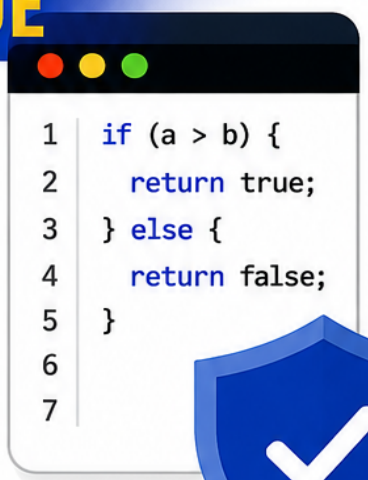
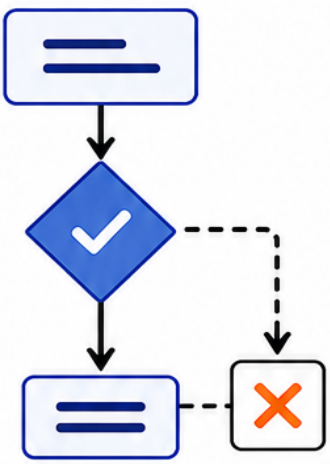
AUTOMATING SOFTWARE LOGIC VERIFICATION WITH CLAUDE CODE

AI REASONING.
DETERMINISTIC TOOLS.
HUMAN EXPERTISE.

A BEGINNER-TO-ADVANCED PRACTICAL GUIDE

- CATCH LOGIC FLAWS EARLY
- COMBINE AI INSIGHT WITH PROVEN TOOLS
- BUILD REPEATABLE VERIFICATION WORKFLOWS
- SCALE VERIFICATION TO PRODUCTION





DANIEL CARTER

Automating Software Logic Verification with Claude Code

A Beginner-to-Advanced Practical Guide

Steve Publications

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

This version was published on 2026-09-02

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 Beginner-to-Advanced Practical Guide

This book teaches you how to use Claude Code, Anthropic's agentic coding assistant, to systematically verify the correctness of software logic across real-world projects. You will learn a disciplined methodology that treats AI as a reasoning collaborator rather than an oracle, combining Claude Code's analytical capabilities with deterministic verification tools and human expertise. Whether you are new to software verification or experienced in testing and quality assurance, this guide takes you from fundamental concepts through production-scale workflows, providing concrete prompts, configurations, examples, and techniques you can apply immediately.

Introduction: Why Software Logic Verification Matters Now

A single misplaced comparison operator caused a 2021 Twitter outage that lasted hours and affected millions of users. A boundary check error in a financial trading system at Knight Capital in 2012 lost approximately 440 million dollars in forty-five minutes. A logic flaw in authentication code at a major healthcare provider allowed unauthorized access to patient records for months before detection. These are not rare anomalies. They are the natural consequence of building complex software systems with incomplete verification practices.

Modern software is more complex than ever. Microservice architectures, distributed data stores, asynchronous event flows, and intricate business rules create vast webs of logic that no human can fully trace mentally. Traditional testing catches many bugs but cannot exhaustively verify correctness. Code review helps but depends on reviewer expertise and attention. Static analysis tools detect patterns but struggle with semantic reaso8ning about what code is supposed to do. We need better approaches.

The Hidden Cost of Logical Errors in Production Systems

Logical errors differ from syntax errors or runtime crashes in an important way: the program runs, produces output, and often appears to work correctly most of the time. A function that calculates tax may return the right answer for common inputs but silently produce w4rong results for edge cases involving specific product codes or customer tiers. An access control check may work for standard users but fail for roles with unusual permission combinations. These errors survive testing because test suites rarely cover every combination of conditions, and they escape code review because the logic looks reasonable at a glance.

The cost accumulates in several forms. Direct financial losses occur when incorrect calculations affect billing, pricing, or trading. Customer trust erodes

when systems produce wrong results or behave inconsistently. Compliance violations emerge when business rules are not correctly enforced. Technical debt grows as developers add workarounds for symptoms without fixing underlying logic flaws. Debugging becomes exponentially harder because the same code path may produce correct results in development but fail under production conditions with different data distributions.

Research on software defects consistently shows that logical errors are among the most expensive to fix once discovered in production. Studies from IBM, NASA, and various industry analyses demonstrate that defect remediation costs increase by factors of ten to one hundred depending on when the error is detected. Errors found during requirements analysis cost baseline amounts to fix. The same errors discovered during testing cost substantially more. Those found after release cost orders of magnitude more when accounting for incident response, customer impact, reputation damage, and regulatory consequences.

Testing Versus Verification: Why One Cannot Replace the Other

Most software teams invest heavily in testing, which is absolutely necessary but fundamentally limited. Testing shows the presence of defects, not their absence, as Edsger Dijkstra famously observed. No matter how comprehensive your test suite, you can never prove that code is correct through testing alone because you cannot execute all possible inputs and states for any nontrivial program.

Testing and verification serve different purposes. Testing executes code with specific inputs and checks whether outputs match expectations. It validates behavior on concrete examples. Verification examines code to determine whether it satisfies specified properties across all possible executions, or at least provides stronger evidence of correctness than testing alone can offer. Testing asks whether the code works for these cases. Verification asks whether the code works for every case.

This distinction matters practically. A well-tested function may still contain unreachable error handlers that would fail if ever reached. It may have correct behavior on all tested paths but incorrect logic on untested branches. It may satisfy current requirements but violate invariants that will matter when the

system evolves. Verification techniques examine these structural and logical properties directly, without relying solely on executed examples.

In practice, verification is not a binary choice between complete mathematical proof and no analysis at all. It exists on a spectrum. At one end are formal methods that mathematically prove correctness against specifications, used in safety-critical domains like aviation and medical devices. At the other end are lightweight techniques like code review, static analysis, and reasoning about control flow. Most software projects benefit from techniques somewhere in the middle: systematic analysis of logic paths, identification of invariants, examination of edge cases, and validation that implementations match their intended behavior.

What AI-Assisted Verification Actually Means

Large language models have transformed many aspects of software development, from code generation to documentation. Their application to verification is equally significant but requires careful understanding of what they can and cannot do.

Claude Code is an agentic coding assistant that reads codebases, executes commands, runs tools, and reasons about software through a conversational interface. Unlike simple chatbots, it operates as an agent with access to your project files, terminal, git history, and development tools. It can trace execution paths across multiple files, identify patterns in logic, suggest properties that should hold, generate test cases targeting specific behaviors, and explain complex code in terms accessible to humans who may not have written it.

When we talk about AI-assisted verification with Claude Code, we mean using its analytical capabilities as part of a disciplined verification workflow. This includes asking it to analyze control flow through critical functions, identify assumptions embedded in the code, discover edge cases that tests might miss, verify that business rules are correctly implemented across multiple modules, check consistency between API contracts and implementations, review state transitions for completeness, investigate concurrency patterns for potential issues, and examine security-sensitive logic for vulnerabilities.

