Member-only story

CLAUDE.md: Stop Explaining the Same Stuff to Claude Code

8 min read · Dec 6, 2025

P Pankaj Follow

Listen Share More

> *If you're using Claude Code and still re-explaining your stack in every session, you're leaving a lot of speed on the table.*

Pair-programming with an AI co-pilot, powered by a single CLAUDE.md file.

You know the feeling:

> "We use TypeScript."
> "Tests run with `npm test`."
> "Follow our auth patterns from `src/auth`."

...on repeat. Every. Single. Time.

There's a simple way out of this. It's called `CLAUDE.md`.

If you're unable to view the full post and wish to read it, please use: "Read Full Post"

What CLAUDE.md Actually Is

`CLAUDE.md` is just a Markdown file that Claude Code reads automatically when you start working in a repo. Think of it as:

> **A tiny README just for Claude:** how your project works, how you like things done and what really matters.

Set it up once and Claude already knows your project before you type the first instruction.

Where to put it:

```
./CLAUDE.md              # Project root (works)
./.claude/CLAUDE.md      # Inside .claude folder (cleaner, my preference)
```

That's it. No config files, no plugin circus. Just one Markdown file.

Before vs After CLAUDE.md

Before CLAUDE.md

```
You: "Add user authentication."
Claude: "Sure! What framework?"
You: "FastAPI with Python."
Claude: "Cool, JWT or sessions?"
```

```
You: "JWT."
Claude: "What database?"
You: "PostgreSQL."
Claude: "Testing framework?"
You: "pytest."
...10 more questions later...
```

You're basically onboarding Claude from scratch every session.

After CLAUDE.md

```
You: "Add user authentication."
Claude: "Adding JWT auth to our FastAPI setup using the patterns from src/auth.
```

No interrogation, no back-and-forth. Claude already knows:

- Your framework
- Your database
- Your auth style
- Your testing setup

...and just gets to work.

A Real Example: E-commerce API CLAUDE.md

Here's a concrete `CLAUDE.md` for an e-commerce backend:

```
#E-commerce backend. FastAPI + PostgreSQL. Handles ~100K
requests/day.

## Commands

- `uvicorn app:app --reload`    # Start dev server
- `pytest`                      # Run tests
- `alembic upgrade head`        # Run migrations

## Tech Stack

- Python 3.12
- FastAPI with Pydantic v2
- PostgreSQL 16
```

```
- Redis for cache
- Stripe for payments

## Code Style

- Type hints on everything
- Max function length: 50 lines
- One file per model in `models/`
- Pydantic models for all API inputs/outputs

### Good example

    @router.post("/orders")
    async def create_order(order: OrderCreate, db: Session =
Depends(get_db)):
        """Create new order."""
        try:
            result = await
db.execute(insert(Order).values(**order.dict()))
            return {"id": result.inserted_primary_key[0]}
        except Exception as e:
            logger.error(f"Order creation failed: {e}")
            raise HTTPException(500, "Order creation failed")

## Critical Rules

- NEVER commit .env file
- ALWAYS validate inputs with Pydantic
- RUN tests before pushing
- USE async/await for all DB operations

## Current Focus

Migrating to Pydantic v2. When touching models:

- Update `Field()` syntax
- Check migration guide: `PYDANTIC_V2.md`

## Common Issues

- Redis: Local dev uses port 6380 (not 6379)
- Stripe: Test mode only, keys in `.env.example`
- DB: Run migrations after pulling `main`
```

Claude now understands:

- This repo handles *real traffic* (100K req/day → performance matters)
- The exact commands it can run
- Your coding style and error-handling vibe
- The current migration work it should be careful about

Why This Works So Well

Let's unpack what makes that file effective.

Project identity right at the top

```
# ShopFast API
E-commerce backend. FastAPI + PostgreSQL. Handles ~100K requests/day.
```

Two lines, huge impact:

- What this repo is
- What stack it uses
- That performance isn't a toy concern

Claude can now suggest solutions that won't fall over in production.

Concrete, runnable commands

```
## Commands

- `uvicorn app:app --reload`    # Start dev server
- `pytest`                      # Run tests
- `alembic upgrade head`        # Run migrations
```

No guessing. Claude can:

- Start your dev server
- Run your tests
- Apply migrations

...without asking, *"How do you usually run this?"*

Show examples, don't just describe style

Instead of saying "We use proper error handling", you show it:

```
### Good example

@router.post("/orders")
async def create_order(order: OrderCreate, db: Session = Depends(get_db)):
    """Create new order."""
    try:
        result = await db.execute(insert(Order).values(**order.dict()))
        return {"id": result.inserted_primary_key[0]}
    except Exception as e:
        logger.error(f"Order creation failed: {e}")
        raise HTTPException(500, "Order creation failed")
```

Claude will pattern-match on this:

- Try/except around DB work
- Logging failures
- Raising an HTTP error with a clean message

You're teaching style by *example*, not by essay.

