# AI Empower Enterprise RAG Service (V4)

# Deployment Manifest

This document serves as the single source of truth for the Version 4.0 architecture, encompassing all Infrastructure as Code (Terraform), application code (Python/Docker), and sequential deployment commands executed to build the fully functional, asynchronous RAG pipeline.

The V4 system is deployed against the GCP Project **`test-rag-backend-v4`**.

## 1. Project Architecture and Repository Structure

### 1.1 Architecture Summary

The V4 system utilizes a Serverless Fan-Out Architecture:

| Component | Function | Status |
|---|---|---|
| **Ingestion Pipeline** | GCS $\rightarrow$ Eventarc $\rightarrow$ Dispatcher $\rightarrow$ Pub/Sub $\rightarrow$ Worker $\rightarrow$ Firestore | **Asynchronous, High-Scale** |
| **Indexing** | Parent-Child (Search Child, Retrieve Parent) | **Precision Optimized** |
| **Persistence** | Firestore Native (`rag_parents`, `rag_children`, `rag_chat_history`) | **Functional w/ Memory** |
| **IaC** | Terraform | **Production Ready** |

### 1.2 Repository Structure (Complete)

```
ai-empower-rag-v4/
├── .gitignore
├── terraform/
│   ├── main.tf
│   ├── variables.tf
│   ├── iam.tf
│   ├── outputs.tf
│   └── provider.tf
├── src/
│   ├── ingestion-dispatcher/
│   │   ├── main.py
│   │   ├── requirements.txt
│   │   └── Dockerfile
│   ├── ingestion-worker/
│   │   ├── main.py
│   │   ├── requirements.txt
│   │   └── Dockerfile
│   └── retrieval-api/
│       ├── main.py
│       ├── requirements.txt
│       └── Dockerfile
└── frontend_app/
    ├── app.py
    └── requirements.txt
```

## 2. Infrastructure as Code (Terraform Files)

These files define the core resources and IAM roles for the entire system.

### `terraform/variables.tf`

```
variable "project_id" {
  description = "The GCP Project ID"
  type        = string
  default     = "test-rag-backend-v4"
}
variable "region" {
  description = "GCP Region"
  type        = string
  default     = "us-central1"
}
variable "bucket_name" {
  description = "Name of the V4 Storage Bucket"
```

```
  type        = string
  default     = "ai-empower-rag-v4-uploads"
}
```

## terraform/provider.tf

```
terraform {
  required_providers {
    google = {
      source  = "hashicorp/google"
      version = ">= 5.0"
    }
  }
}
provider "google" {
  project = var.project_id
  region  = var.region
}
```

## terraform/main.tf

```
# 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

```
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"
}
# Worker Permissions (Firestore Write, Vertex AI, Storage Read)
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}"
}
```

# 4. Application Code Files (Source, Docker, and Frontend)

## 4.1 Ingestion Dispatcher (`src/ingestion-dispatcher/`)

### `requirements.txt`

```
functions-framework==3.*
google-cloud-storage
google-cloud-pubsub
pypdf
cloudevents
```

### `Dockerfile`

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["functions-framework", "--target=handle_upload", "--port=8080"]
```

### `main.py`

```
import functions_framework
import os
import json
import logging
from google.cloud import storage
from google.cloud import pubsub_v1
from pypdf import PdfReader
from io import BytesIO
```

```
PROJECT_ID = os.environ.get("GCP_PROJECT", "test-rag-backend-v4")
TOPIC_ID = "ingestion-tasks"
storage_client = storage.Client()
publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path(PROJECT_ID, TOPIC_ID)
logging.basicConfig(level=logging.INFO)

@functions_framework.cloud_event
def handle_upload(cloud_event):
    data = cloud_event.data
    bucket_name = data["bucket"]
    file_path = data["name"]

    if not (file_path.endswith(".pdf") or file_path.endswith(".pptx")): return

    parts = file_path.split("/")
    client_id = parts[1] if len(parts) >= 3 else "default_tenant"

    logging.info(f"Processing {file_path} for Client: {client_id}")

    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(file_path)
    content = blob.download_as_bytes()

    try:
        if file_path.endswith(".pdf"):
            reader = PdfReader(BytesIO(content))
            total_pages = len(reader.pages)
        else:
            total_pages = 1

        logging.info(f"Found {total_pages} pages in {file_path}")

        futures = []
        for page_num in range(total_pages):
            message_json = {
                "bucket": bucket_name, "file_path": file_path, "page_num": page_num,
                "client_id": client_id, "total_pages": total_pages
            }
            data_str = json.dumps(message_json).encode("utf-8")
            future = publisher.publish(topic_path, data_str)
            futures.append(future)
```

```
        for f in futures: f.result()
        logging.info(f"Successfully dispatched {total_pages} tasks.")

    except Exception as e:
        logging.error(f"Error dispatching {file_path}: {e}")
        raise e
```

## 4.2 Ingestion Worker (`src/ingestion-worker/`)

### `requirements.txt`

```
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
```

### `Dockerfile`

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["gunicorn", "--bind", ":8080", "--workers", "1", "--threads", "8", "--timeout", "0", "main:app"]
```

### `main.py`

