LLM Basics

01

What an LLM Is and What It Isn't

In order to build anything production-ready with Large Language Models (LLMs), you must first strip away the sci-fi marketing gloss and the “AI hype”. If you approach this technology expecting a "mind," a digital entity with intent or a capacity for conscious reasoning, you have already lost the engineering battle. There is no "subject" behind the screen. There is only a massive, high-dimensional statistical engine.

One of the most expensive mistakes an architect can make is treating an LLM as a sophisticated storage system. **An LLM is not a database.** It does not "retrieve" information. It does not "know" facts. When you feed text into a model, it doesn't file it away in a folder. Instead, it shatters the input into pieces (Tokens), maps them into high-dimensional Vectors, and runs them through a series of Matrix Transformations. You get a probabilistic guess, not a search result.

> At its core, the model’s entire existence is tethered to a single statistical task: next token prediction. It does not "know" things; it estimates the probability of text sequences. It calculates what should come next based on the provided context and the mathematical patterns it absorbed during its training phase.

By maintaining this mechanical focus, anthropomorphic intuition gives way to engineering precision. We move away from thinking about AI’s intentions and start asking how its architecture governs its output. In this chapter, we establish the fundamentals of LLMs that form a base of modern AI solutions: from a simple internal tool to a global agentic network.

Parameters: Knowledge of the Model

When a vendor pitches a "70B model," they are giving you a *headcount of floating-point numbers*. These billions of parameters (often called "weights") are the result of the training process. They represent the model's entire worldview, its logic, and its linguistic nuances.

Conceptually, you can think of parameters as trillions of microscopic "tunable knobs" that were set during training. For an architect, however, parameters represent something more pragmatic: the minimum VRAM required to keep the model going.

Training

An LLM starts its life as a mathematical skeleton: complete architecturally but void intellectually. In its raw state, every parameter is assigned a random value. If you prompt it, the result is "digital noise": a statistically meaningless soup of characters. It possesses the capacity for language, but no concept of its rules.

To transform this hollow shell into an asset, it must undergo the resource-intensive phase of pre-training. This is not a data-loading exercise; Instead, the model is exposed to a phenomenal volume of data: terabytes of text processed through millions of iterations. During each run, the model attempts to predict the next token and is immediately corrected when it fails, which forces the model to gradually adjust its parameters to minimise error. It is a process that costs a literal fortune in specialised hardware and electricity.

By the time training concludes, the model has effectively digested the statistical patterns, logic, and nuances of human communication into its trillions of parameters. This process is impartial: the model absorbs the rules of language *alongside* every bias, factual error, and structural flaw present in the training data. These traits become "baked in," defining the boundaries of how the model will behave and respond.

Deterministic Weights, Probabilistic Output

Crucially, the number of parameters is a fixed architectural constraint determined at the design stage. Once training concludes and the model moves to Inference, its parameters are frozen. The model does not "learn" from your queries in real-time, nor does it update its internal logic on the fly. While the *response* might vary, the underlying rules are a rigid, static snapshot of the moment training ended.

Why Parameter Count Matters

- **Capability:** Generally, more parameters allow a model to capture more complex relationships and deliver better answers for complicated questions
- **Performance:** While larger models are often smarter, they are also slower and more expensive to run (higher latency and compute costs)
- **Hardware Requirements:** Parameter count and precision can be used to roughly estimate model size. For example, a 7B model with Float32 parameters (4 bytes per parameter) takes 28 GB of VRAM. In the real world, the calculation is a bit more involved, but this simple math can be used as an initial estimation. When you load a model, you are pouring those billions of parameters into your GPU's memory. If you don't have enough VRAM to hold the parameters plus the overhead for context and execution, the model simply won't run.

 **The Architect's View:** staying within the physical limits of your hardware is as important as being mindful with the money. Your goal is to find the smallest model (lowest parameter count) that can reliably execute your specific task.

Tokens: the Building Blocks of Input and Output

When people say a model "predicts the next word," they're using a shorthand that is, frankly, wrong at an engineering level.

> The model doesn't see words: it processes a sequence of tokens. A token might be a whole word, a fragment or even just a single character.

Your input is broken down into tokens before being processed by the model and the output is generated one token at a time.

Why Tokens are the Key Influencer of the Solution

If you consume LLMs as a service, tokens are your currency. Almost every major vendor bills by the million tokens, meaning that inefficiency is quite literally a "money-burning" exercise. However, the

importance of token dynamics extends far beyond the billing department. Even for on-premise deployments, token inefficiency translates into wasted hardware cycles, crippling latency, and hitting the physical ceilings of the model's architecture.

In the Enterprise, tokens are more than just a billing line item – they are your primary architectural constraint. While functional requirements and compliance often dominate the vendor checklist, Token Dynamics determine whether your solution is a scalable asset or a financial black hole. To build a sustainable strategy, you must look past the marketing fluff and focus on these hard truths:

The Tokenisation Tax

Every model uses a proprietary "dictionary" (typically 50k to 256k tokens). If you feed the same sentence to GPT-5 and Gemini 3, you will get two different token counts.

♖ **The Architect's View:** A model with a poorly optimised tokeniser will fragment your words into more pieces. The result is a silent tax on every single request.

Domain and Linguistic Parity

Tokenisers are not created equal. A model optimised for conversational English might choke on Python code, legal-speak, or CJK languages.

