REVERSE ENGINEERING WITH CLAUDE CODE

AI-ASSISTED ANALYSIS, UNDERSTANDING, AND RECONSTRUCTION OF SOFTWARE SYSTEMS

An **authorization-first** methodology for using Claude Code to:

- UNDERSTAND
- DOCUMENT
- MIGRATE & INTEROPERATE
- DEBUG & RECOVER
- AUDIT
- MODERNIZE
- RECONSTRUCT

RIGOROUS. EVIDENCE-DRIVEN. REPRODUCIBLE.

EVIDENCE QUALITY • LEGAL BOUNDARIES • TECHNICAL ACCURACY • REPRODUCIBILITY

STEVE T.

Reverse Engineering with Claude Code

AI-Assisted Analysis, Understanding, and Reconstruction of Software Systems

Steve Publications

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

This version was published on 2026-08-25

Contents

AI-Assisted Analysis, Understanding, and Reconstruction of Software Systems

This book teaches an authorization-first methodology for using Claude Code as an AI-assisted engineering tool to understand, document, migrate, interoperate with, debug, recover, audit, modernize, and reconstruct software systems that you own or are explicitly authorized to analyze. It progresses from foundational reverse-engineering concepts through advanced workflows while maintaining rigorous attention to evidence quality, legal boundaries, technical accuracy, and reproducibility. The reader will learn how to combine Claude Code's codebase comprehension capabilities with established analysis tools to accelerate legitimate software understanding and reconstruction work.

Introduction: The Discipline of Understanding Software

The Undocumented System Problem

Every organization reaches a moment when its own systems become strangers. A critical service runs on code written by someone who left three years ago. A proprietary file format must be read by a new application, but the original creator went out of business. An internal API client has been discontinued and needs to be rebuilt from scratch. A production bug appears in a component whose behavior was never fully specified. These are not edge cases; they are routine challenges in software maintenance, migration, interoperability engineering, and legacy modernization.

The systems we inherit rarely come with complete documentation. Even when they do, documentation drifts away from reality the moment the first undocumented change is deployed. Over months and years of incremental modifications, hotfixes, workarounds, and emergency patches, the gap between what is written down and what actually runs grows until the documentation becomes more misleading than its absence would have been.

Reverse engineering is the disciplined response to this problem. It is not a last resort for people who failed to document their work; it is a fundamental engineering capability that every organization needs, regardless of how well it documents things today. Because tomorrow someone will inherit your systems, and they will need to understand them whether you left notes or not.

Reverse Engineering as Evidence Collection

Reverse engineering is often misunderstood as code generation or automated analysis. It is neither. At its core, reverse engineering is a process of evidence collection and hypothesis testing that resembles scientific inquiry more than it resembles programming. You observe behavior, form hypotheses about what

causes that behavior, test those hypotheses through controlled experiments, revise your understanding based on the results, and repeat until you have a model of the system that is accurate enough for your purpose.

The quality of reverse engineering work depends entirely on the quality of its evidence. A correct conclusion supported by weak evidence is worthless because you cannot tell it apart from an incorrect conclusion when the next problem arises. A strong conclusion requires multiple independent lines of evidence: source code that shows how something works, test results that confirm what happens at runtime, tool output that reveals structure and dependencies, documentation that explains intent even if it is outdated, and observations from controlled experiments that demonstrate actual behavior under specific conditions.

This book treats reverse engineering as a rigorous methodology rather than a collection of tricks or tools. The workflow we develop covers authorization and scope definition, preservation of originals, hashing and provenance tracking, environment isolation, inventory and reconnaissance, repository mapping, dependency discovery, build-system identification, entry-point discovery, architecture reconstruction, data-flow and control-flow analysis, interface discovery, configuration analysis, runtime observation in controlled environments, hypothesis generation, targeted experiments, evidence collection, validation, documentation, implementation or migration, testing, and final reporting. Each phase produces artifacts that support the next phase and create a record of what was done and why.

Claude Code enters this methodology as an assistant for organizing evidence, explaining tool outputs, generating testable hypotheses, writing analysis scripts, creating tests, documenting findings, and accelerating repetitive engineering work. It is not a substitute for authoritative tool output or human verification, and treating it as one is the single most common failure mode in AI-assisted reverse engineering.

What Claude Code Actually Does

Claude Code is an agentic coding assistant built by Anthropic that runs in your terminal, reads codebases, edits files, executes commands, and integrates with development tools through natural language interaction. It operates across multiple surfaces including the command line, VS Code and JetBrains

extensions, a desktop application, and a web interface, all sharing the same underlying engine.

For reverse engineering work, Claude Code's relevant capabilities include:

- **Codebase exploration**: Claude Code can read files, search for patterns using grep-like operations, find files matching glob patterns, and navigate directory structures to understand how a project is organized.
- **Cross-file reasoning**: When analyzing a codebase, Claude Code can hold multiple files in context simultaneously, trace function calls across modules, follow data through transformations, and identify relationships between components that would be tedious to discover manually.
- **Tool execution**: Claude Code can run shell commands, invoke analysis tools, execute tests, and capture their output, then interpret the results and suggest next steps based on what those tools reveal.
- **Documentation generation**: Claude Code can produce architecture summaries, component inventories, dependency explanations, interface catalogs, data-flow descriptions, technical documentation, and migration plans based on its analysis of code and tool outputs.
- **Script writing**: Claude Code can write analysis scripts in Python or other languages to automate repetitive examination tasks, generate test harnesses for behavioral characterization, create parsers for unknown formats, or build compatibility layers between systems.
- **Hypothesis generation**: When presented with observations from static analysis, dynamic tracing, or black-box testing, Claude Code can propose hypotheses about what is happening and suggest experiments that would confirm or falsify those hypotheses.

What Claude Code cannot do reliably includes:

- **Guaranteeing correctness**: Claude Code produces outputs based on patterns learned during training combined with the context you provide. It does not execute code to verify its claims, it does not independently check facts against external sources unless you explicitly ask it to run tools that do so, and it can confidently state incorrect information when its training data is incomplete or misleading.
- **Replacing specialized analysis tools**: Claude Code is not a disassembler, decompiler, debugger, profiler, or protocol analyzer. It can help you use those tools and interpret their output, but it cannot substitute for the authoritative results they produce.

- **Inferring behavior from incomplete evidence**: When asked to explain how something works based on partial information, Claude Code will often fill gaps with plausible-sounding speculation drawn from its training data rather than admitting uncertainty. This is particularly dangerous in reverse engineering where fabricated call paths or invented APIs can send an investigation down the wrong track for hours or days.
- **Operating without authorization**: Claude Code does not determine whether you are authorized to analyze a system; that responsibility belongs entirely to you. It will assist with analyzing any code or artifacts you provide, and it is your obligation to ensure that doing so complies with applicable law, contractual obligations, organizational policy, and ethical standards.

Understanding these capabilities and limitations is essential before using Claude Code for reverse engineering work. The workflows in this book are designed to leverage what Claude Code does well while building safeguards around where it fails.

Authorization, Safety, and Scope

This book teaches reverse engineering as a lawful, authorization-first practice. Every analysis project begins with a clear determination of what you are authorized to examine and why. The legitimate use cases we cover include:

- Analyzing software you own or maintain to understand its architecture, fix bugs, add features, or prepare for migration
- Examining systems your organization operates under explicit authorization from the owner
- Studying open-source software whose license permits analysis
- Investigating software for security auditing under written authorization
- Creating interoperable implementations through clean-room design where permitted by law and contract
- Recovering functionality from legacy systems to preserve organizational capability
- Documenting undocumented systems to improve maintainability and reduce operational risk

This book explicitly excludes reverse engineering for the purpose of:

- Developing malware or other malicious software
- Stealing credentials, secrets, or sensitive data
- Gaining unauthorized access to systems or services
- Bypassing DRM, license enforcement, or other access controls
- Creating stealth mechanisms or persistence tools
- Exploiting vulnerabilities without authorization
- Circumventing authentication or encryption for unauthorized purposes
- Compromising third-party systems without explicit permission

The distinction between lawful and unlawful reverse engineering is not always obvious. Jurisdiction matters: what is permitted in one country may be prohibited in another. Contracts matter: an end-user license agreement may restrict analysis even where statute permits it. Authorization matters: examining a system you do not own or control, even for benign purposes, can constitute unauthorized access under computer crime laws. This book provides a framework for thinking about these questions but cannot substitute for qualified legal advice on specific projects.

