CHAPTER TWO

Core Foundations of Generative AI

SECTIONS 2.1 TO 2.12

Four quantities decide what is architecturally possible: the token, the context window, the quadratic cost of attention, and the geometry of embedding space. This chapter is the physics.

CHAPTER TWO

Contents

Four quantities determine what is architecturally possible: the token, the context window, the quadratic cost of attention, and the geometry of embedding space. Every decision in the remaining fifteen chapters is downstream of one of them. This chapter is the physics; the rest of the book is engineering that respects it.

2.1 Understanding Large Language Models

Start from the smallest correct statement. **A language model is a function that maps a sequence of tokens to a probability distribution over the next token.** That is the whole thing. Chat, agents, reasoning, tool use and retrieval are constructions built on repeated application of that one function by code you write.

It is worth knowing, at one level of detail, how that function is made, because three training stages produce three behaviours you will observe and design around. **Pretraining** adjusts the weights to predict the next token across a very large general corpus. The result is a base model: a formidable text continuer with no notion of being asked for anything. Hand it a question and it may answer, extend the question, or produce three similar questions, because each is a plausible continuation of the text it has seen.

Instruction tuning continues training on a much smaller corpus of instruction and response pairs, each wrapped in a chat template: the role markers and delimiters that turn a flat token sequence into something shaped like a conversation. This is what "instruct-tuned" means on a model card, and it is why section 4.7 insists on the distinction: a base model behind a chat endpoint continues text rather than following instructions. The same stage teaches the tool call format, which is why a model can be trained to produce tool-call-shaped text, and why that text is still only text.

Preference tuning then shifts the model towards outputs that human or model raters prefer. It is why assistants are agreeable, hedge politely, and open answers with preamble. Two consequences are architectural: the preamble is a latency cost that section 2.11 bounds with an output contract, and the agreeableness biases the model towards answering when it should abstain, which is part of why the abstain path in section 6.9 must be engineered rather than expected.

None of the three stages adds state, execution or ground truth. They shape the distribution the function returns, and the three consequences below survive all of them.

Three consequences follow immediately, and a surprising amount of confused system design comes from not having internalised them.

The model has no state

A conversation exists because *you* resend the entire history on every turn. There is no session on the provider's side holding your context. The model does not remember the previous message; it re-reads it.

This is why Chapter 11 exists at all. Memory is something you build, store, retrieve and pay for on every turn, not a property the model has. It is also why conversation length is a cost curve rather than a constant: the tenth turn of a conversation costs more than the first, because the first nine are in the prompt.

The model has no execution

When an agent "calls a tool", what actually happens is that the model emits text which *looks like* a function call, in a format it was trained to produce. Something in your code parses that text and decides whether to actually invoke the function.

This is good news and it is the entire basis of the permission model in Chapter 12: **the enforcement point is on your side of the boundary.** A model that has been fully persuaded to attempt something harmful still cannot execute anything. It can only produce a string that your code may or may not act on. Section 1.5 already used this reasoning to keep credentials away from tools; Chapter 15 generalises it.

The model has no ground truth

It produces the most probable continuation given its training distribution and your context. Where your context contains the answer, it is likely to use it. Where it does not, the model still produces a fluent continuation, and *fluency is not correlated with correctness*. It is arguably anti-correlated, because a confident, well structured wrong answer is harder to detect than a hesitant one.

This is why retrieval exists, and why the abstain path in section 6.9 is a feature rather than a failure.

The framing that produces good architectures. Do not think of the model as a knowledge base you query. Think of it as an extremely capable text transformer with a fixed size input window. Your job as an architect is to decide *what goes in that window* and *what is permitted to happen to what comes out*. Nearly every chapter of this book is one of those two problems.

2.2 How LLMs Work Under the Hood

Generation is a loop. Tokenize the prompt, run a forward pass, obtain a distribution over the vocabulary, sample one token, append it, repeat. The loop terminates on a stop token, a length limit, or a stop sequence you supplied.

```
def generate(model, tokenizer, prompt: str, max_tokens: int = 200,
             temperature: float = 0.7, top_p: float = 0.9) -> str:
    ids = tokenizer.encode(prompt)
    cache = None                                  # the KV cache, section 2.11

    for _ in range(max_tokens):
        # First iteration processes the whole prompt (PREFILL).
        # Later iterations process ONE token, reusing the cache (DECODE).
        logits, cache = model.forward(ids[-1:] if cache else ids, cache=cache)
        logits = logits[-1]                       # only the last position

        if temperature == 0:
            next_id = int(logits.argmax())
        else:
            probs = softmax(logits / temperature)
            next_id = nucleus_sample(probs, top_p)

        if next_id == tokenizer.eos_id:
            break
        ids.append(next_id)

    return tokenizer.decode(ids)
```

Two architectural facts fall out of that loop, and both are load bearing later.

Cost is asymmetric between input and output. The prompt is processed in one parallel pass. Each output token requires a full forward pass through every layer of the network. This is why every provider prices output tokens several times higher than input tokens, and why section 2.11 argues that output length is the dominant latency lever in interactive products.

Temperature zero is not determinism. Setting temperature to zero removes sampling randomness, which is one of three sources of variation. The other two are floating point non-associativity under varying batch composition on a GPU (your request is batched with different neighbours each time, and the reduction order changes the last bits), and the provider silently changing the model behind your alias. A system that genuinely requires reproducibility needs a cache or pinned self hosted weights, not a temperature setting. This is property five from section 1.6, appearing in its most concrete form.

The sampling parameters, and which ones matter architecturally

Parameter	What it does	Architectural relevance
`temperature`	Flattens or sharpens the distribution before sampling.	Set 0 for extraction, classification and routing. Anything above 0 on a structured output task is buying variance you did not want.
`top_p`	Samples from the smallest set of tokens whose cumulative probability exceeds p.	Prefer over `top_k`, because it adapts to how confident the model is at that position.
`max_tokens`	Hard ceiling on output length.	**The most under-used cost and latency control in the field.** It bounds the expensive half of the request and it is one integer.
`stop`	Sequences that terminate generation.	Cheap insurance against a model that continues past its answer into an invented next turn.

