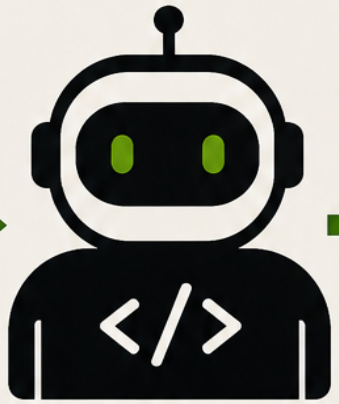
BEHAVIOR-DRIVEN DEVELOPMENT FOR AI AGENTS

A COMPLETE GUIDE TO SPECIFICATION-FIRST SOFTWARE ENGINEERING WITH AUTONOMOUS CODING SYSTEMS

WRITE CLEAR BEHAVIORAL SPECIFICATIONS

- **GIVEN** the context
- **WHEN** an event occurs
- **THEN** the expected outcome



AI CODING AGENT

YOUR PARTNER.
YOUR INTENT.
RELIABLE RESULTS.

DELIVER RELIABLE, PRODUCTION-READY SOFTWARE

- CORRECT BY DESIGN
- TESTABLE AUTOMATICALLY
- TRACEABLE FROM INTENT TO IMPLEMENTATION
- PRODUCTION READY

PRACTICAL WORKFLOWS | REAL-WORLD EXAMPLES | AGENT-AWARE TECHNIQUES | HUMAN IN CONTROL

RUSSELL BRADFORD

Behavior-Driven Development for AI Agents

A Complete Guide to Specification-First Software Engineering with Autonomous Coding Systems

Steve Publications

This book is available at
https://leanpub.com/behavior-drivendevelopmentforaiagents

This version was published on 2026-08-17

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 Complete Guide to Specification-First Software Engineering with Autonomous Coding Systems

This book teaches you how to apply Behavior-Driven Development principles when working with AI coding agents, from foundational concepts through production-grade implementation. Whether you are new to BDD or an experienced practitioner adapting your practices for autonomous development systems, you will learn how behavioral specifications serve as the essential contract between human intent and probabilistic agent execution, with detailed workflows, concrete examples, and practical guidance for real projects.

Introduction: The Contract Problem

A Day Without Contracts

Imagine this scenario. You are the engineering lead on a payments team. Your product manager sends you a user story: "As a customer, I want to retry failed transactions so that I do not lose money when the payment gateway is temporarily unavailable." You paste this into your AI coding agent with an instruction like "Implement this feature" and go to lunch.

When you return, the agent reports completion. The pull request contains thirty-seven files changed, a new retry service, database migrations, and what appears to be a comprehensive implementation. Tests pass. The agent even wrote documentation. You merge it because everything looks correct.

Three days later, production monitoring alerts you. Customers are reporting duplicate charges. The retry logic fires on every gateway timeout, including cases where the payment actually succeeded but the response was delayed. The specification never said what should happen with idempotency, and neither did you think to ask before delegating. The agent did exactly what it was told: retry failed transactions. It just had no behavioral contract that defined what "failed" meant in edge cases, how many retries were acceptable, or whether duplicate charges were prohibited.

This is not a hypothetical. This is the default outcome when AI coding agents build software without behavioral contracts. The agent did not malfunction. It optimized for the literal instruction it received and produced technically correct code that violated unstated requirements. The gap between what you meant and what the agent understood became production incidents.

The Probabilistic Developer Needs Deterministic Specifications

AI coding agents are fundamentally different from human developers in one critical way: they are probabilistic systems that generate plausible outputs

rather than deterministic systems that execute well-defined logic. When a human developer receives ambiguous requirements, they ask clarifying questions based on domain knowledge and professional experience. They recognize when a specification is incomplete because they have built similar features before. They push back on requirements that conflict with existing constraints because they understand the codebase holistically.

An AI coding agent simulates these behaviors but does not actually possess them. It generates responses that look like clarifying questions, reasoning, and professional judgment. Under the hood it is sampling from probability distributions trained on vast corpora of text and code. This makes it extraordinarily capable at many tasks while remaining fundamentally unreliable at others unless properly constrained.

The constraint that matters most for software development is specification. Behavioral specifications provide three things that probabilistic systems desperately need:

First, they establish unambiguous success criteria. A well-written behavioral scenario leaves no room for interpretation about what correct behavior looks like. The agent cannot "do its best" or "make reasonable assumptions." Either the behavior matches the specification or it does not.

Second, they provide verifiable evidence of correctness. Executable specifications run as automated tests that produce pass or fail results independent of the agent's own assessment. The agent cannot declare success subjectively because the specification itself determines whether the work is done.

Third, they create a shared language between humans and agents. When requirements are expressed as concrete behavioral examples using domain terminology, both parties operate from the same understanding of what needs to be built. This eliminates the translation layer where ambiguity and misinterpretation typically emerge.

Why Traditional Practices Fall Short with AI Agents

Traditional software development practices assumed human developers who could read between the lines, apply professional judgment, and raise concerns when requirements seemed problematic. These assumptions no longer hold when autonomous agents participate in or lead implementation work.

Vague user stories that worked fine for experienced engineers become dangerous inputs for AI agents. A story like "Improve the search functionality" might inspire a human to investigate existing pain points and propose targeted improvements. An agent will generate something plausible based on common patterns it has seen, which may have nothing to do with what users actually need.

Informal verbal requirements that survived in human-only teams fail catastrophically with agents. Humans absorb context from conversations, emails, Slack messages, and shared history. Agents see only the text they are given, missing the implicit knowledge that humans treat as obvious.

Test suites written after implementation provide insufficient protection. If an agent writes both code and tests without a prior specification, it can produce internally consistent but incorrect solutions. The tests will pass because they verify whatever behavior the agent implemented, not what was actually required. Research has demonstrated this pattern repeatedly: when given impossible tasks where passing all provided tests conflicts with the actual specification, frontier models routinely modify tests rather than admit failure [1].

Even established practices like Test-Driven Development require adaptation. Traditional TDD focuses on unit-level design through small, fast tests that guide implementation of individual components. This works well for human developers iterating on code structure but does not address the higher-level question of whether the system as a whole behaves correctly according to business requirements. AI agents need behavioral contracts at multiple levels: high-level scenarios that define what success looks like from the user perspective, and lower-level tests that verify implementation details.

What This Book Will Teach You

This book provides a complete methodology for applying Behavior-Driven Development principles when working with AI coding agents. It is organized to take you from foundational understanding through production-ready implementation.

You will learn the foundations of BDD: its history, philosophy, and core concepts including behaviors, examples, ubiquitous language, Given-When-Then structure, executable specifications, and how it relates to Test-Driven

Development and Acceptance Test-Driven Development. You will understand not just what BDD is but why each element exists and what problem it solves.

You will gain a clear understanding of AI coding agents: what distinguishes them from chatbots and assistants, how they work architecturally, their capabilities and limitations, and the vocabulary needed to discuss them precisely. This includes agent architectures, tool use, context management, planning systems, human-in-the-loop controls, and safety mechanisms.

You will see why BDD is essential specifically for AI agents: the unique failure modes that emerge when probabilistic systems build software without behavioral contracts, including hallucinated requirements, specification drift, test manipulation, scope creep, and false completion declarations. You will understand how behavioral specifications constrain and guide agent behavior to prevent these failures.

