ChatGPT

Project 3 – Part 3: Learning & Execution Plan

This part outlines a beginner-friendly roadmap to implement the summarisation and research system. It combines theory, hands-on tasks and deliverables. As a project manager with over 20 years of experience, I recommend organising the work into sprints so that you learn while building.

1. Four-Environment Strategy

Before diving into coding, understand that your solution will run in four environments:

Environment	Purpose	Characteristics
Development (dev)	Daily coding and experimentation.	Runs in a `sum-dev` namespace with minimal resources. Deploys automatically from the `develop` branch. Netlify branch deploy shows the dev UI.
Testing (test)	Preview environments for each pull request.	Each PR creates an isolated release `sum-pr-<PR#>` in the `sum-test` namespace. Netlify deploy previews let reviewers see UI changes [1].
Staging (staging)	Pre-production check.	Mirrors production settings (same replicas, probes, resource limits) in the `sum-staging` namespace. Deploys from the `staging` branch.
Production (prod)	End-user facing system.	Runs in `sum-prod` with horizontal pod autoscaling, PodDisruptionBudgets and stricter security. Deploys from release tags (e.g. `v1.0.0`) with manual approval.

Each environment has its own Postgres database and secrets to avoid data leakage. Helm charts parameterise the number of replicas, resource limits and ingress hostnames.

2. Sprint-by-Sprint Breakdown

The roadmap uses two-week sprints (adjust as needed). Each sprint includes **learning objectives**, **tasks** and **deliverables**.

Sprint 0 – Foundations (1 week)

Learn:

- Refresh Python basics (functions, classes, virtual environments).
- Understand Git workflow (branching, commits, pull requests).
- Read about transformers and summarisation (e.g. blog posts or the original Transformer paper).

Tasks:

1. Set up a virtual environment and install packages (`transformers`, `datasets`, `evaluate`, `fastapi`, `uvicorn`).
2. Clone the repository template and inspect the folder structure (train/service/rag/eval/web/infra).
3. Run the local stack via `docker compose` to start Postgres, vLLM (with Mistral or T5) and the FastAPI API. Call `/health` and `/summarize` using `curl` or Postman.

Deliverables: Local environment runs; notes on any setup issues; updated README.

Sprint 1 – Data Preparation & Baseline Summariser (1 week)

Learn:

- Explore the CNN/Daily Mail dataset structure and the `datasets` library.
- Understand tokenisation, sequence lengths and ROUGE metrics.

Tasks:

1. Write `src/train/data.py` to download and preprocess the dataset (select `article` and `highlights`, clean, split and tokenise). Save processed datasets to `data/`.
2. Fine-tune a baseline summariser using `t5-small` or `t5-base`. Use `Trainer` from `transformers` or `trl` with cross-entropy loss. Train for 1–2 epochs on a subset (e.g. 5 % of data) to verify the pipeline.
3. Implement `src/eval/run_rouge.py` to compute ROUGE-1/2/L scores on a validation set and print results.
4. Read a sample of generated summaries to assess quality.

Deliverables: Processed dataset; baseline model checkpoint; ROUGE report.

Sprint 2 – PEFT & Alternative Models (1 week)

Learn:

- Study the PEFT library (LoRA/QLoRA) and how it reduces training costs.
- Research BART, Pegasus, Mistral 7B and Mixtral 8×7B models [2].

Tasks:

1. Implement `src/train/sft_lora.py` to apply LoRA to your baseline model. Experiment with `t5-base` or `t5-large` and record memory usage.
2. Fine-tune `facebook/bart-large-cnn` on the dataset using LoRA; compare results to T5.
3. If resources permit, fine-tune `mistralai/Mistral-7B-Instruct` or `mixtral-8x7B-Instruct` with LoRA. Use gradient checkpointing to fit the model in GPU memory.
4. Update evaluation scripts to report BERTScore and QAFactEval (for factuality) in addition to ROUGE.

Deliverables: LoRA-tuned models (T5, BART, Mistral); comparison table showing metrics and training cost; lessons learned.

Sprint 3 – RAG Pipeline (2 weeks)

Learn:

- Understand vector embeddings and pgvector usage [3].
- Study the RAG concept and why it improves accuracy [4].