♖ **The Architect's View:** If a model understands your specific "dialect" (e.g., raw server logs or medical terminology) in fewer tokens, it is inherently faster and cheaper. Choosing a model that "speaks your language" natively is a high-leverage architectural decision that pays dividends at scale.

The Headline Price Trap

Never compare "Price per 1M tokens" in a vacuum. A vendor might offer a 20% lower headline price, but if their tokeniser is 40% less efficient at processing your specific data, you are actually paying a premium.

♖ **The Architect's View:** You must benchmark against your production data. Scalability isn't about the cost of a million tokens; it's about how much "intelligence" you can squeeze out of every cent.

From Billing Nuisance to Strategic KPI

Understanding the above will give you the ability to set precise engineering constraints instead of vague performance goals. You can tell your team: "We need to stay under X tokens for this scenario," or tell your vendor: "Here is our real-world data; show us the token profile." Understanding these dynamics allows you to stop "hoping" for efficiency and start engineering it. In the Enterprise, token management is the pivot point where raw compute transforms into business value. If you don't manage the tokens, the tokens will manage your budget. And they won't be kind.

Prompt and Response

Any query you send to the LLM is called a Prompt. However, it is not the only thing the model receives for processing. The entire input the model receives is made up of several components:

- **User Query** – text the user types into the input box;
- **System Prompt** – the instructions for the model on how to behave, including tone and safety guardrails; this is a mandatory part of any prompt if you want a better quality response
- **Conversation history** – all prior exchanges in the current session; this is optional but usually present if a session or chat consists of multiple turns
- **Any additional data or files** added to the prompt (like documents or images), this is an optional but very useful part for complex prompts in most enterprise solutions

> One of the common mistakes is only looking at the user's query as input. In a real production environment, it is often the smallest part of the payload.

When you're calculating costs or latency you must factor in all other parts of the input to get a full picture. Let's look at what the "hidden" overhead of a single model call could be:

Suddenly, that "short" 50-word message is actually a 600-token payload and that does not even include any additional data that can be coming from RAG (Retrieval Augmented Generation, discussed in **Chapter 4**).

If you don't account for these overheads, your scaling projections will be off by orders of magnitude.

How to Estimate Token Load and Cost

If you're working with European languages in a business context, you can use these rough heuristics to avoid being blindsided by the bill:

- 1 token ≈ 3–4 characters (with the spaces and punctuation you probably ignored)
- 1 token ≈ 0.75 words (or, more realistically, 100 words will eat 130–150 tokens)

> The number of tokens does not match the number of either words or characters.

There is also a "Structural Tax" for formats like JSON or XML. All those brackets, quotes, and tags are extra tokens. If you're passing massive "raw" logs, you're paying a premium for syntax that adds zero semantic value.

Cost Estimation

Vendors bill by the million tokens (input and output). If you're inefficient with tokens, you're burning money. To calculate the baseline financial impact of a single AI transaction, use this simple formula:

$$\text{Cost} \approx (\text{Tokens}_{\text{in}} \times \text{Price}_{\text{in}}) + (\text{Tokens}_{\text{out}} \times \text{Price}_{\text{out}})$$

Where:

$\text{Tokens}_{\text{in}}$ = System Prompt + History + Any Additional Data + User Query

$\text{Tokens}_{\text{out}}$ = Generated response length

Workload Classification

Effective token management begins with an analytical classification of your workloads. In the Enterprise, treating all prompts as a monolithic stream is a recipe for architectural collapse. You must segment your traffic into three distinct operational profiles:

Short Interactive Bursts: These are high-frequency, low-latency interactions: support chat queries, search stubs, or single-turn instructions.

⚠ Individually cheap, but collectively they can eat your budget if your system prompts are bloated.

Medium-Weight Payloads: The current "bread and butter" of corporate AI, covering incident reports, ticket triage, and automated drafting.

⚠ These are prone to "scope creep." A medium payload can easily balloon into a heavyweight cost if you aren't strict about input validation.

Heavyweight Content: This is where your architecture is truly tested: legal audits, technical specification analysis, and full board transcripts.

⚠ These push the limits of your context window and, if left unmanaged, will be the primary drivers of your cloud spend and system timeouts.

Optimisation Strategies

Once these profiles are defined, you must move from passive observation to active Token Budgeting: the typical token range (including your hidden system prompt overhead), the maximum acceptable latency (response time goes up with the token volume), and the hard ceiling on cost-per-invocation. This also includes:

- **Enforce Hard Input Caps:** Reject 100-page PDF "copy-pastes" at the front end before they ever hit your API. You need a "Document Trigger" policy: if a file is too massive, it must be diverted to a pre-processing pipeline.
- **History Pruning:** You cannot maintain infinite context. You must decide: How many conversation turns are essential? When do you summarise the past to reclaim space? And when do you simply cut the cord to protect the current query's "room to think"?

- **The System Prompt Tax:** Never forget that your hidden system instructions – the guardrails and personas – consume resources in every turn. If your system prompt is 2k tokens, you're paying a massive "base tax" before the user even types "Hello."

