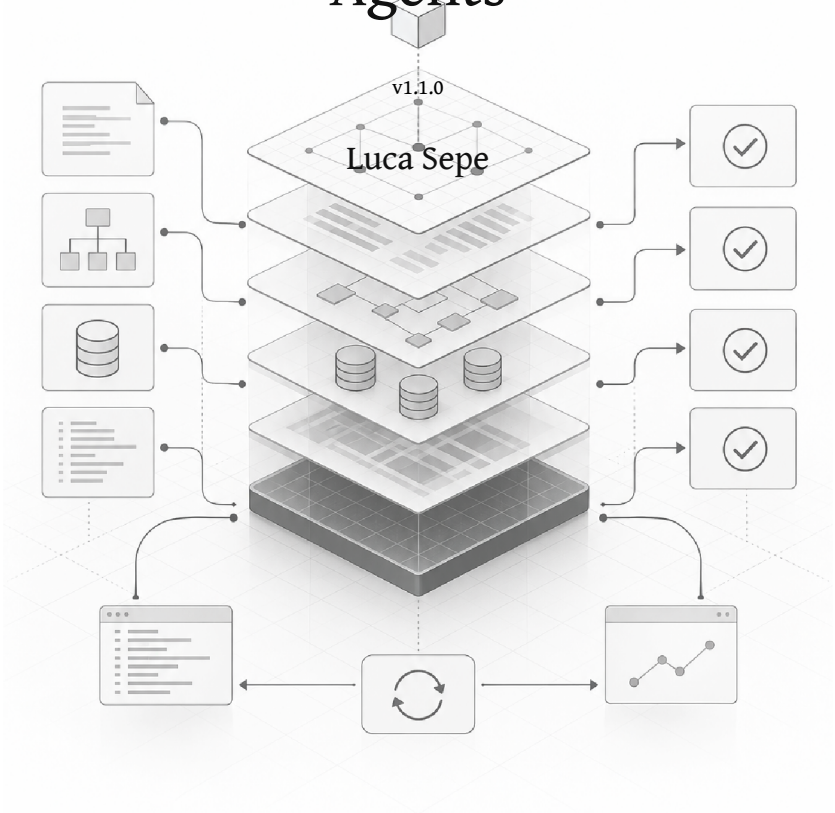


Context Driven Development

(Production Field Manuals)

Beyond Prompt Engineering: A Systematic Approach to Building Software with AI Agents



Preface

Most failures in AI assisted software development are not failures of wording. They are failures of the decision environment.

When a developer asks an agent to "*build a webhook handler*", the agent will return plausible code. The code compiles. It may even pass a basic happy path test.

However, under the hood, the implementation might detect duplicates in memory, publish directly to a message broker before the database transaction commits, hide critical SQL bottlenecks behind an ORM, or claim idempotency without a proper concurrency test.

The typical response to these flaws is to refine the prompt: add more detail, paste examples, insert warnings, and enforce style rules.

Eventually, the prompt transforms into a dense, chaotic container of product intent, architecture, requirements, temporary instructions, and engineering anxiety. It might work for a single task, but it will never scale into a reliable engineering system.

Context Driven Development starts from a different premise:

Reliable AI assisted software development depends less on finding the perfect prompt and more on engineering the context, constraints, feedback loops, and verification mechanisms that surround the agent.

This book defines Context Driven Development, or *CDD*, as a structured approach to software engineering where AI agents operate within explicit, authoritative, and verifiable boundaries.

This environment integrates product intent, requirements, architectural choices, engineering constraints, and task specific instructions.

This definition is intentionally provisional.

CDD is not a novel replacement for traditional software engineering. Instead, it deliberately overlaps with requirements engineering, Domain Driven Design, Test Driven Development, Behavior Driven Development, specification driven development, architecture decision records, docs as code, policy as code, static analysis, CI, and software architecture governance.

The practical value of CDD lies in composing these proven methodologies into a cohesive operational model specifically designed for workflows mediated by AI coding agents.

This book is written for software developers, tech leads, architects, technical founders, and engineering managers.

It assumes a foundational understanding of repositories, testing frameworks, APIs, databases, and CI/CD pipelines. It is designed for readers seeking a disciplined, repeatable way to deploy coding agents in production environments.

The methodology is entirely independent of any specific AI tool. You can apply CDD using terminal-based agents, IDE assistants, Cursor rules, Aider conventions, hosted enterprise agents, or even manual chat workflows.

The core method remains unchanged: identify the authoritative source of truth, select the minimum sufficient context for the task, execute within explicit constraints, verify the output, and feed discoveries back into durable documentation.

