Lecture 02: Write our first code

Part 1: What LLMs Actually Do (First Principles)

The Core Truth: Token Prediction

LLMs don't think, reason, or understand. They predict the next token.

What's a token?

- A piece of text (word, part of word, or character)
- Example: "Hello world" → ["Hello", " world"] (2 tokens)
- Example: "strawberry" → might be ["straw", "berry"] or just ["strawberry"]

What the model does:

```
Given: "The capital of France is" Model predicts: "Paris" (most likely next t
oken based on training)
```

That's it. Every single thing an LLM does is just repeating this process over and over.

Part 2: What is "Thinking" or "Reasoning"? (Spoiler: It's Not Real Thinking)

The Marketing vs Reality

Marketing says: "The model thinks through the problem step by step"

Reality: The model generates more tokens that *look like* thinking before giving the final answer.

Example: Simple Math

Question: "What is 547 + 832?"

Without "thinking" (direct prediction):

```
Input: "What is 547 + 832?" ↓ Model predicts: "1379"
```

With "thinking" (more token predictions):

```
Input: "What is 547 + 832?" ↓ Model predicts these tokens (hidden from you):
"Let me add these step by step" "547 + 832" "7 + 2 = 9" "40 + 30 = 70" "500 +
800 = 1300" "Total: 1300 + 70 + 9 = 1379" ↓ Final output: "1379"
```

What's Really Happening?

At each step, the model asks itself: "**What's the most likely next token?**"

```
Token 1: "Let" ← Most likely after a math question Token 2: " me" ← Most like
ly after "Let" Token 3: " add" ← Most likely after "Let me" Token 4: " these"
← Most likely after "Let me add" ...and so on
```

Important: The model is NOT:

- Actually calculating 547 + 832
- Understanding what numbers mean
- Following a math algorithm

It's just predicting tokens that *look like* someone solving the problem.

Part 3: Why "Thinking" Sometimes Helps

The Pattern Matching Effect

Each token you generate changes what can come next.

Direct prediction:

```
P("1379" | "What is 547 + 832?") = 60% chance P("1478" | "What is 547 + 83
2?") = 30% chance
```

After generating intermediate steps:

```
P("1379" | "What is 547 + 832? + Let me add + 7+2=9 + 40+30=70 + 500+800=130
0") = 95% chance P("1478" | same context) = 1% chance
```

The intermediate tokens make the correct answer much more probable!

Part 4: Real Example - Counting Letters

Question: "How many times does 'r' appear in 'strawberry'?"

Correct answer: 3 times (strawberry)

Without Thinking:

```
Model sees: "How many times does 'r' appear in 'strawberry'?" ↓ Predicts: "2"
or "3" (whatever pattern it learned)
```

With Thinking:

```
Model generates: "Let me spell it out" "s-t-r-a-w-b-e-r-r-y" "I see 'r' at po
sition 3" "I see 'r' at position 8" "I see 'r' at position 9" "Total: 3"
```

But Here's the Truth:

The model is NOT actually:

1. Splitting "strawberry" into characters: `['s','t','r',...]`
2. Looping through the array

3. Counting matches

It's predicting what tokens would appear if someone counted letters!

Why It Can Still Fail:

Try: "How many times does 'r' appear in 'rrrrrr'?"

The model might get confused because:

- It's predicting patterns, not executing code
- Unusual inputs weren't common in training data
- It's just guessing what answer *looks right*

Part 5: When to Use "Thinking" Models

✅ Use Thinking For:

1. Multi-step problems:

```
"A train leaves station A at 60mph. Another train leaves station B (120 miles away) at 40mph. When do they meet?" // Thinking helps: forces step-by-step calculation
```

2. Code debugging:

```
"Why does this code have a bug?" [complex code with closure issue] // Thinking helps: traces execution step by step
```

3. Complex explanations:

```
"Explain closures using first principles" // Thinking helps: structures the explanation systematically
```

4. Planning:

```
"I have $5000, 2 months, and want to visit 5 countries. Help plan." // Thinki
ng helps: breaks down constraints and calculates
```

❌ Don't Use Thinking For:

1. Simple facts:

```
"What's the capital of France?" // Direct: "Paris" ✅ // Thinking: "Let me th
ink... France... capital... Paris" ✅ // Same answer, wasted tokens and mone
y!
```