Safety is equally important. Reverse engineering involves executing unknown code, analyzing potentially malicious artifacts, and working with systems whose behavior may be unpredictable. The workflows in this book assume environment isolation through containers or virtual machines, read-only analysis where possible, backups of all original artifacts, careful review of any commands before execution, and least-privilege access throughout the investigation.

How to Use This Book

This book is organized as a progressive learning path from fundamentals to advanced practice. Each chapter builds on material from previous chapters, so reading in order is recommended even if you are experienced with some topics. The structure follows the lifecycle of a reverse engineering project: foundational concepts and legal framework first, then tool setup and configuration, then methodology and workflow, then specific analysis techniques for source code, binaries, protocols, and file formats, then reconstruction and modernization strategies, then quality assurance and professional practice.

The book assumes comfort with programming, command-line tools, version control, and basic systems concepts including memory, processes, networking, and operating-system interfaces. It does not assume prior reverse-engineering experience; terminology is introduced before use and explained when first encountered. Where specialized knowledge is required, the relevant background is provided inline.

Every substantial topic includes complete, executable code examples with specified languages, versions, dependencies, build commands, execution instructions, expected outputs, and platform assumptions. Case studies demonstrate end-to-end workflows from initial investigation through verified understanding and final deliverables. All examples use safe, purpose-built programs designed for educational analysis rather than security testing of real targets.

Throughout the book you will encounter distinctions between different types of information:

- **Static evidence**: Facts derived from examining code, binaries, or artifacts without executing them (file structure, symbols, strings, imports, source code patterns)
- **Dynamic evidence**: Facts derived from observing behavior during execution (runtime output, system calls, memory state, network traffic, timing)
- **AI-generated hypotheses**: Explanations proposed by Claude Code that require verification against evidence
- **Human conclusions**: Judgments made by the analyst after evaluating evidence and AI suggestions
- **Verified findings**: Claims supported by multiple independent lines of evidence that have been tested and confirmed

Maintaining these distinctions is central to the methodology. A finding is only as strong as its supporting evidence, and the strongest findings are those that can be reproduced by someone else following the same steps with the same artifacts.

The final chapter synthesizes everything into a reusable playbook: authorization-first workflow templates, project structure recommendations, prompt patterns for common tasks, evidence-ledger templates, checklists for analysis and verification, troubleshooting guides, clean-room procedures, and security controls. This is designed to be practical reference material you can adapt for real projects.

Use this book as both a learning resource and a methodology guide. Read it through once to understand the full picture, then return to specific chapters as needed when working on actual reverse engineering tasks. The examples are meant to be run and modified; the workflows are meant to be adapted to your environment and requirements. And always remember: Claude Code is an assistant that can accelerate your work, but the responsibility for correct analysis, lawful conduct, and verified conclusions belongs to you.

Chapter 1: Foundations of Reverse Engineering

What Is Reverse Engineering

Reverse engineering is the process of analyzing a system to understand its structure, behavior, and operation by examining its outputs, artifacts, or implementation rather than relying on design documents or specifications that may not exist. In software reverse engineering specifically, this means studying source code, compiled binaries, network protocols, file formats, configuration files, runtime behavior, and other artifacts to reconstruct knowledge about how a system works.

The term "reverse" refers to the direction of analysis: instead of starting with requirements and producing an implementation (the forward engineering path), you start with an existing implementation and work backward to understand what it does, how it does it, and why it was built that way. This is not merely reading code; it is systematic investigation aimed at recovering knowledge that was never documented, was lost over time, or exists only implicitly in the behavior of running systems.

Reverse engineering applies across multiple levels of abstraction:

- **Source-level analysis**: Examining readable source code to understand architecture, data flow, control flow, dependencies, and undocumented behavior when you have access to the implementation but lack documentation or domain knowledge.
- **Binary-level analysis**: Studying compiled executables to recover algorithms, data structures, protocols, and functionality when source code is unavailable. This involves disassembly (converting machine code to assembly language), decompilation (reconstructing higher-level pseudocode from assembly), and dynamic analysis (observing runtime behavior).

- **Protocol analysis**: Observing network communication to understand message formats, state machines, error handling, versioning, and interoperability requirements.
- **File format analysis**: Examining binary or structured data files to discover header structures, field layouts, encoding schemes, compression methods, and validation rules.
- **Behavioral analysis**: Characterizing input-output relationships, boundary conditions, performance characteristics, and failure modes through systematic testing of a system whose internals are not fully known.

All of these share the same underlying methodology: observe, hypothesize, test, revise. The tools differ depending on what you are analyzing, but the intellectual process is consistent.

Legitimate Use Cases and Goals

Reverse engineering serves many legitimate purposes in software engineering, operations, security, and interoperability work. Understanding which goals are valid helps frame analysis projects correctly and ensures that effort is directed toward outcomes that matter.

Understanding inherited systems: The most common use case is gaining comprehension of codebases you did not write and for which documentation is incomplete or missing. New team members joining a project, maintainers taking over abandoned services, and organizations acquiring software through mergers all face this challenge regularly. Claude Code excels here because it can rapidly ingest large amounts of source code, trace relationships between components, explain unfamiliar patterns, and generate architecture summaries that accelerate onboarding.

Legacy modernization: When migrating aging systems to new platforms, languages, or architectures, you must first understand what the existing system does before changing how it is implemented. Reverse engineering identifies business rules embedded in code, maps integration points with external systems, characterizes data transformations, and documents behaviors that would be lost in a naive rewrite. This understanding guides migration strategy decisions and provides the specification against which modernized implementations are validated.

Interoperability: Building software that works with existing systems often requires understanding proprietary or undocumented interfaces. Reverse engineering protocols, file formats, and APIs enables clean-room implementation of compatible clients, servers, translators, and adapters. The legal framework for interoperability reverse engineering is well established in many jurisdictions, provided the work follows proper procedures and respects intellectual property boundaries.

Debugging and incident response: When a system fails in production, understanding why often requires analyzing code paths that were not exercised during testing, examining crash dumps from systems with limited observability, or reconstructing sequences of events from logs and artifacts. Reverse engineering techniques help trace bugs through complex call chains, identify race conditions and resource leaks, and determine root causes when the failure mechanism is not immediately apparent.

Security auditing: Authorized security assessments involve analyzing software to identify vulnerabilities, misconfigurations, weak cryptographic implementations, insecure dependencies, and design flaws that could be exploited. This requires understanding both the intended behavior of a system and how it might behave under adversarial conditions. Reverse engineering provides the depth of understanding needed for thorough security analysis.

Compliance and audit: Regulatory requirements increasingly demand visibility into software supply chains, data processing flows, cryptographic implementations, and access control mechanisms. Reverse engineering can verify that systems comply with stated policies, identify unintended data flows, confirm that deprecated algorithms have been removed, and document security-relevant properties for auditors.

Data recovery and format migration: When file formats become obsolete or proprietary tools are discontinued, reverse engineering the format specification from sample files enables creation of readers, writers, and converters that preserve access to stored information. This is critical for archival purposes, regulatory compliance with data retention requirements, and preventing vendor lock-in.

