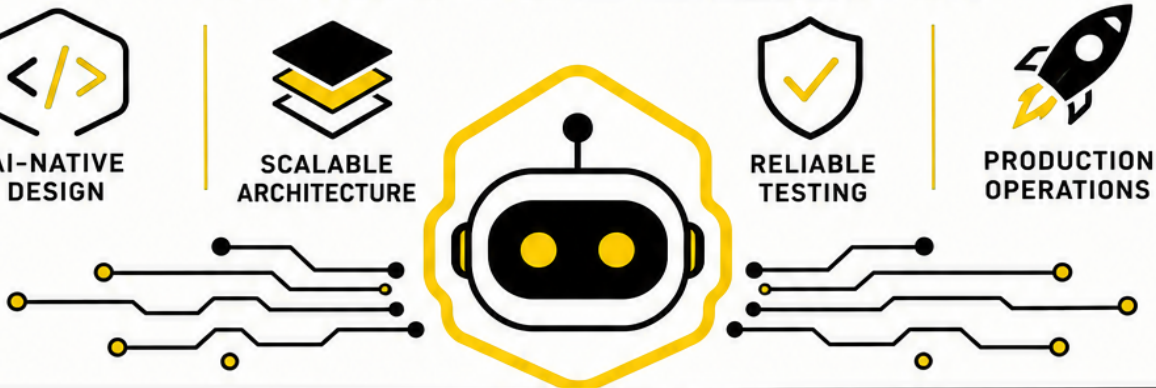
ENGINEERING SOFTWARE FOR AI CODING AGENTS

A PRACTICAL GUIDE TO BUILDING AI-NATIVE CODEBASES THAT SCALE



YURI ALEKSOV

Engineering Software for AI Coding Agents

A Practical Guide to Building AI-Native Codebases That Scale

Steve Publications

This book is available at
https://leanpub.com/engineeringsoftwareforaicodingagents

This version was published on 2026-07-31

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Practical Guide to Building AI-Native Codebases That Scale

About this book: You already know how to build software. Now you need to learn how to build software that works well when your pair programmer is an AI agent. This book teaches professional engineers how to design, organize, test, and operate codebases optimized for tools like GitHub Copilot, Claude Code, Cursor, and other autonomous development assistants. Through two running projects that evolve from requirements to production deployment, you will learn the architectural patterns, workflows, and engineering practices that turn AI coding agents from unpredictable novelty into reliable teammates. No filler, no exercises, no placeholder code. Only actionable technical guidance grounded in real-world experience and current research.

First edition, July 2026.

How to Use the Code Examples

All code in this book is provided inline as complete, runnable examples. Each example includes:

- All necessary imports and type definitions so you can copy and execute it directly
- Environment variable requirements clearly stated where applicable
- Dependencies listed (e.g., "requires: express@4.18+, pg@8.11+, zod@3.23+")
- Build and run instructions for multi-file examples

For the two running projects (TaskFlow and StreamProcessor), code evolves across chapters. Each chapter builds on previous work, and later chapters reference types and modules defined earlier. To follow along:

- Copy code from each chapter into a local project following the directory structures shown in Chapter 2
- Types and interfaces are always provided inline when first introduced
- When a later chapter references an earlier type (e.g., `TaskRepository` in Chapter 8), that type was fully defined in its original chapter (Chapter 3). Scroll back or search for it.

If you want to create your own companion repository, use the directory structures from Chapter 2 as your starting point and add code from each subsequent chapter. The cumulative result is a fully functional TaskFlow API and StreamProcessor service that demonstrate all patterns covered in this book.

Introduction: The New Pair Programmer

In March 2026, a senior engineer at a mid-sized fintech company asked her team to implement a new payment reconciliation feature. She wrote a detailed specification, handed it to Claude Code, and went to lunch. When she returned, the agent had generated three hundred lines of code across four files, created tests, and opened a pull request. The code compiled. The tests passed. But when QA reviewed it, they found that the agent had silently changed how timestamps were stored in the database, breaking compatibility with an existing reporting service.

This is not an outlier story. It is the baseline experience of AI-assisted development in 2026.

You have probably seen the marketing: developers shipping code three times faster, entire features implemented in afternoons, junior engineers becoming productive overnight. You have also seen the reality: subtle bugs that pass tests, architectural drift that accumulates silently, security vulnerabilities introduced at machine speed, and codebases that become harder to understand because no single person wrote them.

The gap between promise and reality exists because most teams are using AI coding agents with software engineered for human developers. We treat these tools as smarter autocompletion or faster interns. That is a mistake. AI agents are not just another tool in your workflow. They are a fundamentally different kind of collaborator, one that reasons about code the way humans do not, fails in ways humans do not, and requires software structured differently than what we have built for decades.

This book is about closing that gap. It teaches you how to engineer software specifically for AI coding agents: repository structures they can navigate, documentation they actually read, testing strategies that catch their mistakes, and workflows that amplify their strengths while constraining their weaknesses. The goal is not to make your code "AI-friendly" as an afterthought. The goal is to treat AI-friendliness as a first-class engineering concern, on par with performance, security, and maintainability.

By the end of this book, you will know how to:

- Design repository architectures where AI agents understand context and make correct cross-file changes
- Write code that communicates intent clearly to both humans and machines
- Build testing pipelines that validate AI-generated code as rigorously as human-written code
- Implement CI/CD workflows optimized for high-velocity, agent-assisted development
- Use specification-driven development to give AI agents precise contracts to implement
- Secure your systems against the unique vulnerabilities AI-generated code introduces
- Deploy and operate production services built with heavy AI assistance
- Avoid the anti-patterns that turn AI productivity gains into technical debt

Throughout the book, two running projects demonstrate these concepts in practice. TaskFlow is a TypeScript/Node.js task management API with PostgreSQL, representing a typical web backend. StreamProcessor is a Go-based real-time event processing service, representing a high-performance data pipeline. Both projects evolve from initial requirements through architecture, implementation, testing, deployment, and maintenance. You will see exactly how each chapter's concepts apply to real code.

This book assumes you are already a professional software engineer. You know how to write code, use version control, write tests, and deploy applications. You do not need to be an AI expert. If you have used GitHub Copilot, Claude Code, Cursor, or similar tools and wondered how to get more out of them while shipping less garbage into production, this book is for you.

Let us begin.

Chapter 1: The AI-Native Codebase

The first step in engineering software for AI coding agents is understanding what changed when these tools arrived, and why traditional practices need adaptation. This chapter establishes the foundation for everything that follows.

What Changed When AI Joined the Team

Before AI coding assistants, software development had a predictable information flow. A human developer read requirements, understood context from code and documentation, wrote implementation, ran tests, and submitted changes. Another human reviewed those changes. The codebase evolved incrementally, with each change traceable to a person who understood its intent.