Crucially, AI-assisted verification does not mean treating Claude Code as an oracle whose conclusions require no questioning. Large language models generate plausible-sounding analysis that may contain errors, hallucinations,

or incomplete reasoning. They may miss subtle bugs that deterministic tools would catch. They may confidently assert incorrect properties about code behavior. The methodology this book teaches treats Claude Code as a powerful collaborator that accelerates and deepens verification work while requiring independent validation of its conclusions through tests, static analysis, deterministic tools, and human judgment.

The appropriate role for AI in verification is analogous to the role of a skilled second reviewer in code review: someone who can catch things you missed, challenge your assumptions, suggest improvements, and bring fresh perspective, but whose suggestions still require your judgment to validate. Claude Code brings scale and analytical capability beyond what any single human reviewer can provide, examining thousands of lines of code, tracing complex execution paths, and systematically exploring edge cases. But it does not replace the need for rigorous validation practices.

How This Book Is Structured and How to Use It

This book is organized as a progressive learning path from foundations through advanced professional workflows. Each chapter builds on previous material, so reading in order is recommended even when you are experienced with some topics. The structure follows a practical progression: understanding what verification is, setting up your tools, learning specific techniques, applying them to real scenarios, and scaling to production use.

The first section establishes foundations. You will learn precise definitions of verification concepts, how they relate to testing and other quality practices, what Claude Code is and how it works, and the theoretical basis for AI-assisted analysis. This grounding matters because using Claude Code effectively requires understanding both what you are trying to verify and what capabilities the tool actually provides.

The second section covers setup and configuration. You will get a complete working environment with every step explained: installation, authentication, project configuration, permissions, hooks, MCP servers, and integration with your existing tooling. The goal is that by the end of this section you have a verified, production-ready Claude Code setup tailored for verification work.

The third section teaches core verification techniques chapter by chapter. Each chapter focuses on a specific aspect of logic verification: control

flow analysis, invariant identification, algorithm correctness, state machine validation, API contract checking, concurrency analysis, security logic review, business rule verification, edge case discovery, and test generation. Every technique is presented with complete Claude Code prompts you can use immediately, realistic examples showing the full workflow from question to answer, and guidance on validating results.

The fourth section addresses practical workflows for real development scenarios: pull request verification, regression prevention during refactoring, repository-wide analysis of large codebases, and integration into CI/CD pipelines. These chapters show how to operationalize the techniques into repeatable processes that your team can use daily.

The fifth section covers critical topics for responsible adoption: validating Claude Code's own output, detecting hallucinations and flawed reasoning, understanding when AI assistance is insufficient, security considerations, and safeguards for production use. These are not optional extras but essential components of any serious verification practice using AI tools.

Throughout the book, you will find reusable artifacts: prompt templates you can adapt to your projects, configuration examples for Claude Code settings, workflow patterns for different scenarios, checklists for systematic analysis, and decision frameworks for choosing appropriate techniques. The goal is that this book serves both as a learning resource when you are building skills and as a reference when you are designing verification systems in production.

Some technical prerequisites will help you get the most from this book: basic programming experience in at least one language, familiarity with version control using git, understanding of what unit tests and integration tests are, and comfort working in a terminal environment. No prior experience with formal methods or Claude Code is required. All specialized terminology is defined when first introduced.

You can read this book linearally from start to finish, or jump to chapters relevant to your immediate needs if you already understand the foundations. If you are new to verification concepts, start from the beginning. If you are experienced but want to learn how to use Claude Code for verification work, you may begin with Chapter 2 and Chapter 3, then move to technique chapters as needed. The setup chapter is self-contained so you can get your environment working before diving into specific techniques.

The examples throughout use realistic code patterns drawn from common

scenarios: API services, data processing pipelines, authentication systems, financial calculations, stateful workflows, and concurrent operations. While no single example covers every language or framework, the principles and prompts transfer across technologies. Where behavior is language-specific or tool-specific, this is explicitly noted.

Let us begin.

Chapter 1: Foundations of Software Logic Verification

Before we can use Claude Code effectively for verification, we need a shared understanding of what verification actually means, how it differs from related concepts that are often confused with it, and where AI-assisted analysis fits in the broader landscape of software quality practices. This chapter establishes those foundations precisely. Getting these definitions right matters because they shape how we approach verification work, what tools we choose, and how we interpret results.

What Is Software Logic Verification

Software logic verification is the systematic examination of program code to determine whether its implementation correctly satisfies specified properties about its behavior. The word logic here refers to the decision-making structure of the code: conditionals that branch execution, loops that repeat operations, state transitions that change system behavior, calculations that transform data, and rules that govern valid operations. Verification asks whether this logic is correct according to some standard of correctness.

That standard of correctness comes from specifications. Specifications may be formal mathematical descriptions, as in safety-critical systems where requirements are expressed in temporal logic or algebraic specifications. More commonly in everyday software development, specifications are informal: natural language requirements documents, user stories, API documentation, design discussions, domain knowledge held by business stakeholders, and sometimes implicit expectations embedded in existing behavior that users depend on. Logic verification examines code against whatever specification is available, assessing whether the implementation matches what it is supposed to do.

Verification differs from simply reading code to understand it. It is goal-directed analysis focused on specific questions: Does this function correctly

handle all valid inputs? Are there paths through this code that violate business rules? Do these state transitions cover all possible scenarios? Is this calculation correct for edge cases involving very large numbers or unusual combinations of parameters? Each verification activity targets particular properties and produces evidence about whether those properties hold.

The output of verification is not a simple pass or fail verdict. It is an assessment: properties that are confirmed to hold, properties that appear to hold but lack conclusive evidence, properties that are violated with specific counterexamples showing the failure, and areas where the specification itself is unclear or incomplete. Good verification work improves both code correctness and specification clarity by exposing gaps in requirements that were never properly defined.

Verification Versus Validation: The V Model Perspective

The distinction between verification and validation is fundamental but frequently muddled in practice. The classic formulation captures it cleanly: verification asks whether we built the product right according to specifications, while validation asks whether we built the right product for the user's needs. Verification checks conformity to requirements. Validation checks fitness for purpose.

Consider a payment processing function. Verification would examine whether the function correctly implements the specified business rules: applying discounts only when conditions are met, calculating tax according to jurisdiction rules, handling currency conversion with proper rounding, rejecting invalid card numbers, and so on. It confirms that the code matches what was specified. Validation would involve testing the function with real payment scenarios, observing whether customers can successfully complete purchases, checking whether the overall checkout experience meets user expectations, and confirming that the processed payments actually settle correctly in banking systems. It confirms that the system works for its intended purpose.

The V model of software development illustrates this distinction structurally. On the left side of the V are verification activities corresponding to each development phase: requirements review verifies that requirements are complete and consistent, design review verifies that designs satisfy requirements,

code review verifies that implementation matches design. On the right side are validation activities at increasing levels of integration: unit testing validates individual components, integration testing validates component interactions, system testing validates the complete system, acceptance testing validates fitness for user needs. Each verification activity on the left has a corresponding validation activity on the right.