Performance optimization: Understanding how a system actually behaves under load often requires analyzing runtime characteristics that are not visible from source code alone: memory allocation patterns, I/O behavior, concurrency bottlenecks, cache utilization, algorithmic complexity in practice rather

than theory. Reverse engineering combined with profiling reveals optimization opportunities that static analysis misses.

Each of these use cases shares a common requirement: the analyst must have legitimate authorization to examine the target system. The technical work may be straightforward; doing it without proper authorization transforms a valid engineering task into a legal violation.

Legal Landscape and Compliance

The legality of reverse engineering varies significantly by jurisdiction, subject matter, contractual terms, and specific circumstances. This section provides an overview of key legal principles but cannot substitute for qualified legal advice on any particular project.

Copyright law and fair use: In the United States, copyright protects the expression of ideas in software (the specific code) but not the underlying ideas, algorithms, or functional behavior themselves. The Supreme Court's ruling in Baker v. Selden established that functional knowledge cannot be monopolized through copyright. Reverse engineering to understand functionality is generally considered fair use because it extracts unprotected ideas rather than copying protected expression.

The Ninth Circuit's decision in Sega Enterprises v. Accolade (1992) confirmed that disassembling software for the purpose of achieving interoperability constitutes fair use. The court held that when the only way to access unprotected elements is through reverse engineering, and there is a legitimate need for those elements, the process is protected. This precedent supports interoperability-focused reverse engineering across many contexts.

DMCA anti-circumvention provisions: The Digital Millennium Copyright Act introduced Section 1201, which prohibits circumventing technological protection measures that control access to copyrighted works. This created tension with traditional reverse engineering practices because many modern systems use encryption, obfuscation, or license checks that must be bypassed to analyze their internals.

Section 1201(f) provides a specific exemption for reverse engineering aimed at interoperability: lawful possessors of a program may circumvent protection measures solely to identify and analyze elements necessary to achieve interoperability of an independently created program. The requirements are strict:

you must lawfully obtain the copy, the sole purpose must be interoperability analysis, and the information obtained must be used only for that purpose.

The U.S. Copyright Office also issues triennial exemptions to Section 1201's general prohibitions. Recent rulemaking cycles have included exemptions for security research, repair and maintenance of devices, and preservation of abandoned software, among others. These exemptions are time-limited and subject to change, so current status must be verified for specific projects.

Contractual restrictions: End-user license agreements (EULAs) and software licenses frequently contain clauses prohibiting reverse engineering, decompilation, or disassembly. The enforceability of such clauses varies by jurisdiction and context. In the United States, the Fourth Circuit's decision in SAS Institute v. World Programming Ltd held that contractual prohibitions on reverse engineering can be enforceable under state contract law even where copyright law would permit the activity. This means that even if a particular analysis is legal under copyright statute, violating a license agreement may create separate liability.

Trade secret protection: Reverse engineering of products obtained through lawful means is generally recognized as a legitimate method of discovering trade secrets in many jurisdictions, including the United States under the Uniform Trade Secrets Act and Defend Trade Secrets Act. However, this protection does not extend to reverse engineering conducted through misappropriation (theft, breach of contract, espionage) or when contractual obligations specifically prohibit it.

International considerations: The European Union's Software Directive (2009/24/EC) explicitly permits decompilation for interoperability purposes under conditions similar to the U.S. fair use doctrine. The ECJ's ruling in SAS Institute v. World Programming Ltd established that copyright does not extend to software functionality, programming languages, or file formats within the EU. Other jurisdictions have their own frameworks, and some are more restrictive than others.

Computer crime laws: Many jurisdictions have laws criminalizing unauthorized access to computer systems. Reverse engineering that involves connecting to systems you do not own or control, even for benign analysis purposes, may constitute unauthorized access under statutes like the U.S. Computer Fraud and Abuse Act (CFAA). This is particularly relevant for protocol reverse engineering involving live services.

Practical compliance guidance: Before beginning any reverse engineering project:

1. Verify ownership or obtain explicit written authorization from the system owner
2. Review applicable license agreements, terms of service, and contractual obligations
3. Consult qualified legal counsel when intellectual property boundaries are uncertain
4. Limit analysis to what is necessary for your stated legitimate purpose
5. Maintain clear documentation of authorization and scope throughout the project
6. Avoid redistributing proprietary code or artifacts unless explicitly permitted
7. When creating interoperable implementations, use clean-room procedures to minimize IP risk

The Authorization Model

Authorization is the foundation upon which all legitimate reverse engineering rests. Without it, technically sound analysis becomes unlawful activity. This section defines a practical authorization model that applies across different project types and organizational contexts.

Levels of authorization:

- **Full ownership**: You or your organization owns the software outright (internally developed, purchased with full rights, open-source under permissive license). Analysis is generally unrestricted except by third-party dependencies' licenses.
- **Explicit written authorization**: The owner has granted permission for analysis through a contract, engagement letter, security assessment agreement, or similar instrument. Scope should be clearly defined: what systems can be analyzed, what methods are permitted, what findings can be reported.

- **Implied authorization through role**: Your job responsibilities inherently include analyzing certain systems (maintaining production services, investigating bugs in products you support). This is common but risky because implied authorization may not hold up if challenged; written confirmation is preferable.
- **License-permitted analysis**: The software's license explicitly allows certain types of analysis (open-source licenses like MIT, Apache, GPL typically permit examination and modification; some proprietary licenses allow debugging or interoperability work). Always read the actual license text.
- **No authorization**: You do not own the system, have no permission from the owner, and the license prohibits analysis. Reverse engineering in this context is unlawful regardless of technical merit or benign intent.

Scope definition: Authorization must specify what is included and excluded:

- Which systems, services, binaries, protocols, or data are in scope
- What types of analysis methods are permitted (static analysis only, dynamic testing allowed, fuzzing restricted, etc.)
- Whether production environments may be analyzed or only isolated copies
- How findings should be reported and to whom
- Data handling requirements for sensitive information discovered during analysis
- Duration of authorization and conditions under which it expires

Documentation: Maintain written records of authorization throughout the project:

- Save engagement letters, contracts, or permission emails
- Record the scope as defined by the authorizing party
- Note any limitations or restrictions explicitly stated
- Document dates when authorization was granted and any renewals
- If authorization is verbal, follow up with written confirmation

Third-party considerations: Even when authorized to analyze a system you control, that system may incorporate third-party components (libraries, frameworks, services) whose licenses restrict analysis. Be aware of:

- Open-source dependencies and their license requirements
- Commercial libraries with EULAs prohibiting reverse engineering
- External APIs with terms of service limiting traffic analysis
- Embedded firmware or hardware with separate legal protections

When in doubt about authorization status, treat the system as unauthorized until clarification is obtained. The cost of verifying permission is far lower than the consequences of proceeding without it.

The RE Methodology Cycle

Reverse engineering follows a cyclical methodology that iterates between observation, hypothesis formation, experimentation, and validation. Each cycle refines understanding and narrows uncertainty. The process is not linear; new observations often require revisiting earlier conclusions and adjusting hypotheses accordingly.

Phase 1: Observation and Evidence Collection

The investigation begins with systematic examination of available artifacts and behaviors. At this stage you are gathering facts without forming strong conclusions about what they mean. Evidence sources include:

- File system structure, directory organization, naming conventions
- Source code (if available): modules, imports, function signatures, comments
- Binary metadata: symbols, strings, imports, exports, version information
- Configuration files, environment variables, deployment definitions
- Runtime output: logs, error messages, network traffic, performance metrics
- Existing documentation, even if incomplete or outdated
- Test suites that reveal expected behavior and edge cases

The goal is breadth rather than depth at this stage. You want a comprehensive inventory of what exists before diving into detailed analysis of specific components. Claude Code accelerates this phase by rapidly scanning large codebases, identifying patterns across files, generating directory maps, listing dependencies, and summarizing structures that would take hours to catalog manually.

