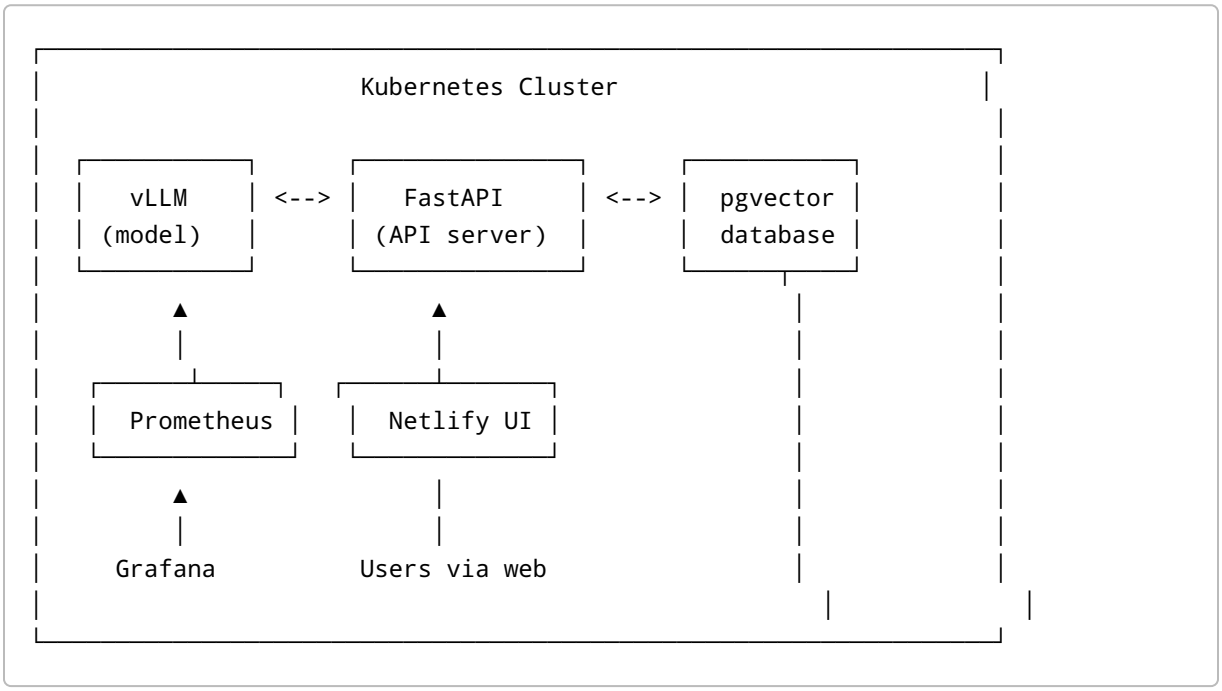
ChatGPT

Project 3 – Part 2: Architecture, RAG & Deep Research

1. End-to-End Architecture

The summarisation system is organised as a **monorepo** with separate packages for training, service, retrieval, evaluation, frontend and infrastructure. It supports four environments—development, test, staging and production—and uses Kubernetes namespaces to isolate them. A high-level component diagram is as follows:



Components:

1. **vLLM** – hosts your fine-tuned summarisation model (T5, BART, Mistral, Mixtral) and exposes OpenAI-compatible endpoints for chat and completions [1].
2. **FastAPI service** – provides REST endpoints (`/summarize`, `/summarize_multi`, `/research`, `/rag/ingest`, `/rag/query`, `/feedback`, `/metrics`, `/health`). It calls vLLM to generate summaries, orchestrates retrieval (for RAG and Deep Research), stores feedback and serves Prometheus metrics.
3. **pgvector database** – a PostgreSQL extension that adds a `vector` type and nearest-neighbour search [2]. Used to store embeddings of documents and queries. The `documents` table might look like:

```
CREATE TABLE IF NOT EXISTS documents (
  id SERIAL PRIMARY KEY,
  url TEXT,
  content TEXT,
  embedding VECTOR(768)
);
CREATE INDEX ON documents USING ivfflat (embedding vector_cosine_ops);
```

1. **Prometheus & Grafana** – monitor latency, throughput, error rates and quality metrics [3] [4]. Prometheus scrapes `/metrics` endpoints; Grafana visualises the data and triggers alerts.
2. **Netlify frontend** – a Vite + React application that interacts with the API. Netlify automatically builds and deploys previews for each branch and pull request [5] [6].

Each environment (dev/test/staging/prod) has its own namespace, API host (`api-dev`, `api-pr-<PR#>`, `api-staging`, `api`), Netlify domain and secrets. Helm charts manage these deployments with values files per environment.

2. Integrating `pgvector` and Netlify

2.1 Data Ingestion & Storage

When you ingest documents for retrieval, the API splits them into chunks (e.g. 500 tokens), generates embeddings and stores them in the `documents` table. `pgvector` allows ACID-compliant storage and supports multiple distance metrics (inner product, cosine, Euclidean) for similarity search [2]. Indexes like IVFFlat accelerate queries.

2.2 Retrieval & Summarisation

At query time the API computes an embedding for the question and performs a similarity search on the `documents.embedding` column. It retrieves the top-k passages and feeds them into the LLM via vLLM. Prompts instruct the model to summarise the information into bullet points, cite the sources and avoid hallucinations. For multi-document summarisation, a map–reduce or refine chain is applied [7] [8]. This retrieval-plus-generation loop constitutes your RAG pipeline.

