CS498: Algorithmic Engineering

Lecture 27: Using LLMs in Practice

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 15

Outline

Recap: How Generation Works

Recall from L26: we generate one token at a time.

Given prompt tokens $y_1, y_2, \ldots, y_t$, to predict y_{t+1}:

1. Embed: look up each token in the embedding table. $x_i = E[y_i]$
2. Transform: pass through all L transformer blocks. Each block computes attention (Q, K, V projections $q_i = x_i W_Q, k_i = x_i W_K, v_i = x_i W_V$, softmax, weighted sum) then FFN $\rightarrow$ linear head $\rightarrow$ logits $\in \mathbb{R}^{|\text{vocab}|}$
3. Sample: apply temperature, top-p, sample one token y_{t+1}
4. Append y_{t+1} to the sequence. Go to step 1.

Every new token requires a full forward pass through the entire model.

The Problem: Redundant Computation

Step 1: generate token 5. Forward pass processes tokens $[1, 2, 3, 4]$.
Step 2: generate token 6. Forward pass processes tokens $[1, 2, 3, 4, 5]$.

Tokens $1, 2, 3, 4$ are processed again. Same embeddings, same Q, K, V projections, same everything. All wasted.

Step 3: generate token 7. Forward pass processes $[1, 2, 3, 4, 5, 6]$.
Now tokens 1–5 are recomputed for the third time.

To generate a sequence of length T: total work is $O(T^2)$ instead of $O(T)$.

For $T = 4096$: we're doing $\sim$8 million redundant projection operations. There must be a better way.

The Key Observation

Look at what attention computes for token t:

$$q_t = x_t W_Q, \quad k_t = x_t W_K, \quad v_t = x_t W_V$$

Then the attention score between token t (query) and token j (key):

$$\text{score}_{t,j} = \frac{q_t \cdot k_j}{\sqrt{d_k}}$$

The keys and values of past tokens don't change when we add a new token.

$k_1, k_2, \ldots, k_{t-1}$ are the same whether we're generating token t or token $t + 100$.

So why recompute them?

The KV Cache

Idea: store the K and V vectors for all past tokens. When generating a new token, only compute Q, K, V for the new token.

Concrete example: generating tokens for "The cat sat on"

Step	Compute Q,K,V for	Attend over	Cache
"The"	token 1	$\{1\}$	k_1, v_1
"cat"	token 2 only	$\{1, 2\}$	k_1, v_1, k_2, v_2
"sat"	token 3 only	$\{1, 2, 3\}$	$+ k_3, v_3$
"on"	token 4 only	$\{1, 2, 3, 4\}$	$+ k_4, v_4$

Each step: compute one new (q, k, v) triple, append k and v to the cache, compute attention of the new query against all cached keys.

Work per step: $O(d^2 + t \cdot d)$ instead of $O(t \cdot d^2)$. Huge savings.

KV Cache: The Memory Cost

We saved compute, but now we're storing vectors.

Per token, per layer: store one K vector and one V vector, each in $\mathbb{R}^{\mathrm{d}}$.

For a single sequence of length T:

$$\text{KV cache} = 2 \times \mathrm{L} \times \mathrm{T} \times \mathrm{d} \times \text{bytes\_per\_param}$$

Example: Non-GQA Llama-3 8B ($\mathrm{L} = 32$, $\mathrm{d} = 4096$, bf16), $\mathrm{T} = 8192$:

$$2 \times 32 \times 8192 \times 4096 \times 2 \text{ bytes} \approx 4 \text{ GB per sequence}$$

Model weights: 16 GB. One sequence's KV cache: 4 GB.

On an 80 GB A100: room for about 16 concurrent sequences. Serving 1000 users requires careful memory management.

Batching Trades Compute for Memory

Each user has their own KV cache

- Cache grows with sequence length T
- Not shared across users

Total memory:

$$\text{Total KV} = \text{batch size} \times (2LTd)$$

Implication:

- Larger batch $\Rightarrow$ more memory usage
- Memory, not compute, becomes the limit

Two Phases of Generation

Given a prompt of n tokens, generation splits into:

Phase 1: Prefill (process the prompt)

- Run a forward pass over all n tokens
- Build internal state (KV cache)

Phase 2: Decode (generate new tokens)

- Produce tokens one at a time
- Each new token depends on all previous ones

Key contrast:

$$\text{Prefill} = \text{parallel} \qquad \text{Decode} = \text{sequential}$$

Batching: Filling the GPU

During decode, a single request uses <1% of the GPU's compute capacity.