Make critical rules loud and obvious

```
## Critical Rules

- NEVER commit .env file
- ALWAYS validate inputs with Pydantic
- RUN tests before pushing
```

Caps + short lines = "Do. Not. Ignore."

Claude is much more likely to respect rules that are:

- Short
- Aggressive
- Isolated under a clear heading

Use "Current Focus" to steer work

```
## Current Focus

Migrating to Pydantic v2...
```

This section keeps Claude aligned with what matters **this week**, not six months ago.

When you're:

- Migrating frameworks
- Refactoring a critical module
- Chasing a performance regression

...put it here. Claude will prioritise changes in that direction.

How to Write a CLAUDE.md That Claude Actually Follows

Here are the patterns that consistently work well.

Keep it short(ish)

Don't turn `CLAUDE.md` into an encyclopedia.

- Aim for **100–300 lines** for a repo
- Split details into per-folder files if needed, e.g.:

```
@docs/api-conventions.md @docs/database-patterns.md
```

Claude gets a crisp *constitution* plus deeper docs *only when relevant*.

Be specific, not vague

❌ Vague rules:

```
- Write clean code
- Follow best practices
```

This doesn't tell Claude anything concrete.

✅ Specific rules:

```
- Max function length: 50 lines
- Extract complex conditions into named booleans
- One responsibility per function
- Use `logger` for all warnings/errors
```

Claude now knows:

- How long functions should be
- How to deal with complex logic
- That logging is expected, not optional

Prefer examples over theory

❌ Only rules:

```
Use proper error handling.
```

✅ Example with real code:

```
### Error handling pattern

try:
    result = risky_operation()
    return {"data": result}
except SpecificError as e:
    logger.error(f"Failed: {e}")
    return {"error": str(e)}
```

Examples are how LLMs *actually* learn your taste.

Make "NEVER" and "ALWAYS" count

Reserve them for things that really matter:

```
IMPORTANT: Run migrations after pulling `main`.
NEVER expose internal IDs in API responses.
ALWAYS use Pydantic models for request/response validation.
```

If everything is IMPORTANT, nothing is.

Quick-Start Template (Copy, Paste, Fill In)

Drop this into `.claude/CLAUDE.md` and customise:

```
# [Project Name]

[One sentence - what does this project do and for whom?]

## Commands

- `[command]`    # Start dev server
- `[command]`    # Run tests
- `[command]`    # Run migrations / builds / lint

## Tech Stack

- Language: [TypeScript / Python / Go / …]
- Framework: [Next.js / FastAPI / Spring Boot / …]
- Database: [PostgreSQL / MySQL / MongoDB / …]
- Other: [Redis / Kafka / Stripe / etc.]

## Code Style

- [Rule 1 - concrete and testable]
- [Rule 2 - e.g. “Prefer composition over inheritance.”]
- [Rule 3 - e.g. “Use feature-based folder structure in `src/`.”]

## Critical

- NEVER [thing that must not happen]
- ALWAYS [thing that must always happen]

## Current Focus

- [What you’re actively working on this week/month]
```

Start small. Every time you catch yourself typing the same explanation to Claude, promote it into this file.

Patterns for Different Tech Stacks

Here are some quick variants you can adapt.

React + TypeScript

```
# Frontend App

## Tech Stack
- React 18
- Next.js 14 (App Router)
- TypeScript (strict mode)
- Tailwind CSS
- React Query for data fetching
- Zod for schema validation
## Commands
- `npm run dev`      # Start dev (http://localhost:3000)
- `npm test`         # Jest + React Testing Library
- `npm run lint`     # ESLint
- `npm run build`    # Production build
## Patterns
- Use Server Components by default; add `"use client"` only when needed.
- Use React Query for all server data fetching.
- Use Zod schemas for validating API responses.
- Keep components in `app/(routes)` and shared UI in `components/`.
## Critical
- NEVER fetch directly with `fetch()` in components; use React Query.
- ALWAYS type API responses using shared types in `@/types`.
```

Go Service

```
# Payments Service

## Tech Stack
- Go 1.22
- Fiber
- GORM
- PostgreSQL
## Commands
- `go run main.go`   # Run server
- `go test ./...`    # Run all tests
## Patterns
- Use table-driven tests.
- Wrap errors using `fmt.Errorf("context: %w", err)`.
- Use `context.Context` across all handlers and DB calls.
## Critical
```

```
- NEVER ignore `context.Context` in HTTP handlers.
- ALWAYS handle errors explicitly; no silent failures.
```

Python ML Service

```
# Model Serving API

## Tech Stack
- Python 3.12
- FastAPI
- PyTorch
- PostgreSQL
## Commands
- `uvicorn app:app --reload`    # API server
- `python train.py`             # Train models
- `pytest -v`                   # Run tests
## Patterns
- Type hints required.
- Async endpoints in FastAPI.
- Models live under `models/`.
- Use Pydantic models for all request/response bodies.
## Critical
- NEVER load large models in request handlers; initialise at startup.
- ALWAYS log inference latency and model version.
```

The “Quick Update” Trick

Don’t let `CLAUDE.md` rot.

