0. Orientation: What "AI Engineer" Means Today (1 week)

Before tools, we align on the role.

What top firms expect

- Not "model training only"
- You design **systems around LLMs**
- You build **reliable, scalable, controllable agents**
- You understand **trade-offs, failure modes, and evaluation**

Key distinctions

- Data Scientist vs ML Engineer vs AI Engineer
- GenAI Engineer vs Agentic AI Engineer
- Research vs Production

1. Advanced Foundations of Generative AI (You go deeper than "LLMs") (2–3 weeks)

You already know LLMs; here we go **under the hood and beyond**.

1.1 Transformer internals (production-level understanding)

- Attention variants (MHA, GQA, MQA)
- KV-cache, memory bandwidth bottlenecks
- RoPE, ALiBi, positional encoding trade-offs
- Scaling laws (Chinchilla, compute-optimal training)

1.2 Tokenization & representation

- BPE vs SentencePiece vs Unigram LM
- Why tokenization failures break reasoning
- Token budgets and cost optimization

1.3 Decoding & controllability

- Greedy, beam, top-k, top-p, typical decoding
- Logit biasing
- Temperature vs entropy trade-offs
- Hallucination mechanics

Outcome:

You can reason about *why* an LLM behaves poorly—not just observe it.

2. Prompt Engineering → Prompt Programming (2 weeks)

Top engineers do not "prompt craft"; they **design prompt systems**.

2.1 Prompt design patterns

- Role prompting
- Chain-of-Thought vs Scratchpads
- ReAct pattern
- Self-consistency
- Tree-of-Thoughts
- Constitutional prompting

2.2 Structured prompting

- JSON schemas
- Function calling
- Tool schemas
- Guardrails and validators

2.3 Prompt versioning & testing

- Prompt unit tests
- Regression testing prompts
- Prompt evaluation metrics

Outcome:

You write prompts as **software artifacts**, not text blobs.

3. Retrieval-Augmented Generation (RAG) — Properly Done (3 weeks)

Most RAG systems fail. You learn **why** and how to fix them.

3.1 Retrieval fundamentals

- Dense vs sparse retrieval
- Hybrid retrieval
- Chunking strategies (semantic, hierarchical)
- Embedding model selection

3.2 Vector databases & indexing

- FAISS internals
- HNSW vs IVF
- Recall–latency trade-offs
- Metadata filtering

3.3 RAG architectures

- Naive RAG (baseline)
- Multi-query RAG
- Parent–child RAG
- Graph-based RAG
- Agentic RAG

3.4 Evaluation of RAG

- Faithfulness
- Context precision / recall
- Answer relevance

- Synthetic data generation for eval

Outcome:

You can build RAG systems that **scale, stay factual, and are evaluatable**.

4. Fine-Tuning & Model Adaptation (2–3 weeks)

You will not train 100B models—but you *must* know adaptation.

4.1 Fine-tuning methods

- Full fine-tuning vs PEFT
- LoRA, QLoRA, adapters
- When fine-tuning beats prompting

4.2 Alignment tuning

- Instruction tuning
- Preference tuning
- Basics of RLHF, DPO, PPO (engineering view)

4.3 Domain adaptation

- Synthetic data pipelines
- Data curation strategies
- Catastrophic forgetting

Outcome:

You can justify **when** to fine-tune and **when not to**.

5. Agentic AI — Core of Modern Systems (4–5 weeks)

This is where you become **rare and valuable**.

5.1 What is an agent (formally)

- Perception → Memory → Reasoning → Action → Feedback
- Stateless vs stateful agents
- Tool-using agents vs autonomous agents

5.2 Agent architectures

- ReAct agents
- Planner–Executor
- Reflexion
- Self-improving agents
- Multi-agent systems (MAS)

5.3 Tool use & function calling

- Tool abstraction
- Tool reliability
- Error recovery
- Cost-aware tool selection

5.4 Memory systems

- Short-term vs long-term memory
- Episodic memory
- Vector memory vs symbolic memory
- Memory decay strategies

Outcome:

You can design agents that **act reliably**, not just chat.

6. Agent Frameworks (Engineering Focus) (2 weeks)

Frameworks are **implementation details**, not the skill—but you must know them.

6.1 LangChain (deep, not superficial)

- LCEL
- Chains vs agents
- Callbacks
- Custom tools & retrievers

6.2 LangGraph

- Stateful agent workflows
- Cycles, interrupts, human-in-the-loop

6.3 Alternatives

- CrewAI
- AutoGen
- Semantic Kernel
- Why many frameworks fail at scale

Outcome:

You can **choose or avoid** frameworks intelligently.