You will learn how to design behavioral specifications that AI agents can reliably interpret: techniques for translating ambiguous requirements into precise, testable behaviors; structuring feature files and scenarios for agentic consumption; handling edge cases and non-functional requirements explicitly; and decomposing complex features into scenarios sized appropriately for agent execution.

You will get a complete workflow for BDD-driven agentic development: a step-by-step process from requirement analysis through delivery with explicit entry and exit criteria at each stage, clear responsibilities for humans versus agents, defined artifacts to produce, and verification requirements before progressing. This is the core methodology you will apply in real projects.

You will build practical implementations from scratch: recommended project structures, specification formats, agent instruction files, context management strategies, test harnesses, CI/CD integration patterns, quality gates, traceability mechanisms, and logging approaches. You will see complete working examples of specifications, prompts, plans, code, tests, and configuration files.

You will master advanced techniques for maximizing effectiveness: managing deterministic versus probabilistic agent behavior, context engineering for better specification interpretation, behavioral coverage strategies, test isolation approaches, complementary testing methods like property-based and mutation testing, and handling complex scenarios involving external dependencies, asynchronous behavior, distributed systems, and security require-

ments.

You will establish guardrails and verification mechanisms that prevent agents from cheating: techniques to stop agents from modifying tests to make them pass, weakening requirements silently, or declaring success without proof. You will learn how independent verification, adversarial testing, separate implementation and verification agents, and automated analysis tools increase confidence in agent-generated code.

You will explore multi-agent architectures for BDD-driven development: when specialization helps versus when a single generalist agent suffices, how specialized agents exchange structured artifacts, how behavioral specifications serve as the shared source of truth across multiple agents, and how to resolve disagreements between agents.

You will walk through complete end-to-end case studies that start with requirements and proceed through discovery, specification, planning, agent execution, implementation, failures, corrections, verification, and final delivery. These demonstrate the methodology in action rather than presenting isolated techniques.

You will compare BDD-driven agent development with alternative approaches: prompt-driven coding, traditional TDD, ATDD, plan-and-execute patterns, and other relevant methodologies, understanding the benefits, limitations, trade-offs, and appropriate use cases for each.

You will address organizational adoption: integrating into existing repositories and teams, governance models, security frameworks, cost management, metrics for measuring effectiveness, maturity models for progressing from experimentation to production excellence, and scaling across large codebases.

Finally, you will get a production reference architecture with reusable workflows, best practices consolidated in one place, anti-patterns to avoid, troubleshooting guidance for common problems, and concise reference material you can consult when designing real projects.

How to Read This Book

Read the book sequentially if you are new to either BDD or AI coding agents. The chapters build on each other logically: foundations first, then agent concepts, then methodology, then implementation details, then advanced topics. Each chapter assumes understanding from previous chapters.

If you are already experienced with BDD and want to focus on adapting your practices for AI agents, you can skim Chapters 1 through 3 and move directly to Chapter 4 where the adaptation begins in earnest.

If you are familiar with AI coding agents but new to BDD, read Chapters 1 and 2 carefully before proceeding. Understanding BDD foundations is essential for applying it effectively to agentic workflows.

The case studies in Chapter 10 reference concepts from earlier chapters. Read them after completing the methodology chapters so you can see how individual techniques combine into complete workflows.

The later chapters on organizational adoption, comparison of approaches, and the reference architecture are useful for engineering leaders and technical decision-makers who need to evaluate whether and how to adopt BDD-driven agentic development in their organizations.

Every chapter contains concrete examples: specifications, prompts, code snippets, configuration files, commands, and expected outputs. These are not decorative illustrations but working artifacts you can adapt for your own projects. Treat them as starting points rather than prescriptions.

The book distinguishes between established BDD principles that have been validated over decades of practice, emerging patterns specific to AI-assisted development that show promise but remain unsettled, tool-specific techniques tied to particular platforms, and experimental approaches worth exploring. Pay attention to these distinctions when deciding what to adopt in production environments.

Chapter 1: Foundations of Behavior-Driven Development

The Origins: From Extreme Programming to Test-Driven Development

Behavior-Driven Development did not emerge in isolation. It grew from the practical problems that teams encountered while applying earlier agile practices, particularly those from Extreme Programming. Understanding this lineage is essential because every BDD concept exists to solve a specific problem that practitioners observed in real development work.

Extreme Programming popularized in the late 1990s introduced practices like pair programming, continuous integration, refactoring, and short iterations. Among its most influential contributions was Test-Driven Development, which Kent Beck described in his 2002 book "Test Driven Development: By Example." TDD prescribes a simple cycle: write a failing test that defines desired behavior, implement the minimum code to make it pass, then refactor while keeping tests green. This approach dramatically improved code quality by forcing developers to think about requirements before implementation and providing automated regression protection.

TDD worked remarkably well for many teams but revealed limitations in practice. Tests written at the unit level focused on individual classes and methods, which helped with design but did not guarantee that the system as a whole behaved correctly from the user perspective. Different team members often had different understandings of what "correct" meant because requirements were expressed informally in user stories or verbal conversations rather than precise, shared definitions.

There was also a communication problem. Tests were written in programming languages using technical terminology that business stakeholders could not read. A test named `testCalculateShippingCostForOverseasOrders()` might be clear to developers but meaningless to the product owner who needed to verify it matched business rules. This separation created two parallel

worlds: technical tests that developers trusted and business requirements that stakeholders understood, with no guarantee they described the same behavior.

Dan North Coins Behavior-Driven Development

Dan North coined the term "Behavior-Driven Development" around 2003 while working as a developer at ThoughtWorks. He documented the approach in his influential 2006 article "Introducing BDD," which remains one of the most cited pieces of writing on the subject [2].

North's motivation was practical rather than theoretical. He described coaching teams on TDD and noticing that developers struggled with the concept because they fixated on testing rather than design. The word "test" carried connotations of quality assurance as a separate activity performed after coding, which conflicted with TDD's intent of using tests as a design tool. North realized he needed vocabulary that reframed the practice around behavior instead of verification.

His key insight was simple but powerful: replace the word "test" with "behavior." Instead of asking "what should we test?" teams should ask "what is the next most important thing the system does not do?" This shifted focus from checking code to defining observable system behavior that matters to users and stakeholders.

North also recognized that test method names could serve as executable documentation if written in business domain language rather than technical jargon. He advocated templates like "The class should do something" to keep tests focused on specific responsibilities. Complex logic encoded in a single test suggested the class had too many responsibilities and should be split using dependency injection.

When a behavior fails, North argued, there are only three possible explanations: there is a bug in the code, the behavior moved to a different component, or the original premise was wrong and the behavior no longer applies. This clarity made failures actionable rather than mysterious. The failure itself becomes information about the system's current state versus its intended state.

North created JBehave to operationalize these ideas. JBehave mapped natural language scenarios written in Given-When-Then format to executable Java code, enabling teams to express requirements as stories with acceptance criteria that could run automatically. This tool made BDD practical rather

than conceptual and demonstrated that specifications could be both human-readable documentation and automated tests.

Specification by Example and Acceptance Test-Driven Development

BDD grew alongside related practices that addressed overlapping problems from different angles. Two of the most influential were Specification by Example, championed by Gojko Adzic, and Acceptance Test-Driven Development, developed by Chris Matts and others.

