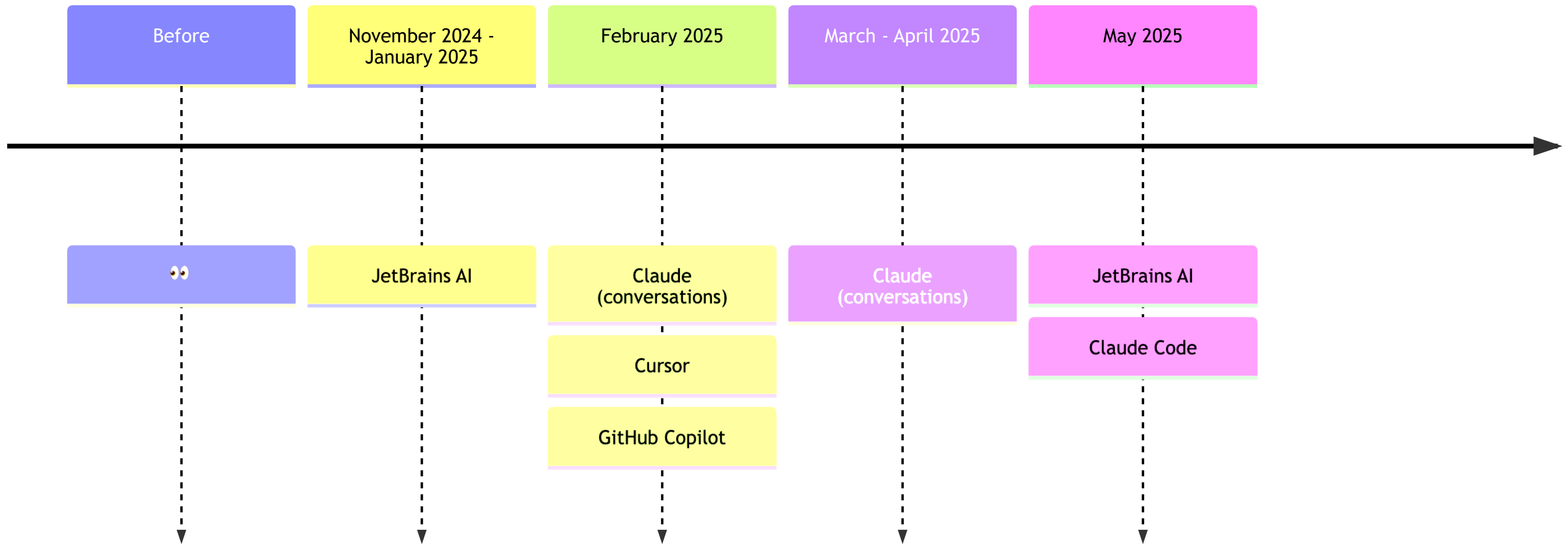
AI for managing technical debt

About me

- Full-stack developer at Kumojin
- Software craftsman
- 14 years of experience
- Advanced AI user for development for 1 year

My journey with AI in development



The problem: technical debt accumulates

How it happens

- External ecosystem degradation
- Accumulation of internal decisions
- Growth complexity

AI can help… or make it worse

✅ Potential

- Analyze thousands of lines
- Detect obsolete patterns
- Propose migrations
- Automate refactoring

⚠️ Risk

- Out-of-context suggestions
- Unrealistic estimates
- Untested changes
- **Technical debt increases instead of decreasing**

Definitions: the tools

LLMs (Large Language Models)

Advanced generative AI systems that understand and generate natural language.

Examples: GPT-4, Claude, Gemini

Coding agent

A tool that reads, modifies, or fixes code autonomously using natural language as input.

Examples: Claude Code, Cursor, GitHub Copilot

How does an LLM work?

The two key steps to create an LLM:

Pre-training: models analyze billions of text examples, learning to predict what comes next

💰 Millions of $ / ⏱️ Weeks/months / 🖥️ Thousands of GPUs

Fine-tuning: models are refined to follow instructions, be helpful, and avoid harmful content

📊 Less data / 👤 Human feedback / ✅ Desired behavior

What happens when you send a prompt?

1. Your prompt is **tokenized** (split into chunks)

```
"Analyze this function" → ["Analyze", "this", "function"]
```

2. The model processes these tokens through its neural network
3. **Predicts the MOST LIKELY next token** based on learned patterns

```
"def calculate(" → next token: probably "x" or "self"
```

4. Adds this token to the sequence
5. Repeats until generating a complete response

⚠️ **Key point**: "Most likely" ≠ "Correct"

This is **statistical prediction**, not magic.

Key capabilities

✅ **Massive analysis**

- Thousands of lines in seconds

✅ **Pattern recognition**

- Anti-patterns, code smells

✅ **Learning by example**

- Give it examples, it adapts

✅ **Tool connection**

- Read files, execute code

Limitations

⚠️ **Knowledge cutoff** (varies by model)

- Doesn't know recent library versions for example

⚠️ **Hallucinations**

- Can invent APIs

⚠️ **Context window** (~200K tokens)

- Memory limit

⚠️ **Complex reasoning**

- Algorithms, advanced math

The philosophy

Two approaches to using AI in development

❌ Vibe Coding

Prompts	"Create a user profile page"
Validation	No review
Understanding	Blind copy-paste
Speed	⚡ Fast at first
Long term	💥 Technical debt

Mindset: "It works, that's good enough"

✅ AI-assisted engineering

Prompts	"Refactor UserService to use DI"
Validation	Review + tests at each step
Understanding	Understand and adapt
Speed	🎯 Slower at first
Long term	📈 Maintainable

