🧠 DAYS 1–5: GENAI CORE

🟰 DAY 1[part-1] How LLMs REALLY Work (For Developers)

🎯 Goal of PART 1

Before touching any backend or code, **destroy the myth**:

❌ “LLMs understand code like humans”
✅ “LLMs predict the next token based on probability”

If this foundation is wrong, **everything later breaks** (RAG, audits, agents).

🧠 PART 1 THEORY (CORE MENTAL MODEL)

1️⃣ What an LLM ACTUALLY Does (Very Important)

❌ What beginners think

“LLM reads my code, understands logic, finds bugs”

☑️ Reality

LLM predicts the next token, again and again

That’s it.
No understanding.
No execution.

No compiler.
No debugger.

🔁 How generation works (simplified)

When you send this:

```
def add(a, b):
  return
```

The model internally does:

```
P(token | previous tokens)
```

Example:

- return → next token might be:
  - a
  - a +
  - a + b
  - None

It picks **one based on probability**.

➡️ This happens **token by token**, not line by line.

🧠 Key Truth

LLMs are **probability machines**, not reasoning engines.

Reasoning is an **illusion created by patterns**.

2️⃣ Tokens ≠ Words (Why Code Breaks Easily)

❌ Common assumption

“My function is 20 lines, that’s small”

✅ Reality

LLMs don’t see **lines**
They see **tokens**

🔹 What is a token?

A token can be:

- Part of a word
- A symbol
- A space
- A newline
- A bracket {
- A tab \t

Example:

```
for(i=0;i<n;i++){
```

This might become:

```
for | ( | i | = | 0 | ; | i | < | n | ; | i | + | + | ) | {
```

⚠️ Code = **token-heavy**

💧 Why this matters

- Long files → token explosion
- Minified JS → more tokens
- Nested logic → more context usage

➡️ This is why **“paste full repo” fails**

3️⃣ Why Temperature Matters (Especially for Code)

🌡️ Temperature = randomness control

Temperature	Behavior
0.0	Deterministic (same output)
0.2	Safe, boring
0.7	Creative
1.0+	Unstable, risky

⚠️ For CODE ANALYSIS

- High temperature = ❌ dangerous
- Low temperature = ✅ stable

Why?

Because code analysis needs:

- Consistency
- Accuracy
- Repeatability

Example

At temperature = 0.9:

“This function has SQL injection risk”
At temperature = 0.9 again:
“Looks secure, no issues found”

❌ Same input, different audit → unacceptable

4️⃣ Why Hallucinations Happen (Critical for CodeAudit)

❌ Hallucination ≠ bug

Hallucination is **expected behavior**

Why hallucinations occur:

1. Model lacks knowledge
2. Context is insufficient
3. Prompt is vague
4. Tokens exhausted
5. Model forced to “continue anyway”

LLMs are **never allowed to say “I don’t know” by default**.

They’d rather:

❌ Invent a vulnerability
❌ Invent a fix
❌ Invent a library

⚠️ In Code Review

This becomes dangerous:

- Fake vulnerabilities
- Wrong severity
- Incorrect fixes

➡️ This is **WHY RAG exists** (later days)

5️⃣ Context Window Limits (Why Long Files Fail)

🧠 Context Window = short-term memory

Example:

- GPT-3.5 → ~4k tokens
- GPT-4 → ~8k–32k tokens

What happens when you exceed it?

- Old tokens are dropped
- Important logic disappears
- Model guesses missing context

❌ Common beginner mistake

"I pasted the full backend, why wrong output?"

Because:

- Imports got dropped
- Helper functions vanished

- Security rules missing

🧠 MINI CHEAT SHEET (PART 1)

LLMs ≠ thinking machines LLMs = next-token predictors Code = token expensive High temperature = unstable audits Low temperature = reliable outputs Hallucination = expected behavior Context window = hard limit, not suggestion

✅ What you Understood After PART 1

✔ Why LLMs feel "smart"
✔ Why they fail on long code
✔ Why outputs change
✔ Why blind trust is dangerous

🧪 DAY 1[Part-2] Hands-On: Breaking an LLM (Temperature + Context)

🎯 Goal of PART 2

- Same code ≠ same output
- Temperature changes behavior

- Long code causes failure

🧱 STEP 0 — Project Structure (IMPORTANT)

Create a fresh folder anywhere:

```
day1-llm-basics/
│
├── main.py
├── requirements.txt
└── README.md    (optional)
```

👉 **We are NOT using DB today**
Day-1 is **pure LLM behavior**, no distractions.

📦 STEP 1 — Install Dependencies

1. Python –m venv venv
2. Source venv/bin/activate[mac] | venv/scripts/activate[windows]

requirements.txt

Paste this exactly:

```
fastapi
uvicorn
python-dotenv
google-generativeai
```

Now install:

3. pip install -r requirements.txt

🔑 STEP 2 — API Key Setup (Critical)

Create a .env file:

```
GEMINI_API_KEY=your_new_rotated_key_here
```

⚠️ Never hardcode keys in code.