2.3 Deep Dive into the GPT Architecture

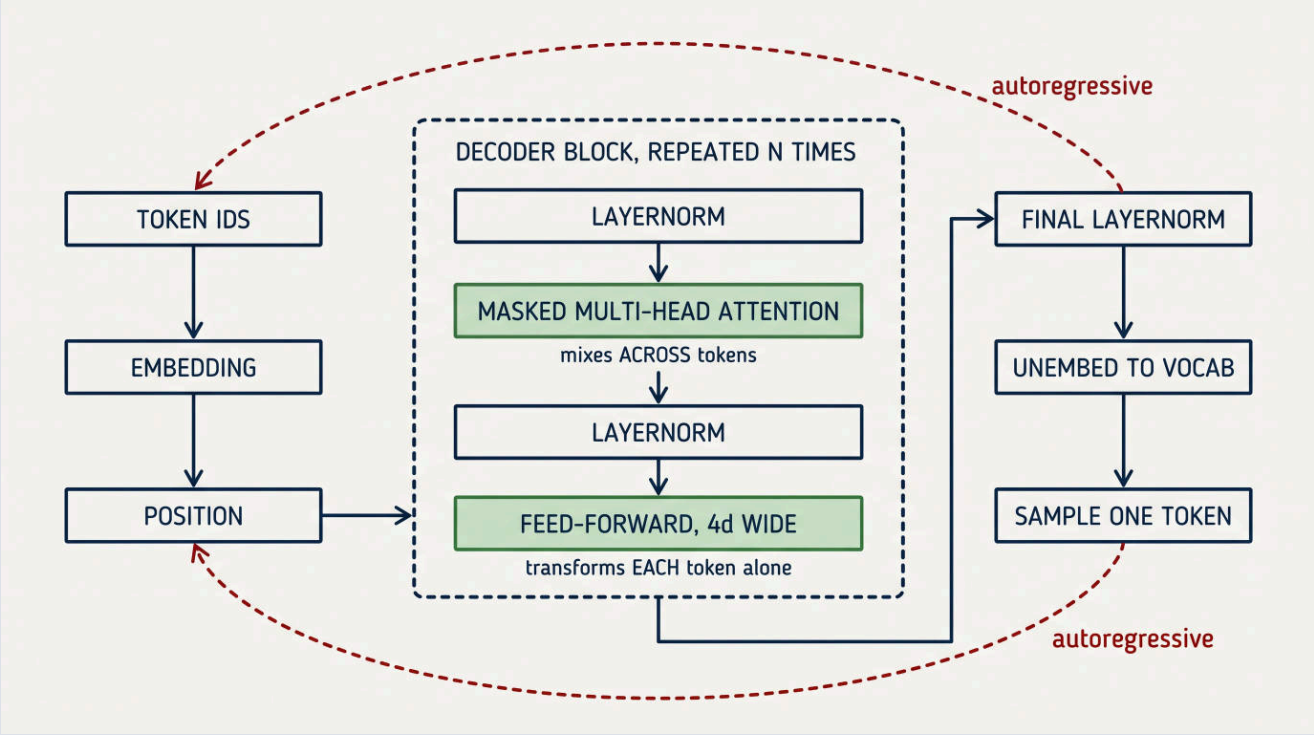

Figure 2.1 The entire architecture. Two operations alternating: mix information across tokens, then transform each token independently.

A decoder only transformer is a stack of identical blocks. Each block does exactly two things, and the distinction between them explains most of the performance characteristics in this chapter.

Attention mixes information across token positions. This is the only place in the entire network where tokens see each other. Everything you know about "the model understood the relationship between clause 4.2 and clause 7" happens here.

The feed-forward layer transforms each position independently. No cross-token communication at all. It projects each token's vector up to roughly four times the hidden dimension, applies a non-linearity, and projects back down. It is where most of the model's learned knowledge is stored.

Around both sits a residual connection and a normalisation, which are what make deep stacks trainable at all.

Where the parameters actually are

The distribution surprises people, and it has direct consequences for quantisation.

```
def parameter_budget(layers: int, d_model: int, vocab: int,
                     ffn_multiplier: int = 4) -> dict:
    """Useful for sanity-checking a model card, and for understanding where
    quantisation actually pays off."""
    attn_per_layer = 4 * d_model * d_model            # q, k, v, out projections
    ffn_per_layer = 2 * ffn_multiplier * d_model * d_model   # up and down
    return {
        "attention": layers * attn_per_layer,
        "feed_forward": layers * ffn_per_layer,
        "embeddings": 2 * vocab * d_model,            # in and out
    }

b = parameter_budget(layers=32, d_model=4096, vocab=128_000)
# attention     2.15B   (29%)
# feed_forward  4.29B   (57%)
# embeddings    1.05B   (14%)
```

Roughly two thirds of the parameters are in the feed-forward layers, which receive a fraction of the discussion that attention receives. The architectural relevance is direct: quantisation targets the weights, the weights are mostly feed-forward, which is why four bit quantisation shrinks a model so effectively. Weight quantisation does nothing to the KV cache, which is an attention artefact; quantising the cache itself, typically to eight bit floating point or integer, is a separate lever that serving stacks now offer as standard. Section 2.11 shows why the asymmetry between weights and cache dominates capacity planning either way.

One architectural variant changes how to read this arithmetic. A **mixture-of-experts** model replaces each feed-forward layer with several parallel expert copies and a small router that sends each token through only one or two of them. The

model card then quotes two parameter counts: *total*, summing every expert, and *active*, the subset a single token actually passes through. The strongest open weight models at the time of writing are built this way, with totals in the hundreds of billions and active counts a tenth of that.

The two numbers govern different resources. **VRAM follows total parameters**, because every expert must be resident: the router decides per token, so no expert can be paged out. **Per-token compute follows active parameters**, which is why a mixture model can match a much larger dense model's quality at a fraction of the latency and energy per token.

The serving arithmetic later in this chapter is written for dense models, where the two counts coincide. To adapt it, use total parameters for weight memory and active parameters when reasoning about throughput, and note that the KV cache term is unchanged, because attention is not where the experts live. Chapter 4 returns to this when sizing hardware for open weights.

2.4 Fundamentals of Tokenization in NLP

A token is a subword fragment produced by a deterministic compression algorithm, almost always byte pair encoding or a close relative. It is simultaneously:

- the unit of **billing**, since providers price per token;
- the unit of the **context limit**;
- the unit of **latency**, since decode emits one per forward pass;
- and the unit of the model's **perception**, since it never sees characters.

Reasoning about your system in words or characters will mislead you by a factor that varies with language, domain, and how much punctuation and code your text contains.

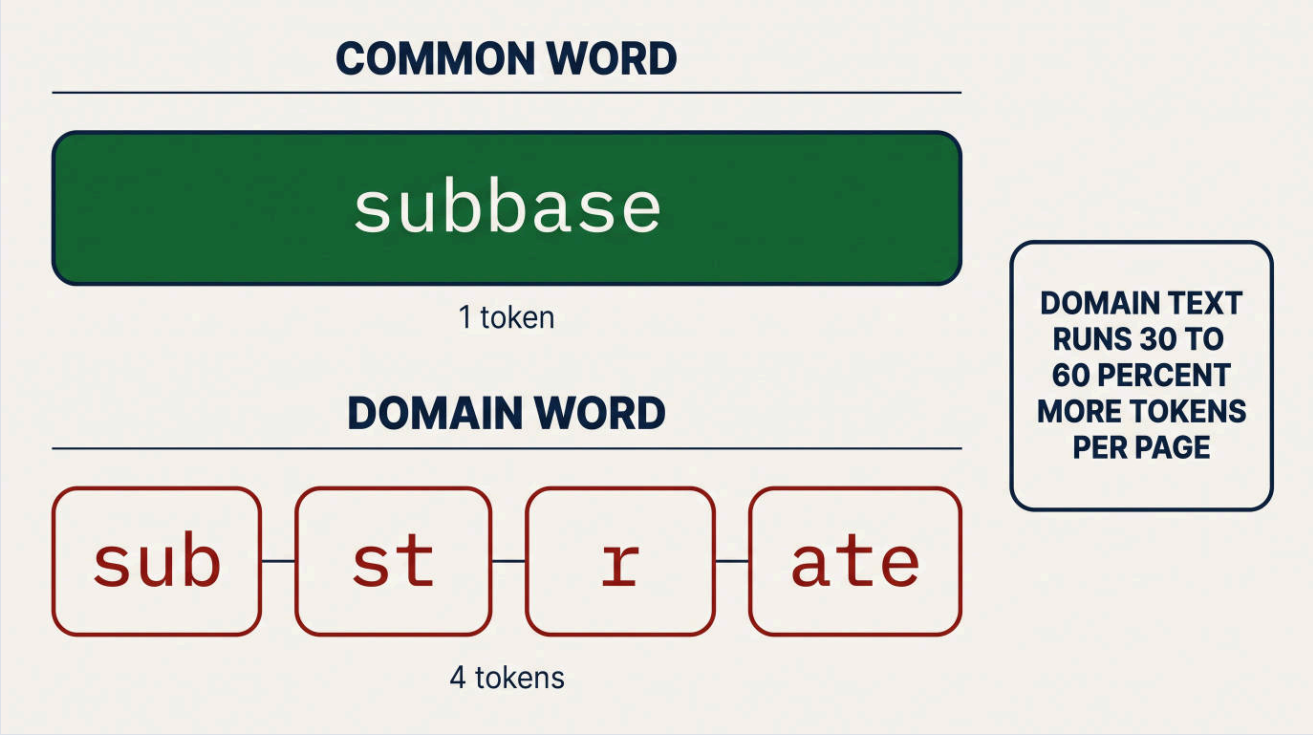

Figure 2.2 Illustrative. Terms frequent in the tokenizer's training corpus become single tokens; terms frequent only in yours do not.

