AI engineering apps

IEOR 4574E001

Fall 2025

LLM APIs

What is an LLM API?

- **A network interface** (usually HTTP/JSON) to a running large language model:
  - You send text (tokens + parameters); you get generated tokens back.
- **Hosted vs. self-hosted**:
  - *Hosted/proprietary (e.g., OpenAI, Anthropic):* no infra to run; strong performance; pay-per-use; your data flows to the provider.
  - *Open/self-hosted (e.g., Llama/Mistral family):* full control & privacy; needs GPUs & MLOps.
- **Why APIs**: Lowers barrier to entry (no GPUs needed) and standardizes access.

COLUMBIA UNIVERSITY

```python
# OpenAI/Anthropic vs vLLM
backend = choose(["provider_api", "self_hosted"])
req = {
    "model": "my-model",
    "messages": chat,
    "temperature": 0.7,
    "max_tokens": 256,
    "stop": ["</end>"], "stream": True
    }
# rate limit per user/tenant
assert limiter.allow(user_id)
# respect context window
if count_tokens(req) > CTX_LIMIT: req = truncate(req)
# cost guardrail
if token_budget.exceeds(user_id, req): raise BudgetError

for attempt in range(3): # retries with backoff + jitter
  try:
      for delta in llm_stream(
            backend,
            req,
            idempotency_key=uuid4()
      ):
          client.send(delta) # SSE/WebSocket streaming to UI
      break
  except ServerError:
      continue

if not client.seen_output():
    client.send(fallback_message())  # handle empty/blocked outputs
    log_metrics(ttft, tps, latency_ms, tokens_in_out, http_codes) # observability
```
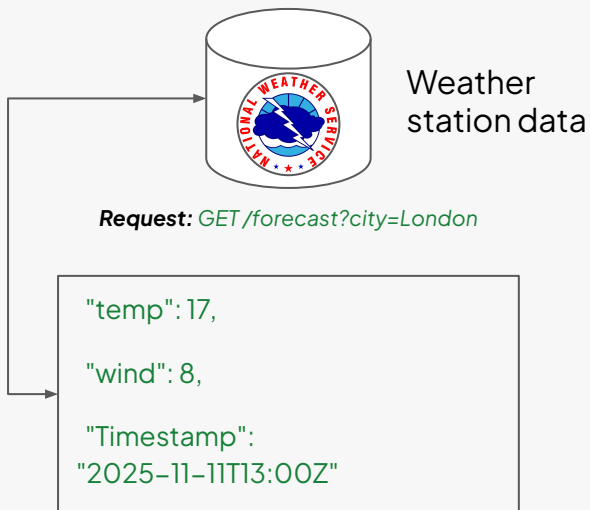
Same but different

- It looks familiar: still HTTP/JSON, but…
- **Probabilistic, not deterministic:** same input ≠ same output.
- **Tokens, not bytes/rows:** cost & latency tied to token counts.
- **Long-running:** seconds, not milliseconds → streaming is essential.
- **Non-binary success:** outputs can be valid, invalid, or low-quality.
- **Observability needed:** must track usage, errors, drift.
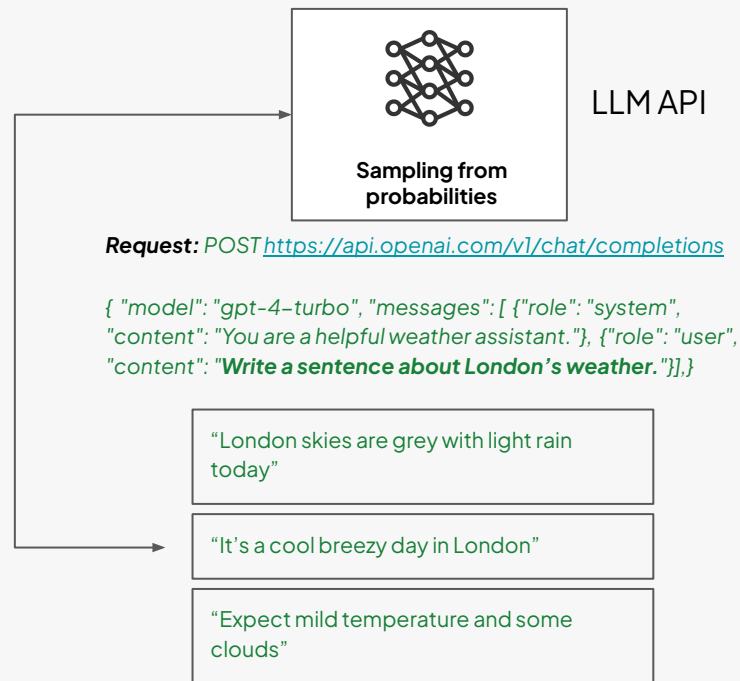
Deterministic vs Probabilistic APIs

Same input + same time = same output.
Output *changes when new data arrives*

Weather station data

Request: *GET /forecast?city=London*

"temp": 17,

"wind": 8,

"Timestamp": "2025-11-11T13:00Z"

Same input = multiple possible outputs.
Output *changes even when nothing changed*

Sampling from probabilities

LLM API

Request: *POST https://api.openai.com/v1/chat/completions*

*{ "model": "gpt-4-turbo", "messages": [{"role": "system", "content": "You are a helpful weather assistant."}, {"role": "user", "content": "**Write a sentence about London's weather.**"}],}*

"London skies are grey with light rain today"

"It's a cool breezy day in London"

"Expect mild temperature and some clouds"

Anatomy of a request

- **Model selection.** Use a specific model/version. Drives quality, latency, and cost. Routable in production.

- **Messages array**. Chat history as {role, content}. Controls context and cost. System sets zolicy, User asks, Assistant can include prior turns.

- **Sampling knobs**. Temperature and top_p shape randomness and style. Lock these down for consistency.

- **Output control**. max_tokens caps spend and latency. stop prevents run-on generations and enforces boundaries.

- **Metadata for ops**. User for tenancy, analytics, and quotas. idempotency_key makes retries safe.

- **Tokens, not characters**. Everything is billed and bounded in tokens. Enforce a context window budget and track prompt + completion tokens.

```
POST /v1/chat/completions
{
 "model": "gpt-4o-mini",
 "messages": [
   {"role": "system", "content": "You are a helpful teaching assistant."},
   {"role": "user", "content": "Explain what tokenization means."}
 ],
 "temperature": 0.7,
 "top_p": 0.9,
 "max_tokens": 256,
 "stop": ["</end>"],
 "user_id": "student123",
 "idempotency_key": "req-9f3a2c"
}
```

Response and Streaming

- **Response (non-streaming):** JSON with the model's text, finish reason, and usage (token counts).
- **Streaming:**
  - Server sends *partial tokens* as they are generated
    i. Server–Sent Events
    ii. WebSocket
  - Server → implement server streaming (e.g., FastAPI StreamingResponse)
  - Client → consumption (e.g. an iterator streamer), render chunks as they arrive.
- **Better UX**: lower **TTFT** (time-to-first-token), steady **TPOT** (Time Per Output Token).