AI coding agents disrupt this flow in three fundamental ways.

First, they generate code at a scale humans cannot match. A developer might write fifty lines of code per hour on average, depending on complexity. An AI agent can generate thousands of lines in minutes, across multiple files and services simultaneously. This speed is not merely an amplification of existing productivity. It changes the economics of software development in ways that break assumptions built into our tooling, processes, and organizational structures.

Second, they reason about code differently than humans. Large language models do not understand code the way programmers do. They have no mental model of system architecture, no intuitive sense of trade-offs, no accumulated wisdom from past mistakes. Instead, they predict what code is likely to appear given patterns in their training data. This produces code that often looks correct on the surface but contains subtle logical errors, security vulnerabilities, and architectural inconsistencies that would be obvious to a human who understood the broader context.

Third, they create code without ownership. When a human writes code, they feel responsibility for it. They think about how it will break, how others

will read it, what happens when requirements change. An AI agent has no such concerns. It generates code that satisfies the immediate prompt and moves on. The resulting codebase becomes a patchwork of contributions from an entity that does not care about long-term consequences.

These three changes (scale, reasoning, and ownership) mean that using AI coding agents effectively requires more than learning new prompts. It requires rethinking how we structure our software.

The evidence for this is not theoretical. A randomized controlled trial by METR (Model Evaluation & Threat Research) in July 2025 found that experienced open-source developers working on real issues in their own large repositories actually took nineteen percent longer when allowed to use AI tools compared to working without them. The same developers believed they had gained twenty percent productivity after the fact, revealing a significant perception gap. In their study, researchers paired sixteen developers with Cursor Pro and Claude 3.5/3.7 Sonnet, tracking work on 246 real issues across mature repositories averaging over one million lines of code. The conclusion was clear: AI tools did not make these experienced engineers faster on complex, real-world tasks. They made them slower, because the verification overhead and error correction consumed more time than the generation speed saved.

Other studies tell a similar story. Faros AI's July 2025 telemetry analysis of over ten thousand developers across 1,255 teams showed that while AI adoption increased individual task completion by approximately twenty-one percent and pull request volume by ninety-eight percent, code review time ballooned by ninety-one percent, average PR sizes grew by one hundred fifty-four percent, and bug counts rose nine percent. The 2025 DORA report (the industry standard for measuring delivery performance) found that while AI adoption reached ninety percent of developers and teams deployed more frequently, software delivery instability increased, with incidents per pull request rising significantly. Lead times dropped, but change failure rates did not improve proportionally, and developer burnout remained unchanged. Individual output went up, but organizational delivery quality did not keep pace. This pattern has been replicated across multiple studies and is now widely referred to as the "AI productivity paradox."

This is not an argument against AI coding agents. It is evidence that using them without adapting our engineering practices creates inefficiency, not productivity. The teams that actually benefit from AI are the ones that engineer

their codebases and workflows specifically for these tools. That is what this book teaches.

To understand why naive AI adoption fails, consider the full lifecycle of an AI-assisted change:

1. **Prompt iteration**: The developer spends time crafting and refining prompts. For non-trivial tasks, this often requires multiple rounds of back-and-forth as the agent misunderstands requirements or generates incomplete solutions.
2. **Code generation**: The agent produces code quickly, perhaps hundreds of lines across multiple files in seconds. This is where the speed advantage is most visible.
3. **Verification**: The developer must read and understand all generated code to verify correctness. This step is often underestimated. AI-generated code can look correct on the surface while containing subtle logical errors, security vulnerabilities, or architectural inconsistencies.
4. **Error correction**: When bugs are found (and they will be), the developer debugs and fixes them. Because AI agents generate code without deep understanding of the broader system, bugs may stem from incorrect assumptions about data flow, dependencies, or business rules.
5. **Code review**: The pull request enters human review. Reviewers must verify not only correctness but also architectural fit, security implications, and test coverage. Large AI-generated PRs are particularly taxing to review because changes span many files and the reasoning behind design choices is opaque.

Steps 1, 3, 4, and 5 are where time is consumed. The generation speed in step 2 does not offset the overhead of the surrounding steps unless the codebase and workflow are specifically designed to minimize verification effort. That is the core thesis of this book: AI-native engineering is about reducing the cost of verification through better architecture, clearer specifications, and automated validation.

How AI Coding Agents Actually Work

Understanding how AI coding agents function under the hood is essential to understanding how to work with them effectively. These are not magic boxes.

They are sophisticated applications of large language models with specific architectural patterns, constraints, and failure modes.

At their core, all modern AI coding agents follow a similar pipeline. The process begins with indexing: the agent builds a representation of your codebase by chunking files into embeddings and parsing them via abstract syntax trees and symbol maps. This creates a searchable index that lets the agent find relevant code quickly. Different tools use different approaches. Cursor builds embedding indexes for its custom IDE environment. Claude Code reads files on demand using tool calls, building context iteratively as it explores your codebase. GitHub Copilot primarily uses open-tab context and inline suggestions, with limited project-wide awareness in its agent mode.

Next comes context assembly: when you give the agent a task, it selects relevant files from its index using semantic search and symbol traversal, fitting them into its context window. Current frontier models have context windows ranging from 128,000 to over one million tokens. This sounds enormous, but a medium-sized codebase can easily exceed even the largest windows. A study by Chroma in 2025 tested eighteen leading LLMs and found that performance degrades as input grows, even on trivial tasks. The assumption that "just use a bigger context window" solves everything is empirically false.

The reasoning phase follows: the language model creates an edit plan based on the assembled context, conversation history, and system instructions. This is where architectural understanding matters. If your codebase has clear structure, explicit contracts, and consistent patterns, the agent's plan will be more accurate. If it is a tangled mess of implicit dependencies and scattered logic, the agent will produce plans that look reasonable but are wrong.

Finally, changes are applied via inline diffs or tool-based file rewrites. The agent edits files, creates new ones, deletes obsolete code. Modern agents can perform multi-file edits in a single operation, making cross-cutting changes that would require careful coordination from a human developer.

This pipeline reveals the agent's fundamental constraint: it only knows what you give it. Its understanding of your codebase is limited by the index quality, the context window size, and the clarity of your project structure. It cannot infer unstated requirements. It cannot read between the lines of vague specifications. It operates on explicit information, and when that information is missing or ambiguous, it fills gaps with plausible but incorrect assumptions.

This constraint explains the agent's most common failure modes:

- **Architecture drift:** The agent makes local optimizations that violate global architectural principles because it does not have the full picture.
- **Hallucinated imports:** The agent references modules, functions, or packages that do not exist, relying on patterns from its training data rather than your actual codebase.
- **Cascading edit errors:** A mistake in one file propagates through subsequent edits as the agent's internal model of the codebase becomes increasingly wrong.
- **Scope creep:** The agent takes a simple task and over-engineers it, adding complexity that was not requested because its training data associates certain patterns with "good code."