Three consequences, all architectural

Domain vocabulary fragments. Pharmaceutical, legal and engineering text commonly runs thirty to sixty percent more tokens per page than general prose. That is a proportional increase in both cost and prefill latency that no amount of prompt trimming recovers, because it is a property of your corpus rather than your prompt.

Non-English text is penalised. The same meaning expressed in a language underrepresented in the tokenizer's training data can consume two to four times more tokens. If your product is multilingual, *your unit economics differ per market*, and a flat per-user price can be quietly unprofitable in one country and comfortable in another. This is worth raising in a pricing meeting before it appears in a margin report.

Models cannot count letters, because they never see letters. Asking how many times a character appears in a word asks the model to reason about a representation it does not have. This is a tokenization artefact, not a reasoning

failure, and no prompt fixes it. The architectural response is to do that class of work in code and hand the model the answer.

> NEVER ESTIMATE TOKENS WITH A CHARACTER RATIO
>
> "Four characters per token" is an English prose average. It is badly wrong for code, tables, JSON and non-Latin scripts. Budget enforcement built on an estimate fails exactly when the content is unusual, which is exactly when the budget matters. The repository's `tokens/count.py` uses the real encoder, and the authoritative number is always the `usage` block on the provider's response, not your own count.

Measure your own corpus before you build a cost model

```
# src/docassist/tokens/count.py
GENERAL_PROSE_BASELINE = 250.0   # tokens per 1000 characters, English prose

def tokens_per_1k_chars(sample_texts: list[str], model: str = "gpt-4o") -> float:
    enc = _encoding(model)
    toks = sum(len(enc.encode(t)) for t in sample_texts)
    chars = sum(len(t) for t in sample_texts) or 1
    return 1000 * toks / chars

def inflation_factor(sample_texts: list[str], model: str = "gpt-4o") -> float:
    """How much more your corpus costs than general prose, as a multiplier."""
    return tokens_per_1k_chars(sample_texts, model) / GENERAL_PROSE_BASELINE
```

Above roughly 320 tokens per thousand characters, your corpus is expensive and any margin assumption built on prose averages is wrong. The measurement takes twenty minutes.

2.5 Implementing a Custom Tokenizer in Python

You will never ship this. Implementing byte pair encoding once makes the three failure modes above permanently intuitive, which is worth an hour.

The algorithm: start with a vocabulary of individual characters, repeatedly find the most frequent adjacent pair across the corpus, merge it into a single symbol, and record the merge. Encoding replays the recorded merges in order.

```
# src/docassist/tokens/bpe.py
def train_bpe(corpus: list[str], num_merges: int) -> list[Merge]:
    vocab: Counter[tuple[str, ...]] = Counter()
    for line in corpus:
        for word in line.split():
            vocab[tuple(word) + (END,)] += 1

    merges: list[Merge] = []
    for _ in range(num_merges):
        pairs: Counter[Merge] = Counter()
        for symbols, freq in vocab.items():
            for i in range(len(symbols) - 1):
                pairs[(symbols[i], symbols[i + 1])] += freq
        if not pairs:
            break
        best = max(pairs, key=pairs.__getitem__)
        merges.append(best)
        vocab = _apply(vocab, best)
    return merges


def encode(word: str, merges: list[Merge]) -> list[str]:
    symbols = list(word) + [END]
    for a, b in merges:              # ORDER MATTERS: replay exactly as learned
        i = 0
        while i < len(symbols) - 1:
            if symbols[i] == a and symbols[i + 1] == b:
                symbols[i : i + 2] = [a + b]
            else:
                i += 1
    return symbols
```

Run it on a small domain corpus and the effect is immediate:

```
corpus = ["the pavement subbase layer", "the subbase thickness",
          "pavement layer thickness", "the layer above the subbase"] * 40
merges = train_bpe(corpus, num_merges=60)

encode("subbase", merges)     # ['subbase</w>']                    1 token
encode("substrate", merges)   # ['sub','s','t','r','a','t','e</w>'] 7 tokens
```

Seven fragments for a word one character longer than the one that learned as a single unit. The corpus saw *subbase* a hundred and sixty times and *substrate* never, so the merges that would compress it were never learned. Scale that to a real tokenizer and a real domain and you have the thirty to sixty percent inflation from section 2.4, arriving one unfamiliar term at a time.

Production tokenizers add two things this implementation omits: a **byte-level alphabet**, so that any input including emoji and arbitrary bytes is representable rather than mapping to an unknown token, and **pre-tokenization**, which splits on whitespace and punctuation classes before merging so that merges cannot span word boundaries. The merge loop itself is exactly what you see above.

EXERCISE 2.5

Train the toy BPE on a page of your own domain text and on a page of news prose with the same number of merges. Compare how many of your domain's key terms survive as single tokens versus how many fragment. Then run `inflation_factor` from section 2.4 on both with the real encoder. The gap between the toy result and the real one is instructive: real tokenizers were trained on a corpus that already contains a lot of your domain, and the residual inflation is what you actually pay.

2.6 The Transformer Breakthrough: Attention

Before attention, sequence models processed tokens in order and compressed everything seen so far into a fixed size hidden state. That design has two defects: the hidden state is a bottleneck, and the computation is inherently serial, so it cannot exploit a GPU.

Attention replaced the bottleneck with direct access. Every position can read every other position, weighted by learned relevance, and all positions compute in parallel.

```
def attention(Q, K, V, mask=None):
    """Q: what each position is LOOKING FOR.
       K: what each position OFFERS.
       V: what each position CONTRIBUTES if selected."""
    d_k = Q.shape[-1]
    scores = Q @ K.transpose(-2, -1) / np.sqrt(d_k)    # [T, T]
    if mask is not None:
        scores = np.where(mask, scores, -np.inf)       # causal: no lookahead
    weights = softmax(scores, axis=-1)
    return weights @ V, weights
```

Two details in those five lines are worth understanding rather than skimming.

The scaling by the square root of the head dimension is load bearing. Dot products of high dimensional vectors have variance proportional to the dimension. Without the scaling, the scores entering the softmax are large, the softmax saturates into a near one-hot distribution, gradients vanish, and the model does not train. One divide, and the whole thing works.

The causal mask is what makes generation possible. Setting the upper triangle to negative infinity means position i can only attend to positions up to i. Without it, training would let the model see the answer it is meant to predict. With it, the same forward pass computes a prediction for every position simultaneously during training, which is why these models can be trained at all at scale.

The shape that determines everything downstream

Look at the dimensions of `scores`: it is T by T, where T is the sequence length. Compute and memory for attention grow with the **square** of sequence length.

That single fact is the most important number in this book for anyone sizing infrastructure. Section 2.11 turns it into arithmetic. Everything about long context being a capacity decision rather than a configuration flag traces back to this line.

