Mastering Claude

The definitive guide for developers: agents, automatión, and AI architectures

Francisco Montes de Oca Molina

2026

Mastering Claude

The definitive guide for developers: agents, automation, and AI architectures

Francisco Montes de Oca Molina

2026

Índice

Table of Contents

PART III — ARCHITECTURES AND SYSTEMS

PART IV — CLAUDE CODE AND THE MODERN STACK

PART V — A REAL PROJECT FROM START TO FINISH

APPENDICES

Mastering Claude

The definitive guide for developers: agents, automation, and AI architectures

Francisco Montes de Oca Molina

Prologue

I've been building software for eighteen years. I've worked at startups that disappeared in six months and at companies that grew to millions of users. I've written code that ran in production for years and code that broke production in minutes. I've led teams and been the only developer in the room. And in all that time, the most valuable lesson wasn't any design pattern or framework: it was learning to have judgment about when to use what.

That judgment takes time to build. It's formed through costly mistakes, bugs that keep you up at night, architecture decisions you regret six months later. There's no shortcut for that. But there are tools that can reduce the friction along the way.

Artificial intelligence is one of them. And Claude, in particular, is the one I've used most in recent years for real technical work.

When I started using it seriously, my first reaction was resistance. The dominant narrative at the time was "AI is going to replace programmers," and I — like you, probably — didn't want to believe it or explore it too much for fear of validating something that seemed absurd to me. But curiosity won. I started using Claude for small tasks: explaining a strange error, proposing a structure for a new module, generating the skeleton of some tests. And what I discovered was something completely different from what the headlines promised.

AI didn't replace the programmer. It multiplied them.

That's not an exaggeration. There are tasks that used to take me forty-five minutes and now take twelve. Not because Claude does them all magically, but because it eliminates the friction of getting started, of finding the right pattern, of remembering the exact syntax of a library I use once a month. That small friction, repeated dozens of times a day, accumulates. And when you eliminate it, you recover something very valuable: mental energy.

That extra mental energy is what has changed my work the most. It's not just speed. It's clarity. When you're not spending attention writing boilerplate or searching for the fifth time how to do a reduce in TypeScript, you have more space to think about what really matters: design, architecture, edge cases, business implications. The decisions that genuinely require human judgment.

And that has a side effect I didn't expect: better work-life balance. When you finish your work faster and with less friction, the margin of time you recover doesn't have to go toward working more. It can go toward living better. Being with your family. Resting. Learning something that has nothing to do with code. For me, that's the most honest and least hyped benefit of using AI well.

This book comes from that experience. It's written for programmers who know their craft — who know TypeScript, who understand how a REST API works, who have suffered production bugs — but who haven't yet found a way to integrate Claude effectively into their daily work.

It's not written to admire the technology from afar. It's written to use it with feet firmly on the ground.

The promise of this book is simple: by the time you finish reading it, you'll know how to use Claude as a copilot, not a crutch. The difference matters. A copilot helps you do your job better while you remain the one deciding the direction. A crutch takes away your responsibility and makes you dependent. The first makes you a better programmer. The second makes you worse.

To achieve that, there are three patterns that will repeat throughout these chapters. First: the difference between a weak prompt and a strong prompt isn't magic, it's communication with context. You'll learn to build prompts that work because they have structure, not because they have magic keywords. Second: iteration is part of the process. Claude's first result is almost always a draft, and that's okay. The best results come after two or three well-directed exchanges. Third: you always verify. Claude can be wrong with great verbal fluency. The responsibility to review, test, and validate is always yours.

That final responsibility isn't a burden. It's what makes your work continue to be valuable. Claude can generate the code; you decide if it's correct, if it's secure, if it fits the architecture, if it solves the real problem. That final decision is irreplaceable.

You can read this book from start to finish or use it as a reference. The chapters are built to make sense in order, especially those in the first two parts. But if you already have basic experience with the API and want to go straight to agents or the production section, you can jump to Part III or IV without losing the thread.

What you shouldn't do is read it without applying it. Every chapter has working code. Copy it, break it, improve it. That's the only way this book will change how you program.

PART I — FUNDAMENTALS AND MINDSET

Chapter 1: Why Claude changes the programmer's work

There's a narrative that has been circulating for years in technical forums, conferences, and technology headlines: artificial intelligence is going to replace programmers. It's a claim that generates strong reactions. Some celebrate it as if it were inevitable. Others dismiss it as pure hype. And most developers hear it with an uncomfortable mix of skepticism and anxiety.

The narrative is false. But understanding why it's false matters almost as much as understanding what does change.

Why the replacement myth persists

The argument sounds logical at first glance: if a tool can write code, and programmers write code, then the tool can do the programmer's job. But that logic has a fundamental category error. It confuses writing code with making software.

