J.C. KÖDEL

CONTEXT ENGINEERING

ENGINEERING INFORMATION FOR AI SYSTEMS

SPECIFICATIONS | ARCHITECTURE | CODE | ADRs | TESTS | DOCUMENTATION | REPOSITORY

Table of Contents

This is a free sample. The complete edition has every chapter.

About the author

I started programming in the nineties, writing software for video rental stores. In 1998 I built my first **e**nterprise **r**esource **p**lanning (ERP) system, in Visual Basic 6 with SQL Server. Real customers used the system for years, and it decayed in my hands for lack of method. That was the first expensive lesson of my career: the hard part is rarely writing code; it is sustaining what you wrote.

From 2002 on I worked on systems with no room for failure: international registries and access control at the Brazilian Federal Police, international internet banking and the implementation of Basel II. A framework I wrote back then is still in production at a large bank almost twenty years later. Later on, I worked on an artificial intelligence system that analyzed 10 million calls a month, in a customer service operation with more than 150,000 employees across 13 countries.

In 2017 I launched a product of my own, Meu Cronograma Capilar, an app that plans hair care routines. It has passed 10 million downloads, holds a 4.8 rating and has been in the top 10 in its category on the Play Store since 2018. I handle all of it alone: code, architecture, tests, operations and publishing. I mention the app because it proves something no job title proves: a full cycle, shipped and sustained for almost a decade, with no team to make up for a shortcut.

Today I build software with AI in production. What daily practice showed me is that the difference between a consistent result and an expensive guess is almost never in the model: it is in the information you hand it. That became this book, which closes a trilogy. *Spec Driven Development* (https://books.kodel.com.br/en/books/sdd/) teaches what to build, trading loose prompts for specifications. *FOCUS Architecture* (https://books.kodel.com.br/en/books/focus/) teaches where the business rule lives, so humans and AI know where to touch. Neither is a prerequisite: you can start here. This one teaches how to feed the AI the right information at the right moment. And this book was produced with the techniques it teaches, from draft to review, the least you should demand of anyone who writes about the subject.

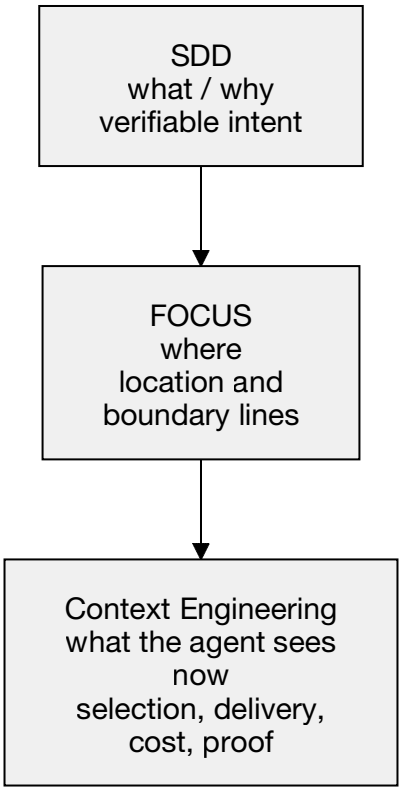
Every claim in the next pages is something I have seen work in production, or I say who I learned it from. Nothing here is armchair theory.

J.C.Ködel

Map of the trilogy

This is the third book of a trilogy, and you do not need to have read the other two: each one stands on its own, and this chapter zero exists so you know what lives in each volume when a bridge shows up in the text.

Spec Driven Development (https://books.kodel.com.br/en/books/sdd/) answers what and why: how to turn intent into a verifiable specification, so the work, yours or an AI's, has a target to check against. *FOCUS Architecture* (https://books.kodel.com.br/en/books/focus/) answers where: how to organize code into slices with their limits declared, so every change has an address. This book answers the question left over when the other two are standing: what the agent sees right now, in the window of this call, and at what cost.



The order of the arrows is the order of the information, not a required reading order: the spec says what to do, the architecture says where the doing happens, and the context carries both, in the right dose, to the model's window. When this book cites the earlier ones, the citation comes by name and with whatever is needed summarized on the spot, precisely so you do not have to depend on them in the middle of a chapter.