Understanding these failure modes is not academic. It directly informs how you should structure your code. Clear boundaries reduce architecture drift. Explicit dependencies prevent hallucinated imports. Small, focused changes limit cascading errors. Tight scoping instructions prevent over-engineering.

The Cost of AI-Unfriendly Codebases

An AI-unfriendly codebase is one that was not designed with AI agents as consumers. This describes most existing codebases and most newly started projects, because we have only recently begun to understand what "AI-friendly" actually means.

The costs of AI-unfriendliness are measurable and significant. The METR study identified five key factors contributing to the productivity loss: low AI reliability requiring constant double-checking, context management overhead, debugging complexity for AI-generated code, review bottlenecks from oversized PRs, and the cognitive load of switching between human and machine reasoning modes. Faros AI's broader telemetry data showed a similar pattern across enterprise teams: while individual developers completed more tasks, the organization-level delivery velocity did not improve because the additional review burden, larger PR sizes, and increased bug counts offset the generation speed gains.

These costs manifest in specific, predictable ways.

Increased review burden. When an agent generates code across multiple files, each file requires careful review. The reviewer must verify not only that

the code is correct but also that it fits within the broader architecture, respects existing patterns, and does not introduce subtle incompatibilities. A single AI-generated pull request might touch twenty files and require hours of review time. Without guardrails, code review becomes a bottleneck that negates any speed advantage from faster generation.

Silent architectural erosion. AI agents optimize for local correctness, not global consistency. Over time, this produces codebases where different parts follow different patterns, module boundaries blur, and the original architectural intent becomes obscured. This is not malicious or even careless. It is an inevitable consequence of a tool that does not understand architecture making incremental changes based on limited context. Without explicit constraints, your codebase drifts toward whatever patterns happen to be most common in the agent's training data, which may not align with your design decisions.

Security vulnerabilities at scale. Veracode's 2025 GenAI Code Security Report tested code generated by over one hundred LLMs across eighty real-world coding tasks spanning Java, JavaScript, Python, and C#. They found that forty-five percent of AI-generated code samples contained OWASP Top 10 vulnerabilities. The same study quantified vulnerability density as 2.74 times higher in AI-generated code than in human-written code. Java was particularly vulnerable, with a seventy-two percent failure rate on security tests. Veracode's Spring 2026 update found that despite improvements in model capabilities, the security pass rate had stalled at fifty-six percent, essentially unchanged from the previous year.

These numbers are not meant to scare you away from AI. They are meant to show that using AI without corresponding security practices is dangerous. An AI agent has no inherent security awareness. It will generate code with SQL injection vulnerabilities, hardcoded secrets, and insecure defaults if your prompts and codebase do not explicitly prevent it. The training data contains vast amounts of insecure code patterns alongside secure ones, and without explicit guidance, agents default to whatever pattern appears most common, which is not always the secure option.

Debugging complexity. When an AI agent writes code, the chain of reasoning behind design decisions is lost. You have a pull request description and some conversation history, but no developer who can explain why a particular approach was chosen. When bugs appear months later, debugging requires reverse-engineering logic that no living person fully understands. This creates what one team called "cognitive debt": the accumulated cost of understanding

code whose intent is not preserved in any accessible form.

The solution to these costs is not to stop using AI. It is to engineer codebases where AI agents are constrained and guided enough that their output is consistently correct, secure, and maintainable. That requires deliberate architectural decisions, explicit documentation, rigorous testing, and workflows designed for machine collaboration. The rest of this book shows you how.

Introducing Our Running Projects

To ground these concepts in practice, this book uses two running projects that evolve through every chapter. You will see requirements, architecture decisions, implementation code, tests, CI/CD configurations, and deployment artifacts. Both projects are designed to be realistic enough to illustrate real-world concerns but small enough to show completely.

TaskFlow is a task management API built with TypeScript and Node.js. It uses PostgreSQL for persistence and Express as the web framework. TaskFlow implements user authentication, task CRUD operations, project organization, and real-time updates via WebSockets. Throughout the book, you will see TaskFlow grow from a bare repository structure to a production-deployed service with full observability, security scanning, and automated testing.

TaskFlow represents the typical web backend: business logic, database interactions, API design, and user-facing features. It demonstrates how AI-native practices apply to applications where correctness matters, where bugs have real consequences for users, and where maintainability is critical because many developers will touch the code over time.

StreamProcessor is a real-time event processing service built in Go. It consumes events from Kafka, applies transformations and aggregations, and writes results to both PostgreSQL and Redis. StreamProcessor implements exactly-once processing semantics, backpressure handling, and health monitoring. It demonstrates how AI-native practices apply to high-performance systems where latency, throughput, and reliability are paramount.

StreamProcessor represents the infrastructure layer: data pipelines, message queues, caching, and distributed systems concerns. It shows how AI agents can assist with complex concurrency patterns, performance optimization, and operational code, and how to structure such systems so agent assistance is safe and effective.

Both projects share common principles that we will establish in this chapter and reinforce throughout the book:

- Explicit repository structure optimized for AI navigation
- Strong type systems as communication channels between humans and agents
- Comprehensive test suites that validate AI-generated code
- CI/CD pipelines with quality gates designed for agent workflows
- Documentation that serves both human readers and machine consumers
- Security practices that treat AI-generated code as untrusted

We will introduce each project incrementally. Chapter 2 begins with repository architecture, where you will see the initial directory structures, configuration files, and naming conventions that make these projects AI-native from day one. Subsequent chapters build on this foundation, adding layers of testing, deployment, observability, and operational rigor.

Before moving to Chapter 2, take a moment to reflect on your current projects. Where do AI agents struggle? What patterns in your codebase cause confusion or errors? The answers to those questions will help you understand the material in the chapters ahead, because every concept in this book addresses a real problem that engineers face when working with AI coding agents.

Chapter 2: Repository Architecture for AI Agents

Repository architecture is the foundation of AI-native software engineering. How you organize files, name directories, and structure dependencies determines whether an AI agent can understand your codebase or gets lost in noise. This chapter shows you how to design repositories that agents can navigate reliably.

Designing Directory Layouts AI Can Parse

AI coding agents do not explore codebases the way humans do. A human developer builds a mental model over time, understanding patterns, conventions, and implicit relationships through repeated exposure. An AI agent starts each session with no persistent memory of your project. It relies entirely on explicit structure: directory names, file names, and the content it can index and retrieve within its context window.

