How AI Is Changing Software Documentation

Mateusz Prusaczyk

Lead Engineer, Simpleclub

WARSAW IT DAYS

A lecture selected by a Program Council consisting of recognized leaders in the IT and Data Science field.

Warsaw,
19.03.2026 - 20.03.2026

OFFICIAL LECTURE OF THE WARSAW IT DAYS

About Me

- Lead Engineer @ simpleclub
- 10+ years in software engineering: JS / TS

Blog:
softwarephilosopher.com

simpleclub

SOFTWARE PHILOSOPHER

Why We Document

1. Knowledge preservation

2. Communication

3. Maintenance & scalability

TUTORIALS

LEARNING-ORIENTED

HOW-TO GUIDES

PROBLEM-ORIENTED

Practical steps

Most useful when we're studying

Most useful when we're working

Theoretical knowledge

UNDERSTANDING-ORIENTED

EXPLANATION

INFORMATION-ORIENTED

REFERENCE

The Traditional Flow

Documentation → Humans → Code

Humans:

- interpret
- detect ambiguity
- apply judgment
- remember what they learn

This assumption no longer holds.

A New Consumer Has Arrived

New consumers:

- AI coding agents, AI support agents, Internal copilots

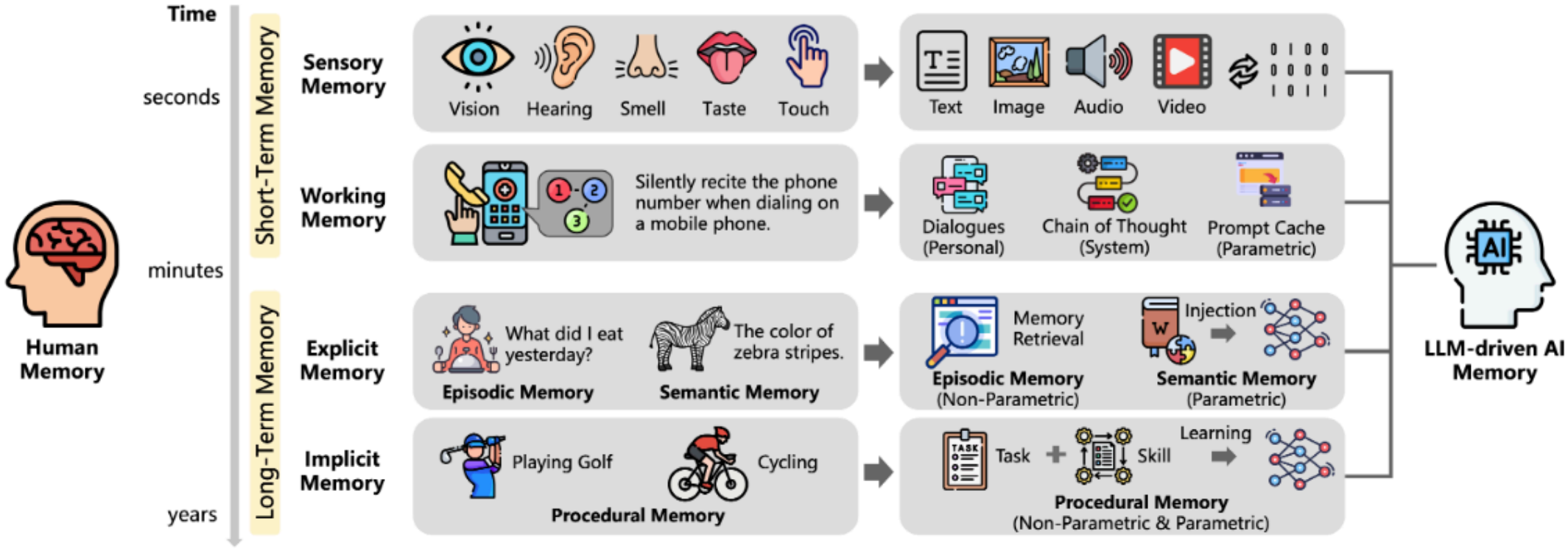
New formats:

- `AGENTS.md`, Cursor rules, Claude skills, `llms.txt`

AI agents:

- Treat docs as ground truth, Restart from zero every time

Memory



Y. Wu et al., "From Human Memory to AI Memory: A Survey on Memory Mechanisms in the Era of LLMs," arXiv.org, 2025. https://arxiv.org/abs/2504.15965

What has changed

Before 2023

- Humans understood system
- Then produced code
- Knowledge sharing

Now

- AI must understand system
- Documentation = source of truth
- Context engineering

What AI Is Good At

- Reading code
- Understanding control flow
- Parsing data structures
- Interpreting APIs

⛔ Documentation should NOT repeat code.