Token management is not a "nice-to-have" optimisation. Without a rigorous budget policy, your LLM implementation will be at the mercy of unpredictable user behaviour and escalating costs. By enforcing these boundaries, you transform the "black box" of LLM costs into a predictable, scalable utility. You don't "hope" for efficiency in the Enterprise, you build it into your validation logic and orchestration itself.

Context Window

The Context Window is the absolute, unyielding limit of an LLM's "working memory." It is the total token budget you have for a single transaction. Everything must fit into this bucket: your system prompt, the conversation history, the user's current query, any background data you retrieve, and the space required for the model to generate its response.

Crucially, this is not a setting you can simply toggle in an admin panel. It is a physical constraint baked into the model's DNA during pre-training. While extending this window post-training is technically possible via engineering hacks, it's a high-risk intervention that rarely delivers the stability required for Enterprise production.

The Quadratic Tax: Why Bigger Isn't Always Better

In the world of modern LLMs, size comes with a heavy performance tax. One of the steps of the answer generation (attention mechanism, discussed in detail later) involves checking every token against every other token in the sequence. Therefore for the prompt of N tokens, the computational complexity at that stage grows as N*N.

Current models optimise that process to some extent, and that is an important to factor when discussing model and solution options. However, in general, if you double the number of tokens in your prompt, the computational cost for the attention mechanism doesn't just double; it quadruples in most cases. In a production environment, this leads to:

⏳ **Latency Spike:** As the window fills, the "Time to First Token" (TTFT) skyrockets. Your snappy AI assistant turns into a spinning hourglass because the GPUs are choking on maths.

🏛️ **Memory Tax:** storing long context requires large amount of VRAM. You might have the tokens to fit a 500-page book, but you may find you lack the GPU memory to actually host the model while it reads it.

How It Blows Up in Production

When you mismanage the context window, the system doesn't politely decline; it breaks catastrophically:

- **The Silent Lobotomy:** When the window overflows, models typically start dropping the oldest data to make room. Usually, the first thing on the chopping block is your System Prompt. Suddenly, the strict corporate guardrails and security instructions you spent weeks tuning simply vanish. Your enterprise bot is now flying blind, unmoored, and unpredictable.
- **The Generation Wall:** Remember, the output tokens also consume the window. If you fill 99% of your budget with a massive input prompt, the model has no room to "speak." It will simply hit the wall mid-sentence, returning truncated JSONs, broken code, and incomplete logic that crashes your entire pipeline.
- **The "Stuffing" Fallacy:** You cannot force-feed a massive technical spec to a model if it exceeds the window. If your use case requires 100k tokens and your model maxes out at 32k, your architecture is dead on arrival.

The Architectural Verdict

In the Enterprise, the goal is not to use the largest possible context window, but the most efficient one. Every token you add to the window is a trade-off between intelligence, latency, and cost. If your data doesn't fit, you don't need a bigger window, you need a better data strategy.

Inference: Generating the Response

Inference is the process that takes the input data through a series of steps to produce an output. It is the bridge between a trained model and a business result. While pre-training is a one-time (albeit massive) capital expenditure, Inference is your ongoing operational reality.

If the model's size defines its "IQ," and the Tokeniser defines its alphabet, then Inference is the engine running the show. In a production environment the world is ruled by three unforgiving deities: Compute Cost, Memory Bandwidth, and Latency.

For an Architect, Inference is about runtime lifecycle of a request, not a simple "answer generation". And talking about Inference requires understanding Transformers.

Transformer

The word "Transformer" is thrown around like a magic spell, but for a Tech Lead, it represents a specific architectural commitment. You need to distinguish between three levels of meaning:

The Design Paradigm

It was introduced in the paper "Attention Is All You Need" by Vaswani et al. Unlike older models that processed text like a slow-moving conveyor belt, the Transformer architecture (the "Attention" mechanism) allows the model to "see" the entire input at once (parallelism over sequence). It is the basis of the majority of modern LLMs.

♖ **Architectural impact:** This is why LLMs can scale. It's what allowed us to throw massive GPU clusters at the problem. Without this parallelism, modern AI wouldn't be feasible at scale.

The Functional Block

When engineers talk about "Transformer layers," they are talking about one self-contained processing module inside the LLM. This is where most of the Parameters (the model's "knowledge") reside.

♖ **Architectural impact:** Every additional layer increases the computational "depth" and, consequently, the time it takes to generate a single token.

The Architectural Variant

Not all Transformers are built the same. There are 3 variants to choose from (Encoder-only, Decoder-only, Encoder-Decoder) and we will look at them in detail later on.

♖ **Architectural impact:** Solutions for some tasks are more optimal if designed with a less popular LLM variant.

Why you should care

If someone says, "We're using a Transformer-based solution," they haven't told you anything useful. It's like saying, "We use a motor." Great, are we building a lawnmower or a jet engine?

As an architect, you must draw a hard line between the two worlds:

The Model Interior: The computational engine. This is the Transformer's domain, containing all the input processing (Inference) modules.

The Production System: The "wrapper" that actually makes it enterprise-ready. This includes guardrails, routing logic, caching strategies, and human-in-the-loop validation.

Don't get distracted by the "magic" of the Transformer. Your job is to manage the friction between the model's massive hunger for resources and your organisation's need for predictable, cost-effective performance.

Inference

02

Why Inference Matters to You as an Architect

In Chapter 1, we established the "what": an LLM is a frozen statistical engine designed for next-token prediction. But knowing the anatomy of an engine is useless if you don't know how it behaves at 7,000 RPM.

