



Athena Digital: The Software Factory Problem

Athena Digital is a fictional company. The scenario is directionally representative of what engineering organizations encounter when moving from individual AI tool usage to systematic software production.

THE SITUATION	THE QUESTION	THE CONSTRAINT
94% of engineers use AI coding tools daily, but cycle time has only improved 11%. Writing code is 22% of the work. The other 78% is untouched.	Can you design a software factory that takes a feature spec and outputs a deployable pull request—across the full stack—matching existing conventions?	Three engineers, 60 days, seven products, three languages. Output must pass CI on first run 80% of the time. Nothing ships without human review.

COMPANY BACKGROUND

Athena Digital is a 180-person product studio in Austin, Texas, with satellite teams in New York and London. It builds and operates **seven SaaS products** across fintech, healthtech, and logistics, generating \$62M in ARR. Each product has a squad of 4–8 engineers, a product manager, and a designer. The company averages 23 production deploys per day.

Eight months ago, Athena gave every engineer access to **Claude Code**, **Cursor**, and **Codex**. 94% use at least one daily. Developers report a **30–40% perceived productivity gain** on tasks like writing functions, debugging, and generating tests.

BY THE NUMBERS

7 Products
Fintech, healthtech, logistics

94%
Engineers using AI tools daily

11%
Actual cycle time improvement

23 deploys/day
Across all products

Measured outcomes are smaller. **Cycle time** (commit to production) improved 11%. Feature throughput per squad is up 8%. Engineers write code faster, but requirements decomposition, architecture decisions, cross-service integration, environment setup, CI/CD configuration, and review still move at human speed. Writing code was never the bottleneck.

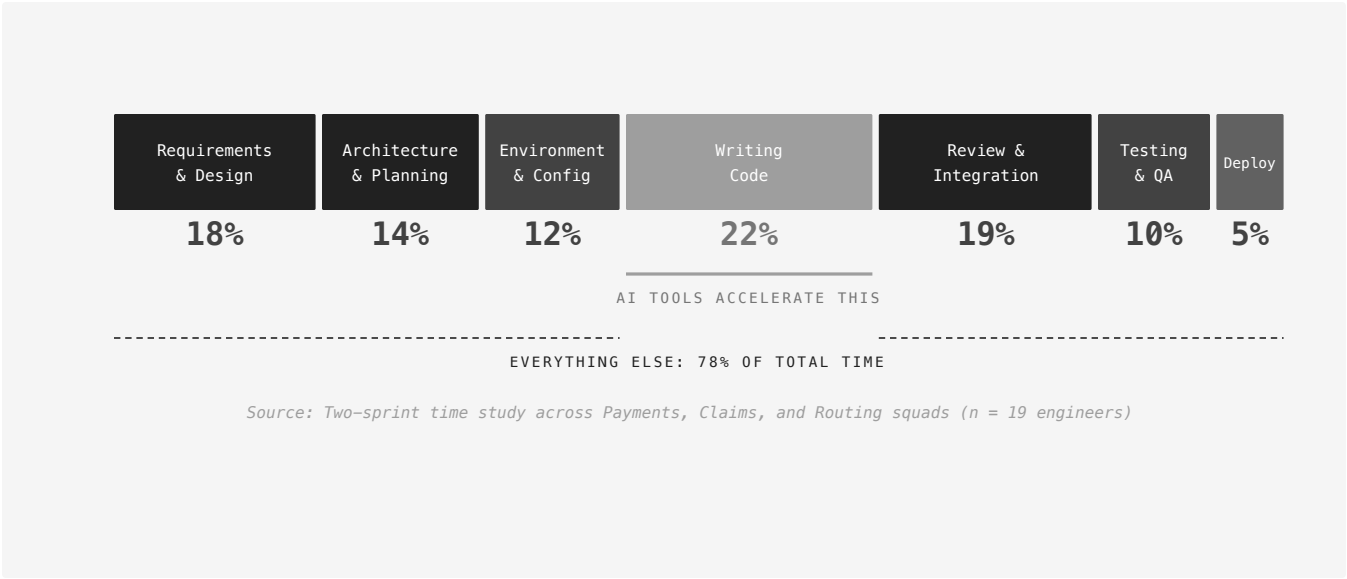
CURRENT TOOL LANDSCAPE

TOOL	PRIMARY USE	ADOPTION
Claude Code	Agentic coding, multi-file edits, terminal workflows	72% of engineers
Cursor	In-editor completions, inline chat, codebase Q&A	88% of engineers
Codex	Async background tasks, batch code changes	41% of engineers
GitHub Actions	CI/CD pipelines, automated testing	100% of repos
Terraform	Infrastructure provisioning	All products
Internal CLI (<code>athena-cli</code>)	Scaffolding, environment setup, deploy shortcuts	All engineers

