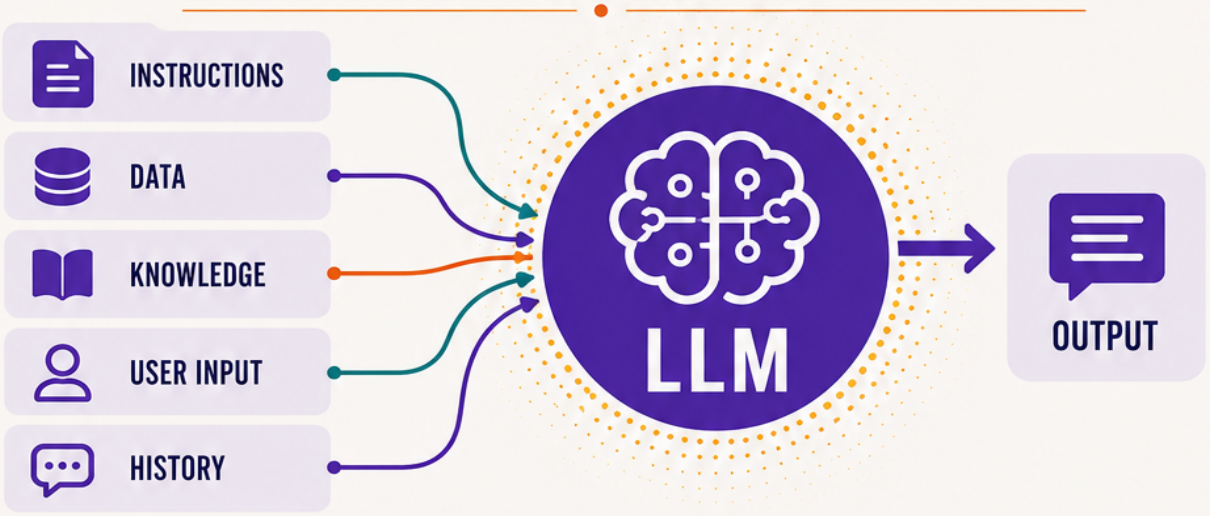
CONTEXT ENGINEERING

DESIGNING INTELLIGENT SYSTEMS WITH LARGE LANGUAGE MODELS



PROVEN PATTERNS THAT WORK

PRACTICAL EXAMPLES WITH CODE

UNDERSTAND TRADE-OFFS MAKE BETTER CHOICES

BUILD RELIABLE PRODUCTION SYSTEMS

STEVE T.

Context Engineering

Designing Intelligent Systems with Large Language Models

Steve Publications

This book is available at https://leanpub.com/contextengineering

This version was published on 2026-07-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Designing Intelligent Systems with Large Language Models

This book teaches you how to design, build, and operate production systems powered by large language models by treating context as a first-class architectural concern. You will learn the theory behind how LLMs use context, the patterns that work in real systems, the trade-offs between competing approaches, and the engineering practices that separate fragile prototypes from reliable products. The material is organized to take you from foundational concepts through advanced production techniques, with complete code examples and detailed explanations of every major pattern in modern AI system design.

Introduction: The Context Problem

In early 2024, a fintech startup launched what should have been a straightforward product: an AI assistant that could answer customer questions using the company's internal documentation. The architecture was simple enough. Documents were chunked into pieces, embedded, stored in a vector database, and retrieved when users asked questions. The team used one of the best models available at the time, with a generous context window and strong evaluation scores on public benchmarks.

For two weeks, everything seemed to work. Then problems began appearing in production that had never shown up during development. Customers received answers that cited policies from a different product line entirely. Some queries returned responses that contradicted information clearly present in the retrieved documents. The system occasionally hallucinated features and pricing that did not exist. When engineers investigated, they found that the retrieval was often working correctly, pulling relevant documents from the knowledge base, but the model was either ignoring them or mixing in incorrect information from its training data.

The root cause was not a bad model or poorly written code. It was context: too much of it in some cases, too little in others, inconsistently structured, and never treated as a first-class component of the system design. The team had focused on getting the pieces to work individually but had not designed how those pieces would fit together into a coherent information package for the model to reason over.

This story is not unusual. It is, in fact, representative of what happens when organizations build LLM-powered systems without treating context as a deliberate engineering concern. The pattern repeats across industries: teams invest heavily in selecting powerful models, building retrieval pipelines, and training staff on prompt techniques, only to discover that their systems perform unreliably once they encounter the messy conditions of real-world use.

The field has a name for this gap between potential and performance, and it is growing faster than most people realize. Context engineering is

the discipline of designing how information flows into large language models so that they can produce reliable, accurate, and useful outputs at scale. It encompasses everything from how you structure instructions to how you retrieve documents, manage conversation history, define tools, orchestrate multi-step workflows, and govern what information a model is allowed to see.

Context engineering is not prompt engineering with a new label. Prompt engineering focuses on crafting individual inputs to coax desired behavior from a model. Context engineering treats the entire information environment of an LLM call as something to be designed, measured, and optimized systematically. It recognizes that the success of a production system depends less on any single prompt and more on how all the contextual elements work together: system instructions, retrieved documents, conversation history, tool definitions, memory artifacts, and structured constraints.

