Your codebase has a context problem.
And so does your AI.

Tomas Peluritis · Uncle Data

Tomas Peluritis

Head of Data @ Mediatech

aka Uncle Data

https://www.linkedin.com/in/tomaspeluritis/

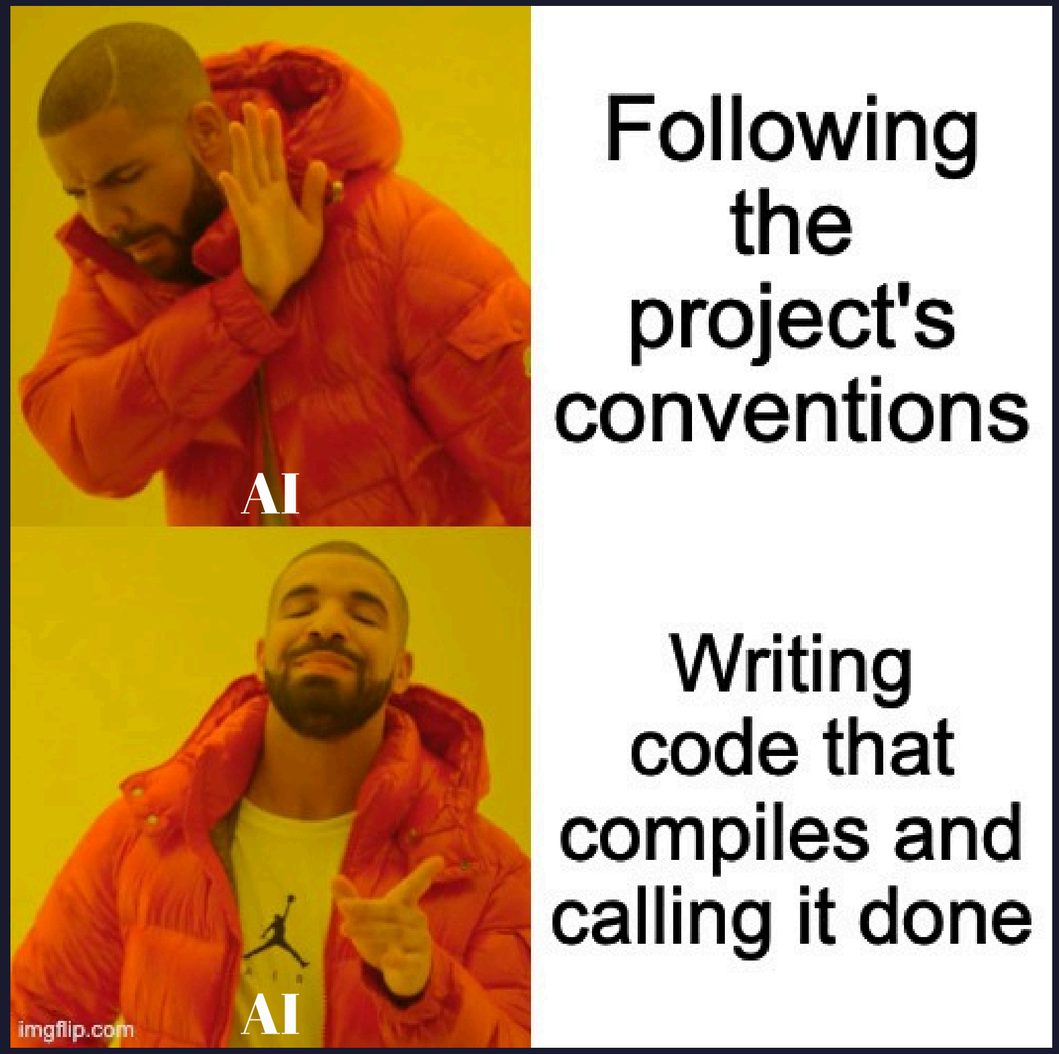
https://podcasters.spotify.com/pod/show/duomenu-dede

https://uncledata.substack.com

It started with a conversation about *product engineering*.

The Problem.

You paste code into an AI tool. It compiles. It also ignores every convention you have.


AI
AI
imgflip.com
Following the project's conventions
Writing code that compiles and calling it done

The output compiled. It also ignored every convention you have.

01 · IMPORTS

Wrong import style.

Absolute paths where you use relative. Or vice versa. Either way: not yours.

02 · NAMING

Wrong naming.

camelCase functions in a snake_case codebase. Suffixes that no one in your repo uses.

03 · TESTING

Wrong test framework.

unittest classes when the whole repo runs on pytest fixtures. Confidently.

This isn't an AI problem. It's a *context* problem.