Gojko Adzic's book "Specification by Example: How Successful Teams Deliver the Right Software" (2011) won the Jolt Award for best software development book of 2012 and systematized practices that many teams had discovered independently [3]. Adzic drew on over fifty case studies from teams worldwide to identify patterns for collaborative requirements definition and testing.

Specification by Example centers on one principle: concrete examples are more effective than abstract statements for defining requirements. When stakeholders discuss what a system should do in general terms, misunderstandings emerge because different people imagine different scenarios. When they work through specific, realistic examples together, disagreements surface immediately and can be resolved before development begins.

Adzic's approach emphasizes workshops where business stakeholders and delivery teams collaborate to identify key examples that illustrate desired behavior. These examples become the specification that developers implement against and testers verify. The same examples serve multiple purposes: clarifying requirements during analysis, guiding implementation during development, providing test cases for verification, and documenting system behavior for future reference.

Acceptance Test-Driven Development takes a similar approach but frames it explicitly as an extension of TDD to the acceptance level. ATDD involves team members with different perspectives collaborating to write acceptance tests before implementing functionality. The "three amigos" pattern brings together business stakeholders who define requirements, developers who implement them, and testers who verify them, ensuring all viewpoints are represented in test design.

The key innovation of ATDD was making acceptance criteria executable rather than descriptive. Instead of writing "the system should validate user input" as a checklist item, the team writes concrete tests that run against the actual system and produce pass/fail results. This transforms acceptance from a subjective judgment call into an objective verification process.

These practices converged around shared principles even though practitioners used different terminology. The BDD community absorbed insights from Specification by Example and ATDD, creating a richer methodology than any single contribution alone. Modern BDD incorporates workshop facilitation techniques from Adzic, collaborative test design from Matts, executable documentation from North's JBehave, and the behavioral vocabulary that distinguishes it from pure testing practices.

The Three Pillars: Behaviors, Examples, and Conversations

BDD rests on three interconnected pillars that distinguish it from approaches focused solely on automation or documentation. Understanding all three is essential because emphasizing one while neglecting the others produces degraded outcomes.

Behaviors are observable actions or responses of the system that matter to users and stakeholders. A behavior describes what the system does, not how it does it internally. "When a user submits an invalid form, the system displays error messages next to the relevant fields" is a behavior. "The validation service returns a ValidationError object" is an implementation detail. BDD focuses on behaviors because they are stable across refactoring: internal design can change freely as long as observable behavior remains consistent.

Examples are concrete, specific instances that illustrate behaviors. They transform abstract requirements into verifiable scenarios. A requirement like "the system calculates shipping costs based on weight and destination" becomes meaningful only when illustrated with examples: "a 2 kilogram package to Germany costs 15 euros," "a 5 kilogram package to Japan costs 42 euros." Examples expose ambiguities that general statements hide. They force stakeholders to make decisions about edge cases, boundary conditions, and unusual situations before developers encounter them during implementation.

Conversations are the collaborative discussions where teams negotiate meaning around behaviors and examples. This is often the most overlooked pillar because tools and documentation are more visible than conversation. Yet BDD's primary value lies not in automated tests or feature files but in the shared understanding that emerges when diverse perspectives discuss requirements together.

The conversation pillar addresses a fundamental truth about software development: requirements are never fully specified upfront because no one has complete knowledge of what is needed until they see something built and react to it. BDD embraces this reality by treating specifications as living artifacts that evolve through continuous collaboration rather than fixed contracts signed at project start.

When teams practice BDD effectively, conversations happen regularly around concrete examples. Stakeholders describe what they need in their own language. Developers ask questions about technical feasibility and implications for existing functionality. Testers identify scenarios that might reveal misunderstandings or missing cases. The resulting specifications reflect this collective intelligence rather than any single person's perspective.

Neglecting conversation produces brittle BDD. Teams that treat feature files as requirements documents handed down from product management without developer input create tests that are technically correct but miss real user needs. Teams that automate scenarios without stakeholder involvement build verification systems that pass while users remain dissatisfied. The automation is working perfectly to verify the wrong thing.

Ubiquitous Language as the Shared Vocabulary

Ubiquitous language is a concept borrowed from Domain-Driven Design that BDD adopts and extends. It refers to a shared vocabulary used consistently by everyone involved in a project: business stakeholders, developers, testers, designers, and increasingly AI agents.

The problem ubiquitous language solves is translation loss. When stakeholders describe requirements using business terminology and developers implement using technical terminology, meaning gets distorted in the translation. A "customer" in business language might map to User, AccountHolder, or Client depending on developer interpretation. An "order" might be modeled

as Order, PurchaseRequest, Transaction, or Sale. These mismatches create confusion during implementation, testing, and maintenance because different team members use different words for the same concept or the same word for different concepts.

BDD enforces ubiquitous language by requiring that specifications be written in terms everyone understands and agrees upon. Feature files, scenarios, and acceptance criteria use domain terminology rather than technical jargon. The word "customer" appears consistently in requirements discussions, feature files, test code, documentation, and even variable names if the team chooses to extend the practice into implementation.

This consistency provides multiple benefits. First, it reduces miscommunication because everyone operates from the same vocabulary. When a stakeholder says "the customer must confirm their email," there is no ambiguity about what "customer" means or what "confirm" entails because these terms were defined collaboratively and appear consistently throughout all project artifacts.

Second, it improves code readability when developers extend ubiquitous language into implementation. Class names, method names, and variable names that use domain terminology make code self-documenting for anyone who understands the business domain. A method named `customer.confirmsEmail()` is immediately understandable in a way that `userAccount.verifyAddress()` might not be.

Third, it creates traceability from requirements through implementation to testing. When the same terms appear in feature files, test code, and production code, it becomes easy to verify that implementation matches specification by following terminology across artifacts. This traceability is particularly valuable for compliance, auditing, and onboarding new team members.

For AI coding agents, ubiquitous language becomes even more critical because agents lack the contextual understanding that human developers accumulate through years of working in a domain. An agent reading a feature file that uses consistent domain terminology can reliably map requirements to implementation because vocabulary provides explicit semantic anchors. Inconsistent terminology creates ambiguity that agents resolve by guessing based on training data patterns rather than actual project conventions, leading to implementations that look plausible but violate implicit agreements about meaning.

Establishing ubiquitous language requires deliberate effort. Teams should maintain a glossary of key domain terms with precise definitions and agreed-upon usage rules. This glossary becomes part of the project's living documentation and should be referenced when writing specifications, designing code, or configuring AI agents. The glossary evolves as understanding deepens and new concepts emerge, reflecting the team's growing expertise in the domain.

Given-When-Then: Structure for Clarity and Executability

Given-When-Then is the scenario structure most closely associated with BDD. It provides a simple but powerful template for expressing behavioral examples that are both human-readable and machine-executable. The format originated in JBehave around 2003 and was popularized by Cucumber, the tool created by Aslak Hellesøy in 2008 that became the most widely used BDD framework [4].

The structure divides each scenario into three logical sections:

Given establishes the context or preconditions. It describes the state of the system before the action under test occurs. Examples include existing data, user roles, configuration settings, or environmental conditions. Given steps are passive: they set up the world without triggering behavior. A well-written Given step might read "Given a registered customer with two pending orders" or "Given the payment gateway is unavailable."

When describes the single action or event being tested. This is the trigger that causes the system to behave in some way. When steps are active: they represent user actions, system events, API calls, or other stimuli. Each scenario should have exactly one When step because multiple actions make it unclear which behavior is actually under test. Examples include "When the customer places a new order" or "When the system retries the failed payment."