This means your directory layout is not just an organizational convenience for humans. It is a primary communication channel with AI agents. A clear, predictable structure reduces context noise, speeds up retrieval, and guides the agent toward correct architectural decisions.

The core principle is separation of concerns at the filesystem level. Each directory should have a single, obvious purpose that its name communicates unambiguously. Avoid generic names like `lib`, `utils`, or `common` that could contain anything. Use domain-specific names that tell an agent exactly what kind of code lives inside.

Let us look at the TaskFlow repository structure:

```
taskflow/
├── AGENTS.md
├── README.md
├── package.json
├── tsconfig.json
├── .eslintrc.cjs
├── .prettierrc
├── docker-compose.yml
├── Dockerfile
├── api/
│   ├── routes/
│   │   ├── tasks.ts
│   │   ├── projects.ts
│   │   └── auth.ts
│   ├── middleware/
│   │   ├── auth.ts
│   │   ├── validation.ts
│   │   └── error-handler.ts
│   ├── controllers/
│   │   ├── task-controller.ts
│   │   ├── project-controller.ts
│   │   └── auth-controller.ts
│   └── app.ts
├── domain/
│   ├── entities/
│   │   ├── task.ts
│   │   ├── project.ts
│   │   └── user.ts
│   ├── repositories/
│   │   ├── task-repository.ts
│   │   ├── project-repository.ts
│   │   └── user-repository.ts
│   └── services/
│       ├── task-service.ts
│       └── project-service.ts
├── infrastructure/
│   ├── database/
│   │   ├── postgres-client.ts
│   │   └── migrations/
│   ├── websocket/
│   │   └── server.ts
│   └── config/
│       └── env.ts
├── shared/
│   ├── types/
│   │   ├── api-types.ts
│   │   └── domain-types.ts
│   └── constants/
│       └── index.ts
├── test/
```

```
│   ├── unit/
│   │   ├── task-controller.test.ts
│   │   ├── task-service.test.ts
│   │   └── auth-middleware.test.ts
│   ├── integration/
│   │   ├── tasks-api.test.ts
│   │   └── projects-api.test.ts
│   └── fixtures/
│       └── seed-data.ts
└── scripts/
    ├── migrate.ts
    └── seed.ts
```

This structure makes several deliberate choices that benefit AI agents:

1. **Domain-aligned directories**: The `domain/` directory contains business logic. An agent looking to implement a new task-related feature knows exactly where to find existing patterns. The separation between `domain/entities`, `domain/repositories`, and `domain/services` follows domain-driven design principles, giving the agent a clear architectural template.
2. **Infrastructure isolation**: Database connections, WebSocket servers, and configuration live in `infrastructure/`. When an agent needs to add database support or configure environment variables, it does not have to search through business logic to find where infrastructure concerns are handled.
3. **API layer clarity**: The `api/` directory contains only web-facing code: routes, controllers, and middleware. An agent adding a new endpoint knows to create a route handler in `routes/`, a controller in `controllers/`, and potentially middleware in `middleware/`. The pattern is explicit.
4. **Test parity**: Tests mirror the source structure under `test/unit/` and `test/integration/`. When an agent modifies code, it can find the corresponding test file by following the same directory pattern. This encourages test-driven development because the relationship between code and tests is structurally obvious.
5. **No ambiguity**: There are no generic catch-all directories. Every directory has a specific purpose. An agent encountering this structure can make correct architectural decisions without guessing where code belongs.

Contrast this with a typical AI-unfriendly structure:

```
bad-example/
├── src/
│   ├── app.js
│   ├── db.js
│   ├── utils.js        <-- what goes here?
│   ├── helpers.js      <-- same as utils?
│   ├── controllers/
│   │   ├── task.js
│   │   ├── auth.js
│   │   └── websocket.js  <-- infrastructure mixed with business logic
│   └── models/
│       ├── Task.js
│       └── User.js
├── tests/
│   └── test.js         <-- all tests in one file?
└── README.md
```

In this structure, an AI agent faces ambiguity at every step. Should new utility code go in `utils.js` or `helpers.js`? Where does WebSocket handling belong? How are tests organized? The agent will make guesses based on patterns in its training data, which may not match your intended architecture. Over time, these guesses accumulate into architectural drift.

The StreamProcessor project follows similar principles with Go-specific conventions:

```
streamprocessor/
├── AGENTS.md
├── go.mod
├── go.sum
├── Makefile
├── Dockerfile
├── cmd/
│   └── streamprocessor/
│       └── main.go
├── internal/
│   ├── consumer/
│   │   ├── kafka-consumer.go
│   │   └── handler.go
│   ├── processor/
│   │   ├── aggregator.go
│   │   └── transformer.go
│   ├── producer/
│   │   └── output-producer.go
│   └── storage/
│       ├── postgres-store.go
│       └── redis-cache.go
```

```
├── pkg/
│   └── events/
│       ├── event-types.go
│       └── schemas.go
├── test/
│   ├── consumer_test.go
│   ├── aggregator_test.go
│   └── fixtures/
│       └── sample-events.json
└── config/
    └── config.yaml
```

Go's `internal/` and `pkg/` conventions naturally create clear boundaries. Code in `internal/` cannot be imported by external packages, giving the agent a structural constraint that enforces encapsulation. The `cmd/` directory clearly marks entry points. An agent adding a new consumer or processor knows exactly where to place its code.

Naming Conventions That Disambiguate Intent

Naming is the second most important architectural decision for AI agents, after directory structure. A consistent naming convention acts as a specification: it tells agents what code does, how it relates to other code, and what patterns to follow.

AI agents are pattern-matching systems. When they see a file named `task-controller.ts`, they understand that this file likely contains request handlers for task-related operations, following a controller pattern. When they see `task-repository.ts`, they understand it handles data access. This understanding comes from patterns in their training data, and you can leverage it by using names that align with established conventions.

The key is consistency within your project. You do not need to follow industry-standard naming exactly, but you must be internally consistent. Mixing patterns confuses agents and leads to architectural inconsistency.

Here are the naming conventions used in TaskFlow and StreamProcessor:

File naming:

- Use kebab-case for all files: `task-controller.ts`, not `TaskController.ts` or `taskController.ts`. This avoids case-

sensitivity issues across operating systems and is the convention most common in the training data of AI models.
- Include type suffixes that describe the file's role: `-controller.ts`, `-repository.ts`, `-service.ts`, `-middleware.ts`, `-test.ts`. These suffixes tell an agent what kind of code to expect.
- Avoid single-character or overly abbreviated names. `t.ts` tells an agent nothing. `task.ts` is clear.