The tool doesn't know how you work. Nothing in the prompt told it.

You would never onboard a human like this.

No README. No style guide. No walkthrough. Just "here's the repo, start coding."

WHAT THE AI SEES

Your prompt. That's it.

"Write a function that…" — 40 words of context for a 400,000-line codebase. No conventions, no examples, no taste.

You would never onboard a human like this.

No README. No style guide. No walkthrough. Just "here's the repo, start coding."

WHAT THE AI SEES

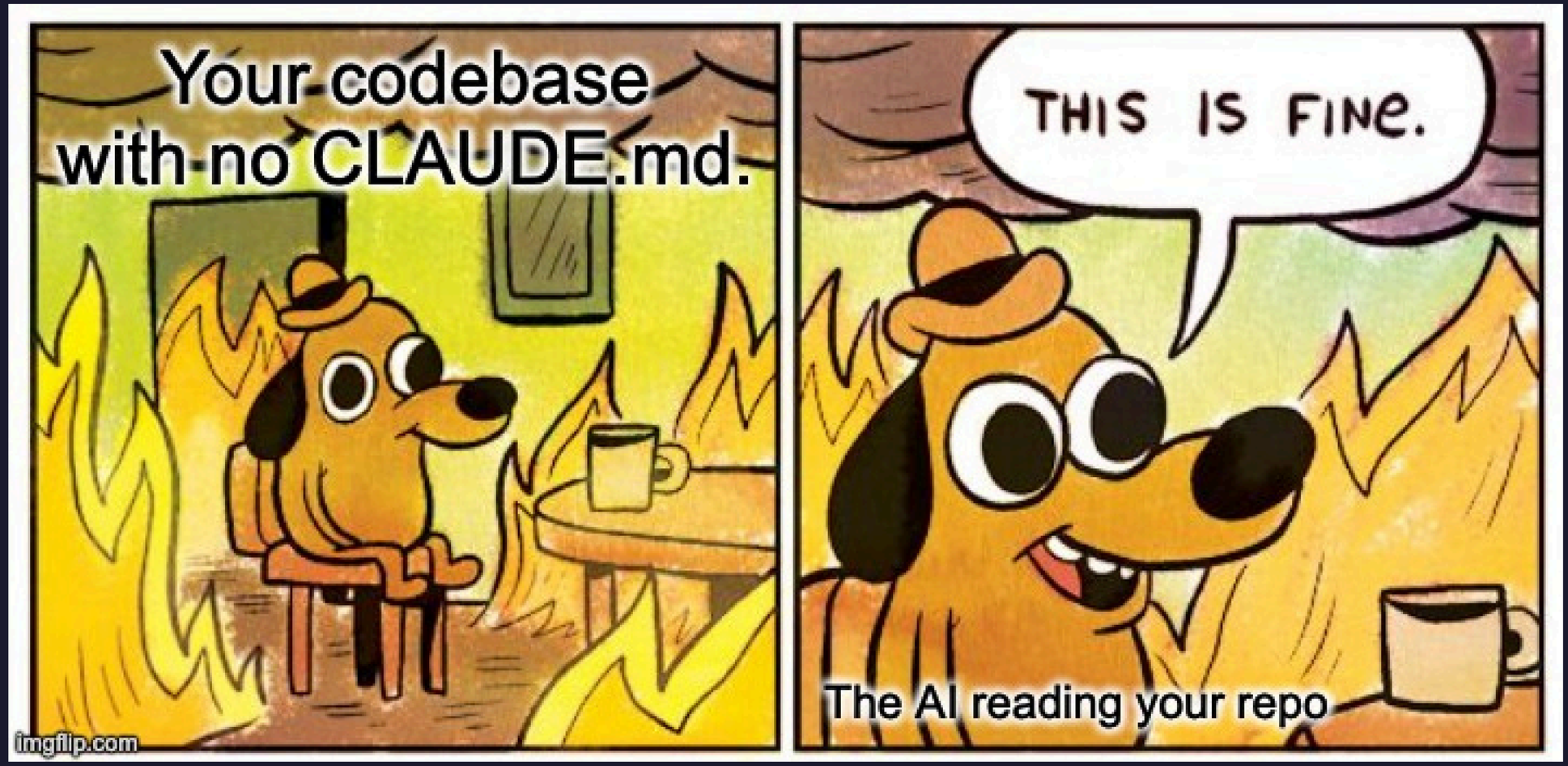
Your prompt. That's it.

"Write a function that…" — 40 words of context for a 400,000-line codebase. No conventions, no examples, no taste.

WHAT YOU SEE

Five years of tacit knowledge.