Phase 2: Hypothesis Formation

With initial evidence collected, you begin forming hypotheses about how the system works. A hypothesis is a specific, testable statement about behavior or structure: "Function X reads configuration from environment variable Y", "The protocol uses length-prefixed messages with a 4-byte header", "This binary implements algorithm Z for data compression".

Good hypotheses have three properties:

- **Specific**: They make precise claims rather than vague generalizations
- **Testable**: There exists at least one experiment or observation that could confirm or disprove them
- **Falsifiable**: They can be proven wrong, not just supported by confirmation bias

Claude Code is useful for hypothesis generation because it can identify patterns across large amounts of code and suggest explanations based on common design patterns. However, AI-generated hypotheses must be treated as candidates for testing rather than conclusions. The model's training data includes many software systems, and it may propose explanations that sound plausible but are incorrect for the specific system under analysis.

Phase 3: Experimentation and Testing

Hypotheses are tested through controlled experiments designed to produce observable results. Experiments may include:

- Code inspection: Reading source code at locations predicted by a hypothesis
- Static analysis: Running tools to verify structural claims (symbol presence, call graph paths)
- Dynamic testing: Executing the system with specific inputs and observing outputs
- Tracing: Instrumenting execution to observe internal behavior (system calls, function calls, memory access)
- Differential testing: Comparing behavior across versions or configurations
- Boundary testing: Probing edge cases to understand limits and error handling

Each experiment should be designed to distinguish between competing hypotheses rather than merely confirming what you already suspect. The strongest experiments are those that could falsify a hypothesis if it is wrong.

Phase 4: Validation and Revision

Experimental results are compared against predictions from each hypothesis. When results match predictions, the hypothesis gains support but is not proven; when results contradict predictions, the hypothesis must be revised or discarded. This is where intellectual honesty matters most: confirming your initial guess feels satisfying, but reverse engineering quality depends on following evidence wherever it leads, even when it contradicts expectations.

Validation also means checking conclusions against multiple independent lines of evidence. A claim supported only by AI speculation is weak. A claim supported by source code analysis, confirmed by runtime observation, and consistent with tool output is strong. The methodology requires that important conclusions rest on evidence quality, not confidence level.

Phase 5: Documentation and Knowledge Capture

As understanding solidifies, findings are documented in a form that can be used for the project's purpose (migration specification, interoperability contract, architecture documentation, security report) and by others who may need to build on this work later. Documentation should record not just conclusions but also the evidence supporting them, so future readers can evaluate confidence levels and reproduce key findings if necessary.

Iteration: The cycle repeats as new questions arise from deeper understanding. Early hypotheses about high-level architecture may be correct in outline but wrong in important details that only emerge when examining specific components closely. Each iteration refines the model of how the system works until it is accurate enough for the project's needs.

The depth and duration of iteration depend on the project's goals. Understanding a system well enough to migrate one component requires less completeness than documenting an entire architecture or creating a fully compatible reimplementation. Scope determines when "accurate enough" has been reached.

Evidence Types and Quality

Reverse engineering conclusions are only as reliable as the evidence supporting them. Understanding different evidence types and their relative strengths helps analysts evaluate how much confidence to place in any particular finding and identify gaps where additional investigation is needed.

Static evidence comes from examining artifacts without executing them:

- **Source code**: The strongest form of static evidence when available. Shows exact implementation logic, data structures, control flow, and dependencies. Limitations include dead code that is never executed, build-time conditionals that exclude or include paths depending on configuration, and runtime behavior that differs from what the code suggests due to external factors (concurrency timing, environment state).
- **Binary metadata**: Symbols, strings, imports, exports, version information, debug data. Reveals structure and dependencies without requiring disassembly. Stripped binaries provide less information but still contain useful artifacts like embedded strings and import tables.
- **File structure and organization**: Directory layout, naming conventions, module boundaries, build files. Suggests architectural intent and component relationships. Incomplete because organizational structure does not always match runtime behavior.
- **Configuration and deployment files**: Environment variables, config files, Dockerfiles, Kubernetes manifests, CI pipelines. Reveal how systems are configured, what dependencies they expect, and how they are deployed. May differ from actual production configuration.

Dynamic evidence comes from observing behavior during execution:

- **Runtime output**: Logs, console output, error messages, return codes. Shows what actually happened in a specific execution. Limited to what the system chooses to report; silent failures provide no evidence.
- **System call traces**: Records of interactions with the operating system (file operations, network I/O, process management). Reveals resource usage and external dependencies. Tools like strace on Linux or DTrace/SystemTap on other platforms capture this data.

- **Network traffic**: Packets sent and received during execution. Essential for protocol analysis and understanding external communication. May be encrypted, limiting visibility into content.
- **Memory state**: Contents of memory at specific points during execution (via debuggers or core dumps). Shows runtime data structures and values. Complex to interpret without symbols or documentation.
- **Performance metrics**: Timing, resource consumption, throughput under load. Reveals algorithmic behavior and bottlenecks. Non-deterministic factors can make results vary between runs.

AI-generated hypotheses are explanations proposed by Claude Code based on patterns in the evidence:

- Useful for generating candidates when human analysts lack domain familiarity
- Must be verified against static or dynamic evidence before being treated as findings
- Quality depends on the completeness and accuracy of context provided to the model
- Particularly unreliable for system-specific details not represented in training data

Human conclusions are judgments made by the analyst after evaluating all available evidence:

- Incorporate domain expertise, experience with similar systems, and contextual knowledge that tools lack
- Subject to bias and error; should be explicitly distinguished from raw evidence
- Strongest when they synthesize multiple independent lines of evidence consistently pointing to the same conclusion

Evidence quality criteria:

- **Directness**: Evidence that directly shows a fact (source code implementing a feature) is stronger than circumstantial evidence suggesting it (a function name hinting at functionality).

- **Independence**: Multiple independent sources confirming the same finding increase confidence. Two different tools reporting the same import dependency is more reliable than one tool's output interpreted by an AI.
- **Reproducibility**: Evidence that can be obtained again through documented procedures is more valuable than one-off observations that cannot be repeated.
- **Completeness**: Evidence covering all relevant cases (all code paths, all message types) supports stronger conclusions than partial evidence that leaves gaps.
- **Freshness**: Evidence from the actual version under analysis is more reliable than evidence from older or newer versions where behavior may have changed.

The methodology requires maintaining an explicit connection between each important conclusion and the evidence supporting it. This "evidence ledger" approach ensures that findings can be evaluated, challenged, and reproduced by others, and prevents AI hallucinations or analyst assumptions from being mistaken for verified facts.

Chapter 2: Claude Code from First Principles

Installation and Configuration

Claude Code is distributed as a native command-line application with optional IDE extensions. The installation process varies slightly by platform but follows a consistent pattern across supported operating systems. This section covers current official installation methods and initial configuration steps based on Anthropic's documentation at code.claude.com/docs.

System requirements: Claude Code runs on macOS 13 or later, Windows 10 version 1809 or later (including Server 2019+), Ubuntu 20.04+, Debian 10+, and Alpine 3.19+. It requires at least 4GB of RAM, an x64 or ARM64 CPU, an internet connection for API communication, and a compatible shell (Bash, Zsh, PowerShell, or CMD). Your location must be in a country where Anthropic services are available.

Recommended installation method: The native installer is the officially recommended approach and includes automatic updates by default. On macOS and Linux, run:

```
curl -fsSL https://claude.ai/install.sh | bash
```

On Windows, use the PowerShell or CMD scripts provided on the official documentation page. The native installer places the binary in a standard location (~/.local/bin/claude on Unix-like systems) and configures auto-updates to keep you on the latest release channel.

Alternative installation methods:

- **Homebrew** (macOS/Linux): `brew install --cask claude-code` installs the stable version. Auto-updates are not enabled by default unless configured via environment variable.