Type and class naming:

- Use PascalCase for TypeScript interfaces, types, and classes: `Task`, `ProjectRepository`, `AuthMiddleware`. This aligns with TypeScript conventions that agents are heavily trained on.
- Prefix interfaces with descriptive names rather than the generic `I` prefix. `TaskRepository` is better than `ITaskRepository` in TypeScript, where the `I` prefix is largely a legacy from C#.
- Use type suffixes that match the role: `Repository`, `Service`, `Handler`, `Middleware`, `Controller`.

Function naming:

- Use camelCase for functions and methods.
- Start with verbs that describe the action: `createTask`, `findProjectById`, `validateRequest`, `handleConnection`.
- Include domain context in the name. `save()` is ambiguous. `saveTask()` or `persistTask()` is clear.

Variable naming:

- Use camelCase for variables.
- Be explicit about types when the type is not obvious from the name: `taskList` instead of `items`, `projectId` instead of `id`.
- Avoid single-letter variables except in trivial contexts like loop counters.

Let us see how these conventions apply in practice. Here is the TaskFlow task entity:

```typescript
// domain/entities/task.ts

import { z } from 'zod';

export const TaskStatus = z.enum(['pending', 'in_progress', 'completed',
↪  'cancelled']);
export type TaskStatus = z.infer<typeof TaskStatus>;

export interface Task {
  id: string;
  projectId: string;
  title: string;
  description: string | null;
  status: TaskStatus;
  assigneeId: string | null;
  priority: number;
  dueDate: Date | null;
  createdAt: Date;
  updatedAt: Date;
}

export const TaskSchema = z.object({
  id: z.string().uuid(),
  projectId: z.string().uuid(),
  title: z.string().min(1).max(200),
  description: z.string().max(5000).nullable(),
  status: TaskStatus,
  assigneeId: z.string().uuid().nullable(),
  priority: z.number().int().min(0).max(10),
  dueDate: z.date().nullable(),
  createdAt: z.date(),
  updatedAt: z.date(),
});

export function createTask(params: {
  projectId: string;
  title: string;
  description?: string | null;
  assigneeId?: string | null;
  priority?: number;
  dueDate?: Date | null;
}): Task {
  const now = new Date();
  return {
    id: crypto.randomUUID(),
    projectId: params.projectId,
    title: params.title,
    description: params.description ?? null,
    status: 'pending',
    assigneeId: params.assigneeId ?? null,
```

```
    priority: params.priority ?? 5,
    dueDate: params.dueDate ?? null,
    createdAt: now,
    updatedAt: now,
  };
}
```

This code demonstrates several conventions that help AI agents:

- The file name `task.ts` in the `entities/` directory tells the agent this defines the Task domain entity.
- The interface `Task` uses PascalCase and matches the file's purpose.
- The `TaskStatus` enum is defined explicitly with Zod, giving the agent a clear, type-safe representation of valid statuses.
- The `createTask` function name describes exactly what it does, with a typed parameter object that shows required and optional fields.
- The Zod schema provides validation logic that an agent can reference when writing API handlers or tests.

When an AI agent needs to create a new entity, such as a Comment type, it can follow this exact pattern. The naming conventions and structure provide a template the agent recognizes from its training data.

Monorepo vs Polyrepo for AI-Assisted Development

The monorepo versus polyrepo debate has raged for years. With AI coding agents as a new consumer of repository structure, the trade-offs shift in important ways. Understanding these shifts helps you make the right choice for your team.

A monorepo keeps multiple projects or services in a single Git repository. A polyrepo (or multi-repo) strategy maintains separate repositories for each project or service. Both approaches have valid use cases, and neither is universally superior. The decision depends on your organization's structure, your deployment model, and how your AI agents will interact with the code.

Monorepos offer advantages for AI agents:

- Full codebase context: An agent working in a monorepo can see all services and shared libraries simultaneously. It understands cross-service

dependencies, shared types, and common patterns without needing special configuration to access multiple repositories.
- Atomic changes: When a change affects multiple services, an agent can make all modifications in a single commit. This eliminates the coordination problems of distributed transactions across repositories. A feature that updates a database schema, modifies the backend API, and changes the frontend can be implemented atomically.
- Unified dependency management: Shared libraries are versioned internally, reducing the risk of incompatible versions across services. An agent updating a shared type sees all consumers at once and can update them consistently.

Monorepos also create challenges:

- Context noise: A large monorepo contains far more code than any AI agent can fit in its context window. Without careful scoping, an agent gets overwhelmed by irrelevant files, leading to slower responses and more errors.
- Blast radius: When an agent makes a mistake in a monorepo, that mistake can propagate across many services. A single bad commit can break builds for dozens of teams.
- Build complexity: Monorepos require sophisticated build systems like Nx, Turborepo, or Bazel to manage dependencies and parallelize builds. These tools add operational overhead.

Polyrepos offer different advantages:

- Natural scoping: Each repository is smaller, giving agents a clearer view of the relevant code. An agent working on a specific service sees only that service's code, reducing noise.
- Contained errors: A mistake in one repository does not affect others. The blast radius is limited by design.
- Team autonomy: Different teams can use different tools, languages, and workflows without coordination overhead.

Polyrepos create their own challenges:

- Cross-service changes are difficult: When a change requires updates across multiple services, an agent must work in multiple repositories sequentially. This creates distributed transaction problems: Service A is updated but Service B fails, leaving the system in an inconsistent state.
- Missing context: An agent working in one repository cannot easily see related code in other repositories. It may make changes that break consumers it cannot see.
- Dependency drift: Shared libraries versioned as external packages can diverge across services, creating incompatibilities that are hard to track.

The research is clear on how AI agents perform in each model. A study by Nazar Boyko in June 2026 concluded that "the biggest variable in how well an AI coding agent performs is whether you put everything in one repo or split it across many." The study found that monorepos give agents a structural advantage through full context and atomic changes, but only when combined with scoping discipline. A monorepo without boundary discipline becomes "a 400-package swamp" where agents drown in noise.

For TaskFlow and StreamProcessor, we will use a hybrid approach that captures the benefits of both models. Each service lives in its own repository (TaskFlow in `taskflow-api`, StreamProcessor in `streamprocessor`), but shared types and contracts live in a third repository (`shared-contracts`) that both services depend on. This is sometimes called a "federated monorepo" or "mono-repo-lite" pattern.

This approach works well when:

- Services are independently deployable with different teams or release cadences.
- Cross-service changes are relatively infrequent compared to within-service changes.
- You want clear ownership boundaries but still need shared contracts.

For tightly coupled services where cross-cutting changes are common, a true monorepo with proper scoping is better. For completely independent services with no shared dependencies, separate polyrepos work fine.

The key insight is that AI agents change the trade-off calculation. In a human-only world, the cost of cross-repository context switching was a major factor favoring monorepos. AI agents can reduce that cost by indexing multiple

repositories and providing unified search across them. This makes polyrepos more viable than they were before, but only if you give agents the tools to navigate across repository boundaries.