> Inference is the complete lifecycle of a request, from the millisecond a query hits your API to the moment the final token is rendered on a user's screen.

In the lab, it's maths; in production, it's a high-stakes balancing act between latency, memory bandwidth, and cold, hard cash. Treating Inference just as a "black box" is not good enough. It isn't a single operation, it's a pipeline of distinct, expensive stages. Each stage has its own "tax": its own memory footprint, its own computational bottlenecks, and its own unique ways to fail.

For an architect or a tech lead, understanding this pipeline is required to:

- Predict performance under load before the system chokes.
- Prevent budget meltdowns caused by inefficient orchestration.
- Select the right infrastructure instead of over-provisioning out of fear.

This chapter deconstructs the Inference pipeline step by step. We aren't here to turn you into an ML researcher – we are here to give you the engineering vocabulary to make decisions that hold up under fire.

The Two Stages of Inference

Most people imagine that a model receives a prompt and simply "returns" an answer – as if it were a database query. The reality is fundamentally different and has direct consequences for your cost and latency projections.

An LLM generates its response one token at a time and uses the previous token in the generation of the next token. This is called autoregressive generation, and it iss the reason inference splits into two distinct stages with very different performance profiles.

Stage I: Prefill

This is the initial, one-shot pass where the model "digests" your entire input. The system processes all your tokens simultaneously, mapping out the relationships between them to determine the very first token of the output. This computation is only done once.

- ∅ **Bottleneck:** Prefill is computationally intense but highly parallel. Modern GPUs love this: they can chew through thousands of tokens at once.
- ⌛ **Latency:** This is where *Time to First Token* (TTFT) is born. If your Prefill takes 5 seconds because you fed it a 50-page legal doc, your user is staring at a blank screen for those 5 seconds. Nothing moves until Prefill is done.

Stage II: Decode

Once the first output token is born, the model enters the Decode phase. This is the generation of every subsequent token, one... by... one. Each new token depends on the one before it. Once generated, it gets appended to your input and takes its place as part of the context.

- ∅ **Bottleneck:** Decode is inherently sequential. You cannot parallelise it. Your powerful $40,000 GPU is forced to wait for the previous step to finish before it can even think about the next one. This is why LLM output feels like a "stream" rather than a sudden burst.
- ⌛ **Latency:** Decode scales linearly with your output length. If you want a 500-word essay, you are signing up for a fixed amount of sequential "grind" that no amount of extra hardware can magically bypass.

What this means for your implementation

Understanding the distinction between these two stages gives you the toolkit to manage performance before problems appear in production — not after. Every architectural decision that touches input size or output length has a direct, measurable consequence on cost and user experience. The two dimensions to watch are latency and cost.

Latency

Your total response time is the sum of two parts: the time to process the input (Prefill) and the time to generate the output (Decode). As your input grows (long system prompts, massive documents, long conversation history), so does TTFT. Only when Prefill completes does the first token of the response appear. From that point, tokens arrive in a steady stream as Decode progresses.

The Economic Asymmetry: Why Output is 3x More Expensive

You've probably noticed that API vendors charge significantly more for output tokens. This is a reflection of the computational reality associated with the decode. Every output token requires a full "forward pass" through the entire model, but because it's sequential, it leaves most of your GPU's parallel processing power idling. You're paying for a Ferrari to drive in a 20mph zone. Moreover, to stay "context-aware" during Decode, the model has to store a massive amount of intermediate data in its high-speed memory. This memory is the most expensive real estate in your data center.

The Architectural Impact

If you are designing SLAs or projecting infrastructure costs, model the Prefill/Decode split for your actual use cases, not just the total token volume. A 1,000-token request isn't a single unit of work:

Optimise your Prefill to save your UX (Latency). Optimise your Decode to save your budget (Cost). If you fail to distinguish between the two, you will end up with a system that is either too slow to use or too expensive to run.

Here is the breakdown of what actually drives your metrics:

	WHAT DRIVES COST AND TIME	YOUR MANAGEMENT LEVERS
Prefill	*Input size:* system prompts, conversation history, uploaded documents, and the user's query	• Set hard limits on input length • Compact conversation history • Control the size of uploaded documents
Decode	*Output volume:* the number of tokens generated	• Set max_new_tokens for every use case • Instruct the model toward concise or structured output formats

The Inference pipeline

The Inference pipeline is the full sequence of operations that make up Prefill and Decode and transform a raw string of text into a probabilistic response. Every stage in this pipeline is a toll booth where you pay in VRAM, compute cycles, and milliseconds. All of the model's parameters live inside the components this section will walk through.

We deconstruct this pipeline to identify the bottlenecks. If you miscalculate the requirements at any single stage, the system could slow down or worse – fail. You'll face latency spikes that kill UX, VRAM exhaustion that crashes your instances, and cloud bills that defy rational explanation.

Understanding the pipeline is what separates a deliberate, scalable architecture from an expensive experiment.