How LLMs use context

On Tuesday you asked the AI assistant for a spreadsheet import function and got back clean code, in your project's style, with the error cases handled. On Thursday you asked for practically the same thing and got a loose script, with generic names, ignoring the conventions the assistant itself had followed two days earlier. The request was the same. The model was the same. The result, the opposite.

The most common explanation you will hear is some variation of "that is just how AI is, a lottery." That explanation is comfortable and wrong. There is a concrete variable that changed between Tuesday and Thursday, and it has a name: the context. On Tuesday the conversation already held pieces of your code, the discussion about the error pattern and two of your own examples. On Thursday you opened a new session and sent the request cold. The model did not get worse. The information it received got worse.

This chapter sets the mental model that holds up the whole book: what exactly the model sees when it answers, and what it does not see at all. Without that, every technique in the parts ahead turns into a memorized recipe.

The model only sees the input

Start with the term that gives the book its name. Context is everything the model receives as input in one call: your question, the conversation history, the system instructions, the pasted files, the tool results. All of it, concatenated, forms a single block of text that the model reads at once. In Tuesday's example, the context was your request plus the code excerpts and the earlier discussion; on Thursday it was only the request.

Every time the model processes a context and produces an answer, an inference happens. An inference is one call to the model: text goes in, text comes out, and nothing else takes part. There is no side channel through which the model consults your repository, your intentions or yesterday's conversation. If the information is not in that call's input, for the model it does not exist.

That sounds obvious written this way, but almost nobody works as if it were true. When you complain that "the AI should know" your project uses a certain pattern, you are crediting the model with knowledge that never came through the only door there is: the input. The model should not know. You should have told it.

The practical consequence flips the usual question. Instead of "why did the model get it wrong?," ask "what was in the input that would justify the right answer?" In most of the frustrating sessions you have had, the true answer is: nothing. Thursday's cold request did not carry the project's error pattern, so the model picked some pattern. From its own point of view, Thursday's answer was as good as Tuesday's: coherent with the input it received.

Attention: how the model weighs what you sent

Inside one inference, the model does not treat the context as a uniform bag of words. The architecture behind today's **l**arge **l**anguage **m**odels (LLMs), described by Vaswani and coauthors in the 2017 paper "Attention Is All You Need" (arXiv:1706.03762), turns on a mechanism called attention. For the purposes of this book, the mental model is enough: when it generates each word of the answer, the model assigns weights to every piece of the input, deciding how much each piece influences the next word. Pieces with high weight pull the answer; pieces with low weight barely take part.

A minimal example. Suppose the input: "Our backend is in Go. Always answer with examples in the backend's language. How do I open a file?" When the answer is generated, attention links "the backend's language" to "Go," and the example comes out in Go. If the sentence about the backend were absent, the weight would spread to the rest and the model would pick the most likely language given everything else in the conversation, maybe Python. The answer changes without the final question having changed a single letter.

Two consequences of that mechanism matter to you. The first: everything in the context takes part in the contest for attention, including what you pasted without thinking. That 300-line log you dropped into the conversation to illustrate an error is still there: it still receives weights and still competes with your instructions at every word generated. The second: attention is a statistical mechanism, not an exact search. The model does not "find" your instruction the way grep finds a string; it weighs your instruction against all the rest. Instructions can lose the contest. Chapters 4 and 5 show when and why that happens more and more often.

If you want to go down to the real mechanism, with the matrices and the attention heads, the 2017 paper by Vaswani and coauthors (arXiv:1706.03762) is the primary source. To use AI well, the mental model above is enough, and this book does not go past it.

Nothing survives between calls

One piece is missing, and it is the one that knocks down the most expensive illusion: that the model remembers.

An LLM is stateless between calls: it keeps no state. Once an inference ends, the model retains nothing of what it processed. The public documentation of the chat **a**pplication **p**rogramming **i**nterfaces (APIs) of the major providers (the docs for Anthropic's Messages API and for OpenAI's API, in 2026) describes the same contract: every request sends the full list of messages in the conversation, and the server answers that list. There is no live session on the other side, no "brain" that follows you from one question to the next. There is a function: context in, answer out, done.