The CONTRIBUTING.md. The team's preferences. The fight about ORMs from 2022. Every code review you've sat through.


Your codebase with no CLAUDE.md.
THIS IS FINE.
The AI reading your repo
imgflip.com

What context means for a Python project.

Not abstract. Five concrete categories any Python dev recognises.

Context is five concrete things — and you already enforce all of them.

01

Structure

Layout, package boundaries, where things live.

02

Conventions

Naming, imports, formatting, type hints.

03

Tooling

ruff, mypy, pytest, uv. What runs in CI.

04

Domain

Vocabulary, invariants, what a "user" actually is.

05

Don't touch

Legacy modules, fragile migrations, the wrapper from 2019.

Both work. Only one passes code review.

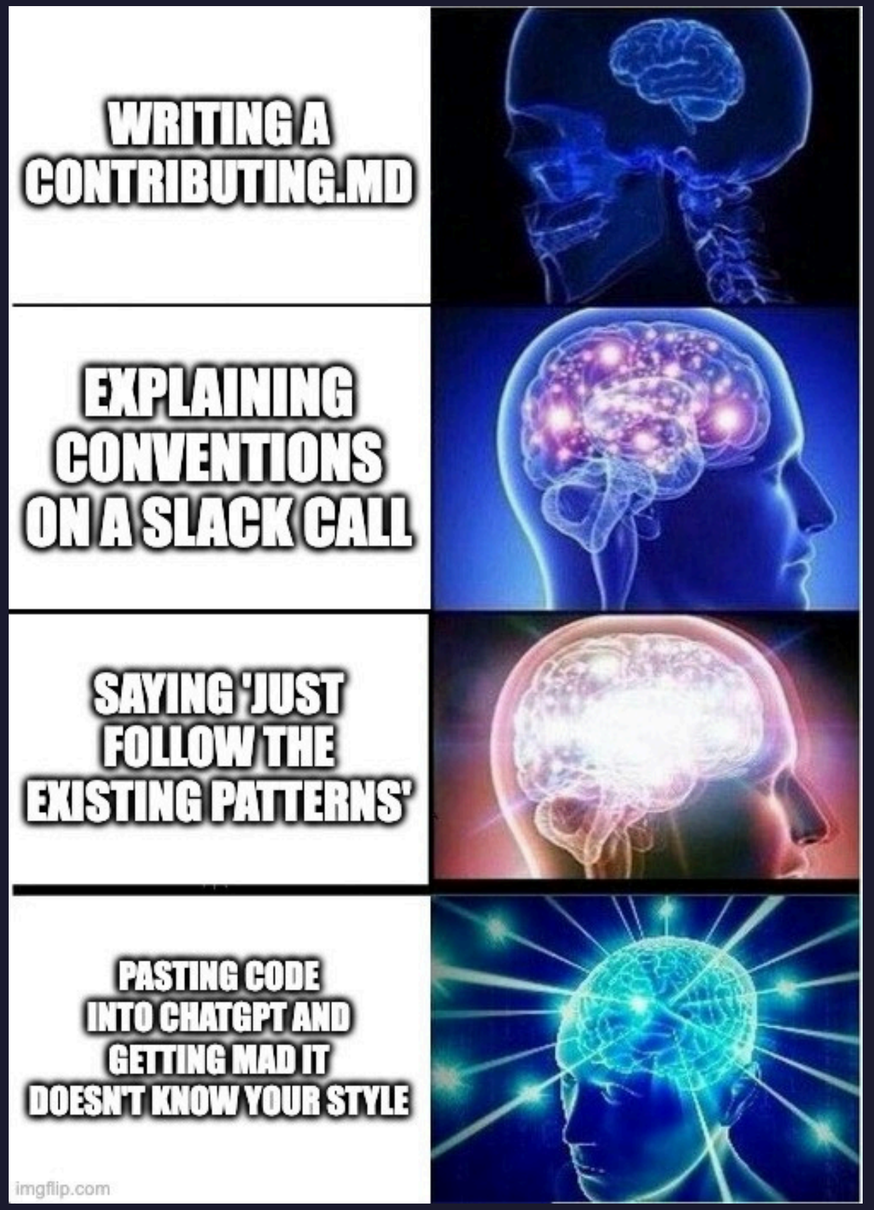
CORRECT, BUT CONTEXTLESS

```python
def getUserInvoices(userId):
    conn = sqlite3.connect("app.db")
    c = conn.cursor()
    c.execute(
        f"SELECT * FROM invoices "
        f"WHERE user_id={userId}"
    )
    rows = c.fetchall()
    return [dict(r) for r in rows]
```

CORRECT, AND IN YOUR STYLE

```python
async def list_invoices_for_user(
    user_id: UUID,
    *,
    session: AsyncSession,
) -> list[Invoice]:
    stmt = select(Invoice).where(
        Invoice.user_id == user_id,
    )
    result = await session.execute(stmt)
    return list(result.scalars())
```

SAME BEHAVIOUR. ONE FILE SHIPS; THE OTHER GETS SIX COMMENTS ON THE PR.