Ultimately, the intended transformation is simple:

Before:

= I need a better prompt so the AI generates better code.

After:

= I need to engineer the environment in which the agent makes decisions.

The Method in One Page

Context Driven Development treats software development as constrained decision making.

An AI coding agent needs more than just a prompt.

It must know what matters most, what must remain unchanged, and which source of truth prevails when documents conflict. It needs to know which decisions are already settled, which assumptions are unsafe, and exactly how success will be verified.

The core CDD loop operates as follows:

```
business intent
-> product context
-> requirements
-> architecture
-> engineering constraints
-> bounded task
-> agent execution
-> implementation
-> verification
-> feedback
-> context evolution
```

CDD is not defined by a specific directory layout.

A small project might only need a single SPEC.md and a TASK.md. A larger system may require product documents, requirements files, architecture decision records (ADRs), module local policies, tests, static analysis checks, and CI gates.

The documents themselves are not the goal. They are simply the vehicle used to make critical decisions explicit, authoritative, scoped, and verifiable.

The working distinction used throughout the book is:

Prompt Engineering
= how a request is expressed

Context Engineering
= what information and instructions are
made available

Context Driven Development
= how software engineering artifacts, decisions,
constraints, tasks, tools, and feedback loops
are organized so agents can operate reliably

This distinction is useful, but not sacred. In practice, boundaries blur. A prompt may carry task context. A repository file may act as a standing instruction. A test can serve as both verification and executable context.

Ultimately, CDD is concerned with the design of the entire system.

If You Are Starting From a Mega Prompt

Many readers will arrive here with a familiar workflow:

1. Write a long prompt.
2. Paste requirements, style rules, warnings, examples, and the current task.
3. Ask the agent to implement.
4. Hope it reads the important parts in the right way.

CDD does not ask you to abandon the agent. It asks you to stop forcing a single prompt to carry the weight of the entire system.

The first practical step is simple: deconstruct the mega prompt.

Move stable operating rules into an agent contract. Place product behavior within requirements. Document durable technical choices in architecture decisions. Delegate recurring engineering rules to policies. Finally, isolate the immediate work in a dedicated task file.

By doing this, the prompt becomes lightweight again:

Work on this task. Read the selected context. Respect these constraints. Run these checks. Report what was verified.

This is the practical essence of CDD.

The workflow allows the reader to systematically move through this sequence:

1. Define the business intent.
2. Extract observable requirements.
3. Record architecture decisions to avoid reinventing the wheel.
4. Establish engineering constraints that apply across tasks.
5. Define a single bounded task.
6. Select the exact context the agent must read for that task.
7. Provide the agent with a concise launch instruction.
8. Verify the output.
9. Integrate durable discoveries back into the appropriate source of truth.

Case Study: Starting from a Business Problem

The case study is a webhook intake service.

The context we build for it is mostly independent from the programming language. Requirements, architecture decisions, idempotency rules, transaction boundaries, outbox behavior, and verification expectations would matter in Go, Java, Python, C#, Rust, or any other production language.

For the implementation sketches, this book uses Go. That gives the example a concrete shape without making Go part of the method.

A companion repository for this case study is available at:

<https://github.com/lucasepe/webhook-intake>

The repository is not required to follow the chapter. It exists so readers can inspect the context files, run the example, and experiment with the same agent handoff described in the book.

The service receives events from a third party provider, validates them, stores them durably, rejects duplicates, and schedules accepted events for asynchronous processing through a broker.

The provider could be Stripe, GitHub, Shopify, or any external system that delivers signed HTTP webhooks.

The system is intentionally small:

```
HTTP request
-> signature validation
-> event identity extraction
-> duplicate protection
-> durable storage
-> outbox record
-> async publisher
-> broker
```

This case is better for CDD than a large business workflow because the logic is easy to explain but hard to fake correctly.

A naive implementation may work in a demo and fail under concurrent delivery, retries, or broker failure.

The business intent is:

```
External provider events must be accepted reliably
```

and processed at least once without allowing duplicate provider events to create duplicate internal work.

The engineering problem is:

The provider may retry the same event many times.
Retries may arrive concurrently.
The service must treat provider event IDs as idempotency keys, persist accepted events, and enqueue processing work without losing accepted events when the broker is unavailable.

This is a good CDD example because the agent's freedom must be tightly constrained.

The correct solution depends less on clever code and more on explicit context:

- the service must implement the Idempotent Consumer pattern
- deduplication must be enforced by the database, not only by memory or a pre check
- event persistence and enqueue intent must be committed atomically
- the broker must not be the source of truth for accepted events
- raw SQL is required
- concurrency behavior must be tested
- completion must report what was actually verified

Why Not Just Ask for a Webhook Handler?

