

Лекция 9: Локални LLM модели

Quantization, Hardware и Deployment

Цели на лекцията

- Защо локални модели — privacy, цена, контрол
- Quantization — как правим моделите по-малки
- Hardware — какво ни трябва за различни модели
- Инструменти — Ollama, vLLM, llama.cpp
- Избор на модел според use case

Част 1: Предизвикателството

Защо локални LLM?

Причина	Обяснение
Privacy	Данните не напускат машината
Цена	Без API такси при голям обем
Latency	Няма network overhead
Контрол	Без content filters
Offline	Работи без интернет

Фундаменталният проблем

LLaMA 70B (full precision):
 $70B \times 4 \text{ bytes} = 280 \text{ GB RAM}$

Типичен laptop:
16 GB RAM

Gap: 17.5x 🤖

Решение: Quantization + Hardware optimization

Три стълба на локален inference

Local LLM		
Quantization (по-малки тегла)	Efficient Inference (по-бързо изчисление)	Hardware Utilization (използвай всичко)

Част 2: Memory Requirements

Къде отива паметта?

Total Memory	
Weights (static)	KV Cache (dynamic)
Зависи от model size	Расте с context length

Пресмятане на памет за weights

Формула:

$$\text{Memory} = \text{Parameters} \times \text{Bytes per Parameter}$$

Model	FP32	FP16	INT8	INT4
7B	28 GB	14 GB	7 GB	3.5 GB
13B	52 GB	26 GB	13 GB	6.5 GB
70B	280 GB	140 GB	70 GB	35 GB

INT4 = 8x по-малко от FP32!

KV Cache: Динамичната част

$$\text{KV Cache} = 2 \times L \times H \times D \times S \times B$$

- L = layers (напр. 32)
- H = heads (напр. 32)
- D = head dimension (напр. 128)
- S = sequence length
- B = batch size

7B модел, 4096 tokens: ~4 GB допълнително

Memory Bandwidth Bottleneck

Ключов insight: LLM inference e **memory-bound**, не compute-bound

$\text{Tokens/sec} \approx \text{Memory Bandwidth} / \text{Model Size}$

Hardware	Bandwidth	7B INT4	70B INT4
DDR4 RAM	50 GB/s	14 t/s	1.4 t/s
Apple M2	100 GB/s	28 t/s	2.8 t/s
RTX 4090	1000 GB/s	285 t/s	28 t/s

Част 3: Quantization

Какво е Quantization?

Mapping от high-precision към low-precision:

FP32: -3.14159265... (32 bits)

↓

INT8: -3 (8 bits)

Цел: Запази колкото се може повече информация с по-малко битове

Linear Quantization

Формула:

$$x_q = \text{round} \left(\frac{x - z}{s} \right)$$

- s = scale factor
- z = zero point

Dequantization:

$$\hat{x} = s \cdot x_q + z$$

Quantization Granularity

Per-Tensor: Един scale за целия тензор
[████████████████████]
scale = 0.5

Per-Channel: Scale за всеки канал
[████] [████] [████]
s=0.3 s=0.5 s=0.7

Per-Group: Scale за група стойности
[██][██] [██][██] [██][██]
s1 s2 s3 s4 s5 s6

Per-group е най-точно, но по-бавно

Bit Levels и Trade-offs

Bits	Размер	Качество	Use case
FP16	2x по-малко	~100%	Training, inference
INT8	4x по-малко	~99%	Production
INT4	8x по-малко	~95-98%	Consumer hardware
INT3	10.7x	~90-95%	Edge devices
INT2	16x	~80-90%	Experimental

Sweet spot: INT4 (Q4) за повечето случаи

Advanced Quantization Methods

GPTQ (2022)

- Post-training quantization
- Layer-by-layer, минимизира reconstruction error
- Популярен за 4-bit модели

AWQ (2023)

- Activation-aware
- Пази "важните" weights с по-висока precision
- По-добро качество от GPTQ

GGUF K-quants

llama.cpp формат с mixed precision:

Quant	Bits	Описание
Q4_K_S	~4.5	Smallest 4-bit
Q4_K_M	~4.8	Medium 4-bit (recommended)
Q5_K_S	~5.5	Smallest 5-bit
Q5_K_M	~5.8	Medium 5-bit
Q6_K	~6.6	Best quality