user

HTTP request

update

update

update

close conn
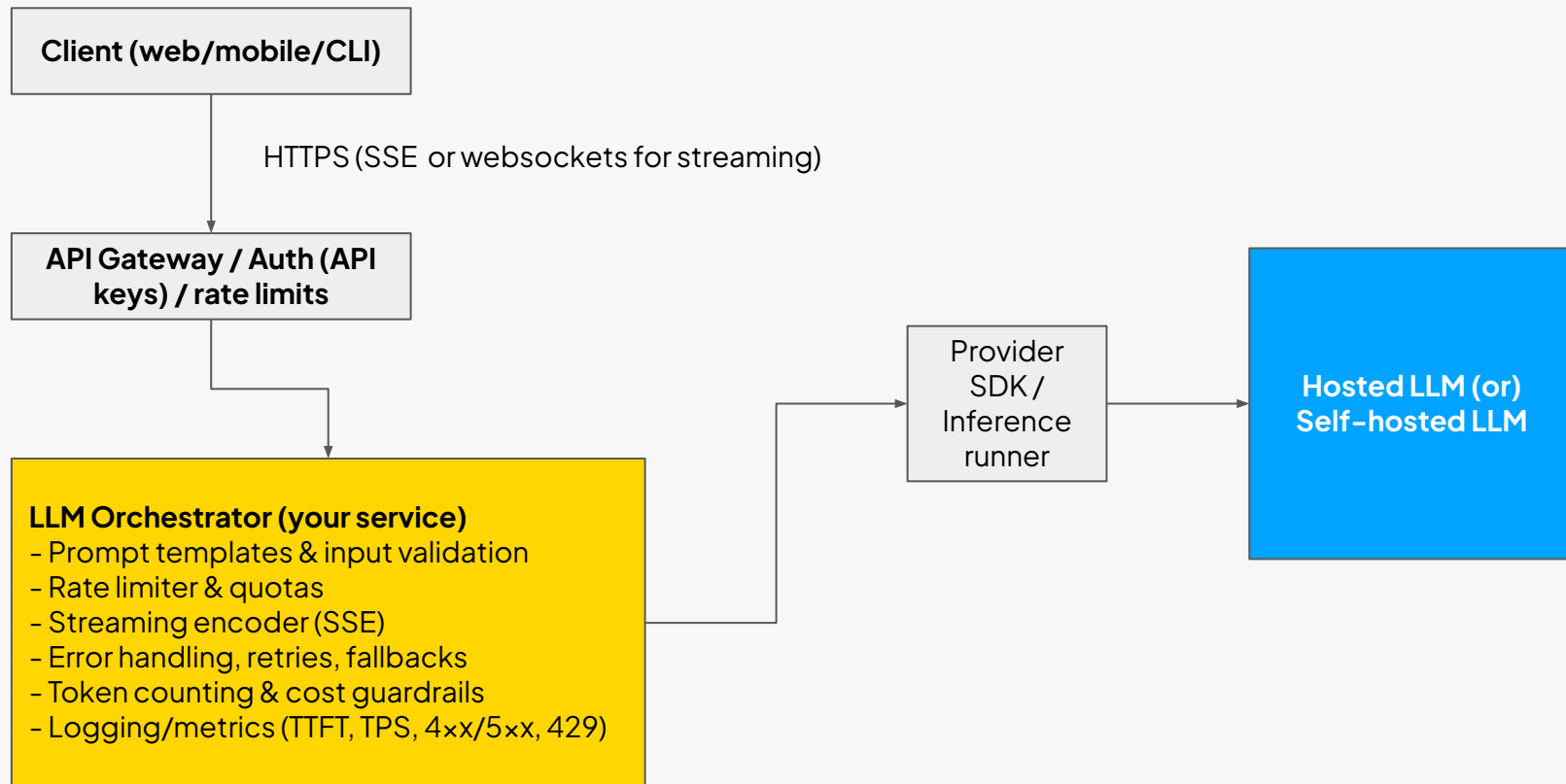
Client

LLM Server
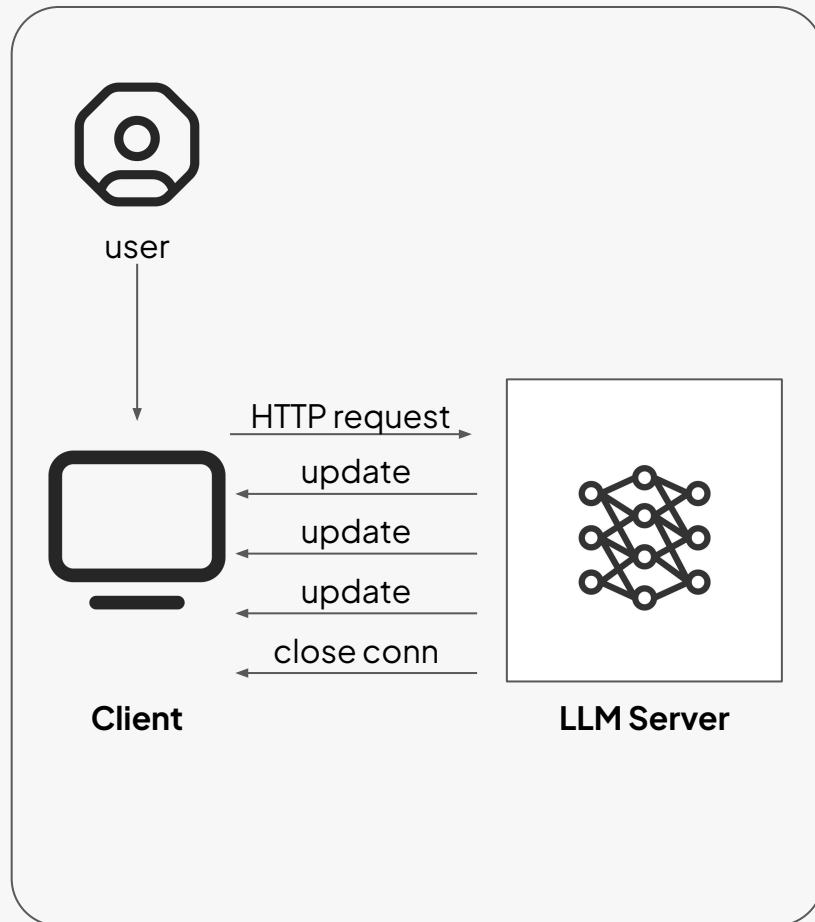
Rate Limits & Quotas

- **Provider quotas:** requests/min and tokens/min; exceeding returns HTTP 429.

- **Rate-limiting of your service** per API key/user:
  - *Fixed window*: count requests per time window.
  - *Sliding window*: A moving window.
  - *Token bucket*: Users earn 'tokens' over time; each request consumes one.