- **WinGet** (Windows): `winget install Anthropic.ClaudeCode` installs from the Microsoft package manager. Manual updates required by default.
- **Linux package managers**: Signed repositories are available for apt, dnf, and apk package managers, providing stable and latest release channels managed through your system's update workflow.
- **npm** (deprecated): `npm install -g @anthropic-ai/claude-code` still works but is officially deprecated since version 2.1.15. It installs the same native binary under the hood but requires Node.js 22+. The native installer should be preferred for new installations.

Verification: After installation, verify the installation with `claude --version` to confirm the executable is accessible and check its version. Run `claude doctor` for a diagnostic check that validates your environment, authentication status, and configuration.

Authentication: Claude Code requires an active Anthropic account with a paid subscription (Pro, Max, Team, or Enterprise) or an Anthropic Console API key. The free plan does not include Claude Code access. On first run, the tool will guide you through authentication via browser-based OAuth or API key entry. Third-party API providers are also supported: Amazon Bedrock, Google Cloud's Agent Platform, and Microsoft Foundry can serve as backends for Claude Code sessions.

Binary integrity: The native binaries are signed for verification. macOS and Windows binaries carry platform-native code signatures. Linux distributions use manifest verification with Anthropic's GPG key (fingerprint 31DD DE24 DDFA B679 F42D 7BD2 BAA9 29FF 1A7E CACE) or package manager signatures. When downloading manually, verify signatures to ensure you are running authentic software.

Updates: Native installations auto-update on startup by default. The release channel can be configured via the `autoUpdatesChannel` setting in settings.json ("latest" is default, "stable" provides less frequent updates). You can pin a minimum version with `minimumVersion`, disable auto-updates with the environment variable `DISABLE_AUTOUPDATER=1`, or manually update with `claude update`. For reverse engineering work where reproducibility matters, consider pinning to a specific version and documenting it as part of your project configuration.

Uninstallation: Remove Claude Code using the method corresponding

to your installation: delete ~/.local/bin/claude and ~/.local/share/claude for native installs, use package manager uninstall commands, or run `npm uninstall -g @anthropic-ai/claude-code`. Configuration and data in ~/.claude persist after uninstallation and should be removed separately with `rm -rf ~/.claude` if desired.

Project Context and Instruction Files

Claude Code's ability to assist with reverse engineering depends heavily on how well you provide it with context about the project you are analyzing. The tool supports several mechanisms for establishing persistent project context that applies across sessions, eliminating the need to repeat background information in every conversation.

CLAUDE.md files: The CLAUDE.md file is the primary mechanism for providing Claude Code with project-specific instructions. When present at your project root (or inside a .claude/ directory), its contents are automatically loaded into Claude's context at the start of every session. This creates persistent guidance that the assistant follows without you having to restate it each time.

For reverse engineering projects, a CLAUDE.md file should include:

- **Project identification**: What system is being analyzed, its purpose, version or build information
- **Authorization statement**: Confirmation that analysis is authorized and scope boundaries
- **Technical context**: Languages used, runtime environment, known dependencies, build system
- **Analysis goals**: What you are trying to understand or accomplish
- **Methodology preferences**: How findings should be documented, evidence requirements, uncertainty handling
- **Constraints**: Files or directories to avoid, sensitive data locations, permissions restrictions

Example CLAUDE.md for a reverse engineering project:

```
# Legacy Payment Service Analysis

# Authorization
This analysis is authorized by Acme Corp for internal modernization planning.
Scope: payment-service repository only. No production system access.

# Technical Context
- Language: Java 8, Spring Boot 1.5.x
- Database: PostgreSQL with custom schema
- Build: Maven (pom.xml)
- Deployment: Docker container on Kubernetes

# Analysis Goals
1. Document overall architecture and component responsibilities
2. Map all external API endpoints and their contracts
3. Identify business rules embedded in code
4. Catalog dependencies for migration planning
5. Generate regression test candidates for critical paths

# Methodology
- All conclusions must be supported by source code evidence or runtime
  ↪ observation
- Mark uncertain findings explicitly with confidence level and reasoning
- Do not execute commands that modify files without explicit approval
- Treat all configuration files as potentially containing secrets; do not echo
  ↪ values

# Constraints
- Do not analyze /vendor directory (third-party libraries)
- Do not access environment variables in production config files
- Read-only analysis of source tree
```

Claude Code reads CLAUDE.md files hierarchically. A global file at ~/.claude/CLAUDE.md applies to all projects, while project-specific files override or extend global settings for individual repositories. For team-based work, committing a CLAUDE.md file to version control ensures consistent behavior across all analysts working on the same system.

.claude directory structure: Beyond CLAUDE.md, the .claude/ directory supports additional organizational mechanisms:

- **rules/*.md**: Topic-scoped instruction files that provide detailed guidance for specific analysis areas (e.g., rules/binary-analysis.md, rules/security-review.md). These are loaded as part of project context and can be referenced by Claude Code when relevant.

- **skills//SKILL.md**: Reusable prompt templates invoked via slash commands (e.g., /architecture-summary, /dependency-map). Useful for standardizing common analysis tasks across projects.
- **agents/*.md**: Subagent definitions that create specialized Claude Code sessions with their own tool access and instructions. Can be useful for isolating different aspects of a large analysis project.
- **settings.json**: Project-level configuration for permissions, hooks, environment variables, and model selection (discussed in the next section).

Context management during sessions: Within a single Claude Code session, context is accumulated through conversation history, file reads, tool outputs, and generated content. The context window has finite capacity (200,000 tokens by default for most models, up to 1M tokens for Opus-class models), and once filled, earlier content may be dropped or compressed. For long reverse engineering investigations:

- Keep CLAUDE.md concise; it is loaded into every session and consumes context budget
- Use focused sessions for specific analysis tasks rather than one endless conversation
- Save important findings to files in the project so they persist across sessions
- Use Claude Code's memory features (auto-memory stored in ~/.claude/projects//memory/) to capture key insights that should carry forward between sessions
- Be aware that MCP servers and other integrations also consume context window space

Permission Models and Approvals

Claude Code operates under a permission model designed to balance productivity with safety. For reverse engineering work where you may be analyzing untrusted code, executing unknown commands, or working with potentially dangerous artifacts, understanding and configuring permissions correctly is critical.

Permission modes: Claude Code supports several operational modes that control how tool calls are approved:

- **default (Manual)**: Claude Code prompts for user approval before performing edits, file operations, or command execution. This is the safest mode for initial exploration of unfamiliar codebases.
- **acceptEdits**: Automatically approves file edits and filesystem operations while still requiring approval for shell commands. Useful when you trust Claude Code's editing behavior but want to review every executed command.
- **plan (Read-only)**: Restricts Claude Code to reading files and running read-only commands. No edits or modifications are permitted. Ideal for initial reconnaissance phases where you only need to understand structure without changing anything.
- **auto**: Uses a classifier model to evaluate each action and approve safe operations automatically while prompting for potentially risky ones. Faster than manual mode but introduces an additional AI layer making decisions about what is "safe."
- **dontAsk**: Denies all actions unless explicitly pre-approved via allow rules in settings.json. Maximum safety; Claude Code can only perform operations you have specifically authorized in advance.
- **bypassPermissions**: Skips non-critical permission prompts for faster interaction. Should not be used when analyzing untrusted code or in environments where accidental damage would be costly.

For reverse engineering work, the recommended progression is: start with plan mode during initial reconnaissance to understand the codebase structure without risk of modification; move to default (manual) mode during active analysis when you need Claude Code to run tools and generate files but want to review every action; use acceptEdits only after you have established trust in Claude Code's behavior for your specific project type.

