Prometheus AI — Full Stack Notes (A–Z)

These notes take you **from zero to a complete AI system**: local LLM, RAG, live web search, backend API, frontend UI, and deployment. Think **ChatGPT + Perplexity**, built by you.

---

0. What You Are Building (Big Picture)

Prometheus AI is a hybrid AI system that: - Uses a **local LLM** (Ollama + LLaMA) - Searches the **live web** (SerpAPI) - Reads **private documents** (RAG with Chroma) - Shows **answers + sources** - Has a **modern chat UI** - Can be deployed (Vercel + backend host)

Architecture:

```
User
 ↓
Frontend (Next.js / UI)
 ↓ HTTP
Backend API (FastAPI)
 ↓
 ├── LLM (Ollama)
 ├── RAG (Chroma + Embeddings)
 └── Web Search (SerpAPI)
 ↓
Final Answer + Sources
```

---

1. Core Concepts (Must-Know Theory)

1.1 LLM (Large Language Model)

- A neural network trained on massive text
- Examples: LLaMA, Qwen, DeepSeek
- Ollama lets you **run them locally**

1.2 Ollama

- Local LLM runtime
- Exposes a server at `localhost:11434`
- Handles model download, quantization, inference

Common commands:

```
ollama pull llama3.2
ollama run llama3.2
ollama list
ollama rm modelname
```

1.3 Embeddings

- Convert text → vectors (numbers)
- Used for similarity search
- Example model: `nomic-embed-text`

1.4 Vector Database (Chroma)

- Stores embeddings
- Finds relevant chunks for a question
- Core of **RAG**

1.5 RAG (Retrieval-Augmented Generation)

- Retrieve relevant documents
- Inject them into LLM prompt
- LLM answers using **your data**

1.6 Web Search Augmentation

- Live internet results
- Used for current / factual info
- Example: SerpAPI

---

2. Environment Setup

2.1 System Requirements

- macOS / Linux / Windows
- Python 3.9+
- 8–16 GB RAM (MacBook Air OK)
- Disk space: 15–25 GB recommended

2.2 Install Ollama

macOS:

```
brew install ollama
ollama serve
```

Verify:

```
ollama run llama3.2
```

3. Backend — Complete Notes

3.1 Why Backend?

Frontend should **not**: - Talk directly to Ollama - Store secrets (API keys) - Handle heavy logic

Backend = AI brain.

3.2 Backend Tech Stack

- FastAPI (API server)
- LangChain (LLM orchestration)
- Ollama (LLM runtime)
- ChromaDB (vector DB)
- SerpAPI (web search)

4. Backend Folder Structure

```
prometheus-backend/
|── main.py          # API entry
|── llm.py           # LLM config
|── rag.py           # Vector DB + retrieval
|── web.py           # Web search
|── ingest.py        # PDF ingestion
|── requirements.txt
|── .env
```

5. Backend Dependencies

```
fastapi
uvicorn
langchain
langchain-ollama
langchain-chroma
chromadb
python-dotenv
requests
pypdf
```

Install:

```
pip install -r requirements.txt
```

6. LLM Module (`llm.py`)

Purpose: - Central place to configure LLM

Concepts: - `temperature`: creativity vs accuracy - `model`: which LLaMA variant

```
from langchain_ollama import OllamaLLM

llm = OllamaLLM(
    model="llama3.2",
    temperature=0.2
)
```

7. RAG Module (`rag.py`)

Purpose: - Load vector DB - Retrieve relevant chunks

Concepts: - Chunking - Similarity search

```
from langchain_ollama import OllamaEmbeddings
from langchain_chroma import Chroma

embeddings = OllamaEmbeddings(model="nomic-embed-text")

vectorstore = Chroma(
    persist_directory="./chroma_db",
    embedding_function=embeddings
)

def retrieve(query):
    docs = vectorstore.similarity_search(query, k=3)
    return "\n".join([d.page_content for d in docs])
```

8. Ingesting PDFs (`ingest.py`)

Purpose: - Convert PDFs → chunks → embeddings

```
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from rag import vectorstore

loader = PyPDFLoader("notes.pdf")
```

```
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(
    chunk_size=800,
    chunk_overlap=100
)

splits = splitter.split_documents(docs)
vectorstore.add_documents(splits)
```

Run once.

---

9. Web Search Module (`web.py`)

Purpose: - Fetch live internet info

```
import os, requests

SERP_KEY = os.getenv("SERPAPI_API_KEY")

def web_search(query):
    r = requests.get(
        "https://serpapi.com/search.json",
        params={
            "q": query,
            "engine": "google",
            "api_key": SERP_KEY,
            "num": 5
        },
        timeout=15
    )
    return r.json().get("organic_results", [])
```

---

10. Main API (`main.py`)

Purpose: - Orchestrate everything

Flow: 1. Receive question 2. Retrieve RAG context 3. Fetch web context 4. Build prompt 5. Ask LLM

```
from fastapi import FastAPI
from pydantic import BaseModel
from llm import llm
from rag import retrieve
from web import web_search
```

```python
app = FastAPI()

class Query(BaseModel):
    question: str

@app.post("/ask")
def ask(data: Query):
    q = data.question

    rag_context = retrieve(q)
    web_results = web_search(q)

    web_context = "\n".join(
        [f"{r['title']}: {r['snippet']}" for r in web_results]
    )

    prompt = f"""
You are PROMETHEUS AI.

PRIVATE KNOWLEDGE:
{rag_context}

LIVE WEB:
{web_context}

QUESTION:
{q}

Answer clearly and cite sources.
"""

    answer = llm.invoke(prompt)

    return {
        "answer": answer,
        "sources": [r["link"] for r in web_results]
    }
```

---

11. Running Backend

```
uvicorn main:app --reload
```

Test:

```
curl -X POST http://localhost:8000/ask
  -H "Content-Type: application/json"
  -d '{"question":"What is quantum computing?"}'
```

---

12. Frontend — Complete Notes

12.1 Why Next.js?

- React-based
- Server-side rendering
- Perfect for Vercel

12.2 Frontend Stack

- Next.js
- Tailwind CSS
- Fetch API
- Streaming UI (later)

---

13. Frontend Architecture

```
prometheus-ui/
|── app/
|   ├── page.tsx
|   ├── components/
|   |   ├── ChatBox.tsx
|   |   ├── Message.tsx
|   |   └── Sources.tsx
|── styles/
|── tailwind.config.js
```

---

14. Chat Flow (Frontend Logic)

1. User types message
2. Frontend calls `/ask`
3. Backend returns answer + sources
4. UI renders message

---

15. Deployment Notes

Frontend (Vercel)

- Push repo
- Set API URL env
- Deploy

Backend (Railway / EC2)

- Needs Python + Ollama access
- Store `.env`

⚠️ Ollama **cannot run on Vercel**.

---

16. Scaling & Enhancements

- Streaming tokens
- Query classification (web vs RAG)
- Memory (conversation history)
- Auth
- Rate limiting

---

17. Mental Model (Final)

Prompt = Brain RAG = Memory Web = Eyes LLM = Reasoning Backend = Nervous system Frontend = Face

---

18. You Are Here

You now have **full A–Z system knowledge**.

Next steps are execution, polish, and deployment.

End of Notes