The shift in terminology reflects a deeper shift in understanding. Early adopters of LLMs discovered that cleverly worded prompts could produce impressive results on specific tasks. But as systems grew more complex, it became clear that prompt-level tricks were insufficient for production reliability. The real work lies in designing architectures that manage context effectively: deciding what information to include, when to retrieve it, how to format it, and how to validate that the model used it correctly.

Research supports this view. A comprehensive survey of retrieval-augmented generation systems published in 2024 found that over 70 percent of errors in modern LLM applications stem not from insufficient model capability but from incomplete, irrelevant, or poorly structured context [1]. The models are often perfectly capable of doing what you ask. They fail because they do not have the right information, or because the information they have is organized in ways that make it difficult to use correctly.

This book is built around a central thesis: context is the fundamental resource in LLM-based systems, and how you design, manage, optimize, and govern context determines whether those systems succeed or fail in production. Mastering context engineering requires understanding everything from transformer mechanics to distributed retrieval architectures to evaluation methodologies. It is not an optional layer you add on top of your AI system. It is the core of what you are building.

The chapters that follow are organized to take you from first principles through advanced production techniques. We begin with foundations:

defining context engineering as a discipline, explaining how transformers actually use context, and establishing the conceptual vocabulary you need for everything that follows. From there, we move through prompt construction patterns, retrieval-augmented generation architectures, vector search and semantic indexing, memory systems, tool use and the Model Context Protocol, agent architectures, structured outputs, orchestration frameworks, performance optimization, evaluation methodologies, security considerations, production deployment at scale, and emerging research directions.

Each chapter is designed to stand on its own while building on earlier material. You will find complete code examples in Python, TypeScript, Go, Rust, and SQL that you can adapt for your own systems. We discuss trade-offs explicitly: when to use one approach over another, what problems each technique solves and creates, and how to make informed decisions based on your specific requirements.

This book is written for software engineers, AI practitioners, researchers, and experienced developers who are building production systems with LLMs. You should be comfortable with programming concepts and have some familiarity with machine learning terminology, but we explain everything from first principles so that you do not need deep expertise in any particular area to benefit.

By the time you finish, you will understand why context engineering matters as a discipline, know the major patterns and techniques for designing context-intensive systems, and be equipped to build AI applications that are reliable, scalable, and maintainable in real-world conditions. The journey ahead is substantial, but it is also necessary: if you are serious about building AI systems that work in production, you cannot afford to treat context as an afterthought.

Chapter 1: Foundations: What Is Context Engineering?

From Prompts to Systems: The Evolution of LLM Interaction

The history of how humans interact with large language models is short, compressed, and instructive. Understanding it helps explain why context engineering emerged as a distinct discipline and why earlier approaches are insufficient for production systems.

When GPT-3 was released in 2021, the dominant interaction pattern was zero-shot prompting: users typed natural language requests into a text box and received responses. The models were impressive for their time, but they operated within tight constraints. Context windows of around 2,048 tokens meant that each interaction had to be self-contained. There was no conversation history management, no external knowledge retrieval, no tool integration. The prompt was the entire universe the model could see.

As models grew more capable and context windows expanded, prompt engineering emerged as a craft. Practitioners discovered that carefully structured prompts with explicit instructions, few-shot examples, and chain-of-thought reasoning patterns could dramatically improve output quality. By 2023, prompt engineering had become a recognized skill, with consultants charging premium rates to optimize individual prompts for specific tasks [2].

The limitations of this approach became apparent quickly. Prompts that worked well in isolation proved fragile when embedded in larger systems. They broke when users phrased questions differently than expected, failed when relevant information was not included in the prompt itself, and produced inconsistent results across similar inputs. Teams building production applications discovered that prompt engineering alone could not solve fundamental problems: how to give models access to up-to-date domain knowledge, how to maintain state across multiple interactions, how to integrate with external APIs and databases, or how to ensure outputs met strict reliability requirements.

The response was a wave of architectural innovation. Retrieval-augmented generation (RAG) emerged as the dominant pattern for injecting external knowledge into LLM calls. Tool calling mechanisms allowed models to invoke functions, query databases, and trigger workflows. Memory systems were designed to persist information across sessions. Agent architectures interleaved reasoning with action, allowing models to plan, execute, observe results, and adapt their strategies.

With each new capability came new complexity. A single LLM call in a production system might now include system instructions, conversation history spanning dozens of turns, retrieved documents from multiple sources, tool definitions for dozens of available functions, structured output schemas, and various metadata artifacts. The total context could easily exceed 50,000 tokens. Designing this information environment became its own engineering challenge.

The field needed a name for this work. In June 2025, Tobi Lütke, CEO of Shopify, tweeted that context engineering described the core skill better than prompt engineering: "the art of providing all the context for the task to be plausibly solvable by the LLM" [3]. Andrej Karpathy echoed this view around the same time, stating that in every serious LLM application, context engineering rather than prompt engineering is the fundamental capability [4]. Simon Willison extended the framing, arguing that context engineering involves deliberately assembling system prompts, retrieved documents, conversation history, and tool outputs into a coherent information package [5].

