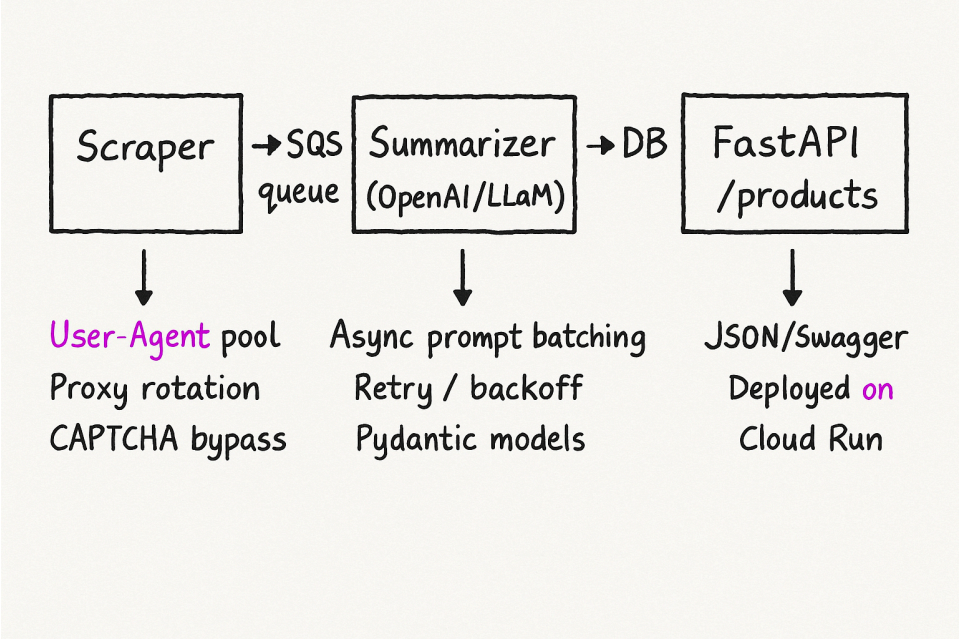
Goal: Build a **production-grade mini-pipeline** that scrapes 10+ products, enriches them with LLM summaries, and exposes the data via a REST API….packaged, tested, containerized, and deployable in hopefully 6 hrs…

---

1 Solution Architecture (bird's-eye)



Loose coupling via an SQS-style queue keeps scraping & LLM compute independent, enabling parallelism and future scale-out. ^^^

---

2 Tech Stack Choices

Layer	Choice	Rationale
Scraper	**Playwright (async Python)**	Handles infinite scroll & JS-heavy DOM; built-in stealth & device emulation.
Data Layer	**SQLite → Postgres (Docker)**	Start lightweight; switchable via SQLAlchemy.
Summarization	**OpenAI GPT-4o API** (fallback: **Mistral-7B via Ollama**)	Guarantees quality; local model keeps cost optional.

API	**FastAPI + Uvicorn**	Async, type-hinted, auto-docs (Swagger).
Packaging	**Poetry**, **Docker**, **docker-compose**	Reproducible builds.
CI/CD	**GitHub Actions → GCP Cloud Run**	Push-to-deploy; secrets via GCP Secret Manager.
Tests	**pytest-asyncio**, **Playwright trace viewer**	Coverage for scraping & API.

---

3 Pipeline Walkthrough

1. **Discovery**
    - Target: *Best Buy Canada → "Laptops under $1000"* category (static URL etc).
    - Robots.txt checked; only public HTML + meta scraped.
2. **Scraping (Playwright)**
    - Headless Chromium, device = desktop.
    - Concurrency: `asyncio.gather()` with 5 workers, rate-limited at 1 req/sec per domain.
    - Anti-bot: random user-agents, residential proxy pool, CAPTCHA fallback (2Captcha API).
    - Output JSON to SQS-like queue (`aiosqs` local).
3. **LLM Enrichment**
    - Prompt template: *"Act as a retail copywriter. Given {title}, {specs} … return 3-bullet USP + 1-line tagline."*
    - Streaming batching (≤4k tokens) for cost control. Or any engineering constraint given.
    - Response validated by Pydantic; retries on `RateLimitError` with exponential back-off. Very plausible here.

4. **Persistence**
    - Normalized tables: `products`, `summaries`, `meta_job_run`.
    - Alembic migrations ready for Postgres prod swap.
5. **API Layer**
    - `GET /products` (list, pagination)
    - `GET /products/{id}` (detail)
    - CORS enabled for future frontend.
6. **Observability**
    - Structured logs (loguru → GCP Logging).
    - Prometheus metrics endpoint `/metrics` (scrape latency, LLM cost).
7. **Security & Compliance**
    - Secrets via Docker env vars / GCP Secrets.
    - Data retained ≤30 days; GDPR delete endpoint.

---

4 File/Repo Layout (just the overview honestly, I will take a deeper dive into the engineering constraints and potential edge cases)

```
.
├── README.md
├── docker-compose.yml
├── app/
│   ├── main.py          # FastAPI
│   ├── models.py        # SQLAlchemy
│   ├── scraper.py       # Playwright engine
│   ├── summarizer.py    # LLM integration
│   ├── schemas.py       # Pydantic
│   └── utils/
```

```
├── tests/
│   ├── test_scraper.py
│   └── test_api.py
└── slides/
    └── deck.pptx        # 10-slide brief
```

---

5 PowerPoint (10-slide outline) - 10 slides is the goal…but it can vary

1. **Problem & Objective**
2. **Target Site & Ethics Check**
3. **High-Level Architecture Diagram**
4. **Tech Stack & Why**
5. **Scraping Strategy** (Playwright demo trace gif)
6. **Prompt Engineering** (sample IO)
7. **API Contract** (Swagger screenshot)
8. **Observability & Security**
9. **Limitations & Next Steps**
10. **Results Snapshot + Cost Breakdown**

---

6 Delivery & Timeline (6 hrs) - can vary, but 6 hours is doable (or at least the goal here)

Time	Activity
0:00-1:00	Repo scaffold (Poetry, Docker), Playwright boilerplate.
1:00-2:30	Scraper implementation + tests (10 products).

2:30-3:30	LLM prompt + summarizer service.
3:30-4:15	SQLAlchemy models, FastAPI endpoints.
4:15-5:00	Docker compose, GitHub Actions CI, unit tests.
5:00-5:30	Cloud Run deploy (optional).
5:30-6:00	PPT creation, README polish, final QA.

---

7 Future Enhancements (roadmap teaser)

- **Vector store** for semantic search across product corpus.
- **RabbitMQ/Kafka** for true distributed job scaling.
- **LangChain agent** to auto-adjust scraper on DOM drift.
- **User-facing dashboard** (Next.js) consuming `/products` API.

These are just my thoughts on how I'm going to approach it… in terms of the time limit and delivery time line, it can change depending on how fast certain parts go by. Hopefully I can better understand the engineering constraints, long term objectives, or any assumptions depending on the real world application (if I have to deploy it to market and have some real use case for it,... but in this case its just to assess my knowledge). But for now, I can work on this project specifically due to the strong guidelines provided.