Making software involves understanding a business problem that is often poorly defined. It involves making architecture decisions under real constraints of time, team, and budget. It involves anticipating how something will fail before it fails. It involves negotiating with a client who wants everything done yesterday. It involves maintaining a system that was written by four different people over three years, with inconsistent standards and no tests.

Claude can write a function. It can't replace the judgment needed to know which function to write, when not to write it, how to integrate it with the existing system without breaking anything, or how to explain to the team why this approach is better than that one.

The myth persists because the demos are impressive. Seeing a model generate a complete React component in ten seconds looks spectacular. But demos always show the happy case in a clean context. They don't show the

legacy code with twenty years of accumulated decisions. They don't show requirements that change three times in a week. They don't show the conversation with the business user who describes the problem one way and means another.

What does change: the friction points

What does change, and changes significantly, are the friction points that drain energy during a programmer's day.

Think about your last week of work. How much time did you spend writing boilerplate that you already knew how to do, but that still took time because it had to be done? How much time did you spend in the documentation looking up the exact syntax of something you use once a month? How much time did you spend explaining a bug to someone, or to yourself out loud, before finding the cause? How long did it take you to write the tests for a function whose behavior you already knew perfectly?

Those activities aren't the core of the programmer's craft. They're the friction surrounding the core. And Claude is exceptionally good at reducing that friction.

A task that used to take forty-five minutes — writing an endpoint with validation, error handling, and tests — can now take twelve. Not because Claude does everything, but because it generates eighty percent of the structural code while you review, adjust, and add what requires real judgment: the specific business rules, the edge cases that come from domain knowledge, the correct way to handle errors according to the API contract.

Those twelve minutes include your review. It's not that you generate and accept blindly. You read the code, understand it, test it. But the starting point is much further advanced.

The copilot model

The most useful metaphor for understanding the correct relationship with Claude is that of the aviation copilot. The copilot doesn't fly the plane. They assist the pilot, monitor systems, execute checklist procedures, communicate with air traffic control. But when a critical decision has to be made under difficult conditions, the pilot is in command.

You are the pilot. Claude is the copilot.

This distinction matters because it determines how you use the tool. A well-used copilot reduces your cognitive load on mechanical tasks so you can concentrate your attention on those that require judgment. A poorly used copilot — when you delegate decisions that the pilot should make — creates real risk.

The correct question isn't "can Claude do this?" but "should Claude be the one to decide this?" For writing the skeleton of a module: yes. For deciding how to handle authentication in your payment system: no, or at least not without careful review.

What Claude doesn't do well

Being honest about limitations isn't pessimism. It's what allows you to use the tool without unpleasant surprises.

Claude doesn't have genuine architectural judgment. It can propose options, compare trade-offs, and generate decision documentation, but it doesn't know your system, your team, your technical debt, or the implicit constraints that come from years of organizational context. An architecture proposal from Claude is a starting point for thinking, not a final recommendation.

Claude doesn't understand implicit business rules. It knows what you tell it. What you don't tell it — the exceptions that everyone on the team knows but are never written down, the special client who has different rules, the

historical reason why that module does something that seems odd — it doesn't know and can't guess well.

Claude doesn't validate security reliably. It can identify common vulnerability patterns, but it's not a security auditor. Code that Claude generates and that has no obvious errors can have subtle vulnerabilities that are only detected with expert review.

Claude isn't always right even when it sounds confident. This is the most dangerous characteristic: the model generates fluent, confident text even when it's wrong. A programmer who doesn't review Claude's code because it looks correct is taking a real risk.

The recovered time

When you eliminate two hours of repetitive work per week, the question that arises is: what do you do with that time?

The most obvious — and least interesting — answer is to work more. Add more features, close more tickets, take on more projects. That can have value, but it's not the most important benefit.

The most important benefit is recovering mental energy for what does require your best attention. Thinking through design more carefully. Reviewing your team's code more thoroughly. Dedicating time to understand the complete system instead of just the module you have to deliver this week. Learning something new without feeling like you're stealing time from urgent work.

And outside of work: being present when you're not working. The programmer who comes home with a mind saturated with things to do doesn't rest well. AI doesn't eliminate that problem, but it reduces the accumulation of small tasks that contribute to it.

The programmer's identity

There's something more I want to name directly, because I hear it frequently in conversations with other developers.

Using AI to generate code doesn't make you less of a programmer. The value of an experienced developer isn't in their typing speed or the number of library APIs they have memorized. It's in their judgment. In their ability to design systems that scale, to anticipate failures, to make technical decisions with incomplete information, to communicate those decisions to people who aren't technical.

All of that remains yours when you use Claude. In fact, when friction decreases, those capabilities are expressed more. You have more time to design well because you spend less time writing what you already know you need to write.