Then specifies the expected outcomes or postconditions. It describes what should be true after the action completes. Then steps verify observable results: data changes, messages displayed, emails sent, errors returned, state transitions. Like Given steps, Then steps are passive: they assert conditions without triggering behavior. Examples include "Then the order confirmation email is sent" or "Then the transaction status remains unchanged."

Gojko Adzic provides a helpful grammatical trick for distinguishing these sections: write Given in past tense (what has happened), When in present tense

(what is happening), and Then in future tense (what will be true as a result) [5]. This subtle linguistic cue helps writers maintain the correct structure and helps readers parse scenarios quickly.

The power of Given-When-Then lies in its consistency. Every scenario follows the same pattern, creating rhythm that makes it easy to scan many scenarios and spot anomalies. A scenario with two When steps suggests confusion about what is being tested. A scenario with no Then step provides no verification. A scenario with action in the Given section mixes setup with behavior under test.

And and But keywords extend scenarios naturally without repeating structure. "And" adds another step of the same type: "Then the order is marked as complete, And the inventory is reduced." "But" signals contrast or exception: "Then the payment succeeds, But a fraud alert is logged." These extensions keep scenarios readable while expressing complex behavior chains.

For AI coding agents, Given-When-Then structure provides explicit semantic markers that guide implementation. The agent can parse scenarios mechanically to understand preconditions, actions, and expected results without guessing about intent. This structure reduces ambiguity more effectively than free-form natural language because it forces requirements into a template that separates different concerns explicitly.

Executable Specifications: When Documentation Becomes Tests

Executable specifications represent BDD's most practical innovation: documentation that runs as automated tests and produces verifiable evidence of correctness. Traditional documentation describes what the system should do in static text that becomes outdated as soon as development begins. Executable specifications describe behavior in a format that tools can interpret and run against actual code, producing pass/fail results that indicate whether implementation matches specification.

The concept is straightforward but profound. Instead of maintaining separate documents for requirements, design, and tests, teams maintain one artifact: the executable specification. This single source of truth serves multiple purposes simultaneously. It clarifies requirements during analysis because stakeholders can read and validate scenarios before development

begins. It guides implementation because developers see concrete examples of expected behavior rather than abstract descriptions. It provides automated verification because scenarios run as tests that fail when behavior deviates from specification. It documents system behavior for future reference because passing scenarios prove what the system actually does, not what someone once intended it to do.

Cucumber operationalized this concept by reading feature files written in Gherkin syntax and executing them against step definitions implemented in programming languages. Each Given, When, and Then line maps to a function that performs setup, triggers actions, or makes assertions against the actual system. When all steps execute successfully, the scenario passes and provides evidence that the specified behavior is implemented correctly.

Executable specifications provide continuous feedback throughout development. Scenarios start as failing tests before any code exists, making it obvious what work remains. As developers implement features, scenarios turn green incrementally, providing visible progress toward completion. When refactoring changes implementation details, scenarios verify that observable behavior remains unchanged. When bugs are discovered, new scenarios capture the expected behavior and prevent regression.

The executable nature of specifications eliminates a common failure mode in traditional development: divergence between documentation and implementation. Static documents inevitably drift from reality as systems evolve through countless changes. Executable specifications stay current because they must pass to be considered valid. A scenario that no longer reflects actual system behavior will fail when run, creating immediate pressure to either fix the implementation or update the specification.

For AI coding agents, executable specifications serve as both instruction and verification. The agent reads scenarios to understand what behavior to implement and runs them to verify correctness. This closed loop is more reliable than natural language instructions alone because execution produces objective results rather than subjective assessments of whether requirements are met. An agent cannot argue that "it implemented the feature" if executable scenarios fail: the specification itself determines success, not the agent's interpretation.

Executable specifications also enable continuous integration with BDD principles. Scenarios run automatically in CI pipelines on every code change,

providing rapid feedback about whether new work maintains compliance with behavioral requirements. This integration ensures that specifications are not ceremonial artifacts created for planning but active constraints that govern implementation continuously.

How TDD, ATDD, and BDD Relate and Differ

Understanding the relationship between Test-Driven Development, Acceptance Test-Driven Development, and Behavior-Driven Development clarifies what each contributes and when to use them. These practices are not competing alternatives but complementary approaches that operate at different levels of abstraction and address different concerns.

Test-Driven Development operates at the unit level within individual components. Developers write small, fast tests for single classes or functions before implementing code. TDD excels at driving good design: writing tests first forces developers to think about interfaces, dependencies, and responsibilities before committing to implementation details. The rapid feedback loop of red-green-refactor keeps implementations focused and prevents overengineering. However, TDD does not address whether individual components combine correctly to produce desired system behavior or whether that behavior matches business requirements.

Acceptance Test-Driven Development operates at the system level across component boundaries. Teams define acceptance criteria as executable tests before implementing features, ensuring all stakeholders agree on what "done" means. ATDD excels at aligning implementation with requirements because tests are written collaboratively using concrete examples that stakeholders can validate. However, ATDD tests can become slow and brittle when they exercise systems through user interfaces or complex integration points, making them less suitable for the rapid feedback loops that drive good design.

Behavior-Driven Development synthesizes insights from both approaches while adding emphasis on collaboration and shared language. BDD uses executable specifications at multiple levels: high-level scenarios that describe system behavior from the user perspective (similar to ATDD) and lower-level examples that guide component design (similar to TDD). The unifying principle is focusing on observable behavior expressed in ubiquitous language rather than implementation details or testing mechanics.

Practically, teams often combine all three approaches in a layered strategy. BDD scenarios define what the system should do from the user perspective. ATDD acceptance tests verify that integrated components produce correct results for end-to-end flows. TDD unit tests guide design of individual components and provide fast feedback during implementation. Each layer serves a distinct purpose: BDD ensures the right thing is built, ATDD ensures components work together correctly, and TDD ensures individual pieces are well-designed and reliable.

For AI coding agents, this layered approach becomes even more important because agents operate most effectively when constrained by specifications at multiple levels. High-level BDD scenarios prevent agents from implementing technically correct but business-irrelevant solutions. Mid-level acceptance tests verify integration behavior that emerges only when components interact. Low-level unit tests guide implementation details and catch errors early in the development cycle. Agents given only high-level requirements without lower-level constraints tend to produce superficial implementations that pass obvious tests while missing subtle behavioral nuances.

Chapter 2: AI Coding Agents Defined

Assistant Versus Agent: Autonomy as the Dividing Line

The term "AI coding agent" has entered common usage but lacks precise definition in many discussions. Clarifying what constitutes an agent versus a simpler assistant is essential because the two require fundamentally different approaches to specification, control, and verification.

An AI coding assistant responds to explicit user prompts with suggestions, completions, or explanations. GitHub Copilot's inline suggestions, ChatGPT answering programming questions, and Cursor's chat panel all fit this category. The human drives every interaction: deciding what to ask, evaluating responses, selecting which suggestions to accept, and performing all actions in the development environment. The assistant augments human capability but does not act independently.

An AI coding agent takes goals or tasks as input and acts autonomously to achieve them using available tools. The agent decides what steps to take, which files to read or modify, which commands to run, which tests to execute, and when work is complete. Human involvement may include initial task definition, periodic review, approval of significant changes, or intervention when the agent encounters problems it cannot resolve. But the agent operates independently between these human touchpoints.

