RAG Ingestion System Architecture

Focus: Excel ingestion pipeline for structured policy data + semantic retrieval.
Flow: Client → API → Parsing → Transformation → Storage (Postgres & Pinecone) → Retrieval.
Monitoring & Authentication applied across services.

Component Documentation

Component Name: Client Application (Web/Mobile)
Purpose: Provides user interface to upload Excel files and trigger ingestion.
Limitations:

- Limited by network latency and device constraints
- Requires responsive design across platforms
- Security risks if uploads are not validated client-side
  Necessity: Critical – User entry point into the system
  Alternatives: Desktop app, CLI tools, automated batch jobs

Component Name: API Gateway (FastAPI)
Purpose: Receives ingestion requests (`/ingest-excel`), validates payloads, routes to parser.
Limitations:

- Single-threaded by default (requires scaling with workers)
- API rate limits under heavy traffic
- Relies on external server deployment (Uvicorn/Gunicorn)
  Necessity: Critical – Central request router
  Alternatives: Flask + Celery, Django REST, Spring Boot

Component Name: Excel Parser & Validator
Purpose: Extracts data from uploaded Excel files, enforces schema validation.
Limitations:

- Memory intensive for large files
- Limited to `.xls/.xlsx` formats
- Manual intervention required for invalid schema cases
  Necessity: Critical – Ensures clean, structured ingestion
  Alternatives: Apache POI, openpyxl, ETL tools

Component Name: Data Transformer
Purpose: Normalizes and cleans policy data; splits insurance period into start/end dates.
Limitations:

- Transformation logic must evolve with requirements
- Complex transformations impact performance

- Error-prone with inconsistent date formats
  Necessity: Critical – Prepares data for structured storage
  Alternatives: Pandas scripts, Spark ETL pipelines

Component Name: PostgreSQL Database
Purpose: Stores structured policy data with relational integrity and indexing.
Limitations:

- Scaling challenges with very large datasets
- Connection pool bottlenecks under concurrency
- Requires schema tuning for performance
  Necessity: Critical – Primary structured data store
  Alternatives: MySQL, Amazon Aurora, Microsoft SQL Server

Component Name: Embedding Generator (Gemini)
Purpose: Generates vector embeddings from text for semantic search.
Limitations:

- API latency and throughput limits
- Cost per token for large-scale embedding generation
- Limited by model's context window
  Necessity: Important – Enables semantic search in retrieval phase
  Alternatives: OpenAI embeddings, Cohere, SentenceTransformers (local)

Component Name: Vector Database (Pinecone)
Purpose: Stores document embeddings for semantic similarity search and retrieval.
Limitations:

- Query cost and storage scale with dataset size
- Requires careful index dimension setup
- Latency increases as dataset grows
  Necessity: Critical – Core requirement for semantic retrieval in RAG
  Alternatives: Weaviate, FAISS, Elasticsearch dense vectors, Qdrant

Component Name: Monitoring & Logging
Purpose: Provides system observability via health checks, performance metrics, and error tracking.
Limitations:

- Adds monitoring overhead
- Alerting requires proper configuration
- Needs separate storage for metrics/logs
  Necessity: Important – Ensures production reliability
  Alternatives: Prometheus/Grafana, DataDog, New Relic

Component Name: Authentication Service
Purpose: Secures API endpoints, enforces authentication & authorization policies.
Limitations:

- Adds request latency
- Requires secure key/token management
- Single point of failure if not replicated
  Necessity: Important – Required for compliance & security
  Alternatives: OAuth2 providers, JWT tokens, API Gateway Auth

Production Considerations

- Load balancing for scalability
- Database connection pooling
- Async I/O for API performance
- Retry logic for embedding/DB calls
- Comprehensive error handling
- Configurable deployments (Docker/Kubernetes)
- CI/CD integration for reliable updates