❌ **Weak prompt:**

```
Write me a function for users.
```

✅ **Strong prompt:**

```
Write a TypeScript function called groupUsersByRole with this
signature:
  function groupUsersByRole(users: Array<{ id: string; name: string;
role: 'admin' | 'editor' | 'viewer' }>): Record<string, string[]>

Requirements:
- No any
- Handles empty array by returning {}
- The value of each key is an array of names (not ids)
- Include a usage example in a comment
- Explain the time complexity in one line
```

The difference isn't in the length. It's in the specificity: input type, output type, explicit constraints, behavior in edge cases. Claude doesn't need to guess anything. The result is directly usable.

Mini summary: Claude changes the programmer's work by reducing friction in mechanical tasks, not by replacing technical judgment. The copilot model — you decide, Claude assists — is the correct way to think about the relationship. Time recovered, well used, translates into better design, higher quality, and better work-life balance.

Chapter 2: How Claude thinks — models, tokens, and context

To use Claude well, you don't need to understand the mathematical details of how a transformer works. But you do need to understand some concepts that directly affect how you design your interactions, how much they cost, and what you can expect from the results.

What a language model is, in practical terms

A large language model is, in essence, a system trained to predict text. During training, it processed massive amounts of text — code, documentation, books, conversations — and learned the statistical patterns of how words and concepts relate to each other.

When you ask it a question, the model doesn't look up the answer in a database. It generates text token by token, choosing at each step the most probable continuation given your instructions, the conversation history, and the patterns learned during training. It's text prediction at massive scale, with an emergent reasoning capability that nobody explicitly designed but that appears when the model is large enough and the training extensive enough.

This explains something important: Claude can give plausible but incorrect answers. The model doesn't verify its statements against a source of truth. It generates what is statistically coherent with the context. When the context you give it is clear and complete, the answers are good. When the context is ambiguous, the model fills gaps with what seems most probable, which may not match the reality of your project.

The practical implication: you always verify. Not because Claude is bad, but because that's how it works.

Tokens: the model's currency

Before understanding context, you need to understand tokens, because they're the basic unit the model works with.

A token isn't exactly a word. It's a fragment of text that can be a complete word, part of a word, a punctuation mark, or a space. In English and in code, the approximate ratio is 3-4 characters per token. In Spanish and other languages with more morphology, it tends to be a bit more.

As a practical reference: - A 20-line TypeScript function: approximately 120-150 tokens - A 200-line TypeScript file: approximately 1,200-1,500 tokens - This complete chapter: approximately 2,500-3,000 tokens - A medium repository with 50 files: can easily exceed 100,000 tokens

Tokens matter for two reasons. First, cost: models charge per input and output tokens. Second, the context limit: each model has a maximum number of tokens it can process in a single call.

```
import Anthropic from '@anthropic-ai/sdk';

const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY
});

const response = await client.messages.create({
  model: 'claude-sonnet-4-5',
  max_tokens: 1024,
  messages: [{ role: 'user', content: 'Explain this error: Cannot
read property of undefined' }],
});

console.log(response.content[0].text);
console.log(`Tokens used: ${response.usage.input_tokens} input,
${response.usage.output_tokens} output`);
```

The `response.usage` field is your best ally for understanding the real cost of each call. Save those numbers when you're designing systems that will make many calls.

The context and its limits

The context is everything the model can "see" in a call: the system prompt, the message history, the text you attach. Claude 3.5 Sonnet and later versions have context windows of 200,000 tokens, which is equivalent to several books or a medium-sized complete repository.

But the size of the available context doesn't mean you should fill it all. There are two problems with excessive context.

First, cost. You pay for every input token, including the context. A 10,000-token system prompt that you send in every call can represent a significant fraction of the total cost if you make many calls.

Second, quality. This is less obvious: irrelevant context doesn't just take up space, it actively damages the quality of responses. The model pays attention to everything in the context. If you fill the context with information that isn't relevant to the current task, the model devotes attention to that information and can be distracted from what matters.

The practical rule: include everything that's relevant, nothing that isn't.

When a file or repository is too large for the context, you have three strategies: - **Summarize**: extract the relevant parts of the code or documentation and pass them instead of the complete file - **Fragment**: divide the work into smaller parts, each with the specific context it needs - **Retrieve**: use RAG (Chapter 13) to find and pass only the relevant fragments

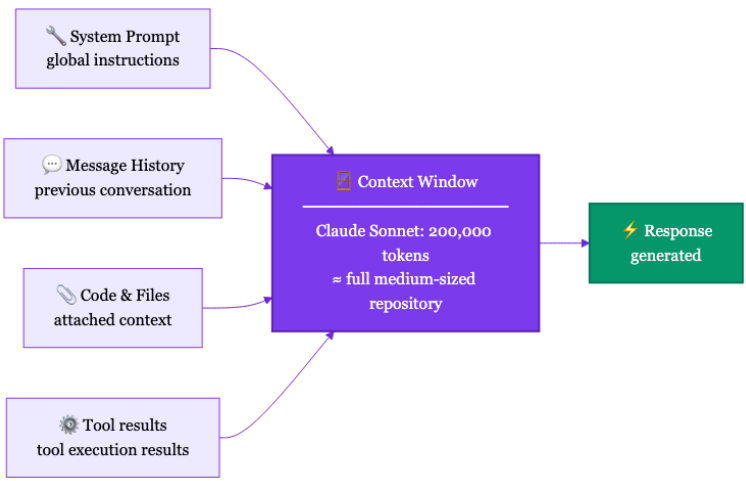


Figure 1. Everything that enters the context window in a model call.