2. Creative writing:

```
"Write a poem about sunset" // Thinking doesn't improve creativity
```

3. Simple definitions:

```
"What is a variable?" // No multi-step logic needed
```

Part 6: Google AI Studio - Getting Started

Setup

1. Install the package:

```
npm install @google/genai
```

2. Get API Key:

- Go to Google AI Studio
- Create new API key
- Copy it

3. Set up environment variable:

```
# Create .env file GEMINI_API_KEY=your_api_key_here
```

4. Install dotenv:

```
npm install dotenv
```

Part 7: Basic Chat Implementation

Simple Single Message

```
import 'dotenv/config'; import { GoogleGenAI } from "@google/genai"; const ai = new GoogleGenAI({}); async function main() { const response = await ai.models.generateContent({ model: "gemini-2.5-flash", contents: "Explain what a variable is in programming", }); console.log(response.text); } await main();
```

What Happens:

1. You send ONE message
2. AI responds
3. **AI forgets everything** (no memory!)

Part 8: Multi-Turn Conversations (The Right Way)

The Problem: No Memory

```
// Message 1 await ai.models.generateContent({ contents: "My name is Rohit", }); // Output: "Nice to meet you, Rohit!" // Message 2 await ai.models.generateContent({ contents: "What's my name?", }); // Output: "I don't know your name" ❌
```

Why? Each API call is completely independent. The model has NO memory.

The Solution: Send History

You must send the **entire conversation history** with each new message.

```
const history = []; // Message 1 history.push({ role: "user", parts: [{ text: "My name is Rohit" }] }); let response = await ai.models.generateContent({ model: "gemini-2.5-flash", contents: history, }); history.push({ role: "model", parts: [{ text: response.text }] }); // Message 2 - NOW includes history history.push({ role: "user", parts: [{ text: "What's my name?" }] }); response = await ai.models.generateContent({ model: "gemini-2.5-flash", contents: history, }); console.log(response.text); // "Your name is Rohit!" ✅
```

Part 9: Interactive Chat with User Input

Using readline-sync (Simple Way)

Install:

```
npm install readline-sync
```

Code:

```
import 'dotenv/config'; import { GoogleGenAI } from "@google/genai"; import r
eadlineSync from "readline-sync"; const ai = new GoogleGenAI({}); async funct
ion main() { console.log("Chat started! Type 'exit' to quit.\n"); // Store co
nversation history const history = []; while (true) { // Get user input const
userMessage = readlineSync.question("You: "); // Exit condition if (userMessa
ge.toLowerCase() === "exit") { console.log("Goodbye!"); break; } // Add user
message to history history.push({ role: "user", parts: [{ text: userMessage
}], }); // Send entire history to API const response = await ai.models.genera
teContent({ model: "gemini-2.5-flash", contents: history, }); const aiRespons
e = response.text; console.log(`AI: ${aiResponse}\n`); // Add AI response to
history history.push({ role: "model", parts: [{ text: aiResponse }], }); } }
await main();
```

How It Works:

First message:

```
Sent to API: [ { role: "user", parts: [{ text: "Hello" }] } ]
```

Second message:

```
Sent to API: [ { role: "user", parts: [{ text: "Hello" }] }, { role: "model",
parts: [{ text: "Hi! How can I help?" }] }, { role: "user", parts: [{ text:
"I have 2 dogs" }] } ]
```

Third message:

```
Sent to API: [ { role: "user", parts: [{ text: "Hello" }] }, { role: "model",
parts: [{ text: "Hi! How can I help?" }] }, { role: "user", parts: [{ text:
"I have 2 dogs" }] }, { role: "model", parts: [{ text: "That's nice!" }] }, {
role: "user", parts: [{ text: "How many paws?" }] } ]
```

Notice: The history keeps growing. You send EVERYTHING every time.

Part 10: Using Chat Sessions (Easier Method)

Google provides a helper that manages history automatically:

```
import 'dotenv/config'; import { GoogleGenAI } from "@google/genai"; import r
eadlineSync from "readline-sync"; const ai = new GoogleGenAI({}); async funct
ion main() { console.log("Chat started! Type 'exit' to quit.\n"); // Create c
hat session - handles history automatically const chat = ai.chats.create({ mo
del: "gemini-2.5-flash", }); while (true) { const userMessage = readlineSync.
question("You: "); if (userMessage.toLowerCase() === "exit") { console.log("G
oodbye!"); break; } // Send message - history is managed automatically! const
response = await chat.sendMessage({ message: userMessage, }); console.log(`A
I: ${response.text}\n`); } } await main();
```

What's different?