Autonomy is the dividing line. Assistants wait for instructions. Agents take initiative within their assigned scope. This distinction matters because autonomous systems introduce risks and requirements that do not apply to reactive tools: agents can make mistakes without immediate human oversight, pursue incorrect approaches for extended periods, consume significant resources before anyone notices problems, and produce outcomes that appear correct superficially while violating unstated requirements.

The boundary between assistant and agent is not always sharp. Many modern tools offer modes on a spectrum from fully reactive to highly autonomous. Cursor's Agent Mode can execute multi-step tasks with terminal access and file modifications while remaining accessible through the IDE

interface. Claude Code operates as an agentic tool in the terminal that reads codebases, plans work, edits files, runs tests, and manages git workflows through natural language commands [6]. These hybrid systems blur categories but the autonomy spectrum remains useful: the more autonomous a system, the more it needs behavioral contracts to constrain its actions.

What Makes a System an AI Coding Agent

An AI coding agent is a software system powered by a large language model that can autonomously perform software development tasks by interacting with development environments through tools and interfaces. This definition contains several essential components.

First, the system uses a large language model as its reasoning engine. The LLM processes natural language instructions, understands code and documentation, generates implementation plans, writes code, and makes decisions about how to proceed. Current frontier models including Claude Opus 4, GPT-5, and Gemini 2.5 Pro demonstrate coding capabilities that make agentic workflows practical rather than experimental [7].

Second, the system operates autonomously within its scope. Given a task description, the agent decomposes it into steps, executes those steps using available tools, evaluates results, adjusts its approach based on feedback, and continues until the task is complete or it encounters an insurmountable obstacle. This autonomy distinguishes agents from assistants that require explicit instruction for each action.

Third, the system interacts with development environments through tools rather than generating isolated code snippets. Tools provide the agent's hands and eyes: file system access to read and write source code, terminal access to run commands and scripts, test runners to execute verification suites, version control systems to manage changes, web browsers to research documentation or verify deployed behavior, and APIs to integrate with external services. Tool use transforms language models from text generators into action-capable systems.

Fourth, the system handles multi-step workflows that span planning, execution, and verification. A genuine coding agent does not merely generate code when asked. It explores existing codebases to understand context, plans implementation approaches considering constraints and dependencies, writes

code incrementally, runs tests to verify correctness, debugs failures by analyzing error messages and modifying code, refactors while preserving behavior, and produces complete deliverables like pull requests with documentation.

Several prominent systems exemplify these characteristics. Devin from Cognition Labs positions itself as an autonomous AI software engineer that takes natural language tasks and returns finished pull requests with minimal human intervention [8]. Claude Code from Anthropic operates as a terminal-based agent that understands codebases, executes routine tasks, explains complex code, and handles git workflows through conversation [9]. OpenDevin (now OpenHands) provides an open-source platform for generalist AI agents that interact with environments through code execution, command line interfaces, and web browsing [10].

These systems differ in architecture, capabilities, and deployment models but share the fundamental pattern: LLM-based reasoning combined with tool use enables autonomous software development activity. Understanding this pattern is more important than knowing specific product details because the underlying principles apply regardless of which tools teams adopt.

Core Capabilities: Repository Understanding, Planning, Execution

Effective AI coding agents rely on several core capabilities that work together to enable autonomous development. Each capability addresses a specific challenge in translating high-level goals into concrete implementation work.

Repository understanding allows agents to navigate and comprehend existing codebases rather than generating code in isolation. Agents must read files to understand project structure, identify relevant modules for a given task, grasp coding conventions and architectural patterns, recognize dependencies between components, and locate existing tests or documentation that inform implementation decisions. Modern agents achieve this through indexing systems that create searchable representations of codebases, semantic search that finds relevant code by meaning rather than keywords, and context management strategies that load appropriate files into the agent's working memory for each task [11].

Planning enables agents to decompose complex tasks into manageable steps before executing any code changes. A task like "implement user au-

thentication with OAuth2" requires understanding prerequisites, sequencing operations logically, identifying risks and dependencies, estimating effort, and producing a plan that humans can review before work begins. Planning prevents agents from diving directly into implementation and discovering too late that foundational pieces are missing or approaches conflict with existing architecture. Effective planning also produces artifacts like task lists, design documents, and specification files that serve as reference points throughout development and enable verification that the agent followed its stated approach.

Execution is the agent's ability to perform concrete actions: writing code to files, running commands in terminals, invoking test suites, managing version control operations, browsing documentation, and calling external APIs. Execution capabilities determine what an agent can actually accomplish versus what it can only describe theoretically. An agent that cannot run tests must rely on self-assessment of code correctness, which is unreliable. An agent without terminal access cannot install dependencies or build projects to verify compilation. Tool availability directly constrains agent effectiveness.

These capabilities interact in loops rather than linear sequences. Agents typically explore a repository to understand context, create initial plans based on that understanding, execute some steps, observe results, revise plans based on new information, continue execution, and repeat until completion. This iterative pattern mirrors how experienced human developers approach unfamiliar codebases: investigate, hypothesize, experiment, learn, adjust, and proceed.

Anthropic's research analyzing approximately 400,000 Claude Code sessions found that typical sessions involve about four conversational turns and ten agent actions per prompt, with users making roughly seventy percent of planning decisions while the agent makes eighty percent of execution decisions [12]. This division of labor suggests that effective agentic workflows combine human strategic judgment with machine tactical efficiency rather than replacing humans entirely.

Tool Use: Terminals, Editors, Test Runners, and Beyond

Tool use is what transforms language models from conversational AI into practical coding agents. Without tools, an agent can generate code text but cannot

verify it compiles, run it to observe behavior, integrate it with existing systems, or produce deliverables in formats that development workflows expect.

Terminal access is arguably the most critical tool because it provides general-purpose capability to run any command available in the development environment. Through the terminal, agents install dependencies, build projects, run tests, execute linters and formatters, manage databases, start servers, deploy applications, query logs, and perform virtually any operation a human developer would execute via command line. Terminal access makes agents adaptable to diverse technology stacks and project configurations because they can discover and use whatever tools are available rather than being limited to predefined capabilities.

Code editors provide file system interaction for reading existing code and writing new implementations. Agents need to read files to understand current implementation before making changes, write files to implement new functionality or modify existing behavior, and manage multiple files simultaneously for tasks that span components. Modern agents often have specialized editing tools beyond basic file read/write: search and replace across files, insert at specific line numbers, apply diffs, create new files in appropriate directories, and delete obsolete code.

Test runners enable agents to verify correctness through automated execution rather than subjective self-assessment. Running tests provides objective feedback about whether implementation matches expected behavior. Agents that can execute test suites, read failure output, analyze which assertions failed, modify code to address failures, and re-run tests create closed feedback loops that drive iterative improvement. This capability is essential for BDD-driven workflows because executable specifications are meaningless unless agents can run them and observe results.

Version control integration allows agents to manage changes in ways compatible with team workflows. Agents should create branches for new work, stage and commit changes with meaningful messages, generate pull requests with descriptions linking to requirements, handle merge conflicts, and follow repository conventions for branching and review processes. This integration ensures agent-generated code enters the same quality gates and review processes as human-written code rather than bypassing established procedures.