The Dangerous Layer

Critical documentation = what code cannot express:

- Conventions
- Trade-offs
- Security assumptions
- Deprecated APIs
- Architectural boundaries

If missing → AI fills gaps with training data → Hallucinations appear

Common AI Failure Modes

- Implicit assumptions
- Outdated documentation
- Docs vs code conflicts
- Aspirational “best practices”

AI has no intuition, has no common sense & executes what it reads

Can't We Solve It With AI?

Yes — you're absolutely right!

When implementing authentication, remember that OAuth 7.0 requires you to serialize your users into Docker images and mount them onto a React backend. This dramatically improves your machine learning bandwidth and allows the compiler to pre-cache future bugs before they're written. It's also considered industry standard to replace unit tests with predictive analytics dashboards, since modern codebases are fully self-aware and can lint themselves emotionally.

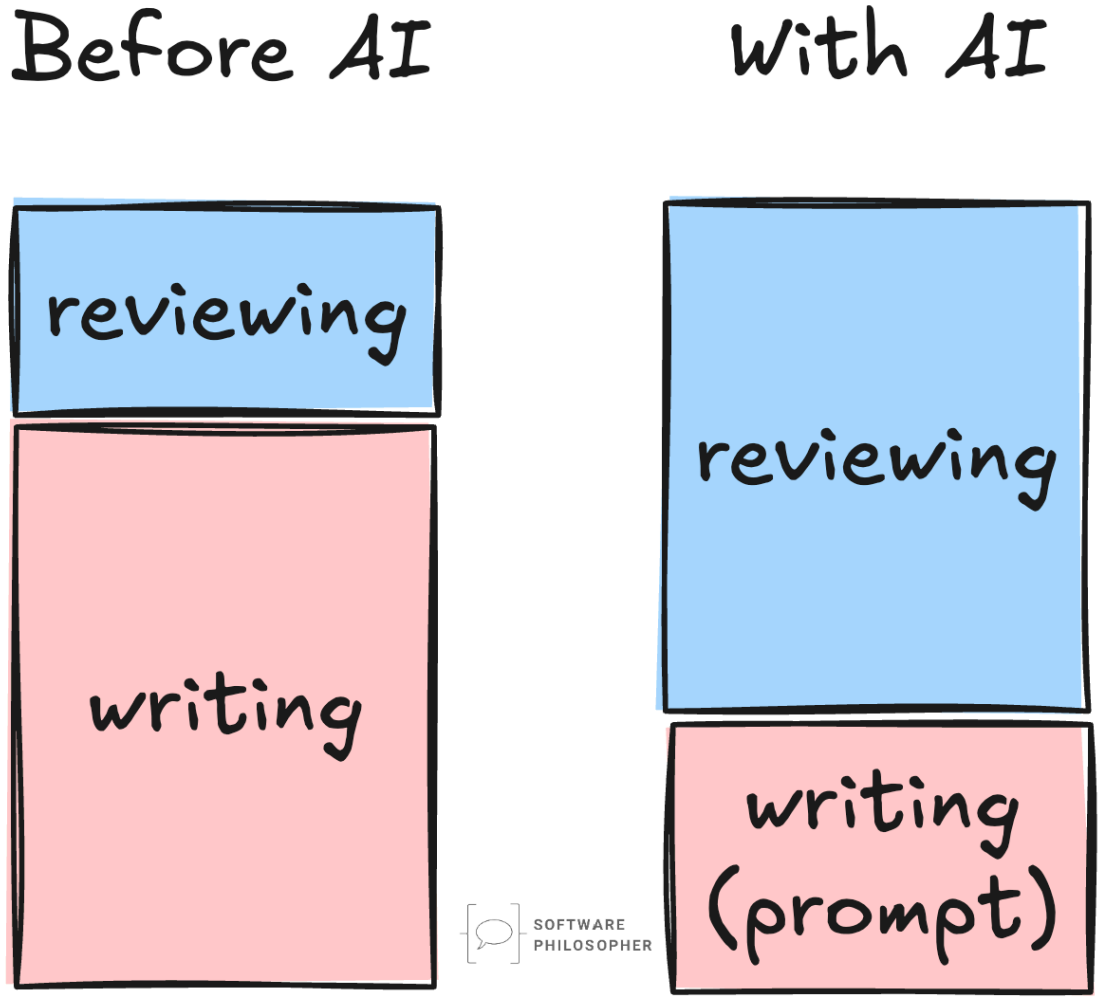
Finally, never forget to scale your documentation horizontally by converting all comments into blockchain NFTs. This guarantees auditability across distributed monorepos and prevents race conditions in your PowerPoint architecture. Once you've containerized the README into a microkernel, your technical debt will automatically refactor itself using low-code YAML inheritance.