🧠 STEP 3 — First LLM Call (Baseline)

main.py

Paste **everything below** :

```
from fastapi import FastAPI
from pydantic import BaseModel
import google.generativeai as genai
import os
from dotenv import load_dotenv

load_dotenv()

app = FastAPI()

genai.configure(api_key=os.getenv("GEMINI_API_KEY"))

model = genai.GenerativeModel(
   model_name="models/gemini-2.5-flash",
   generation_config={
      "temperature": 0.2
   }
)
```

```python
class CodeInput(BaseModel):
    code: str
    temperature: float = 0.2


@app.post("/analyze")
def analyze_code(payload: CodeInput):
    response = model.generate_content(
        contents=[
            "You are a senior backend engineer doing a strict code review.",
            f"Review this code and find issues:\n\n{payload.code}"
        ],
        generation_config={
            "temperature": payload.temperature
        }
    )

    return {
        "analysis": response.text
    }
```

▶️ STEP 4 — Run the Server

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

Open browser:

http://127.0.0.1:8000/docs

👉 You should see Swagger UI.

🧪 STEP 5 — Temperature Experiment (IMPORTANT)

Test Code (paste in Swagger):

```
def login(user, password):
  if user == "admin" and password == "1234":
    return True
  return False
```

🔁 Test 1 — Low Temperature

```
{
  "code": "def login(user, password):\n    if user == \"admin\" and password == \"1234\":\n        return True\n    return False",
  "temperature": 0.1
}
```

➡️ You'll get:

- Consistent issues
- Same explanation every time

🔥 Test 2 — High Temperature

```
{
  "code": "def login(user, password):\n    if user == \"admin\" and password == \"1234\":\n        return True\n    return False",
  "temperature": 0.9
}
```

You'll observe:

- Different vulnerabilities
- Sometimes exaggerated
- Sometimes imaginary

"Nothing changed except **temperature**
Yet audit results changed
This is why AI code audits must be controlled"

Why the SAME code gives DIFFERENT answers

You ran **the same code**, changed **only temperature**:

- Temperature = **0.9**
- Temperature = **0.1**

Yet:

- Both responses were **correct**
- One was **huge, aggressive, creative**
- One was **structured, conservative, predictable**

This is **not randomness by accident**.
This is **how LLMs fundamentally work**.

🧠 Core Truth (Drill This)

LLMs do NOT "analyze code"
They predict the NEXT TOKEN based on probability

Everything you saw comes from this.

1️⃣ What an LLM is *actually* doing (developer view)

Internally, the model is doing this loop:

```
Given previous tokens →
Predict probability of next token →
Sample one token →
Append →
Repeat
```

It is **not**:

- running the code
- compiling it
- checking syntax trees
- understanding intent like a human

It's doing:

```
P(next_token | all_previous_tokens)
```

That's it.

2️⃣ Temperature = how much risk the model is allowed to take

🔥 Temperature 0.9 (HIGH)

- Probability distribution is **flattened**
- Lower-probability tokens get a chance
- Model becomes:
    - verbose
    - opinionated
    - creative
    - dramatic
    - sometimes *overconfident*

That's why you saw:

- very strong language
- long explanations
- architectural suggestions
- security lectures
- extra code examples

💡 **This is dangerous for code review**
Because creativity ≠ correctness.

❄️ Temperature 0.1 (LOW)

- Probability distribution is **sharp**
- Only high-confidence tokens survive
- Model becomes:
  - conservative
  - repetitive
  - predictable
  - less creative
  - less hallucination-prone

That's why:

- response was calmer
- more structured
- fewer speculative ideas
- safer tone

💡 **This is preferred for analysis tasks**

3️⃣ Why hallucinations happen (VERY IMPORTANT)

Hallucination is NOT a bug.
It's a **mathematical outcome**.

Hallucination happens when:

1. Model is forced to answer
2. Context is incomplete or long
3. Temperature allows exploration
4. The model fills gaps with **statistically plausible tokens**

Example:

- You paste a large codebase
- Important functions fall outside context window
- Model still must predict next token

- It **guesses** patterns it has seen before

➡️ Confidently wrong output.

4️⃣ Context Window: the silent killer (you already demonstrated it)

When you pasted **short code**, results were good.

If you paste:

- large files
- multiple files
- entire repositories

What happens?

- Older tokens fall out of memory
- Model loses definitions
- Dependencies disappear
- Reasoning collapses

The model **does NOT warn you**.

It will still answer.