WRITING A CONTRIBUTING.MD
EXPLAINING CONVENTIONS ON A SLACK CALL
SAYING 'JUST FOLLOW THE EXISTING PATTERNS'
PASTING CODE INTO CHATGPT AND GETTING MAD IT DOESN'T KNOW YOUR STYLE
imgflip.com

We do this for people. We skip it for AI.

FOR PEOPLE, WE WROTE:

A style guide. A CONTRIBUTING.md. An onboarding walkthrough. A Slack channel where the team answers questions.

FOR THE AI, WE WROTE:

…the prompt. That's it. Then we got mad it didn't know.

The AI config layer.

Tools that solve the context problem. Files AI assistants actually read.

There are files AI tools actually read — at session start, automatically.

01

CLAUDE.md

Repo-root markdown. Claude Code reads it at session start.

02

.github/copilot-instructions.md

GitHub Copilot's per-repo instructions. Auto-loaded.

03

.cursor/rules/

Cursor's rules directory. Scoped per glob.

04

AGENTS.md

Emerging cross-tool convention. Same idea.

A real CLAUDE.md for a Python project.

CLAUDE.MD · REPO ROOT

```
# Acme Billing

## Stack
- Python 3.12, FastAPI, SQLAlchemy 2.x async
- Postgres 16. uv for envs. ruff + mypy in CI.
- pytest, pytest-asyncio. Coverage gate at 85%.

## Conventions
- snake_case for functions, PascalCase for classes
- Type hints on every public function.
- Pydantic models for API I/O, ORM for storage.
- No raw SQL outside src/db/migrations/.
```

CLAUDE.MD (CONTINUED)

```
## Testing
- One test file per module, mirror the path.
- pytest fixtures, not unittest.TestCase.
- factory_boy for object builders. No mocks of our own code.

## Don't
- Touch src/legacy/ — frozen, scheduled for kill.
- Add new sync DB calls. We're fully async.
- Introduce new top-level dependencies without a note in docs/adr/.
```

FORTY LINES OF PLAIN MARKDOWN. THAT'S THE WHOLE TRICK.

Prompt: "add an endpoint to list invoices by status."

WITHOUT CLAUDE.MD

```python
@app.route("/invoices")
def list_invoices():
    status = request.args.get("status")
    conn = sqlite3.connect("app.db")
    c = conn.cursor()
    c.execute(f"SELECT * FROM invoices
        WHERE status='{status}'")
    rows = c.fetchall()
    return jsonify([dict(r) for r in rows])
```

WITH CLAUDE.MD

```python
@router.get("/invoices",
        response_model=list[InvoiceOut])
async def list_invoices(
    status: InvoiceStatus,
    session: AsyncSession =
Depends(get_session),
) -> list[InvoiceOut]:
    stmt = select(Invoice).where(Invoice.status
== status)
    result = await session.execute(stmt)
    return [InvoiceOut.model_validate(i)
        for i in result.scalars()]
```

Line 1 → Wrong framework Flask, not FastAPI.

Line 2 → Sync, no types No async, no hints.

Line 6 → SQL injection f-string straight into SQL.

Line 8 → No schema Raw dict response, no Pydantic.

Better prompts look like specs, not wishes.

WISH

"Make a function that processes payments."

No contract. No constraints. No tests. You'll get something that compiles.

Better prompts look like specs, not wishes.

WISH

"Make a function that processes payments."

No contract. No constraints. No tests. You'll get something that compiles.

BETTER

"Write process_payment(order_id, amount). Should call Stripe, return PaymentResult."

Signature exists. Side effects named. Still: what about failures?

Better prompts look like specs, not wishes.

WISH

"Make a function that processes payments."

No contract. No constraints. No tests. You'll get something that compiles.

BETTER

"Write process_payment(order_id, amount). Should call Stripe, return PaymentResult."

Signature exists. Side effects named. Still: what about failures?

SPEC-DRIVEN

*"**Goal:** process_payment(order_id, amount). **Returns** PaymentResult. **Errors:** CardDeclined, NetworkError → retry 3x with backoff. **Idempotent** on order_id. **Tests first.**"*

Write the spec. Then the code.