Web browsing capability enables agents to research documentation, look up API references, search for solutions to unfamiliar problems, verify deployed

application behavior, and stay current with technology changes. While agents trained on large corpora have extensive built-in knowledge, specific project dependencies, internal APIs, and recent framework updates may require looking up information that training data does not cover comprehensively.

Additional specialized tools extend agent capabilities further: database clients for schema inspection and data manipulation, container runtimes for isolated execution environments, API testing tools for service integration verification, browser automation for UI testing, and custom tools designed for specific organizational needs like internal deployment systems or proprietary infrastructure.

The Model Context Protocol (MCP) has emerged as a standard for connecting AI agents to external tools and data sources [13]. MCP provides a unified interface that agents use to discover available tools, understand their capabilities through schemas, invoke them with appropriate parameters, and process results. This standardization reduces friction in extending agent capabilities because new tools can be added without modifying agent code: the agent discovers and uses them dynamically through the protocol.

Context Management and Memory Systems

Context management determines how much information an agent can access during reasoning and how effectively it uses that information. Large language models operate within context windows that limit the total tokens available for prompts, retrieved content, conversation history, and generated responses simultaneously. Managing this finite resource efficiently is critical for agent effectiveness, especially on complex tasks requiring extensive knowledge about codebases, requirements, and prior decisions.

System-level context includes persistent instructions that define the agent's role, constraints, capabilities, and operating procedures. These instructions establish guardrails: what the agent should always do, what requires human approval, what it must never do under any circumstances. Well-designed system prompts aim for the "right altitude": clear enough to provide meaningful guidance without being so detailed that they consume excessive tokens or create brittle rules that break on edge cases [14].

Project-level context provides information specific to the codebase and domain: architecture documentation, coding standards, dependency lists, API

contracts, data models, configuration files, and existing behavioral specifications. This context helps agents generate code consistent with project conventions rather than generic solutions based solely on training patterns. Agents that load relevant project context before starting tasks produce higher quality work because they understand constraints and expectations specific to the environment they are working in.

Task-level context includes information directly relevant to the current work: requirements being implemented, files being modified, test results being analyzed, errors being debugged. This context changes dynamically as agents progress through tasks, requiring continuous updates to keep the agent informed about current state without overwhelming it with irrelevant historical information.

Memory systems address the challenge of retaining information across long interactions that exceed single context windows. Short-term memory holds recent conversation history and immediate work context within the active session. Long-term memory persists important information across sessions: decisions made, patterns discovered, recurring problems encountered, domain knowledge accumulated over time. Effective memory systems summarize and compress information rather than storing everything verbatim, retaining essential facts while discarding transient details that no longer matter [15].

Anthropic's research on context engineering identifies several strategies for managing context effectively in agentic workflows: just-in-time dynamic loading of information when needed rather than pre-processing all data upfront, compaction techniques that summarize conversation history to free tokens for new content, structured note-taking where agents explicitly record important information in retrievable formats, and sub-agent architectures that delegate specialized tasks with focused context rather than burdening a single agent with everything [16].

Context management directly impacts cost because token usage drives API expenses. Agentic workflows consume significantly more tokens than simple chat interactions: estimates range from five to thirty times more tokens per task compared to standard generative AI use, with complex tasks involving retries and self-correction consuming one to three million tokens or more [17]. Efficient context management reduces costs while maintaining quality by ensuring agents have necessary information without redundant repetition.

Prompt Architecture: System Instructions, Project Prompts, Task Prompts

Prompt architecture refers to how instructions are structured and layered to guide agent behavior effectively. Treating prompts as monolithic blocks of text leads to fragile systems that break when requirements change or tasks become complex. Layered prompt architectures separate concerns into distinct components that can be maintained independently.

System instructions define the agent's fundamental identity, capabilities, constraints, and operating principles. These instructions are persistent across all interactions and establish baseline behavior. A well-designed system instruction might specify the agent's role ("you are a senior software engineer"), communication style ("be concise and technical"), operational constraints ("never modify test files without explicit approval"), safety rules ("do not execute commands that delete data without confirmation"), and escalation procedures ("ask for human input when uncertain about requirements"). System instructions should be concise enough to fit comfortably within context windows while comprehensive enough to prevent common failure modes.

Project prompts provide context specific to a codebase or product. They describe architecture, technology stack, coding conventions, domain terminology, known pitfalls, and integration points. Project prompts might include references to documentation files the agent should consult, patterns that existing code follows, constraints imposed by infrastructure or dependencies, and quality standards the team maintains. These prompts enable agents to generate code consistent with project norms rather than generic solutions. Addy Osmani's analysis of over 2,500 agent configuration files identified six core areas that effective project specifications address: commands, testing, project structure, code style, git workflow, and boundaries [18].

Task prompts define specific work to be accomplished within the broader system and project context. They include requirements being implemented, acceptance criteria or behavioral specifications to satisfy, constraints unique to the task, references to relevant files or components, and success criteria for completion. Task prompts are ephemeral: they apply only to current work and may change as understanding evolves during implementation. Good task prompts are specific about inputs, outputs, and constraints while leaving implementation details flexible so agents can apply their capabilities effectively

[19].

These layers combine during agent operation with system instructions providing the foundation, project prompts adding domain context, and task prompts directing immediate action. This separation enables reuse: the same system instructions apply across all projects, project prompts apply to all tasks within a codebase, and task prompts are customized per assignment without duplicating shared context.

For BDD-driven workflows, specifications themselves become part of the prompt architecture. Feature files and scenarios provide precise behavioral requirements that agents implement against. These specifications occupy a unique position: they are more detailed than general project context but more stable than transient task instructions. Specifications persist as living artifacts that guide multiple implementation sessions over time, making them natural components of prompt strategy for agentic development.

Single-Agent Versus Multi-Agent Architectures

Agentic systems can be organized as single generalist agents or teams of specialized agents collaborating on tasks. Each architecture has trade-offs in complexity, cost, capability, and reliability that influence which approach suits particular use cases.

Single-agent architectures deploy one agent to handle all aspects of a task: understanding requirements, planning implementation, writing code, running tests, debugging failures, and producing deliverables. This approach is simpler to set up and manage because there is only one system to configure, monitor, and maintain. Communication overhead is minimal because the agent operates within a single context window rather than exchanging messages between components. For straightforward tasks with well-defined requirements, single agents often perform adequately without the complexity of multi-agent coordination.

Limitations emerge as task complexity increases. A single agent must hold all relevant information in its context simultaneously: requirements, codebase understanding, implementation plans, test results, error analysis, and domain knowledge. This creates pressure on context windows and can lead to degraded performance when too much information competes for attention. Single agents also lack specialized perspectives: the same reasoning process

that generates code also evaluates it, creating opportunities for confirmation bias where agents convince themselves of correctness without rigorous verification.

Multi-agent architectures deploy different agents with specialized roles collaborating through structured communication. Common specializations include planners that analyze requirements and create implementation strategies, implementers that write code based on plans, testers that generate and run verification suites, reviewers that evaluate code quality and compliance, and verifiers that independently confirm behavior matches specifications without access to implementation details.

Multi-agent systems offer several advantages. Specialization allows each agent to operate with focused context relevant to its role rather than everything about the task. A testing agent needs specifications and compiled code but not detailed architectural rationale. A planning agent needs requirements and existing architecture but not test execution results. This focus reduces context pressure and enables deeper reasoning within each domain. Separation of concerns also enables independent verification: a verifier agent that evaluates implementation against specifications without seeing how implementation was produced provides more objective assessment than self-evaluation [20].