In practice this distinction is not always sharp. Code review can serve both purposes depending on what reviewers are checking. A reviewer examining whether a function matches its specification is doing verification. A reviewer asking whether the function's behavior makes sense from a user perspective is doing validation. Testing similarly straddles both: unit tests that check specific requirements are closer to verification, while end-to-end tests that simulate real user workflows are closer to validation.

For our purposes in this book, we focus primarily on verification activities: examining code logic against specifications to determine correctness. We assume that the specifications themselves have been validated through appropriate processes with stakeholders and users. If the specifications are wrong, no amount of verification will produce a correct system, because the code will correctly implement incorrect requirements.

How Verification Differs From Testing and Debugging

Testing and debugging are essential software quality practices but serve different purposes than verification. Understanding these differences helps you choose the right approach for each situation and combine them effectively.

Testing executes code with specific inputs and checks whether outputs match expected values. It is empirical: it observes actual behavior on concrete examples. Testing can demonstrate that bugs exist by finding inputs where behavior is wrong, but it cannot prove that no bugs remain because it never covers all possible executions. A function that processes user input may pass thousands of tests yet still fail for an input combination nobody thought to test.

Verification examines code without necessarily executing it, reasoning about what the code will do for all possible inputs and states within some defined domain. It is analytical rather than empirical. Verification can provide stronger guarantees than testing because it reasons about entire classes of inputs at once rather than individual examples. A verification that a sorting

function maintains array length for all possible inputs covers infinitely many cases through a single logical argument, whereas testing can only check finitely many specific arrays.

In practice the boundary is porous. Property-based testing combines both approaches by defining general properties and automatically generating many test inputs to check them. Formal model checking exhaustively explores all reachable states of a system model, which is verification in principle but implemented through systematic execution. AI-assisted analysis with Claude Code similarly blends approaches: it reads and reasons about code analytically like verification while also being able to run tests and commands empirically like testing.

Debugging differs from both testing and verification in its purpose. Debugging starts from a known failure and works backward to find its cause. It is reactive problem-solving triggered by observed incorrect behavior. Verification is proactive: it examines code before failures occur, searching for potential defects rather than investigating known ones. Testing is also proactive but less comprehensive than verification because it checks specific cases rather than general properties.

A practical analogy helps clarify the relationship. Imagine building a bridge. Testing is driving cars across it and checking that they make it to the other side. Verification is examining the engineering calculations, material specifications, and construction plans to determine whether the bridge will support all expected loads under all expected conditions. Debugging is investigating why a particular car had problems crossing. All three are necessary for a safe bridge, but they answer different questions.

Static Analysis, Dynamic Analysis, and Their Complementary Roles

Static analysis and dynamic analysis represent two fundamentally different approaches to examining software, each with strengths and limitations that make them complementary rather than competitive.

Static analysis examines code without executing it. Tools parse source files into abstract syntax trees, build control flow graphs representing possible execution paths, analyze data dependencies between variables, match patterns against known bug signatures, and apply mathematical reasoning about

program properties. Static analysis can examine all possible execution paths in principle, including paths that tests never trigger. It catches issues early in development, sometimes before code is even compiled or run. Common static analysis tools include linters like ESLint and Pylint, type checkers like TypeScript and MyPy, security scanners like SonarQube and Semgrep, and specialized analyzers for concurrency bugs or memory safety.

Dynamic analysis examines software during execution. Unit tests, integration tests, fuzzing, runtime monitoring, and profiling are all forms of dynamic analysis. Dynamic analysis observes actual behavior with real data in real environments. It catches issues that static analysis cannot predict: race conditions that only occur under specific timing, memory exhaustion from realistic workloads, incorrect interactions between components, performance degradation under load. Dynamic analysis validates that the system works as intended when actually used.

The key insight is that neither approach alone is sufficient. Static analysis may report theoretical issues that never occur in practice because certain code paths are unreachable given actual usage patterns. It may also miss bugs that depend on runtime conditions it cannot model precisely, such as external service behavior or user input distributions. Dynamic analysis validates real behavior but only for executed paths. Bugs in untested branches remain hidden until someone exercises those paths in production.

Claude Code operates across both categories depending on how you use it. When you ask it to analyze control flow through a function and identify potential issues, it is performing static analysis: reading code and reasoning about possible behaviors without execution. When you ask it to generate and run tests that exercise specific paths, it is facilitating dynamic analysis. The most effective verification workflows use both modes: Claude Code analyzes code statically to identify risky areas and missing coverage, then generates targeted tests that dynamically validate behavior in those areas.

Formal Methods and Where They Fit in Practice

Formal methods are mathematically rigorous techniques for specifying, developing, and verifying software systems. They use formal languages with precise semantics, mathematical logic for specifications, and automated or interactive proof tools to establish correctness guarantees. Examples include model

checkers like SPIN and TLA+ that exhaustively verify system properties by exploring state spaces, theorem provers like Coq and Isabelle where developers construct mathematical proofs of correctness, and deductive verification tools like Frama-C and Why3 that prove programs satisfy specifications using Hoare logic.

Formal methods provide the strongest possible correctness guarantees: if a formal proof is valid, the verified property holds for all executions without exception. This makes them invaluable for safety-critical systems where failures have catastrophic consequences: flight control software, medical device firmware, cryptographic protocols, railway signaling systems. Organizations like NASA, Airbus, and various defense contractors use formal methods routinely for critical components.

For most everyday software projects, full formal verification is impractical due to cost, complexity, and the expertise required. Writing formal specifications demands mathematical rigor that many teams lack. Proofs can be difficult to construct and maintain as code evolves. State space explosion makes exhaustive model checking infeasible for large systems. The return on investment is unclear when bugs are annoying rather than catastrophic.

This does not mean formal methods have nothing to offer typical software projects. Lightweight formal techniques can be highly effective: design by contract with preconditions and postconditions, type systems that prevent entire classes of errors, property-based testing that checks general properties across many inputs, model checking for small critical subsystems like state machines or consensus protocols. These approaches borrow ideas from formal methods without requiring full mathematical rigor.

Claude Code can assist with formal method activities at various levels. It can help write preconditions and postconditions for functions in design by contract style. It can suggest properties suitable for property-based testing. It can explain formal specifications written in temporal logic. It can even help construct proofs in theorem provers for users who understand the tools but need assistance with proof tactics. However, Claude Code itself is not a formal verification tool: its reasoning is probabilistic rather than mathematical, and its conclusions must be validated by proper formal tools when rigorous guarantees are required.

Property-Based Testing as a Bridge Between Worlds

Property-based testing occupies a unique position between traditional example-based testing and formal verification. Instead of writing individual test cases with specific inputs and expected outputs, property-based testing defines general properties that should hold for all valid inputs, then uses automated tools to generate many random inputs and check whether the properties hold.

For example, consider a function that sorts an array. Example-based tests might check that sorting [3, 1, 2] produces [1, 2, 3], and perhaps a few other specific arrays. Property-based testing defines properties like: the output length equals the input length, every element in the output appears in the input, the output is in non-decreasing order, and sorting twice produces the same result as sorting once. A property-based testing framework then generates thousands of random arrays and checks all these properties for each one.