7. Reliability, Safety & Alignment (2 weeks)

This is what separates production systems from demos.

7.1 Failure modes

- Hallucination
- Tool misuse
- Prompt injection
- Data poisoning

7.2 Guardrails

- Input/output validation
- Policy enforcement
- Sandboxing tools
- Red-teaming

7.3 Observability

- Traces
- Token usage
- Cost monitoring
- Error analytics

Outcome:

You can build systems that **do not embarrass companies**.

8. Evaluation & Benchmarking (2 weeks)

Top firms care deeply about evaluation.

8.1 Offline evaluation

- Task-specific metrics
- LLM-as-judge (pros/cons)
- Human evaluation pipelines

8.2 Online evaluation

- A/B testing
- Canary deployments
- User feedback loops

Outcome:

You can prove your system is **actually better**, not “feels better”.

9. Scaling & Production Systems (3 weeks)

This is where many ML engineers fail.

9.1 System design

- Async inference
- Streaming responses
- Caching strategies
- Batch vs real-time

9.2 Cost optimization

- Token reduction
- Model routing
- Distillation
- Caching embeddings & outputs

9.3 Deployment

- APIs
- Kubernetes basics for GenAI
- Model serving patterns

Outcome:

You can deploy GenAI systems that **scale and stay affordable**.

10. Advanced Topics (Electives for Top Firms)

You choose based on interest.

Options:

- Multimodal GenAI (text-image-audio-video)
- Code generation agents
- Research agents
- Autonomous data analysis agents
- Long-horizon planning agents
- Neuro-symbolic agents

11. Portfolio Projects (Mandatory)

No one hires without this.

Capstone projects I would assign:

1. **Production-grade RAG system**
2. **Multi-tool autonomous agent**
3. **Domain-specific AI copilot**
4. **Evaluation framework for GenAI**
5. **Cost-optimized agent pipeline**

Outcome:

Your GitHub looks like an **AI engineer**, not a student.

Final Reality Check (Important)

If you complete this path:

- You are **not just "learning GenAI"**
- You are learning **AI systems engineering**
- You become competitive for:
  - AI Engineer
  - Applied Scientist

- GenAI Engineer
- Agent Engineer

Why Decoder-Only (GPT-Style) Models Dominate Open-Ended Reasoning

1. The Three Competing Architectures (Quick Alignment)

Before we reason about *why*, we must be precise about *what*.

Architecture	Examples	Core Use
Encoder-only	BERT	Understanding, classification
Encoder–Decoder	T5, BART	Translation, summarization
Decoder-only	GPT, LLaMA	Reasoning, dialogue, agents

2. The Central Insight (Key Takeaway Upfront)

Open-ended reasoning is fundamentally an *autoregressive simulation problem*, not a conditional mapping problem.

Decoder-only models are *natively aligned* with this reality.

Everything else flows from this.

3. Encoder–Decoder Models: Their Hidden Assumption

Encoder–decoder models assume:

“The full input is known *before* generation begins.”

This is perfect for:

- Translation
- Summarization
- Style transfer

But reasoning does **not** behave this way.

Why This Matters

In reasoning:

- The *input grows as you think*
- Earlier tokens affect how later tokens are interpreted
- The model must continuously reinterpret its own output

Encoder–decoder models **freeze the encoder representation** before decoding starts.

That is the fundamental limitation.

4. Decoder-Only Models: Continuous Re-Interpretation

Decoder-only models operate as:

A single evolving state over **(prompt + all prior thoughts)**

Every new token:

- Re-enters attention
- Reconditions the entire context
- Alters future reasoning

This enables:

- Chain-of-thought
- Self-correction
- Multi-step planning
- Tool invocation loops

5. Attention Geometry (This Is the Real Reason)

Let's be exact.