Tokeniser

Before a single byte of your input reaches the neural network, it passes through the Tokeniser. Its job comes down to two operations:

✂ **SLICING:** the raw text string is cut into chunks according to a fixed set of rules

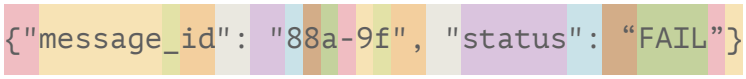
\# **MAPPING:** each chunk is matched to a Token ID – a unique numerical identifier in the model's vocabulary

Most modern models use subword tokenisation, with Byte Pair Encoding (BPE) being the most common method. It is a pragmatic middle ground: common words get their own unique ID, while rare words, specialised jargon, or names are split into multiple fragments. The same input will produce different token counts on different models – because each model has its own vocabulary and its own tokenisation rules.

To make this concrete, let's look at the couple of examples. Run through the Google Gemma tokeniser, normal text returns 9 tokens per 5 words:


```
[11792, 861, 139307, 819, 4401, 6803, 8447, 5076, 235341]
```

However a simple JSON string of a similar length run through the same tokeniser results in a very different picture:




```
[9766, 1529, 235298, 539, 1192, 664, 235321, 235321, 235250, 235290, 235315,
 235266, 824, 664, 3325, 1192, 1080, 22198, 235369, 235270]
```

This came up to 22 tokens (more than *2x* compared to the normal text) because the vocabulary penalises symbols, brackets, and alphanumeric hex-codes.

The Architectural Impact

- **Hidden Cost:** You aren't billed by the word; you are billed by the token. If your business domain heavily relies on strict JSON schemas, XML wrappers, or proprietary code structures, the tokeniser will heavily fragment your input. Your operational costs will be 30–50% higher than your basic "character count" estimates suggest.
- **Context Drain:** Every single bracket and fragment consumes a precious slot in your finite context window. A poorly chosen model with a "token-hungry" vocabulary effectively shrinks your available memory, forcing you into expensive context-handling workarounds much sooner than expected.

Embedding Layer

A Token ID by itself is a dumb serial number. `ID#402` carries zero semantic weight; the model cannot perform calculus on a catalogue index. This is where the Embedding Layer comes in: it translates discrete IDs into continuous, high-dimensional vectors called Input Embeddings (massive arrays of floating-point numbers) that the model can perform calculations on. It is essentially a massive lookup table – a matrix of parameters defined by `[Vocabulary Size] x [Vector Dimension]`. For each Token ID, the model retrieves the corresponding row: a specific fixed-length vector of numbers shaped during training to capture the token's meaning and relationships. From this point forward, the model is blind to "words" or "IDs". It operates exclusively on these vectors.

The Architectural Impact

Permanent VRAM Tax: This massive matrix is loaded into your GPU memory at startup and never leaves. If a model has a vocabulary of 128,000 tokens and a vector dimension of 8,192, this single layer consumes gigabytes of your most expensive hardware real estate before you even send the first prompt.

Memory Bandwidth Choke: The Embedding Layer isn't a one-and-done setup operation. Your entire input is converted to the embeddings once at the time of Prefill, but during the Decode phase every single output token must be also be converted through these representations. It is a high-frequency read operation that hammers your GPU's memory bandwidth on every step.

"Resolution" Trade-off: The Vector Dimension represents the model's semantic resolution. Higher dimensions capture richer linguistic nuance and deeper reasoning, but they exponentially inflate the computational weight of every subsequent step in the pipeline. You are trading hardware cycles for intellectual depth.

Transformer Block

The Transformer Block is where the model stops shuffling data and starts simulating intelligence. Everything prior was just prep work, but this is the core computational engine where isolated vectors are shaped into context-aware representations. The work that actually defines the quality of the model's response happens here. Everything after this point is output generation.

In this section, we will cover the two most important components of the Transformer Block.

Attention

The beating heart of the Transformer Block is the Attention mechanism. This is where the «magic» (and the massive hardware cost) resides.

> Forget the philosophical debates about whether models "think." Attention is purely mechanical: it is an algorithm that forces every token to look at every other token in the sequence and calculate a mathematical "influence weight." It stops treating tokens like isolated islands and starts mapping the relationships between them.

This doesn't make the model "primitive." Far from it. The sheer scale, billions of parameters trained on petabytes of data, gives rise to emergent behaviour that looks like reasoning, stylistic flair, and vast knowledge. But make no mistake: this isn't a human-like "act of thought." It's a massive, statistically-driven computational process that happens to be exceptionally good at mimicking the outputs of human reason.

During the Prefill stage, if you feed the model 10,000 tokens, it is not performing 10,000 sequential calculations. It is mapping a 10,000 × 10,000 grid. This is the exact moment where your computational cost scales quadratically (N^2). Once this massive calculation is done, the raw vectors are permanently saturated with the context of your entire prompt. The word "bank" now knows if it's sitting next to "river" or "robbery."

If the model had to recalculate the entire N^2 grid for every single new token it generates during Decode, inference would be impossible – it would take hours to write a paragraph.

To solve this, the model stores the results of the Prefill calculations in a high-speed memory ledger: the KV Cache (discussed in detail in **Chapter 3**). When a new token is generated, the model doesn't need to recalculate everything. It simply checks the new token against the pre-computed history stored in the KV Cache. This makes Decode feasible to finish in reasonable time.

The KV Cache is not a static file. Every single token generated during Decode adds to its size, and it must be held entirely in the GPU's expensive VRAM for the duration of the request.