The power of property-based testing lies in its ability to discover edge cases that human test designers might not anticipate. A developer writing example tests for a string parsing function might try typical inputs but forget about empty strings, strings with only whitespace, extremely long strings, or unusual Unicode characters. Property-based testing generates all these automatically and finds failures where the code does not handle them correctly.

Modern property-based testing frameworks include shrinking capabilities that reduce failing inputs to minimal counterexamples. If a test fails on a complex input with hundreds of elements, the framework systematically simplifies it until it finds the smallest input that still triggers the failure. This dramatically reduces debugging effort by pinpointing exactly what causes the problem.

Popular frameworks include QuickCheck for Haskell (the original), Hypothesis for Python, proptest for Rust, fast-check for JavaScript and TypeScript, and jqwik for Java. Each integrates with standard test runners and follows similar patterns: define properties as functions that take generated inputs and return boolean values indicating whether the property holds.

Claude Code is particularly effective at assisting with property-based testing because it excels at identifying what properties should hold for a given piece of code. You can show Claude Code a function and ask it to suggest properties

suitable for property-based testing, then have it generate the actual test code using your framework of choice. This combines Claude Code's analytical reasoning about code behavior with the rigorous automated checking that property-based testing provides.

Specification-Based Testing and Contract Programming

Specification-based testing derives test cases directly from formal or semi-formal specifications of expected behavior. Instead of designing tests based on implementation details, testers examine requirements documents, user stories, API contracts, domain rules, and other specifications to determine what the system should do, then create tests that verify each specified behavior.

Contract programming, also known as design by contract, is a specific form of specification-based approach where interfaces between components are defined with formal contracts consisting of preconditions, postconditions, and invariants. Preconditions specify obligations of callers: conditions that must be true before a function can be called. Postconditions specify guarantees of the callee: conditions that will be true after the function completes successfully. Invariants specify conditions that must always hold for an object or system regardless of which operations have been performed.

For example, a withdraw function in a banking system might have these contract elements: precondition that the account exists and is not frozen, precondition that the withdrawal amount is positive and does not exceed the available balance, postcondition that the new balance equals old balance minus withdrawal amount, invariant that no account balance can be negative. Specification-based testing would generate test cases covering each precondition (calling with non-existent account, calling with negative amount), each postcondition (verifying correct balance after withdrawal), and each invariant (ensuring balance never goes negative).

Many languages support contract programming through libraries or language features: Eiffel has it built in, Java supports JML annotations, C# has CodeContracts library, Python has the contracts package, Rust uses assert macros and result types. Even without formal contract tools, teams can document contracts informally in function comments and use assertions to enforce them at runtime.

Claude Code can assist with specification-based testing by analyzing requirements documents or existing code to extract implicit specifications,

suggesting preconditions and postconditions for functions, generating test cases that cover specified behaviors, and checking whether implementations satisfy documented contracts. This is particularly valuable for systems where business rules are complex and scattered across multiple sources: Claude Code can consolidate them into a coherent specification and derive comprehensive tests.

AI-Assisted Code Analysis: Capabilities and Honest Limitations

AI-assisted code analysis using large language models like those powering Claude Code represents a significant advance in practical verification capabilities, but it is important to understand both what these systems can do well and where they fall short.

Capabilities that Claude Code brings to verification work include the ability to read and reason about large amounts of code across multiple files and modules, trace complex execution paths through conditional logic and function calls, identify patterns that indicate potential bugs or design issues, suggest properties and invariants that should hold for given code, generate test cases targeting specific behaviors or edge cases, explain code behavior in terms accessible to humans who may not have written it, compare implementations against specifications expressed in natural language, detect inconsistencies between related pieces of code, and recommend improvements based on established best practices.

These capabilities are particularly valuable for tasks that require semantic understanding rather than pattern matching. Traditional static analysis tools excel at detecting known bug patterns: null pointer dereferences, unused variables, format string mismatches, common security vulnerabilities. Claude Code can go further by understanding what a piece of code is supposed to do based on its context, naming conventions, surrounding code, and natural language descriptions, then reasoning about whether it actually does that thing correctly.

However, there are honest limitations that must be acknowledged. Large language models generate text based on statistical patterns learned during training, not through rigorous logical deduction or mathematical proof. They can produce plausible-sounding analysis that is subtly wrong. They may

confidently assert that a property holds when in fact a counterexample exists. They may miss bugs that deterministic tools would catch reliably. They may hallucinate details about APIs, libraries, or language semantics that do not match reality.

Research studies on LLM-assisted code review show mixed results. Some studies find that models like Claude Sonnet achieve bug detection performance comparable to human reviewers but not clearly superior. Others find high rates of false positives where the model reports bugs that do not exist, or hallucinated issues based on misunderstanding the code context. A systematic review published in 2025 found that while LLMs show strong capabilities for identifying obvious defects and suggesting improvements, their reliability for detecting subtle logic errors remains inconsistent.

The practical implication is clear: AI-assisted analysis should augment rather than replace existing verification practices. Claude Code is an excellent collaborator that accelerates understanding, suggests things to check, generates tests, and provides perspectives you might not have considered. But its conclusions require validation through deterministic methods: running tests it suggests, checking findings against static analysis tools, verifying claims by reading the code yourself, and using formal methods for critical properties where appropriate.

This book teaches a methodology that leverages Claude Code's strengths while respecting its limitations. You will learn how to prompt it effectively for verification tasks, how to structure workflows that combine AI analysis with deterministic validation, how to detect when Claude Code may be giving unreliable guidance, and how to build confidence in verification results through multiple independent checks. The goal is not blind trust in AI but disciplined collaboration that produces better outcomes than either human or machine could achieve alone.

Chapter 2: Understanding Claude Code

Effective use of Claude Code for verification requires understanding the tool itself: what it is, how it works, what capabilities it provides, what constraints limit its operation, and how to configure it appropriately. This chapter gives you that understanding so you can use Claude Code confidently and correctly in your verification workflows.

What Is Claude Code and How It Relates to Claude Models

Claude Code is Anthropic's agentic coding tool that brings Claude's language model capabilities directly into your development environment through a command-line interface, IDE extensions, desktop applications, and web access. It is not merely a chatbot for asking questions about code. It is an agent that can read your project files, execute commands in your terminal, modify source code, run tests, manage git operations, connect to external tools, and perform multi-step workflows autonomously based on natural language instructions.

Claude Code runs Claude models from Anthropic's family of large language models. Different versions of Claude Code may use different underlying models depending on configuration and subscription tier. Generally, Claude Sonnet models handle routine tasks efficiently, while Claude Opus models are used for more complex reasoning when needed. You can often control which model is used through command-line flags or session commands. The specific model names and capabilities change over time as Anthropic releases new versions, so consult current documentation for the latest details.

The distinction between Claude Code and Claude accessed through a web chat interface is important. Web-based Claude has limited access to your local environment: you must manually copy and paste code, it cannot run commands on your machine, it cannot read files from your project without you providing them. Claude Code removes these friction points by running locally with direct

access to your filesystem, terminal, and development tools while maintaining the same conversational interface.