Cross-Repository Context and Dependency Graphs

When your code spans multiple repositories, AI agents need explicit guidance to understand relationships between them. Without this guidance, agents working in one repository cannot see dependencies in others, leading to breaking changes that go unnoticed.

There are three primary mechanisms for providing cross-repository context:

Shared contract repositories: The most straightforward approach is a dedicated repository for shared types, interfaces, and schemas. Both TaskFlow and StreamProcessor depend on a `shared-contracts` repository that defines common event types, API response formats, and domain concepts. When an agent modifies a shared type, it can see all consumers because they import from the same source.

Here is an example of how this works in practice:

```
shared-contracts/
├── AGENTS.md
├── package.json
├── src/
│   ├── events/
│   │   ├── task-events.ts
│   │   └── schema-version.ts
│   ├── api/
│   │   ├── response-types.ts
│   │   └── error-types.ts
│   └── types/
│       └── common.ts
└── test/
    └── contract-validation.test.ts
```

The `task-events.ts` file defines events that StreamProcessor consumes:

```typescript
// shared-contracts/src/events/task-events.ts

import { z } from 'zod';

export const TaskCreatedEvent = z.object({
  eventType: z.literal('task.created'),
  version: z.literal('1.0.0'),
  timestamp: z.string().datetime(),
  data: z.object({
    taskId: z.string().uuid(),
    projectId: z.string().uuid(),
    title: z.string(),
    assigneeId: z.string().uuid().nullable(),
    priority: z.number().int(),
  }),
});

export type TaskCreatedEvent = z.infer<typeof TaskCreatedEvent>;

export const TaskUpdatedEvent = z.object({
  eventType: z.literal('task.updated'),
  version: z.literal('1.0.0'),
  timestamp: z.string().datetime(),
  data: z.object({
    taskId: z.string().uuid(),
    projectId: z.string().uuid(),
    changes: z.array(z.enum(['title', 'description', 'status', 'priority',
    ↪  'assigneeId', 'dueDate'])),
  }),
});

export type TaskUpdatedEvent = z.infer<typeof TaskUpdatedEvent>;

export const TaskCompletedEvent = z.object({
  eventType: z.literal('task.completed'),
  version: z.literal('1.0.0'),
  timestamp: z.string().datetime(),
  data: z.object({
    taskId: z.string().uuid(),
    projectId: z.string().uuid(),
    completedBy: z.string().uuid(),
  }),
});

export type TaskCompletedEvent = z.infer<typeof TaskCompletedEvent>;

export const TaskCancelledEvent = z.object({
  eventType: z.literal('task.cancelled'),
  version: z.literal('1.0.0'),
  timestamp: z.string().datetime(),
```

```
  data: z.object({
    taskId: z.string().uuid(),
    projectId: z.string().uuid(),
    cancelledBy: z.string().uuid(),
    reason: z.string().max(500),
  }),
});

export type TaskCancelledEvent = z.infer<typeof TaskCancelledEvent>;
```

When an agent in the StreamProcessor repository needs to understand what events it should consume, it reads this file. When an agent in TaskFlow needs to emit a new event type, it adds the schema here first. The shared contract becomes the source of truth that both agents reference.

Dependency graphs and affected builds: Tools like Nx and Turborepo maintain dependency graphs that track which packages depend on which others. When an agent modifies code in one package, these tools can determine which other packages are affected and run their tests automatically. This is essential for catching cross-package regressions that an agent might not notice.

In a polyrepo setup, you can achieve similar results with tooling that indexes multiple repositories. Nx Polygraph, for example, provides cross-repository dependency graphs that help AI agents understand how changes in one repo affect others.

AGENTS.md files across repositories: Every repository should have an AGENTS.md file that describes its purpose, dependencies, and how it relates to other repositories. This gives agents navigating between repos the context they need to make correct decisions. We will cover AGENTS.md in detail in Chapter 5, but here is a brief example:

```
## StreamProcessor

## Purpose
Real-time event processing service. Consumes events from Kafka, applies
↪  transformations, writes to PostgreSQL and Redis.

## Dependencies
- Depends on shared-contracts for event type definitions
- Consumes events produced by taskflow-api
- Does not directly communicate with other services

## Architecture
- consumer/: Kafka consumers and message handlers
- processor/: Business logic for event transformation
- producer/: Output producers for downstream systems
- storage/: Database and cache clients

## Build and Test
- `make build` - Compile the service
- `make test` - Run all tests
- `make lint` - Run static analysis

## Cross-Repo Notes
When modifying event handling:
1. Check shared-contracts for latest event schemas
2. Update consumer code to handle new fields
3. Add integration tests with sample events from test/fixtures/
```

This file tells an agent exactly what this repository does, what it depends on, and how to work safely within it. When combined with similar files in other repositories, agents can build a mental model of the entire system.

The combination of shared contracts, dependency graphs, and agent instruction files creates a navigation layer that lets AI agents work effectively across repository boundaries. Without this layer, polyrepos become silos where agents make changes in isolation, breaking integrations that they cannot see.

Complete Infrastructure Setup for Running Projects

To ensure you can run both TaskFlow and StreamProcessor locally as complete, integrated systems, here is the full infrastructure configuration. This docker-compose setup provides PostgreSQL, Redis, and Kafka, all the services both projects depend on. You can copy this directly and execute it to have a working environment.

docker-compose.yml for local development:

```yaml
# docker-compose.yml - Complete local development environment
# Supports both TaskFlow (TypeScript/Node.js) and StreamProcessor (Go)
# Run with: docker compose up -d

version: '3.8'

services:
  # ===== DATABASE SERVICES =====

  postgres:
    image: postgres:16-alpine
    container_name: taskflow-postgres
    environment:
      POSTGRES_USER: taskflow
      POSTGRES_PASSWORD: taskflow_dev_password
      POSTGRES_DB: taskflow
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./scripts/init-db.sql:/docker-entrypoint-initdb.d/init-db.sql
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U taskflow -d taskflow"]
      interval: 10s
      timeout: 5s
      retries: 5

  # ===== CACHE SERVICES =====

  redis:
    image: redis:7-alpine
    container_name: taskflow-redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  # ===== MESSAGE QUEUE (for StreamProcessor) =====

  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: taskflow-zookeeper
```

```yaml
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: taskflow-kafka
    depends_on:
      zookeeper:
        condition: service_started
    ports:
      - "9092:9092"
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      ↪ PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS:
      ↪ PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
    healthcheck:
      test: ["CMD-SHELL", "kafka-broker-api-versions --bootstrap-server
      ↪ localhost:9092"]
      interval: 15s
      timeout: 10s
      retries: 5

  # ===== TOOLS AND UTILITIES =====

  kafka-ui:
    image: provectuslabs/kafka-ui:latest
    container_name: taskflow-kafka-ui
    depends_on:
      - kafka
    ports:
      - "8090:8080"
    environment:
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:29092
      KAFKA_CLUSTERS_0_ZOOKEEPER: zookeeper:2181

volumes:
  postgres_data:
  redis_data:
```

