

Enterprise Knowledge Intelligence Framework - SharePoint Version 2.1

Design Document: Agentic RAG with Vertex AI Search & SP Connector

1. Introduction

This design defines an **Agentic RAG (Retrieval Augmented Generation)** architecture capable of ingesting, understanding, and answering complex queries across a corporate SharePoint ecosystem.

The core of this design leverages the **Google Cloud Vertex AI Search SharePoint Connector** in **Data Ingestion Mode**. This decision is foundational: by ingesting data into a Google-managed index (as opposed to Federated Search), we enable semantic understanding, cross-document reasoning, and multimodal analysis (text, charts, images) ¹.

The architecture is divided into two distinct planes:

- **The Ingestion Plane:** Responsible for the secure synchronization of SharePoint artifacts and ACLs. It addresses the "Master-Detail" relationship where list items point to detailed pages.
- **The Inference Plane:** The runtime environment where an **ADK (Agent Development Kit)** Agent orchestrates retrieval using tools to answer user queries via **Gemini 1.5 Pro**.

2. Ingestion Plane Design

The Ingestion Plane moves data from the Microsoft Tenant to the Google Cloud Platform while maintaining security boundaries and data fidelity.

2.1. SharePoint Data Store Setup (Step-by-Step)

To support "Data Ingestion" and "Real-time Sync," the following specific configuration steps must be executed ².

Prerequisites

- **Google Cloud Project:** Active project with billing enabled.
- **SharePoint Online Account:** Administrative access to the SharePoint instance.
- **Microsoft Entra Access:** Permissions to register applications and grant admin consent.

Step 1: Register Application in Microsoft Entra ID

1. Navigate to **Microsoft Entra Admin Center > App registrations > New registration**.
2. **Name:** VertexAISearch-SharePointConnector.
3. **Supported Account Types:** "Accounts in this organizational directory only".
4. **Redirect URI:** Select "Web" and enter:
https://vertexaisearch.cloud.google.com/console/oauth/sharepoint_oauth.html.
5. **Record IDs:** Note the **Application (client) ID** and **Directory (tenant) ID**.

Step 2: Configure Federated Credentials (Security)

1. In the new App Registration, go to **Certificates & secrets > Federated credentials > Add credential**.
2. **Issuer:** <https://accounts.google.com>.
3. **Subject Identifier:** (Obtain this from the Google Cloud Console during Step 4 below, then return here to paste it).

Step 3: Grant API Permissions

1. Go to **API permissions > Add a permission > Microsoft Graph** ³.
2. Add **Application permissions:** Sites.FullControl.All and User.Read.All (for identity syncing) ⁴.
3. Add **SharePoint Permissions:** Sites.FullControl.All.
4. **Critical:** Click "**Grant admin consent for [Tenant Name]**" to apply these organization-wide.

Step 4: Create Data Store in Vertex AI Agent Builder

1. In Google Cloud Console, go to **Agent Builder > Data Stores > Create Data Store** ⁵.
2. Select **SharePoint Online** as the source ⁶.
3. **Authentication:** Enter the **Client ID** and **Tenant ID** from Step 1. You do *not* need a client secret if using Federated Credentials ⁷.

2.2. Processing Images, Charts, and Tables (Multimodal Ingestion)

A critical requirement for this design is the ability to handle visual data embedded within SharePoint pages.

The Problem: Standard Ingestion

By default, the SharePoint connector functions primarily as a text extractor. It treats documents as plain text streams, which leads to significant data loss for structured content:

- **Tables:** Are often flattened into jumbled lines of text, causing the Agent to lose the relationship between "Row Headers" and "Cell Values"⁸⁸⁸.
- **Images & Charts:** Are completely ignored unless they have Alt-Text, making them invisible to the Agent⁹⁹⁹.

The Solution: Layout-Aware Parsing

To resolve this, the Data Store must be configured with **Document AI** capabilities during the initial setup. This cannot be changed retroactively without recreating the store¹⁰.

Configuration Checklist:

1. **Chunking Strategy:** Select **Layout-aware chunking** (instead of standard/static chunking)¹¹.
2. **Parser:** Select **Layout Parser** (backed by Google's Document AI)¹².
3. **Enable Table Annotation:** This forces the indexer to respect row/column structures, converting tables into structured formats (like Markdown or HTML) that the Agent can query accurately (e.g., "What is the value in Column B for Row 3?")¹³.
4. **Enable Image Annotation / OCR:** This uses Optical Character Recognition to read text inside scanned images or charts. While it does not allow the search engine to "view" pixels, it extracts labels and numbers so the Agent can "read" the chart's content¹⁴.

Comparison of Ingestion Modes:

Feature	Standard Ingestion (Default)	Ingestion with Layout Parser
Tables	Flattened text; structure lost.	Preserves structure (Markdown/HTML) ¹⁵ .
Charts	Ignored / Invisible.	OCR extracts title, axis labels, and data points ¹⁶ .
Images	Ignored.	Captures text inside images (e.g., receipts, diagrams) ¹⁷ .
Headings	Plain text.	Hierarchical chunking preserves context ¹⁸ .

Note on Cost & Latency: Enabling these features increases indexing costs per 1,000 pages and ingestion latency due to the visual analysis required ¹⁹.