K = важните слоеве са по-малко quantized

Качество vs Размер



Q4_K_M е оптималният компромис за повечето

Практически съвети

Use case	Препоръчително
Максимален размер	Q2_K, Q3_K
Consumer GPU (8GB)	Q4_K_M
Consumer GPU (16GB+)	Q5_K_M, Q6_K
Server	Q8_0, FP16
Code completion	Q5+ (precision matters)

Част 4: Други оптимизации

KV Cache Optimization

Grouped Query Attention (GQA)

- Споделя KV heads между Q heads
- 8x по-малък KV cache

Sliding Window Attention

- Внимание само на последните N токени
- Константен KV cache size

PagedAttention (vLLM)

- Virtual memory за KV cache
- По-добро batch utilization

Flash Attention

Проблем: Standard attention e $O(n^2)$ memory

Решение: Tiling + recomputation

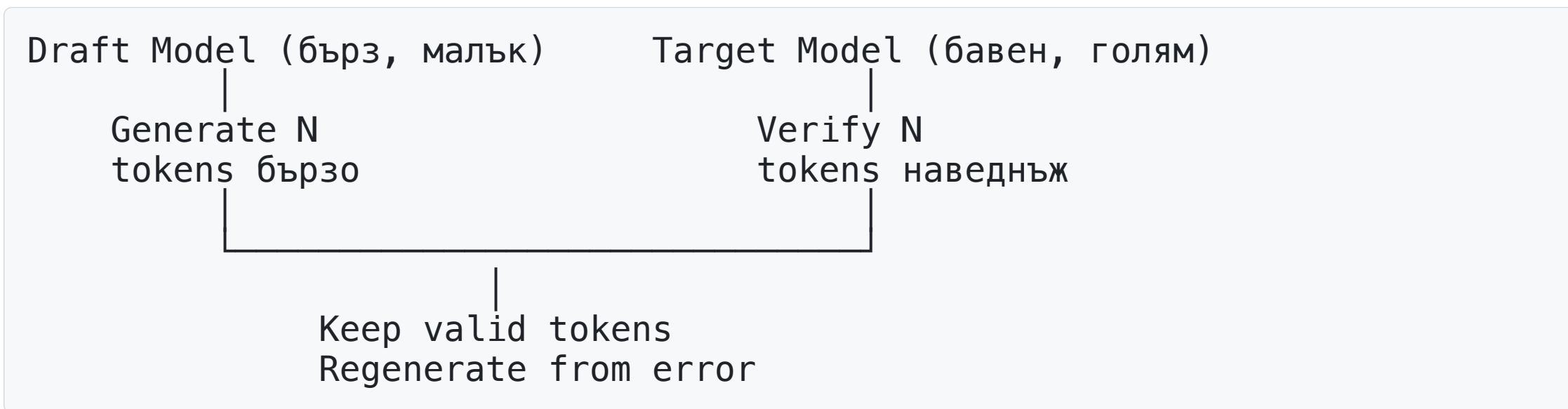
Standard: Load all Q,K,V
Compute full attention
Store full result

↓

Flash: Load blocks
Compute partial
Accumulate on-the-fly

2-4x по-бързо, много по-малко memory

Speculative Decoding



Speedup: 2-3x когато acceptance rate е висок

Mixture of Experts (MoE)

Input → Router → Expert 1
Expert 2
Expert 3
Expert 4 (inactive)
...



The diagram illustrates the Mixture of Experts (MoE) architecture. It shows a flow from 'Input' to a 'Router', which then directs the data to a set of experts. Experts 1, 2, and 3 are shown with lines connecting them to a central point, which then leads to the 'Output'. Expert 4 is labeled '(inactive)'. Ellipses (...) indicate that there can be more than four experts in total.

- Само 2-4 experts активни от 8-64 total
- Повече параметри, същият compute
- Пример: Mixtral 8x7B = 47B params, 13B active

Част 5: Hardware

Ключови ограничения

Priority:

1. Memory (VRAM/RAM) ← Модел трябва да се събере
2. Bandwidth ← Определя tokens/sec
3. Compute ← Рядко е bottleneck

Правило: Първо провери дали се събира,
после оптимизирай за скорост

CPU-Only Inference

Кога:

- Нямаш GPU
- Модел не се събира в VRAM
- Background tasks

Очаквания:

- 7B Q4: 5-15 tokens/sec
- 13B Q4: 2-8 tokens/sec
- 70B Q4: 0.5-2 tokens/sec

Tools: llama.cpp, Ollama

Consumer GPUs

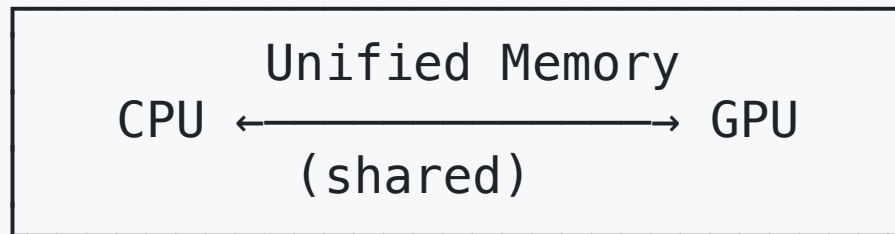
GPU	VRAM	Max Model (Q4)	Speed
RTX 3060	12GB	7B	30-50 t/s
RTX 3090	24GB	13B	40-60 t/s
RTX 4090	24GB	13B	80-120 t/s

Hybrid offloading: Part GPU + Part CPU

- По-бавно от pure GPU
- Но позволява по-големи модели

Apple Silicon

Предимство: Unified Memory Architecture (UMA)



Chip	RAM	Max Model	Speed
M1	16GB	7B Q4	15-25 t/s
M2 Pro	32GB	13B Q4	20-35 t/s
M3 Max	128GB	70B Q4	15-25 t/s

Server Hardware

GPU	VRAM	Use Case
A100	40/80GB	Training + Inference
H100	80GB	State of the art
RTX A6000	48GB	Inference focused

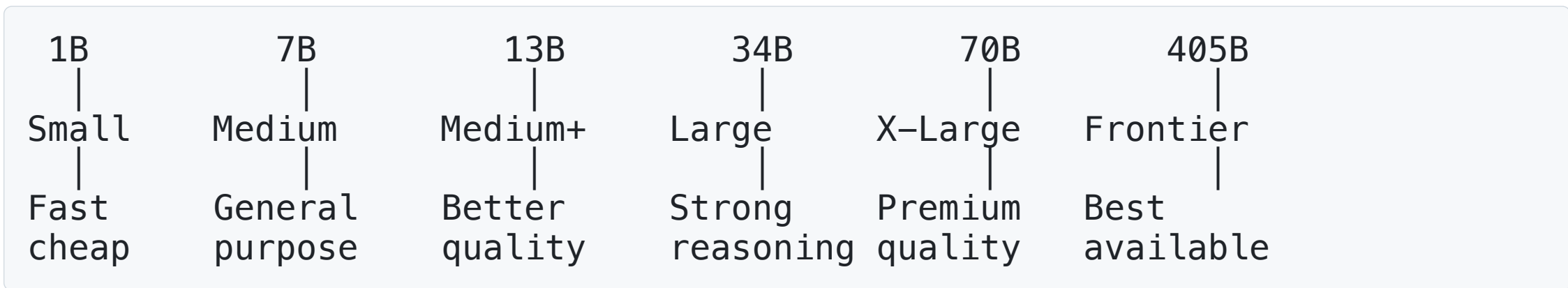
Multi-GPU: Tensor parallelism за много големи модели

Hardware Selection Guide

Budget	Model?	Use Case?
< \$500	→ CPU only	(Ollama)
\$500–1500	→ RTX 3060/4060	(7B)
\$1500–3000	→ RTX 4090	(13B)
\$3000+	→ Mac Studio или Server	

Част 6: Model Sizes и Use Cases

Спектърът на размерите



Small Models (1-3B)

Силни страни:

- Много бързи (100+ t/s на GPU)
- Работят на телефони
- Добри за специфични задачи

Use cases:

- Code completion (в IDE)
- Embeddings
- Classification
- Edge deployment

Примери: Phi-2, TinyLlama, Gemma 2B

Medium Models (7-13B)

Силни страни:

- Добър баланс качество/скорост
- Работят на consumer hardware
- General purpose

Use cases:

- RAG chatbots
- Summarization
- Translation
- General assistance

Примери: LLaMA 3 8B, Mistral 7B, Qwen2 7B

Large Models (34-70B)

Силни страни:

- Силно reasoning
- Следват сложни инструкции
- По-малко hallucinations

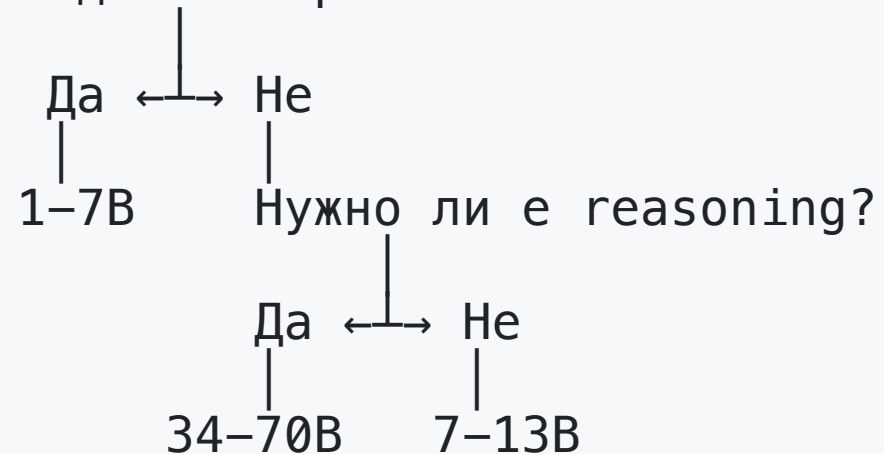
Use cases:

- Code generation
- Complex analysis
- Enterprise applications
- Когато качеството е критично

Примери: LLaMA 3 70B, Qwen2 72B, DeepSeek 67B

Как да изберем?

Задачата проста?



Quality vs Latency Trade-off

Модел	First Token	Tokens/sec	Quality
3B Q4	50ms	100+	★ ★
7B Q4	100ms	50-80	★ ★ ★
13B Q4	200ms	30-50	★ ★ ★ ★
70B Q4	500ms	10-20	★ ★ ★ ★ ★

За interactive: първите са по-добри

За batch processing: качеството има приоритет

Част 7: Deployment Tools

llama.cpp

Какво е: C++ inference engine за GGUF модели

Кога да го използваш:

- Нужен е максимален контрол
- Custom integration
- Ресурсно ограничена среда

```
./main -m model.gguf -p "Hello" -n 100
```

Ollama

Какво е: Docker-like experience за LLMs

```
# Install
curl -fsSL https://ollama.ai/install.sh | sh

# Pull модел
ollama pull llama3:8b

# Run
ollama run llama3:8b

# API
curl http://localhost:11434/api/generate \
  -d '{"model": "llama3:8b", "prompt": "Hello"}'
```

Ollama: Защо е добър?

- ✓ Лесен setup (една команда)
- ✓ Model management (pull, rm, list)
- ✓ OpenAI-compatible API
- ✓ Automatic GPU detection
- ✓ Model library (ollama.ai/library)
- ✗ Не е за production throughput
- ✗ Ограничени batch capabilities

vLLM

Какво е: High-throughput inference server

Кога да го използваш:

- Production deployment
- Много concurrent requests
- Максимален throughput

```
from vllm import LLM, SamplingParams

llm = LLM(model="meta-llama/Llama-3-8B")
outputs = llm.generate(["Hello"], SamplingParams())
```

vLLM Features

- **PagedAttention:** Ефективно memory management
- **Continuous batching:** Максимален GPU utilization
- **Tensor parallelism:** Multi-GPU support
- **OpenAI-compatible API**

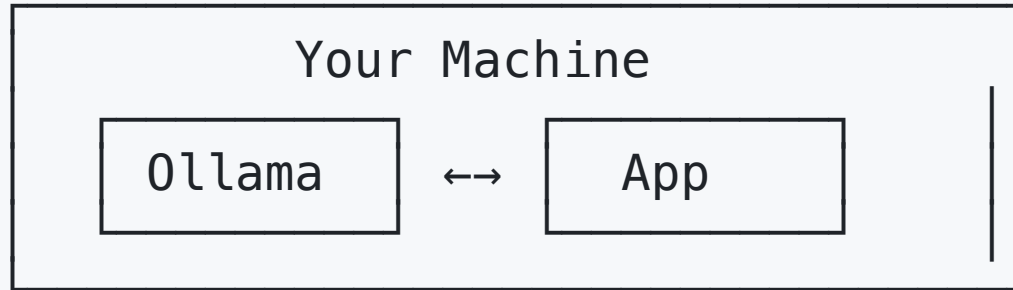
```
python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Llama-3-8B
```

Tool Selection Guide

Нужда	Инструмент
Personal use	Ollama
Integration	llama.cpp
Production API	vLLM
Apple Silicon	Ollama или mlx
Maximum control	llama.cpp