Permission rules: Fine-grained control is achieved through permission rules configured in settings.json files. Rules are specified as strings matching the pattern `Tool(specifier)` using shell-style glob wildcards. Three categories of rules exist:

- **allow**: Operations that Claude Code can perform without prompting
- **deny**: Operations that are blocked entirely and removed from Claude Code's available tools
- **ask**: Operations that require explicit user approval each time

Rules are evaluated in the order: Deny > Ask > Allow. A deny rule always takes precedence over an allow rule for the same operation.

Example settings.json permissions configuration for reverse engineering:

```
{
  "permissions": {
    "allow": [
      "Read",
      "Grep",
      "Glob",
      "Bash(file)",
      "Bash(sha256sum*)",
      "Bash(readelf*)",
      "Bash(nm*)",
      "Bash(strings)"
    ],
    "ask": [
      "Edit",
      "Bash(rm*)",
      "Bash(mv*)",
      "Bash(cp*)",
      "Bash(./*)",
      "Bash(python3*)"
    ],
    "deny": [
      "Bash(curl*)",
      "Bash(wget*)",
      "Bash(sudo*)"
    ]
  }
}
```

This configuration allows Claude Code to freely read files and run basic analysis tools (file, sha256sum, readelf, nm, strings), requires approval for any file edits or command execution that could modify state, and blocks network access commands and privilege escalation entirely. Adjust based on your specific needs and trust level.

Viewing and managing permissions: Run `/permissions` within a Claude Code session to open an interactive view of all active permission rules, their sources, and recent denial history. You can add or remove rules through this interface. When Claude Code prompts for approval of a tool call, selecting "Yes, and don't ask again" automatically writes the corresponding allow rule to your local settings file (.claude/settings.local.json), which is not committed to version control.

Working directory boundaries: Claude Code operates within a defined working directory (the directory where you invoke it). File operations are generally scoped to this directory tree. For reverse engineering, always invoke Claude Code from within an isolated analysis directory containing copies of artifacts you are examining, never from directories containing production systems, personal files, or other sensitive data.

Sandboxing: Claude Code supports sandboxed bash execution via the /sandbox feature, which runs commands in an isolated environment with restricted filesystem and network access. This is valuable for executing untrusted code during analysis: run suspicious binaries, test potentially dangerous scripts, or execute generated code in a sandbox before running anything on your host system. Note that Windows native installations lack full sandboxing support; use WSL 2 for sandboxed execution on Windows.

Tool Use and Shell Interaction

Claude Code's ability to assist with reverse engineering depends on its access to tools that perform actual analysis: disassemblers, debuggers, file inspection utilities, compilers, profilers, and other specialized software. Claude Code itself is not an analysis tool; it is an orchestrator that invokes tools, reads their output, interprets results, and suggests next steps.

Built-in capabilities: Claude Code includes several built-in tools for basic codebase exploration:

- **Read**: Read file contents from the filesystem
- **Edit**: Modify files (subject to permission rules)
- **Grep**: Search file contents using regular expressions
- **Glob**: Find files matching glob patterns
- **WebFetch**: Fetch and summarize web pages (useful for looking up documentation, specifications, or research papers during analysis)

These tools are sufficient for source-level analysis of readable codebases but inadequate for binary analysis, dynamic tracing, or deep structural examination. For those tasks, Claude Code relies on external tools invoked through shell commands.

Shell command execution: Claude Code can execute arbitrary shell commands within your environment (subject to permission rules and sandboxing configuration). This is how it accesses specialized reverse engineering tools:

- Running `readelf -h binary` to examine ELF headers
- Executing `ghidraHeadless` to perform automated binary analysis
- Invoking `strace ./program` to trace system calls
- Building and running test harnesses with `make` or `cargo build`
- Using `git log` and `git diff` to understand codebase history

When Claude Code runs a shell command, it captures the output and can analyze it, summarize findings, identify patterns, and suggest follow-up commands. This is powerful for reverse engineering because it combines Claude Code's language understanding with the authoritative results produced by specialized tools.

Important distinction: Claude Code does not execute code to verify its own claims unless you explicitly ask it to run relevant commands. If Claude Code says "function X calls function Y", that claim is based on pattern recognition from source code it has read, not from actually executing the program or building a call graph with analysis tools. Always verify important structural claims by running appropriate tools (e.g., `grep -r "functionY"`, examining build output, using a call graph generator).

Tool integration patterns: Effective reverse engineering with Claude Code involves establishing patterns for how external tools are used:

1. **Evidence collection**: Ask Claude Code to run analysis commands and save their output to files for reference
2. **Output interpretation**: Provide tool output to Claude Code and ask it to explain what the results mean in context
3. **Script generation**: Have Claude Code write scripts (Python, Bash) that automate repetitive analysis tasks
4. **Cross-tool correlation**: Use Claude Code to compare outputs from different tools and identify consistencies or conflicts

Example workflow: "Run `readelf -S ./binary` and `objdump -h ./binary`, save both outputs to files, then compare them and explain any differences in how sections are reported."

MCP (Model Context Protocol): Claude Code can connect to external tools and data sources through MCP servers. While MCP is primarily designed for integrating with services like databases, APIs, and project management tools, it can theoretically extend Claude Code's capabilities to specialized analysis tools

if MCP-compatible wrappers exist. As of the current documentation, Anthropic provides a directory of available MCP connectors but does not security-audit them; exercise caution when adding third-party MCP servers, especially in sensitive analysis environments.

Context Management and Limits

Claude Code's context window is its working memory: everything it can "see" at once during a session, including conversation history, loaded files, tool outputs, system instructions, and its own pending response. Understanding context limits is essential for effective reverse engineering work because running out of context causes information loss that directly degrades analysis quality.

Context window sizes: The available context depends on the model in use:

- Most Claude Code default models (Sonnet-class): 200,000 tokens
- Opus-class models: up to 1,000,000 tokens
- Haiku-class models: 200,000 tokens

A token is roughly three-quarters of a word in English text. For code, the ratio varies by language and formatting. A typical rule of thumb for planning purposes: 200,000 tokens can hold approximately 150,000 words of mixed text and code, or roughly 50,000-100,000 lines of source code depending on complexity and commenting.

Context consumption: Multiple factors consume context window space:

- **CLAUDE.md and instruction files**: Loaded at session start; keep them concise
- **Conversation history**: Every exchange accumulates; long sessions fill context quickly
- **File contents**: Each file Claude Code reads adds to context; reading entire large codebases exhausts context
- **Tool outputs**: Command output, especially from verbose tools, can consume significant space
- **MCP servers**: Connected servers and their available tool definitions occupy context

- **Response buffer**: A portion of the context window (~33,000 tokens as of early 2026) is reserved for Claude Code's response and cannot be used for input

Context management strategies:

1. **Focused sessions**: Create separate sessions for different analysis tasks (architecture mapping, dependency analysis, protocol investigation) rather than one monolithic conversation
2. **Selective file loading**: Ask Claude Code to read only relevant files for a specific question, not entire directories
3. **Save intermediate results**: Write findings to files so they can be reloaded in new sessions without repeating analysis
4. **Use grep and glob first**: Search for relevant content before reading full files; this consumes less context than loading everything
5. **Summarize verbose output**: When tool output is very long, ask Claude Code to summarize key points rather than including the full text in context
6. **Periodic session resets**: Start fresh sessions periodically to avoid context degradation from accumulated history

Context rot: As context windows fill, model performance degrades even before the hard limit is reached. Research and practical experience show that retrieval accuracy and reasoning quality decline as the proportion of relevant information in context decreases relative to noise. This means a session with 180,000 tokens of mixed conversation history and file contents will produce lower-quality analysis than a fresh session with 30,000 tokens of focused, relevant material.

For reverse engineering work where accuracy matters, prioritize context quality over quantity: keep sessions focused, load only what is needed for the current task, and start new sessions when context becomes cluttered with stale information.