The Architectural Impact

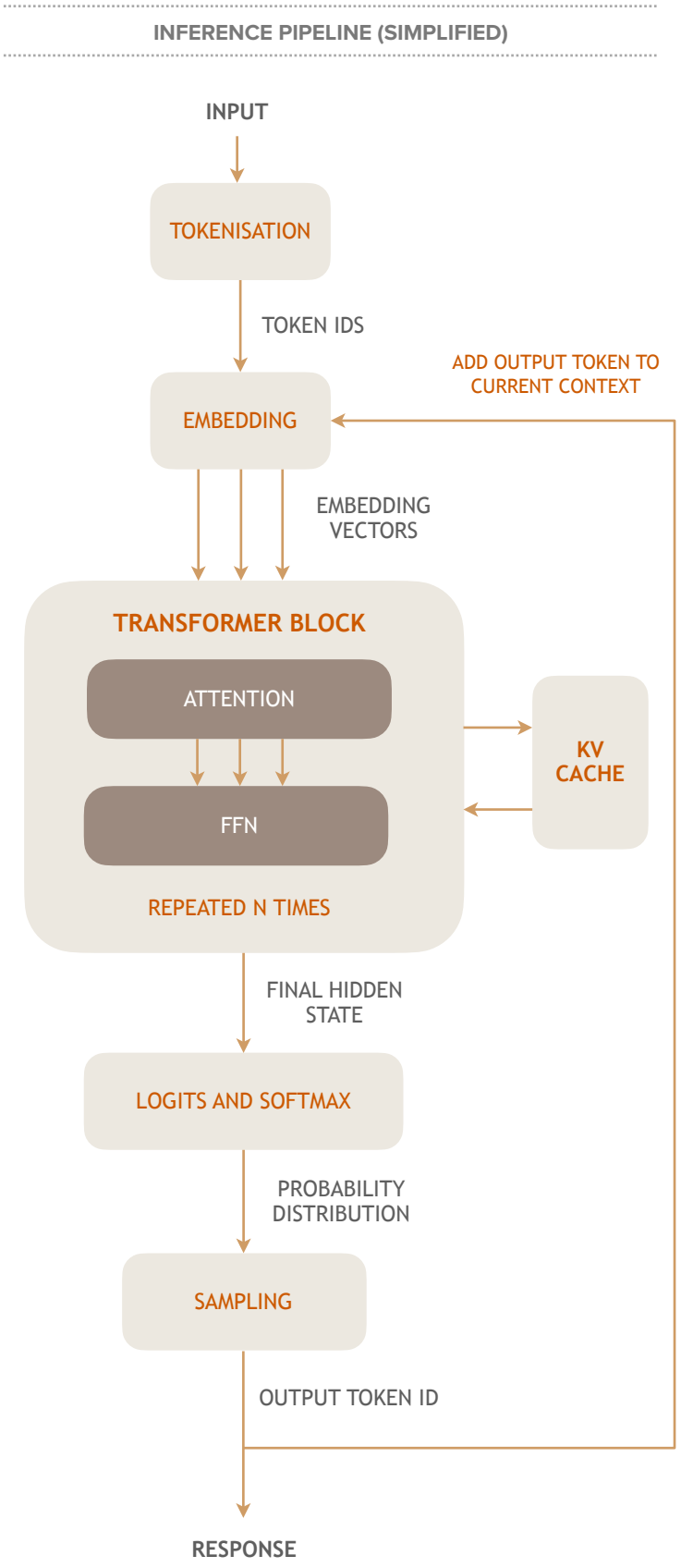
For extended enterprise workflows (like processing large legal documents or multi-turn agentic loops) the KV Cache will often exceed the memory occupied by the model's actual parameters. You are paying to host not only a model, but also its infinitely expanding memory ledger. If you do not explicitly budget for KV Cache capacity, your GPU instances will crash with Out-Of-Memory (OOM) errors under peak load.

FFN

If Attention is the mechanism that figures out how tokens relate to each other, the Feed-Forward Network (FFN) is the part that actually does the heavy lifting. Attention figures out the context; the FFN applies the "knowledge."

Once Attention has mapped the relationships, the altered token is passed into the FFN. Here, there are no cross-token comparisons. The FFN processes each token individually, applying deep mathematical transformations based on the model's training.

This is where the sheer mass of the



model lives. The FFN typically consumes 60–70% of the total parameter count.

Depth vs. The Latency Multiplier

The cycle of [Attention + FFN] constitutes one Transformer Block. Models stack these blocks across dozens of layers. Each pass through the layer creates a Hidden state for each token, refines the representations further, and finally produces the Final Hidden State for the output token selection at the last layer.

Myth: More layers automatically equal a "smarter" model.

Reality: More layers give you a multiplier for latency and compute cost. There is a rigid threshold where adding more depth yields diminishing returns for reasoning but guarantees escalating infrastructure costs.

The Economic Crossover: Linear vs. Quadratic

For an architect, the FFN is fascinating because its computational cost scales linearly with the input size, unlike Attention, which scales quadratically (N^2). This creates a critical "crossover point" in your production economics:

Short Prompt Dominance: At short prompt lengths (e.g., quick chat queries), the FFN actually dominates your compute budget. Since it contains 70% of the weights, dragging a few tokens through the massive FFN is the heaviest part of the process.