Encoder–Decoder Attention

- Encoder self-attends **input only**
- Decoder cross-attends to **static encoder output**
- Decoder self-attends to **generated tokens**

This creates **two representations**:

- One frozen
- One evolving

Decoder-Only Attention

- One attention space
- Everything attends to everything (causally)

This produces:

- Unified semantic space
- Emergent reasoning depth
- Better long-horizon coherence

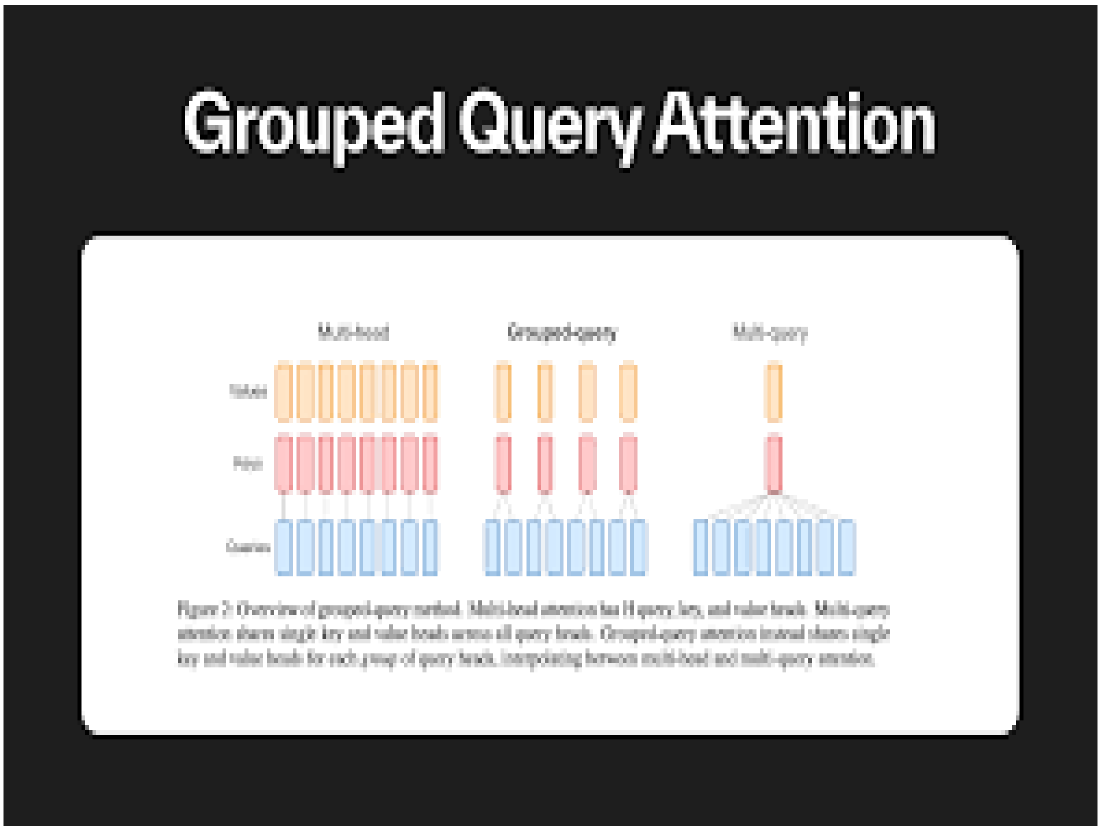
Reasoning prefers a single evolving state.

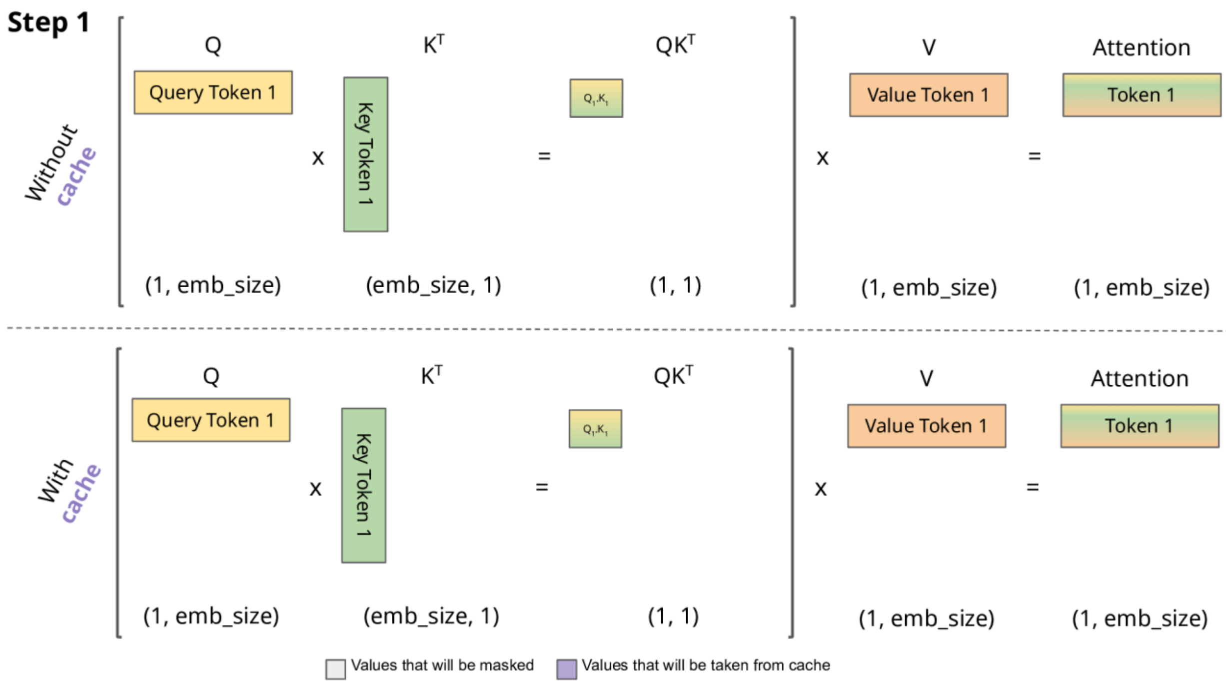
8. Agentic Behavior Requires Decoder-Only Structure