- `chat.sendMessage()` automatically adds messages to history
- You don't manually manage the `history` array
- Cleaner code!

Part 11: System Instructions

What Are System Instructions?

A special message that tells the AI **how to behave** throughout the entire conversation.

Think of it like:

- **Regular messages:** The actual conversation
- **systemInstruction:** The personality/rules the AI should follow

Example 1: JavaScript Tutor

```
const chat = ai.chats.create({ model: "gemini-2.5-flash", systemInstruction: `You are a JavaScript tutor for beginners. - Explain concepts using simple an alogies - Always provide code examples - Ask if the student understands befor e continuing - Never use complex jargon without explaining it`, });
```

Result:

```
Student: "What is async/await?" AI: "Think of async/await like ordering food at a restaurant. You place your order (async function), get a buzzer (promis e), and can do other things while you wait (await)... Here's a code example: [provides example] Does that make sense? Would you like another example?"
```

Example 2: Code Reviewer

```
const chat = ai.chats.create({ model: "gemini-2.5-flash", systemInstruction: `You are a senior code reviewer. - Point out bugs and security issues - Sugge st better approaches - Explain WHY something is wrong, not just WHAT - Be con structive, not harsh`, });
```

Example 3: Pirate Bot (Fun Example!)

```
const chat = ai.chats.create({ model: "gemini-2.5-flash", systemInstruction: "You are a pirate. Always talk like a pirate.", });
```

Result:

```
User: "How many days in a week?" AI: "Arrr matey! There be 7 days in a week, by Blackbeard's beard! ⚓"
```

Part 12: Thinking Configuration

How to Control Thinking

```
const response = await ai.models.generateContent({ model: "gemini-2.5-flash", contents: "What is 547 + 832?", config: { thinkingConfig: { thinkingBudget: 0, // No thinking - direct answer // thinkingBudget: 500 // 500 tokens for thinking // thinkingBudget: 1000 // 1000 tokens for thinking }, }, });
```

What thinkingBudget means:

- **0:** Direct answer, no intermediate steps
- **500:** Generate up to 500 intermediate tokens before answering
- **1000:** Generate up to 1000 intermediate tokens

When to Use Each:

```
// Simple question - no thinking needed thinkingBudget: 0 "What does 'const' mean?" // Medium complexity - some thinking helps thinkingBudget: 500 "Debug this code with a closure issue" // Complex problem - more thinking helps thinkingBudget: 1000 "Explain the event loop using first principles with examples"
```

Part 13: Understanding Tokens and Cost

What Are Tokens?

Pieces of text the model processes:

```
"Hello world" → ["Hello", " world"] = 2 tokens "JavaScript" → ["Java", "Script"] or ["JavaScript"] = 1-2 tokens
```

What Gets Counted?

Every API call includes:

1. **systemInstruction** (sent EVERY time!)
2. **Conversation history** (grows with each message)
3. **New user message**
4. **AI response** (output tokens)

Example Token Count:

```
const chat = ai.chats.create({ model: "gemini-2.5-flash", systemInstruction: `You are a helpful tutor...`, // 100 tokens }); // Message 1 await chat.sendMessage({ message: "Hello" }); // 2 tokens // Sent to API: // systemInstruction: 100 tokens // history: 0 tokens // message: 2 tokens // Total input: 102 tokens // Response: ~20 tokens // Total: 122 tokens // Message 2 await chat.sendMessage({ message: "Explain variables" }); // 3 tokens // Sent to API: // systemInstruction: 100 tokens ← Sent AGAIN! // history: 22 tokens (previous exchange) // message: 3 tokens // Total input: 125 tokens // Response: ~50 tokens // Total: 175 tokens
```

Key point: systemInstruction is sent with EVERY request, so keep it short!