- **Batching (self-hosting):** Group multiple requests and process them in a single forward pass on the GPU (vLLM, Hugging Face TGI).
  - Huge throughput gains (2–10×).
  - Better GPU utilization (lower cost per token).
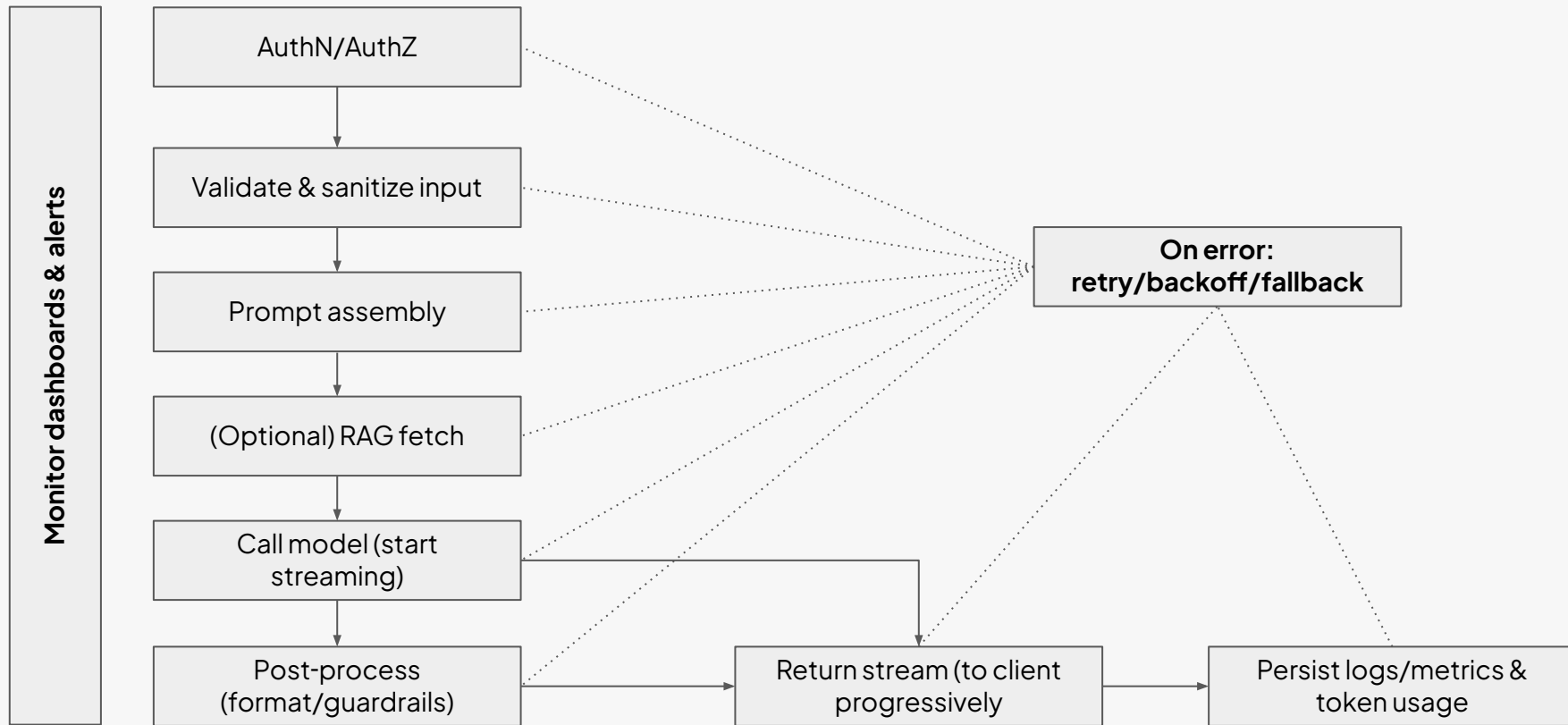 - Higher latency.
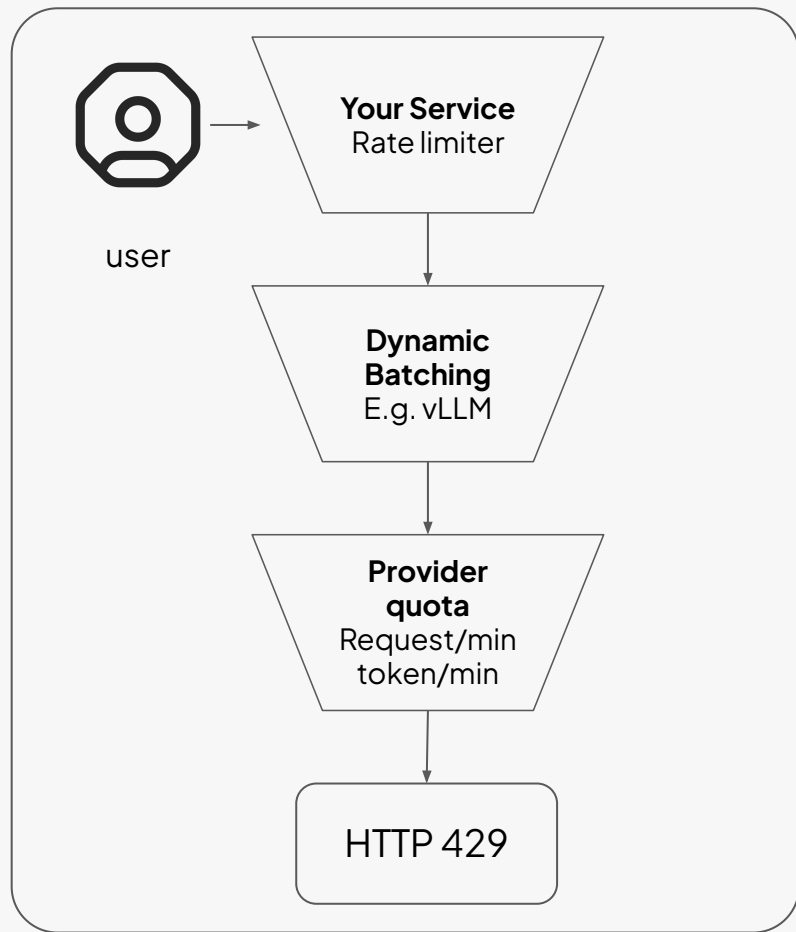
Error handling

- **Classify and handle:**
  - **4×x** (client-side errors – bad request/auth), **429** (rate limited),
  - **5×x** (server-side errors – downtime/timeouts).
- **Retry policy:** Exponential backoff + jitter; cap retries; use idempotency keys. Consider parallel redundant calls (latency vs. cost).
- **Empty/malformed output:** detect (e.g., invalid JSON), auto-repair or reprompt, or fall back to a human.
- **Monitor:** Observe TTFT, TPS, error codes; design for detection (MTTD), response (MTTR), and low change failure rate (CFR).

Minimal LLM Service Architecture

Client (web/mobile/CLI)

HTTPS (SSE or websockets for streaming)

API Gateway / Auth (API keys) / rate limits

LLM Orchestrator (your service)
- Prompt templates & input validation
- Rate limiter & quotas
- Streaming encoder (SSE)
- Error handling, retries, fallbacks
- Token counting & cost guardrails
- Logging/metrics (TTFT, TPS, 4xx/5xx, 429)

Provider SDK / Inference runner

Hosted LLM (or) Self-hosted LLM

COLUMBIA UNIVERSITY

End to end request lifecycle

Monitor dashboards & alerts

AuthN/AuthZ

Validate & sanitize input

Prompt assembly

(Optional) RAG fetch

Call model (start streaming)

Post-process (format/guardrails)

On error: retry/backoff/fallback

Return stream (to client progressively

Persist logs/metrics & token usage

COLUMBIA
UNIVERSITY

LLMOps

What are LLMOps?

- *LLMOps is the art of managing, deploying, and monitoring LLMs in production to ensure efficiency, reliability, and safety at scale.*

- **Compute resources**: GPU/accelerator need; cut inference cost with quantization/distillation.

- **Transfer learning**: Fine-tune foundation models for special domains.

- **Human feedback**: Use RLHF feedback for eval and future fine-tunes.

- **Eval**: Go beyond accuracy (different metrics BLEU/ROUGE and model/human-as-judge).

- **Prompt engineering:** Robust templates to reduce hallucinations and prompt attacks.

- **LLMs Pipelines**: Chain LLM + retrieval/tools (LangChain/LlamaIndex) for complex tasks.

- **DataOps**: Still hard; curation, labeling, versioning, governance matter.

The role of GPUs

- **Every token = matrix multiply.**
  GPUs are designed for massively parallel computation. Without GPU acceleration, a model like Llama-2–7B would take minutes to produce a sentence. With GPUs, it's milliseconds.

- **VRAM, model/batch size, and context.**
  GPUs are also scarce and expensive. The model must fit entirely inside that VRAM, along with intermediate activations.