The model families

Anthropic offers different models with different capabilities and costs. Understanding when to use each one is an important part of working effectively with Claude.

Claude Haiku is the fastest and most economical model. It responds in fractions of a second and its cost is significantly lower than the larger models. It's ideal for classification tasks, routing, short summaries, simple rewrites, and anything where you don't need complex reasoning. If you're building a system that makes many simple calls — classifying whether an error is critical or not, deciding which agent to send a task to, generating a user-friendly error message from a technical exception — Haiku is the right choice.

Claude Sonnet is the general work model. It balances reasoning capability, speed, and cost well. For most programming tasks — generating code, debugging, refactoring, writing tests, explaining systems — Sonnet is the correct starting point. When in this book we say "Claude" without specifying a model, we mean Sonnet.

Claude Opus is the most capable model. It has the greatest reasoning depth and is the best for complex problems: architecture decisions with many variables, debugging distributed systems with emergent behavior, trade-off analysis in contexts with multiple constraints. It's also the most expensive and the slowest. Use it when the problem genuinely requires it, not by default.

The selection rule is simple: start with Sonnet. If the results are consistently good, stay there. If you need more reasoning depth for specific tasks, try Opus for those tasks. If you have high-frequency, low-complexity tasks, move them to Haiku.

Non-determinism and its implications

The same prompt sent twice doesn't produce the same result. This isn't a bug — it's a fundamental characteristic of the design. The model uses probabilistic sampling, not deterministic search.

For code, this has practical implications. You can't assume that because Claude generated a specific solution yesterday, it will generate the same one today. Tests that validate Claude's output have to evaluate behavior, not exact text.

Temperature is the parameter that controls how much randomness the sampling has. A temperature of 0 makes the model more deterministic (always chooses the most probable option). A high temperature makes responses more varied and creative. For code, the default temperature is generally correct. You don't need to adjust it except in specific cases where you want to explore variations or where you need more predictability.

The quality of context

A principle that's worth more than any specific technique: quality context produces quality results.

Quality context has three characteristics. It's relevant: it includes what the model needs to do the task, no more. It's complete: it doesn't omit constraints, conventions, expected behaviors that the model needs to know in order to give a useful response. It's accurate: it correctly describes what exists, not what you think exists or what you'd like to exist.

Poor context has the opposite effect. If you give Claude a stack trace without the relevant code, it has to guess the cause. If you give it the code without describing the expected behavior, it has to assume what you want it to do. If you give it vague constraints ("make it scalable"), it can't fulfill them because they have no operational meaning.

The investment in building good context at the beginning of a session always pays off in the quality of the results.

Mini summary: Claude is a text prediction system trained on massive data, not a search engine or an oracle. Tokens determine cost and limits. Quality context produces quality results — irrelevant context isn't neutral, it actively damages. Sonnet is the default working model; Haiku for volume and simplicity, Opus for complex reasoning.

Chapter 3: Setting up your working environment with Claude

There are three distinct ways to work with Claude, and each has its place. Confusing them — or always using the same one for everything — is one of the most common mistakes for those starting out.

The three access paths

Claude.ai (web interface) is the fastest way to get started. You open the browser, type, receive a response. It's excellent for exploration, for asking questions that don't require project context, for experimenting with prompts, for long conversations where you want to see the complete thread. Its limitation is that it doesn't have access to your codebase and each session starts from scratch.

When to use claude.ai: conceptual questions ("what's the difference between Promise.all and Promise.allSettled?"), exploring approaches before committing to one, learning new technologies, high-level design conversations.

The API directly is for when you're building something. If you want to integrate Claude into an application, automation, script, or agent, you use the API. You have complete control over the model, the context, the parameters, and how you process the results. The cost is per use, without a subscription.

When to use the API: building internal tools, automations in CI/CD, scripts that process code or documentation, agent systems.

Claude Code (CLI) is for daily work inside your codebase. It's a command-line client that understands the structure of your repository, can read and edit files, execute commands, and maintain context between operations. It's not a chat — it's a collaborator that lives in your terminal.

When to use Claude Code: refactorings that affect multiple files, debugging in the real project context, writing tests for existing modules, code review with access to the complete repository.