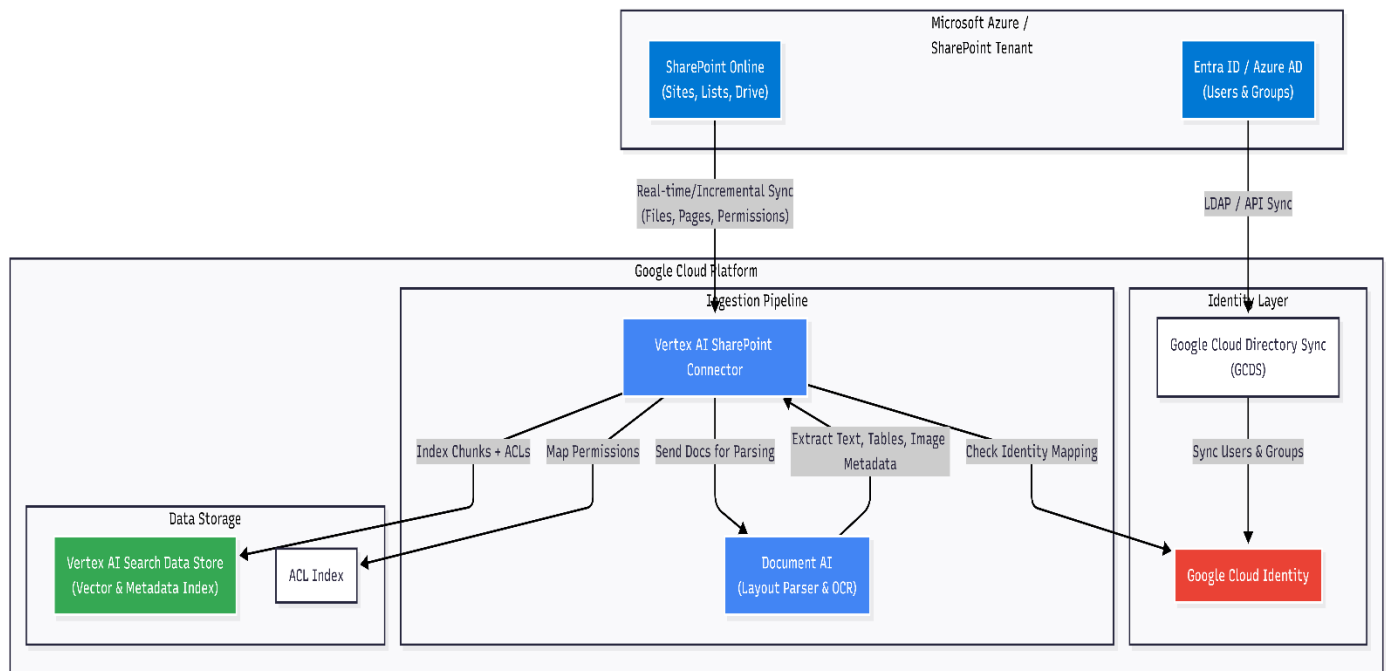
2.3. The "Double-Sync" Security Architecture

A critical design gap addressed is the **ACL Propagation Delay**. Security relies on two parallel sync processes:

1. **Content Sync:** The Connector uses **Real-time/Incremental Sync** to update file ACLs (e.g., "File A is viewable by Group 'HR'") ²⁰.
2. **Identity Sync:** **Google Cloud Directory Sync (GCDS)** must be configured to run frequently (e.g., every 15-30 mins) to sync Entra ID Group Memberships to Google Cloud Identity ²¹.

2.4. Ingestion Plane Diagrams (Exhaustive)

Component Diagram: Ingestion Plane



This diagram illustrates the flow of content and identity data, highlighting the separation between Content Sync (Connector + DocAI) and Identity Sync (GCDS).

Code snippet

graph TD

```

subgraph Azure["Microsoft Azure / SharePoint Tenant"]
    SP["SharePoint Online<br/>(Sites, Lists, Drive)"]
    Entra["Entra ID / Azure AD<br/>(Users & Groups)"]
end

subgraph GCP["Google Cloud Platform"]
    subgraph Identity["Identity Layer"]
        GCDS["Google Cloud Directory Sync<br/>(GCDS)"]
        CloudID["Google Cloud Identity"]
    end

    subgraph IngestionPipeline["Ingestion Pipeline"]
        VASPC["Vertex AI SharePoint Connector"]
        VASPC --> ICA["Index Chunks + ACLs"]
        VASPC --> MP["Map Permissions"]
        VASPC --> SDP["Send Docs for Parsing"]
        VASPC --> ETIM["Extract Text, Tables, Image Metadata"]
        VASPC --> CIM["Check Identity Mapping"]
    end

    subgraph DataStorage["Data Storage"]
        VASDS["Vertex AI Search Data Store<br/>(Vector & Metadata Index)"]
        ACLIndex["ACL Index"]
    end

    DocumentAI["Document AI<br/>(Layout Parser & OCR)"]

    ICA --> VASDS
    MP --> ACLIndex
    SDP --> DocumentAI
    ETIM --> DocumentAI
    CIM --> Identity
    GCDS --> CloudID
end