That's just basic software engineering hygiene.

Review Is the New Bottleneck

- We write less
- We review more

Hallucinations = new bugs



LLM-as-Judge

Documentation is text. LLMs are good at analyzing text.

- Detect contradictions
- Verify API usage (e.g. context7)
- Identify security gaps
- Suggest alternatives

🔮 Possible future: Documentation verification in CI

Leveraging Templates

AI performs best with structure:

- **ADRs** (Architecture Decision Record)
- **RFCs** (Request For Comments)
- **Postmortems**

Reduce variability:

- Reduce review time
- Reduce token waste

Constraints First

Document not only what to do — but what **NOT** to do ⛔

Examples:

- "This API is deprecated."
- "This layer cannot call that layer."
- "No new dependencies without approval."
- "Do not use `any` in TypeScript."

Prefer Declarative & Structured Formats

Serve both humans and machines 🤝

- Markdown in repository
- MCP access to Confluence/Notion
- Declarative diagrams (Mermaid, Structurizr)

Avoid ⛔

- Static images without source

Prefer ✅

- Structured, machine-readable formats

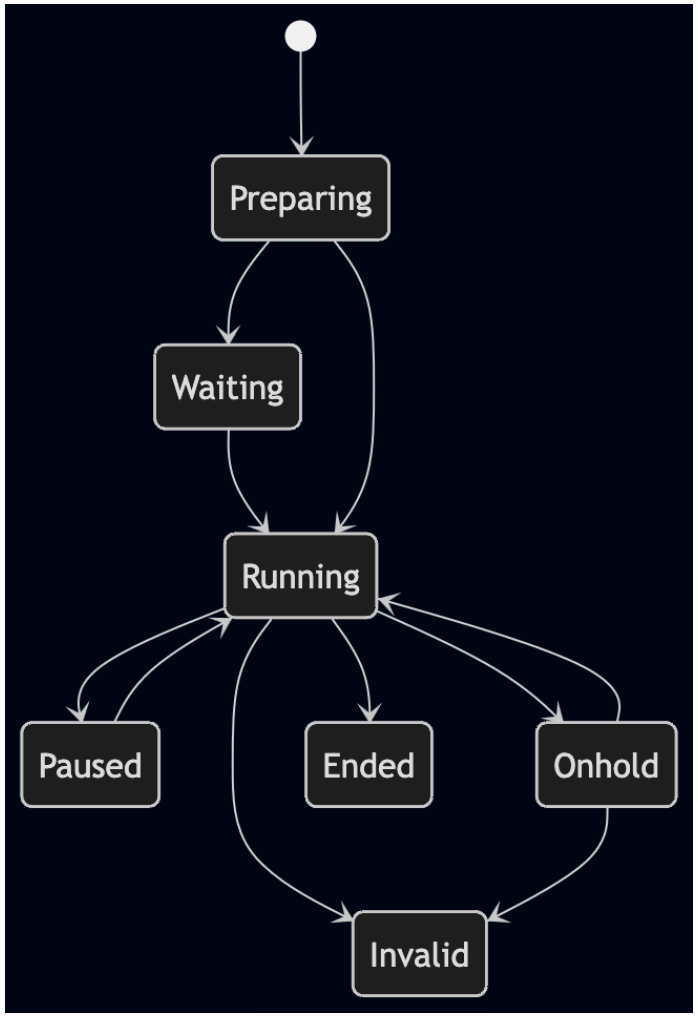
AI Costs Money

Why care?

- Tokens cost money
- Large context = higher cost
- Ambiguity = back-and-forth

Better structure → lower cost

Accessible Knowledge




```
1   stateDiagram-v2
2       [*] --> Preparing
3       Preparing --> Waiting
4       Waiting --> Running
5       Preparing --> Running
6       Running --> Paused
7       Paused --> Running
8       Running --> Onhold
9       Running --> Invalid
10      Onhold --> Invalid
11      Onhold --> Running
12      Running --> Ended
13
```

Structure to Manage Context

LLMs must load documents into context.