API configuration

The process is straightforward:

```
# Install the SDK
pnpm add @anthropic-ai/sdk

# Or with npm
npm install @anthropic-ai/sdk
```

```
// src/lib/claude.ts
import Anthropic from '@anthropic-ai/sdk';

// The API key comes from an environment variable, never hardcoded
const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY
});

export default client;
```

```
// First complete call
import client from './lib/claude';

async function firstCall(): Promise<void> {
  const response = await client.messages.create({
    model: 'claude-sonnet-4-5',
    max_tokens: 1024,
    messages: [
      { role: 'user', content: 'What is the difference between
interface and type in TypeScript?' },
    ],
  });

  const textBlock = response.content.find(b => b.type === 'text');
  if (textBlock && textBlock.type === 'text') {
    console.log(textBlock.text);
  }
}

firstCall().catch(console.error);
```

For environment variables, the standard flow is:

```
# .env (in .gitignore, never in the repository)
ANTHROPIC_API_KEY=sk-ant-...

# Load it with dotenv in development
pnpm add -D dotenv
```

```
// At the start of the process, before any SDK imports
import 'dotenv/config';
```

The API key doesn't go in the code. Never. If you accidentally commit it, rotate it immediately at console.anthropic.com.

Claude Code installation

```
# Global installation
npm install -g @anthropic-ai/claude-code

# Authentication
claude auth login

# Verify it works
claude --version
```

Once installed, you navigate to your project directory and simply type `claude`. An interactive session opens where you can make requests in natural language and Claude Code has access to your repository's files.

```
cd my-project
claude

# Inside the session:
# > Explain the structure of this project
# > There's a failing test in src/users/users.test.ts, help me
diagnose it
# > Refactor the authentication module to use zod for input
validation
```

The CLAUDE.md file

This is probably the most undervalued concept for those starting with Claude Code. A `CLAUDE.md` file in the project root gives Claude persistent context about your project — context you don't have to repeat in every session.

Without CLAUDE.md, every session starts from scratch and you have to explain your stack, your conventions, and your constraints. With CLAUDE.md, Claude arrives at each session already knowing how your project works.

````
# CLAUDE.md — My API Project

## Technical stack
- Node.js 20 + TypeScript 5.x in strict mode
- Express for HTTP
- Prisma with PostgreSQL
- Vitest for tests
- Zod for input validation

## Main commands
```bash
pnpm dev          # server at localhost:3000 with hot reload
pnpm test         # vitest run (no watch)
pnpm test:watch   # vitest in watch mode
pnpm lint         # eslint . --ext .ts
pnpm build        # tsc --noEmit && esbuild src/index.ts
````
````

Important conventions

- TypeScript strict: no any, no as where not necessary
- Validation with zod on ALL endpoints before processing
- Tests in *.test.ts files alongside the file they test
- HTTP errors with format: { error: string, code: string }
- Never modify the Prisma schema without discussing first

Project structure

```
src/
  routes/      → HTTP handlers (Express Router)
  services/    → pure business logic
  lib/         → utilities, external clients
  types/       → shared types
  middleware/  → Express middleware
prisma/
  schema.prisma
```

What NOT to do

- Do not use callbacks where there is an async/await equivalent
- Do not leave console.log in code that isn't temporary debugging
- Do not create files in the project root without asking

This file lives in the repository and is shared with the team. When someone new joins the project and uses Claude Code, they already have the project context without having to configure anything.

The session start ritual

A productive session with Claude has a structure. It's not mandatory, but it produces more consistent results:

First, project context — if you don't have CLAUDE.md or need to add specific context for the session:

```
Context: REST API in TypeScript with Express and Prisma. Today I'll
be working on the authentication module. The system uses JWT with
refresh tokens, implemented in src/auth/.
```

Second, the concrete session objective:

```
Objective: the POST /auth/refresh endpoint is failing when the
refresh token is expired. It returns 500 instead of 401. I need to
diagnose and fix it.
```

Third, the applicable constraints:

```
Constraints: don't change the error response format. Don't modify the
signature of existing endpoints. Maintain TypeScript strict.
```

With that start, Claude's first response already goes in the right direction.

Mini summary: There are three ways to use Claude with different purposes. The API is for building, Claude Code is for working inside the repository, claude.ai is for exploration. The CLAUDE.md file is the most cost-effective time investment for daily work with Claude Code — set it up from day one.

Chapter 4: The art of prompting for programmers

Prompting isn't a mystical art. There are no magic words or hidden tricks. It's communication with specific constraints: a sender (you) who has context that the receiver (Claude) doesn't have, and who needs to transfer that context efficiently to get a useful result.

If you think of it that way, the principles of a good prompt are the same as those of a good technical specification. Clarity, completeness, explicit constraints, defined success criteria.

The four-part structure

Almost every effective prompt for programming has four components:

1. Context: what exists, what system, what stack, what's relevant about the current state. The context tells Claude what world it's operating in.

2. Objective: what you want it to produce. Concrete, measurable, unambiguous. Not "make it better" but "refactor this function to eliminate the duplication of validation logic, keeping the external behavior identical."