If it helps, picture a peculiar call center. Every time you call this company, whoever answers is a completely different person, with no access at all to what you dealt with on earlier calls: no customer database, no history, no "as we discussed yesterday." Everything that agent knows about your case is what you say on this call. In return, this is the best-prepared agent on the planet: every language, every framework, every pattern ever published. And it is exactly that breadth that creates the problem. Faced with a vague request, the agent has no way to know which of the thousand correct answers on hand is the right one for your case, so it picks the answer that is most likely in general, which is rarely yours. All the knowledge in the world, with no focus, produces a generic answer; the focus is what you bring, in what you say during the call. Every call to the model is that phone call: it starts over from zero, with someone on the line who knows everything and remembers nothing.

"But the chat does remember the conversation," you will say, "it answers my second question knowing about the first." It answers because the chat interface resends the whole conversation with every message you send. The memory you notice does not live in the model; it lives in the text the tool piles up and resends. It is a legitimate stage trick, and chapter 3 takes it apart in detail, with a real transcript of the point where it breaks.

For now, hold on to the contract: one call, one context, one answer, no residue. That contract explains the Thursday at the start of the chapter in full. The new session had no access to Tuesday's session, because there is no place where Tuesday could have been kept. You did not lose model quality from one day to the next; you lost the context, and the quality went with it.

Where the context hides

Before closing the mental model, a second illusion deserves to be undone: that the context is only what you type. In practice, the text you write tends to be the smallest part of what the model receives.

When you use a coding assistant, the tool assembles the input on its own before calling the model. It usually includes a system instruction (the text that defines the assistant's behavior), project configuration files you may not even remember exist, excerpts from the files open in the editor, results of searches the tool itself ran and the output of every command it executed. You type one line; the model receives tens of thousands of words.

Try it in your own tool: look for the option that shows the request it sends or how much the session has consumed. The first time you see the whole package tends to be uncomfortable, like opening the payload of a request you thought was lean and finding megabytes of extras. And each of those extras, you now know, competes for attention with your instruction.

That is not a flaw in the tools; it is their job. Assembling context automatically is what makes a coding assistant more useful than a bare chat. But it hands you a new responsibility: knowing what is being assembled on your behalf. If you have never looked at the real input, you have no way to diagnose why the output came out wrong. Throughout the book, "look at the context" will show up as the first step of almost every diagnosis, the same way "look at the log" is the first step of almost every production investigation.

What changes in your practice

Put the three pieces together. The model only sees the input. Inside the input, attention weighs each piece, and everything competes. Between calls, nothing persists. Out of those three sentences comes a working definition that the rest of the book only refines: the quality of the answer is a function of the quality of the information present in the context of that call.

Notice what that definition does to your room to maneuver. You do not control the model's weights, you do not control the training, and you do not control the architecture. You control one single thing: what goes into the context. That single thing determines, more than any other variable within your reach, whether you get Tuesday's code or Thursday's script.

Here is my own position: after years of using AI in production every day, I have not seen any adjustment of tool, model or phrasing return as much as treating the input with the same care I give a public interface. That practice is what this book systematizes.

One dimension is still missing, and this chapter treated it as abstract: the input has a size, and that size has a limit and a price. The model does not read "text" but tokens, and the context window that receives them is a finite resource. The next chapter defines those two measures, because without them you cannot reason about what fits, what costs and what stays out.

Tokens and context windows

You ask the agent to analyze your project. It reads file after file, runs searches and dumps command output, and the session moves along fine until, with no warning, the tool announces it is going to “compact the conversation” or simply starts answering while ignoring instructions you gave twenty minutes ago. Nobody showed you what piled up, how much fit or how heavy each read was. You are negotiating with an invisible limit, in a unit you cannot see.

The previous chapter established that the answer is a function of what is in the input. This chapter gives the two measures of that input: the unit it is counted in, the token, and the container that limits it, the context window. With both in hand you start to estimate what fits and to predict when it will overflow; chapters 5 and 6 turn the same measures into the cost of wasted room, in quality and money.

The model reads tokens, not words

A token is the smallest unit of text the model processes: a piece of a word, a whole short word, a punctuation mark or a space. Before the model processes anything, the context goes through a tokenizer, which slices it into those units and turns them into numbers. The model never sees letters; it sees the sequence of numbers the tokenizer produced.