Agent loops require:

1. Think
2. Act
3. Observe

4. Think again





1.1 Multi-Head Attention (MHA)

What it does

- Each attention head has its **own Q, K, V projections**
- Heads attend independently
- Outputs are concatenated

Why it exists

- Different heads specialize:
  - Syntax
  - Coreference
  - Long-range dependencies
  - Local patterns

Mathematical view

If you have:

- h heads
- `d_model` hidden size

Then each head has:

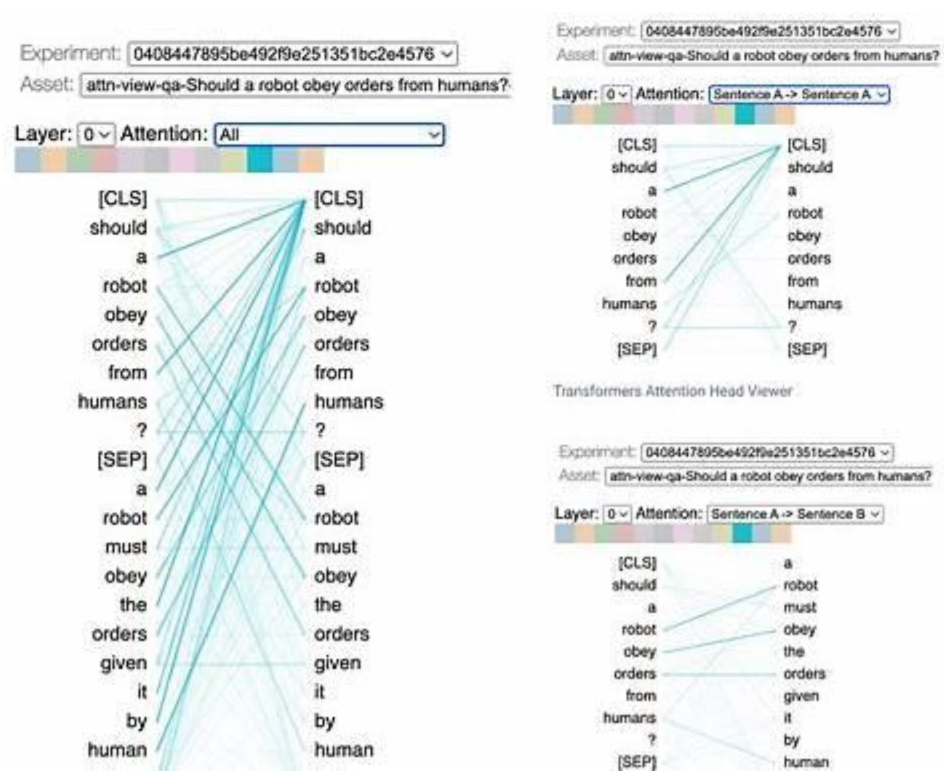
- `d_head = d_model / h`
- Separate **K and V matrices per head**

Cost implication

For every token generated, you must store and read:

```
O(sequence_length × num_heads × d_head)
```

This explodes at scale.