2.7 Role of Positional Encodings in Transformers

Attention as written above is *permutation invariant*. Shuffle the input tokens and the set of outputs is unchanged, because the mechanism has no notion of order. Since word order carries meaning, position must be injected explicitly.

The mechanism has drifted over the years from additive sinusoids to rotary position embeddings, which encode *relative* position by rotating the query and key vectors by an angle proportional to their index. Two practical consequences matter more than the mathematics.

Extended context is often interpolated, not trained

A model advertised at 128k may have been trained at 8k and extended by rescaling the rotation frequencies. This usually works, and it degrades unevenly: retrieval from the far end of an extended window can be measurably worse than the advertised limit suggests.

The architectural instruction: **test long context retrieval on your own data before designing around a published number.** Place a known fact at varying depths in a long context and measure whether the model uses it. It takes an afternoon and it occasionally reveals that your effective window is a third of the advertised one.

Positional degradation is real and reproducible

Material at the beginning and the end of a context is used more reliably than material in the middle. The research literature named this **lost in the middle** when it was first measured, and the name stuck because it describes exactly what happens: the effect reproduces across model families and it is not a bug to be fixed by a better prompt.

This has a direct design consequence that section 6.8 acts on: **the order in which you place retrieved evidence is an engineering decision, not an implementation detail.** A relevant chunk at position fourteen of twenty is measurably less likely to be used than the same chunk at position two, even though both are inside the window and the model was asked to consider all of them.

The practical rule. Put the most relevant evidence first, put the instruction that governs the output last, and keep the total small. The last of those three does more than the first two, which is the subject of section 2.10.

2.8 Understanding Multi-Head Attention

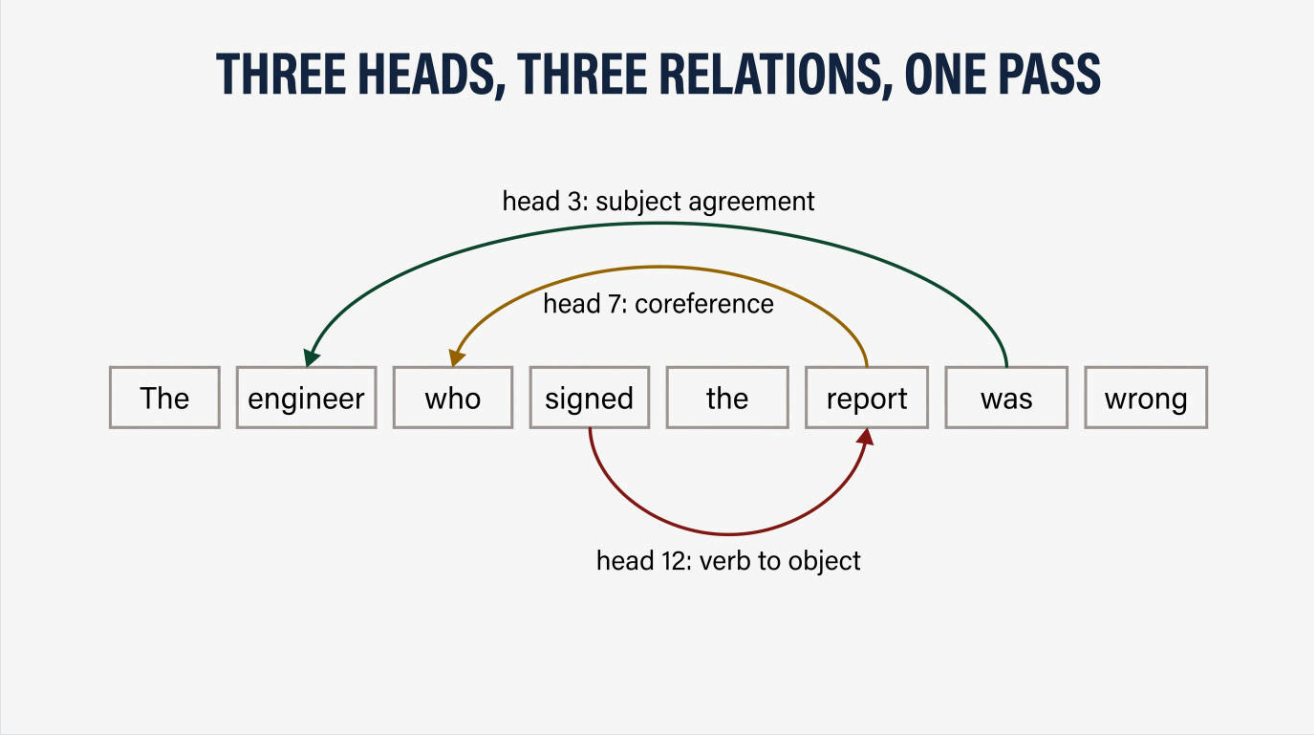

Figure 2.3 A single head produces one weighted mixture per position, so it can express one relation. Splitting the hidden dimension buys several simultaneously at the same total cost.

The reasoning for multiple heads is economical. A single attention head produces one weighted average per position, so it can encode one kind of relationship. Splitting the hidden dimension d into h heads of size d/h buys h simultaneous relation types at the same total parameter count and the same total compute. The cost is a reshape; the gain is expressiveness.

The variant that decides your serving cost

The architecturally decisive detail is not the head count but the **key and value** head count. In grouped query attention, several query heads share one key and value head. This is invisible on every benchmark leaderboard and it changes serving cost by a factor of four.

```
# src/docassist/tokens/capacity.py
def kv_bytes_per_token(self, dtype_bytes: int = 2) -> int:
    # 2 for keys and values
    return 2 * self.layers * self.kv_heads * self.head_dim * dtype_bytes


# Same 8B model, two attention designs:
LLAMA_8B_GQA = ModelShape(layers=32, kv_heads=8,  head_dim=128, params_b=8.0)
LLAMA_8B_MHA = ModelShape(layers=32, kv_heads=32, head_dim=128, params_b=8.0)

LLAMA_8B_GQA.kv_bytes_per_token()   # 131,072  = 128 KiB per token
LLAMA_8B_MHA.kv_bytes_per_token()   # 524,288  = 512 KiB per token
```

Work that through to what it means for a customer. An eight thousand token conversation holds **1 GiB** of KV cache per concurrent sequence under grouped query attention, and **4 GiB** without it. On a 40 GiB accelerator, the repository's capacity function returns 23 concurrent sequences for the first and 5 for the second. Identical parameter count. Identical benchmark scores, most likely. A 4.6x difference in how many users fit on the card you are paying for.

```
def test_grouped_query_attention_is_a_fourfold_serving_difference():
    """Two models of identical parameter count, four times the concurrency.
    No leaderboard reports this."""
    gqa = serving_capacity(40, LLAMA_8B_GQA, 8_000)   # 23
    mha = serving_capacity(40, LLAMA_8B_MHA, 8_000)   #  5
    assert gqa >= 4 * mha
```

WHAT TO READ ON A MODEL CARD

Parameter count tells you almost nothing about serving cost per user. The number you need is `num_key_value_heads` in the config file, alongside `num_hidden_layers` and `head_dim`. Three integers, and they let you compute your concurrency before you rent anything. Sections 4.5 and 4.10 use exactly this arithmetic to size self hosted serving; section 4.11 then sets the result beside utilisation, which is where the decision is actually made.

2.9 Deep Diving into Vector Embeddings

An embedding maps text to a vector such that semantic similarity corresponds to angular proximity. Three properties set a hard ceiling on what any retrieval architecture can achieve, which is why they belong in the physics chapter rather than in Chapter 7.