The shift from prompt engineering to context engineering is not merely semantic. It represents a change in scope, methodology, and maturity:

- Scope: Prompt engineering focuses on individual inputs. Context engineering treats the entire information environment of an LLM call as the design space, including how different contextual elements interact with each other.
- Methodology: Prompt engineering often relies on trial-and-error iteration with subjective evaluation. Context engineering applies systematic design principles, quantitative measurement, and rigorous testing to context-related decisions.
- Maturity: Prompt engineering can be done ad-hoc in a chat interface. Context engineering requires architectural thinking, version control, monitoring, and continuous improvement processes comparable to traditional software engineering practices.

This evolution mirrors the history of other engineering disciplines. Early web developers hand-crafted HTML pages with inline styles and JavaScript. Over time, the field matured into structured frameworks, design systems, build pipelines, and automated testing. Context engineering is undergoing a similar maturation, moving from artisanal craft to systematic discipline.

Defining Context Engineering: Scope, Goals, and Principles

Context engineering is the practice of designing, managing, and optimizing the information that large language models receive during inference so that they produce reliable, accurate, and useful outputs for specific tasks in production environments.

This definition contains several important elements worth unpacking.

Designing implies intentionality. Every piece of context included in an LLM call should serve a purpose: providing instructions, supplying factual information, establishing constraints, or enabling capabilities. Context that does not contribute to the model's ability to complete its task is noise that increases cost and potentially degrades performance.

Managing acknowledges that context is dynamic. In real systems, context changes over time: conversation histories grow, documents are updated, tools are added or modified, user preferences evolve, and external data sources fluctuate. Effective context engineering requires mechanisms for updating, pruning, invalidating, and versioning contextual elements as conditions change.

Optimizing recognizes that context has costs. Tokens consume API credits or GPU cycles. Larger contexts increase latency. Irrelevant context can confuse models and reduce accuracy. Optimization involves balancing completeness against efficiency: including enough information for the model to succeed while excluding what is unnecessary.

Reliable, accurate, and useful specifies the goals. Reliability means the system behaves consistently across similar inputs and over time. Accuracy means outputs are factually correct and grounded in the provided context rather than hallucinated. Useful means outputs actually help users accomplish their tasks, not just that they look plausible.

Production environments adds a crucial qualifier. Techniques that work in demonstrations or controlled experiments often fail when faced with real users, edge cases, scale, and operational constraints. Context engineering for production requires attention to failure modes, observability, rollback capabilities, and continuous monitoring.

The scope of context engineering encompasses several interconnected domains:

- Prompt design: structuring instructions, examples, and formatting guidelines that guide model behavior
- Retrieval architecture: selecting, chunking, embedding, storing, and fetching external knowledge relevant to each request
- Memory systems: managing conversation history, user preferences, learned patterns, and persistent state across interactions
- Tool integration: defining functions, APIs, and capabilities that models can invoke, along with the schemas and documentation they need to use them correctly
- Orchestration: designing workflows that sequence multiple LLM calls, tool invocations, and data processing steps into coherent operations
- Output control: constraining model outputs through structured schemas, validation rules, and post-processing pipelines
- Evaluation: measuring how well contextual elements contribute to desired outcomes and iterating based on empirical evidence
- Security and governance: ensuring that context does not leak sensitive information, introduce vulnerabilities, or violate compliance requirements

Several principles guide effective context engineering across these domains.

First, context should be minimal but sufficient. Include only what the model needs to complete its task correctly, and make sure that everything included is actually needed. Extraneous context wastes tokens, increases latency, and can degrade performance by introducing noise that competes with relevant information for the model's attention.

Second, context should be structured deliberately. Models respond better to organized, clearly labeled information than to unstructured text dumps. Use headings, sections, XML tags, or other formatting conventions to help

the model distinguish between different types of content: instructions versus facts, examples versus rules, current query versus background information.

Third, context should be versioned and tested like code. Treat prompts, retrieval configurations, tool definitions, and other contextual elements as first-class software artifacts with their own change management, testing, and deployment processes. Changes to context can have dramatic effects on system behavior, and those changes need to be tracked, reviewed, and validated.

Fourth, context should be observable and debuggable. You cannot improve what you cannot measure. Production systems need instrumentation that captures what context was provided for each LLM call, how the model used it, and whether the result met quality standards. This observability enables debugging when things go wrong and continuous improvement over time.

The Three Layers of Context: Task, Domain, and System

Understanding context as layered helps clarify design decisions. Every LLM call in a production system typically includes contextual elements at three distinct levels, each serving different purposes and requiring different management strategies.

Task context is specific to the immediate request. It includes the user's query, the current conversation turn, any files or data the user has provided, and the specific instructions for what the model should do right now. Task context is ephemeral: it exists only for a single interaction or short sequence of interactions and then becomes historical information.

The design challenge for task context is relevance filtering. Users often provide more information than is necessary, or ask questions that are ambiguous or poorly formed. Effective systems preprocess task context to extract the essential elements, clarify ambiguities, decompose complex requests into manageable subtasks, and discard irrelevant details before passing information to the model.

Domain context provides the specialized knowledge the model needs to perform correctly in a particular area. This might include product documentation, legal regulations, company policies, technical specifications, or any other information that is not part of the model's general training data but is essential for accurate responses in your specific application.