Idea: process multiple users' requests at the same time.

Instead of generating 1 token for 1 user per step, generate 1 token for each of 32 users per step. The matrix multiply is barely larger, but you get 32x throughput.

Problem with static batching: if user A's response is 10 tokens and user B's is 500 tokens, user A's GPU slot sits idle for 490 steps.

Continuous batching: when user A finishes, immediately give that slot to user C. Requests join and leave the batch independently.

This is like a restaurant that seats new guests as soon as a table opens, rather than waiting for the entire dining room to finish.

vLLM: Putting It All Together

vLLM (Kwon et al., 2023): the standard open-source engine for LLM serving.

It solves the problems we just described:

Problem	vLLM's solution
Redundant computation	KV cache (automatic)
KV cache wastes memory	PagedAttention (virtual memory for KV)
GPU idle during decode	Continuous batching
Model too big for 1 GPU	Tensor parallelism (shard across GPUs)

PagedAttention: manages the KV cache like an OS manages RAM. Fixed-size blocks allocated on demand, freed when a request finishes. Shared prefixes (e.g., system prompts) share cache pages.

Result: 2–4x higher throughput than naive HuggingFace generation.

The Open Model Landscape (2025)

You don't have to train from scratch. Many strong models are freely available:

Model	Params	Strengths	A100-40GB
Qwen3-8B	8B	General, multilingual	1 GPU (bf16)
Llama-3.1-8B	8B	General purpose	1 GPU (bf16)
Qwen2.5-Math-7B	7B	Math reasoning	1 GPU (bf16)
Qwen3-32B	32B	Very strong	4 GPUs (bf16)
Llama-3.1-70B	70B	Near-frontier	4 GPUs (int4)

On Delta (4×A100-40GB = 160 GB total), you can run:

- Any 7–8B model on a single GPU in bf16
- 32B models across 4 GPUs in bf16
- 70B models across 4 GPUs with 4-bit quantization

Setting Up on Delta

Step 1: Install vLLM (one-time, on login node):

```
module load pytorch-conda/2.8
pip install --target=$HOME/pylibs vllm
```

Step 2: We have pre-downloaded Qwen3-32B to the shared directory:

```
ls /projects/bgvu/shared_data/models/Qwen3-32B
# config.json  model-00001-of-00017.safetensors  ...  tokenizer.json
```

No need to download anything yourself. The model is ∼64 GB on disk (bf16).

To download a different model (from the login node, which has internet):

```
python -c "from huggingface_hub import snapshot_download; \
  snapshot_download('Qwen/Qwen2.5-Math-7B', \
  local_dir='/projects/bgvu/shared_data/models/Qwen2.5-Math-7B')"
```

Running Inference with vLLM

```
from vllm import LLM, SamplingParams

# Load from the shared directory (4 GPUs for 32B model)
llm = LLM(
    model="/projects/bgvu/shared_data/models/Qwen3-32B",
    tensor_parallel_size=4,
)

params = SamplingParams(temperature=0.7, top_p=0.9, max_tokens=256)

prompts = [
    "Prove that the square root of 2 is irrational.",
    "Explain P vs NP to a 10-year-old.",
]
outputs = llm.generate(prompts, params)

for output in outputs:
    print(output.outputs[0].text)
```

vLLM handles KV cache, batching, and tensor parallelism. You just call generate.

SLURM Script

```
#!/bin/bash
#SBATCH --job-name=vllm-inference
#SBATCH --account=bgvu-delta-gpu
#SBATCH --partition=gpuA100x4
#SBATCH --gpus-per-node=4
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=64
#SBATCH --mem=200G
#SBATCH --time=01:00:00

module load pytorch-conda/2.8
export PYTHONUNBUFFERED=1
export PYTHONPATH=$HOME/pylibs:$PYTHONPATH

python my_inference_script.py
```

For a 7B model on 1 GPU, change to –gpus-per-node=1, –mem=64G, and set tensor_parallel_size=1 in the script.

What's an "OpenAI-compatible API"?

ChatGPT, Claude, and other LLM services expose an HTTP API. You send a POST request with your messages, you get back a response.

The format OpenAI defined became the de facto standard:

```
# This works with OpenAI, AND with vLLM, AND with many other providers
response = client.chat.completions.create(
    model="some-model",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "What is 2+2?"},
    ],
)
print(response.choices[0].message.content)
```

vLLM can serve this exact API. Point the openai Python library at your vLLM server instead of OpenAI's servers, and the same code works.