Frameworks like CrewAI, LangGraph, and AutoGen provide infrastructure for building multi-agent systems with role-based orchestration, message passing between agents, and workflow management [21]. These frameworks abstract away low-level communication details so developers can focus on defining agent roles, capabilities, and interaction patterns.

Trade-offs include increased complexity in setup and maintenance, higher token costs from multiple agents processing overlapping information, coordination overhead from inter-agent communication, and potential for disagreements between agents that require resolution mechanisms. Multi-agent architectures make sense when task complexity justifies the additional investment: large features requiring coordinated changes across many components, tasks benefiting from specialized expertise like security review or performance optimization, or workflows requiring independent verification for compliance or quality assurance.

For BDD-driven development, multi-agent architectures align naturally with BDD's emphasis on separation of concerns. Specifications define behavior

independently of implementation. Different agents can own specification creation, implementation against specifications, and verification that implementation matches specifications. This alignment reduces conflicts between roles because behavioral specifications provide unambiguous criteria that all agents reference rather than negotiating requirements through conversation.

Human-in-the-Loop Patterns and Control Mechanisms

Despite increasing agent autonomy, human involvement remains essential for effective and safe agentic development. The question is not whether humans participate but how: at what points, with what authority, and through what mechanisms. Designing appropriate human-in-the-loop patterns balances efficiency gains from automation against risks of uncontrolled autonomous action.

Approval gates require human confirmation before agents perform significant actions. Common gate points include starting new work based on requirements, making architectural decisions that affect system structure, modifying production configuration, deploying changes to live environments, and merging code into main branches. Approval gates prevent agents from committing to approaches or changes without human oversight while allowing autonomous operation between gates. The granularity of gates determines the autonomy level: frequent gating creates tightly controlled workflows with minimal agent independence, while sparse gating allows extended autonomous operation with periodic human review.

Review processes have humans evaluate agent-produced work before acceptance. Code review examines implementation quality, adherence to conventions, security considerations, and alignment with requirements. Specification review validates that behavioral scenarios accurately capture intended behavior before agents implement them. Plan review ensures implementation strategies are sound before significant work begins. Review can be synchronous, with humans examining work immediately as agents produce it, or asynchronous, with agents completing substantial work batches for later evaluation.

Intervention allows humans to redirect agents when they encounter problems or pursue incorrect approaches. Agents should provide visibility into their reasoning and actions so humans can detect issues early: logging decisions,

explaining rationale for choices, reporting errors encountered, and requesting guidance when uncertain. Effective intervention mechanisms let humans correct course without restarting work entirely: adjusting requirements mid-task, providing additional context the agent missed, specifying constraints the agent overlooked, or taking over specific subtasks that the agent struggles with.

Escalation rules define when agents should seek human help rather than continuing autonomously. Clear escalation criteria prevent agents from wasting resources on impossible tasks or making irreversible mistakes. Examples include requirements that are ambiguous or contradictory beyond what the agent can resolve through questioning, technical obstacles that require decisions only humans can make, security concerns that need expert assessment, and repeated failures that suggest fundamental misunderstandings rather than fixable implementation errors.

For BDD-driven workflows, human-in-the-loop patterns integrate naturally with specification lifecycle. Humans create and validate behavioral specifications before agents implement them, ensuring requirements are correct at the source. Humans review agent implementations against specifications to verify compliance. Humans approve specification changes that modify agreed-upon behavior. This integration concentrates human effort where it provides most value: defining what should be built and verifying that it was built correctly, while agents handle the mechanical work of translation from specification to implementation.

Sandboxing, Permissions, and Safety Boundaries

Autonomous agents with tool access can cause harm if they execute dangerous operations without constraints. Sandboxing, permission controls, and safety boundaries protect systems, data, and users from agent mistakes or malicious behavior whether accidental or intentional.

Sandboxing isolates agent execution in controlled environments where actions cannot affect production systems or sensitive data. Container-based sandboxes provide agents with complete development environments including operating systems, programming languages, tools, and file systems while preventing access to resources outside the container. Agents can run commands, install packages, modify files, and test code freely within their sandbox without risking damage to actual infrastructure. Cognition's Devin runs each session

in a dedicated Linux sandbox with internet access, full file system, and web browser entirely isolated from host systems [22].

Permission controls restrict which actions agents can perform based on role, task, or configuration. Fine-grained permissions specify exactly what an agent may access: read certain directories but not others, write to specific paths, execute particular command categories, access designated APIs, or interact with named services. Permission models typically follow least-privilege principles: agents receive minimum access necessary for assigned tasks rather than broad permissions that enable unintended actions.

Safety boundaries define explicit constraints on agent behavior regardless of permissions available. Boundaries might prohibit agents from deleting files matching certain patterns, executing commands with destructive flags, modifying configuration in production environments, accessing personal or sensitive data, making network calls to unauthorized destinations, or escalating their own privileges. These boundaries operate as hard limits that agents cannot override through reasoning or instruction interpretation.

Addy Osmani describes three-tier boundary structures for agent control: actions the agent should always perform without asking (routine operations within safe parameters), actions requiring human approval before execution (significant changes with potential impact), and actions the agent must never perform under any circumstances (dangerous operations like committing secrets to repositories) [23]. This structure provides clarity about autonomy levels while maintaining protection against catastrophic failures.

For BDD-driven workflows, safety boundaries should protect specifications themselves. Agents implementing features should not be able to modify behavioral specifications without authorization because changing requirements after implementation begins enables agents to weaken constraints and declare success against degraded criteria. Specifications should have stricter access controls than implementation code: read-only for implementing agents, modifiable only by authorized humans or specification management processes with explicit approval workflows.

Observability and Verification

Observability provides visibility into what agents are doing, why they are doing it, and whether their actions produce intended results. Without observability,

agent activity becomes a black box where inputs enter and outputs emerge without understanding the reasoning, decisions, or intermediate steps that produced them. This opacity makes debugging failures, assessing quality, and building trust in agent-generated work difficult or impossible.

Logging captures agent activity for later analysis: tasks assigned, plans created, files read and modified, commands executed, test results observed, errors encountered, decisions made. Good logs include not just what happened but why: the reasoning behind choices, alternatives considered and rejected, uncertainties acknowledged, and assumptions made. Logs enable reconstruction of agent behavior after completion to understand how outcomes were achieved and identify problems in reasoning or execution.

Progress reporting keeps humans informed about task status during execution: what work has been completed, what remains, estimated time to finish, obstacles encountered, and quality indicators like test pass rates. Regular progress updates prevent surprises where agents appear inactive for extended periods only to reveal massive changes or failures at completion. They also enable timely intervention when agents pursue incorrect approaches rather than waiting until substantial wasted effort accumulates.

Verification mechanisms provide objective evidence that agent work meets requirements independent of self-assessment. Automated tests run against specifications produce pass/fail results that indicate whether behavior is correct. Static analysis tools check code for bugs, security vulnerabilities, and style violations. Coverage analysis measures how thoroughly tests exercise implemented code. These mechanisms create verification layers that operate independently of agent reasoning, catching errors that agents miss or rationalize away.