- **GPUs constrain everything**. TTFT/TPOT are how users feel those constraints; batching and streaming, and quantization exists to make the model fit the hardware you can afford.
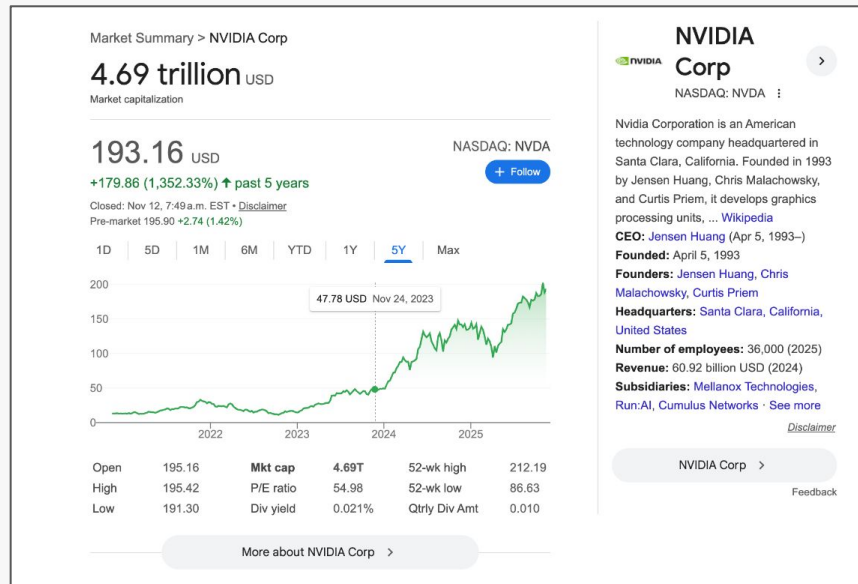
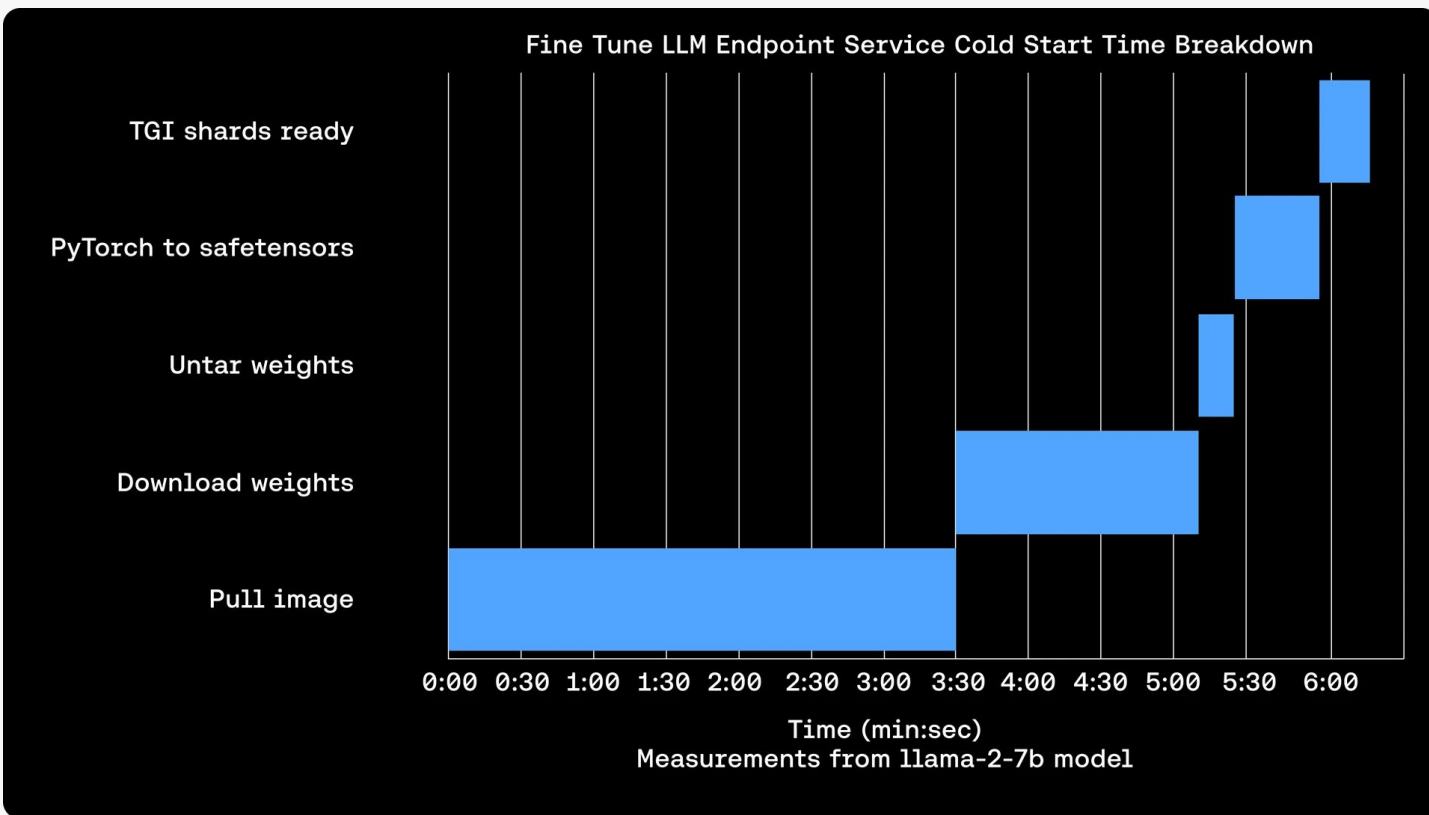*Gpus utilization exploded with the introduction of LLMs*

Fundamentals of LLMOps

- **GPUs** → availability, memory bandwidth, kernels, context length, throughput, and cost.

- **Model size** → downloads, deployment, developer feedback loop.

- **Latency** → Time To First Token (TTFT), Time Per Output Token (TPOT).

- **Compute & memory** → GPU parallelism and utilization.

- **Tokens & context limits** → cost spikes and bottlenecks.

- **Model quality & safety** → hallucinations, bias, security.

Model size

- **Long downloads & cold starts:** moving multi-GB weights dominates boot time; choose storage & boot strategies wisely.

- **Slower deploys/rollouts:** bigger artifacts → longer rollout windows → more user impact.

- **Slower dev loop:** every restart/redeploy drags feedback cycles.

- **Boot/serve strategies:**
    - *download at boot,*
    - *fuse/bake into image,*
    - *mount shared volumes (amortize downloads across replicas).*