Domain context requires persistent management. It must be kept up to date as underlying knowledge changes, organized so that relevant pieces can be retrieved efficiently, and validated to ensure accuracy. Poorly managed domain context is a primary source of errors in production systems: outdated information leads to wrong answers, incomplete coverage forces the model to rely on its training data (and potentially hallucinate), and disorganized structures make it difficult for the model to find what it needs.

System context defines how the system behaves as a whole, independent of any specific task or domain. It includes system instructions that establish the model's role, tone, and operating constraints; tool definitions that specify available capabilities; output format requirements; security guardrails; and other cross-cutting concerns that apply uniformly across all interactions.

System context is the most stable layer but also the most consequential when it is wrong. A flaw in system instructions can cause every interaction to behave incorrectly. Poorly designed tool definitions can lead models to call the wrong functions or pass invalid arguments. Inconsistent output format requirements can break downstream processing. System context needs careful design, thorough testing, and disciplined change management because its effects are pervasive.

These three layers interact in important ways. System context constrains how task and domain context are processed: it might instruct the model to prioritize certain sources of information, follow specific reasoning patterns, or apply particular validation rules. Domain context enriches task context by providing the factual foundation the model needs to address the user's request accurately. Task context activates relevant portions of domain and system context, determining which pieces of stored knowledge are actually needed for a given interaction.

A common mistake is treating all context as if it belongs at the same level. Systems that mix system instructions with conversation history, or embed domain facts directly into task prompts without retrieval, create brittle architectures that are difficult to maintain and scale. Separating context by layer allows each to be managed appropriately: system context can be versioned and deployed like configuration, domain context can be updated independently through data pipelines, and task context can be processed dynamically for each request.

Why Naive Prompting Fails in Production

To understand why context engineering is necessary, it helps to examine what happens when organizations rely on naive prompting strategies in production environments. Several failure modes appear consistently across different applications and domains.

The first is context overflow. As systems grow more complex, the temptation is to include everything that might be relevant: extensive system instructions, complete conversation histories, large retrieved documents, detailed tool definitions, and comprehensive examples. This approach seems safe in principle: if the model has access to all possible relevant information, it should be able to find what it needs.

In practice, context overflow degrades performance. Models exhibit a well-documented phenomenon known as "lost in the middle," where their ability to use information depends heavily on its position within the context window [6]. Information placed at the beginning or end of the context is used reliably, while information in the middle is frequently ignored, even when it is directly relevant to the task. Simply adding more context does not linearly improve performance; beyond a certain point, additional context introduces noise that interferes with the model's ability to focus on what matters.

The second failure mode is context staleness. Naive approaches often embed domain knowledge directly into prompts or system instructions, treating them as static artifacts that rarely need updating. When underlying facts change, the prompts become outdated, and the model produces responses based on incorrect information. This is particularly problematic in domains with rapidly changing knowledge: financial regulations, product catalogs, technical documentation, or news content.

Effective context engineering addresses staleness through retrieval mechanisms that fetch current information at inference time rather than relying on baked-in knowledge. But naive systems frequently implement retrieval poorly: using inappropriate chunking strategies that fragment important information, selecting embedding models that do not capture domain-specific semantics, or failing to update their indexes when source documents change.

The third failure mode is context inconsistency. In complex systems with multiple LLM calls, different components may receive conflicting or redundant context. One agent might be instructed to use a particular format while

another expects something different. Retrieved documents might contradict each other. Tool definitions might overlap in confusing ways. Without deliberate coordination, these inconsistencies propagate through the system and produce unreliable behavior.

The fourth failure mode is context fragility. Naive prompts often depend on specific phrasing, formatting, or structural assumptions that break when conditions change slightly. A prompt that works perfectly with one model may fail with another. A prompt optimized for English may not translate to other languages. A prompt that assumes a fixed set of tools breaks when tools are added or removed. Fragile context requires constant manual tuning and does not scale.

Robust context engineering uses patterns that reduce fragility: structured formats that are explicit about their expectations, schema-driven outputs that enforce consistency, retrieval systems that adapt to available data, and orchestration frameworks that manage context flow systematically across multiple components.

The fifth failure mode is context opacity. Naive systems provide no visibility into what context was provided for each LLM call, how the model used it, or why it produced a particular output. When things go wrong, engineers cannot diagnose the root cause because they lack the necessary information. They cannot tell whether the model ignored relevant retrieved documents, misinterpreted instructions, hallucinated facts, or encountered a genuinely ambiguous situation.

Observable context engineering instruments every stage of context assembly and usage, capturing enough detail to reconstruct what happened for any given interaction. This observability is essential not only for debugging failures but also for continuous improvement: analyzing patterns in how context contributes (or fails to contribute) to successful outcomes.

The Cost of Context: Tokens as a First-Class Resource

Context is expensive, and the costs manifest in multiple dimensions that engineers must account for when designing production systems.

The most obvious cost is financial. API providers charge per token, with different rates for input and output tokens. A system that includes 30,000 tokens of context for every request, much of it redundant or irrelevant, incurs

significantly higher costs than one that carefully curates context to include only what is necessary. At scale, the difference between a well-designed and poorly designed context strategy can amount to thousands or tens of thousands of dollars per month.