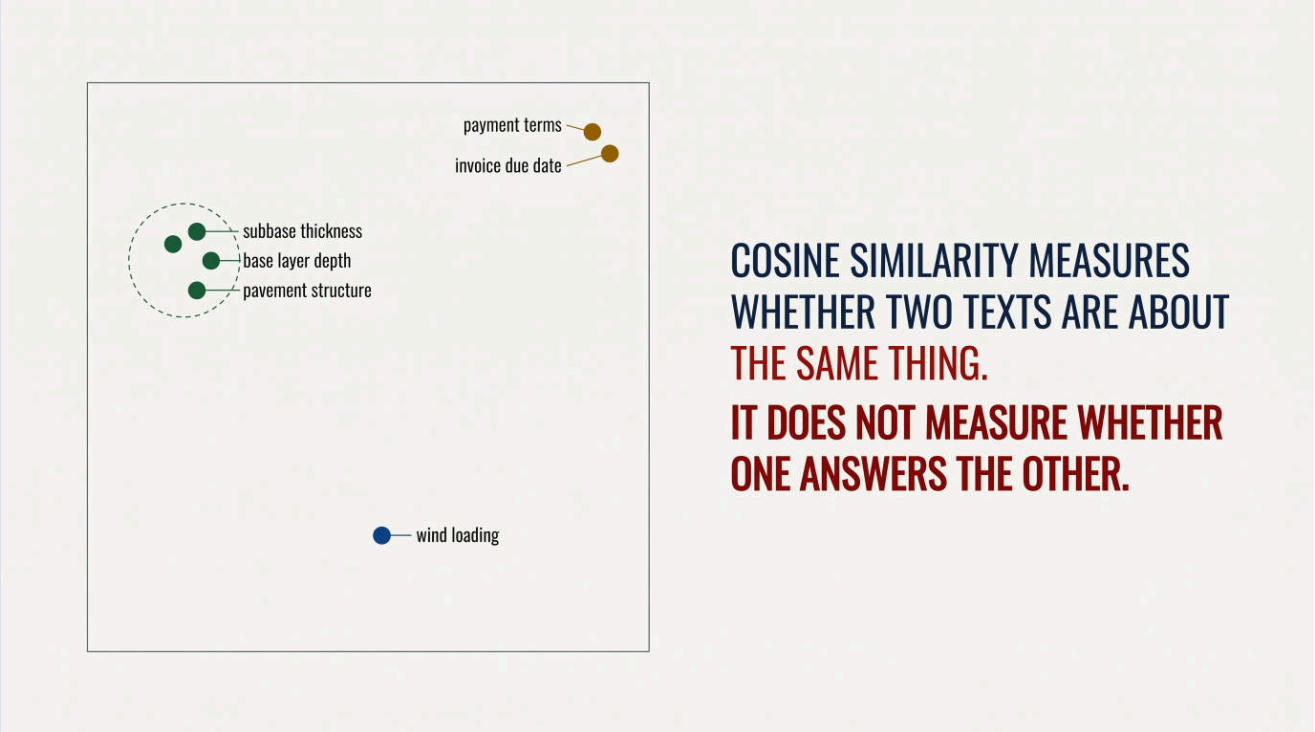

Figure 2.4 Topical neighbourhoods, not answer relations. This gap is irreducible at the embedding layer.

Property one: similarity is not relevance

Cosine similarity measures whether two passages are *about* the same thing. It does not measure whether one *answers* the other.

The canonical example: the question "what is the maximum allowable subbase thickness" and the sentence "subbase thickness shall be determined by the engineer" are extremely similar in embedding space, and the second answers nothing. A bi-encoder embeds the query and the document independently, so it never gets to consider them together, which is precisely why it cannot make this distinction.

This gap is **irreducible at the embedding layer** and it is the entire structural justification for the reranking stage in section 7.7. Reranking is not an optimisation you add when you have time. It is the component that supplies a capability the embedding layer does not have.

Property two: dimensionality trades recall against cost, non-linearly

A 384 dimensional model and a 1536 dimensional model differ by four times in storage, index memory and comparison cost, and typically by a few points of retrieval quality on general corpora.

The correct choice depends on corpus size, and the decision reverses:

Corpus size	Usually right	Why
Below ~100k chunks	Small model, spend the saving on reranking	Index memory is irrelevant at this scale; a cross-encoder over 30 candidates buys more than four times the dimensions.
~100k to ~2M chunks	Measure. It is genuinely close.	Both cost and quality are in play. This is where an afternoon of measurement pays for itself.
Above ~2M chunks	Larger model, or a compressed variant	Index memory dominates operating cost, and quality per byte becomes the governing metric.

Property three: approximate search loses recall, silently

Exact nearest neighbour search is linear in corpus size, so every production vector index is approximate. **Approximate means a configurable probability of not returning the true nearest neighbour.**

If you have never measured your index's recall against exact search on your own data, there is an unknown amount of quality loss sitting under every answer your product gives, and it is invisible in every metric you currently have. The measurement is straightforward:

```
# src/docassist/tokens/recall.py
def index_recall_at_k(index_search, corpus, queries, k: int = 10) -> float:
    hits = 0.0
    for q in queries:
        exact = exact_top_k(corpus, q, k)          # brute force cosine
        approx = set(index_search(q, k))           # your production index
        hits += len(exact & approx) / k
    return hits / len(queries)


def sweep(make_search, corpus, queries, settings, k: int = 10):
    """Sweep an index parameter (ef_search, nprobe, ...) against recall."""
    return [(s, index_recall_at_k(make_search(s), corpus, queries, k))
            for s in settings]
```

A typical finding: HNSW at `ef_search=40` returns recall@10 of about 0.87, and raising it to 128 gives about 0.98 at roughly two and a half times the query latency. On a forty millisecond retrieval budget that trade is nearly always worth taking, and almost nobody makes it deliberately, because the default was never measured.

> THE COMPOUNDING VERSION OF THIS
>
> Recall loss compounds with the positional and dilution effects of section 2.10. If your index returns the right chunk only 87 percent of the time, and when it does return it the chunk lands in a position where the model is less likely to use it, your end to end accuracy is the product of several probabilities you have never measured individually. Section 7.2 gives the procedure for separating them, and it is the single most valuable afternoon in the book.

> EXERCISE 2.9
>
> Take a thousand chunks from your corpus, embed them, and run `index_recall_at_k` against a brute force baseline at your index's default settings. Then sweep the parameter. Write down the recall you were silently accepting, and the latency cost of fixing it. If the recall was above 0.97, note that too: it means your defaults were fine and you now know that rather than assuming it.

2.10 The Context Window Is a Budget

The most common design error in retrieval systems is treating the context window as space to be filled. A 128k window is an *upper bound imposed by the model*, not a target set by your architecture.