Moving artifacts vs. model compute

Fine Tune LLM Endpoint Service Cold Start Time Breakdown

TGI shards ready

PyTorch to safetensors

Untar weights

Download weights

Pull image

0:00 0:30 1:00 1:30 2:00 2:30 3:00 3:30 4:00 4:30 5:00 5:30 6:00

Time (min:sec)
Measurements from llama-2-7b model

COLUMBIA UNIVERSITY

Make the model smaller: quantization

- Quantization makes a model **smaller and faster** by storing numbers with **fewer bits** (e.g., from decimals to integers).
- The model behaves the same (mostly), but there is **less memory to load and less data to move**, so startups are quicker and tokens come out faster.
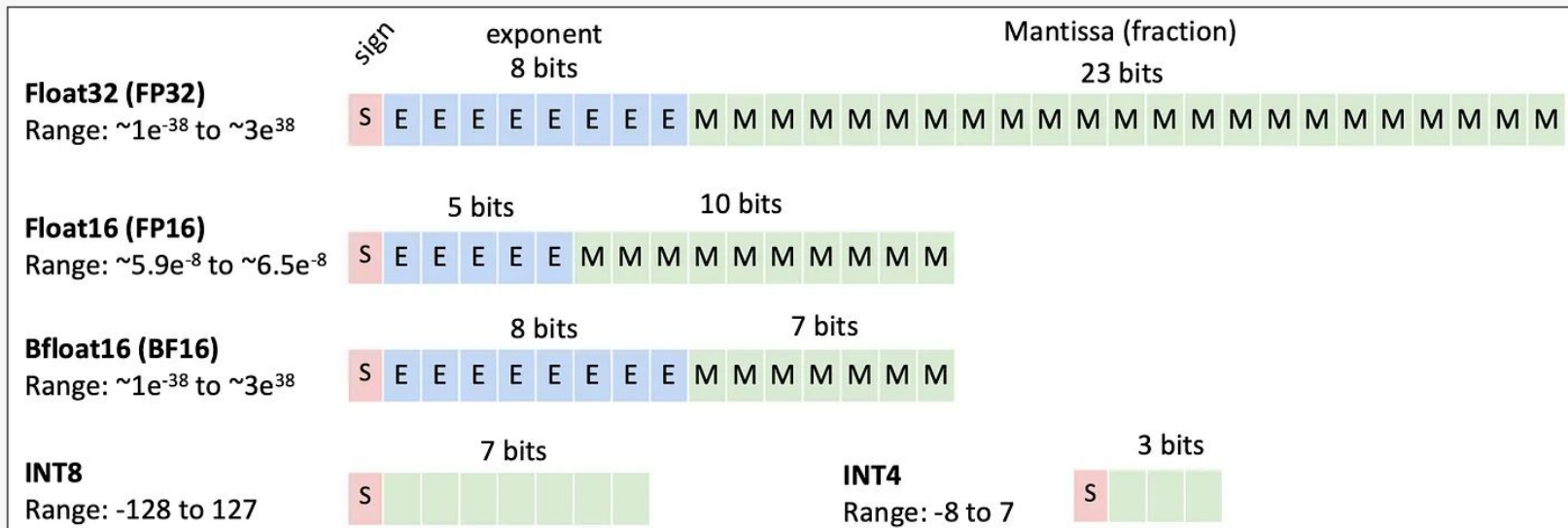
Make the model smaller: quantization

Post–Training Quantization (PTQ) — *no retraining.*

- Start with INT8 weights-only. It's the default that "just works" most of the time.
- If memory is still too tight, try INT4 (expect a bit more quality risk).
- Optional later: also quantize activations if you need more savings (more tuning, higher risk).

Quantization–Aware Training (QAT) — *short fine-tune.*

- Use if PTQ hurts quality beyond your SLOs.
- The model "learns" to live with lower precision; better quality, more effort.

Make the model smaller: quantization

Original Image

"Quantized" Image
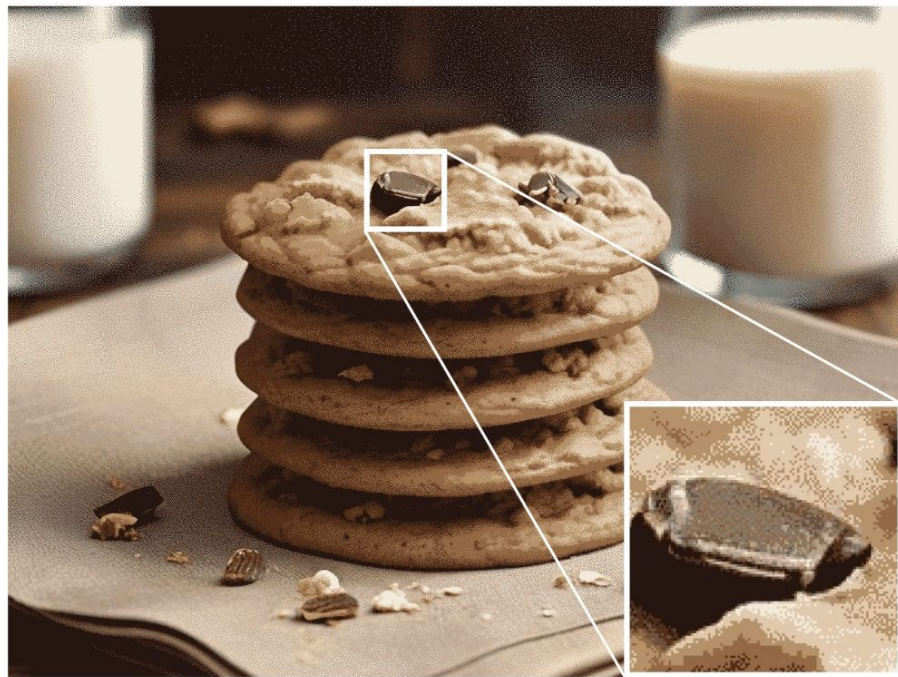
Thinking about quantization

When to use

You need task/domain lift but cannot afford full model fine-tuning.

You serve multiple tasks/tenants and want hot-swappable behavior (load different adapters on the same base).

You want small artifacts (MBs) for fast rollout and easy rollback.

What to check before

Precision vs. tolerance: Is the task okay with tiny rounding errors? (Most chat, summarization and classification are).

Memory vs. quality trade-off: INT8 is usually safe; INT4 gives more savings but may lose subtle reasoning accuracy.

Hardware support: Make sure your inference stack (PyTorch, TensorRT, ONNX Runtime) actually supports the target bit width.

What it depends on

How aggressively you quantize: The more compact, the riskier.
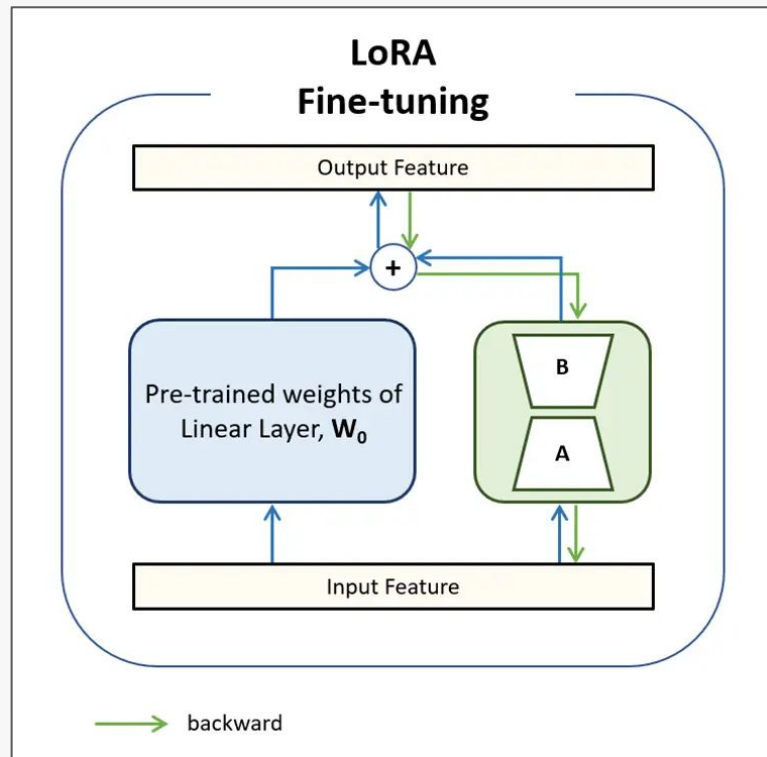
What you quantize:

- *Weights only* = fastest win, minimal quality loss.
- *Weights + activations* = more compression, more tuning needed.

Whether you fine-tune: *quantization-aware training (QAT):* short retrain — more work, best accuracy.

Freeze your model PEFT, LoRA

- Parameter-Efficient Fine-Tuning (**PEFT**) → Low-Rank Adaptation (**LoRA**).

- You keep the big base model **frozen**.

- You train only tiny add-on pieces (called *adapters*) on your task data.

- At inference, you run: the base model plus these small adapters to get task-specific behavior.

- Small training cost, small artifacts, and fast iteration.

LoRA Fine-tuning

Output Feature

Pre-trained weights of Linear Layer, W_0

B

A

Input Feature

→ backward

Thinking about PEFT

When to use

Model doesn't fit in GPU memory?

TTFT too high? Smaller weights mean faster loading.

TPOT too low? Less data to move between GPU memory and compute → higher tokens per second.

Cost too high? Quantized models often run on fewer or smaller GPUs, saving money.