Same interface, your own GPUs, your own data, no API fees.

Running vLLM as a Server

Start the server in a SLURM job:

```
python -m vllm.entrypoints.openai.api_server \
--model /projects/bgvu/shared_data/models/Qwen2.5-Math-7B  \
--tensor-parallel-size 1 --port 8000
```

The server runs on the compute node (e.g., gpua050). To reach it from the compute node on Delta, create an SSH tunnel:

```
# From Delta compute node:
ssh -L 8000:gpua027:8000 localhost
```

Now localhost:8000 on your compute node routes to the compute node:

```
import openai
client = openai.OpenAI(base_url="http://localhost:8000/v1", api_key="unused")
r = client.chat.completions.create(model='/projects/bgvu/shared_data/models/Qwen2.5-Math-7B',
      messages=[{'role':'user','content':'What is 2+2?'}], max_tokens=32 )
print(r.choices[0].message.content)
```

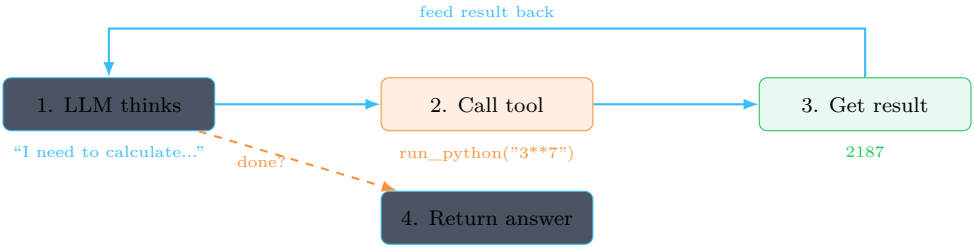
You now have your own private ChatGPT running on Delta.

What's an Agent?

An LLM alone can only generate text. It cannot:

- Do arithmetic reliably ("What is 397×8423?")
- Look up current information ("What's the weather?")
- Run code and check if it works

Agent = LLM + tools + a loop.



The LLM decides when to use a tool and which tool. The loop continues until it has enough information to answer.

Under the Hood: The For-Loop Agent

At its core, every agent is a loop:

```
def agent(llm, query, tools, max_steps=5):
    messages = [{"role": "system", "content": SYSTEM_PROMPT}, {"role": "user", "content": query}]
    for step in range(max_steps):
        response = llm.chat(messages)
        text = response.content
        if "ANSWER:" in text:
            return text.split("ANSWER:")[1].strip()
        if "ACTION:" in text:
            action = parse_action(text)       # regex extraction
            result = execute_tool(action, tools)
            messages.append({"role": "assistant", "content": text})
            messages.append({"role": "user", "content": f"RESULT: {result}"})
    return "Agent exceeded max steps."
```

Generate → parse → act → observe → repeat. Educational, but brittle: regex parsing breaks, no error recovery.

The Real Way: Agent Frameworks

Production agents don't parse free text with regex.

Structured tool calling (Anthropic / OpenAI APIs):

- The model outputs tool calls as structured JSON, not free text
- No regex. The model is fine-tuned to produce valid tool schemas.

Agent SDKs handle the full loop for you:

- Claude Agent SDK: Python SDK for building agents with Claude. You define tools, it handles orchestration.
- LangChain / LangGraph: open-source framework for chains of LLM calls, tool routing, memory.
- Claude Code: Anthropic's coding agent. Reads files, writes code, runs tests, commits. Built on the Agent SDK.

Let's build something real with the Claude Agent SDK.

Claude Agent SDK: Setup

```
pip install claude-agent-sdk
export ANTHROPIC_API_KEY="sk-ant-..."   # uses Claude by default
```

Not locked to Claude. You can point it at any OpenAI-compatible API:

```
# Use your own vLLM server on Delta instead:
export ANTHROPIC_BASE_URL="http://gpua027:8000/v1"
export ANTHROPIC_MODEL="Qwen/Qwen3-32B"
```

The simplest possible agent (no custom tools):

```
import asyncio
from claude_agent_sdk import query

async def main():
    async for message in query(
        prompt="Write a Python function that checks if a "
               "graph is bipartite, then test it."
    ):
        if hasattr(message, 'content'):
            print(message.content)

asyncio.run(main())
```

The agent will automatically write code, run it, fix bugs if needed, and return the

Output

[ThinkingBlock(thinking='The user wants me to write a Python function that checks if a graph is bipartite, then test it. Let me write this directly ....', signature='...')]

