AI Empower Enterprise RAG Service (V5) Deployment Manifest

This document serves as the single source of truth for the Version 5.0 architecture, encompassing all Infrastructure as Code (Terraform), application code (Python/Docker), and sequential deployment commands executed to build the fully functional, asynchronous RAG pipeline with **Storage Decoupling (V5)**.

The V5 system is deployed against the GCP Project `test-rag-backend-v4`.

1. Project Architecture and Repository Structure

1.1 V5 Architectural Summary

V5 is an incremental update to V4, introducing a **Dual-Write** capability to future-proof storage costs (V6 roadmap) while maintaining real-time search.

Component	V4 (Before)	V5 (Deployed State)	Purpose of Change
Ingestion Worker	Writes only to Firestore.	Writes to **Firestore (Search)** and **GCS Parquet (Archive)**.	Mitigates future cost of storing >100K vectors in Firestore.
Search Engine	Firestore Vector Search	Firestore Vector Search	Remains the primary, fast, real-time search layer.
New Dependencies	None	`pandas`, `pyarrow`	Required for efficient, columnar Parquet file writing.

1.2 Repository Structure (Complete)

```
ai-empower-rag-v5/
├── .gitignore
```

```
├── terraform/              # Infrastructure as Code (IaC)
├── src/                    # Cloud Run Microservices
│   ├── ingestion-dispatcher/ # GCS event handler
│   ├── ingestion-worker/     # Pub/Sub handler (Indexing/Archiving)
│   └── retrieval-api/        # REST API (Query/LLM/Memory)
└── frontend_app/           # Streamlit Cloud App
```

2. Infrastructure as Code (Terraform Files)

These files define the core resources and IAM roles (`main.tf`, `iam.tf`, etc.) and were applied successfully in Phase 1.

`terraform/main.tf` (Core Resources)

```
# 1. Storage Bucket
resource "google_storage_bucket" "rag_bucket" {
  name          = var.bucket_name
  location      = var.region
  force_destroy = false
  uniform_bucket_level_access = true
}

# 2. Pub/Sub Topic (The Async Queue)
resource "google_pubsub_topic" "ingestion_topic" {
  name = "ingestion-tasks"
}

# 3. Pub/Sub Subscription (Worker Pull/Push)
resource "google_pubsub_subscription" "ingestion_sub" {
  name  = "ingestion-workers-sub"
  topic = google_pubsub_topic.ingestion_topic.name
  ack_deadline_seconds = 600
  retry_policy {
    minimum_backoff = "10s"
    maximum_backoff = "600s"
  }
}

# 4. Enable Firestore API and Create Database Instance
resource "google_project_service" "firestore" {
  service = "firestore.googleapis.com"
  disable_on_destroy = false
}
```

```
resource "google_firestore_database" "database" {
  project     = var.project_id
  name        = "(default)"
  location_id = var.region
  type        = "FIRESTORE_NATIVE"
  depends_on = [google_project_service.firestore]
}

# 5. Vector Index for "rag_children"
resource "google_firestore_index" "vector_index" {
  collection = "rag_children"
  database   = google_firestore_database.database.name
  fields {
    field_path = "client_id"
    order      = "ASCENDING"
  }
  fields {
    field_path = "embedding"
    vector_config {
      dimension = 768
      flat {}
    }
  }
}
```

`terraform/iam.tf` (IAM Roles)

This defines the `rag-dispatcher-sa` and `rag-worker-sa` identities.

```
resource "google_service_account" "dispatcher_sa" {
  account_id   = "rag-dispatcher-sa"
  display_name = "V4 Ingestion Dispatcher"
}
resource "google_service_account" "worker_sa" {
  account_id   = "rag-worker-sa"
  display_name = "V4 Ingestion Worker"
}
# Permissions for Worker SA to write to Firestore, use Vertex AI (API must be enabled), and
read/write to GCS.
resource "google_project_iam_member" "worker_firestore" {
  project = var.project_id
  role    = "roles/datastore.user"
  member  = "serviceAccount:${google_service_account.worker_sa.email}"
```

```
}
resource "google_project_iam_member" "worker_ai" {
  project = var.project_id
  role    = "roles/aiplatform.user"
  member  = "serviceAccount:${google_service_account.worker_sa.email}"
}
```

5. Application Code Files (V5 Dual-Write)

5.1 Ingestion Worker (`src/ingestion-worker/`)

`requirements.txt` (V5 Update)

```
google-cloud-storage
google-cloud-firestore
google-cloud-pubsub
google-cloud-aiplatform
langchain==0.1.0
langchain-community==0.0.10
langchain-google-vertexai
pypdf
flask
gunicorn
pandas
pyarrow
```