```
# Includes Parent-Child logic, deterministic hashing, and Firestore batch writing
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

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

PROJECT_ID = os.environ.get("GCP_PROJECT", "test-rag-backend-v4")
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"]

    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)
        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)
            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, child_text in enumerate(child_chunks):
                    child_key = f"{parent_key}|{c_idx}"
                    child_id = get_deterministic_id(child_key)

                    child_ref = db.collection("rag_children").document(child_id)
                    batch.set(child_ref, {"client_id": client_id, "parent_id": parent_id, "content":
child_text, "embedding": Vector(vectors[c_idx])})

        batch.commit()
        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)
```

### 4.3 Retrieval API (`src/retrieval-api/`)

**`requirements.txt`**

```
flask
gunicorn
```

```
google-cloud-firestore
google-cloud-aiplatform
langchain==0.1.0
langchain-community==0.0.10
langchain-google-vertexai
```

**Dockerfile**

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["gunicorn", "--bind", ":8080", "--workers", "1", "--threads", "8", "--timeout", "0", "main:app"]
```

**main.py**

```
# Implements Parent-Child fetching and conversational memory (last 4 turns)
import os
import logging
from flask import Flask, request, jsonify
from google.cloud import firestore
from google.cloud.firestore_v1.vector import Vector
from google.cloud.firestore_v1.base_vector_query import DistanceMeasure
from langchain_google_vertexai import VertexAIEmbeddings, ChatVertexAI
from langchain.prompts import ChatPromptTemplate
from google.cloud import firestore as fs_module

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

PROJECT_ID = os.environ.get("GCP_PROJECT", "test-rag-backend-v4")
CHAT_HISTORY_COLLECTION = "rag_chat_history"
MAX_HISTORY_TURNS = 4

db = firestore.Client(project=PROJECT_ID)
embeddings = VertexAIEmbeddings(model_name="text-embedding-004",
project=PROJECT_ID)
llm = ChatVertexAI(model_name="gemini-2.5-pro", project=PROJECT_ID, temperature=0.1)

@app.route("/query", methods=["POST"])
def handle_query():
    data = request.get_json()
```

```
    if not data or "query" not in data: return jsonify({"error": "Missing 'query' field"}), 400

    user_query = data["query"]
    client_id = data.get("client_id", "default_client")
    session_id = data.get("session_id", "default_session")

    try:
        # --- 1. Retrieve Chat History ---
        history_ref = db.collection(CHAT_HISTORY_COLLECTION).document(session_id)
        history_doc = history_ref.get()
        past_conversation = ""

        if history_doc.exists:
            past_msgs = history_doc.to_dict().get("messages", [])[-MAX_HISTORY_TURNS:]
            for msg in past_msgs:
                past_conversation += f"{msg['role']}: {msg['content']}\n"

        # 2. Vector Search (Parent-Child Logic)
        query_vector = embeddings.embed_query(user_query)
        collection = db.collection("rag_children")
        vector_ref = Vector(query_vector)

        results = collection.where(filter=firestore.FieldFilter("client_id", "==", client_id))\
                        .find_nearest(
                            vector_field="embedding",
                            query_vector=vector_ref,
                            distance_measure=DistanceMeasure.COSINE,
                            limit=7,
                            distance_result_field="distance"
                        ).get()

        context_text = "No relevant documents found."
        if results:
            parent_ids = set(doc.to_dict()["parent_id"] for doc in results)
            parent_refs = [db.collection("rag_parents").document(pid) for pid in parent_ids]
            parent_docs = db.get_all(parent_refs)

            context_text = ""
            for p_doc in parent_docs:
                if p_doc.exists:
                    chunk = p_doc.to_dict()
                    context_text += f"\n[Source: {chunk['source']}, Page:
{chunk['page']}]\n{chunk['content']}\n"
```

```
        # 3. Generate Answer
        prompt = ChatPromptTemplate.from_template("""
        You are an expert medical AI assistant. Answer the question strictly based on the provided
context.
        Use the Chat History to understand follow-up questions or resolve ambiguous pronouns
(e.g., "the two").

        Chat History:
        {history}

        Context:
        {context}

        Question:
        {question}

        Answer:
        """)

        chain = prompt | llm
        response = chain.invoke({
            "context": context_text,
            "history": past_conversation if past_conversation else "No previous conversation
history.",
            "question": user_query
        })

        final_response = {"answer": response.content, "context_used": context_text}

        # --- 4. Save State (Update History) ---
        new_messages = [
            {"role": "user", "content": user_query},
            {"role": "assistant", "content": response.content}
        ]
        history_ref.set({"messages": fs_module.ArrayUnion(new_messages)}, merge=True)

        return jsonify(final_response), 200

    except Exception as e:
        logging.error(f"Retrieval Error: {e}")
        return jsonify({"error": str(e)}), 500

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

### 4.4 Streamlit Frontend (`frontend_app/`)

**`requirements.txt`**

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

**`app.py`**

```
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)" # REPLACE with live URL

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}")
```