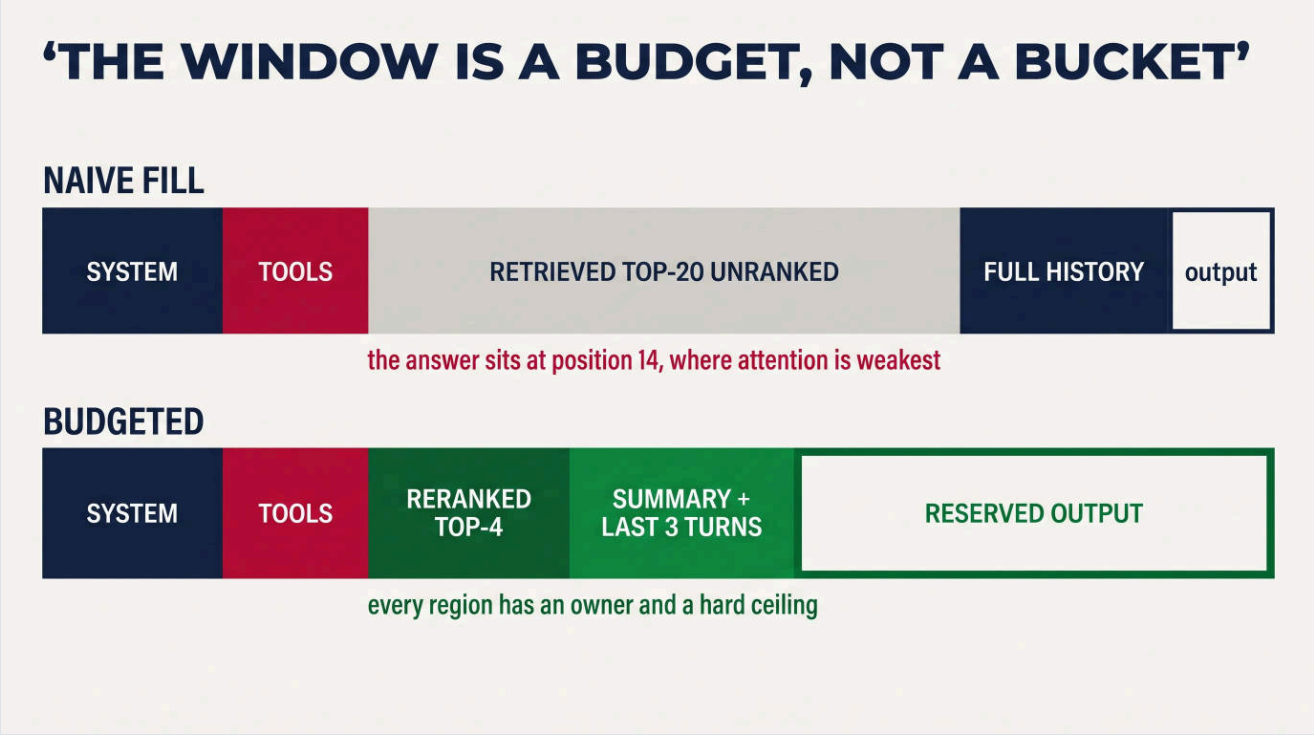

Figure 2.5 The same request assembled two ways. The lower costs roughly a fifth as much and measures better, because relevance density rose while volume fell.

Two effects make naive filling worse than the arithmetic suggests

Positional degradation, from section 2.7: a relevant chunk at position fourteen of twenty is measurably less likely to be used than the same chunk at position two.

Distractor dilution: adding plausible but irrelevant passages reduces answer accuracy *even when the correct passage is still present*. The model has to discriminate, and every additional near-miss makes that harder.

The field's collective name for the pair is **context rot**: the observation that answer quality decays as the context fills, well before the advertised window is reached, so the effective window is a quality budget rather than a capacity. The name is informal; the two mechanisms above are what it decomposes into, and they are the measurable form. Together these produce a conclusion that governs all of Part II: **retrieval quality and generation quality are not independent.** A retriever tuned to maximise recall at k equals twenty can produce worse end to end answers than one tuned for precision at k equals four, while looking better on every retrieval metric you are tracking. Section 7.10 returns to this as the central trap in retrieval measurement.

GOOD

Concatenate, and hope it fits

```
context = "\n\n".join(chunk.text for chunk in hits)
prompt = f"{SYSTEM}\n\n{context}\n\n{history}\n\nQ: {question}"
```

Fast to write, works in a demo, and correct while you are still answering "does retrieval help at all".

What it leaves unbounded: everything. There is no ceiling on any region, so a long document silently crowds out the conversation history, or the history crowds out the evidence, depending on concatenation order. When the total exceeds the window the provider returns an error the user sees, and when it merely approaches the window the request costs forty times a normal one with nothing noticing. Worst of all, *overflow is silent*: a runtime that truncates rather than erroring produces a plausible answer based on two thirds of the evidence, and nothing in your system knows.

BETTER

Named regions with hard ceilings

Give every region of the prompt an owner, a ceiling and a priority. This is `tokens/budget.py` in the repository.

```
@dataclass(frozen=True)
class Region:
    """priority: LOWER survives longer under pressure.

    The system message and the user's actual question must never be the things
    that get evicted, which is exactly what happens when assembly is a string
    concatenation in whatever order someone happened to write it.
    """
    name: str
    max_tokens: int
    priority: int
    compact: Compactor | None = None


def default_doc_qa_budget(window: int = 128_000) -> ContextBudget:
    budget = ContextBudget(
        window=window,
        reserved_output=4_000,
        regions=[
            Region("system",   1_200, priority=0),
            Region("tools",    2_000, priority=1),
            Region("query",    1_000, priority=1),
            Region("evidence", 6_000, priority=2),
            Region("history",  3_000, priority=3),
        ],
    )
    budget.validate()
    return budget
```

Note the totals: roughly thirteen thousand tokens declared against a hundred and twenty eight thousand token window. **That is deliberate.** The window is the model's limit; the budget is your architecture's decision, and it should be an order of magnitude tighter, expanded only where measurement shows it buys quality.

Two details make this more than a set of constants. `validate()` runs at startup and refuses to boot if the declared regions cannot fit, because a budget that only fails under load is not a budget. And truncation is done by binary search on the real token count rather than a character estimate:

```
def _truncate_to_tokens(text: str, limit: int, count: Counter) -> str:
    """A character-ratio estimate is wrong on exactly the content that
    triggers truncation, which is the content that is unusual."""
    lo, hi = 0, len(text)
    while lo < hi:
        mid = (lo + hi + 1) // 2
        if count(text[:mid]) <= limit:
            lo = mid
        else:
            hi = mid - 1
    return text[:lo]
```

What it leaves unbounded: pressure is handled, but silently from the operator's point of view. You know a ceiling exists; you do not know how often it is being hit, on which region, or by how much.

BEST

Overflow as an observable event

The decisive addition is that every act of compaction, truncation or dropping emits a named, quantified event rather than happening quietly.

```
@dataclass(frozen=True)
class BudgetEvent:
    kind: EventKind          # "compacted" | "truncated" | "dropped"
    region: str
    tokens: int              # HOW MUCH was lost, not merely that something was


@dataclass(frozen=True)
class Assembled:
    parts: dict[str, str]
    used_tokens: int
    events: tuple[BudgetEvent, ...]

    @property
    def had_pressure(self) -> bool:
        return bool(self.events)
```

Those events go straight into the trace seam from section 1.6. Three things become possible that were not before.

Early warning. "history compacted" rising sharply means a conversation pattern has changed, usually before any user complains.

Attribution. When a specific answer is wrong, the trace records that the evidence region was truncated by 800 tokens on that request. That converts "the model got it wrong" into "the model never saw it", which is a different bug in a different layer.

A tuning signal. If the evidence region is truncated on twelve percent of requests, either the ceiling is too low or the retriever is returning too much. Both are actionable; neither is visible without the event.

```
def test_overflow_is_an_event_not_a_silence():
    b = ContextBudget(window=100, reserved_output=0,
                      regions=[Region("evidence", 10, 0)])
    out = b.assemble({"evidence": "e" * 400}, count)
    assert out.had_pressure
    assert out.events[0].kind == "truncated"
    assert out.events[0].region == "evidence"
    assert out.events[0].tokens > 0
```

PROMOTION TRIGGERS

Good to better: as soon as any region can vary in size beyond your control, which means as soon as a user can upload a document or hold a conversation of unbounded length. In practice, before the first external user.