THE GAP

A time study across three squads over two sprints tracked where engineering hours go from feature prioritization to production:

EXHIBIT A: WHERE ENGINEERING TIME GOES



AI tools accelerate the 22% spent writing code. The other 78% is untouched. A developer generates a React component in 30 seconds, then spends 45 minutes configuring the feature flag, updating the API contract, writing the migration, and aligning staging with production.

WHAT EXISTS IN THE WILD

The team evaluated two leading approaches to systematizing AI-assisted development:

<u>BMAD METHOD</u>	<u>RALPH</u>
Structured persona-based workflow	Multi-agent coordination layer
Analyst → Architect → Dev → QA	Spawns specialized sub-agents
Heavy use of markdown templates	Parallel task execution
Sequential handoff between phases	Context sharing via shared memory
Strong on documentation generation	Dynamic task decomposition
<i>Verbose; prompt-heavy; brittle at scale</i>	<i>Complex; hard to debug; nondeterministic</i>

BMAD assumes a single developer on a greenfield project with a linear flow. It breaks when you have seven existing codebases, established conventions, and teams shipping incrementally. Ralph-style orchestrators are opaque: when a multi-agent system produces a broken migration, no one can trace why, and the debugging cost erases the productivity gain.

WHAT ATHENA ACTUALLY SHIPS

The last 100 features shipped across all products follow one of four patterns:

PATTERN	DESCRIPTION	FREQUENCY
CRUD + UI	New entity with API endpoints, database table, and management interface	38%
Integration	Connect to external service (payment processor, EHR, carrier API) with sync logic	24%
Workflow	Multi-step business process (approval chain, claim lifecycle, onboarding flow)	22%
Analytics	Dashboard, report, or data export with aggregation queries and visualization	16%

Each pattern has a predictable anatomy: database changes, API layer, business logic, frontend, tests, environment configuration, deployment. The specifics vary—a payments CRUD differs from a patient records CRUD—but the structure is consistent. Today, every feature is built from scratch, with each developer re-making decisions that have been made before on other products.

THE PROVOCATION



We have seven products, four repeating patterns, and engineers who’ve proven they can work with AI tools. Why are we still treating every feature like a bespoke craft project? We need a factory, not a workshop.

CTO, Athena Digital

The CTO wants a **software factory**: a system that takes a high-level feature specification and outputs a complete, tested, deployable pull request—across the full stack—conforming to the target product’s existing conventions. Not a code generator. A production system that covers the four patterns above, and produces output a senior engineer can review rather than rewrite.

CONSTRAINTS

CODEBASE REALITY

Seven products, three languages (TypeScript, Python, Go), four database engines (Postgres, MongoDB, DynamoDB, Redis), two frontend frameworks (React, Next.js). No monorepo—each product has its own repo and conventions.

QUALITY BAR

Factory output must pass the existing CI pipeline (linting, type checking, unit tests, integration tests) on first run at least 80% of the time. Output requiring more than 30 minutes of human cleanup is a failure.

SECURITY & COMPLIANCE

Two products (fintech, healthtech) operate under SOC 2 and HIPAA. No credential leaks, no bypassed auth checks, no migrations that risk data loss. All factory output must be auditable.

HUMAN-IN-THE-LOOP

Nothing ships without human review. Output must include clean diffs, clear commit messages, and an explanation of what was generated and why. Engineers can intervene, correct, and re-run at any stage.

TOOLING BUDGET

Three engineers, 60 days to build v1: one pattern (CRUD + UI) across two products. Architecture must support incremental addition of patterns and products.

A CONCRETE EXAMPLE

This feature shipped last month on the Payments squad. One senior engineer, 3.5 days, with Claude Code:

“

Add a “Payment Methods” management page to the merchant dashboard. Merchants should be able to add, edit, delete, and set a default payment method. Each payment method has a type (credit card, ACH, wire), a label, and credentials stored via our Vault integration. List view with sorting and filtering, detail/edit form. Audit log every change. Gate behind `payment_methods:manage` .