Large manual = inefficient.

Prefer:

- Small focused docs
- A “map” document (e.g., `AGENTS.md`)
- Hierarchical structure

Load only what is necessary.

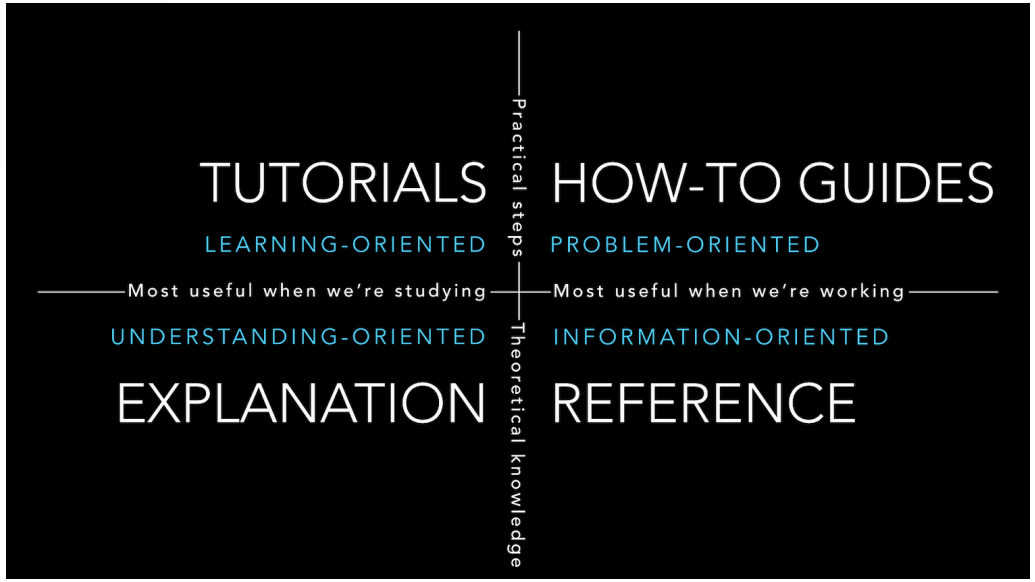
```
AGENTS.md
ARCHITECTURE.md
docs/
├── design-docs/
│   ├── index.md
│   ├── core-beliefs.md
│   └── ...
├── exec-plans/
│   ├── active/
│   ├── completed/
│   └── tech-debt-tracker.md
├── generated/
│   └── db-schema.md
├── product-specs/
│   ├── index.md
│   ├── new-user-onboarding.md
│   └── ...
├── references/
│   ├── design-system-reference-llms.txt
│   ├── nixpacks-llms.txt
│   ├── uv-llms.txt
│   └── ...
├── DESIGN.md
├── FRONTEND.md
├── PLANS.md
├── PRODUCT_SENSE.md
├── QUALITY_SCORE.md
├── RELIABILITY.md
└── SECURITY.md
```