This is why:

"Paste entire repo → ask GPT to review"
❌ **fails silently**

5️⃣ Why code breaks more easily than text

Tokens ≠ characters ≠ words

Code has:

- symbols
- indentation
- syntax
- long identifiers

So:

- context fills faster
- truncation happens earlier
- one missing function = fake analysis

That's why **LLMs hallucinate more on code** than English text.

🧠 Mini Cheat Sheet (Day-1 students MUST remember)

```
LLMs predict tokens, not truth
Temperature controls creativity vs safety
Low temp = analysis
High temp = brainstorming
Long code breaks context
Confidence ≠ correctness
```

Q: Which output would you trust more in production?

Ans: Neither blindly — architecture matters more than prompts

This creates **demand for RAG**, which comes next.

🧩 How this sets up the rest of the course

Now they understand:

- why naïve GPT usage fails
- why hallucinations exist

- why **systems** matter more than models

Next logical step:

“How do we CONTROL LLMs?”

Which is:

- structured outputs
- chunking
- retrieval
- grounding

💥 STEP 6 — “Break the LLM” (Context Limit)

Why this feels confusing → because your brain is thinking like a compiler

But an LLM does NOT work like a compiler

First: Forget everything you know about programs

A compiler / interpreter:

- Reads the **entire file**
- Keeps **all code in memory**
- Errors if anything is missing

❌ LLMs do NONE of this

What an LLM ACTUALLY sees

An LLM sees **a sliding window of tokens**, not a file.

That's it.
No AST.
No symbol table.
No project awareness.

🧠 What is a Context Window (very important)

Think of context window as:

"How many words the model can *see at one time*"

Example (simplified):

Model context limit = 8,000 tokens

That means:

- It can ONLY "look at" ~8,000 tokens
- If you send more → **older tokens are dropped**

Now let's map this to YOUR demo

You sent this:

```
# utils.py
def helper1(): pass
def helper2(): pass
def helper3(): pass
# repeated 1000+ times
```

Let's say:

- Each function ≈ 10 tokens
- 1000 repetitions ≈ 10,000+ tokens

💥 BOOM — **context overflow**

🩸 What ACTUALLY happens inside the model

Step-by-step (this is the key part)

🟢 Step 1: Tokens start filling the window

```
[ helper1 ][ helper2 ][ helper3 ][ helper1 ][ helper2 ] ...
```

🟡 Step 2: Context limit reached

```
MAX TOKENS REACHED ❌
```

🔴 Step 3: Old tokens are DROPPED

❌ First 300 helper functions are gone
❌ Imports are gone
❌ Comments are gone

The model **never sees them anymore**.

🧠 Critical misunderstanding students have

❌ "The model read everything but forgot"

NO.

✅ The model **never sees everything at once**

Why the output looks “dumb” or “wrong”

Now the model sees something like this internally:

```
def helper2(): pass
def helper3(): pass
def helper1(): pass
def helper2(): pass
```

So it thinks:

> “Hmm… this looks like repetitive boilerplate. Probably utility functions.”

So it replies with:

- Generic advice
- Fake suggestions
- Hallucinated problems

Why hallucinations happen here

Remember Day-1 rule:

> **LLMs MUST produce an answer**
>
> They are **probability machines**, not truth machines.
>
> So when code is missing:
>
> Unknown logic → Guess likely pattern → Sound confident

That = hallucination.

⚠️ This is NOT a bug (this is crucial)

Why OpenAI / Gemini did NOT “fix” this

Because:

- Context window = memory cost 💰
- Unlimited memory = impossible
- Every model has a hard limit

Even GPT-4, Gemini, Claude — ALL have this.

Real-world consequence (this is why CodeAudit exists)

Naive approach ❌

Paste entire repo → Ask “review my code”

Result:

- Missed bugs
- Fake vulnerabilities
- False confidence

Correct mental model (this is GOLD)

```
LLM ≠ Code reviewer
LLM ≠ Static analyzer

LLM = Probabilistic text engine
```

How professionals fix this (preview, not Day-1 yet)

They **never** send full files.

They:

- Chunk code
- Summarize chunks
- Build RAG
- Track context manually

👉 That's literally what your **CodeAudit project** is about.

One-line explanation (memorize this)

"If the code doesn't fit in context, the model guesses — and guessing looks like hallucination."

🧠 MINI CHEAT SHEET (PART 2)

```
Temperature controls randomness
High temp = creative + unstable
Low temp = boring + reliable

Long code ≠ fully read code
Context window is a HARD LIMIT

LLMs guess when unsure
They don't say "I don't know"

Long input → context overflow
Context overflow → token dropping
Token dropping → missing awareness
Missing awareness → hallucination
```

After PART 2, You will be aware off

- Why same code gives different audits
- Why production GenAI must control temperature
- Why full-repo paste is useless
- Why CodeAudit NEEDS chunking + RAG