Product brief, Payments squad

Time breakdown: **0.5 days** database migration and API endpoints. **0.5 days** Vault integration and encryption. **1 day** React components (list, form, confirmation modals). **0.5 days** permissions and audit logging. **0.5 days** tests. **0.5 days** environment config, feature flags, and PR review feedback. The coding—the part AI accelerated—was roughly a third. The rest was looking up how Athena handles

permissions, finding the right Vault client pattern from another service, matching existing table component conventions, and configuring staging.

In the factory model, the engineer describes this feature at the level of the product brief. The factory produces the migration, endpoints, components, tests, audit logging, permission gates, and config as a reviewable pull request. The engineer's job shifts from building to specifying and reviewing.

ASSIGNMENT

You are the VP of Engineering at Athena Digital. Design a software factory that transforms feature specifications into complete, deployable, pull-request-ready code across your product portfolio. Scope to a 60-day v1 (CRUD + UI pattern, two products), but design so the four areas below extend to all patterns and products.

Your proposal must address four areas:

- ☐ **Codebase Intelligence:** How does the factory learn each product's conventions well enough to generate code that looks like a team member wrote it? What is indexed, what is retrieved at generation time, what is baked into prompts? How do you handle seven repos with different languages and frameworks without building seven separate systems? How does it stay current as codebases evolve?
- ☐ **Orchestration & Decomposition:** "Add payment methods management" decomposes into migration, API, frontend, tests, config—with dependencies (frontend needs the API contract; tests need the migration). How does the factory break a spec into ordered generation tasks? How does output from one stage feed the next? What happens when a mid-pipeline stage fails? How is this better than BMAD's sequential handoff or Ralph's multi-agent free-for-all?
- ☐ **Quality & Verification:** Output must pass CI on first run 80% of the time: syntactically valid, type-safe, pattern-consistent, and functionally correct. How do you build verification into the pipeline rather than bolting it on at the end? How do you close the gap between "code that compiles" and "code that is correct"?
- ☐ **Human Interface & Trust:** Engineers will reject a black box. What does the engineer see at each stage? How do they intervene when the factory makes a wrong decision? How do you build trust so engineers move from "I review every line" to "I review architecture choices and spot-check implementation"?

Your deliverable should include:

- ☐ A **system architecture diagram** showing every component (specification parser, codebase indexer, plan generator, code generators, verification pipeline, review interface) and data flows between them.

- ☐ A **worked example** showing how the Payment Methods feature flows through your factory, stage by stage, from product brief to final pull request. Include intermediate artifacts at each stage.

- ☐ A **codebase intelligence strategy**: what you index, how you retrieve context, and how the factory handles cold-start on a codebase it has never seen.

- ☐ An **orchestration model** defining task decomposition, ordering, execution, dependency handling, and mid-pipeline failure recovery.

- ☐ A **scope table** separating v1 (60-day) deliverables from later phases, with rationale for each deferral.

Reference artifacts:

INPUT: FEATURE SPECIFICATION

```
{
  "product": "payments-dashboard",
  "pattern": "crud_ui",
  "entity": "PaymentMethod",
  "fields": [
    {"name": "type", "kind": "enum", "values": ["credit_card", "ach", "wire"]},
    {"name": "label", "kind": "string"},
    {"name": "is_default", "kind": "boolean"},
    {"name": "vault_ref", "kind": "string", "sensitive": true}
  ],
  "permissions": "payment_methods:manage",
  "audit": true,
  "integrations": ["vault"],
  "ui": {"list": true, "detail": true, "form": true}
}
```

OUTPUT: GENERATION PLAN

```
{
  "plan_id": "pf-20260315-001",
  "stages": [
    {"id": "migrate", "type": "db_migration", "depends_on": [], "status":
"pending"},
    {"id": "api", "type": "rest_endpoints", "depends_on": ["migrate"], "status":
"pending"},
    {"id": "frontend", "type": "react_components", "depends_on": ["api"],
```

```
"status": "pending"},
  {"id": "tests", "type": "test_suite", "depends_on": ["api", "frontend"],
"status": "pending"},
  {"id": "config", "type": "feature_flags_and_permissions", "depends_on": [],
"status": "pending"}
],
"estimated_files": 14,
"requires_human_approval_before": ["migrate"]
}
```

Be specific. BMAD and Ralph prove the concept is viable but optimize for different contexts. Your factory must work for a multi-product company with existing codebases and engineers who need to trust the output. The hard problem is not getting an LLM to generate code—it is getting it to generate the right code, in the right place, in the right style, with the right dependencies. Design for that.



SF-2026-02 · Rev. March 2026