Tasks:

1. Write `rag/ingest.py` to ingest documents: pass local files or URLs, extract text (use `BeautifulSoup` for HTML), split into 500–1,000-token chunks, generate embeddings (use `SentenceTransformer` or `OpenAIEmbeddings`) and upsert into the `documents` table. Use a `pgvector` column with dimension matching your embedding.
2. Write `rag/retrieve.py` to compute query embeddings and perform vector search using SQL (inner product or cosine similarity). Test retrieval with sample questions.
3. Extend the FastAPI service: add `/rag/ingest` (POST) and `/rag/query` (POST) endpoints. The former ingests documents; the latter returns the top passages for a query.
4. Modify the summarisation endpoint `/summarize` to optionally call `retrieve()` and include retrieved context in the prompt. Use this to summarise articles that are longer than the model's context window.
5. Add logging and metrics for ingestion and retrieval times; expose them via `/metrics`.

Deliverables: Working RAG ingestion and retrieval scripts; integrated API endpoints; demonstration of improved accuracy on queries that require external information.

Sprint 4 – Deep Research & RLHF (2 weeks)

Learn:

- Explore open-domain search APIs (e.g. SerpAPI, Bing). Understand how to fetch and parse web pages ethically (respecting rate limits and robots.txt).
- Learn about map–reduce vs. refine strategies for multi-document summarisation [5] [6].
- Review RLHF pipelines and alternative DPO/IPO methods [7].

Tasks:

1. Implement a `search_client.py` that queries an external search API, collects result URLs, fetches the articles (using `requests` + `BeautifulSoup`) and stores the content in the RAG database. Include metadata (URL, title, date).
2. Create a `/research` endpoint in FastAPI: accept a question, call the search client, ingest articles, retrieve relevant passages and summarise them. Use the map–reduce or refine chain to generate a bulleted answer with citations.
3. Design prompts that instruct the model to plan, verify and summarise: "List the key subtopics you will research, find supporting passages, then summarise them into 5 bullets with citations."
4. Collect human feedback on research answers. For each complex question, generate 2–3 summaries with different prompts or strategies; ask reviewers to rank them.
5. Train a reward model from the research feedback. Use TRLX to run a PPO loop on the research dataset. Alternatively, implement DPO to update the summariser directly [7].

Deliverables: Fully functional Deep Research endpoint; research reward model; RLHF-tuned model; analysis of improvements in answer completeness and citation quality.

Sprint 5 – Multi-Document & Multi-Modal Summarisation (1 week)

Learn:

- Deepen understanding of map–reduce and refine summarisation chains.
- Explore summarising non-text modalities (e.g. transcripts or slides).

Tasks:

1. Implement the `/summarize_multi` endpoint to accept multiple documents (via URLs or raw text). Use the map–reduce strategy to summarise each document individually, then combine the summaries. Optionally implement the refine strategy for better context preservation.
2. Extend evaluation scripts to compute ROUGE and QAFactEval across multi-document tasks. Compare map–reduce vs. refine on a small benchmark.
3. Experiment with summarising audio transcripts (use a speech-to-text tool like `whisper` to convert audio to text). Store transcripts in the RAG database and summarise them.

Deliverables: Multi-document summarisation API; evaluation report comparing strategies; demonstration of summarising other modalities.

Sprint 6 – Backend & Frontend Integration (1 week)

Learn:

- Review asynchronous programming in FastAPI and how to call vLLM.
- Learn basic React (components, hooks, state) if new to frontend development.

Tasks:

1. Build the React UI under `src/web/`: create pages/tabs for **Summarise** and **Deep Research**. Use forms to collect text or URLs and call your API via `fetch` or `axios`. Display results with citations and allow users to submit feedback (e.g. thumbs up/down or star rating).
2. Configure CORS in FastAPI (`fastapi.middleware.cors.CORSMiddleware`) to allow calls from your Netlify domain(s).
3. Add a feedback storage table in Postgres (`feedback(id, prompt, summary, rating, created_at)`). Implement `/feedback` endpoint to insert new feedback.
4. Use Netlify's contexts in `netlify.toml` to set `VITE_API_URL` per environment. Verify that branch deploys and deploy previews call the correct API hosts [8].

Deliverables: Interactive web app hosted on Netlify; ability to submit summaries and research queries; feedback stored in the database.

Sprint 7 – Observability & SLOs (1 week)

Learn:

- Study Prometheus metrics exposition, Grafana dashboard creation and alert rules [9] [10].