The dominant slicing algorithm derives from **b**yte **p**air **e**ncoding (BPE), described for use in language models by Sennrich, Haddow and Birch in 2016 (arXiv:1508.07909). The principle can be stated in one sentence: character sequences that show up often in the training corpus become a single token; rare sequences get broken into smaller pieces. “The” is one token. An identifier like `calculateTotalWithDiscount` becomes several.

A minimal example, with the open-source tokenizer tiktoken, which OpenAI publishes on GitHub (github.com/openai/tiktoken): “the quick brown fox” comes to 4 tokens, one per word, because all four words are common. The same sentence in Portuguese, “a raposa marrom veloz,” comes to 7, because English dominates the training corpus of these tokenizers and words in other languages get sliced more often. Both numbers come from tiktoken 0.13.0, encoding `o200k_base`, measured in July 2026, and the older encoding `cl100k_base` returns the same pair. That asymmetry gives you the two rules of thumb you

will use every day: in English, one token is roughly 4 characters, or about three quarters of a word; the same content in another language costs more tokens, on the order of 20 to 40% more in Romance languages and well above that in Japanese, Chinese or any other non-Latin script. They are approximations, and the only exact measure is running your own tool's tokenizer, but to estimate orders of magnitude they are enough.

Now apply that yardstick to your own day. A page of running text lands in the hundreds of tokens. A 300-line code file, a few thousand. The output of that build command the agent ran and captured in full, tens of thousands. And here is what the agent era changed: you are no longer the one pasting text into the conversation; the agent is the one piling it up, one `Read` at a time, one search at a time, one log at a time, and all of it goes into the same count. You do not need precision; you need to stop treating those reads as weightless. Do the exercise once, to calibrate your instinct: take a file the agent read in your last session, count the characters and divide by four. Compare that number with the usage your tool reports for the session. After the first measurement, you will never again send an agent off to "read the whole project" without a second of hesitation, and that hesitation is exactly the habit this chapter wants to build. Every read has a size in tokens, and from the next chapter on that size takes center stage.

What tokenization explains as a bonus

Knowing that the model sees tokens, and not letters, undoes a few mysteries you have probably watched happen and chalked up to the model being dumb.

Before the example, a reminder that defuses some frustration: a **l**arge **l**anguage **m**odel (LLM) is not an intelligent entity in the sense the fluent conversation suggests. The whole mechanism is one thing: predicting the most likely text in the output given the text in the input. There is no understanding, no intention and no "somebody" on the other side who knows what they are saying; the intelligence you perceive is something you project onto it, an impression fluency creates. Whether that prediction amounts to some form of intelligence is an open debate among researchers, but what matters here is the mechanism. If you expect to be talking to something genuinely intelligent, you will expect the model to have abilities the mechanism simply does not have.

The classic case: you ask how many letter "r"s there are in a word and the model botches a count a child gets right. The error is only shocking because of the expectation above; you assumed intelligence where there is text prediction. It stays frustrating until you remember that the model never saw the letters. The word arrived as one or two tokens, whole numbers in a sequence, and what characters make up each token is not part of what the model

processes directly. Asking it to count letters is like asking you to count the bytes of an image by looking at the photo: the information exists at some level of the representation, but not at the level where you operate.

The same reasoning explains why swapping a word for a synonym sometimes changes the answer more than it should: different words slice into different tokens, with different statistical neighborhoods in the training data. And it explains why long code identifiers full of abbreviations use more tokens than clean names, because the tokenizer slices what it has never seen. None of those effects requires you to memorize the tokenizer's vocabulary. What they require is that you remember there is a slicing layer between your text and the model, and that this layer has a countable cost.

It is that countable cost that matters from here on. If every piece of text has a price in tokens, the next question is unavoidable: how many tokens fit in one call?

The context window is the container

The second measure is the limit. The context window is the largest number of tokens one call to the model holds, everything included: system instructions, conversation history, every file the agent read, every command output it captured and the answer the model is going to generate. The answer counts too, because the model generates token by token inside the same window it read the input in. And in the reasoning models that are the default at the major providers as of July 2026, the answer you read is not everything the model generated: before it come the thinking tokens, the internal draft the tool hides or summarizes, which takes up room in the window like any other generated token. A ten-line answer may have cost a few thousand tokens of draft, and a budget that ignores that invisible portion runs out of window sooner than the math predicted. When the total gets close to the ceiling, something has to give: the call fails, the answer comes out truncated, or the tool compacts or discards part of the history to make room, as in the scene that opened this chapter. Of the three, the third is the most treacherous, and chapter 3 shows the damage it does.