Latency is another critical cost. Processing longer contexts takes more time, both during the prefill phase when the model reads the input and during generation when it attends to the context while producing output. Users perceive latency directly: a response that takes ten seconds instead of two feels sluggish and unresponsive, regardless of its quality. Context engineering techniques that reduce unnecessary context or cache frequently used contextual elements can dramatically improve perceived performance.

There is also a correctness cost. As discussed above, adding more context does not guarantee better results. Irrelevant context can confuse models, cause them to attend to the wrong information, or introduce contradictions that make it difficult for them to produce accurate responses. The marginal benefit of additional context diminishes rapidly, and beyond a certain point, additional context actively harms performance.

Resource constraints matter as well. In self-hosted deployments, context length directly affects memory usage through the key-value cache that stores intermediate attention computations. Longer contexts require more GPU memory, which limits batch sizes, reduces throughput, and may force teams to use smaller models or fewer concurrent users than they would otherwise prefer [7].

Treating tokens as a first-class resource means making deliberate trade-offs about what context to include, when to retrieve it, how much detail is appropriate, and whether caching or compression techniques can reduce consumption without sacrificing quality. It means measuring token usage across different components of your system, identifying hotspots where context is excessive, and optimizing those areas systematically.

Context engineering is not about minimizing context at all costs. It is about ensuring that every token included serves a purpose and contributes to the model's ability to produce the desired output. When context is designed with this discipline, systems become more reliable, faster, cheaper, and easier to maintain. Without it, they become fragile, expensive, and unpredictable.

Chapter 2: The Context Window: Architecture and Mechanics

How Transformers Use Context: Attention and Positional Encoding

The context window is not a simple buffer that holds text for the model to read. It is an architectural feature with specific computational properties that determine how information flows through the system, what the model can do with that information, and where its limitations lie. Understanding these properties is essential for designing effective context strategies.

At the core of every transformer-based language model is the self-attention mechanism. When a model processes input text, it first converts each token into a vector representation called an embedding. These embeddings capture semantic meaning: similar tokens produce similar vectors, and the relationships between vectors encode information about how tokens relate to each other.

Self-attention then computes relationships between all pairs of tokens in the context window. For each token, the mechanism produces three vectors: a query vector that represents what the token is looking for, a key vector that represents what the token offers, and a value vector that contains the token's actual content. The attention score between any two tokens is computed as the dot product of one token's query with another token's key, scaled and passed through a softmax function to produce a probability distribution.

Mathematically, for a sequence of tokens with queries Q, keys K, and values V, the attention output for each position is:

Attention(Q, K, V) = softmax(QK^T / sqrt(d_k))V

where d_k is the dimension of the key vectors. The scaling factor prevents the dot products from becoming too large, which would push the softmax into regions with vanishing gradients.

This computation means that every token in the context window can potentially influence every other token. The attention weights determine how much influence each token has: tokens with high attention scores contribute more to the output representation, while tokens with low scores contribute less. This is why the model can track references across long distances, understand that a pronoun refers back to an entity mentioned earlier, or connect related concepts scattered throughout a document.

Multi-head attention extends this mechanism by running multiple independent attention computations in parallel, each with its own learned query, key, and value projections. Different heads can specialize in capturing different types of relationships: one head might focus on local syntactic dependencies, another on long-range semantic connections, and another on specific patterns like named entities or numerical values. The outputs from all heads are concatenated and linearly transformed to produce the final representation.

The context window size determines how many tokens can participate in these attention computations simultaneously. When a model has a 128K token context window, it means that up to 128,000 tokens can be present in the input sequence, and each token can attend to all others (subject to causal masking in autoregressive models, which prevents attending to future tokens).

Positional encoding is equally important. The self-attention mechanism itself is permutation-invariant: it treats tokens as a set rather than a sequence. If you rearrange the order of tokens, the attention computations would produce different results only because the queries and keys change positions, not because the mechanism inherently understands order. To give the model information about token positions, positional encodings are added to the input embeddings.

The original transformer used sinusoidal positional encodings: fixed mathematical functions that encode each position as a unique vector pattern. Modern models typically use more sophisticated approaches. Rotary Position Embedding (RoPE) has become particularly popular because it encodes relative positions through complex-plane rotations, which naturally captures how the relationship between two tokens depends on their distance from each other [8]. ALiBi (Attention with Linear Biases) takes a different approach, adding position-dependent biases to attention scores that penalize attending to distant tokens while allowing some heads to maintain long-range dependencies [9].

These positional encoding schemes have important implications for context window behavior. RoPE-based models can sometimes generalize to longer contexts than they were trained on, because the rotation mechanism scales smoothly with distance. However, this generalization is not unlimited: beyond a certain point, the model encounters positions it has never seen during training, and its ability to use information at those positions degrades. ALiBi tends to extrapolate more gracefully but may underperform on very long-range dependencies compared to well-tuned RoPE variants.

The quadratic complexity of self-attention is a fundamental constraint. Computing attention for n tokens requires O(n^2) operations and memory, because every token must attend to every other token. For a 128K token context, this means approximately 16 billion pairwise attention computations per layer. This complexity drives many of the optimizations and architectural choices in modern long-context models.

KV Caching and Its Implications for Latency

KV caching is an optimization that dramatically affects how context length impacts inference performance, and understanding it is crucial for designing systems with predictable latency characteristics.