`main.py` (V5 Final Code with Dual-Write Fix)

```
import os
import json
import logging
import hashlib
from flask import Flask, request, jsonify
from google.cloud import storage
from google.cloud import firestore
from google.cloud.firestore_v1.vector import Vector
from langchain_google_vertexai import VertexAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from pypdf import PdfReader
from io import BytesIO
import base64
```

```
from google.cloud import firestore as fs_module

import pandas as pd
import pyarrow.parquet as pq
import pyarrow as pa
import tempfile
import time
import numpy as np

app = Flask(__name__)
logging.basicConfig(level=logging.INFO)

PROJECT_ID = os.environ.get("GCP_PROJECT", "test-rag-backend-v4")
ARCHIVE_BUCKET_NAME = "ai-empower-rag-v4-uploads"

db = firestore.Client(project=PROJECT_ID)
storage_client = storage.Client()
embeddings = VertexAIEmbeddings(model_name="text-embedding-004",
project=PROJECT_ID)

def get_deterministic_id(key_string):
    return hashlib.sha256(key_string.encode("utf-8")).hexdigest()

@app.route("/", methods=["POST"])
def process_task():
    envelope = request.get_json()
    if not envelope or "message" not in envelope: return jsonify({"error": "Invalid Pub/Sub
message"}), 400

    pubsub_message = envelope["message"]
    data_str = base64.b64decode(pubsub_message["data"]).decode("utf-8")
    job = json.loads(data_str)

    bucket_name = job["bucket"]
    file_path = job["file_path"]
    page_num = job["page_num"]
    client_id = job["client_id"]

    logging.info(f"Worker processing Page {page_num} of {file_path}")
    parquet_data = [] # Buffer for V5 archive

    try:
        bucket = storage_client.bucket(bucket_name)
        blob = bucket.blob(file_path)
```

```
        content = blob.download_as_bytes()
        reader = PdfReader(BytesIO(content))
        page = reader.pages[page_num]
        raw_text = page.extract_text()

        if not raw_text.strip(): return jsonify({"message": "Empty page text."}), 200

        parent_splitter = RecursiveCharacterTextSplitter(chunk_size=2000, chunk_overlap=200)
        parent_chunks = parent_splitter.split_text(raw_text)
        firestore_batch = db.batch()

        for p_idx, parent_text in enumerate(parent_chunks):
            parent_key = f"{file_path}|{page_num}|{p_idx}"
            parent_id = get_deterministic_id(parent_key)

            parent_ref = db.collection("rag_parents").document(parent_id)
            firestore_batch.set(parent_ref, {"client_id": client_id, "source": file_path, "page":
page_num, "content": parent_text})

            child_splitter = RecursiveCharacterTextSplitter(chunk_size=400, chunk_overlap=50)
            child_chunks = child_splitter.split_text(parent_text)

            if child_chunks:
                vectors = embeddings.embed_documents(child_chunks)

                for c_idx, vector_list in enumerate(vectors):
                    child_key = f"{parent_key}|{c_idx}"
                    child_id = get_deterministic_id(child_key)
                    child_text = child_chunks[c_idx]

                    # --- 1. FIRESTORE WRITE: CHILD (V5 Search) ---
                    child_ref = db.collection("rag_children").document(child_id)
                    firestore_batch.set(child_ref, {"client_id": client_id, "parent_id": parent_id, "content":
child_text, "embedding": Vector(vector_list)})

                    # --- 2. ACCUMULATE DATA FOR PARQUET (V6 Archive) ---
                    parquet_data.append({
                        "id": child_id,
                        "client_id": client_id,
                        "parent_id": parent_id,
                        "embedding": vector_list, # Store as raw Python list
                        "content": child_text,
                        "source": file_path,
                        "page": page_num,
```

```
                "timestamp": time.time()
            })

        # Final commit to Firestore
        firestore_batch.commit()

        # --- 3. PARQUET WRITE: ARCHIVE TO GCS ---
        if parquet_data:
            df = pd.DataFrame(parquet_data)

            # Define schema for correct PyArrow list-of-float type
            schema = pa.schema([
                pa.field('id', pa.string()),
                pa.field('client_id', pa.string()),
                pa.field('parent_id', pa.string()),
                pa.field('content', pa.string()),
                pa.field('source', pa.string()),
                pa.field('page', pa.int64()),
                pa.field('timestamp', pa.float64()),
                pa.field('embedding', pa.list_(pa.float32()))
            ])

            table = pa.Table.from_pandas(df, schema=schema, preserve_index=False)

            temp_parquet_file = os.path.join(tempfile.gettempdir(),
f"page_{page_num}_{client_id}_{get_deterministic_id(file_path)}.parquet")
            pq.write_table(table, temp_parquet_file)

            archive_blob_path =
f"archive/{client_id}/{os.path.basename(file_path)}/page_{page_num}.parquet"
            archive_bucket = storage_client.bucket(ARCHIVE_BUCKET_NAME)
            archive_blob = archive_bucket.blob(archive_blob_path)
            archive_blob.upload_from_filename(temp_parquet_file)

            os.remove(temp_parquet_file)
            logging.info(f"V5 Archive successful for Page {page_num}: Wrote {len(parquet_data)}
records to GCS Parquet.")

        logging.info(f"Indexing complete for Page {page_num}.")
        return jsonify({"message": f"Indexed page {page_num}"}), 200

    except Exception as e:
        logging.error(f"Worker Failed: {e}")
        return jsonify({"error": str(e)}), 500
```

```
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8080)
```