Claude Code is distributed as an official Anthropic product available through multiple installation methods. The recommended approach is a native installer that works on macOS, Linux, and Windows without requiring Node.js. Homebrew users can install via brew command, Windows users via WinGet, and all platforms support installation via curl script from claude.ai. npm-based installation has been deprecated in favor of these approaches.

Access to Claude Code requires either a Claude subscription plan (Pro, Max, Team, or Enterprise) or an Anthropic Console account with API access. Subscription plans provide included usage limits that vary by tier. API-based access charges per token consumed. For CI/CD integration, Claude Code supports authentication through cloud providers like Amazon Bedrock, Google Vertex AI, and Microsoft Foundry using OIDC identity federation to avoid storing static credentials in repositories.

The Workflow Model: Sessions, Context, and Conversations

Claude Code operates through sessions that represent individual working conversations with the agent. Each session starts fresh with its own context window and maintains conversation history within that window until it is closed or reset. Understanding how sessions work is essential for effective verification workflows because context management directly affects what Claude Code knows and can reason about.

When you start a session by running claude in your terminal, Claude Code initializes by reading configuration files and project context. It loads CLAUDE.md if present in the repository root or parent directories, which provides persistent project-specific instructions. It may load skills defined in skill directories, connect to configured MCP servers, and apply settings from settings.json files at various scopes. This initialization gives Claude Code awareness of your project's conventions, tools, and workflows before you ask any questions.

Within a session, Claude Code operates through an agentic loop: it gathers context by reading files and running commands as needed, takes action based on its analysis and your instructions, and verifies results by checking outputs

or running tests. You can interrupt this loop at any point to steer the conversation, provide additional information, or correct misunderstandings. The agent does not run completely autonomously without your involvement unless you explicitly configure it to do so through specific permission modes.

Context windows are finite resources that limit how much information Claude Code can hold in working memory at once. When a session's context approaches its limit, Claude Code automatically compacts by clearing old outputs and summarizing conversation history. You can manually manage context using commands like /clear to reset the conversation or /compact to summarize history aggressively. For verification work on large codebases, effective context management is critical: you want Claude Code to have relevant information loaded but not wastefully consuming tokens on irrelevant details.

Sessions are saved locally as JSONL files under ~/.claude/projects/ in your home directory. You can resume previous sessions using the -c flag to continue the most recent session or -r with a specific session identifier to resume an older one. Session resumption preserves conversation history and context, which is valuable for verification work that spans multiple interactions: you can pause analysis, investigate something separately, then return to where you left off without repeating earlier steps.

Claude Code supports parallel sessions for different tasks or different parts of a codebase. You can run multiple instances simultaneously in different terminal windows or use subagents to delegate specific subtasks while maintaining separate context windows. This is useful for verification workflows where you might want one session analyzing architecture while another generates tests, or where you are verifying multiple modules independently.

Project Context and Repository Awareness

Claude Code's ability to reason about your code depends on what project context it has access to. Unlike web-based AI tools that only see what you explicitly provide, Claude Code can explore your repository structure, read files on demand, examine git history, and build understanding of how different parts of your system relate. Understanding this capability and its limits is important for verification work.

When you start Claude Code in a project directory, it has access to all files within that directory tree unless restricted by configuration or permissions.

It can use file system tools to list directories, search for patterns in code, read specific files, and trace references across the codebase. For verification tasks, this means you can ask Claude Code questions like what functions call this critical authentication routine, where is user input validated before being stored in the database, or show me all places where we calculate pricing, and it will explore your repository to find answers rather than guessing based on training data.

CLAUDE.md files are the primary mechanism for providing persistent project context. These markdown files live in your repository and are automatically loaded into Claude Code's system prompt whenever a session starts in that project or its subdirectories. A well-written CLAUDE.md describes your project's architecture, coding conventions, build commands, testing framework, deployment process, and any non-obvious rules or constraints that Claude Code should know about. The /init command can auto-generate a starter CLAUDE.md based on codebase analysis, which you then refine with team-specific knowledge.

For monorepos with multiple projects in a single repository, CLAUDE.md files can exist at different levels: a root-level file provides shared context for all projects, while subdirectory files provide project-specific details. Claude Code loads CLAUDE.md files from the current directory upward through parent directories, accumulating context from each level. This layered approach lets you define organization-wide conventions once and override or extend them per project.

Claude Code is aware of git state: it knows which branch you are on, what files have been modified but not committed, what changes exist between commits or branches, and the commit history. This awareness enables verification workflows specific to version control: comparing code between versions to check for regressions, analyzing pull request diffs for logic errors, examining how a function evolved over time to understand its intended behavior, or verifying that changes on a feature branch are consistent with main branch conventions.

However, Claude Code's repository awareness has practical limits imposed by context windows and token constraints. It cannot hold an entire large codebase in active memory at once. For very large repositories with hundreds of thousands of lines of code, you need to guide Claude Code to focus on relevant areas rather than expecting it to understand everything simultaneously. Techniques for managing this include using CLAUDE.md files that summarize

architecture so Claude Code knows where to look, breaking verification tasks into focused subtasks targeting specific modules, using grep and glob tools to find relevant files before deep analysis, and employing subagents to handle different parts of the codebase in parallel with isolated contexts.

Configuration Options and Environment Setup

Claude Code's behavior is controlled through multiple configuration mechanisms that operate at different scopes with a defined precedence order. Understanding these configurations is essential for setting up Claude Code correctly for verification work and ensuring consistent behavior across your team.

Settings are stored in JSON files at four primary scopes. User-level settings live in ~/.claude/settings.json and apply to all projects for that user. Shared project settings live in .claude/settings.json within a project directory and are committed to version control so all team members get the same configuration. Project-local settings live in .claude/settings.local.json and are gitignored, allowing individual developers to override shared settings without affecting teammates. Managed settings are organization-enforced configurations that cannot be overridden by any other level, typically deployed through enterprise management systems.

The precedence order from highest to lowest is: managed settings override everything, then command-line arguments, then project-local settings, then shared project settings, then user-level settings. This hierarchy means that team standards defined in shared project settings can be overridden locally for experimentation but not accidentally committed, while organization policies take absolute priority.

Key configuration options relevant to verification work include permission modes that control whether Claude Code asks for approval before actions like file edits or command execution, model selection that determines which Claude model handles requests, tool restrictions that limit which capabilities Claude Code can use, hook configurations that define automated scripts triggered at lifecycle events, MCP server configurations that connect external tools, and environment variables injected into bash commands.

The /config menu command within a Claude Code session provides an interactive interface for viewing and modifying many settings without directly

editing JSON files. The /status command shows which configuration files are currently loaded and their effective values, which is useful for debugging when behavior does not match expectations. For verification workflows, you typically want to configure Claude Code with appropriate permissions that allow it to read files freely but require approval for potentially destructive actions, hooks that run quality checks after code modifications, and MCP connections to any static analysis or testing tools your team uses.