During autoregressive generation, a model produces output tokens one at a time. Each new token depends on all previous tokens in the sequence: both the input prompt and any tokens generated so far. Without caching, computing each new token would require re-processing the entire sequence from scratch, recomputing attention over all previous tokens. For long contexts and long outputs, this would be prohibitively slow.

KV caching avoids this redundancy by storing the key and value vectors computed during the initial processing of the input (the prefill phase). When generating each new token, the model only needs to compute queries for that single token and attend to all cached keys and values. This reduces the per-token computation from O(n^2) to O(n), where n is the current sequence length.

The memory cost of KV caching is significant. The cache stores key and value vectors for every token in the context, for every layer of the model. For a typical large model with 32 or more layers, the KV cache can consume gigabytes

of GPU memory for long contexts. The memory requirement grows linearly with context length: doubling the context doubles the KV cache size.

This has direct implications for latency. The prefill phase, which processes the initial input and builds the KV cache, is computation-bound and scales roughly quadratically with input length. Longer prompts take disproportionately longer to process initially. The decoding phase, which generates output tokens one at a time using the cached keys and values, is memory-bound: each step requires reading the entire KV cache from GPU memory. As the context grows, each decoding step takes longer because more data must be read.

For production systems, this means that context length directly affects both time-to-first-token (TTFT, dominated by prefill) and tokens-per-second during generation (dominated by decoding). A system that includes 50K tokens of context for every request will have significantly worse latency than one that uses 5K tokens, even if the additional context is not actually needed for many requests.

Several optimizations address KV cache costs. Quantization reduces the precision of cached values: FP16 (16-bit floating point) can be reduced to FP8, INT8, or even lower precision with minimal quality degradation [10]. This reduces memory usage and improves memory bandwidth utilization. Paged attention, introduced in vLLM, manages KV cache memory more efficiently by allocating it in variable-sized pages rather than contiguous blocks, reducing fragmentation and allowing better sharing of cached prefixes across concurrent requests [11].

KV cache reuse is another powerful optimization. When multiple requests share a common prefix (such as the same system prompt or shared context), the KV cache for that prefix can be computed once and reused, avoiding redundant computation. This is particularly valuable in systems with standardized prompts serving many users. Some inference frameworks support cross-request KV cache sharing, where cached prefixes are persisted and looked up for new requests that match them [12].

Understanding KV caching helps explain why some context engineering decisions matter more than they might appear. Including a 10K token system prompt that is identical across all requests is less costly than it seems if the inference system can cache and reuse the prefix. But including 10K tokens of unique, per-request context cannot be cached in the same way and will directly impact latency for every request.

Sliding Windows, Long Context, and Compression Strategies

As context window demands have grown beyond what full attention can efficiently support, researchers and engineers have developed various strategies to extend effective context length while managing computational costs.

Sliding window attention limits each token's attention scope to a fixed-size window of nearby tokens. Instead of attending to all n tokens in the sequence, each token attends only to tokens within a window of size w (for example, 4K or 8K tokens). This reduces complexity from O(n^2) to O(nw), which is effectively linear for fixed window sizes.

Sliding window attention works well for tasks where most relevant information is locally proximate: reading comprehension, conversation, and many coding tasks rely primarily on nearby context. However, it struggles with tasks that require connecting information across very long distances. A question at the end of a 100K token document might reference facts from the beginning, which would be outside any practical sliding window.

Several models use hybrid approaches that combine sliding windows with sparse global attention. Tokens at certain positions (such as the first few tokens, or tokens marked as globally important) attend to the full sequence, while most tokens use limited windows. This provides a compromise: efficient processing for local context plus selective long-range connectivity.

Long-context models that support 100K, 1M, or even 10M token windows typically use one of several architectural strategies. Some train with full attention on shorter sequences and rely on positional encoding extrapolation to generalize to longer lengths. Others use techniques like ring attention, which distributes the attention computation across multiple GPUs in a way that allows processing sequences longer than any single GPU's memory can hold [13].

Compression strategies reduce context length by summarizing or filtering information before it reaches the model. Summarization-based compression generates concise summaries of conversation history or retrieved documents, preserving key facts while discarding details. This approach is lossy: some information is inevitably lost in compression, and the quality depends heavily on how well the summarization captures what matters.

Selective context pruning identifies and removes tokens or chunks that are unlikely to be relevant for a given task. Techniques might analyze attention patterns from previous turns, use heuristics based on recency and importance, or employ lightweight models to score the relevance of different contextual elements. Pruning can significantly reduce token counts while preserving the information the model actually needs.

Hierarchical retrieval and processing is another compression-related strategy. Instead of retrieving individual chunks and including all of them in the context, systems first retrieve a larger set of candidates, then use an LLM to summarize or select the most relevant portions. This two-stage approach reduces context length while improving signal-to-noise ratio.

Each strategy involves trade-offs. Sliding windows are efficient but lose long-range connectivity. Full attention on very long contexts is computationally expensive and may degrade in quality due to attention dilution. Compression saves tokens but risks losing important information. The right choice depends on the specific requirements of your application: what types of tasks you support, how much context is actually needed, and what latency and cost constraints you must meet.

Positional Overflow and Context Degradation

Even when a model technically supports a large context window, its ability to use that context effectively does not scale linearly with length. Several phenomena cause context degradation as sequences grow longer.