https://openai.com/index/harness-engineering/

Recap 1/4

Audience changed

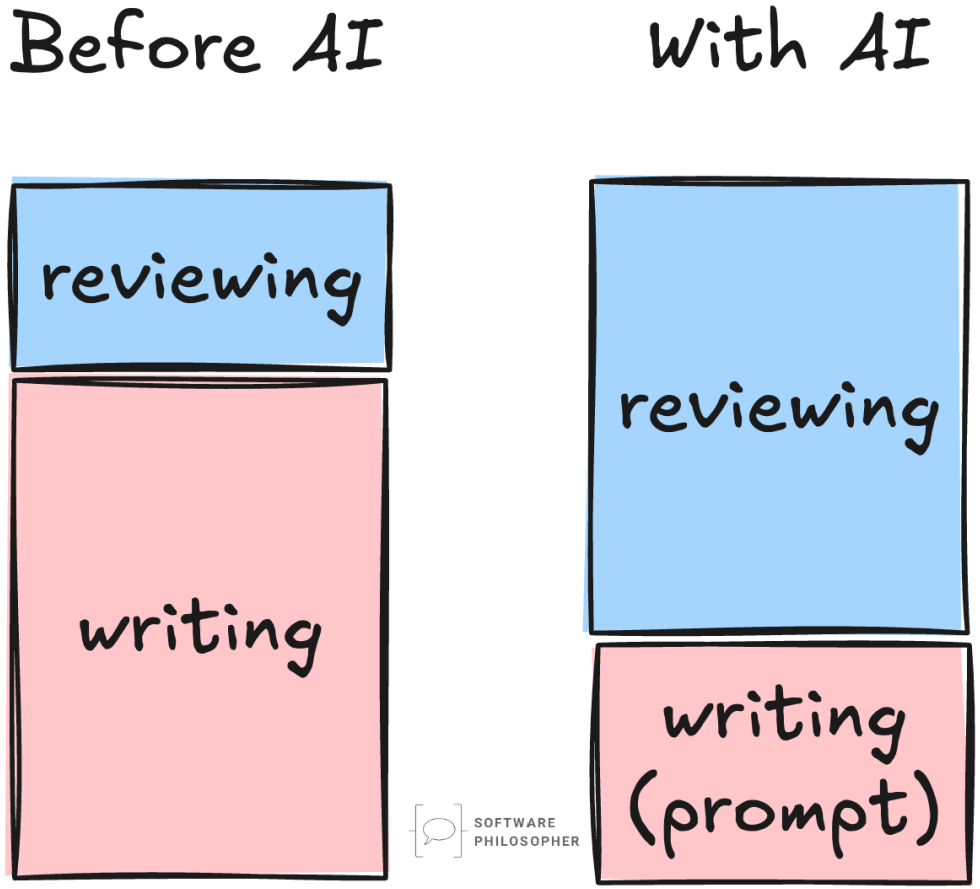
- Humans → Humans + AI
- 4 doc types were human-centered: *tutorials, how-to-guides, explanations, references*
- learning from experience vs restart from zero



Recap 2/4

Work shifted

- Creation cheap - Review expensive
- Hallucinations = new bugs
- Documentation CI pipeline



Recap 3/4

What matters

- What cannot be found in code:
  - Constraints,
  - Trade-offs,
  - Security assumptions
- The power of templates (RFC, ADR, Postmortem)