3. Constraints: what it can't do, what it can't change, what conventions it must follow, what tools it can use. Constraints are just as important as the objective because they eliminate the space of incorrect solutions.

4. Output format: how you want it to respond. Just the code? Code with explanation? A table of options? Only the changes, not the complete file? When you don't specify, Claude chooses, and it doesn't always choose what's most useful for you.

Let's see how this is built incrementally:

```
# Just the objective (weak)
Write a function to paginate results.

# Objective + context
I'm building a REST API with Express and TypeScript.
Write a function to paginate results from database queries.

# Objective + context + constraints
I'm building a REST API with Express and TypeScript strict.
Write a function `paginate<T>` that takes an array of items and
pagination parameters,
and returns { data: T[], total: number, page: number, pageSize:
number, hasNext: boolean }.
No any. Handles cases of page <= 0 or pageSize <= 0 by throwing an
error with a descriptive message.

# Complete: objective + context + constraints + output format
I'm building a REST API with Express and TypeScript strict.
Write a function `paginate<T>` that takes an array of items and
pagination parameters,
and returns { data: T[], total: number, page: number, pageSize:
number, hasNext: boolean }.
No any. Handles page <= 0 or pageSize <= 0 by throwing a descriptive
error.
Give me: 1) the function with complete types, 2) a usage example, 3)
two test cases with Vitest.
```

Each addition eliminates ambiguity and makes it more likely that the result is directly useful.

System prompts vs user messages

When you use the API directly, you have the option to use a system prompt. The distinction matters:

The **system prompt** establishes the global context and instructions that apply to the entire conversation. Here go: Claude's role in this system, the project conventions, permanent constraints, the expected response format.

User messages contain the specific tasks within that context.

```
const response = await client.messages.create({
  model: 'claude-sonnet-4-5',
  max_tokens: 2048,
  system: `You are a technical assistant for a TypeScript project
with Express and Prisma.
Project conventions:
- TypeScript strict, no any
- Input validation with zod
- Tests with Vitest
- HTTP errors: { error: string, code: string }
Always respond with commented code and concise explanations.`,
  messages: [
    {
      role: 'user',
      content: 'Write a JWT authentication middleware that validates
the Authorization header.',
    },
  ],
});
```

If you're always going to work on the same project, putting the project context in the system prompt is more efficient than repeating it in each message.

Four pairs of weak vs strong prompts

Pair 1: Function request

❌ Weak:

```
Write a function to validate emails.
```

✅ Strong:

```
Write a TypeScript function `validateEmail(email: unknown): email is
string` that:
- Returns true only if the argument is a string with valid email
format (basic RFC 5322)
- Uses a reasonable regex, not an external library
- Correctly handles: empty strings, null, undefined, numbers, objects
Include 5 test cases that cover edge cases.
```

Pair 2: Bug report

❌ Weak:

```
My function doesn't work, what's wrong?

function processOrder(order) {
  return order.items.reduce((sum, item) => sum + item.price, 0);
}
```

✅ Strong:

```
This function fails with TypeError when order.items is empty or when
an item has no price.

function processOrder(order: { items: Array<{ name: string; price?:
number }> }): number {
  return order.items.reduce((sum, item) => sum + item.price, 0);
}

Error: TypeError: Cannot read properties of undefined (reading
'price')
Expected behavior: return 0 if items is empty, ignore items without
price.
Constraint: maintain the same signature, no any.

Identify the root cause and propose the minimal fix.
```

Pair 3: Test request

❌ Weak:

```
Write tests for this function.
```

✅ Strong:

```
Write tests with Vitest for this `groupUsersByRole` function.
I want to cover: normal case with multiple roles, empty array, single
user per role,
user with a non-standard role (TypeScript wouldn't allow it but you
might receive it at runtime).
Format: describe/it, no mocks (the function is pure), descriptive
test names in English.
```

Pair 4: Code explanation

❌ Weak:

```
Explain this code.
```

✅ Strong:

```
Explain this function to someone who knows TypeScript well but just
joined the project.
Focus on: what problem it solves, why it uses this approach instead
of a simpler one,
what side effects I need to consider if I want to modify it,
and what invariants it assumes are true at runtime.
```

Few-shot: examples in the prompt

When you want Claude to follow a specific pattern that's hard to describe in words, the most efficient way is to show examples:

```
Convert each of these technical error messages to a user-friendly
message.
Follow exactly this format: [Technical error] → [Message for user]

Examples:
ECONNREFUSED → We couldn't connect to the server. Try again in a
moment.
ValidationError: email is required → Please enter your email address.
JWT expired → Your session has expired. Sign in again to continue.

Now convert these:
- ETIMEDOUT
- Error: User not found
- Cannot read property 'id' of null
```

Examples eliminate ambiguity about format and tone better than any verbal description.