```

```

    subgraph Ingestion["Ingestion Pipeline"]
        Connector["Vertex AI SharePoint Connector"]
        DocAI["Document AI<br/>(Layout Parser & OCR)"]
    end

    subgraph Storage["Data Storage"]
        Index["Vertex AI Search Data Store<br/>(Vector & Metadata Index)"]
        ACL_Store["ACL Index"]
    end
end

%% Flows
SP -->|"Real-time/Incremental Sync<br/>(Files, Pages, Permissions)"|
Connector
Entrance -->|"LDAP / API Sync"| GCDS
GCDS -->|"Sync Users & Groups"| CloudID

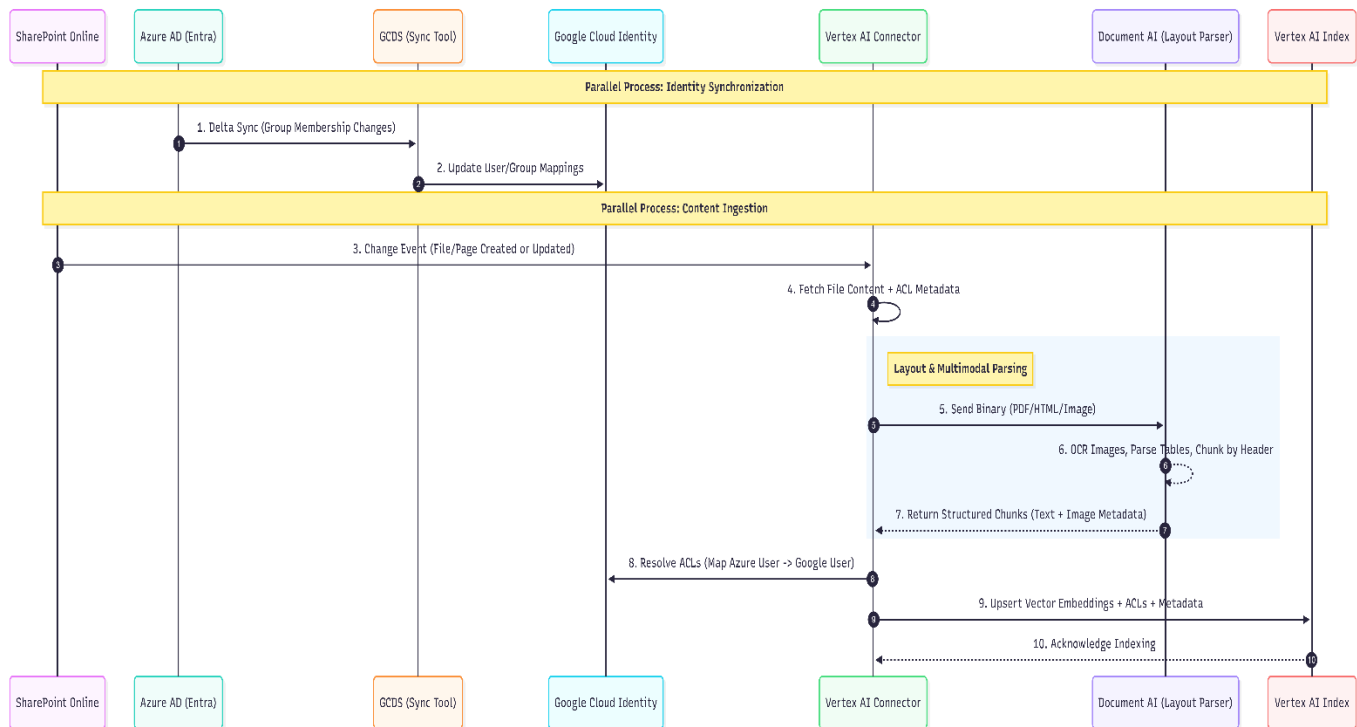
Connector -->|"Check Identity Mapping"| CloudID
Connector -->|"Send Docs for Parsing"| DocAI
DocAI -->|"Extract Text, Tables, Image Metadata"| Connector
Connector -->|"Index Chunks + ACLs"| Index
Connector -->|"Map Permissions"| ACL_Store

%% Styling
style SP fill:#0078d4,stroke:#fff,color:#fff
style Entrance fill:#0078d4,stroke:#fff,color:#fff
style Connector fill:#4285f4,stroke:#fff,color:#fff
style DocAI fill:#4285f4,stroke:#fff,color:#fff
style Index fill:#34a853,stroke:#fff,color:#fff
style CloudID fill:#ea4335,stroke:#fff,color:#fff

```

Sequence Diagram: Ingestion Pipeline

This details the parallel processing of Identity and Content, including the Layout Parser loop for multimodal data.



Code snippet
sequenceDiagram

```

autonumber
participant SP as SharePoint Online
participant Entra as Azure AD (Entra)
participant GCDS as GCDS (Sync Tool)
participant CloudID as Google Cloud Identity
participant Connect as Vertex AI Connector
participant DocAI as Document AI (Layout Parser)
participant VAI as Vertex AI Index

```

Note over SP, VAI: Parallel Process: Identity Synchronization

Entra->>GCDS: 1. Delta Sync (Group Membership Changes)

GCDS->>CloudID: 2. Update User/Group Mappings

Note over SP, VAI: Parallel Process: Content Ingestion

SP->>Connect: 3. Change Event (File/Page Created or Updated)

Connect->>Connect: 4. Fetch File Content + ACL Metadata

rect rgb(240, 248, 255)

Note right of Connect: Layout & Multimodal Parsing

Connect->>DocAI: 5. Send Binary (PDF/HTML/Image)

DocAI-->>DocAI: 6. OCR Images, Parse Tables, Chunk by Header

DocAI-->>Connect: 7. Return Structured Chunks (Text + Image Metadata)

end

Connect->>CloudID: 8. Resolve ACLs (Map Azure User -> Google User)
Connect->>VAI: 9. Upsert Vector Embeddings + ACLs + Metadata
VAI-->>Connect: 10. Acknowledge Indexing

3. Inference Plane Design

The Inference Plane uses an **Agentic RAG** workflow. The Agent orchestrates retrieval using a custom tool to answer user queries via **Gemini 1.5 Pro**.

