LLM Fundamentals

Agentic Cybersecurity — Module 1

What an LLM actually is, and how to pick one

Introductions

Welcome

Round the room, 1 minute each:

- Your name, team, and what you work on
- **Your AI experience**: never touched it? daily chatbot user? built agents?
- **Your security background**: pentesting, product security, secure development, something else?
- One thing you want to walk out of this class able to do

No wrong answers. This class assumes zero AI background; the spread in the room tells me where to slow down and what to skip.

Why This Matters for the Next Two Days

- You'll work solo by default, but pairing is allowed — knowing who's around you helps you find a good partner
- Day 2 is adversarial: you will attack an agent, then audit and **exploit** a deliberately vulnerable app
- The debriefs are class-wide comparisons; they get better when people talk

Rules of the room: questions any time, war stories welcome, nothing you build here leaves the lab network.

What Is an LLM?

Glorified Autocomplete

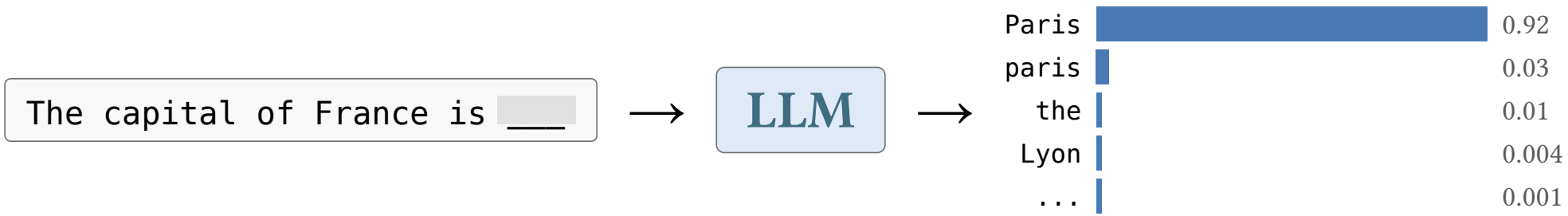
An LLM does exactly one thing: **predict the next token.**

- Input: a sequence of tokens
- Output: a probability distribution over the next token

"Glorified autocomplete" is **accurate**: no planning module, no database, no reasoning engine hiding inside.

It is also **misleading**: predicting text well enough requires modeling the world that produced it. That is where the surprising capability comes from.

The Prediction, As a Picture



One forward pass, one distribution over the whole vocabulary.

Sample one token from it, append it, run the pass again: `...France is Paris`. That loop is all "generation" is.

Why Models Make Things Up

The model always produces a plausible next token. **Plausible is not the same as true.**

- No lookup step. Facts are statistical residue in the weights.
- The model cannot reliably tell you what it doesn't know.
- Confidence of tone is uncorrelated with correctness.

Security consequence: **never treat unverified model output as fact.**
You will see this rule again in every module of this class.

Tokens

Models don't see words or characters. They see **tokens**.

- Chunks from a fixed vocabulary: `understanding` → `under` + `standing`
- Rare strings, code, base64 fragment into many tokens

Why you care:

- **Cost**: you pay per token, in and out
- **Behavior**: character-level tasks (spelling, rot13) are weirdly hard
- **Limits**: the context window is measured in tokens
- Try it live: platform.openai.com/tokenizer — Lab 1.1 exercise

The Context Window

The model's entire universe is the token sequence you send it.

- Fixed maximum length (the "context window")
- No memory between requests: **every request starts from nothing**
- "Memory" in chat apps = resending the whole conversation every turn

You will build exactly this in Lab 1.1 and see it firsthand.

Inside the Model

Weights and Parameters

A model is a set of learned numbers: **weights** (parameters).

- "70B" = 70 billion parameters
- Bigger generally means more capable, slower, more expensive
- Capability per parameter keeps improving; a modern 8B beats an old 70B on many tasks

Parameter count is a **rough** capability signal, not a ranking.

Dense vs. MoE

Dense: every parameter participates in every token.

Mixture of Experts (MoE): the model is split into "experts"; a router activates only a few per token.