Asking for reasoning

For complex tasks, asking Claude to explain its reasoning before giving the result produces better results. It's not magic — it's that the model also generates better code when it "thinks out loud" about the problem before writing the solution.

```
Before writing the code, briefly explain:
1) what the complex parts of this problem are
2) what approach you're going to use and why
3) what edge cases need special handling

Then write the implementation.
```

Iteration is the process

A common mistake: believing that a good prompt should produce the perfect result on the first try. That's not the case. Iteration is part of the normal flow, not a sign that something went wrong.

The first prompt establishes the direction. The second refines. The third adjusts details. That three-step process is faster and produces better results than trying to write the perfect prompt upfront.

```
# Iteration 1: direction
Write a React hook to handle login form state.

# Iteration 2: refinement
Add validation with zod. The schema is: { email: email string,
password: min 8 characters }.
Handle validation errors by field.

# Iteration 3: adjustment
Change the error state to be string | null instead of string |
undefined.
And add an `isSubmitting` state that is true while the form is being
processed.
```

Each step is short and clear. You don’t rewrite from scratch — you refine from where you are.

Mini summary: An effective prompt has four parts: context, objective, constraints, and output format. Constraints are just as important as the objective. The three-step iteration is more efficient than seeking the perfect prompt. Explicit reasoning improves code quality in complex tasks.

Chapter 5: Advanced prompting techniques

Once you have the basic structure of a good prompt, there's a set of techniques that produce significant improvements in specific situations. Not all of them apply at all times — part of the work is knowing when to use each one.

Chain of thought: thinking before writing

Chain of thought is asking Claude to reason about the problem step by step before arriving at the answer. For simple tasks it makes no difference. For complex tasks — non-trivial algorithms, debugging systems with multiple pieces, architecture decisions — the difference can be considerable.

The intuition is similar to what happens with a human: a programmer who stops to think through the problem before writing code almost always produces a better solution than one who starts typing immediately.

❌ Without chain of thought:

```
Design a function that processes concurrent orders without race
conditions.
```

✅ With chain of thought:

```
I'm going to ask you to design a function that processes concurrent
orders.
Before writing the code:
1. Identify the possible race conditions in a multi-worker scenario
2. Evaluate three synchronization approaches and their trade-offs for
this case
3. Choose the most appropriate one and explain why

Then write the TypeScript implementation.
```

The result with chain of thought tends to be more grounded and have fewer subtle errors because the model has “considered” the problems before generating the solution.

Role prompting: perspective context

Defining a specific role for Claude isn’t a verbal magic trick. It works because the role acts as context that guides the type of response. A “senior TypeScript engineer doing code review” will focus on different things than a generic “TypeScript developer.”

```
Act as a security engineer reviewing an authentication PR.
Your focus: vulnerabilities, secrets handling, input validation,
attack surface, authorization issues.
Don't worry about style or readability.

[PR code]
```

The most useful roles for programming work: - Senior engineer of [specific technology] for focused code review - Systems architect for design decisions - Security reviewer for vulnerability analysis - Function author to defend

design decisions

The role should be specific and make sense for the task. “Act as an expert” with no further detail adds nothing useful.

Structured output: forcing JSON format

When Claude is part of a system — not just a conversation tool — you need its output to be reliably parseable. The most direct way is to explicitly ask for JSON.

```
import Anthropic from '@anthropic-ai/sdk';

const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY
});

interface CodeAnalysis {
  bugs: string[];
  improvements: string[];
  riskLevel: 'low' | 'medium' | 'high';
  testCoverage: { missing: string[]; recommended: string[] };
}

async function analyzeCode(code: string): Promise<CodeAnalysis> {
  const response = await client.messages.create({
    model: 'claude-sonnet-4-5',
    max_tokens: 1024,
    messages: [{
      role: 'user',
      content: `Analyze this TypeScript code and respond ONLY with
valid JSON, no additional text.
The JSON must follow exactly this structure:
{
  "bugs": ["bug description 1", "..."],
  "improvements": ["suggested improvement 1", "..."],
  "riskLevel": "low" | "medium" | "high",
  "testCoverage": {
    "missing": ["uncovered case 1", "..."],
    "recommended": ["recommended test 1", "..."]
  }
}

Code to analyze:
```

For production systems, add zod validation after the parse:

```
import { z } from 'zod';

const CodeAnalysisSchema = z.object({
  bugs: z.array(z.string()),
  improvements: z.array(z.string()),
  riskLevel: z.enum(['low', 'medium', 'high']),
  testCoverage: z.object({
    missing: z.array(z.string()),
    recommended: z.array(z.string()),
  }),
});

// After JSON.parse:
const validated = CodeAnalysisSchema.parse(JSON.parse(cleaned));
```