Better to best: when you have a trace seam to emit into, which after Chapter 1 you do. The events cost about twenty lines. Build them at the same time as the ceilings; splitting the work means the ceilings ship and the events never do.

2.11 Prefill, Decode and the Cost of Long Context

A generation request has two phases with entirely different performance characteristics, and conflating them is why so many latency budgets are wrong.

Phase	What happens	Bound by	Determines
Prefill	The whole input is processed in one parallel pass. Every token attends to every other token.	Compute. Scales with the *square* of input length.	Time to first token
Decode	Output emitted one token at a time. Each new token attends to all previous ones, whose keys and values are cached.	Memory bandwidth. Roughly linear per token.	Time per output token, and therefore total time

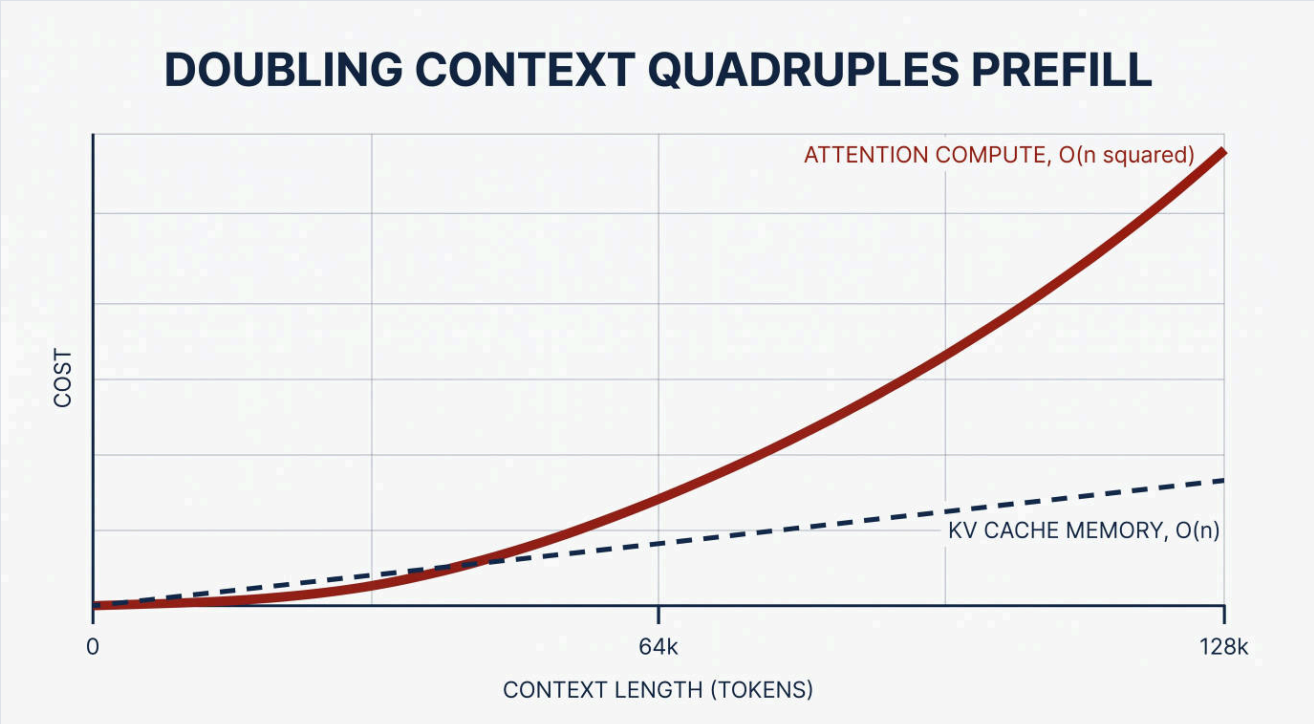

Figure 2.6 The gap between the curves is why long context is a capacity planning problem rather than a configuration flag.

GOOD

Choose an instance type and find the limit empirically

Rent something, load the model, increase concurrency until it falls over, back off twenty percent. This is what most teams do and it produces a number that is roughly right.

What it leaves unbounded: the number is only valid for the context length you happened to test at. When conversations grow, or when a customer uploads longer documents, the limit moves and you discover it the same way you discovered it the first time, which is with an out of memory kill in production. It also gives you no way to evaluate hardware you have not yet rented.

BETTER

Compute it from the KV cache arithmetic

KV bytes per token = 2 (keys and values) × layers × kv_heads × head_dim × bytes_per_element
For a 32 layer, 8 KV head, 128 head-dim model at 16 bit: 2 × 32 × 8 × 128 × 2 = **131,072 bytes = 128 KiB per token**

```
def serving_capacity(vram_gib, shape, avg_context_tokens,
                     bytes_per_param=2.0, overhead_gib=2.0) -> int:
    """How many concurrent sequences actually fit. This is the number that
    determines your cost per user, and almost nobody computes it before
    choosing hardware."""
    free = vram_gib - shape.weight_gib(bytes_per_param) - overhead_gib
    return max(0, int(free // shape.kv_gib(avg_context_tokens)))
```

Now the three lines that change how you think about long context. These are the actual outputs of the repository's function, not estimates:

```
serving_capacity(40, LLAMA_8B_GQA,  8_000, bytes_per_param=2.00)   # 23
serving_capacity(40, LLAMA_8B_GQA,  8_000, bytes_per_param=0.55)   # 34
serving_capacity(40, LLAMA_8B_GQA, 32_000, bytes_per_param=0.55)   #  8
```

