

Prometheus AI — Advanced Sections (Separate Deep Dive)

This document continues from the A-Z notes and covers **production-grade features**, explained step-by-step.

PART 1 Step-by-Step Frontend Code (Next.js UI)

1.1 Why Next.js for Prometheus AI

- React-based
- Fast rendering
- Easy Vercel deployment
- Clean separation of UI + backend

1.2 Frontend Stack

- Next.js (App Router)
- TypeScript
- Tailwind CSS
- Fetch API

1.3 Project Setup

```
npx create-next-app@latest prometheus-ui  
cd prometheus-ui  
npm install
```

Enable Tailwind (if not already):

```
npm install -D tailwindcss postcss autoprefixer  
npx tailwindcss init -p
```

1.4 Folder Structure

```
app/  
  └── page.tsx  
  └── components/  
      |   └── ChatBox.tsx  
      |   └── Message.tsx
```

```
    └── Sources.tsx  
    └── globals.css
```

1.5 Chat UI Logic

Core idea: - Store messages in state - Call backend `/ask` - Render answer + sources

1.6 ChatBox.tsx

```
'use client'  
import { useState } from 'react'  
  
export default function ChatBox() {  
  const [messages, setMessages] = useState<any[]>([])  
  const [input, setInput] = useState('')  
  
  async function send() {  
    const userMsg = { role: 'user', text: input }  
    setMessages(m => [...m, userMsg])  
  
    const res = await fetch('http://localhost:8000/ask', {  
      method: 'POST',  
      headers: { 'Content-Type': 'application/json' },  
      body: JSON.stringify({ question: input })  
    })  
  
    const data = await res.json()  
  
    setMessages(m => [...m, { role: 'ai', text: data.answer, sources: data.sources }])  
    setInput('')  
  }  
  
  return (  
    <div className="max-w-3xl mx-auto p-6">  
      {messages.map((m, i) => (  
        <div key={i} className="mb-4">  
          <b>{m.role === 'user' ? 'You' : 'Prometheus'}:</b>  
          <p>{m.text}</p>  
        </div>  
      ))}  
  
      <input  
        value={input}  
        onChange={e => setInput(e.target.value)}  
        className="w-full p-3 text-black rounded"  
        placeholder="Ask anything...">  
    </div>  
  )  
}
```

```
    />
    <button onClick={send} className="mt-3 px-4 py-2 bg-blue-500
rounded">Send</button>
  </div>
)
}
```

PART 2 Streaming Responses (Real-Time Typing)

2.1 Why Streaming Matters

- Feels alive
 - Faster perceived response
 - ChatGPT-like UX
-

2.2 Backend Streaming (FastAPI)

```
from fastapi.responses import StreamingResponse

@app.post('/ask-stream')
def ask_stream(data: Query):
    def generate():
        for chunk in llm.stream(prompt):
            yield chunk
    return StreamingResponse(generate(), media_type='text/plain')
```

2.3 Frontend Streaming Fetch

```
const res = await fetch('/ask-stream', { method: 'POST', body:
JSON.stringify({ question }) })
const reader = res.body.getReader()
let text = ''

while (true) {
  const { value, done } = await reader.read()
  if (done) break
  text += new TextDecoder().decode(value)
  setAnswer(text)
}
```

PART 3 Auto Web vs RAG Decision Logic

3.1 Why This Is Important

- Saves tokens
 - Faster
 - Smarter answers
-

3.2 Classification Prompt

```
def decide_route(question):
    prompt = """
Classify the question:
1 = needs live web
2 = needs private knowledge
3 = needs both

Question: {question}
Answer only 1, 2, or 3.
"""
    return llm.invoke(prompt).strip()
```

3.3 Smart Orchestration

```
route = decide_route(q)

rag_context = retrieve(q) if route in ['2', '3'] else ''
web_context = web_search(q) if route in ['1', '3'] else ''
```

This is **real AI reasoning**, not hard-coding.

PART 4 Deployment Guide (Vercel + Railway)

4.1 Deployment Split

Layer	Platform
Frontend	Vercel
Backend	Railway / EC2
Ollama	Same backend server

4.2 Backend Deployment (Railway)

Steps: 1. Push backend repo 2. Create Railway project 3. Add env vars 4. Install Ollama on server 5. Start FastAPI

4.3 Frontend Deployment (Vercel)

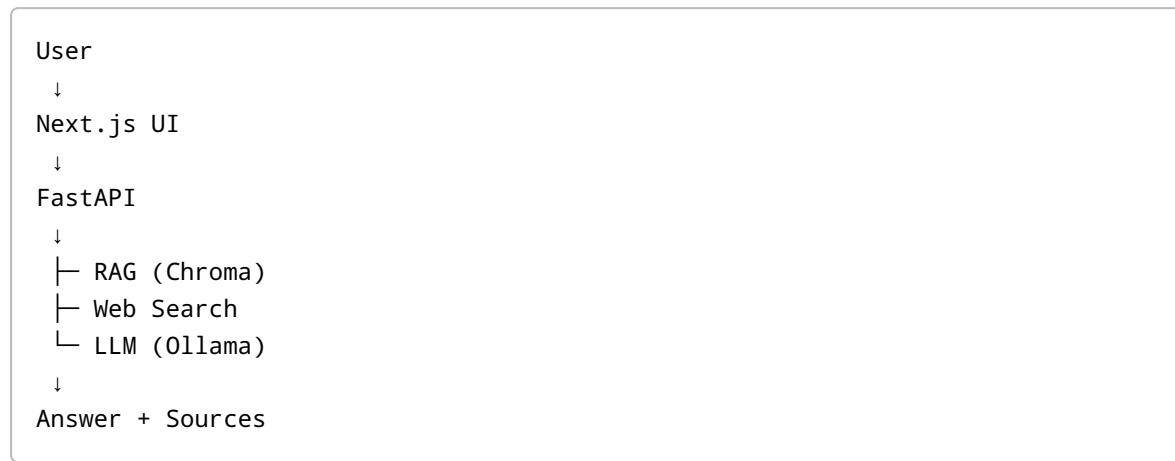
Steps: 1. Push frontend repo 2. Import in Vercel 3. Set backend URL env 4. Deploy

PART 5 Interview-Ready Notes + Mental Diagrams

5.1 One-Line Explanations

- **RAG:** Retrieval + Generation using private data
 - **Vector DB:** Similarity search over embeddings
 - **Streaming:** Token-by-token response
 - **Ollama:** Local LLM runtime
-

5.2 Architecture Diagram (Mental)



5.3 Interview Questions You Can Now Answer

- How does RAG differ from fine-tuning?
 - Why vector DB instead of SQL?
 - How do you reduce hallucinations?
 - How do you deploy LLM systems?
 - Why not put LLM on frontend?
-

FINAL STATE

You now have: - Full stack AI system - Production architecture - Interview-level understanding - Real deployable product

This is **senior-level AI engineering knowledge.**

End of Advanced Sections