Workflow

1. Pick a strong base (close to your domain if possible).

2. Define the task and collect a small, clean dataset (100s–10k examples can move the needle).

3. Choose adapter config: target layers (attention/FFN), rank (e.g., r=8–64), dropout if needed.

4. Train adapters (few hours on a single GPU for small ranks).

5. Evaluate: task accuracy + safety; run your SLO metrics (TTFT, tokens/sec) to confirm no regressions.

6. Serve: load base model + adapter file; hot-swap adapters by route/tenant.

7. Optionally merge into a single artifact if you don't need multi-tenant swapping.

If you can't go smaller, serve smarter: amortize I/O

- **Problem:** Every new copy of your model server (a "replica") downloads the same multi-GB weights → slow cold starts, wasted bandwidth.
- **Download once, share many.** Put the weights on shared storage and have all replicas mount those files instead of re-downloading them.

Avoid repeated multi-GB transfers, cut cold-start variance, and keep auto scaling responsive.

COLUMBIA
UNIVERSITY
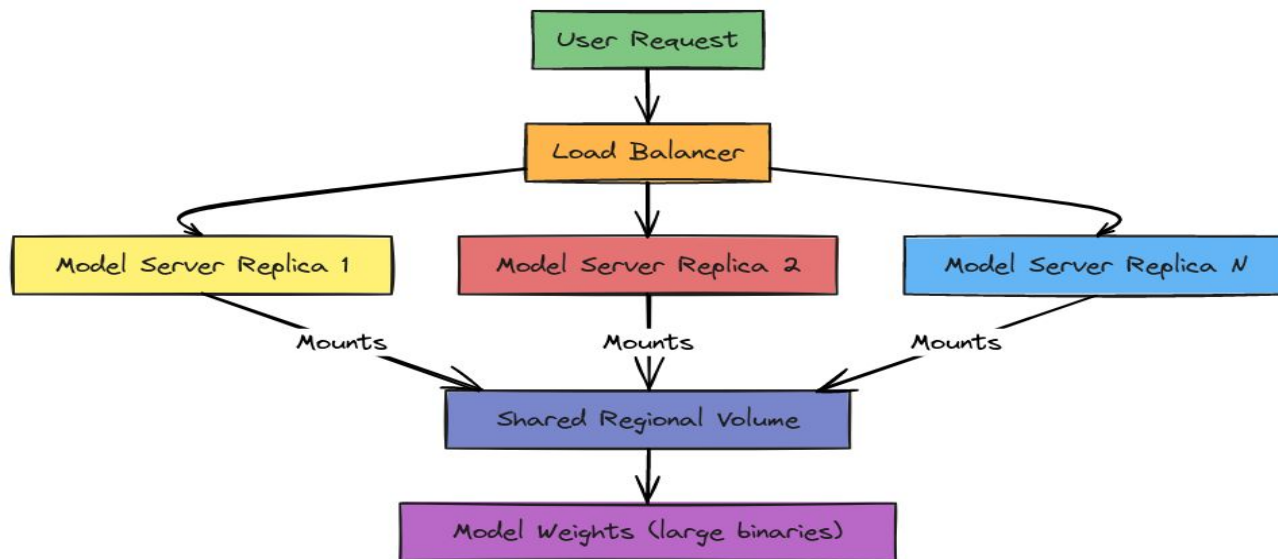
Latency: TTFT and TPOT

TTFT (Time-to-First-Token):

- How long it takes before the model **starts talking**.
- You send a question → nothing happens yet → and then *finally* the first word appears on screen.
- That waiting period — from when you press Enter to when you see the first word — is **TTFT**.
- It depends on how long the model needs to read and understand your input before it can begin answering.

TPOT – Time Per Output Token

- Once the model starts talking, it sends words one by one.
- **TPOT** is how fast those words come out.
- If the model "types" one word every 0.1 seconds, TPOT = 0.1 s/token (or 10 tokens per second).
- It's basically the **typing speed** of the model.
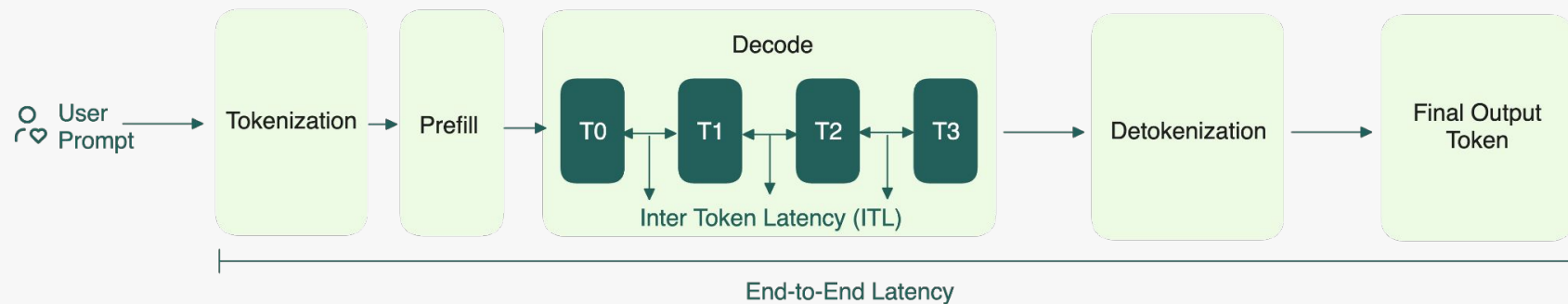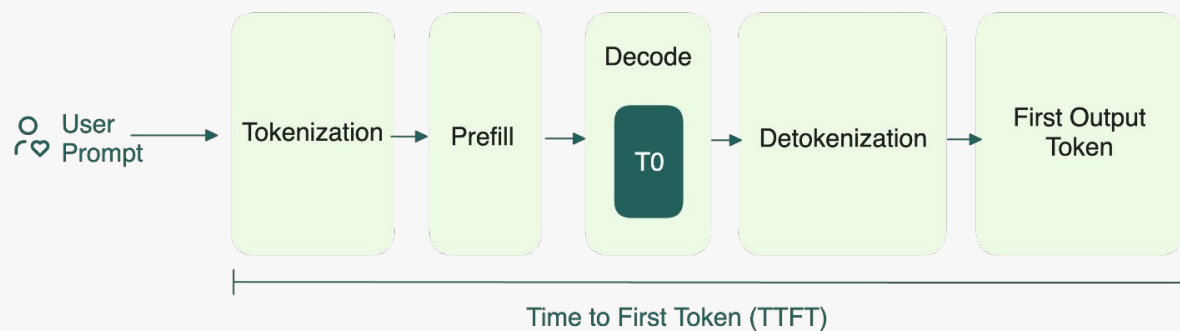
Note: Detokenization happens after each decode step. Each token (T0, T1, T2) is detokenized and output sequentially, not just the final one (T3).
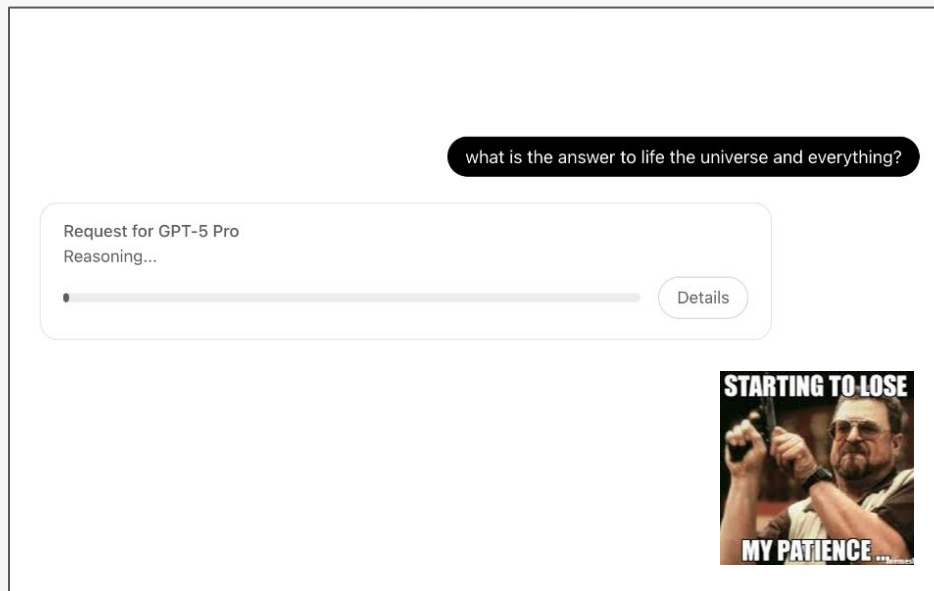
Why does it matter?