[TextBlock(text="I'll write a bipartite-checking function using BFS 2-coloring, then test it.")]

[ToolUseBlock(id='toolu_01PuXmUomauPje9wyYARHBdo', name='Write', input={'file_path': '/tmp/bipartite.py', 'content': 'from collections import deque\n\n\ndef is_bipartite(graph: dict[int, list[int]]) -> bool:\n   """\n ...]

[ToolResultBlock(tool_use_id='toolu_01GDkkSSMJSYyouQaiXwfnYT', content=' Bipartite (expect True)
\n [PASS] Empty graph\n [PASS] ...]

[TextBlock(text="All **16/16 tests pass**. Here's a full walkthrough ...")]

Custom Tools: A TSP Solver with Gurobi

We have Gurobi. Let's give the agent a tool that solves TSP:

```
@tool("solve_tsp",
      "Solve a TSP given city names and a distance matrix.",
      {"cities": list, "distances": dict})
async def solve_tsp(args):
    cities = args["cities"]      # ["A","B","C","D","E","F"]
    dist = args["distances"]     # {"A-B": 10, "A-C": 15, ...}
    n = len(cities)
    idx = {c: i for i, c in enumerate(cities)}
    # Build cost matrix
    c = [[0]*n for _ in range(n)]
    for edge, w in dist.items():
        u, v = edge.split("-")
        c[idx[u]][idx[v]] = c[idx[v]][idx[u]] = w
```

(continued on next slide)

TSP Solver (continued)

```
# Gurobi ILP (MTZ formulation)
m = gp.Model(); m.Params.OutputFlag = 0
x = {(i,j): m.addVar(vtype=GRB.BINARY)
     for i in range(n) for j in range(n) if i != j}
u = {i: m.addVar(lb=1, ub=n-1) for i in range(1, n)}
m.setObjective(gp.quicksum(
    c[i][j]*x[i,j] for i,j in x), GRB.MINIMIZE)
for i in range(n):  # degree constraints
    m.addConstr(gp.quicksum(x[i,j] for j in range(n)
                            if j!=i) == 1)
    m.addConstr(gp.quicksum(x[j,i] for j in range(n)
                            if j!=i) == 1)
for i in range(1,n):  # MTZ subtour elimination
    for j in range(1,n):
        if i != j:
            m.addConstr(u[i] - u[j] + (n-1)*x[i,j]
                        <= n - 2)
```

TSP Solver (continued)

```
m.optimize()
# Reconstruct tour
tour, cur = [cities[0]], 0
for _ in range(n - 1):
    nxt = [j for j in range(n)
           if j!=cur and x[cur,j].X > 0.5][0]
    tour.append(cities[nxt]); cur = nxt
tour.append(cities[0])
return {"content": [{"type": "text",
        "text": f"Tour: {' -> '.join(tour)}, "
                f"cost: {m.ObjVal:.0f}"}]}
```

Real Gurobi, real MTZ formulation. Same solver you used in HW1–5.

Building a CS498 Problem Solver

Goal: an agent solving algorithmic problems by combining reasoning and computation.

```
from claude_agent_sdk import query, ClaudeAgentOptions
# Create MCP server
server = create_sdk_mcp_server(name="cs498_tools", version="1.0.0", tools=[solve_tsp])

PROBLEM = """
A delivery truck must visit 6 cities and return home.
Distances: A-B: 10, A-C: 15, A-D: 20, B-C: 35, B-D: 25, C-D: 30, B-E: 12, C-F: 10, D-F: 22, E-F: 8.
Find the shortest tour visiting all cities exactly once.
"""
async def main():
    options = ClaudeAgentOptions(mcp_servers={"cs498_tools": server},
        allowed_tools=["mcp___cs498_tools___solve_tsp"],
        system_prompt="You have a solve_tsp tool.")
    async for msg in query(prompt=PROBLEM, options=options):
        if hasattr(msg, 'content'): print(msg.content)
```

Note: mcp_servers connects the Gurobi tool to the agent. Tool name follows the convention mcp___{server}___{tool}.

Real Trace: What Actually Happened (1/2)

We ran this. Here is the actual output, unedited:

Step 1: Agent loads the tool schema (deferred tools load on demand).

[ToolSearch: select:mcp___cs498__tools___solve_tsp]

Step 2: Agent reads the problem, extracts cities and distances. Tries a nested dict format:

solve_tsp(cities=["A",...], distances={"A":{"B":10,...},...})

Step 3: ERROR. Our tool expects "A-B": 10, not nested dicts.