Multi-turn conversations: the messages array

Conversations with accumulated context are built by adding messages to the `messages` array. The model sees the entire history and responds in that context.

```
const messages: Anthropic.MessageParam[] = [];

async function chat(userMessage: string): Promise<string> {
  messages.push({ role: 'user', content: userMessage });

  const response = await client.messages.create({
    model: 'claude-sonnet-4-5',
    max_tokens: 2048,
    messages,
  });

  const assistantMessage = response.content.find(b => b.type ===
'text');
  if (!assistantMessage || assistantMessage.type !== 'text') {
    throw new Error('Unexpected response');
  }

  // Add the response to the history for the next turn
  messages.push({ role: 'assistant', content: assistantMessage.text
});

  return assistantMessage.text;
}

// Usage:
await chat('I have a function that groups users by role. How would I
make it efficient?');
await chat('Now add handling for users with multiple roles.');
await chat('What would the correct return type be in that case?');
```

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

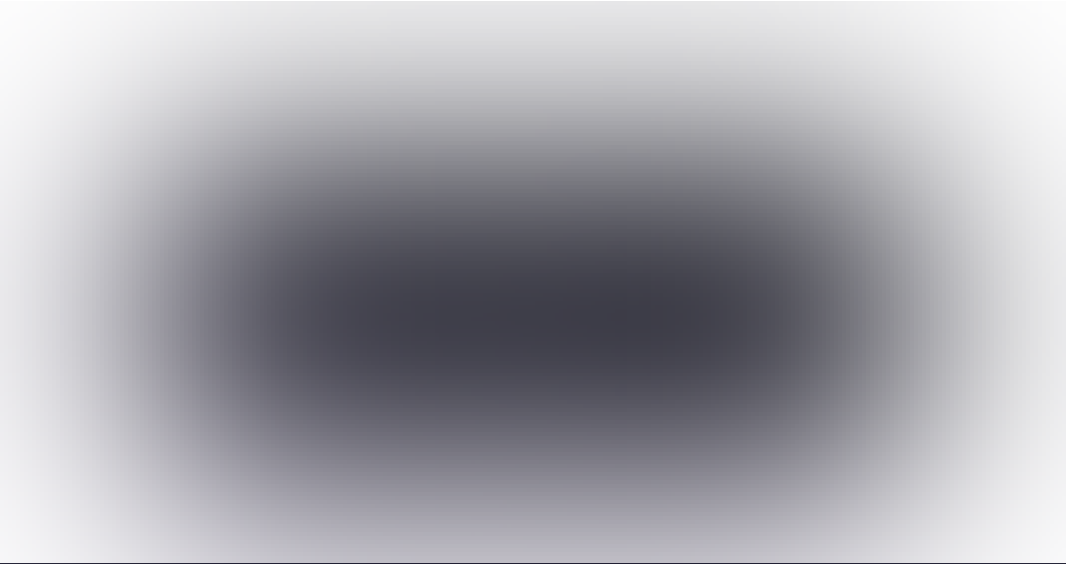
? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com



? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

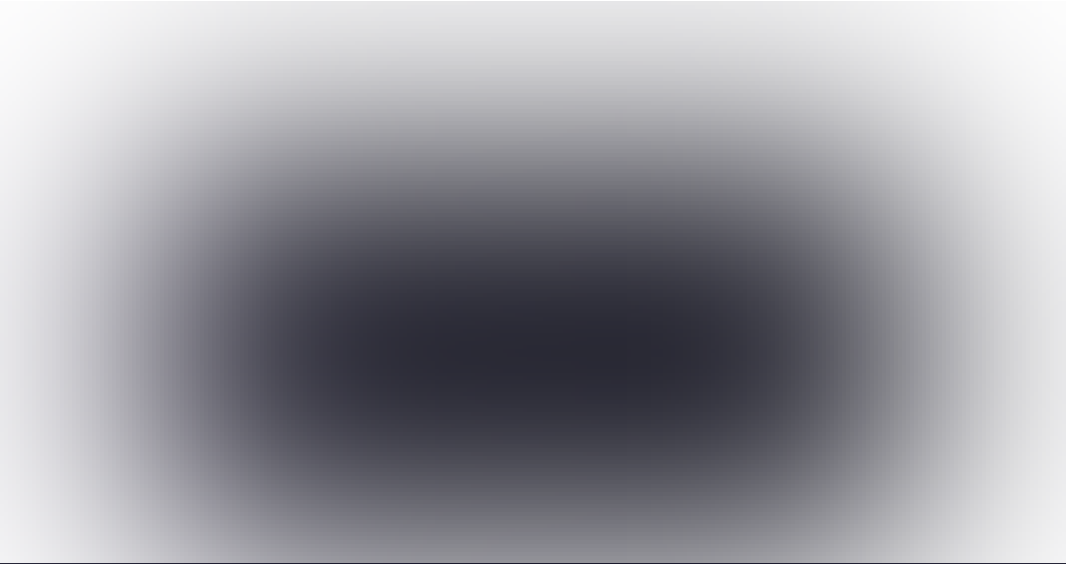
Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com



? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com

? Content locked

Get the full book at masterclaudebook.com