Cost Optimization Tips:

❌ **Bad (wasteful systemInstruction):**

```
systemInstruction: `You are my programming tutor and you should always explain things clearly with examples and you should use analogies and you should ask if I understand and you should be patient and you should use simple language and...` // 200+ tokens sent EVERY time!
```

✅ **Good (concise systemInstruction):**

```
systemInstruction: `Programming tutor: clear explanations, use analogies, pro
vide examples, check understanding.` // ~20 tokens sent every time
```

Part 14: Complete Example - Interactive Learning Assistant

```
import 'dotenv/config'; import { GoogleGenAI } from "@google/genai"; import r
eadlineSync from "readline-sync"; const ai = new GoogleGenAI({}); async funct
ion main() { console.log("╔═══════════════════════════════════╗"); console.l
og("║ Programming Learning Assistant ║"); console.log("║ Type 'exit' to quit
 ║"); console.log("╚═══════════════════════════════════╝\n"); const chat = a
i.chats.create({ model: "gemini-2.5-flash", systemInstruction: `You are a pro
gramming tutor. - Use first-principles thinking - Explain WHY, not just WHAT
- Provide practical code examples - Keep explanations concise`, }); while (tr
ue) { const userMessage = readlineSync.question("\n You: "); if (userMessage.
toLowerCase() === "exit") { console.log("\n👋 Keep learning! Goodbye!"); brea
k; } if (!userMessage.trim()) { continue; // Skip empty messages } try { // S
how loading process.stdout.write("AI: Thinking...\r"); // Decide thinking bud
get based on question complexity const needsThinking = userMessage.length > 5
0 || userMessage.includes("why") || userMessage.includes("explain") || userMe
ssage.includes("debug"); const response = await chat.sendMessage({ message: u
serMessage, config: { thinkingConfig: { thinkingBudget: needsThinking ? 500 :
0, }, }, }); // Clear loading indicator process.stdout.write("\r\x1b[K"); con
sole.log(`AI: ${response.text}`); } catch (error) { console.error(`\n❌ Erro
r: ${error.message}`); } } } await main();
```

Part 15: Key Takeaways

What You MUST Remember:

1. **LLMs predict tokens, they don't think or understand**
   - Every output is "what's the most likely next token?"
   - No actual reasoning, just pattern matching

2. **"Thinking" = More Token Predictions**
    - Generates intermediate steps before final answer
    - Helps for complex problems, useless for simple ones
    - Costs more tokens and money
3. **LLMs Have No Memory**
    - Each API call is independent
    - YOU must send conversation history every time
    - History grows → costs increase
4. **systemInstruction is Sent Every Time**
    - Keep it short and concise
    - Long instructions = wasted money
    - Gets sent with EVERY single message
5. **Use Thinking Strategically**
    - Complex problems: thinkingBudget > 0
    - Simple questions: thinkingBudget = 0
    - Don't waste tokens on simple tasks

The Mental Model:

```
User types message ↓ Your code adds to history ↓ Send to API: systemInstructi
on + full history + new message ↓ API predicts tokens one by one ↓ Returns co
mplete response ↓ Add response to history ↓ Repeat
```

Part 16: Practice Exercises

Exercise 1: Basic Chat

Create a simple chatbot that:

- Takes user input

- Remembers previous messages
- Exits when user types "quit"

Exercise 2: Math Tutor

Create a math tutoring bot that:

- Uses systemInstruction to act as a patient tutor
- Uses thinking for complex problems
- Shows step-by-step solutions

Exercise 3: Code Reviewer

Create a code review assistant that:

- Takes code as input
- Points out issues
- Suggests improvements
- Uses thinking to analyze complex code