Mindset: "AI proposes, I validate"

Objectives of this presentation

What you'll see

- ✅ Practice **AI-assisted engineering** (not vibe coding)
- ✅ Understand and **validate each step**
- ✅ Apply on **real legacy code**

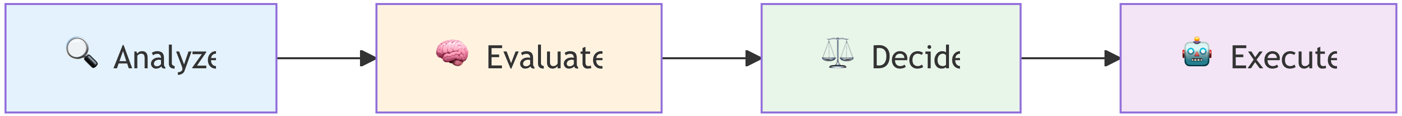
Our test subject: extreme-carpaccio

A kata that illustrates a **common professional situation**:

- Inheriting a project that hasn't been maintained for several years.
- Despite its initial quality, the code now has significant technical debt.

Positive point: an existing test suite

Demo roadmap



The pattern to observe

AI proposes → Human validates → AI executes

Why this pattern?

- Leverages AI speed without sacrificing control
- Reduces cognitive load: AI explores, you decide
- Scalable: adaptable from individual code to complex systems

Code analysis

Technical debt analysis prompt

```
Analyze this codebase for technical debt and architectural issues.

Focus on:
- Code smells and anti-patterns
- Outdated dependencies and framework versions
- Architectural problems (coupling, separation of concerns, etc.)
- Security vulnerabilities from deprecated packages
- Performance bottlenecks or inefficient patterns
- Missing error handling
- Inconsistent code patterns across the codebase

For each issue you identify:
1. Specify the exact location (file and line numbers)
2. Explain why it's problematic
3. Rate the severity (critical, high, medium, low)
4. Estimate effort to fix (hours or story points)
5. Suggest a specific remediation approach

After the analysis, generate a prioritized action plan with:
- Quick wins (high value, low effort)
- Critical issues that block modernization
- Long-term refactoring goals
```

Code analysis - Claude Code in action

What Claude does

- 🔍 **Glob**: Finds all files
- 📖 **Read**: Reads package.json, server.js, routes, etc.
- 🧠 **Analyze**: Identifies patterns, dependencies, code smells
- 📝 **Generate**: Structured report with priorities

Why Plan mode?

- ✅ Read-only (no risk)
- ✅ Ideal for exploration and review
- ✅ No accidental modifications
- ✅ Perfect for learning a new codebase

Code analysis - Results

What the AI found

- **Outdated dependencies**: React 0.12, jasmine-node
- **CVE vulnerabilities**: known security flaws
- **Anti-patterns**: callback hell, duplicated code
- **Clear structure**: Exact location + severity + effort + solution

What needs validation

- **Business context**: "arbitrary code execution" → yes, it’s a feature
- **Estimates**: "TypeScript migration: 8-10 days" → indicator, not truth
- **Knowledge cutoff**: Doesn’t always know the latest versions
- **Priorities**: AI doesn’t know your project constraints

Major migration: React 19.2

Migration prompt

```
Plan the React migration from the frontend to the latest version (19.2).

Constraints:
- Do not migrate to TypeScript
- Ensure libraries compatibility with React 19.2
- Propose E2E testing strategy to validate before and after
```

Major migration: React 19.2

Interactive approach

- Claude analyzes and asks questions
- You guide the strategy
- Claude generates a detailed plan

Compare alternatives before migrating

The agent analyzes options and recommends the best approach for your context.

Example prompt

```
Search body-parser alternatives.

Compare:
- Performance and bundle size
- API compatibility
- Migration effort
- Active maintenance

Recommend the best option for our use case.
```

Replacement with native APIs

The agent identifies when a dependency can be replaced with native platform APIs.

Example prompt

```
Analyze our use of lodash in the codebase.

Identify functions that can be replaced with native JavaScript APIs.

For each replacement:
- Show the native equivalent
- Confirm identical behavior
- Highlight any edge cases
- Estimate migration effort

Prioritize high-usage functions first.
```

Automated migration with codemod

The agent writes the codemod and you validate the result

Example prompt

```
Write a jscodeshift codemod to convert all var declarations to const/let based on reassignment.

Workflow:
1. Generate the codemod code
2. Explain what it does
3. Run with --dry-run on javascripts/*.js
4. Show me the diff of proposed changes
5. STOP and wait for my approval
6. After I approve, provide the final command to run without --dry-run

Do not execute the actual transformation until I explicitly approve.
```

And in real life?

Using AI to update and/or replace dependencies on projects

- Replace React-Script with Vite
- Migrate from React class components to function components
- Migrate React-Router 5x to 7x
- Replace moment with dayjs
- Major Material UI migration

Key takeaways

Five demos, one approach

1. **Code analysis**: AI analyzes, we evaluate
2. **Critical thinking**: AI estimated effort, we contextualize
3. **Migration decisions**: AI compared options, we chose
4. **Automation**: AI wrote code, we validated

Where to start on your projects

- **Start small**: code analysis (low risk)
- **Develop critical thinking**: question everything
- **Expand progressively**: migration → refactoring → automation
- **Keep control**: AI proposes, you decide

Questions?

https://github.com/xballoy/presentation-ai-tech-debt