Tasks:

1. Instrument the FastAPI service using `prometheus_client` to expose metrics: request counts, latency histograms, error rates, summary length distribution and custom RLHF reward metrics.
2. Enable vLLM's built-in metrics endpoint and configure Prometheus to scrape it.
3. Add a Prometheus `ServiceMonitor` to the Helm chart for each environment. Configure the scrape interval (e.g. 15 seconds in prod, 30 seconds in dev).
4. Create Grafana dashboards: display p95 latency, throughput, error rate, RLHF reward progression and QAFactEval scores. Define alert thresholds (e.g. error rate > 1 %, latency > 0.8 s).
5. Document service level objectives (SLOs) for availability, latency and quality. Integrate these into your CI/CD gates so that a deployment is blocked if metrics regress.

Deliverables: Metrics exposition; Grafana dashboards; alert rules; SLO definitions; integration into CI/CD.

Sprint 8 – CI/CD & Production Readiness (1 week)

Learn:

- Review Helm templating and GitHub Actions syntax. Understand canary or blue-green deployment strategies.

Tasks:

1. Write Helm charts under `infra/helm/` describing Deployments, Services, Ingress, ConfigMaps, Secrets, HorizontalPodAutoscalers and ServiceMonitors. Parameterise image names, resource limits and environment variables.
2. Create values files for `dev`, `test`, `staging` and `prod` (e.g. `values-dev.yaml`, `values-staging.yaml`). Tune replicas, CPU/memory requests and ingress hosts per environment.
3. Define GitHub Actions workflows in `.github/workflows/`:
4. **preview.yml** – build API and front-end images; deploy PR previews to `sum-test`; build Netlify deploy preview.
5. **ci.yml** – run lint (ruff), tests (pytest) and small E2E checks.
6. **deploy-env.yml** – deploy to `sum-dev` on `develop`, `sum-staging` on `staging` merges; require manual approval for `sum-prod` deploys and check metrics before releasing.
7. **nightly-train-eval.yml** – run RLHF training on recent feedback; evaluate and store metrics; keep track of improvements.
8. Harden Docker images (non-root user, read-only filesystem, cosign signatures); generate a Software Bill of Materials (SBOM). Use network policies and PodDisruptionBudgets in prod.
9. Write runbooks for incident response, rollback and database migration. Provide instructions for on-call engineers.

Deliverables: Working CI/CD pipeline; Helm charts; production deployment; documentation and runbooks.

Sprint 9 – Final Polishing & Presentation (optional)

If time allows, perform an extra sprint to polish and present your work:

1. Conduct user testing of the summariser and Deep Research feature; collect qualitative feedback and iterate on prompts.
2. Optimise latency by experimenting with different inference servers (e.g. `quantised-ggml` models) or adjusting vLLM batch sizes.
3. Write a final report summarising your improvements (e.g. ROUGE gain vs. baseline, reduction in hallucinations, human preference scores). Prepare a presentation or blog post.

Deliverables: Refined models; final report; polished web app.

3. Common Pitfalls & Tips

- **Insufficient context:** Splitting documents into very small chunks can cause the model to miss context. Tune chunk sizes and overlap.
- **Reward hacking:** When designing the RLHF reward, ensure it balances relevance, brevity and factuality; otherwise the model may exploit loopholes.
- **Data leakage:** Keep training and test sets separate. Avoid injecting test articles into your RAG database.
- **Resource constraints:** Fine-tuning large models requires GPUs. Use LoRA/QLoRA and start with small subsets to debug.
- **Monitoring blindness:** Deploying without metrics makes it hard to diagnose issues. Instrument from the beginning.

By following this sprint plan, you will gradually build a sophisticated summarisation and research system while learning the underlying technologies. Continue to Part 4 for CI/CD details, monitoring and final recommendations.

---

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

2 Mistral 7B | Mistral AI
https://mistral.ai/news/announcing-mistral-7b

3 The pgvector extension - Neon Docs
https://neon.com/docs/extensions/pgvector

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

5 6 Document Summarization Solution Patterns using Azure Open AI & Langchain - ISE Developer Blog
https://devblogs.microsoft.com/ise/solution-patterns-for-document-summarization-azureopenai/

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

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

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

10 What is Grafana?
https://www.statsig.com/perspectives/what-is-grafana