Claude Code also respects many environment variables that control behavior. ANTHROPIC_API_KEY provides API authentication when not using subscription-based access. CLAUDE_MODEL overrides the default model selection. Various CLAUDE_CODE_ prefixed variables control advanced behaviors like maximum concurrent subagents, budget limits, and feature flags. The official documentation maintains a current list of all supported settings keys and environment variables.

Permissions Model and Security Boundaries

Claude Code's permissions system is designed to balance its power as an autonomous agent with safety constraints that prevent unintended or malicious actions. For verification work, configuring permissions correctly matters both for security and for workflow efficiency: too restrictive and Claude Code cannot perform necessary analysis, too permissive and it might make unsafe changes.

Permission modes control the overall behavior regarding approval prompts. The default mode requires manual approval for most actions beyond reading files and searching code. Auto mode approves many operations automatically while still prompting for potentially dangerous actions like destructive shell commands or modifications to configuration files outside the project. Plan mode restricts Claude Code to planning and analysis without making changes, which is useful for exploratory verification before committing to specific actions. Other modes include acceptEdits that auto-approves file modifications within the project, dontAsk that minimizes prompts, and bypassPermissions that disables most permission checks entirely for trusted automation scenarios.

Permission rules provide fine-grained control beyond modes. Rules are defined in settings.json using a priority system where deny rules take precedence

over ask rules which take precedence over allow rules. You can specify rules for specific tools with wildcards: Bash(git commit *) allows git commits but not other bash commands matching that pattern, Read(.env) denies reading environment files, Edit(src/**) restricts file edits to the src directory. Rules use gitignore-style path patterns and are enforced by Claude Code's runtime rather than relying on the model to follow instructions.

For verification workflows, a sensible permission configuration allows Claude Code broad read access across the project so it can analyze code freely, permits running test commands and static analysis tools through bash, requires approval for file modifications so you review any suggested changes, and blocks access to sensitive files like secrets, credentials, or personal data directories. The exact configuration depends on your security requirements and how much autonomy you want Claude Code to have.

Workspace trust is a related security concept: Claude Code requires explicit workspace trust before applying project-level allow rules or accessing additional directories beyond the current project root. This prevents malicious repositories from automatically granting Claude Code excessive permissions when you first open them. In interactive sessions, you will be prompted to trust a workspace before certain configurations take effect. In non-interactive mode like CI/CD pipelines, repository settings are automatically trusted since the pipeline context itself is controlled by your organization.

Understanding these permission boundaries helps you design verification workflows that are both effective and safe. Claude Code can analyze your entire codebase for logic errors without risking accidental damage if permissions are configured appropriately. It can run tests and analysis tools automatically while still requiring your approval for any changes it proposes to make.

Available Tools and Integration Capabilities

Claude Code extends its capabilities through a system of tools that it can invoke during operation. Understanding what tools are available and how they work helps you design verification workflows that leverage the full range of Claude Code's abilities.

Built-in tools include file operations like Read for reading file contents, Write for creating or overwriting files, Edit for modifying existing files with string replacement, Glob for finding files matching patterns, and Grep for

searching file contents with regular expressions. Shell execution is provided through Bash which runs commands in your terminal. Web access tools allow Claude Code to search the internet and fetch web pages for research. Task tools enable spawning subagents for parallel work. These core tools cover most needs for code analysis and verification.

MCP servers extend capabilities beyond built-in tools by connecting to external services and applications through the Model Context Protocol, an open standard for AI tool integration. MCP servers can provide database access, API integrations, specialized analysis tools, custom workflows, and domain-specific functionality. You configure MCP servers using claude mcp add commands or configuration files, specifying transport methods like stdio for local processes or HTTP for remote services. For verification work, useful MCP servers might include static analysis tools, test runners, documentation generators, security scanners, or internal systems that provide context about your project.

Skills are reusable prompt templates and workflow definitions that Claude Code can invoke automatically when relevant. Skills are defined as SKILL.md files with YAML frontmatter specifying metadata like name, description, and allowed tools. When Claude Code detects that a task matches a skill's description, it loads the skill's instructions and follows its workflow. For verification, you might create skills for specific analysis patterns: a code review skill that systematically checks functions for common issues, a security audit skill that examines authentication logic, or a test generation skill that creates comprehensive test suites for given modules.

Subagents are isolated Claude Code instances spawned to handle specific subtasks with their own context windows. Subagents can be built-in agents like Explore and Plan, or custom agents defined in .claude/agents/ directories. When you delegate work to a subagent, it accumulates knowledge independently without polluting the parent session's context, then summarizes results when complete. For large-scale verification across multiple modules, subagents let you parallelize analysis: one subagent examines the authentication module while another reviews payment processing logic, each producing reports that you synthesize in the main conversation.

Hooks are automated scripts or handlers triggered at specific lifecycle events during Claude Code operation. PreToolUse hooks run before tool execution and can approve, deny, or modify tool calls based on their parameters. PostToolUse hooks run after tools complete and can perform

cleanup, validation, logging, or feedback injection. For verification workflows, hooks enable deterministic enforcement of quality standards: a PostToolUse hook that runs linters after every file edit ensures code always meets style requirements, a PreToolUse hook that blocks reading sensitive files protects secrets, a Stop hook that runs the test suite when Claude Code finishes a turn catches regressions immediately.

Token Limits, Context Windows, and Practical Implications

Claude Code operates within token limits that constrain how much information it can process in a single request and how much conversation history it can retain in a session. These limits have practical implications for verification work on large codebases that you must understand to use Claude Code effectively.

Tokens are the basic units that language models process, roughly corresponding to words or word fragments in text. Input tokens include your prompts, file contents Claude Code reads, and conversation history. Output tokens are Claude Code's responses. Different operations consume different amounts of tokens: reading a large source file consumes many input tokens, generating detailed analysis consumes output tokens, maintaining long conversations accumulates tokens over time.

Context windows define the maximum total tokens that can be active in Claude Code's working memory at once. When context approaches the limit, Claude Code automatically compacts by removing old outputs and summarizing earlier conversation turns. You can trigger compaction manually with /compact or reset entirely with /clear. For verification work, this means you cannot expect Claude Code to hold an entire large codebase plus extensive analysis history in memory simultaneously. You must structure work to keep relevant information accessible while discarding completed analysis that is no longer needed.

Token consumption has cost implications when using API-based access. Each token processed incurs a charge, so inefficient workflows that read unnecessary files or generate verbose output for tasks that could be handled concisely increase costs unnecessarily. Subscription plans include usage limits that may constrain how much you can use Claude Code within a billing period.

Understanding token costs helps you design efficient verification workflows that get maximum value from your access tier.

Practical strategies for managing token constraints in verification work include using CLAUDE.md files to provide concise project summaries rather than loading full documentation into every session, reading only relevant files when analyzing specific modules rather than the entire codebase, using grep and glob to find target files before reading them deeply, clearing context between distinct verification tasks with /clear, using subagents for parallel analysis of different areas so each has its own fresh context window, and summarizing completed analysis into notes files that can be selectively reloaded when needed rather than keeping everything in conversation history.

