AI Tech Debt Forge

Why This Matters

Every engineering organization accumulates technical debt. Dependencies go stale, patterns become anti-patterns, and the codebase drifts further from modern practices. The typical response is to either ignore the problem until it becomes critical or embark on a massive rewrite that takes months and often fails.

What if an AI agent could analyze your codebase, create a modernization plan validated by multiple engineering perspectives, and then execute that plan incrementally with human oversight? Not a magic wand that rewrites everything overnight, but a systematic forge that reshapes legacy code into something maintainable.

The "forge" metaphor is intentional: we take raw or degraded material and, through careful application of heat and pressure, reshape it into something stronger.

The Product Vision

A user provides a git repository URL. The system analyzes the codebase and produces a modernization report covering outdated dependencies, security vulnerabilities, deprecated APIs, code smells, and architectural concerns. But here is where it gets interesting: before presenting recommendations, the system validates them through multiple AI personas representing different engineering perspectives.

A Backend Engineer persona evaluates API contracts and data handling. A Security Engineer flags authentication issues and input validation gaps. An SRE persona checks observability and error handling. An Infrastructure persona reviews deployment and scaling concerns. Each persona critiques the plan from their viewpoint, and conflicts are surfaced for human resolution.

The user reviews the consolidated plan, makes adjustments, and approves sections for execution. The AI then implements changes incrementally, generating tests and documentation as it goes, with rollback capability if something breaks.

Agentic Architecture

The system uses a hierarchical multi-agent design:

****Orchestrator Agent**** manages workflow and resolves conflicts between specialists. This is the conductor that keeps everything coordinated.

****Analysis Agent**** scans the repository and builds a semantic model: dependency graphs, call hierarchies, data flows, configuration patterns. It connects to external sources via MCP

servers (Context7 for documentation, GitHub MCP for repo operations) and web search for migration guides.

Planning Agent takes the analysis and generates a prioritized modernization roadmap. It estimates effort, identifies breaking changes, and sequences work to minimize risk.

Validation Panel consists of multiple persona agents (Backend, Frontend, Security, SRE, Infrastructure) that independently review the plan. The Orchestrator synthesizes their feedback into a consolidated critique.

Execution Agent implements approved changes, generates tests, creates PRs, and documents modifications. It works incrementally and maintains rollback capability.

Implementation Hints

Start narrow. Pick one technology stack you know well (Python/FastAPI, Node/Express, Java/Spring) and build deep expertise there before expanding. The temptation to build a universal tool will sink you.

The semantic model is everything. Invest heavily in understanding not just the AST but the relationships: what calls what, what depends on what, how configuration flows through the system. Poor analysis leads to wrong recommendations.

Build trust incrementally. Start with low-risk suggestions (update this dependency, add this type hint) before proposing architectural changes. Users need to see the AI get small things right before trusting it with big things.

Test generation is non-negotiable. The Execution Agent should never make a change without generating tests that verify behavior is preserved. Look into property-based testing for stronger guarantees.

Plan for disagreement. What happens when the Security persona says "absolutely not" and the Backend persona says "this is fine"? The system needs a principled way to surface and resolve conflicts.

Interesting Questions

How do you measure technical debt reduction? Lines changed is meaningless. Cyclomatic complexity captures some things but misses others. What metrics actually matter?

Should the system refuse to proceed on codebases with no test coverage? Modernizing untested code is risky. Should the AI first generate a test suite that captures current behavior?

How much external context should the AI gather? Git history, PR discussions, Jira tickets, Slack

conversations all contain valuable context. But more context means slower, more expensive, and potentially more confused analysis.

Can the persona agents learn from feedback? If human security engineers consistently reject certain recommendations, can the Security persona adapt?

Can you track DORA, DX, and other software productivity metrics for the AI agent? DORA metrics (deployment frequency, lead time, change failure rate, time to restore) measure delivery performance. DX metrics capture developer experience and cognitive load. If the forge modernizes a codebase, do these metrics improve in subsequent sprints? Can you attribute improvements to specific AI interventions versus other factors? This would provide empirical grounding for claims about AI-assisted modernization.

Proof of Concept

Use the [Foundation Model Benchmarking Tool (FMBench)](https://github.com/aws-samples/foundation-model-benchmarking-tool) as the test case. It is a real codebase with real debt, with clear functionality to verify nothing breaks.

Success means: all existing functionality preserved, code quality metrics improved, dependencies updated, and humans feel the AI helped rather than replaced them.