How big is the window? Here I refuse to print a table, on purpose. Window numbers age in months, and a book that pinned them down would be lying to you before its second printing. Take the order of magnitude, anchored in time: in 2026, the major providers' frontier models (Anthropic, OpenAI, Google), the largest each one offers, have windows between hundreds of thousands and a few million tokens, and the exact numbers are on each model's public page, one click away. By the time you read this paragraph, the values will have grown. The mechanics described here will not have.

Do the math that matters: a window of hundreds of thousands of tokens holds roughly a few hundred pages of text or a small code project in its entirety. That sounds like plenty. And that is where the trap is, the one that separates the people who have read this book from the people who have read the marketing page.

A big window is no license to fill it

The natural reaction to windows growing is "great, now I can send the agent to read the whole repository and let the model sort it out." That reaction assumes the window works like a disk, where taking up 10% or 90% amounts to the same thing as long as it fits. It does not.

Remember chapter 1: inside the window, every token competes for attention every time a word is generated. Filling the window changes that contest. Your instruction, which dominated the attention in a lean input, now competes with tens of thousands of tokens of log, dead code and old conversation. There is published research measuring how much quality drops as the context grows and where the drop is worst, and chapter 5 is entirely about it. For now, note the asymmetry: the window grew because it is an easy number to sell, but the capacity to hold tokens and the capacity to use those tokens well are different things, and the second did not keep up with the first.

There is also a part no model page advertises: every token in the window is billed. Providers price per million input tokens and per million output tokens, so a window full of garbage costs real money on every call, even when quality survives. Chapter 6 does that math with you.

Measure it yourself: what travels with a one-line question

You do not have to take abstract numbers on faith; the experiment fits in one prompt. While writing this chapter, I opened a fresh session of my coding agent, cleared the history and asked it for one thing only:

> Repeat back exactly what you received in this request. I want to see everything that is in the context besides my prompt.

Two sentences. A few dozen tokens. The answer listed what else was in the window on that call, and the list is long: the tool's system instruction, with rules about how it should behave and the complete state of the repository; the schemas of every tool the agent can call, the

machine-readable description of what each one takes, plus a list of another hundred tools available on demand; my global instruction file, which dragged in a whole Flutter style guide, useless for a project that was not Flutter; a behavior mode injected by a session hook, a script the tool runs on its own at startup; the descriptions of some thirty-five installed skills, packaged instruction sets the agent loads on demand; and instructions from three **M**odel **C**ontext **P**rotocol (MCP) servers. Added up, the material that traveled with my question measured tens of thousands of tokens, three orders of magnitude larger than the prompt itself.

None of that is a flaw in the tool; it is the price of a well-equipped agent. That is not the point, though: every subsequent call in the session reloads that baggage, and I had put a good part of it there myself and forgotten. Run the same prompt in your own tool before you read on. Knowing what your one-line question drags along with it is the first act of context engineering this book asks of you.

The yardstick you take from this chapter

Recall the session that opened the chapter. In that session, the agent read, say, a dozen files of a few hundred lines each, plus two build outputs. At a few thousand tokens per file and tens of thousands per log, the sum passes a hundred thousand tokens before you notice. The tool's system instruction, the conversation history and the room needed for the answers pushed the total against the ceiling of the window, and the tool started to discard history to survive and took your instructions with it. None of that was invisible; it was only unmeasured. Now you measure: you estimate the tokens of each piece, you know the sum competes for a finite window, and you know that filling the window has a double cost, in attention and money.

One piece is still missing before the mechanism closes. If every call is isolated, as chapter 1 showed, and every call carries at most one window of tokens, as this chapter measured, then why does it feel as though the chat remembers what you said ten messages ago? The answer is that the tools send everything again for you, and understanding that trick explains why long sessions forget what was agreed on. That is the next chapter.