Long Prompt Trap: As the input grows (typically past the 5k–6k token mark, the exact crossover depends on the model architecture), the maths suddenly flips. Attention's N^2 complexity accelerates and overtakes the FFN. For massive documents or extended conversation histories, the FFN becomes a background operational cost, while Attention at Prefill becomes the primary driver of latency, sizing, and hardware exhaustion.

The Decode Sledgehammer

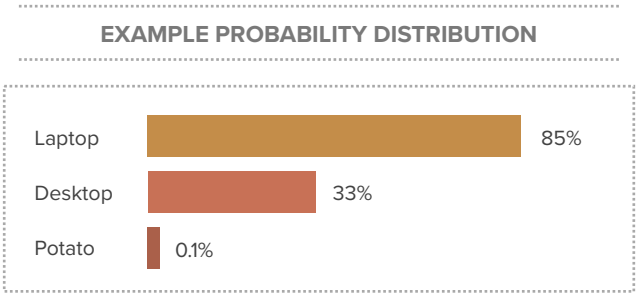
Never forget the reality of the Decode phase: This entire Transformer sequence — every layer, every Attention mechanism, every massive FFN — runs for every single output token generated.

Every word your system produces carries the full cost of a complete forward pass through billions of parameters. If you don't constrain the length of your outputs, the FFN will drain your budget token by token. That's why the final steps of the pipeline are just as important.

Final Steps

After passing through all Transformer Block layers, the Final Hidden State of the last layer reaches the exit point of the pipeline. Here the model performs the last sequence of operations that converts its internal representation into an actual token – the next word of your response.

Before the model selects a word, it generates a raw confidence score (a Logit) for every single token in its entire vocabulary. Through a function called Softmax, these raw scores are squashed into a clean probability distribution (ranging from 0% to 100%). If the model simply picked the 85% probability every single time, it would be a boring deterministic robot. But it doesn't have to.



The Decoding Policy: Picking the Next Token

Once the model generates the probability distribution across its vocabulary, it still hasn't chosen a word. It needs a rule for how to choose. This is your Decoding Policy – a set of runtime API parameters that determine whether your system behaves like a deterministic engineer or a hallucinating poet. This is not a fixed part of the model's weights; this is a runtime configuration that you control via API parameters (like Temperature, Top-K, and Top-P). While the computational cost of this final step is almost zero, its architectural impact is massive.

The Two Levers That Matter the Most

ML literature discusses many sampling methods, but in a production API environment, you are primarily managing two dials:

Temperature (The Hallucination Dial)

If you set the policy to prioritise "creativity" (high Temperature), you are explicitly instructing the model to occasionally pick the 10% or even the 1% probability. In an Enterprise environment, this is how you engineer a hallucination.

If you force the model to always pick the highest probability (Temperature = 0), you get maximum predictability. This is mandatory for generating JSONs, SQL queries or routing logic, but it strips the model of its linguistic flexibility.

Top-P / Nucleus Sampling (The Sanity Guardrail)

Instead of looking at the whole dictionary, Top-P dynamically restricts the model's choices to a subset of tokens whose combined probability reaches a specific threshold (e.g. 0.9).

Top-P acts as a dynamic filter. When the model is highly confident, the pool of choices stays small. When it is uncertain, the pool expands. It allows for controlled variety without letting the model select a completely absurd token that derails the entire response.

The Architectural Impact

The final stage of Inference not only impacts output tokens, but also defines the nature of the output, making it more predictable and repeatable.

> In the Enterprise world, "creative" is usually a synonym for "broken."

For analytical, classification, or operational tasks keep your temperature low and use conservative Top-P settings if you want outputs that are stable, reproducible and auditable. Save the high temperature and wide Top-P settings for use cases where variety is genuinely valuable, such as marketing copy generation or brainstorming tools.

You do not deploy a model without a properly defined Decoding Policy. If you leave these settings on "default," you are surrendering control of your application's reliability to the vendor's whims.

The Circuit Breakers: Defining Stopping Criteria

The Decode loop is not a perpetual motion machine, but if left unchecked, it will happily act like one – spinning out tokens until your context window overflows and your API budget is exhausted. In a production environment, an LLM without strict boundaries is a financial liability.

The model only stops generating when it hits a specific "Circuit Breaker." You have three, and you must architect all of them deliberately:

Natural Limit (EOS Token)

During pre-training, the model learns a special vocabulary token: the End-of-Sequence (EOS) token. When the model reaches a logical conclusion based on its training, the probability of the EOS token peaks, it gets selected, and generation halts.

♖ **Architect's view:** This is the model policing itself. It is polite, but it is not a security strategy. You cannot rely solely on the model deciding it is "finished," especially when handling complex, open-ended tasks where the model might hallucinate and just keep talking.

Stop Sequences

A Stop Sequence is a specific string of characters (like `\n\n`, `}`, or `User:`) that you feed into the inference API. The second the model generates this string, the infrastructure severs the connection and stops the Decode phase.

♖ **Architect's view:** This is your primary tool for forcing structure. If you are generating JSON, your stop sequence is the closing bracket. If you are building a chatbot, your stop sequence is the trigger for the next speaker. It is how you prevent a model from "playing both sides" of a conversation or appending unsolicited essays to a precise answer.