3.1. Multimodal Agentic Workflow (Deep Dive)

To use Gemini 1.5 Pro's vision capabilities with the SharePoint Data Store, we use a **"Retrieve, Then View"** pattern. The standard search tool returns *text*; a secondary tool returns the *image*.

How the Agent Handles Multimodal Queries

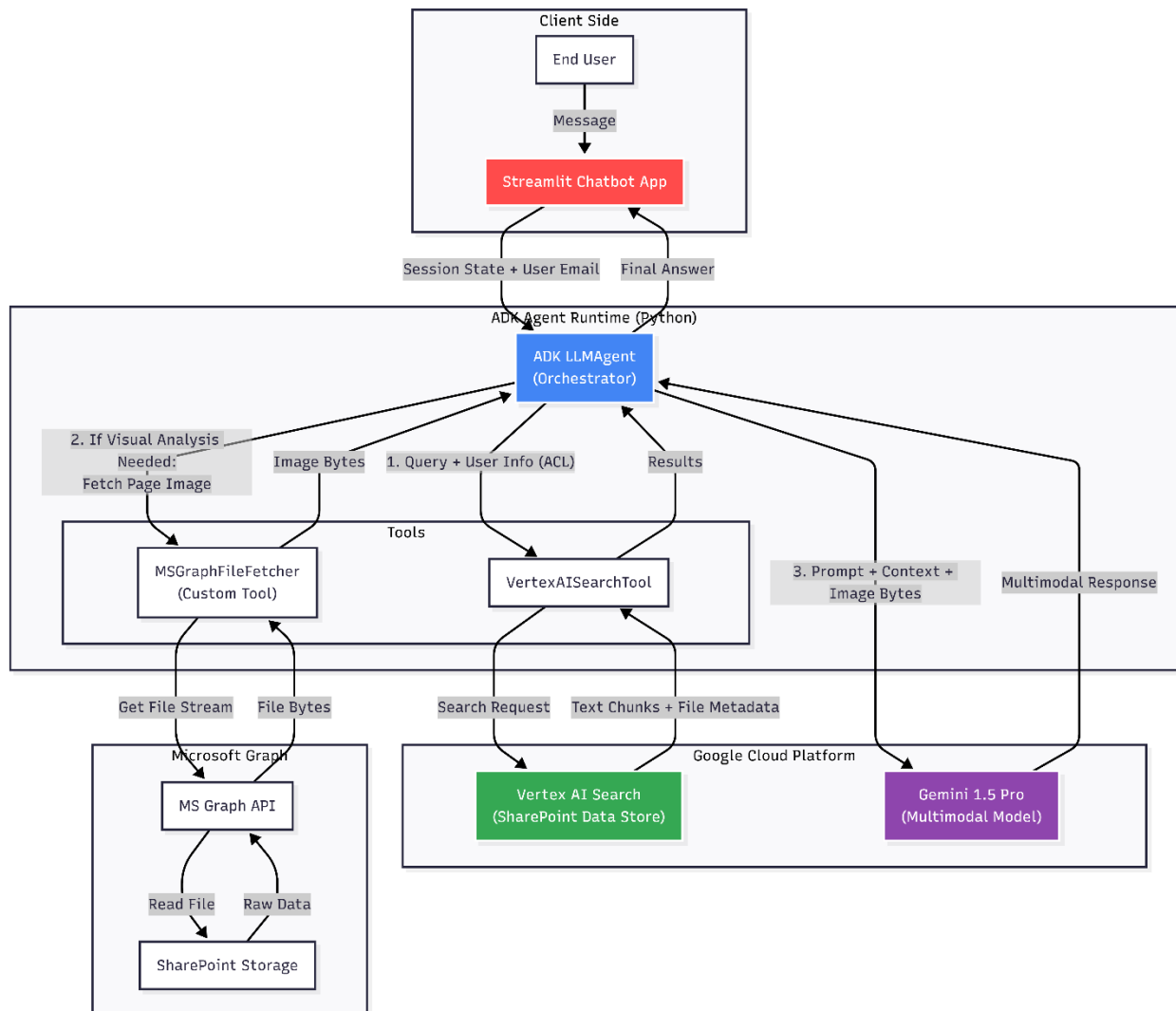
When a user asks: *"Look at the sales chart in the Q3 Report and tell me if it is volatile."*

1. **Step 1: Retrieval (Locating the Artifact)**
 - The Agent uses the SharePointSearchTool.
 - **Query:** "Q3 Report sales chart"
 - **Result:** The Search Engine (using Image Annotation) identifies a relevant chunk: *"Figure 3: Sales Trend 2024"*.
 - **Metadata:** The result contains file_path (".../Q3_Report.pdf") and page_number ("12")²².
2. **Step 2: Visual Acquisition (Fetching the Pixels)**
 - The Agent recognizes the user wants a *visual* analysis ("Look at...").
 - The Agent calls the VisualFetchTool (a custom tool wrapping MS Graph API)²³.
 - **Action:** fetch_page_image(file_path=".../Q3_Report.pdf", page=12).
 - **Output:** The tool returns the raw image bytes (JPEG/PNG) of Page 12.
3. **Step 3: Multimodal Generation (Reasoning)**
 - The Agent calls Gemini 1.5 Pro.
 - **Payload:** [User Prompt: "Is this volatile?", Image: <Page_12_Bytes>]²⁴.
 - **Reasoning:** Gemini looks at the jagged lines on the chart (which OCR missed) and concludes "Yes, highly volatile."