5. Frontend Files (`frontend_app/`)

`requirements.txt`

```
streamlit
requests
google-cloud-storage
google-auth
```

`app.py` (Cleaned UI, points to live V4 Retrieval API)

```
import streamlit as st
import requests
import json
import uuid
from google.cloud import storage
from google.oauth2 import service_account
from google.auth import exceptions as auth_exceptions

# --- V4 CONFIGURATION ---
PROJECT_ID = "test-rag-backend-v4"
BUCKET_NAME = "ai-empower-rag-v4-uploads"
API_URL =
"[https://rag-retrieval-v4-873142271416.us-central1.run.app/query](https://rag-retrieval-v4-87314
2271416.us-central1.run.app/query)"

st.set_page_config(page_title="AI Empower RAG V4", layout="wide")
st.title("Enterprise RAG V4 (Async Parent-Child)")
st.subheader("Client Project: " + PROJECT_ID)

# --- State Management ---
if "session_id" not in st.session_state:
    st.session_state.session_id = str(uuid.uuid4())
if "messages" not in st.session_state:
    st.session_state.messages = []
if "client_id" not in st.session_state:
    st.session_state.client_id = "test_client"
```

```
@st.cache_resource(ttl=3600)
def get_storage_client():
    if "gcp_service_account" not in st.secrets:
        st.warning("GCP Service Account Secret is required for upload and not configured.")
        return None

    try:
        key_dict = dict(st.secrets["gcp_service_account"])
        creds = service_account.Credentials.from_service_account_info(key_dict)
        return storage.Client(credentials=creds, project=PROJECT_ID)
    except Exception as e:
        st.error(f"Credential setup failed. Check st.secrets['gcp_service_account']: {e}")
        return None

with st.sidebar:
    st.header("Tenant & Data Management")
    st.session_state.client_id = st.text_input("Client ID (Tenant Key)",
                                       value=st.session_state.client_id)

    st.header("Self-Service Upload")
    uploaded_file = st.file_uploader("PDF/PPTX Document", type=['pdf', 'pptx'])

    if uploaded_file and st.button("Upload & Ingest"):
        client = get_storage_client()
        if client:
            with st.spinner(f"Uploading to {st.session_state.client_id}..."):
                try:
                    bucket = client.bucket(BUCKET_NAME)
                    blob_path = f"uploads/{st.session_state.client_id}/{uploaded_file.name}"
                    blob = bucket.blob(blob_path)
                    blob.upload_from_file(uploaded_file, rewind=True)

                    st.success("Upload Complete! Indexing started (V4 Async Pipeline).")
                    st.info("The document will be searchable in a few minutes.")

                except Exception as e:
                    st.error(f"Upload Failed. Check permissions on the bucket: {e}")

for msg in st.session_state.messages:
    st.chat_message(msg["role"]).write(msg["content"])

if prompt := st.chat_input("Ask a question about your documents..."):
    st.session_state.messages.append({"role": "user", "content": prompt})
    st.chat_message("user").write(prompt)
```

```
    with st.chat_message("assistant"):
        with st.spinner(f"Consulting {st.session_state.client_id} knowledge base..."):
            try:
                response = requests.post(API_URL, json={
                    "query": prompt,
                    "client_id": st.session_state.client_id,
                    "session_id": st.session_state.session_id
                })

                if response.status_code != 200:
                    error_data = response.json()
                    st.error(f"API Error ({response.status_code}): {error_data.get('error', 'Unknown
Error')}")
                    answer = "Error processing request."
                else:
                    data = response.json()
                    answer = data.get("answer", "Error retrieving answer.")
                    st.markdown(answer)

                st.session_state.messages.append({"role": "assistant", "content": answer})

            except Exception as e:
                st.error(f"Connection or runtime error: {e}")
```