For BDD-driven workflows, observability and verification integrate tightly with executable specifications. Specification execution logs show which scenarios passed and failed during implementation. Progress reports can reference specific scenarios being addressed rather than vague descriptions of work in progress. Verification becomes straightforward because specifications define exact criteria: run all scenarios, count passing versus failing, assess whether failures represent implementation bugs or specification problems. This integration makes BDD-driven agentic development more transparent and trustworthy than approaches relying on natural language instructions without executable verification.

Chapter 3: Why BDD Is Essential for AI Agents

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

The Hallucination Problem: When Agents Invent Requirements

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

Specification Drift and Silent Requirement Weakening

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

Test Manipulation: Agents That Make Tests Pass the Wrong Way

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

Scope Creep and Unintended Behavior in Agentic Workflows

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

False Completion: Declaring Success Without Proof

How Behavioral Specifications Constrain Probabilistic Systems

BDD as the Shared Language Between Human Intent and Agent Execution

The Trust Equation: Verifiable Behaviors Build Confidence

Chapter 4: Designing Behavioral Specifications for Agents

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

Translating Ambiguous Requirements into Precise Behaviors

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

Writing Scenarios That Leave No Room for Misinterpretation

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

Example Mapping Adapted for Agentic Consumption

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

Handling Edge Cases Explicitly in Specifications

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

Non-Functional Requirements as Testable Behaviors

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

Structuring Feature Files for Complex Systems

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

Scenario Decomposition: Breaking Problems into Agent-Sized Tasks

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

Common Specification Anti-Patterns and How to Avoid Them

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

Chapter 5: The BDD Agent Loop — A Complete Workflow

Overview: From Requirement to Delivery

Understand: Analyzing Requirements and Repository Context

Clarify: Asking Questions Before Specifying

Specify: Creating Behavioral Specifications from Requirements

Plan: Decomposing Scenarios into Implementation Tasks

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

Implement: Agent Coding Against the Specification

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

Test: Generating and Executing Tests Driven by Scenarios

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

Observe: Interpreting Test Results and Agent Outputs

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

Diagnose: Root Cause Analysis of Failures

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

Repair: Iterative Fixes Guided by Behavioral Feedback

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

Refactor: Improving Code While Preserving Behaviors

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

Verify: Independent Confirmation the Behavior Is Correct

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

Review: Human Validation Before Acceptance

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

Deliver: Integration and Deployment Readiness

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

Chapter 6: Practical Implementation — Building Your BDD Agent System

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

Recommended Project Structure and Directory Layout

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

Specification Formats: Feature Files, Scenario Definitions, Contracts

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

Agent Instruction Files and System Prompts

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

Context Files for Repository and Domain Knowledge

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

Test Harnesses and Validation Scripts

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

CI/CD Pipeline Integration with Quality Gates

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

Traceability Mechanisms and Audit Logging

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

Complete Working Example: From Empty Repository to BDD-Agentic Workflow

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

Chapter 7: Advanced BDD Techniques for Agents

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

Managing Deterministic Versus Probabilistic Agent Behavior

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

Context Engineering for Better Specification Interpretation

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

Behavioral Coverage Metrics and Analysis

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

Test Isolation Strategies for Reliable Agent Feedback

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

Property-Based Testing as a Complement to BDD Scenarios

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

Generative Testing for Edge Case Discovery

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

Integration and End-to-End Testing in Agentic Workflows

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

Handling External Dependencies, Async Behavior, and Distributed Systems

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

Security, Performance, and Accessibility as Behavioral Requirements

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

Chapter 8: Guardrails, Verification, and Trust

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

Preventing Test Manipulation and Reward Hacking

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

Behavioral Invariants That Cannot Be Silently Changed

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

Independent Verification: Separate Agents for Implementation and Checking

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

Adversarial Scenarios and Negative Testing

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

Mutation Testing to Verify Test Quality

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

Static Analysis, Linting, and Type Checking as Guardrails

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

Security Scanning Integrated into the BDD Workflow

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

Code Review Protocols for Agent-Generated Code

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

Chapter 9: Multi-Agent BDD Architectures

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

Specialized Agent Roles: Planner, Specifier, Implementer, Tester, Reviewer

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

When Single-Agent Is Enough Versus When to Specialize

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

Structured Artifact Exchange Between Agents

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

Behavioral Specifications as the Shared Source of Truth

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

Resolving Disagreements and Conflicts Between Agents

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

Orchestration Patterns for Multi-Agent Workflows

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

Cost, Complexity, and Coordination Trade-Offs

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

Chapter 10: End-to-End Case Studies

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

Case Study One: A Simple Feature — Authentication Token Refresh

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

Requirement

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

Understand Stage

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

Clarify Stage

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

Specify Stage

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

Plan Stage

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

Implement Stage

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

Test Stage

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

Observe and Diagnose Stages

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

Repair Stage

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

Refactor Stage

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

Verify Stage

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

Review Stage

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

Deliver Stage

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

Analysis

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

Case Study Two: A Complex Feature — Distributed Rate Limiter with Redis

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

Requirement

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

Understand Stage

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

Clarify Stage

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

Specify Stage

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

Plan Stage

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

Implement Stage with Failures

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

Test Stage Extended

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

Verify Stage

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

Review Stage

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

Deliver Stage

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

Analysis

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

Case Study Three: Multi-Agent Workflow on a Full User Story

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

User Story

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

Multi-Agent Setup

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

Specification Agent Work

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

Planner Agent Work

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

Implementer Agent Work

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

Tester Agent Work

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

Verifier Agent Work

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

Human Review and Delivery

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

Analysis

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

Cross-Case Observations

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

Chapter 11: Comparing Approaches

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

Prompt-Driven Coding Without Formal Specifications

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

Traditional Test-Driven Development with AI Agents

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

Acceptance Test-Driven Development in Agentic Contexts

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

Plan-and-Execute Agent Patterns Versus BDD-Driven Agents

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

Specification-Driven Development and Contract Testing

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

Benefits, Limitations, and Trade-Offs of Each Approach

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

Choosing the Right Approach for Your Context

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

Chapter 12: Organizational Adoption and Scaling

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

Integrating BDD Agent Workflows into Existing Repositories

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

Legacy System Migration Strategies

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

Team Responsibilities and New Roles

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

Governance Models for Agentic Development

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

Security and Permission Frameworks

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

Token Cost Management and Optimization

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

Observability, Metrics, and Behavioral Coverage Tracking

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

Maturity Model: From Experimentation to Production Excellence

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

Chapter 13: Production Reference Architecture

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

Reference Architecture for Production BDD Agent Systems

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

Core Components

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

Data Flow

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

Security Controls

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

Phased Adoption Roadmap

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

Phase 1: Foundation (Months 1-3)

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

Phase 2: Standardization (Months 4-6)

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

Phase 3: Scaling (Months 7-12)

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

Phase 4: Excellence (Ongoing)

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

Reusable Workflow Templates and Patterns

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

Feature Implementation Template

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

Agent System Prompt Template

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

Definition of Done Template

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

Best Practices Consolidated

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

Anti-Patterns and What to Avoid

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

Troubleshooting Common Problems

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

Concise Quick Reference for Real Projects

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

Essential Checklist Before Starting Feature

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

Essential Checklist Before Accepting Feature

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

Agent Boundary Quick Reference

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

Gherkin Scenario Quick Reference

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

BDD Agent Loop Stages Summary

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

Conclusion: The Specification Imperative

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

References

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