Database initialization script (scripts/init-db.sql):

```sql
-- scripts/init-db.sql
-- Run automatically on first PostgreSQL container start
-- Creates all tables for TaskFlow and StreamProcessor

-- ===== TASKFLOW TABLES =====

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(20) NOT NULL DEFAULT 'member' CHECK (role IN ('admin',
    ↪  'member')),
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Projects table
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(200) NOT NULL,
    description TEXT,
    created_by UUID NOT NULL REFERENCES users(id),
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Project membership (many-to-many between users and projects)
CREATE TABLE project_members (
    user_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    project_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
    role VARCHAR(20) NOT NULL DEFAULT 'member' CHECK (role IN ('admin',
    ↪  'member')),
    joined_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    PRIMARY KEY (user_id, project_id)
);

-- Tasks table
CREATE TABLE tasks (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    project_id UUID NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
    title VARCHAR(200) NOT NULL,
    description TEXT,
    status VARCHAR(20) NOT NULL DEFAULT 'pending'
        CHECK (status IN ('pending', 'in_progress', 'completed', 'cancelled')),
    assignee_id UUID REFERENCES users(id),
```

```
    priority INTEGER NOT NULL DEFAULT 5 CHECK (priority >= 0 AND priority <= 10),
    due_date TIMESTAMPTZ,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Indexes for common query patterns
CREATE INDEX idx_tasks_project_id ON tasks(project_id);
CREATE INDEX idx_tasks_assignee_id ON tasks(assignee_id);
CREATE INDEX idx_tasks_status ON tasks(status);
CREATE INDEX idx_tasks_project_status ON tasks(project_id, status);
CREATE INDEX idx_tasks_due_date ON tasks(due_date) WHERE due_date IS NOT NULL;
CREATE INDEX idx_project_members_project_id ON project_members(project_id);

-- ===== STREAMPROCESSOR TABLES =====

-- Aggregated event results (written by StreamProcessor)
CREATE TABLE event_aggregations (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    project_id UUID NOT NULL REFERENCES projects(id),
    window_start TIMESTAMPTZ NOT NULL,
    window_end TIMESTAMPTZ NOT NULL,
    count INTEGER NOT NULL DEFAULT 0,
    total_priority DOUBLE PRECISION NOT NULL DEFAULT 0,
    avg_priority DOUBLE PRECISION NOT NULL DEFAULT 0,
    max_priority INTEGER NOT NULL DEFAULT 0,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Index for time-range queries on aggregations
CREATE INDEX idx_aggregations_project_window ON event_aggregations(project_id, window_start, window_end);

-- ===== SAMPLE DATA (for development/testing) =====

-- Create a test admin user (password: "testpassword123" - hashed with bcrypt round 10)
INSERT INTO users (id, email, password_hash, role) VALUES
    ('550e8400-e29b-41d4-a716-446655440000', 'admin@example.com',
     '$2b$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy', 'admin');

-- Create a test member user
INSERT INTO users (id, email, password_hash, role) VALUES
    ('550e8400-e29b-41d4-a716-446655440001', 'member@example.com',
     '$2b$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy', 'member');

-- Create a test project
INSERT INTO projects (id, name, description, created_by) VALUES
    ('550e8400-e29b-41d4-a716-446655440010', 'Demo Project',
```

```
     'A sample project for testing TaskFlow functionality',
     '550e8400-e29b-41d4-a716-446655440000');

-- Add users to the project
INSERT INTO project_members (user_id, project_id, role) VALUES
    ('550e8400-e29b-41d4-a716-446655440000',
    ↪  '550e8400-e29b-41d4-a716-446655440010', 'admin'),
    ('550e8400-e29b-41d4-a716-446655440001',
    ↪  '550e8400-e29b-41d4-a716-446655440010', 'member');

-- Create sample tasks
INSERT INTO tasks (id, project_id, title, description, status, assignee_id,
↪  priority) VALUES
    ('550e8400-e29b-41d4-a716-446655440020',
    ↪  '550e8400-e29b-41d4-a716-446655440010',
     'Set up project repository', 'Initialize Git repo and CI/CD pipeline',
     ↪  'completed',
     '550e8400-e29b-41d4-a716-446655440000', 8),
    ('550e8400-e29b-41d4-a716-446655440021',
    ↪  '550e8400-e29b-41d4-a716-446655440010',
     'Implement authentication', 'Add JWT-based auth with refresh tokens',
     ↪  'in_progress',
     '550e8400-e29b-41d4-a716-446655440001', 9),
    ('550e8400-e29b-41d4-a716-446655440022',
    ↪  '550e8400-e29b-41d4-a716-446655440010',
     'Build task CRUD API', 'Create REST endpoints for task management',
     ↪  'pending',
     NULL, 7);
```

TaskFlow environment file (.env):

```
# TaskFlow .env - Copy to your local directory and modify as needed
NODE_ENV=development
PORT=3000

# Database
DATABASE_URL=postgresql://taskflow:taskflow_dev_password@localhost:5432/taskf
↪  low

# JWT Configuration
JWT_SECRET=dev-secret-key-change-in-production-2026
JWT_EXPIRES_IN=1h
JWT_REFRESH_EXPIRES_IN=7d

# Redis
REDIS_URL=redis://localhost:6379

# Kafka (for event emission by TaskFlow)
KAFKA_BROKERS=localhost:9092
```

```
KAFKA_TOPIC_TASK_EVENTS=taskflow.task-events

# CORS
ALLOWED_ORIGINS=http://localhost:5173,http://localhost:3000

# Logging
LOG_LEVEL=debug
```

StreamProcessor environment file (.env):

```
# StreamProcessor .env - Copy to your local directory and modify as needed

# Kafka Configuration
KAFKA_BROKERS=localhost:9092
KAFKA_TOPIC_CONSUME=taskflow.task-events
KAFKA_GROUP_ID=streamprocessor-aggregator
KAFKA_AUTO_OFFSET_RESET=earliest

# Database
DATABASE_URL=postgresql://taskflow:taskflow_dev_password@localhost:5432/taskf
↪ low

# Redis
REDIS_URL=redis://localhost:6379

# Aggregation Configuration
AGGREGATION_WINDOW_SECONDS=60
AGGREGATION_MAX_EVENTS=100
AGGREGATION_FLUSH_ON_CLOSE=true

# Logging
LOG_LEVEL=info

# Health Check
HEALTH_PORT=8081
```