Known Limitations and Failure Modes

Understanding what Claude Code cannot do reliably is as important as understanding its capabilities. Awareness of limitations helps you design verification workflows that compensate for weaknesses and avoid over-reliance on the tool where it is likely to fail.

Hallucination is the most significant limitation: Claude Code may generate plausible-sounding analysis that is incorrect. It might claim a function handles edge cases correctly when examination reveals it does not, assert that a property holds across all inputs when counterexamples exist, reference APIs or libraries with wrong signatures, or invent details about code behavior that do not match reality. Hallucinations are more likely when Claude Code reasons about code without reading it directly, relies on training data rather than actual project context, or tackles problems outside its reliable capability range.

Reasoning errors occur even when Claude Code reads code correctly. Complex logical analysis involving many conditions, nested control flow, intricate state transitions, or subtle concurrency interactions can exceed Claude Code's reliable reasoning capacity. It might miss a bug that a careful human reviewer would find, or report a false positive where logic looks suspicious but is actually correct given the full context. Research studies consistently show that while LLMs perform well on straightforward code analysis tasks, their reliability degrades for complex reasoning involving many interacting factors.

Context sensitivity means Claude Code's analysis quality depends heavily on what information it has access to. If relevant files are not loaded

into context, if CLAUDE.md is incomplete or misleading, if domain-specific knowledge required to understand the code is not provided, Claude Code's analysis may be superficial or wrong. It cannot read your mind about implicit requirements or unstated assumptions. Verification work requires explicitly providing Claude Code with all necessary context: specifications it should check against, business rules that apply, constraints on valid inputs, and any domain knowledge needed to evaluate correctness.

Determinism limitations mean Claude Code is not a deterministic tool like a compiler or static analyzer. Running the same verification prompt twice may produce different results due to inherent randomness in language model generation. It might find a bug in one session but miss it in another. This nondeterminism makes Claude Code unsuitable as a standalone gate in CI/CD pipelines where consistent behavior is required, though it can complement deterministic tools by providing additional perspective that humans review.

Tool and environment constraints limit what Claude Code can verify dynamically. It runs commands in your environment subject to the same limitations you face: tests may fail due to infrastructure issues unrelated to code correctness, external services may be unavailable, database state may not match test expectations. Claude Code cannot distinguish between failures caused by logical bugs versus environmental problems without additional context or investigation guidance.

When Claude Code Is Appropriate and When It Is Not

Knowing when to use Claude Code for verification and when to rely on other methods is critical for building effective quality practices. Claude Code excels in some scenarios and should not be the primary tool in others.

Claude Code is highly appropriate for exploratory analysis of unfamiliar code: understanding how a legacy system works before modifying it, tracing complex execution paths through multiple modules, identifying potential issues in new code before formal review, generating initial test suites that humans then refine, suggesting properties and invariants that should hold for given code, reviewing pull requests as an additional perspective alongside human reviewers, investigating bugs by analyzing code paths that might cause observed failures, and documenting implicit behavior by examining what code actually does versus what documentation claims.

Claude Code is also valuable for tasks requiring semantic understanding beyond pattern matching: verifying that business rules are correctly implemented across the codebase based on natural language requirements, checking consistency between API contracts documented in prose and actual implementations, analyzing whether refactoring preserved behavioral semantics, reviewing security-sensitive logic for subtle vulnerabilities, and examining complex workflows to identify missing error handling or incomplete state transitions.

Claude Code is not appropriate as a replacement for deterministic verification tools. It should not replace static analysis tools that reliably detect specific bug classes like null pointer dereferences or memory leaks. It should not replace type checkers that guarantee type safety properties. It should not replace formal verification tools when mathematical proof of correctness is required for safety-critical systems. It should not replace comprehensive automated test suites that provide regression protection through repeated execution.

Claude Code is also not appropriate as a final authority on correctness claims without independent validation. Any assertion it makes about code behavior should be verified by reading the relevant code yourself, running tests that exercise the claimed behavior, checking against static analysis results, or using formal methods for critical properties. Treating Claude Code's conclusions as definitive without validation invites accepting hallucinated analysis as fact.

The right approach is to view Claude Code as one component in a layered verification strategy: deterministic tools provide reliable baseline checks for well-understood bug patterns, automated tests validate behavior through repeated execution, human expertise provides judgment about complex design decisions and business logic correctness, and Claude Code adds analytical depth, semantic understanding, exploratory capability, and scale that none of the other components alone can provide. Used together with appropriate validation of its output, Claude Code significantly strengthens your overall verification practice without introducing unacceptable risk from its limitations.