3.2. Inference Plane Diagrams (Exhaustive)

Component Diagram: Agentic RAG Workflow

This shows the relationship between the Agent, its Tools (Search vs. Fetch), and the Multimodal Model.



Code snippet

graph TD

subgraph "Client Side"

User[End User]

Streamlit[Streamlit Chatbot App]

end

subgraph "ADK Agent Runtime (Python)"

Agent["ADK LLM Agent (Orchestrator)"]

subgraph "Tools"

SearchTool[VertexAISearchTool]

FetchTool["MSGraphFileFetcher (Custom Tool)"]

```

    end
end

subgraph "Google Cloud Platform"
    VAIS["Vertex AI Search<br/>(SharePoint Data Store)"]
    Gemini["Gemini 1.5 Pro<br/>(Multimodal Model)"]
end

subgraph "Microsoft Graph"
    GraphAPI[MS Graph API]
    SP_Store[SharePoint Storage]
end

%% Flows
User -- "Message" --> Streamlit
Streamlit -- "Session State + User Email" --> Agent

Agent -- "1. Query + User Info (ACL)" --> SearchTool
SearchTool -- "Search Request" --> VAIS
VAIS -- "Text Chunks + File Metadata" --> SearchTool
SearchTool -- "Results" --> Agent

Agent -- "2. If Visual Analysis Needed:<br/>Fetch Page Image" --> FetchTool
FetchTool -- "Get File Stream" --> GraphAPI
GraphAPI -- "Read File" --> SP_Store
SP_Store -- "Raw Data" --> GraphAPI
GraphAPI -- "File Bytes" --> FetchTool
FetchTool -- "Image Bytes" --> Agent

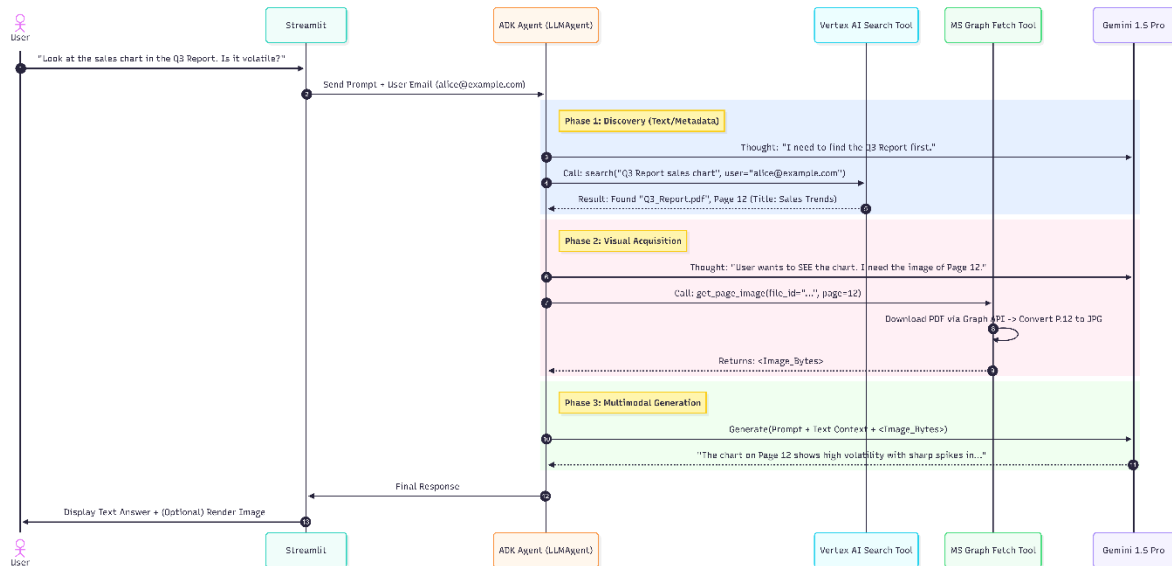
Agent -- "3. Prompt + Context + Image Bytes" --> Gemini
Gemini -- "Multimodal Response" --> Agent
Agent -- "Final Answer" --> Streamlit

%% Styling
style Streamlit fill:#ff4b4b,stroke:#fff,color:#fff
style Agent fill:#4285f4,stroke:#fff,color:#fff
style Gemini fill:#8e44ad,stroke:#fff,color:#fff
style VAIS fill:#34a853,stroke:#fff,color:#fff

```

Sequence Diagram: Inference Workflow

This details the logic flow for a multimodal query, showing the distinct "Search" vs. "Fetch" phases.



Code snippet

sequenceDiagram

autonumber

actor User

participant Streamlit

participant Agent as ADK Agent (LLMAgent)

participant Search as Vertex AI Search Tool

participant Fetch as MS Graph Fetch Tool

participant Gemini as Gemini 1.5 Pro

User->>Streamlit: "Look at the sales chart in the Q3 Report. Is it volatile?"

Streamlit->>Agent: Send Prompt + User Email (alice@example.com)

rect rgb(230, 240, 255)

Note right of Agent: Phase 1: Discovery (Text/Metadata)

Agent->>Gemini: Thought: "I need to find the Q3 Report first."