ERROR: not enough values to unpack (expected 2, got 1)

The agent got it wrong on the first try. What does it do?

Real Trace: What Actually Happened (2/2)

Step 4: Agent reads the error, reasons about what went wrong, and tries the flat key format:

```
solve_tsp(cities=["A","B","C","D","E","F"],
          distances={"A-B":10, "A-C":15, ..., "E-F":8})
```

Step 5: Gurobi solves it.

```
Optimal tour: A -> D -> B -> E -> F -> C -> A, cost: 90
```

Step 6: Agent verifies the answer:

$20 + 25 + 12 + 8 + 10 + 15 = 90$ ✓

Step 7: Presents the final answer with explanation.

The error recovery is the point. The agent debugged itself: read the error, changed the input format, and succeeded on retry. This is what makes agents useful.

Fixing the System Prompt

```
options = ClaudeAgentOptions(
    mcp_servers={"cs498_tools": server},
    allowed_tools=["mcp__cs498_tools__solve_tsp"],
    system_prompt="""You have a solve_tsp tool. It expects: - cities: list of strings, e.g. ["A", "B", "C"]
    - distances: flat dict with "X-Y" keys, e.g. {"A-B": 10, "B-C": 20}. Do NOT use nested dicts.""")
```

[ThinkingBlock: TSP with 6 cities. Use `solve_tsp` tool. Load schema via ToolSearch first.]

[ToolUseBlock: ToolSearch(query='select:mcp__cs498_tools__solve_tsp')]

[ToolResultBlock: schema loaded for mcp__cs498_tools__solve_tsp]

[ThinkingBlock: Schema loaded. Cities: A–F. Distances use flat "X-Y" keys: A-B:10, ..., E-F:8. Invoke the tool.]

[TextBlock: "Now let me call the TSP solver with all 6 cities and the provided distances!"]

[ToolUseBlock: mcp__cs498_tools__solve_tsp(cities=['A','B',...,'F'], distances={'A-B':10, ..., 'E-F':8})]

[ToolResultBlock: "Optimal tour: A -> D -> B -> E -> F -> C -> A, cost: 90"]

[ThinkingBlock: Verify: $20 + 25 + 12 + 8 + 10 + 15 = 90$]

[TextBlock: Final answer: Optimal route: A → D → B → E → F → C → A. Total cost: 90.]

Why This Matters

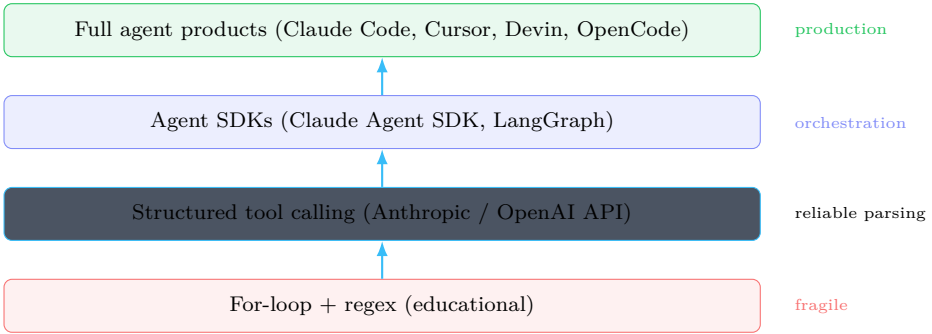
The TSP agent is a toy example. The pattern is general:

Domain	Agent = LLM + …
Software engineering	LLM + file I/O + shell + tests
Data science	LLM + pandas + matplotlib + SQL
Math research	LLM + Lean/Coq + computation
Chip design	LLM + EDA tools + simulation
Scientific computing	LLM + solvers + data analysis

The LLM provides reasoning and planning. The tools provide reliable execution.

Neither is sufficient alone: the LLM can't multiply large numbers, and the solver can't decide which formulation to use.

Agents: The Stack



Every layer is the same generate → parse → act → observe loop.

The difference is robustness and what tools are available.

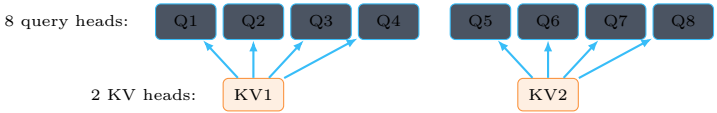
Claude Code is at the top of this stack.

Grouped Query Attention (GQA)

Full multi-head attention: each head has its own W_K and W_V.

Problem: the KV cache scales linearly with number of KV heads.