Complete Go Makefile for StreamProcessor:

```
# streamprocessor/Makefile

BINARY_NAME=streamprocessor
GO_FILES=$(shell find . -name "*.go" -not -path "./vendor/*")

.PHONY: all build run test clean lint format deps

all: build

build:
    @echo "Building $(BINARY_NAME)..."
    go build -o $(BINARY_NAME) ./cmd/streamprocessor

run: build
    @echo "Running $(BINARY_NAME)..."
    ./$(BINARY_NAME)

test:
    @echo "Running tests..."
    go test -v -race -coverprofile=coverage.out ./...

test-coverage: test
    @echo "Generating coverage report..."
    go tool cover -html=coverage.out -o coverage.html
    @echo "Coverage report saved to coverage.html"

lint:
    @echo "Running linters..."
    golangci-lint run ./...

format:
    @echo "Formatting code..."
    gofmt -s -w $(GO_FILES)
    goimports -w $(GO_FILES)

clean:
    @echo "Cleaning build artifacts..."
    rm -f $(BINARY_NAME) coverage.out coverage.html

deps:
    @echo "Downloading dependencies..."
    go mod download
    go mod tidy

# Create Kafka topic for development
create-topic:
    @echo "Creating Kafka topic taskflow.task-events..."
    docker exec -i taskflow-kafka kafka-topics --create \
        --bootstrap-server localhost:9092 \
        --replication-factor 1 \
```

```
        --partitions 3 \
        --topic taskflow.task-events \
        --if-not-exists

# List Kafka topics
list-topics:
    docker exec -i taskflow-kafka kafka-topics --list --bootstrap-server
    ↪ localhost:9092

# Consume messages from topic (for debugging)
consume:
    docker exec -it taskflow-kafka kafka-console-consumer \
        --bootstrap-server localhost:9092 \
        --topic taskflow.task-events \
        --from-beginning

# Produce test message to topic (for debugging)
produce-test:
    @echo
    ↪ '{"event_type":"task.created","version":"1.0.0","timestamp":"2026-07-31T12:00:00Z","
    ↪ Task","assignee_id":null,"priority":5}}' | \
    docker exec -i taskflow-kafka kafka-console-producer \
        --bootstrap-server localhost:9092 \
        --topic taskflow.task-events
```

With these files in place, you can start the full environment and run both projects:

```
# 1. Start infrastructure
docker compose up -d

# 2. Verify services are healthy
docker compose ps

# 3. Create Kafka topic for StreamProcessor
cd streamprocessor && make create-topic

# 4. Run TaskFlow (from taskflow-api directory)
cd ../taskflow-api
npm ci
npm run migrate    # If using migrations instead of init script
npm run dev

# 5. Run StreamProcessor (in another terminal)
cd ../streamprocessor
make build
./streamprocessor

# Both services are now running and connected:
```

```
# - TaskFlow API at http://localhost:3000/api/v1
# - StreamProcessor consuming from Kafka, writing aggregations to PostgreSQL
# - Kafka UI at http://localhost:8090 for monitoring topics
```

This complete infrastructure setup ensures that every code example in this book can be executed in a real environment. The docker-compose file is designed to work with both projects simultaneously, allowing you to test the full event-driven flow: TaskFlow creates tasks and emits events to Kafka, StreamProcessor consumes those events and writes aggregated results back to PostgreSQL.

In the next chapter, we will move from repository structure to code design itself: how to write code that AI agents can understand, generate correctly, and modify safely.

Chapter 3: Prompt-Aware Code Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Self-Documenting Code Beyond the Buzzword

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Type Systems as AI Communication Channels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Function Granularity and Single Responsibility for Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Error Handling Patterns That Guide Agent Reasoning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 4: Specification-Driven Development with AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Writing Specifications AI Can Execute Against

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Interface Design for Machine Readability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Contract Testing and AI Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

From Requirements to Code via Structured Specs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 5: Documentation Strategies That AI Actually Uses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

The Three Audiences of Technical Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

README Architecture for Human and Machine Readers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Inline Comments That Guide Agent Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Living Documentation and Knowledge Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 6: Testing Practices for AI-Assisted Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

TDD When Your Pair Programmer Is an LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Test Structure That Catches Agent Hallucinations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Property-Based Testing as a Safety Net

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Snapshot Testing and Visual Regression for AI Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 7: CI/CD Pipelines Optimized for AI Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Pipeline Stages for AI Code Quality Assurance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Automated Code Review as a Gate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Branching Strategies for High-Velocity AI Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Rollback Procedures When AI Makes Mistakes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 8: Modular Architecture and Dependency Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Bounded Contexts and Module Boundaries for AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Dependency Injection as an Agent Constraint Mechanism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Complete StreamProcessor Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Versioning Strategies That Protect Against AI Regressions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Managing Transitive Dependencies in AI Workflows

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 9: Debugging, Observability, and Incident Response with AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Reading Stack Traces When AI Wrote the Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Logging Strategies That Help Agents Diagnose Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Distributed Tracing in AI-Assisted Microservices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Incident Response Playbooks With AI Assistants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 10: Security Engineering for AI-Assisted Development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Vulnerabilities Common in AI-Generated Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Secure Coding Standards Enforced by AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Secrets Management When Agents Have Broad Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Supply Chain Security and Dependency Scanning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 11: Agent Orchestration, MCP, and Tool Use

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

The Model Context Protocol Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Designing Tools AI Agents Can Call Reliably

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Multi-Agent Workflows and Task Decomposition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Agent Sandboxing and Permission Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 12: RAG, Knowledge Bases, and Context Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Context Window Economics and Chunking Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Building Internal Knowledge Bases for Your Team

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

RAG Patterns for Code Understanding and Generation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Vector Databases and Semantic Search for Projects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

RAG Tuning for Production Code Retrieval

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 13: Infrastructure as Code and Cloud Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Writing Infrastructure Code AI Can Evolve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Containerization Patterns for Agent-Assisted DevOps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Kubernetes Manifests That Agents Can Maintain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Cloud Configuration as Machine-Readable Specs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 14: Production Operations and Performance Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Metrics That Matter When AI Writes the Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Performance Profiling with AI Assistance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Capacity Planning for AI-Driven Feature Velocity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Long-Term Maintenance of AI-Assisted Codebases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Chapter 15: Anti-Patterns, Pitfalls, and Lessons Learned

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Over-Reliance on AI Without Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Context Pollution and Prompt Leakage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Architectural Drift When Agents Refactor Freely

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Team Dynamics and Knowledge Loss with Heavy AI Use

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Other Common Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

Lessons Learned from Early Adopters

Conclusion: The AI-Native Engineer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/engineeringsoftwareforaicodingagents.