1.2 Multi-Query Attention (MQA)

What changes

- **Multiple Q heads**
- **Single shared K and V**

In other words:

- Queries stay independent
- Keys and Values are shared across all heads

Why this matters

- KV-cache size drops by **~num_heads×**
- Memory bandwidth requirement collapses

Trade-off

- Less expressive than MHA
- Heads lose some specialization

Where it works

- In very large models
- Where scale compensates for reduced per-head expressivity

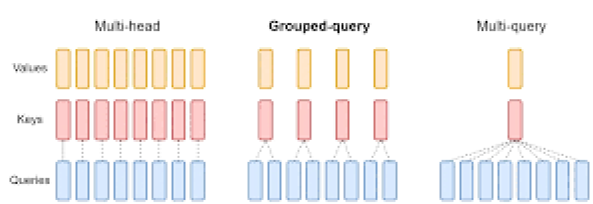
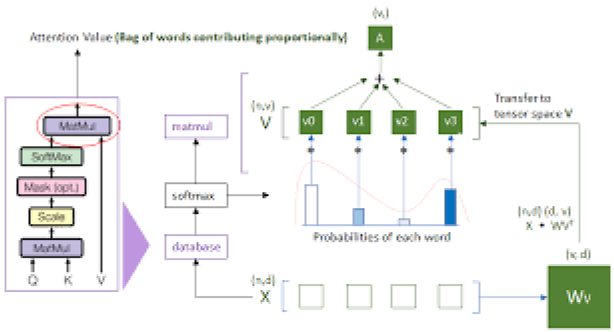


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.



1.3 Grouped-Query Attention (GQA)

The compromise solution

- Queries are split into **groups**
- Each group shares one K/V

Example:

- 32 query heads
- 8 KV groups
- Each KV shared by 4 Q heads

Why this is powerful

You get:

- Much lower KV memory cost than MHA
- More expressivity than MQA
- Near-MHA quality at a fraction of the cost

This is why **modern frontier models prefer GQA**.

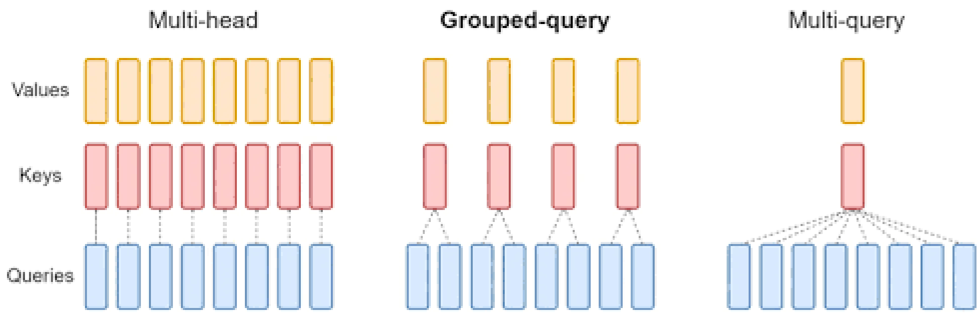
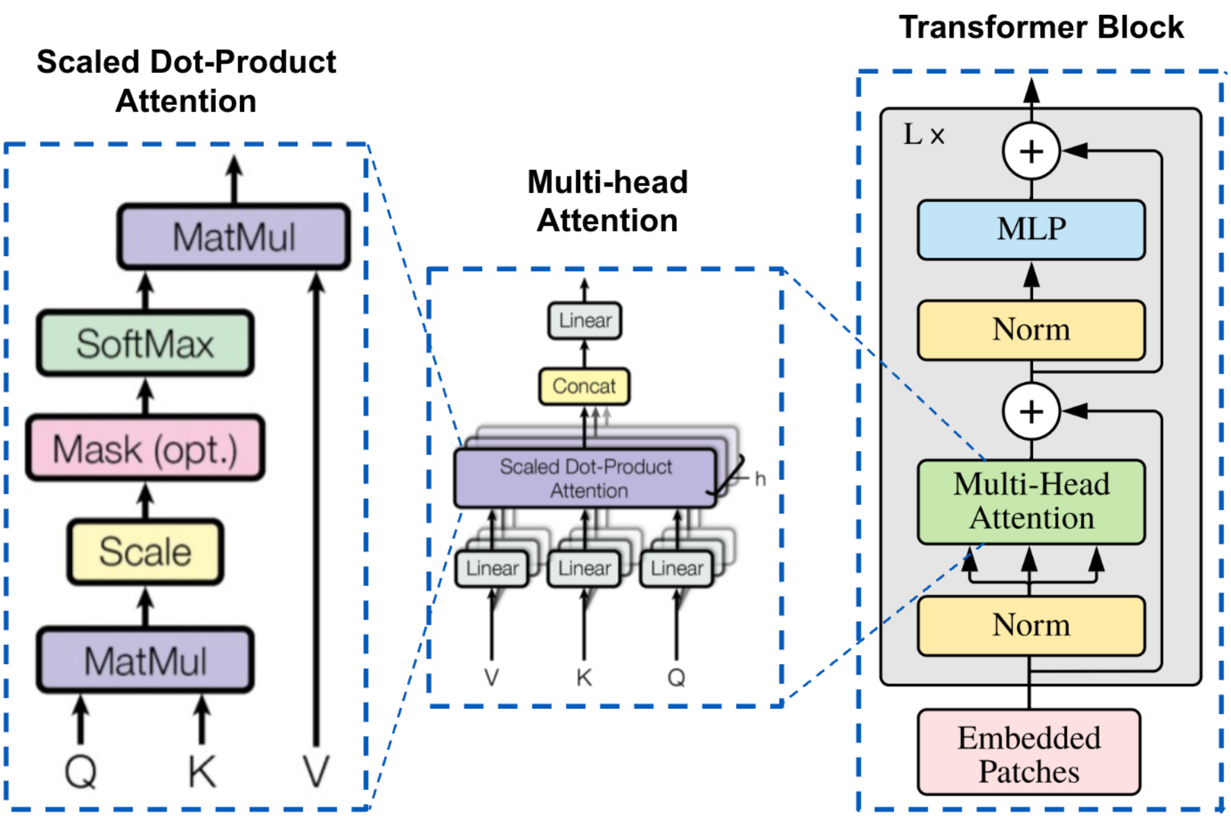