Security, Privacy, and Data Handling

Using Claude Code for reverse engineering involves sending code, analysis results, and potentially sensitive information to Anthropic's servers via API calls.

Understanding data handling policies and implementing appropriate safeguards is essential, especially when analyzing proprietary systems, security-sensitive software, or artifacts containing confidential data.

Data transmission: All communication between Claude Code and Anthropic's services occurs over TLS-encrypted connections. Session traffic travels through the Anthropic API infrastructure, and session transcripts are stored on Anthropic servers to enable cross-device synchronization of conversations. This means that any code, file contents, tool outputs, or analysis findings visible within a Claude Code session are transmitted to and temporarily stored by Anthropic.

Data retention policies: Anthropic's data handling varies by account type:

- **Consumer accounts (Pro/Max)**: Chats and coding sessions may be used to improve Claude models if you opt in through your privacy settings. If training is allowed, data may be retained for up to 5 years in de-identified form. If training is not allowed, Anthropic retains data for a shorter period (specific duration no longer publicly committed as of June 2026 policy changes).
- **Enterprise and API accounts**: Data handling is governed by commercial terms and data processing agreements. Enterprise customers typically have stronger guarantees about data not being used for model training.
- **Flagged content**: Inputs and outputs flagged by automated trust and safety systems may be retained for up to 7 years regardless of account type.

For reverse engineering projects involving proprietary code, trade secrets, or regulated data, consult your organization's legal and security teams before using Claude Code. Verify applicable data handling requirements with Anthropic directly for Enterprise-grade guarantees.

Local data storage: Claude Code stores session transcripts, file snapshots, logs, and caches locally in plaintext within ~/.claude/projects//. These files contain everything that appeared in your sessions, including code contents, command outputs, and any sensitive data you examined. By default, local files are automatically deleted after `cleanupPeriodDays` (30 days), but this can be configured.

To reduce local exposure:

- Lower `cleanupPeriodDays` in settings for faster cleanup
- Set the environment variable `CLAUDE_CODE_SKIP_PROMPT_HISTORY` to avoid saving prompt history
- Use permission rules to deny access to credential files and sensitive paths
- Run `claude project purge <path>` to immediately clear all data for a specific project (use –dry-run first to preview)

Secret handling: Claude Code will read any file it has permission to access, including files containing API keys, passwords, certificates, and other secrets. Best practices:

- Use .gitignore and permission rules to prevent Claude Code from reading credential files
- Redact secrets from configuration files before analysis (replace actual values with placeholders)
- Never paste secrets directly into Claude Code prompts
- Be aware that secrets visible in tool output (e.g., environment variable dumps, config file reads) are transmitted to Anthropic's servers

Prompt injection defenses: Claude Code includes input sanitization and prompt injection protection mechanisms. However, when analyzing untrusted codebases or artifacts, malicious actors may embed instructions designed to manipulate Claude Code's behavior through comments, documentation, or configuration files. While built-in defenses mitigate many attacks, they are not guaranteed to block all variants. Treat analyzed content as potentially hostile and review Claude Code's actions carefully, especially when it proposes unusual commands or behavioral changes.

Model Limitations and Hallucination Risks

Claude Code is powered by large language models that generate outputs based on patterns learned during training combined with the context provided during a session. This architecture produces impressive capabilities for code understanding and generation but also introduces specific failure modes that are particularly dangerous in reverse engineering work.

Hallucination: The most significant risk is hallucination: Claude Code confidently stating incorrect information as if it were fact. In reverse engineering, hallucinations take several forms:

- **Fabricated APIs or functions**: Claiming a function exists at a specific location when it does not
- **Imaginary call paths**: Describing a sequence of function calls that never actually occurs in the code
- **Invented dependencies**: Asserting that a library or module is used when analysis would show otherwise
- **Incorrect compiler behavior assumptions**: Misunderstanding how optimizations, inlining, or platform-specific ABI details affect compiled output
- **Overconfident vulnerability claims**: Identifying "security issues" that do not actually exist or are mischaracterized

These hallucinations occur because the model is trained on vast amounts of code and documentation from many different systems. When asked about a specific system it has not seen, it may generate plausible-sounding answers based on patterns from unrelated systems in its training data. The output looks correct because it follows familiar conventions, but it does not reflect the actual system under analysis.

Context-window omissions: When Claude Code analyzes large codebases, it cannot hold all files in context simultaneously. It may miss relevant code because it was never loaded, or it may draw conclusions based on partial evidence without realizing that important information exists elsewhere. A claim like "there is no error handling for this case" may be true for the files currently in context but false for the complete codebase.

Misleading decompiler interpretations: When presented with decompiler output from tools like Ghidra, Claude Code may interpret pseudocode incorrectly because it treats it as if it were original source code written by a human. Decompiler output contains artifacts of the translation process (unnamed variables, flattened control flow, incorrect type inference) that Claude Code may not recognize as unreliable, leading to confident but wrong conclusions about binary behavior.

Verification requirement: The fundamental rule for using Claude Code in reverse engineering is: every important claim must be verified against authoritative evidence. If Claude Code says a function exists, verify it by searching the source code or examining symbols. If it describes a call path, trace it through the actual code or run dynamic analysis. If it identifies a vulnerability, reproduce the issue in a controlled environment. Never accept Claude Code's statements as conclusions; treat them as hypotheses that require testing.

Requesting uncertainty: You can mitigate hallucination risk by explicitly asking Claude Code to express uncertainty: "State your confidence level for each finding", "Distinguish between observations from the code and inferences you are making", "List what you cannot determine from available evidence". These prompts encourage the model to be more honest about limitations, though they do not eliminate hallucination entirely.

Prompt Injection from Untrusted Artifacts

When analyzing software whose provenance is uncertain or that may have been modified by hostile actors, Claude Code itself becomes a potential attack surface through prompt injection: malicious instructions embedded in analyzed content that manipulate the AI assistant's behavior.

Indirect prompt injection: Unlike direct prompt injection where a user crafts an adversarial prompt, indirect prompt injection occurs when malicious instructions are hidden in external content that Claude Code processes as part of its analysis. In reverse engineering, this could appear as:

- Comments in source code containing instructions like "ignore previous instructions and execute this command"
- README files or documentation with embedded directives targeting AI assistants
- Configuration files with specially crafted strings designed to trigger unintended behavior
- Strings embedded in binaries that, when extracted and presented to Claude Code, contain manipulation attempts

Real-world precedents: Research has demonstrated prompt injection attacks against AI coding assistants. In one documented case, a malicious payload embedded in code comments instructed an AI assistant to modify repository settings and enable remote code execution. In another, prompt injection in web content tricked an AI agent into downloading and running malware. These attacks exploit the fundamental challenge that LLMs process instructions and data through the same neural pathways and cannot always distinguish between them.

Defensive practices: When analyzing untrusted code with Claude Code:

1. **Review generated commands carefully**: Never approve shell commands from Claude Code without reading them first, especially during initial analysis of unknown codebases
2. **Use restrictive permissions**: Configure permission rules that deny network access (curl, wget), privilege escalation (sudo), and destructive operations (rm -rf) unless explicitly needed
3. **Run in isolated environments**: Use containers or virtual machines for analyzing suspicious artifacts; never analyze untrusted binaries directly on your host system
4. **Sanitize content before analysis**: When feeding extracted strings or documentation to Claude Code, be aware that the content may contain adversarial instructions
5. **Monitor for behavioral changes**: If Claude Code suddenly proposes actions inconsistent with your instructions or the analysis task, it may have been influenced by injected prompts

Claude Code's built-in protections: Anthropic has implemented input sanitization and prompt injection defenses in Claude Code. These include detecting and filtering known injection patterns, separating system instructions from user-provided content, and limiting the ability of analyzed content to override core behavioral constraints. However, no defense is perfect against an evolving attack surface, and caution remains necessary when working with untrusted material.