01 **Write a one-page spec.**
Goal, signature, errors, edge cases. Markdown. Half an hour, max.

02 **Ask the AI to critique it.**
What's missing? Ambiguous? Untestable? Fix the spec before writing code.

03 **Tests first, then the implementation.**
The spec defines the contract; the tests enforce it; the code is whatever fills the gap.

04 **The spec lives in the repo.**
Next person — human or model — gets the same context you had.

Eivydas 15:45

[redacted]siaip crazy tas brainstorm and spec wtf

😄

nenugrybauja zostkai

va per pusdieni mazdaug veikiantis MVP

labai minimaliai kazka tweakint reikejo

Skills are reusable instruction bundles — for the moves you do over and over.

SKILL 01

Add a new endpoint.

Route, schema, ORM query, test, OpenAPI doc. Done the same way every time.

Skills are reusable instruction bundles — for the moves you do over and over.

SKILL 01

Add a new endpoint.

Route, schema, ORM query, test, OpenAPI doc. Done the same way every time.

SKILL 02

Write a migration.

Alembic revision, forward + backward, idempotent data step, test on a copy.

Skills are reusable instruction bundles — for the moves you do over and over.

SKILL 01

Add a new endpoint.

Route, schema, ORM query, test, OpenAPI doc. Done the same way every time.

SKILL 02

Write a migration.

Alembic revision, forward + backward, idempotent data step, test on a copy.

SKILL 03

Triage a flaky test.

Re-run, bisect, isolate fixture, log the timing. Quarantine or fix.

— THE WHOLE IDEA

It's not magic. It's a README the *AI actually reads*.

The side effect nobody talks about.

Writing it down for the AI is writing it down. Full stop.

Writing it for the AI documents it for humans.

.GITHUB/COPILOT-INSTRUCTIONS.MD

```
1  # How we work here
2
3  ## Naming
4  - list_* for plural reads, get_* for one.
5  - Repository methods are async by default.
6
7  ## Errors
8  - Raise domain exceptions in services.
9  - Translate to HTTP at the router layer only.
10
11 ## Pull requests
12 - Tests before behaviour. Spec link in
   description.
13 - Squash merge. Conventional commits.
```

FOR COPILOT

Generates code that lands closer to "review-ready" on the first try.

FOR THE NEW HIRE

Twelve bullets they would have learned in a month of code reviews — given on day one.

FOR FUTURE-YOU

A reminder of what you decided, before you stop remembering why.


Our onboarding doc for new devs
Our onboarding doc for AI tools
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.
imgflip.com

— THE TALK BENEATH THE TALK

Your codebase docs aren't good enough.

Not for the AI. Not for the humans either.

The AI just made it obvious. Most projects run on oral tradition — the senior who knows where the bodies are buried, the Slack channel that answers everything, the unwritten rules you only learn by getting a PR rejected. The model can't read your team's group chat. Neither can the next person you hire.

What still breaks.

Honest about the limits. None of this turns AI into a senior dev.

It will still confidently write wrong code. Plan for it.

CAVEAT 01

Context windows aren't infinite.

A 200-page spec doesn't all fit, or doesn't all get attended to. Be ruthless about what's at the top.

CAVEAT 02

Domain nuance is hard to encode.

"An invoice is paid only after the webhook AND the ledger reconcile" lives in someone's head.

CAVEAT 03

Plausible-looking wrong code.

It compiles. Tests pass. Names look right. The semantics are quietly wrong. Read carefully.

CAVEAT 04

Config files drift.

Your CLAUDE.md from six months ago might be lying. Treat it like a doc; version it; review it.

NOT "AI WRITES YOUR APP."
"AI WRITES THE BORING PARTS CORRECTLY — INSTEAD OF ALMOST CORRECTLY."


Write the documentation.
Complain the AI doesn't understand my code.
MAJORITY OF DEVELOPERS
imgflip.com
JAKE-CLARK.TUMBLR

Three things to do tomorrow.

Small. Concrete. Do them on one project, not all of them.

Three moves. All cheap.

01 **Add a CLAUDE.md to one project.**
Or copilot-instructions.md, or cursor rules. Whichever tool the team actually uses. Forty lines is enough.

02 **Write down the conventions you enforce in code review but never documented.**
Open the last ten PRs you commented on. Every comment is a missing line in the doc.

03 **Notice how much of your project's knowledge lives only in people's heads.**
That's the real risk. The AI just made it legible.

The best docs you'll write this year might be for a machine.

The humans will thank you too.

"Be precise. Think about why. Anticipate who comes next."

Questions?

uncledata.github.io