Max New Tokens

This is a hard, infrastructure-level ceiling on the number of tokens the model is allowed to generate per request. If the limit is 500, generation terminates at token 500, even if it happens mid-sentence or breaks the output logic.

♖ **Architect's view:** Never leave Max New Tokens as a default setting. Every output token costs compute cycles and API credits. If a user asks a simple question and the model glitches into a repetitive loop, a default limit of 4,096 tokens means you just paid for 4,000 tokens of rubbish. You must define a strict mathematically justified cap for every specific use case.

The Bottom Line: An Enterprise architecture does not "hope" the model gives a concise answer, but engineers the boundaries so that runaway generation is physically impossible.

Conclusion and Architect's Audit

Understanding the mechanics of Inference doesn't make you an ML engineer, and that was never the goal. It makes you a nonsense detector in an industry flooded with hype. You now possess the vocabulary to cut through the marketing noise, pressure-test vendor claims, and expose the hidden architectural assumptions that will exhaust your cloud budget six months post-launch.

Do not walk into your next infrastructure review or vendor pitch to simply "listen." Walk in with these questions, and do not accept hand-waving as an answer:

- **Latency vs. Cost Split:** "What is the exact ratio of Prefill to Decode for our primary use case? Are we optimising our hardware for processing massive inputs or generating long outputs?"
- **KV Cache Size:** "At our expected peak concurrency and maximum context length, what is the exact memory footprint of the KV Cache? Does it fit in our VRAM alongside the model weights, or will our instances crash with Out-Of-Memory errors under load?"
- **Tokenisation Tax:** "Show me the token efficiency of your model on our proprietary data (code, logs, legal docs), not a public benchmark. How much of a 'fragmentation premium' are we paying just to parse our own domain jargon?"
- **Financial Kill-Switches:** "What is our hard 'Max New Tokens' limit for this specific pipeline? What are our Stop Sequences? If the model gets stuck in a hallucinatory loop, who in this room owns the financial responsibility?"
- **Breaking Point:** "If we scale to 10x our current request volume tomorrow, which specific component of this Inference pipeline chokes first? Is it compute limits, memory bandwidth, or API rate limits? And what is the exact cost of remediation?"

You are no longer interacting with a "magic black box." You are managing a heavy, expensive, statistical engine. Treat it with the architectural paranoia it deserves.

Architect's Reference

The table below maps the key components of the Inference pipeline to their practical infrastructure impact and the decisions they drive. Its purpose is to give you a single reference point: the

vocabulary to ask the right questions, the context to interpret the answers, and the judgment to know when a vendor or engineer is glossing over something that matters.

	FUNCTION	PRACTICAL IMPACT	ACTIONABLE ADVICE
I. STATIC ASSETS (VRAM Baseline)			
Model Parameters & Layers	Total weights (Attention, FFN, Embeddings) and the depth (Layers) of the network.	Defines the hard VRAM requirement. Larger models or more layers improve output quality, but increase latency and memory consumption.	**Quantize or resize.** Use 4-bit/ 8-bit versions to fit larger models into limited VRAM.
Embedding Layer	Converts Token IDs into numeric vectors for computation.	A "resident" tax. Stays in VRAM for the entire session regardless of usage.	**Check Vocabulary size.** Avoid models with bloated embedding matrices if your domain jargon is limited.
II. DYNAMIC PIPELINE (Performance & Costs)			
Tokeniser	Parses raw text into IDs; converts output IDs back to text.	Efficiency varies. Poor tokenisation on domain-specific data (code, legal) increases token counts and costs.	**Test on proprietary data.** Don't rely on public benchmarks; measure "fragmentation premium" for your specific jargon.
Attention & KV Cache	Maps token relationships and stores intermediate data to avoid re-computation.	Dominant memory consumer at long contexts. Can exceed the model weights themselves. Memory usage scales with context, computation scales non-linearly.	**Enforce hard token limits.** Use KV Cache quantisation for high-concurrency deployments.
Feed-Forward Network (FFN)	Processes each token's representation independently after Attention.	The heaviest per-token computation consumer. Drives raw latency (Time Per Output Token).	**Prioritize Memory Bandwidth.** For low-latency needs, select hardware with high bandwidth alongside raw FLOPs.
III. OPERATIONAL CONTROL (Safety & Quality)			
Context Window	The hard limit of tokens (input + output) processed in one call.	Overflow causes "amnesia" or generation failure. Unnecessarily large windows increase cost without benefit.	**Right-size your prompts.** Use RAG or chunking instead of hoping for an "infinite" window.

	FUNCTION	PRACTICAL IMPACT	ACTIONABLE ADVICE
Decoding Policy	Mathematical rules (T, Top-p) for selecting the next token.	Controls the balance between creativity/hallucination and consistency/logic.	**Match settings to task.** Use low Temp/Top-p for analytical tasks; high for creative content.
Stopping Criteria	Limits that terminate the loop (EOS, Stop Sequences, Max New Tokens).	Without these, the model can enter "runaway loops," causing catastrophic costs and latency.	**Set Financial Kill-Switches.** Explicitly define Max New Tokens and Stop Sequences for every pipeline.

Complete book is coming out end of June 2026 and will be available to download at Igor's website: **www.notesofea.com/book**

Follow Igor Ageyev on **LinkedIn** for updates and announcements.