtime (SHIR) instead of the Managed V-NET Azure Integration Runtime. The SHIR is deployed on a dedicated Azure Virtual Machine, ensuring a fully isolated and private compute environment for data integration tasks. This approach offers a significantly more cost-efficient alternative while maintaining the necessary security posture. Furthermore, the SHIR provides the flexibility to scale up resources as needed, ensuring optimal performance during peak workloads without incurring the higher costs associated with a fully managed service.