Agent->>Search: Call: search("Q3 Report sales chart", user="alice@example.com")

Search-->>Agent: Result: Found "Q3_Report.pdf", Page 12 (Title: Sales Trends)

end

rect rgb(255, 240, 245)

Note right of Agent: Phase 2: Visual Acquisition

Agent->>Gemini: Thought: "User wants to SEE the chart. I need the image of Page 12."

Agent->>Fetch: Call: get_page_image(file_id="...", page=12)

Fetch-->>Fetch: Download PDF via Graph API -> Convert P.12 to JPG

Fetch-->>Agent: Returns: <Image_Bytes>

end

rect rgb(240, 255, 240)

Note right of Agent: Phase 3: Multimodal Generation

Agent->>Gemini: Generate(Prompt + Text Context + <Image_Bytes>)

Gemini-->>Agent: "The chart on Page 12 shows high volatility with sharp spikes in..."

end

Agent->>Streamlit: Final Response

4. Agentic RAG Implementation (Production Code & Logic)

This section details the production-ready implementation for three critical use cases: **Master-Detail synthesis**, **Table extraction**, and **Multimodal chart analysis**.

4.1. Use Case 1: Master-Detail (List-Detail Page Relationship)

Goal: Answer queries that require information from both a SharePoint List Item (e.g., "Pattern Catalog") and the detailed page linked within it.

The "Agentic Join" Pattern:

The SharePoint Connector indexes Lists and Pages separately. The Agent must bridge them.

Implementation (Python):

Python

```
from google.cloud import discoveryengine_v1 as discoveryengine
```

```
class MasterDetailTool(Tool):
```

```
    """
```

```
    Orchestrates the retrieval of a List Item and its linked Detail Page.
```

```
    """
```

```
    def run(self, pattern_name: str, user_email: str):
```

```
        # Step 1: Search for the Master List Item
```

```
        user_info = discoveryengine.UserInfo(user_id=user_email)
```

```
        list_request = discoveryengine.SearchRequest(
            query=f'"{pattern_name}" AND context:"Pattern Catalog"',
            user_info=user_info,
            page_size=1
        )
```

```
        list_response = self.client.search(request=list_request)
```

```
        if not list_response.results:
```

```
            return "Pattern not found in Catalog."
```

```
        # Extract Metadata
```

```
        item = list_response.results[0].document.derived_struct_data
```

```
        status = item.get('Status', 'Unknown')
```

```
        owner = item.get('Owner', 'Unknown')
```

```
        # KEY STEP: Extract the Link to the Detail Page
```

```
        detail_url = item.get('LinkToDocumentation', None)
```

```
        if not detail_url:
```

```
            return f"Found Pattern: {pattern_name} (Status: {status}), but no detail link exists."
```

```

# Step 2: Search for the Detail Page Content using the URL as a filter
# We search for the *content* associated with this specific URL
detail_request = discoveryengine.SearchRequest(
    query=pattern_name, # Re-use query to highlight relevant chunks
    filter=f'url: "{detail_url}"', # Filter by the URL found in Step 1
    user_info=user_info
)
detail_response = self.client.search(request=detail_request)

# Step 3: Synthesize
detail_content = detail_response.results[0].document.derived_struct_data.get('snippets')[0]['snippet']

return f"""
**Pattern Metadata (from List):**
- Name: {pattern_name}
- Status: {status}
- Owner: {owner}

**Detail Documentation (from Page):**
{detail_content}
"""

```

4.2. Use Case 2: SP Detail Page Query about Tables

Goal: Accurately answer questions about specific rows/columns in a table (e.g., "What is the SLA for Silver Tier?").

Logic:

Because Layout Parser and Table Annotation are enabled²⁵, tables are indexed as structured Markdown/HTML rather than jumbled text. No special "Table Tool" is needed; the standard search tool returns structured chunks.

Agent Implementation:

Python

```

# Standard Search Tool Configuration
# Ensure 'max_extrative_answer_count' is set to 1 to get precise table row hits.
request = discoveryengine.SearchRequest(
    content_search_spec=discoveryengine.SearchRequest.ContentSearchSpec(
        extractive_content_spec=discoveryengine.SearchRequest.ContentSearchSpec.ExtrativeContentSpec(
            max_extrative_answer_count=1 # Helps isolate the specific table row
        )
    ),
    # ... other params ...
)

# LLM Prompt Strategy for Tables
prompt = """
You are analyzing structured documentation.
If the search result contains a Markdown table (e.g. | Col A | Col B |),
read the row corresponding to the user's query carefully.

```

Example: If asked "Value for X", find the row starting with "X" and read the "Value" column.
"""

4.3. Use Case 3: SP Detail Page Query about Charts/Images

Goal: Provide visual analysis of a chart (e.g., "Is the trend increasing?").

Production Logic:

A. The MSGraphFetcher Implementation

This class handles the "Visual Fetch" responsibility. It authenticates with Microsoft Graph, resolves a SharePoint URL to a file ID, downloads the file stream, and renders a specific page as an image.