GQA: multiple query heads share the same K and V head.



Llama-3 8B: 32 query heads, 8 KV heads. KV cache is 4× smaller.

Quality loss is minimal. Inference speedup is significant.

Mixture of Experts: The Idea

Recall: each transformer block has a FFN that processes every token independently.

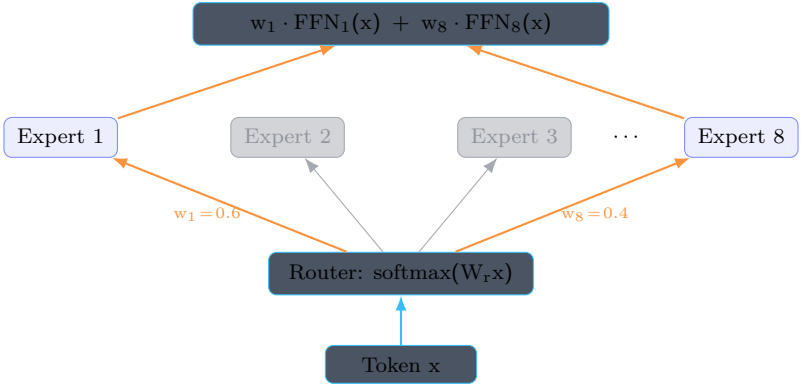
In a standard transformer, every token goes through the same FFN. The FFN has $\sim 8d^2$ parameters and all of them are used for every token.

Observation: not every token needs the same processing. The word "theorem" might benefit from "math expert" neurons, while "DNA" benefits from "biology expert" neurons.

MoE idea: instead of one big FFN, have E separate FFNs ("experts"). For each token, a small router network picks the top k experts. Only those k experts run.

The attention layer is unchanged. Only the FFN is replaced.

Mixture of Experts: How It Works



- Router: a tiny linear layer $W_r \in \mathbb{R}^{E \times d}$ that scores each expert. Pick top-k, softmax over those k to get weights $w_1, \ldots, w_k$.
- Each expert: a full SwiGLU FFN (same architecture as L26, $\sim 8d^2$ params each).

MoE: Why Bother?

	Dense (Llama-3 8B)	MoE (Mixtral 8x7B)
Total params	8B	47B
Active params per token	8B	13B
Inference cost per token	$\propto$ 8B	$\propto$ 13B
Quality	Good	Much better

Mixtral has 47B total parameters (8 experts $\times$ 7B FFN params each, plus shared attention). But only 2 experts fire per token, so inference cost is like a 13B model.

Memory: you still need to store all 47B params in GPU memory. MoE saves compute, not memory. This is why MoE models need more GPUs than their active parameter count suggests.

Speculative Decoding: The Problem

Decode is slow: one token at a time, each requiring a full forward pass through the big model.

Observation: most tokens are "easy." After "The capital of France is", the next token is almost certainly "Paris." You don't need 70B parameters to predict that.

Idea: use a small, fast draft model (e.g., 1B params) to guess the next k tokens. Then check whether the big model would have generated the same tokens.

If the draft model's guesses are good (they usually are for easy tokens), we skip k forward passes of the big model and do just one.

Speculative Decoding: The Algorithm

1. Draft: small model generates k tokens $t_1, t_2, \ldots, t_k$ autoregressively (fast).
2. Score: big model processes all k tokens in one forward pass (parallel, like prefill). This gives us $p_{big}(t_i \mid \text{context})$ for each position.
3. Accept/reject: for each token t_i in order:
   - Compare $p_{big}(t_i)$ to $p_{draft}(t_i)$
   - If $p_{big} \geq p_{draft}$: accept (big model agrees or is even more confident)
   - If $p_{big} < p_{draft}$: accept with probability p_{big}/p_{draft}, otherwise reject and resample from an adjusted distribution
4. Stop at first rejection. Keep all accepted tokens.

Key property: this acceptance scheme guarantees the output distribution is the same as the big model alone. No quality loss. This is provable (rejection sampling).

Putting It All Together

Modern LLM serving combines all of these:

Technique	What it does
KV cache	Avoid recomputing past keys/values
GQA	Shrink the KV cache (fewer KV heads)
PagedAttention	Manage KV cache memory efficiently
Continuous batching	Keep the GPU busy across requests
Quantization	Fit larger models in less memory
MoE	More parameters, same inference cost
Speculative decoding	Generate multiple tokens per step

vLLM implements most of these out of the box. You load a model, call generate, and it handles the rest.