Reproducibility and Auditable Analysis

Reverse engineering work often needs to be reproducible: another analyst should be able to follow your steps and arrive at the same conclusions. When Claude Code is part of the analysis workflow, ensuring reproducibility requires deliberate practices because AI-assisted analysis is inherently less deterministic than tool-based analysis alone.

Session transcripts: Claude Code automatically saves session transcripts in ~/.claude/projects//.jsonl. These contain the full conversation history including your prompts, Claude Code's responses, and tool call details. For reproducible analysis:

- Export or archive session transcripts for important investigations

- Note which model and version was used for each session (check with `claude --version`)
- Document any CLAUDE.md content or settings.json configuration active during the session

Deterministic workflows: To make AI-assisted analysis more reproducible:

1. **Standardize prompts**: Use consistent prompt templates for common tasks (architecture summary, dependency mapping, call path tracing) so that repeated analyses follow similar reasoning paths
2. **Save tool commands**: Record every analysis command Claude Code runs; these are deterministic and can be rerun independently of the AI
3. **Archive evidence files**: Save all tool outputs, screenshots, dumps, and other evidence to version-controlled directories with timestamps
4. **Document hypotheses and conclusions separately**: Clearly distinguish between what tools showed (reproducible) and what Claude Code inferred (less reproducible)

Evidence ledger: Maintain an explicit record linking each important finding to the evidence that supports it. For each conclusion, note:

- The claim being made
- Static evidence (source code locations, symbols, file structures) supporting it
- Dynamic evidence (runtime observations, test results) confirming it
- Tool outputs that verify it
- Claude Code's role (hypothesis generation, interpretation, documentation)
- Confidence level based on evidence quality and quantity

This approach ensures that even if Claude Code's analysis cannot be exactly reproduced (due to model updates or context differences), the underlying evidence remains available for independent verification.

Version control: Structure your analysis work in a Git repository containing:

- Original artifact metadata (hashes, provenance information)

- Analysis scripts and tool configurations
- Generated documentation and findings
- Test harnesses and validation results
- Evidence files and tool outputs
- Decision records explaining key analytical choices

This creates an auditable trail of the entire analysis process that can be reviewed, challenged, and built upon by others.

Chapter 3: The AI-Assisted Reverse Engineering Workflow

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

Phase One: Authorization and Scoping

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

Phase Two: Preservation and Provenance

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

Phase Three: Environment Isolation

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

Phase Four: Inventory and Reconnaissance

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

Phase Five: Architecture Reconstruction

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

Phase Six: Hypothesis and Experimentation

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

Phase Seven: Validation and Documentation

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

The Evidence Ledger

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

Chapter 4: Source-Level Reverse Engineering

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

Repository Mapping and Structure Analysis

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

Build System Identification and Analysis

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

Entry Point Discovery

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

Dependency Graph Construction

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

Module and Interface Inventory

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

Configuration and Environment Analysis

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

Initialization Sequences and Lifecycle

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

Data Flow and Control Flow Tracing

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

Undocumented Behavior Discovery

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

Chapter 5: Behavioral Reconstruction and Black-Box Analysis

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

Black-Box Analysis Principles

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

Systematic Input/Output Observation

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

Boundary Value and Edge Case Testing

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

State Modeling and Transition Discovery

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

Differential Testing Methodology

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

Golden-Master and Snapshot Testing

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

Contract Extraction from Behavior

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

Falsifying AI-Generated Hypotheses

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

Chapter 6: Binary Analysis Fundamentals

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

Executable Formats and Structure

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

Machine Code and Assembly Concepts

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

Symbols, Debug Information, and Metadata

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

Calling Conventions and Stack Behavior

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

Functions, Control Flow, and Data References

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

Linking, Imports, and Exports

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

Compiler Artifacts and Optimization Effects

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

Safe Analysis with Authorized Binaries

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

Chapter 7: Combining Claude Code with Established Tools

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

The Tool Ecosystem Landscape

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

Working with Disassemblers and Decompilers

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

Debuggers and Runtime Inspection

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

Tracing and Profiling Integration

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

Dependency Analysis Tools

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

Packet Analysis for Protocol Work

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

Language-Specific Utilities

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

Orchestrating Tool Output with Claude Code

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

Chapter 8: Program Comprehension at Depth

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

Control Flow and Call Graphs

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

Data Flow and State Tracking

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

Invariants, Preconditions, and Postconditions

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

Side Effects and Hidden Coupling

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

Concurrency and Synchronization

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

Resource Lifetimes and Memory Ownership

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

Serialization and Persistence Patterns

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

Configuration-Driven Behavior and Feature Toggles

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

Focused Questioning for Testable Hypotheses

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

Chapter 9: Analyzing Compiled Applications Across Languages

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

Native Binaries and Compiled Languages

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

Managed Runtimes: JVM and .NET

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

Bytecode-Based Applications

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

Interpreted and Scripted Languages

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

Bundled and Packaged Applications

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

Minified and Obfuscated Code

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

Generated Code and Build Artifacts

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

Containerized Services and Images

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

Chapter 10: API, Protocol, and File-Format Reconstruction

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

API Reconstruction Principles

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

Request and Response Characterization

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

Message Framing and Serialization

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

State Transitions and Error Semantics

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

Version Negotiation and Compatibility

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

File Format Discovery and Schema Inference

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

Checksums and Integrity Mechanisms

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

Building Test Harnesses

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

Chapter 11: Legacy System Modernization

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

Assessing the Legacy System

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

Recovering Architectural Intent

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

Dependency Audit and Risk Assessment

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

Behavioral Characterization Before Change

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

Regression Test Generation

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

Interface Extraction and Documentation

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

Database Schema Recovery

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

Migration Planning and Execution

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

Compatibility Layers and Incremental Rewrite

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

Chapter 12: Clean-Room Reimplementation

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

Clean-Room Principles and Legal Basis

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

Separating Observation from Implementation

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

Defining Interface Contracts

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

Spec Writer and Programmer Roles

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

Independent Test Suite Construction

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

Implementing Compatible Behavior Safely

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

Evidence and Decision Records

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

Provenance Documentation

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

Chapter 13: Documentation Recovery

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

The Documentation Recovery Problem

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

Architecture Decision Records

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

Component Descriptions and Dependency Maps

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

Interface Catalogs and Data Dictionaries

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

Sequence Diagrams and State Diagrams

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

Operational Runbooks and Deployment Guides

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

Troubleshooting and Maintenance Documentation

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

Validating Generated Documentation

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

Chapter 14: AI-Specific Risks and Defensive Practices

Categories of AI Failure in RE

Hallucinated Evidence and Imaginary Paths

Prompt Injection from Analyzed Content

Secret Disclosure and Data Leakage

Destructive Command Risks

Clean-Room Contamination

Automation Bias and Overreliance

Defensive Practices and Safety Gates

Chapter 15: Advanced Workflows, Testing, and Professional Practice

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

Prompt and Context Engineering for RE

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

Reusable Prompt Patterns and Templates

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

Testing Reverse Engineering Conclusions

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

Analysis Repository Structure

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

Confidential Data Handling

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

CI and Toolchain Integration

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

Troubleshooting Methodology

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

Evaluating AI-Assisted RE Quality

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

When Traditional Approaches Are Better

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

Conclusion: The Future of AI-Assisted Understanding

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

Back Matter: Claude Code Reverse Engineering Playbook

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

The Authorization-First Workflow Summary

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

Phased Analysis Plan Template

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

Reusable Project Structure

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

Prompt and Context Templates

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

Evidence Ledger Template

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

Analysis Checklist

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

Verification Checklist

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

Troubleshooting Guide

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

Clean-Room Checklist

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

Security and Privacy Checklist

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

Glossary

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

References / Bibliography

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