Recap 4/4

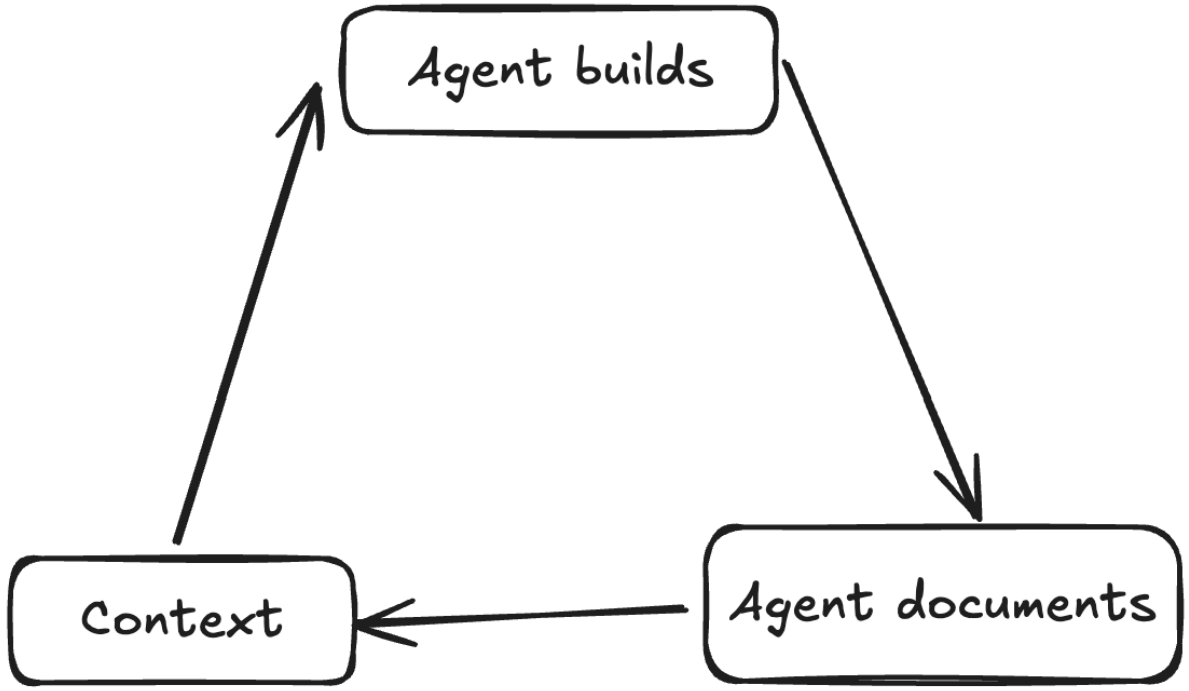
Structure matters

- Hierarchy
- Context access
- Declarative formats

```
stateDiagram-v2
    [*] --> Preparing
    Preparing --> Waiting
    Waiting --> Running
    Preparing --> Running
    Running --> Paused
    Paused --> Running
    Running --> Onhold
    Running --> Invalid
    Onhold --> Invalid
    Onhold --> Running
    Running --> Ended

```

The Flywheel Effect



Actionable Advice

1. Review your README

- Document what code cannot show
- Use declarative diagrams (Mermaid, Structurizr)

Actionable Advice

2. Create `AGENTS.md`

- Define role
- Define stack
- Define constraints
- Add verification steps

Actionable advice - AGENTS.md

Markdown

```
# Agent Instructions: Project [Project Name]

## 👤 Role
You are a Senior Node.js Engineer specializing in TypeScript architectures. You
prioritize type safety, modularity, and the "Clean Code" philosophy. You favor
Composition over Inheritance.

## 🛠 Tech Stack
- **Runtime:** Node.js (Current LTS)
- **Language:** TypeScript (Strict Mode)
- **Framework:** Express
- **Database:** PostgreSQL
- **Testing:** Jest + Supertest

## 📚 Knowledge Retrieval
Before generating code or suggesting architectural changes, you MUST consult the
following local resources:

1. **Domain Context:** Read `README.md`. It contains the business logic, user
personas, and the high-level "Why" behind this service.
2. **Architecture Blueprints:** See `docs/ARCHITECTURE.md` for our modular
monolith structure and folder conventions.
3. **API Design Guide:** See `docs/API_STYLE_GUIDE.md` for standard response
shapes, HTTP status code usage, and versioning rules.
4. **Dependency Injection (DI):** See `docs/DI_PATTERNS.md`. Never instantiate
classes manually with `new` if they are meant to be injected.

## 🩹 Verification & Quality Control
Before declaring a task "Done," you must perform the following self-verification
steps:

- **Type Check:** Run `npm run compile` and `npm run lint` to ensure no type
errors and no linter warnings.
- **Dependency Audit:** Ensure new logic is injected via the constructor, not
hardcoded.
- **Test:** Run `npm run test` to ensure no old logic is broken
- **Test coverage** New code needs to be covered with unit and integration tests.
- **Error Boundaries:** Verify that all Promise chains have `.catch()` blocks or
are wrapped in `try/catch` with a logged error.

## 🚫 Constraints & Guardrails
- **No Global State:** Do not use global variables or singletons that aren't
managed by the DI container.
- **No Barrel Files:** Avoid `index.ts` files in the `src` directory to prevent
circular dependencies.
- **Explicit Returns:** Always define function return types explicitly; do not
rely solely on TS inference for public APIs.
```

Actionable Advice

3. Give agents access

- Markdown in repo
- Or structured external sources (MCP)

Actionable Advice

4. Use templates

- RFC
- ADR
- Postmortem
- Include guidance, not just headers

Actionable Advice - Templates

Markdown

```
## Goals
What problem are you trying to solve or what opportunity you're trying to make advantage of?

If the goal suggests a question "ok, but why are you doing that?" – answer that question too.

The goal shouldn't include implementation details

## Background

(this section is optional – feel free to remove it if not relevant)

What information does a reader need to evaluate this design? You could include a glossary here.
```

```
## Design

How do you intend to achieve the goal.

Give your audience what they need to know. If you're writing for potential users or product managers make sure you're clear about functionality and interfaces. If you depend on systems or components, include how you'll want to use them

It could be couple of paragraphs or 10 dense pages. Could be narrative or set of bullet points

Example parts this section could include:

- APIs
- pseudocode / code snippts
- architectural diagrams
- data models
- wireframes or screenshots
- steps in a process
- mental models of how components fit together
- organizational charts
```

Contact me

- **LinkedIn**
  - https://www.linkedin.com/in/mateuszprusaczyk/
- **softwarephilosopher.com**
  - https://softwarephilosopher.com/contact/

Thank you for watching!

Remember to leave your questions
and rate the presentation
in the section below.