A weak prompt might be:

Build a Go service that receives Stripe webhooks and sends them to Redis.

A plausible agent response may:

- parse JSON
- validate a header superficially
- insert an event row
- publish directly to Redis

- add a unit test for one request

That is not enough. It misses the core failure modes:

- two identical events arrive at the same time
- the process crashes after saving the event but before publishing
- the broker is down
- duplicate detection happens in application memory
- the code checks for duplicates with SELECT and then INSERT, creating a race
- an ORM hides the exact SQL and transaction boundaries
- the completion report says "*idempotent*" without evidence

CDD turns the vague request into a constrained engineering task.

From Mega Prompt to Method

A developer who has learned to work with agents may try to fix the weak prompt by making it much longer:

```
You are a senior Go engineer.

Build a production ready webhook intake
service for provider events.

Use Go and net/http.
Validate signatures.
Make the service idempotent.
Use the Idempotent Consumer pattern.
Use a transactional outbox.
Do not use an ORM.
Use raw SQL for performance.
Avoid duplicate processing.
Do not publish directly to Redis or RabbitMQ
from the HTTP handler.
Make sure concurrency is handled correctly.
Write tests.
Simulate 100 identical requests in parallel.
Follow clean architecture.
Keep the code maintainable.
Report what you changed.
```

This prompt is not foolish. Most of its content is useful.

The problem is that it asks one message to do too many jobs.

Some lines are product requirements. Some are architecture decisions. Some are engineering policies. Some are task scope. Some are verification expectations. Some are vague preferences. Some are instructions for how the agent should behave.

CDD keeps the useful information, but gives each piece a better home:

```
You are a senior Go engineer.
  -> usually unnecessary, or part of agent
      operating guidance

Validate signatures.
Avoid duplicate processing.
  -> requirements

Use Idempotent Consumer.
Use transactional outbox.
  -> architecture decisions

Do not use an ORM.
Use raw SQL.
  -> engineering persistence policy

Do not publish directly from the HTTP handler.
```

-> requirement and architecture constraint

Simulate 100 identical requests in parallel.

-> verification plan

Build this slice now.

-> task contract

After this decomposition, the agent still receives a prompt. It is just smaller and less fragile:

Work on `tasks/2026-09-13-webhook-intake.md`.

Before editing, read every file listed in Relevant Context.

Inspect the listed modules.

Report the requirements, constraints, and verification steps you found.

If anything is missing or conflicting, stop before implementing.

The important change is not that the prompt became shorter.

The important change is that the prompt stopped being the only source of truth.

Where Tooling Fits

You can practice CDD with Cursor rules, Aider conventions, an IDE assistant, a hosted coding agent, issue templates, or a manual chat workflow.

The tool is not where the method lives. The method lives in the separation of concerns:

durable rules
-> repository context

current work
-> task contract

selected context
-> Relevant Context

agent handoff
-> prompt or bundle

confidence
-> verification evidence

The Central Pattern

The case study uses two patterns together:

Idempotent Consumer

The system records which provider events it has accepted and treats repeated delivery of the same provider event ID as a duplicate.

Transactional Outbox

The system records the intent to publish work in the same database transaction that accepts the event. A separate dispatcher publishes outbox rows to the broker.

The combined flow is:

```
receive webhook
-> validate signature
-> begin transaction
-> insert inbound event using
    unique(provider, provider_event_id)
-> if duplicate, commit no new work and
    return success
-> insert outbox message in same transaction
-> commit
-> dispatcher publishes pending outbox messages
```

The unique constraint is not an implementation detail. It is a correctness constraint. Without it, the concurrency test should fail.

Product Context Is Still Present

Even technical infrastructure has product intent. The product context here is small:

Customers depend on provider events to trigger internal workflows.
Provider retries are normal and must not create duplicate work.
The service should prefer accepting an event durably over processing it synchronously during the HTTP request.

This context explains why the service returns success for duplicates. It also explains why direct broker publication inside the request path is not sufficient.

What the Case Study Will Demonstrate

The case study will focus on method rather than finished code:

- how to write product context for an infrastructure component
- how to express reliability requirements as observable obligations
- how to turn architectural patterns into constraints
- how to forbid implementation shortcuts that agents commonly take
- how to design a task contract
- how to define a verification loop that exposes concurrency mistakes
- how implementation feedback updates context

Code appears only where it clarifies a constraint or failure mode.

The book is not trying to teach Go webhook programming. It is trying to teach how to build the context that makes the right implementation shape more likely, and the wrong one easier to catch.

That is why this case study works: the verification loop can punish the wrong architecture, not merely check that code exists.