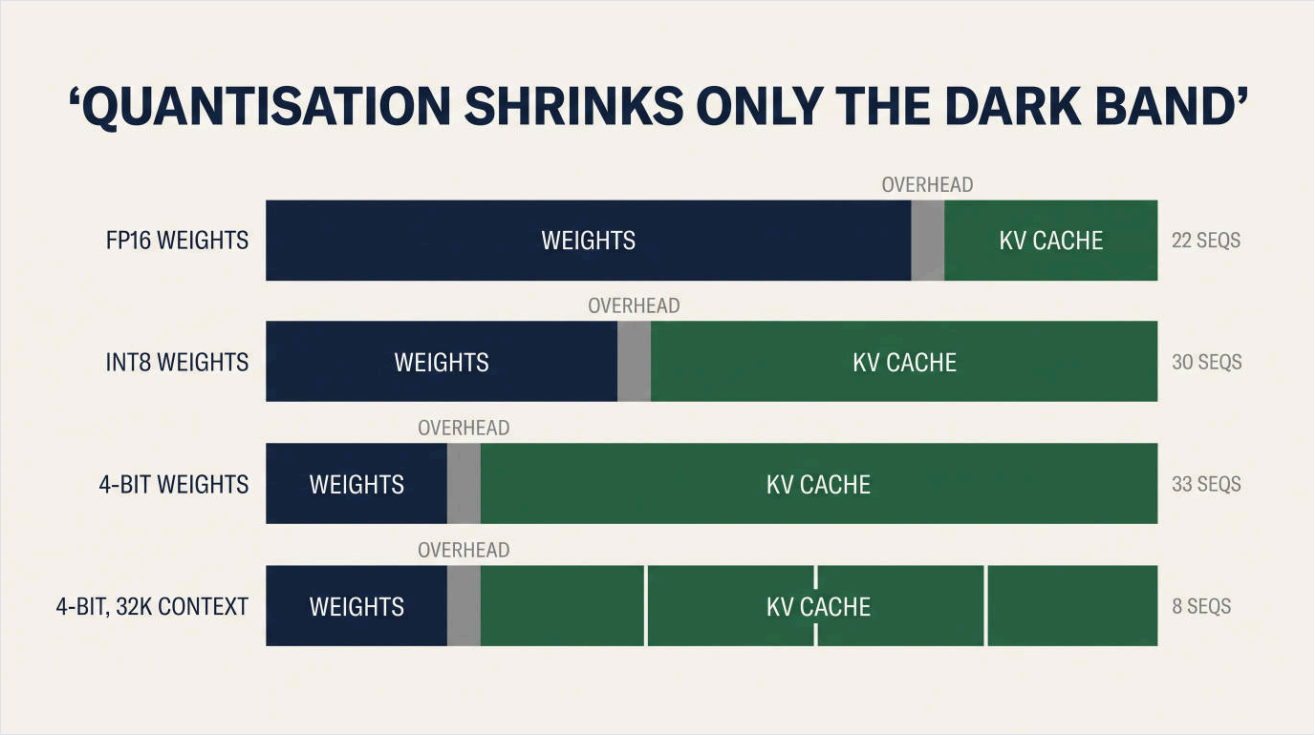

Figure 2.7 Quantisation shrinks the dark band only. The figure's counts are rounded and include an intermediate int8 step for comparison; the exact values from the repository are 23, 34 and 8.

Read the third line carefully, because it is the architectural finding of this chapter. Quantising to four bits bought fifty percent more concurrency, which is a good result for a technique with a real quality cost. Allowing conversations to grow four times longer **gave back four times more than quantisation gained**, and it was almost certainly nobody's explicit decision.

The arithmetic above holds the cache at sixteen bit. Serving stacks now quantise the KV cache itself, typically to eight

bit, which halves the bytes per token and roughly doubles each concurrency figure; the structural conclusion survives, because context length still moves the same number further than any precision choice does.

> CONTEXT LENGTH POLICY IS A CAPACITY DECISION
>
> It should not be made by whoever is writing the conversation history code, and it should not be an emergent property of how long users happen to chat. It belongs in the same conversation as instance sizing, because it moves the same number. This is the specific reason section 2.10's budget caps *history* as well as evidence.

BEST

Make the arithmetic a test that fails on regression

A calculation you performed once is a calculation that will be invalidated by the first person who raises a default. Encode the relationships as assertions:

```
def test_context_length_dominates_quantisation():
    """Quantisation bought 50% more concurrency. Letting conversations grow
    four times longer gave back four times more than that."""
    fp16    = serving_capacity(40, LLAMA_8B_GQA,  8_000, bytes_per_param=2.00)
    q4      = serving_capacity(40, LLAMA_8B_GQA,  8_000, bytes_per_param=0.55)
    q4_long = serving_capacity(40, LLAMA_8B_GQA, 32_000, bytes_per_param=0.55)
    assert q4 > fp16
    assert q4_long < fp16        # the finding, as an executable claim
```

The latency arithmetic to keep in your head

User visible latency is time to first token, plus output tokens multiplied by time per output token. The repository models it directly:

```
m = LatencyModel(prefill_tokens_per_s=10_000, decode_tokens_per_s=50)

m.explain(14_000, 700)
# input 14,000 -> TTFT 1.40s | output 700 -> 14.00s | total 15.40s | output is 91% of the wait

m.explain(3_400, 700)     # trimmed the PROMPT by 75%
# input 3,400 -> TTFT 0.34s | output 700 -> 14.00s | total 14.34s

m.explain(14_000, 300)    # trimmed the ANSWER by 57%
# input 14,000 -> TTFT 1.40s | output 300 -> 6.00s | total 7.40s
```

Trimming the prompt from fourteen thousand tokens to three thousand four hundred saved about one second. Trimming the answer from seven hundred tokens to three hundred saved eight.

This is counter-intuitive and it changes design priorities. **For most interactive products, output length is the dominant latency lever, and it is controlled by the prompt and the output schema rather than by infrastructure.** Teams routinely spend a quarter shaving two hundred milliseconds off retrieval while the model writes a four paragraph preamble nobody reads. The fix is a `max_tokens` ceiling and an output contract that forbids preambles, and it costs an afternoon.

One model class complicates this arithmetic. **Reasoning models** emit a hidden chain of working tokens before the visible answer, and providers bill those tokens as output. They are not bounded by your output contract: a schema that forbids preamble constrains the answer, not the working, and on a hard request the working can run to thousands of tokens. The result is a request whose visible output is two hundred tokens and whose bill and decode time correspond to several thousand.

The levers of this section still work, but the reasoning budget must be counted inside them. Providers expose it as a separate control, an effort level or an explicit thinking-token ceiling, and it belongs under the same governance as `max_tokens`: set deliberately per task tier, recorded in the trace, and never left at the default on an interactive path. Section 3.5 returns to the accounting, because a cost model that ignores reasoning tokens is wrong on exactly the expensive requests.

> EXERCISE 2.11
>
> Take one real request from your product. Record its input tokens, output tokens, TTFT and total latency from the trace seam. Compute what fraction of the wait was decode. Then set `max_tokens` to half your current p50 output length and re-measure quality on your eval set. In most document question answering systems the accuracy is unchanged and the latency halves, because the tokens you removed were preamble.

2.12 What Chapter Two Constrains

Every constraint below is one you will design around for the remaining fifteen chapters. The right hand column is the forward reference.

Physical constraint	Architectural consequence
Tokens are the billing, latency and perception unit, and fragment on domain vocabulary	Count with the real encoder. Unit economics differ per language and per domain. Measure your corpus before pricing. *(2.4, 2.5)*
The model is stateless and has no execution	Memory is something you build (Ch 11). The tool enforcement point is on your side (Ch 12, 15).
Context degrades positionally and dilutes with distractors	Precision beats recall in the assembled prompt. Retrieval and generation quality are not independent. *(6.8, 7.7, 7.10)*
Prefill is quadratic; decode is linear and bandwidth bound	Output length dominates interactive latency. Long context is a capacity decision, not a product one. *(3.5, 8.1, 16.2)*
KV cache is computable per token, and GQA changes it fourfold	Concurrency per accelerator is arithmetic. Read `num_key_value_heads`, not just parameter count. *(4.5, 4.11)*
Cosine similarity is topical, not answer bearing	A reranking stage is structural, not an optimisation. *(7.7)*
ANN indexes lose recall silently	Index recall must be measured against exact search and monitored. *(7.2, 7.10)*

Chapter checklist

Can you...	Section
State what a language model is in one sentence, and derive from it why memory must be built	2.1
Explain why temperature zero is not determinism, and name the three sources of variation	2.2
Say where two thirds of a model's parameters live, and why that matters for quantisation	2.3
Measure your own corpus's token inflation factor and say what it implies for pricing	2.4, 2.5
Explain why attention is quadratic, from the shape of the score matrix	2.6
Name the design consequence of positional degradation	2.7, 2.10
Compute concurrent sequences per accelerator from three numbers on a model card	2.8, 2.11
State why reranking is structural rather than optional	2.9
Explain why a budget with named regions beats a larger window	2.10
Say which lever, prompt length or answer length, dominates your product's latency	2.11

What Chapter 3 builds

Chapter 2 established the physics. Chapter 3 builds the first thing that has to respect it: the model access layer. How a request reaches a model, three times over. The direct call that ships first and takes down the marketing site during a provider incident. The gateway that bounds the damage. And the policy routed gateway with tier routing, a semantic cache and a budget ledger, which is where the cost per successful task from section 1.6 stops being a number you compute and becomes a number you control.