Chapter 3: Setting Up Your Verification Environment

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

Prerequisites and System Requirements

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

Installing Claude Code and Verifying the Installation

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

Preparing Your Development Environment

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

Repository Setup and Project Structure Recommendations

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

Claude Code Configuration Files and Their Purpose

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

Permissions Configuration for Verification Tasks

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

Creating Reusable Instructions and Prompt Templates

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

Automation Scripts and Workflow Helpers

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

Integrating With Existing Tooling: Linters, Type Checkers, Test Frameworks

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

Verifying Your Complete Setup Works End-to-End

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

Troubleshooting Common Failures

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

Chapter 4: Planning a Verification Workflow

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

Assessing Verification Needs for Your Project

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

Risk-Based Prioritization of Code Areas

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

Extracting Requirements and Business Rules From Documentation

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

Deriving Testable Properties and Invariants

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

Designing Verification Scope: Functions, Modules, Services, Repositories

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

Building a Verification Plan Document

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

Adapting the Approach to Legacy Systems Versus Greenfield Projects

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

Chapter 5: Understanding an Unfamiliar Codebase With Claude Code

Initial Repository Exploration and High-Level Architecture Discovery

Mapping Dependencies and Component Relationships

Identifying Entry Points, Public APIs, and Critical Paths

Understanding Data Models and State Management

Tracing Feature Implementation Across Files

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

Building a Mental Model With Claude Code as Guide

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

Validating Your Understanding Against the Code

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

Chapter 6: Analyzing Control Flow and Data Flow

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

Tracing Control Flow Through Complex Conditionals and Branches

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

Mapping Decision Trees and State Transitions

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

Following Data Flow Across Function Boundaries

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

Identifying Unreachable Code and Dead Paths

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

Detecting Missing Error Handling in Control Paths

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

Analyzing Loops, Recursion, and Termination Conditions

Chapter 7: Identifying Invariants, Properties, and Preconditions

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

What Invariants Are and Why They Matter for Verification

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

Classifying Types of Invariants: Object, Loop, Class, System-Level

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

Using Claude Code to Identify Implicit Invariants in Existing Code

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

Deriving Preconditions and Postconditions From Function Signatures

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

Extracting Business Rule Constraints as Formal Properties

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

Validating That Identified Invariants Are Actually Enforced

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

Chapter 8: Verifying Algorithms and Calculations

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

Step-by-Step Algorithm Walkthroughs With Claude Code

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

Verifying Edge Cases in Sorting, Searching, and Traversal Algorithms

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

Checking Mathematical Correctness of Formulas and Computations

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

Validating Financial Calculations and Monetary Logic

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

Verifying Hash Functions, Checksums, and Cryptographic Usage

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

Cross-Checking Implementations Against Reference Specifications

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

Identifying Off-by-One Errors, Boundary Issues, and Precision Problems

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

Chapter 9: Analyzing State Machines and State Transitions

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

Identifying Implicit State Machines in Existing Code

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

Mapping All Possible States and Transitions

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

Verifying That Invalid Transitions Are Prevented

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

Detecting Missing or Unhandled States

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

Analyzing Concurrency Effects on State Transitions

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

Validating Persistence Layer State Consistency

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

Building Explicit State Machine Representations From Implicit Code

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

Chapter 10: Validating API Contracts and Data Transformations

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

Extracting Contract Specifications From API Documentation and Signatures

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

Verifying Input Validation and Parameter Checking

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

Analyzing Return Value Correctness and Error Response Handling

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

Tracing Data Transformations Through Processing Pipelines

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

Validating Serialization and Deserialization Logic

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

Checking Compatibility Between Client and Server Contracts

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

Verifying Version Migration and Backward Compatibility Logic

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

Chapter 11: Investigating Concurrency and Asynchronous Behavior

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

Identifying Concurrent Execution Paths in Code

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

Analyzing Shared Mutable State Access Patterns

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

Detecting Potential Race Conditions and Data Races

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

Verifying Lock Acquisition and Release Patterns

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

Analyzing Asynchronous Control Flow and Callback Chains

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

Checking Timeout Handling and Retry Logic Correctness

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

Validating Event Ordering Assumptions

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

Chapter 12: Assessing Security-Sensitive Logic

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

Identifying Security-Relevant Code Paths and Decision Points

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

Verifying Authentication and Authorization Logic Correctness

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

Analyzing Input Validation for Injection and Abuse Vectors

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

Checking Cryptographic Usage Patterns for Logical Errors

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

Validating Session Management and Token Handling Logic

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

Reviewing Access Control Decisions and Permission Checks

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

Detecting Business Logic Vulnerabilities and Abuse Scenarios

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

Chapter 13: Verifying Complex Business Logic

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

Extracting Business Rules From Requirements and Domain Knowledge

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

Mapping Business Rules to Code Implementation

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

Verifying Consistency of Rule Application Across the Codebase

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

Analyzing Conditional Business Logic for Completeness

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

Checking Date and Time Handling in Business Calculations

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

Validating Multi-Step Business Processes and Workflows

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

Ensuring Regulatory and Compliance Requirements Are Met in Code

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

Chapter 14: Discovering Edge Cases and Corner Conditions

Systematic Edge Case Enumeration Techniques

Analyzing Boundary Values for Numeric and String Inputs

Identifying Rare but Valid Input Combinations

Checking Behavior With Empty, Null, and Missing Data

Verifying Behavior Under Resource Constraints

Discovering Interaction Edge Cases Between Components

Chapter 15: Generating Verification Tests From Claude Code Analysis

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

Translating Verified Properties Into Test Cases

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

Generating Unit Tests That Cover Identified Logic Paths

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

Creating Property-Based Tests From Discovered Invariants

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

Building Integration Tests for Cross-Component Verification

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

Writing Negative Tests and Failure Mode Tests

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

Reviewing Generated Tests for Completeness and Correctness

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

Integrating Generated Tests Into Existing Test Frameworks

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

Chapter 16: Pull Request Verification Workflows

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

Setting Up Automated PR Review With Claude Code

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

Analyzing Changed Files for Logic Errors and Regressions

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

Verifying That Tests Adequately Cover New Logic

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

Checking Consistency With Existing Code Patterns

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

Validating Documentation Matches Implementation Changes

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

Handling Large PRs and Incremental Verification

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

Building a Human-AI Collaborative Review Process

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

Chapter 17: Regression Prevention and Refactoring Verification

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

Establishing a Baseline of Known-Correct Behavior

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

Verifying That Bug Fixes Address Root Cause Without Side Effects

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

Checking Refactored Code Against Original Logic Semantics

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

Using Claude Code to Compare Before and After Implementations

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

Identifying Potentially Affected Areas Outside Changed Files

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

Building Regression Verification Into Your Development Workflow

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

Tracking Known Issues and Their Verification Status

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

Chapter 18: Repository-Wide and Monorepo Verification Strategies

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

Strategies for Large-Scale Codebase Analysis

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

Managing Context Limits Across Multiple Files and Repositories

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

Prioritizing Verification Effort Across a Monorepo

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

Orchestrating Multi-Step Verification Tasks

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

Building Incremental Verification Pipelines

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

Aggregating Findings Across the Entire Repository

Maintaining Verification Coverage Over Time

Chapter 19: Validating Claude Code's Own Output

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

Why You Must Never Trust Claude Code Blindly

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

Common Failure Modes and Hallucination Patterns in Verification Tasks

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

Techniques for Detecting Flawed Reasoning in AI Output

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

Requiring Evidence: Asking Claude Code to Show Its Work

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

Cross-Checking Findings With Deterministic Tools

Independent Validation Procedures for Critical Claims

Knowing When AI-Assisted Verification Is Insufficient

Escalation Paths: When Formal Methods and Expert Review Are Necessary

Chapter 20: Integrating Into CI/CD and Team Workflows

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

Designing CI/CD Verification Gates With AI Assistance

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

Automating Claude Code Invocation in Build Pipelines

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

Generating Automated Verification Reports

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

Building Audit Trails for Compliance and Review

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

Team Processes for Human-AI Collaborative Verification

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

Training Teams on Effective Verification Prompt Patterns

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

Measuring Impact: Metrics for Verification Effectiveness

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

Chapter 21: Security, Operations, and Safeguards

Managing Permissions for Verification Tasks

Protecting Secrets and Sensitive Configuration

Handling Proprietary and Confidential Source Code

Guarding Against Prompt Injection in Repository Files

Detecting Malicious or Manipulative Repository Instructions

Supply Chain Risks in AI-Assisted Verification

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

Auditability and Reproducibility of Verification Results

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

Safeguards for Automated Changes and Recommendations

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

Chapter 22: Advanced Patterns, Optimization, and Best Practices

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

Building a Reusable Prompt Library for Verification Tasks

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

Creating Custom Claude Code Workflows and Commands

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

Orchestrating Multiple Tools in Verification Pipelines

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

Optimizing Context Usage and Token Efficiency

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

Decision Frameworks: When to Use Which Technique

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

Common Failure Modes and Troubleshooting Procedures

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

Version-Dependent Behavior and Migration Considerations

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

Best Practices Summary for Production-Scale Adoption

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

Conclusion: The Future of AI-Assisted Software Verification

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

References

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

Claude Code Documentation and Announcements

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

Software Verification Fundamentals

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

AI-Assisted Code Analysis Research

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

Specific Verification Techniques

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

Static Analysis and Linting Tools

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

Testing Frameworks and Tools

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

CI/CD and Pipeline Integration

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

Security and Prompt Injection Research

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

Formal Methods and High-Assurance Verification

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

Development Practices and Methodology

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

Additional Resources

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