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.



2. Why GPT-4–Class Models Move Away from Full MHA

This decision is **not about accuracy**.
It is about **serving reality**.

2.1 The Real Bottleneck Is NOT FLOPs

Most people assume transformers are compute-bound.

They are not.

They are **memory-bandwidth bound** during inference.

Why?

- Attention requires reading **past K/V tensors**
- These tensors live in GPU memory (HBM)
- Each token generation requires massive memory reads

Even if the GPU has idle compute units, it **waits for memory**.

2.2 KV-Cache Growth Kills MHA at Scale

Let's be concrete.

Assume:

- Sequence length = 8,000
- Heads = 32
- d_head = 128

KV-cache size per layer:

```
2 × seq_len × num_heads × d_head
```

Multiply this by:

- Number of layers
- Batch size
- Concurrent users

This becomes **economically infeasible**.

2.3 What GPT-4–Class Systems Optimize For

Production systems optimize:

- Tokens per second
- Cost per 1K tokens
- Latency under load
- Memory per request

MHA fails on all four at large context lengths.

So the shift is:

- **MHA → GQA → sometimes MQA**

Not because MHA is “bad”, but because it is **too expensive to serve**.

2.4 Key Insight (Interview-Critical)

GPT-4–class models do not abandon MHA because of model quality; they abandon it because **memory bandwidth and KV-cache size dominate inference cost at scale**.

If you say this clearly, you sound like someone who has **actually deployed models**.

3. KV-Cache and Memory Bandwidth Bottlenecks

This is one of the **most important concepts** in GenAI systems.

3.1 What Is KV-Cache (Really)?

During autoregressive generation:

- Keys and Values for previous tokens **do not change**
- Recomputing them is wasteful

So models:

- Compute K/V once
- Cache them
- Reuse them for every future token

This cache is called **KV-cache**.

3.2 Why KV-Cache Dominates Inference Cost

Each new token:

1. Generates a Query
2. Reads **all past K/V**
3. Computes attention scores
4. Produces output

The cost grows **linearly with sequence length**.

This means:

- Long conversations get slower

- Agents degrade over time
- Latency spikes unexpectedly

3.3 Memory Bandwidth > Compute

Modern GPUs:

- Have enormous compute capacity
- But limited memory bandwidth

Attention is mostly:

- Memory reads
- Not math

So performance is capped by:

How fast can we move KV tensors?

This is why:

- FlashAttention exists
- GQA exists
- Context window extensions are hard

3.4 Why This Matters for Agentic AI

Agent loops:

- Accumulate long histories
- Re-read them constantly
- Inflate KV-cache

Without careful design:

- Agents slow down

- Costs explode
- Systems become unstable

This is why later we will study:

- Context pruning
- Memory compression
- External memory (vector DBs)

Mental Model You Must Keep

Attention quality is rarely the bottleneck.
Memory movement almost always is.

Once you internalize this, many GenAI design decisions suddenly make sense.