Python

```
import os
import requests
import msal
import base64
from io import BytesIO
from pdf2image import convert_from_bytes
from typing import Optional, Any

class MSGraphFetcher:
    """
    Production tool to fetch files from SharePoint via MS Graph and convert
    specific
    pages to images for Multimodal RAG.
    """
    def __init__(self, tenant_id: str, client_id: str, client_secret: str):
        self.base_url = "https://graph.microsoft.com/v1.0"
        self.authority = f"https://login.microsoftonline.com/{tenant_id}"
        self.client_id = client_id
        self.client_secret = client_secret
        self.scopes = ["https://graph.microsoft.com/.default"]

        # Initialize MSAL Confidential Client
        self.app = msal.ConfidentialClientApplication(
            self.client_id,
            authority=self.authority,
            client_credential=self.client_secret
        )

    def _get_headers(self) -> dict:
        """Helper to get a valid access token."""
        result = self.app.acquire_token_silent(self.scopes, account=None)
        if not result:
            result = self.app.acquire_token_for_client(scopes=self.scopes)

        if "access_token" in result:
            return {"Authorization": f"Bearer {result['access_token']}"}
        else:
```

```

        raise Exception(f"Could not acquire token:
{result.get('error_description')}")

    def _resolve_sharepoint_url_to_drive_item(self, sharepoint_url: str) ->
str:
        """
        Converts a direct SharePoint URL (returned by Vertex AI Search) into
a
        Graph API 'DriveItem' ID using the /shares/ endpoint encoding trick.
        """
        # 1. Base64 encode the URL according to MS Graph spec
        # (base64 encoded, strip padding, replace characters)
        b64_url = base64.urlsafe_b64encode(sharepoint_url.encode("utf-
8")).decode("utf-8")
        encoded_url = "u!" + b64_url.rstrip("=")

        # 2. Call the /shares endpoint to resolve the Item ID
        endpoint = f"{self.base_url}/shares/{encoded_url}/driveItem"
        response = requests.get(endpoint, headers=self._get_headers())

        if response.status_code == 200:
            return response.json()["id"]
        else:
            raise Exception(f"Failed to resolve URL: {sharepoint_url}. Graph
Error: {response.text}")

    def get_page_image(self, file_url: str, page_number: int) -> bytes:
        """
        Downloads the PDF from the URL and converts the specific page (1-
based index) to JPEG bytes.
        """
        try:
            # Step 1: Resolve URL to Graph ID
            drive_item_id =
self._resolve_sharepoint_url_to_drive_item(file_url)

            # Step 2: Download File Content
            # Endpoint: /drive/items/{item-id}/content
            # Note: We query the global /drive/items (works for any site
library)
            # If using specific sites, path might vary, but /shares
resolution usually handles context.
            # Using the resolved item ID is safer.
            # We need to construct the call. Since /shares returned the item,
we can use its reference.

            # Alternative robust path:
            download_url =
f"{self.base_url}/shares/u!{base64.urlsafe_b64encode(file_url.encode('utf-
8')).decode('utf-8').rstrip('=')}/driveItem/content"

            response = requests.get(download_url,
headers=self._get_headers(), stream=True)

            if response.status_code != 200:
                raise Exception(f"Failed to download file content:
{response.text}")

```

```

pdf_bytes = response.content

# Step 3: Convert Specific Page to Image
# page_number is 1-based from the User/Agent
images = convert_from_bytes(
    pdf_bytes,
    first_page=page_number,
    last_page=page_number,
    fmt="jpeg"
)

if not images:
    raise Exception(f"Page {page_number} could not be extracted
(File might be too short).")

# Step 4: Return Bytes
img_byte_arr = BytesIO()
images[0].save(img_byte_arr, format='JPEG')
return img_byte_arr.getvalue()

except Exception as e:
    print(f"Error in MSGraphFetcher: {str(e)}")
    return None

```

B. The MultimodalChartAgent Implementation

This class acts as the orchestrator. It combines the `SharePointSearchTool` (for text/metadata) and the `MSGraphFetcher` (for vision).

Python

```

import vertexai
from vertexai.generative_models import GenerativeModel, Part, SafetySetting
from google.cloud import discoveryengine_v1 as discoveryengine

class MultimodalChartAgent:
    def __init__(self, project_id, location, data_store_id, graph_creds:
dict):
        self.project_id = project_id

        # 1. Initialize Gemini 1.5 Pro (Multimodal)
        vertexai.init(project=project_id, location=location)
        self.model = GenerativeModel("gemini-1.5-pro-001")

        # 2. Initialize Search Client
        self.search_client = discoveryengine.SearchServiceClient()
        self.serving_config = self.search_client.serving_config_path(
            project=project_id,
            location=location,
            data_store=data_store_id,
            serving_config="default_search"
        )

        # 3. Initialize Graph Fetcher

```

```

self.graph_fetcher = MSGraphFetcher(
    tenant_id=graph_creds['tenant_id'],
    client_id=graph_creds['client_id'],
    client_secret=graph_creds['client_secret']
)

def _find_document_location(self, query: str, user_email: str):
    """
    Helper: Searches Vertex AI to find the File URL and Page Number.
    """
    user_info = discoveryengine.UserInfo(user_id=user_email)
    request = discoveryengine.SearchRequest(
        serving_config=self.serving_config,
        query=query,
        page_size=3, # Get top few results

content_search_spec=discoveryengine.SearchRequest.ContentSearchSpec(
    # We want snippets to help us guess the page if metadata is
missing

snippet_spec=discoveryengine.SearchRequest.ContentSearchSpec.SnippetSpec(
    return_snippet=True
),
    # IMPORTANT: Request Extractive Content to find specific page
hits

extractive_content_spec=discoveryengine.SearchRequest.ContentSearchSpec.Extra
ctiveContentSpec(
    max_extractive_answer_count=1
)
),
    user_info=user_info
)

response = self.search_client.search(request=request)

if not response.results:
    return None, None, "No results found."

# Logic to pick the best result
# In production, you might pass these snippets to an LLM to decide
which file is best.
# Here, we take the top rank.
best_doc = response.results[0].document.derived_struct_data
file_url = best_doc.get('link', None)

# Extract Page Number
# Vertex AI Indexer usually populates 'page_number' in derived data
if Layout Parser is on.
# Otherwise, it might be in the extractive segment data.
page_num = 1 # Default

# Check extractive segments for page info
if hasattr(response.results[0], 'pages'):
    # If page_numbers are returned in the response metadata
    page_num = int(response.results[0].pages[0])
elif 'page_number' in best_doc:

```

```

        page_num = int(best_doc['page_number'])

        return file_url, page_num, best_doc.get('title')

def run(self, user_query: str, user_email: str):
    print(f"--- Agent Received Query: {user_query} ---")

    # Step 1: Discovery
    file_url, page_num, title = self._find_document_location(user_query,
user_email)

    if not file_url:
        return "I couldn't find any documents matching your request."

    print(f"Step 1: Found Document '{title}' at URL: {file_url} (Page
{page_num})")

    # Step 2: Visual Fetch
    # Only perform if the query implies visual analysis (charts, trends,
diagrams)
    # We assume the user wants visual analysis based on the Agent call,
    # or we could use a lightweight LLM call to classify intent here.

    print("Step 2: Fetching visual context via MS Graph...")
    image_bytes = self.graph_fetcher.get_page_image(file_url, page_num)

    if not image_bytes:
        return f"I found the document '{title}', but I couldn't retrieve
the visual page image due to permissions or format issues."

    # Step 3: Multimodal Reasoning
    print("Step 3: Sending Text + Image to Gemini 1.5 Pro...")

    prompt = f"""
You are an expert analyst.
The user asked: "{user_query}"

Attached is an image of Page {page_num} from the document "{title}".
Analyze the charts, diagrams, or tables in this image to answer the
user's question.
Be specific about data points you see.
"""

    response = self.model.generate_content(
        [
            Part.from_text(prompt),
            Part.from_data(data=image_bytes, mime_type="image/jpeg")
        ]
    )

    return response.text

```

C. Example Usage in Production

```

Python
if __name__ == "__main__":

```

```

# Configuration (Load from Env Variables in prod)
PROJECT_ID = "my-gcp-project"
LOCATION = "global"
DATA_STORE_ID = "sharepoint-ds-id"

# MS Graph Service Principal Credentials
# Ensure this App has 'Sites.Read.All' in Azure AD
GRAPH_CREDS = {
    "tenant_id": "my-azure-tenant-id",
    "client_id": "my-azure-client-id",
    "client_secret": "my-azure-client-secret"
}

# Instantiate
agent = MultimodalChartAgent(PROJECT_ID, LOCATION, DATA_STORE_ID,
GRAPH_CREDS)

# Run Query
# Scenario: User wants to analyze a chart in a PDF report
answer = agent.run(
    user_query="Look at the 'Quarterly Revenue' chart in the Q3 Financial
Report. Is the growth consistent?",
    user_email="alice@company.com" # Passed for Search ACLs
)

print("\n--- Final Answer ---")
print(answer)
    Part.from_text(f"User Query: {user_query}"),
    Part.from_text("Analyze this chart image to answer the user."),
    Part.from_data(data=image_bytes, mime_type="image/jpeg")
]
)

return response.text

# Example Usage
agent = MultimodalChartAgent()
print(agent.answer_chart_query(
    "Look at the 'Server Load' chart in the 'Ops Guide' and tell me the peak time.",
    "alice@company.com"
))

```

5. Conclusion: Vertex AI Search vs. Custom Pipeline

This section compares the proposed Vertex AI Search design (Data Ingestion mode) against a custom-built Python RAG pipeline.

Pros of Proposed Design (Vertex AI Search)

- **Reduced Engineering Overhead:** The connector handles the complexities of SharePoint authentication, rate limiting, and crawling logic out of the box. There is no need to write or maintain custom crawlers.

- **Native Multimodal Support:** Features like Layout Parser and OCR are integrated directly into the ingestion pipeline, requiring simple configuration rather than complex integration with Vision APIs.
- **Managed Scalability:** Google manages the index scale, syncing, and infrastructure, reducing operational burden.
- **Security & ACLs:** Identity mapping and ACL inheritance are built-in features, significantly reducing the risk of accidental data exposure compared to manually coding permission filters.

Cons of Proposed Design

- **Less Granular Control:** You are limited to the connector's predefined sync schedules (e.g., incremental sync latency) and parsing logic. A custom pipeline allows for exact control over when and how data is crawled.
- **Latency in "Real-Time":** While "real-time" sync is efficient, it still has a propagation delay (minutes) compared to a direct API query which is instant.
- **Black Box Chunking:** While Layout Aware chunking is powerful, you cannot fine-tune the chunking algorithm (e.g., "split by specific regex") as precisely as you could with a custom Python script using libraries like `LangChain`.
- **Cost:** Document AI (Layout Parser) adds a cost per page indexed, which can be higher than simple text extraction in a custom pipeline.

Verdict: For an Enterprise Knowledge Framework, the **Vertex AI Search design** is the superior choice due to its robust security, lower maintenance overhead, and integrated multimodal capabilities. The trade-offs in granular control are outweighed by the reliability and security of a managed platform.