2.3 Frontend Routing & Environments

The Netlify app reads `VITE_API_URL` from the build context (production, branch deploy, deploy preview) to call the correct backend. For example:

- **Production:** `https://api.yourdomain.com`
- **Staging:** `https://api-staging.yourdomain.com`
- **Preview (PR #42):** `https://api-pr-42.yourdomain.com`

This separation allows you to test new features without impacting users and to review UI changes via Netlify's preview URLs [5]. Each environment has its own Postgres database (`sum_dev`, `sum_test`, etc.) to prevent cross-pollution.

3. RAG Pipeline Implementation

Implementing RAG involves several scripts and endpoints:

1. `rag/ingest.py` – accepts files or URLs, extracts text, splits into chunks (500–1,000 tokens), generates embeddings using a sentence-transformer or the LLM's embedding function, and upserts them into the `documents` table.
2. `rag/retrieve.py` – given a query, computes its embedding and performs a vector similarity search using `SELECT id, content, url, embedding <-> query_embedding AS distance FROM documents ORDER BY distance LIMIT k;` (for Euclidean distance). Returns the top passages and their metadata.
3. **Integration in FastAPI** – endpoints `/rag/ingest` and `/rag/query` call these scripts. The summarisation endpoints (`/summarize`, `/research`) call `retrieve()` internally when retrieval is needed.
4. **Citation enforcement** – prompts are designed to include the passage text or a footnote marker pointing to the `url` field. This encourages the model to cite its sources and reduces hallucinations [9].

4. Multi-Document Summarisation & Deep Research

4.1 Map–Reduce and Refine Strategies

To summarise multiple documents you need to balance completeness and context preservation. In the **map–reduce** strategy, the API summarises each retrieved document independently and then combines those summaries into a global one; this scales well and can run in parallel [7]. The **refine** strategy processes documents sequentially, refining the summary as new information arrives [8]. You might implement both and select the approach based on the number of documents and available compute.

4.2 Deep Research Workflow

The **Deep Research** feature expands on RAG by performing open-ended research across the web:

1. **Search & fetch:** Given a complex query, the API uses a `search_client` (e.g. Bing or Google Programmable Search) to find relevant articles. It fetches the full text (respecting rate limits and robots.txt) and stores it in the RAG database.
2. **Chunk & embed:** Text is cleaned, split into chunks and embedded as in the RAG pipeline.
3. **Retrieve & summarise:** The API retrieves the most relevant passages using vector search and summarises them into 3–5 bullet points. Prompts instruct the model to plan the research (e.g. brainstorm subtopics), cite sources and avoid speculation.
4. **Iterative planning:** You can optionally implement multi-step planning where the model first identifies information gaps, then issues additional search queries to fill them. This transforms the system into an agentic researcher.
5. **Feedback loop:** Users can rate the quality of the research summary. This feedback is stored and later used in RLHF to train the model to produce deeper, more comprehensive research answers.

4.3 Training Deep Research (RLHF)

To make the Deep Research feature truly robust, you can train the model with RLHF on a research-specific dataset. As described in Part 1, collect complex questions, compile source documents, and produce gold-standard bullet answers. Generate candidate research summaries, collect human rankings and train a reward model that penalises hallucinations and rewards accurate citations. Fine-tune the summariser using PPO or DPO so that it plans, verifies and synthesises information like an expert researcher [10].

5. Hallucination Reduction & Quality Assurance

To ensure reliability, the architecture includes multiple layers of quality control:

1. **RAG & citations** provide context and traceability [9].
2. **Factuality metrics** (e.g. QAFactEval, NLI) detect unsupported statements and gate deployments.
3. **Reward penalties** discourage hallucination during RLHF training.
4. **Prometheus & Grafana** dashboards track error rates, latency, length distribution and factuality trends [3] [4]. Alerts notify the team when metrics degrade.

By combining an architecture grounded in retrieval and monitoring with careful training, you build a trustworthy summariser and research assistant. Continue to Part 3 for a step-by-step implementation plan and sprint breakdown.

---

[1] OpenAI Compatible Server — vLLM
https://nm-vllm.readthedocs.io/en/0.5.0/serving/openai_compatible_server.html

[2] The pgvector extension - Neon Docs
https://neon.com/docs/extensions/pgvector

[3] What is Prometheus? | New Relic
https://newrelic.com/blog/best-practices/what-is-prometheus

[4] What is Grafana?
https://www.statsig.com/perspectives/what-is-grafana

[5] Deploy Previews | Netlify Docs
https://docs.netlify.com/deploy/deploy-types/deploy-previews/

[6] Branch deploys | Netlify Docs
https://docs.netlify.com/deploy/deploy-types/branch-deploys/

[7] [8] Document Summarization Solution Patterns using Azure Open AI & Langchain - ISE Developer Blog
https://devblogs.microsoft.com/ise/solution-patterns-for-document-summarization-azureopenai/

[9] What Is Retrieval-Augmented Generation aka RAG | NVIDIA Blogs
https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

[10] Reinforcement learning from human feedback - Wikipedia
https://en.wikipedia.org/wiki/Reinforcement_learning_from_human_feedback