Here’s an easy workflow:

1. You’re explaining something new to Claude in chat (e.g., “We use React Query for all fetching now”).
2. When you realise “this should be permanent knowledge”, promote it into `CLAUDE.md` instead of just saying it once.
3. Keep a tiny “To add to CLAUDE.md” section at the bottom while you’re iterating.
4. Clean it up and commit with your next PR.

Some Claude Code setups also support shortcuts (like `#`) to propose updates directly from your current conversation, so you don’t even have to leave the flow.

Real Impact (From Actual Usage)

I tested this across multiple projects.

Before `CLAUDE.md`:

- ~15 messages per task
- ~20 minutes per medium feature
- 5+ style corrections (“we don’t do it like that here”)
- Repeating the same context every day

After `CLAUDE.md`:

- ~3 messages per task
- ~5 minutes per similar feature
- 0–1 minor corrections
- Essentially zero context repetition

Call it roughly **4× faster** for the kind of work Claude Code is good at — refactors, feature wiring, tests, small scripts.

Common Mistakes

Writing way too much

Don’t dump your entire onboarding guide in here.

- Keep `CLAUDE.md` focused
- Move big docs into separate files and reference them with `@path/to/doc.md` if needed

If Claude needs more context, it can look those up.

Being vague

“Write good tests” is useless.

“Use `pytest` with fixtures from `tests/conftest.py`” is something Claude can actually do.

Letting it drift out of date

If `CLAUDE.md` says React 16 but your app is on React 18:

- Claude may suggest APIs that no longer exist
- You'll spend time correcting it instead of letting it run

Do a **5-minute review once a month** or whenever you ship major architectural changes.

Not emphasising what truly matters

If you bury your most important rules in the middle of a paragraph, they will be ignored.

Pull them out:

```
## Critical

- NEVER [X]
- ALWAYS [Y]
```

Make it visually obvious.

Using CLAUDE.md in Teams

You can also split things into **team-wide** vs **personal** preferences.

Shared file (commit this):

```
./.claude/CLAUDE.md
```

```
## Team Standards

- TypeScript strict mode enabled
- Tests required for new features
- Max cyclomatic complexity: 10
- Use feature-based folders in `src/`
```

Personal file (do *not* commit this):

`./CLAUDE.local.md` (add to `.gitignore`)

```
## My Preferences


- Prefer arrow functions for React components
- Add `TODO(username)` comments for follow-ups
- Prefer early returns over deep nesting
```

Everyone gets the same **baseline behaviour** and individuals can still tune Claude to their own taste without fighting in PRs.

Troubleshooting

Claude seems to ignore `CLAUDE.md` ? Check:

1. File is named exactly `CLAUDE.md` (case-sensitive).
2. It lives either in the repo root or inside `.claude/` .
3. The Markdown is valid (no half-open code fences breaking parsing).
4. Restart Claude Code or open a fresh session in that repo.

Still confused? Just ask:

> “Can you summarise what you learned from `CLAUDE.md` in this repo?”

If the summary looks wrong, fix the file.

Bottom Line

`CLAUDE.md` is the difference between:

- A generic coding assistant that keeps asking basic questions
- And a *project-aware teammate* that already knows your stack, patterns and current priorities

It takes **10 minutes** to set up the first version.

It saves **hours every week** after that.

Do this today:

1. Create `.claude/CLAUDE.md` in your main repo.
2. Add: tech stack, core commands and 3–5 critical rules.
3. Add one or two *real code examples* that show your preferred style.
4. Any time you repeat yourself to Claude, treat that as a signal:

> *"This belongs in `CLAUDE.md`."*

References

- *Claude Code: Best practices for agentic coding*
- *Claude Code overview & docs*

Claude Code | Developer Productivity | Ai Tools | Context Engineering | Software Engineering

P

Follow

Written by Pankaj

1.1K followers · 12 following

Expert in software technologies with proficiency in multiple languages, experienced in Generative AI, NLP, Bigdata, and application development.

No responses yet

qiulang

What are your thoughts?