User perception & UX

- TTFT = "Is this thing on?" A long TTFT feels broken even if total time is OK.
- TPOT = "C'mon! c'mon! c'mon!." It needs to be fast enough so users read comfortably while tokens arrive.

Meet your SLOs with capacity planning

- Splitting latency into TTFT (prefill) and TPOT (decode) tells you *where* time is going and which lever (prompt budget vs. model throughput) to pull.

Cost control

- Long TTFT wastes GPU time doing non-streaming work; slow TPOT extends GPU occupancy per request → higher $/req.

what is the answer to life the universe and everything?

Request for GPT-5 Pro
Reasoning...

Details

STARTING TO LOSE
MY PATIENCE...

Thinking about TTFT and TPOT

TTFT — Time to First Token (prefill)

1. **How long is the prompt?** Longer input = more reading and setup for the model → slower first response.

2. **Is the model ready?** If weights need to be downloaded or loaded into GPU memory, that adds time.

3. **Is the system warm?** The first request often triggers things like kernel compilation or memory setup. Also waiting in queues, autoscaling delays, or network hiccups all add to TTFT.

4. **Is there work to do before answering?** Tasks like retrieval (RAG), safety filters, or tool-planning may run *before* generation starts.

5. **Batching behavior?** if the server waits to fill a batch before starting, that can delay the first token.
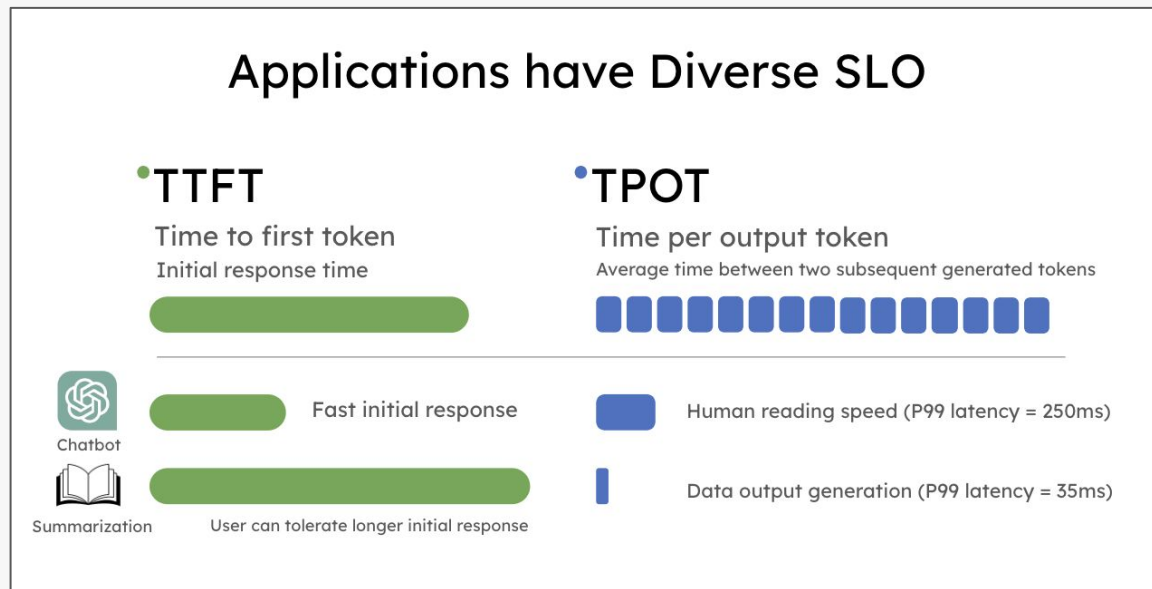
TPOT — Time Per Output Token (decode)

1. **How big is the model?** Bigger models do more math per token → slower. Smaller or quantized models → faster.

2. **What GPUs are we using?** Generating tokens is mostly limited by how fast the GPU can read/write memory. Faster hardware = higher token rate.

3. **Batch size?** Larger batches improve total throughput but can slow down individual requests.

4. **Sampling & rules?** Features like temperature, top-p sampling, or structured output checks add small per-token overheads.

5. **Parallelism & communication:** Splitting a model across multiple GPUs or servers introduces coordination time; fast interconnects help keep TPOT low.

Know your applications

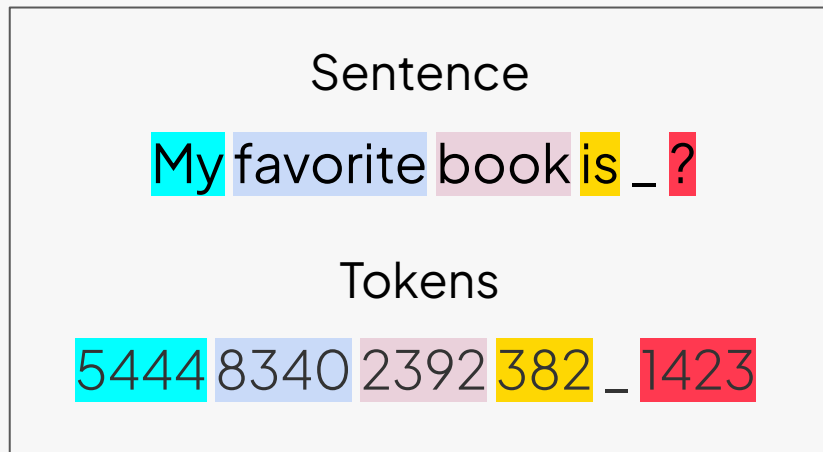**Different applications have different latency goals:**

- Chatbots need a fast *time to first token (TTFT)* for responsiveness
- Summarization can tolerate longer startup as long as *time per output token (TPOT)* is high for throughput.

Applications have Diverse SLO

TTFT
Time to first token
Initial response time

TPOT
Time per output token
Average time between two subsequent generated tokens

Chatbot — Fast initial response

Summarization — User can tolerate longer initial response

Human reading speed (P99 latency = 250ms)

Data output generation (P99 latency = 35ms)

Source: https://hao-ai-lab.github.io/blogs/distserve/

Sampling

- The model doesn't pick one fixed next word; it picks from a probability list of likely next words.

- Sampling is how we choose from that list: more random (creative) or more deterministic (factual/consistent).

- By turning a few knobs (e.g., temperature, top-k/top-p), you can make outputs safe & steady or diverse & imaginative.

Sentence

My favorite book is _ ?

Tokens

5444 8340 2392 382 _ 1423

Moby Dick	4.1%
War and Peace	3.9%
the one	1.4%
written	2.2%

Columbia University

How sampling works

- For each step, the model produces a **distribution over tokens** (a ranked list with probabilities).

- **Greedy** decoding: always pick the top token → fast, consistent, but can get stuck in loops or be dull.