The lost-in-the-middle effect, documented in research by Liu et al. [6], shows that models are significantly worse at using information placed in the middle of long contexts compared to information at the beginning or end. This pattern resembles human memory effects: primacy (better recall for early items) and recency (better recall for recent items). The effect is robust across different models and tasks, and it has practical implications for how you organize context: critical information should be placed at the beginning or end of the prompt, not buried in the middle.

Attention dilution occurs because attention weights are normalized across all tokens. As the number of tokens increases, each token receives a smaller share of total attention on average. Even if relevant tokens receive higher-than-average attention, the absolute amount of attention they receive de-

creases as more competing tokens are added. This can make it harder for the model to focus on what matters when the context is very long.

Positional overflow happens when a model encounters sequence lengths during inference that exceed its training distribution. Models trained on sequences up to 32K tokens may perform poorly on 128K token inputs, even if their architecture technically supports those lengths. The positional encodings at positions beyond 32K represent patterns the model has never learned to interpret correctly. Various techniques attempt to address this: position interpolation scales down positional frequencies to fit longer sequences into the trained range, while methods like YaRN (Yet another RoPE extension) adjust temperature parameters in rotary embeddings to improve extrapolation [14].

Context rot is a broader term encompassing various ways that model performance degrades as context grows. It includes not only the specific phenomena described above but also practical issues like increased hallucination rates, reduced instruction-following fidelity, and greater sensitivity to formatting inconsistencies. As context length increases, there are more opportunities for the model to encounter confusing or contradictory information, and the probability of at least one problematic element rises.

These degradation effects mean that advertised context window sizes should be treated as upper bounds rather than guarantees. A model with a 1M token context window does not perform equally well on all inputs up to that length. In practice, effective context lengths are often significantly shorter, particularly for tasks requiring precise information retrieval or complex reasoning.

For production systems, this means testing context behavior empirically rather than assuming linear scaling. Evaluate your system's performance at different context lengths, measure how accuracy changes as you add more contextual elements, and identify the practical limits for your specific use case. Do not design your architecture around theoretical maximums that the model cannot actually utilize effectively.

Choosing a Model Based on Context Requirements

Selecting a language model involves many factors, but context-related requirements deserve particular attention because they constrain what architectures and patterns are feasible.

The first question is how much context you actually need. Many teams overestimate this requirement. A customer support bot might seem to need access to all product documentation, but in practice, most queries can be answered using a few relevant excerpts retrieved dynamically. Estimating realistic context needs requires analyzing actual usage patterns: what information do users typically ask about, how much of it is needed per query, and how does this vary across different types of interactions?

For applications with modest context needs (under 32K tokens), most modern models perform well. The choice can focus on other factors like cost, latency, and specific capabilities. For applications requiring substantial context (32K to 128K tokens), you need models specifically trained and tested for those lengths. Many models advertise large context windows but have not been rigorously evaluated at those scales.

For very long contexts (128K+ tokens), the landscape is more constrained and rapidly evolving. Models like Claude with 1M token windows, Gemini with similar capabilities, and emerging open-source models offer these capacities, but they come with trade-offs: higher costs, longer latencies, and potentially less mature tooling and ecosystem support. Before committing to a long-context architecture, consider whether retrieval-based approaches could achieve your goals more efficiently: instead of including all relevant documents in the context, retrieve only what is needed for each specific query.

Cost per token varies significantly across models and providers. A model with a large context window might charge premium rates for inputs beyond certain thresholds. Calculate the total cost of your expected context patterns, not just the base rate. If your typical request includes 50K tokens of context, a model charging $10 per million input tokens costs $0.50 per request before any output is generated. At scale, these costs compound quickly.

Latency requirements also influence model selection. Long-context models typically have higher time-to-first-token due to the prefill computation required. If your application demands sub-second response times, very long contexts may be impractical regardless of model capabilities. Consider whether caching strategies (prefix caching for shared context, semantic caching for similar queries) can mitigate latency while still supporting the context depth you need.

Ecosystem and integration factors matter as well. Some models have better support for structured outputs, tool calling, streaming, or specific frameworks.

If your architecture depends heavily on particular features like guaranteed JSON schema compliance or sophisticated function calling, verify that your chosen model supports those features reliably.

Finally, consider the trajectory of the field. Context capabilities are improving rapidly, and today's limitations may be obsolete in six months. Design your architecture to be adaptable: abstract model interactions behind interfaces that allow swapping models as they evolve, and build evaluation pipelines that can compare different models on your specific tasks. This flexibility lets you take advantage of improvements without requiring major architectural changes.

Key Takeaways

- The context window is an architectural feature with specific computational properties, not a simple text buffer. Understanding attention mechanisms, positional encodings, and KV caching is essential for designing effective context strategies.
- KV caching dramatically affects latency characteristics: prefill is computation-bound and scales roughly quadratically with input length, while decoding is memory-bound and each step requires reading the entire cache. Design systems with these properties in mind.
- Long-context models support large windows but do not use all context equally. The lost-in-the-middle effect, attention dilution, and positional overflow mean that effective context lengths are often shorter than advertised. Place critical information at the beginning or end of prompts.
- Compression strategies (sliding windows, selective pruning, hierarchical retrieval) trade coverage for efficiency. Choose based on whether your tasks require global context access or primarily local coherence.
- Model selection should be driven by actual context requirements, not maximum advertised window sizes. Estimate realistic needs from usage patterns, account for cost and latency implications of your expected context patterns, and design for adaptability as the field evolves.