Част 8: Deployment Patterns

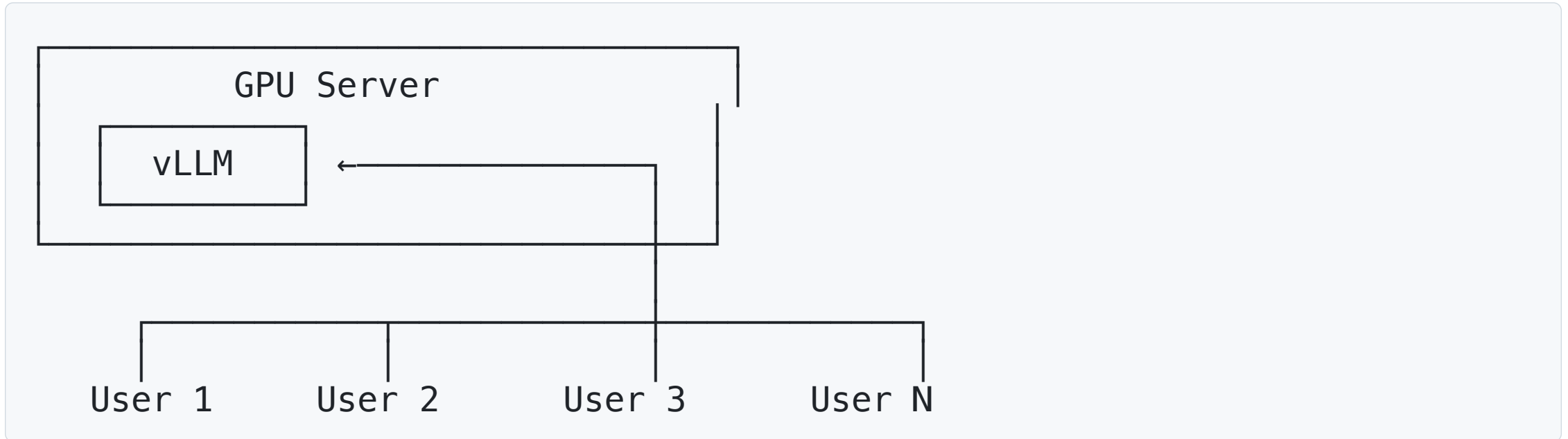
Pattern 1: Single User Local



Pros: Full privacy, no cost, offline

Cons: Limited by your hardware

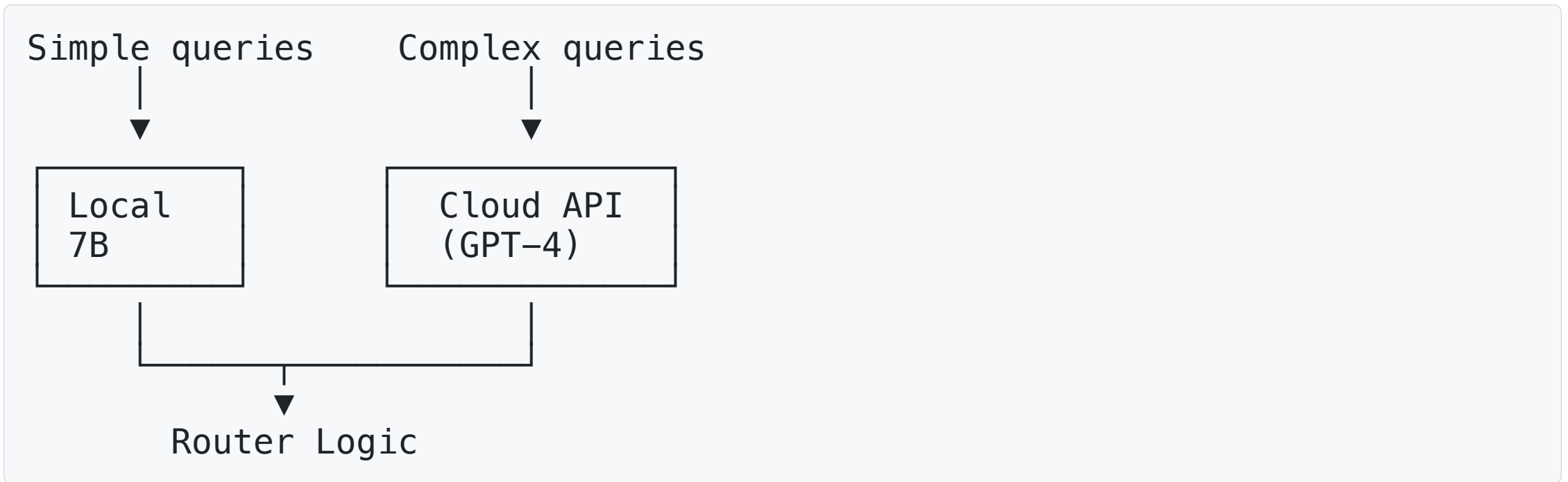
Pattern 2: Team Server



Pros: Better utilization, consistent experience

Cons: Need server management

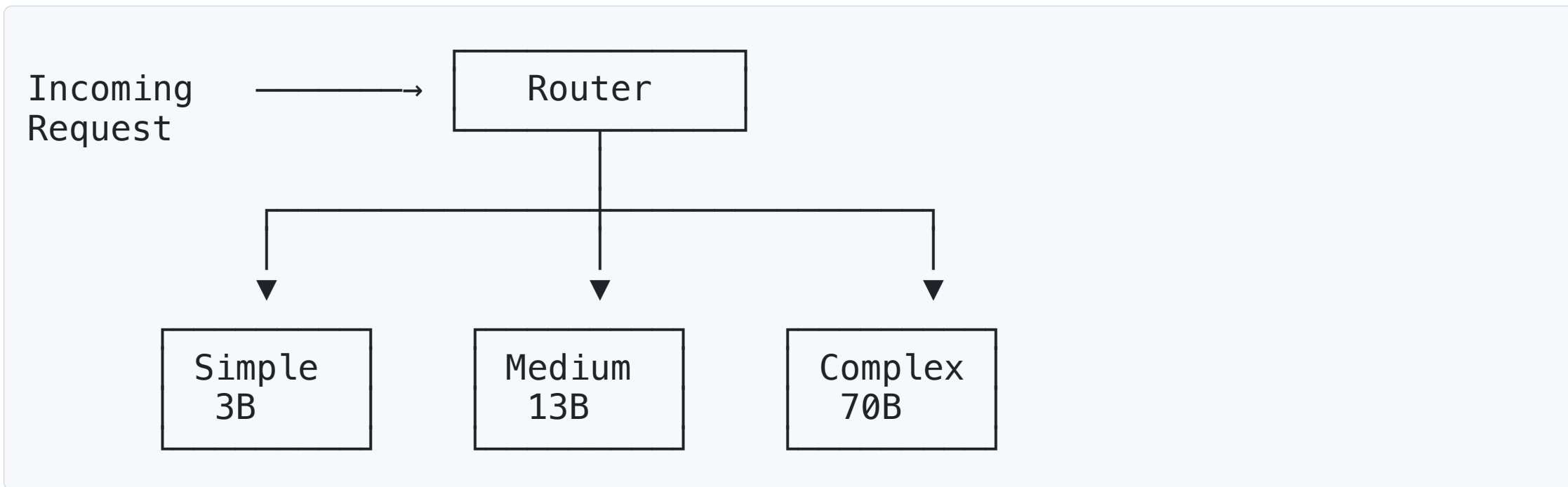
Pattern 3: Hybrid Local + Cloud



Pros: Cost optimization, best of both worlds

Cons: More complex architecture

Pattern 4: Model Routing



Класифицирай заявката → Изпрати до подходящ модел

Обобщение

Ключови идеи

1. Quantization е ключът

- Q4 прави 70B модели достъпни на consumer hardware
- Trade-off качество/размер е управляем

2. VRAM/RAM е главното ограничение

- Memory bandwidth определя скоростта

3. Избирай модел според задачата

- Small за скорост, large за качество

4. Tools: Ollama за простота, vLLM за production

Практически Quick Start

```
# 1. Инсталирай Ollama
curl -fsSL https://ollama.ai/install.sh | sh

# 2. Изтегли модел
ollama pull llama3:8b

# 3. Пробвай
ollama run llama3:8b "Explain quantum computing"

# 4. Използвай API
curl http://localhost:11434/api/generate \
  -d '{"model":"llama3:8b","prompt":"Hello"}'
```

Следваща лекция

Лекция 10: Advanced Prompting и Reasoning Models

- Chain-of-Thought и варианти
- Few-shot vs Zero-shot
- Reasoning models (o1, DeepSeek R1)
- Prompt engineering best practices

Ресурси

Papers:

- Frantar et al. (2022) — GPTQ
- Lin et al. (2023) — AWQ
- Kwon et al. (2023) — vLLM/PagedAttention

Tools:

- ollama.ai
- github.com/ggerganov/llama.cpp
- github.com/vllm-project/vllm

Въпроси?