- Big total parameter count, small **active** count
- Cheaper inference per token than an equivalent dense model
- Why spec sheets now say things like "671B total, 37B active"

Quantization

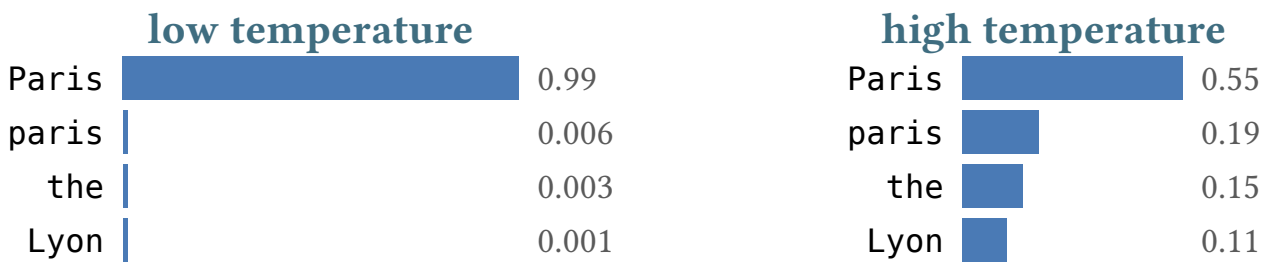
Weights are numbers; you can store them with less precision.

- 16-bit ⟶ 8-bit ⟶ 4-bit: smaller, faster, cheaper
- Costs some quality; degradation grows as precision drops
- How large open models fit on small hardware

Caveat: a hosted “model name” doesn’t tell you how it’s quantized. Two providers serving the “same” model can behave differently.

Sampling and Temperature

The model outputs a **distribution**; sampling picks the token:



- **Temperature 0** sharpens toward the top token: (near-)greedy
- **Higher temperature** flattens it: more diverse, more wrong
- Rule of thumb: low for extraction, higher for brainstorming

The API

The OpenAI API Spec

One interface won. Nearly every provider and tool speaks it.

```
{"model": "small", "messages": [
  {"role": "system", "content": "You are a security
analyst."},
  {"role": "user", "content": "Summarize this finding..."}]}
```

- **Roles**: `system`, `user`, `assistant` (+ tool messages, later)
- **Stateless**: the `messages` array is the whole conversation
- **Streaming**: tokens arrive as they're generated

What a Transcript Actually Is

`system`	“You are a security analyst.”
`user`	“Summarize this finding…”
`assistant`	“Improper session invalidation in the logout flow…”
`user`	“What’s the severity?”
`assistant`	“High — an attacker with a captured token can…”

One JSON array, growing downward. The whole stack is resent, top to bottom, on **every** call: the “conversation” lives on your side of the wire.

Stateless, Except for the Cache

There is no server-side conversation. You resend everything, every call.

- …except providers cache the **KV-state** of long, stable prefixes
- A cache hit skips recomputing everything before the first changed token
- Cached input tokens: roughly **10× cheaper**, and faster

Statelessness didn’t go away. The provider is just betting you’ll resend the same prefix, and discounting it when you do.

What Caching Does to Practice

One changed token invalidates the cache for everything after it.

- Stable system prompt **first**; volatile content last
- Append-only history: new messages go at the end, old ones stay untouched
- Don’t edit early messages mid-session — that buys the whole transcript back at full price

This is why harness vendors obsess over prefix stability: it’s the bill.

What Makes the System Prompt Special?

Almost nothing.

- It's the first message in the array, with `role: "system"` — that's the whole mechanism
- Mechanically it's just tokens, in the same sequence as everything else
- Models are **trained** to weight it more heavily (the "instruction hierarchy")

The specialness is a training-time convention, not an enforcement boundary. Hold that thought until Module 4.

Why This Matters Here

Our lab endpoint is OpenAI-compatible with Bedrock behind it.

- Your first API call in Lab 1.1 is a bare curl command — no library, no wrapper, just the HTTP request on the slide before this one
- The chat program you then write and `pi` hit the **same endpoint**
- Swap models by changing one string: `small` → `large`
- Everything above the API is replaceable; everything below is a commodity

If you understand the messages array, you understand every AI tool you'll ever use. They are all building this array for you.