- **Temperature** rescales the distribution:
  - **Low** (≈0–0.3): sharper, more deterministic.
  - **Mid** (≈0.7–1.0): balanced.
  - **High** (>1.0): more diverse, more risk of nonsense.

- **Top–k**: restrict choices to the **k best** tokens; **Top–p (nucleus)**: restrict to the **smallest set** whose probabilities sum to *p*.

- **Repetition / presence/frequency penalties**: nudge the sampler away from repeating the same words.

- **Structured / constrained decoding**: enforce a JSON schema or grammar, or force valid function-call arguments; this narrows the choices each step.

- **Test–time compute** (quality boosters):
  - **Best-of–n**: generate many candidates, pick the best with a scorer (e.g., task heuristic, reward model, "AI as a judge").
  - **Self-consistency**: sample several chains-of-thought and select the most consistent answer.
  - These increase **quality** at the cost of **latency** and **$$**.

Sentence

My favorite book is _ ?

Tokens

5444 8340 2392 382 _ 1423

| Moby Dick | 4.1% |
War and Peace	3.9%
the one	1.4%
written	2.2%

Sampling's core knobs

Temperature: 0.2–0.7 for factual/tasks; 0.7–1.0 for creative.

Top-p: 0.8–0.95 (nucleus sampling) is a solid default; leave **top-k** off unless you need hard caps.

Repetition control: enable presence/frequency penalties for chat; tune lightly to avoid stilted output.

Constrained decoding: require JSON/schema for tool calls and integrations; reduces cleanup and errors.
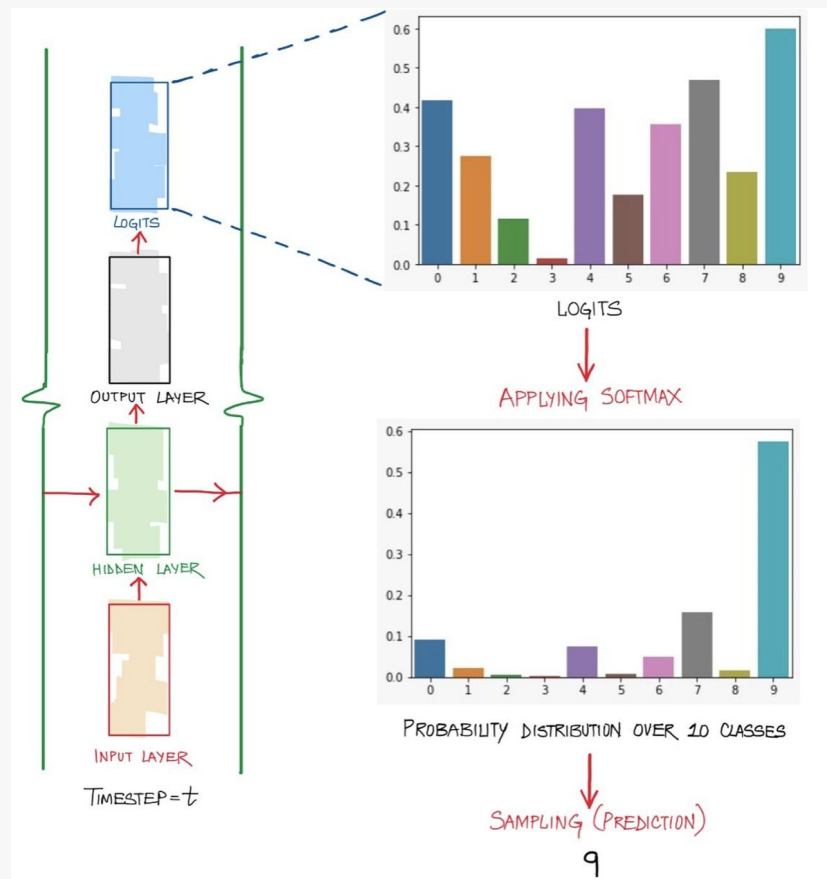
Trade-offs

- **Best-of-n**: sample n candidates, pick best by a task-specific scorer (regex, reward model, judge). Quality ↑, cost ↑.

- **Self-consistency** (reasoning): sample multiple chains, vote/select the most consistent final answer. Robustness ↑, cost ↑.

- **Use budgeted n** (e.g., 3–5) and stop early if a high-confidence candidate appears.

- **Diversity vs Reliability**: Higher temperature/top-p → more variety, more risk.

- **Quality vs Cost**: Best-of-n/self-consistency improve accuracy but multiply tokens and latency.

- **Speed vs Format**: Constraining output ensures validity but can slow decode slightly.

- **Global vs Per-turn**: You can adapt knobs per step (e.g., low temperature until a tool call returns; then relax).

Buy vs. Build

Provider APIs vs. Self-Hosting

Privacy: API ⇒ data leaves org; Self-host ⇒ data stays in-house.

Lineage/IP: Closed ⇒ opaque data; Open ⇒ inspectable datasets.

Performance: API ⇒ top quality but external latency; Self-host ⇒ tuned SLOs.

Functionality: API ⇒ rich features (function calling, guardrails); Self-host ⇒ raw access (logprobs, finetune).

Cost: API ⇒ pay-per-token; Self-host ⇒ infra/ops, cheaper at scale.

Control: API ⇒ vendor limits/opacity; Self-host ⇒ freeze versions, full observability.

On-device: API ⇒ always online; Self-host ⇒ quantized models run offline.
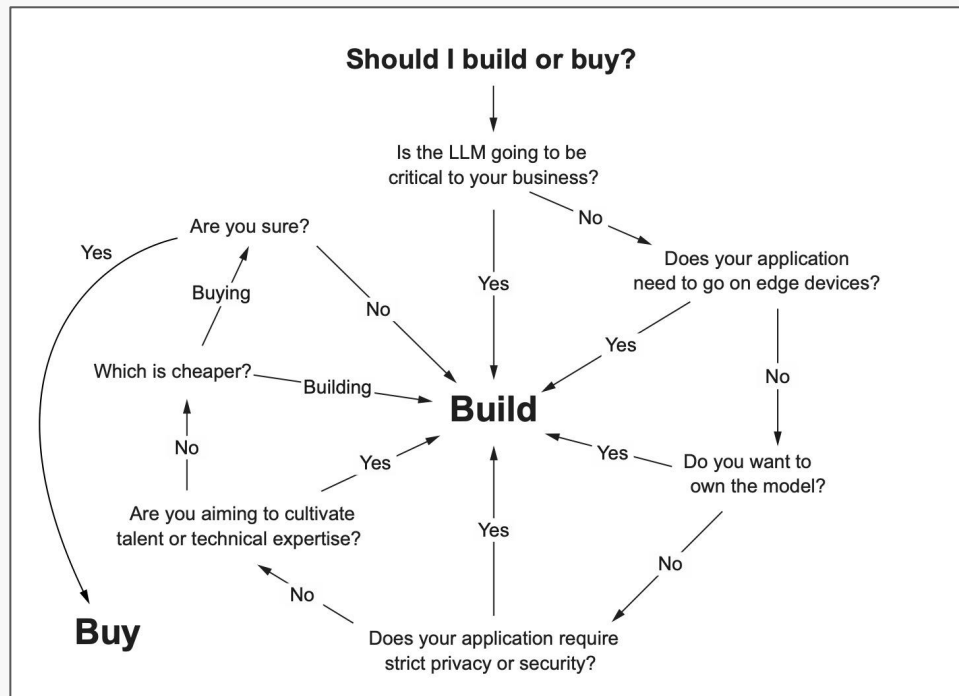

Image from Brusseau and Sharp

Provider APIs vs. Self-Hosting

- **A. Clinical notes summarizer (hospital)**

- **B. Consumer copywriting assistant (global app)**

- **C. Field support chatbot (spotty connectivity)**

COLUMBIA
UNIVERSITY