Chapter 3 builds on this foundation by examining how to construct prompts deliberately as structured information packages rather than free-form text. As we will see, prompt structure interacts directly with the attention properties described here: well-organized prompts leverage primacy and recency effects while minimizing noise that dilutes attention.

Chapter 3: Prompt Construction as Context Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Structural Patterns for Robust Prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Few-Shot Design: Selection, Ordering, and Calibration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

System Instructions vs. User Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Prompt Templates as Code: Versioning, Testing, Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Common Anti-Patterns and How to Avoid Them

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 4: Retrieval-Augmented Generation: Core Patterns

The RAG Pattern: Architecture and Variants

Document Chunking Strategies and Their Trade-offs

Query Transformation: Rewriting, Expansion, and Decomposition

Re-ranking and Multi-stage Retrieval

Evaluating Retrieval Quality Independently of Generation

Advanced RAG Patterns: GraphRAG, Agentic Retrieval, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

RAG Failure Modes and Diagnosis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 5: Vector Search and Semantic Indexing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Embeddings: Selection, Fine-Tuning, and Dimensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Vector Database Architectures: In-Memory vs. Persistent vs. Managed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Approximate Nearest Neighbor Algorithms Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Hybrid Search: Combining Semantic, Lexical, and Metadata Filters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Index Maintenance, Refresh Strategies, and Consistency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 6: Memory Systems and State Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Short-Term vs. Long-Term Memory Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Conversation History Management and Summarization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Persistent Knowledge: Entity Extraction and Knowledge Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Memory in Multi-turn Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Privacy, Consent, and Forgetting in Memory Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 7: Tool Use and the Model Context Protocol

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Tool Calling: Patterns, Schemas, and Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

The Model Context Protocol: Architecture and Design Philosophy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Building MCP Servers: Tools, Resources, and Prompts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Tool Discovery, Selection, and Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Security Considerations for Tool Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 8: Agents, Planning, and Reasoning Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Agent Architectures: From ReAct to Advanced Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Planning with Context: Decomposition and Subgoal Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Reflection, Self-Correction, and Iterative Improvement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Agent Reliability Patterns for Production Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Multi-Agent Systems: Coordination and Communication Protocols

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

When Agents Help and When They Hurt

Chapter 9: Structured Outputs and Controlled Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Schema-Driven Output Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Constrained Decoding and Grammar-Based Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Validation Pipelines: Pre- and Post-Generation Checks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Structured Outputs in Streaming Contexts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Trade-offs: Flexibility vs. Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 10: Orchestration and Workflow Engines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Workflow Patterns: Sequential, Parallel, Conditional, Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Orchestration Frameworks: LangGraph, Temporal, and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Error Handling, Retries, and Circuit Breakers in LLM Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

State Management Across Workflow Steps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Designing Observable and Debuggable Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 11: Performance Engineering: Caching, Optimization, Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Caching Strategies: Exact Match, Semantic, Prefix, and Hybrid

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Token Optimization: Compression, Pruning, and Rewriting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Batching, Parallelization, and Throughput Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Speculative Decoding and Draft Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Latency Budgeting and Performance SLOs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 12: Evaluation, Testing, and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Offline Evaluation: Datasets, Metrics, and LLM-as-Judge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Online Evaluation: A/B Testing and Canary Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Regression Testing for Prompts, Retrieval, and Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Observability: Tracing, Logging, Metrics, and Dashboards

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Building a Continuous Evaluation Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 13: Reliability and Safety: Hallucination Mitigation, Security, Governance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Hallucination: Causes, Detection, and Mitigation Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Prompt Injection and Indirect Attacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Data Privacy, PII Handling, and Confidentiality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Security for Tool Use and External Integrations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Governance: Policies, Guardrails, and Compliance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Advanced Indirect Injection Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

MCP Security Hardening Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 14: Production Deployment at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Cloud-Native Deployment Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Scaling Patterns: Horizontal, Vertical, and Hierarchical

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Multi-Region and Edge Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Cost Management and FinOps for LLM Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Incident Response and Post-Mortem Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 15: Case Studies in Context Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Case Study 1: Optimizing Latency for a Fintech RAG System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Case Study 2: Migrating from Monolithic Prompts to MCP-Based Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Case Study 3: Building a Multilingual Customer Support System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Case Study 4: Implementing Structured Outputs for a Legal Document Analysis Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Case Study 5: Scaling a Research Agent System with Evaluation-Driven Iteration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Chapter 16: Frontiers of Context Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Scaling Laws for Long Context and Their Limits

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Multimodal Context: Vision, Audio, and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

In-Context Learning Theory: What We Know and Do Not Know

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Neuro-Symbolic Integration and Structured Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

The Next Three Years: Trends to Watch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Conclusion: The Discipline of Context

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Index

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

A

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

B

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

D

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

E

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

F

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

G

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

H

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

I

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

K

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

L

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

M

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

N

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

O

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

P

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Q

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

R

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

S

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

T

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

V

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

W

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Y

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.

Z

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/contextengineering.