The `openai` Python Library

The official client. It adds the auth header, retries, types, and streaming.

```
from openai import OpenAI
client = OpenAI(base_url=CLASS_API_URL,
api_key=CLASS_API_KEY)
resp = client.chat.completions.create(
    model="small", messages=[{"role": "user", "content": "Say
hello."}])
print(resp.choices[0].message.content)
```

Same POST you just curled. Keep this handy in Lab 1.1; it's your chat program's core.

The Model Landscape

Who Makes Models

- **Frontier labs**: Anthropic, OpenAI, Google — strongest, hosted-only
- **Open-weight**: Meta, Mistral, Qwen, DeepSeek — downloadable, self-hostable
- **Hosted open-weight**: someone else runs the open model for you

Considerations for security work: data handling, model policy, availability, and whether you can pin a version.

Benchmarks

Standardized test sets: coding, reasoning, knowledge, agentic tasks.

How they get gamed:

- **Contamination**: test questions leak into training data
- **Teaching to the test**: tuning for benchmark formats
- **Selective reporting**: labs publish the wins

A benchmark score tells you the model can do **the benchmark**. Treat it as a screening filter, not a verdict.

Reading the Leaderboards

Sites like ArtificialAnalysis aggregate scores, price, and speed.

Use them for:

- Shortlisting: rough capability tiers, cost/latency comparison

Don't use them for:

- Predicting performance on **your** task
- Differences of a few points (noise)

The fix: **build a private eval for your task.** You'll build the API skills for exactly that in Lab 1.1; the benchmark habit starts there.

Alignment

Models are trained after pretraining to be helpful, follow instructions, and refuse some requests.

Why security people care:

- Refusals hit security work disproportionately (exploit code, malware analysis)
- Alignment differs by model and version; test yours
- Alignment is a **behavioral tendency**, not a security boundary — it can be prompted around; never rely on it as a control

Picking a Model

cost × latency × capability — pick the cheapest that clears your quality bar

- High volume, simple task ⟶ small fast model
- Deep reasoning, high stakes ⟶ frontier model
- In this class: build with `small`, review and attack with `large` — that split is a preview of how you should tier in production

Working the Context Window

System Prompt vs. User Message

Same context window, different authority.

- **System prompt**: standing instructions — role, rules, output format
- **User message**: the task and the data

Rules of thumb:

- Behavior and format ⟶ system prompt; content being processed ⟶ user message
- The separation is **convention, not enforcement** — Module 4 shows what that costs

Context Engineering

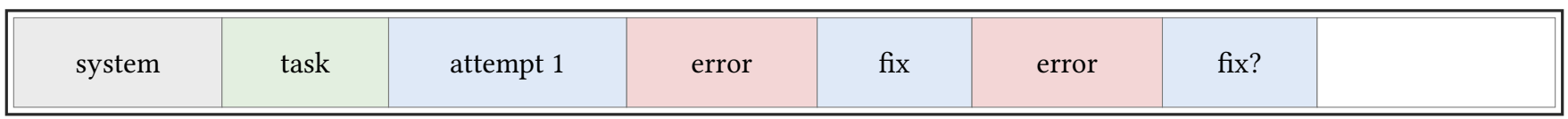
Everything in the window influences the output. Curate it.

- **Include**: the task, the rules, the data, a relevant example
- **Exclude**: stale turns, irrelevant files, dead ends
- More context is not better — irrelevant context actively hurts
- Position matters: the middle of a long window gets the least attention

Think of the window as a working set, not a scrapbook. If the model doesn't need it for **this** output, it's noise with a token bill.

Context Limits and Context Rot

Long sessions degrade before they hit the limit:



one fixed-size window — the errors never leave it

- Early mistakes keep influencing output; conflicts accumulate ("respond in JSON" … "actually just prose")

The fix: **start a fresh session** with a clean, curated prompt. Tomorrow this compounds: agents generate their own context, mistakes included.

Wrap-Up

What You Should Take Away

- An LLM predicts tokens; everything else is scaffolding around that
- Plausible ≠ true; verify anything that matters
- The context window is the model's entire world, and you control it
- Benchmarks screen; private evals decide
- Pick the cheapest model that passes **your** eval
- Curate the window; fresh session beats rotten context

Lab 1.1: curl the API raw, then write your own chat program — and keep it; it becomes an agent after lunch