Exercise 4: Token Optimization

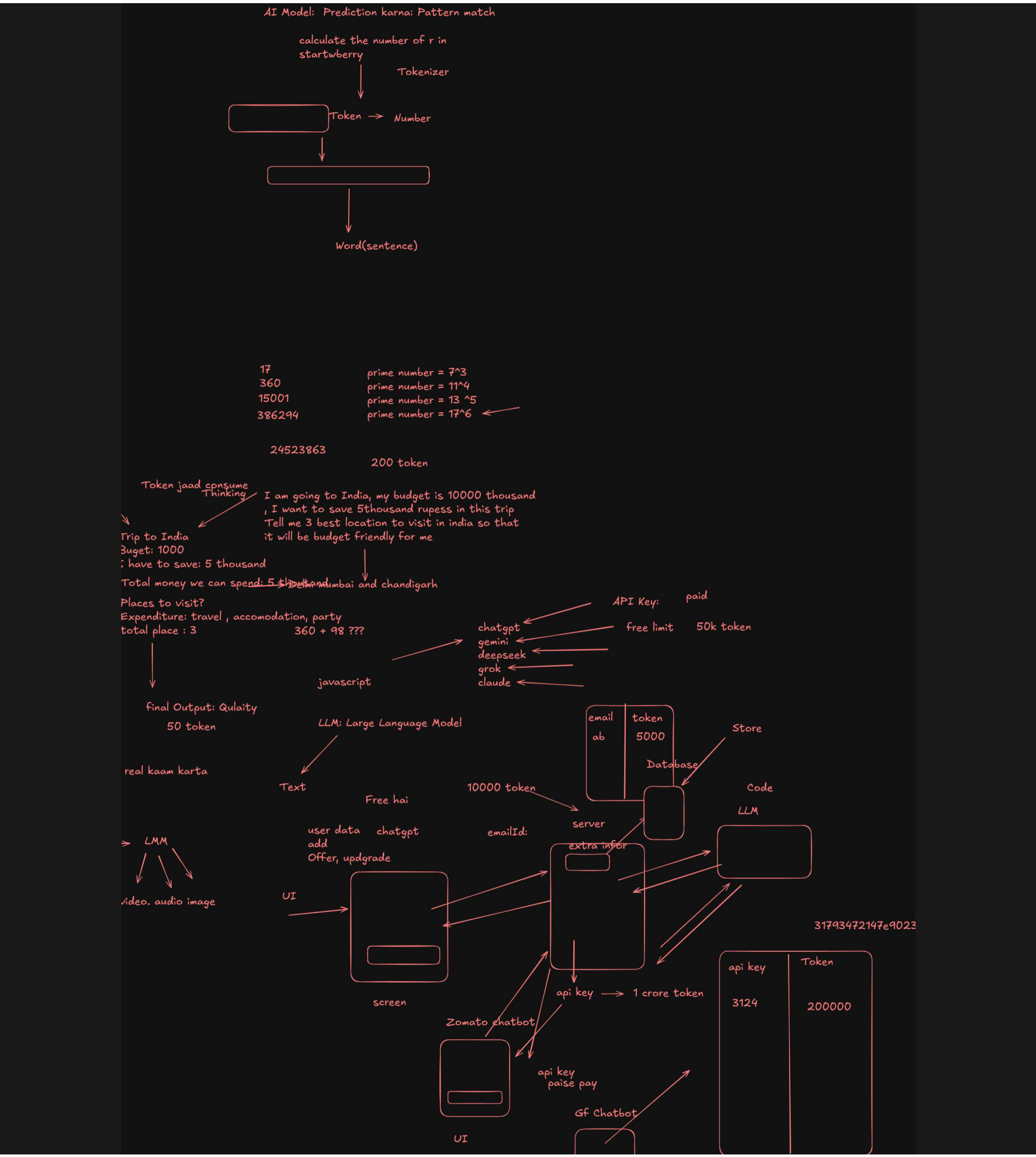
Take a working chatbot and:

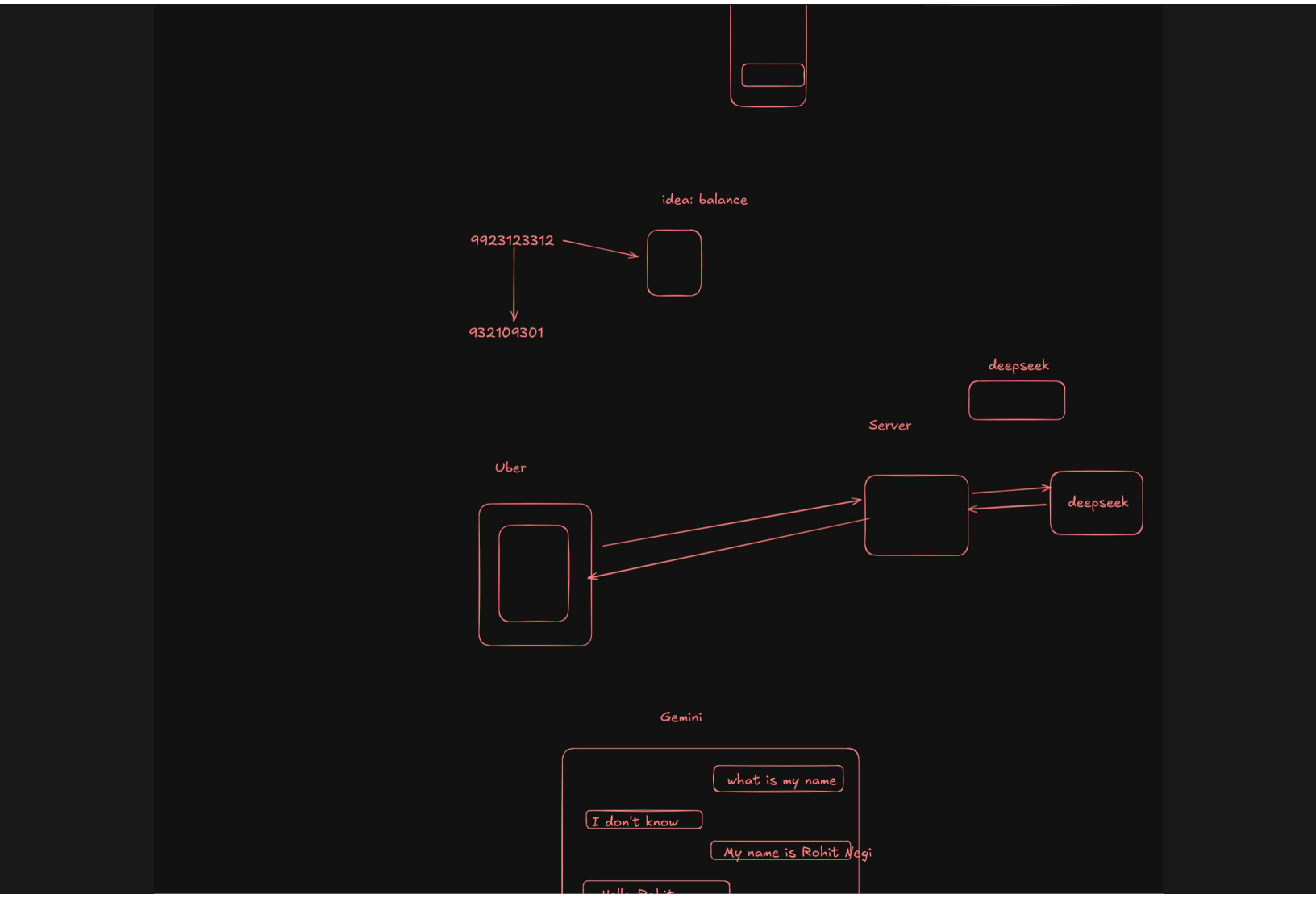
- Measure token usage
- Optimize systemInstruction
- Implement history trimming
- Compare costs before/after

Additional Resources

- Google AI Studio Documentation: https://ai.google.dev/
- Token counting tools: Use online tokenizers to understand token usage
- Practice projects: Build real applications to solidify understanding

---


AI Model: Prediction karna: Pattern match
calculate the number of r in startwberry
Tokenizer
Token → Number
Word(sentence)
17
360
15001
386294
prime number = 7^3
prime number = 11^4
prime number = 13 ^5
prime number = 17^6
24523863
200 token
Token jaad consume
Thinking
I am going to India, my budget is 10000 thousand , I want to save 5thousand rupess in this trip Tell me 3 best location to visit in india so that it will be budget friendly for me
Trip to India
Buget: 1000
I have to save: 5 thousand
Total money we can spend: 5 thousand
Delhi, mumbai and chandigarh
Places to visit?
Expenditure: travel , accomodation, party
total place : 3
360 + 98 ???
final Output: Qulaity
50 token
javascript
API Key:
paid
chatgpt
gemini
deepseek
grok
claude
free limit
50k token
LLM: Large Language Model
real kaam karta
Text
Free hai
email
token
ab
5000
Database
Store
10000 token
server
emailId:
extra infor
Code
LLM
LMM
video. audio image
user data
chatgpt
add
Offer, updgrade
UI
screen
31793472147e9023
api key
Token
3124
200000
api key → 1 crore token
Zomato chatbot
api key
paise pay
Gf Chatbot
UI


idea: balance
9923123312
932109301
deepseek
Server
Uber
deepseek
Gemini
what is my name
I don't know
My name is Rohit Negi