Verified Intent Development

A Methodology for the Age of AI-Augmented Software Development

SEMCL.ONE Community

First Edition - December 2025

Contents

1 Preface **3**
1.1 Chapter 1: The Inversion . . . 3
1.2 Chapter 2: Why Existing Approaches Fall Short . . . 6
1.3 Chapter 3: The Core Insight . . . 8
1.4 Chapter 4: Principle One — Intent Before Generation . . . 10
1.5 Chapter 5: Principle Two — Graduated Trust . . . 11
1.6 Chapter 6: Principle Three — Understanding Over Acceptance . . . 12
1.7 Chapter 7: Principle Four — Provenance Awareness . . . 14
1.8 Chapter 8: Principle Five — Continuous Calibration . . . 15
1.9 Chapter 9: The Intent Specification Practice . . . 16
1.10 Chapter 10: The Verification Ritual Practice . . . 17
1.11 Chapter 11: The Learning Loop Practice . . . 18
1.12 Chapter 12: The Provenance Hygiene Practice . . . 19
1.13 Chapter 13: The Junior Engineer's Path to Verification Mastery . . . 20
1.14 Phase 1: Foundations (Weeks 1-3) . . . 21
1.15 Phase 2: Building Verification Skills (Weeks 4-7) . . . 25
1.16 Phase 3: Integration and Judgment (Weeks 8-11) . . . 30
1.17 Phase 4: Mastery and Teaching (Week 12+) . . . 35
1.18 Chapter 14: For Senior Engineers . . . 36
1.19 Chapter 15: For Teams and Organizations . . . 37
1.20 Chapter 16: Adopting VID . . . 38
1.21 Chapter 17: The Continuing Evolution . . . 38
1.22 Chapter 18: Summary . . . 39
1.23 Chapter 19: Patterns and Anti-Patterns . . . 40
1.24 Chapter 20: The Verification Toolkit . . . 47

2 Chapter 21: Test Verification Framework **56**
2.1 The Problem with AI-Generated Tests . . . 56
2.2 Core Principle: Tests Are Code . . . 56
2.3 Part 1: Test Provenance . . . 56
2.4 Part 2: Meta-Verification Strategies . . . 57
2.5 Part 3: AI in Test Verification . . . 61
2.6 Part 4: Test Quality Checklist . . . 61

2.7 Part 5: Common Test Anti-Patterns . . . 62
2.8 Part 6: Test Verification Workflow . . . 64
2.9 Part 7: Tools & Techniques Summary . . . 64
2.10 Part 8: Integration with VID Principles . . . 65
2.11 Conclusion . . . 65

3 Appendix A: Quick Reference 66
3.1 The Five Principles . . . 66
3.2 Verification Depth Quick Guide . . . 66
3.3 Intent Specification Template . . . 67

4 Appendix B: Discussion Questions 67

5 Appendix C: Glossary 67

6 Appendix D: Risk Scoring Rubric 68
6.1 Purpose . . . 68
6.2 The Four Risk Dimensions . . . 68
6.3 Risk Score Calculation . . . 70
6.4 Trust Level Mapping . . . 71
6.5 Verification Requirements by Trust Level . . . 71
6.6 Worked Examples . . . 72
6.7 Special Cases & Escalation . . . 73
6.8 Team Calibration Exercise . . . 73
6.9 Integration with Development Workflow . . . 74
6.10 Customization Guidelines . . . 75
6.11 Common Pitfalls . . . 75
6.12 Quick Reference Card . . . 76
6.13 Conclusion . . . 76

7 Appendix E: Decision Trees 76
7.1 Purpose . . . 76
7.2 Decision Tree 1: Should I Use AI for This Task? . . . 76
7.3 Decision Tree 2: What Trust Level Should I Apply? . . . 77
7.4 Decision Tree 3: How Do I Verify AI-Generated Tests? . . . 78
7.5 Decision Tree 4: Should I Accept This AI-Generated Code? . . . 80
7.6 Decision Tree 5: What Do I Do When Production Breaks? . . . 82
7.7 Decision Tree 6: How Do I Onboard a New Developer to VID? . . . 83
7.8 Decision Tree 7: When Should I Update Our VID Practices? . . . 85
7.9 How to Use These Decision Trees . . . 86
7.10 Quick Reference . . . 87

8 Appendix F: VID Checklists 87
8.1 Purpose . . . 87
8.2 Checklist 1: Pre-Generation (Intent Specification) . . . 87
8.3 Checklist 2: Post-Generation Verification (All Trust Levels) . . . 88
8.4 Checklist 3: Moderate Trust Verification . . . 88
8.5 Checklist 4: Guarded Trust Verification . . . 89
8.6 Checklist 5: Minimal Trust Verification . . . 89

8.7 Checklist 6: Test Verification . . . 89
8.8 Checklist 7: Code Acceptance . . . 90
8.9 Checklist 8: Incident Response . . . 90
8.10 Checklist 9: Team Calibration Session . . . 91
8.11 Checklist 10: Onboarding New Developer to VID . . . 91
8.12 Checklist 11: PR Review (for Reviewers) . . . 92
8.13 How to Use These Checklists . . . 92
8.14 Printable Quick Reference Cards . . . 93
8.15 Digital Checklist Templates . . . 94

1 Preface

This book introduces Verified Intent Development (VID), a methodology designed for software development in an era where artificial intelligence can generate code faster than humans can verify it.

If you learned to code before 2023, everything you know about development workflow is built on a fundamental assumption: **writing code is the hard part**. Methodologies like Agile, Scrum, and Kanban optimize for this assumption. They help teams coordinate the slow, cognitively demanding work of translating requirements into working software.

That assumption is now obsolete.

AI coding assistants can generate functional code in seconds. A junior developer with the right prompts can produce more lines of code in an afternoon than a senior developer could write in a week. The bottleneck has shifted. Writing code is no longer the constraint.

Verification is.

The question is no longer "How do we write code faster?" but rather "How do we know the code we generated is correct, secure, maintainable, and legally safe to use?"

VID addresses this new reality. It's not about restricting AI or slowing down development. It's about building the judgment, practices, and habits that allow you to move fast *with confidence*.

This book will teach you how to think about AI-augmented development, not just what steps to follow. The goal is to make you effective whether you're using today's tools or whatever comes next.

1.1 Chapter 1: The Inversion

1.1.1 The Old World

For fifty years, software development methodologies have solved the same fundamental problem: **code is expensive to produce**.

Writing code requires understanding the problem, designing a solution, translating that design into syntax, debugging errors, and iterating until it works. This process is slow, cognitively demanding,

and error-prone. A skilled developer might produce 100-200 lines of production-quality code per day.

Every methodology you've heard of optimizes around this constraint:

Waterfall tried to minimize wasted coding effort by doing extensive upfront planning. If writing code is expensive, don't write code until you're certain what to write.

Agile accepted that requirements change, so it optimized for shorter feedback loops. Write less code at a time, verify it works, adjust course. Don't invest too much before validating direction.

Scrum added coordination structures around Agile. If coding is the expensive part, make sure everyone is working on the right code and not duplicating effort.

Extreme Programming recognized that code quality problems compound, so it front-loaded quality practices. Pair programming, test-driven development, continuous integration — all designed to catch problems before code goes too far.

These methodologies disagree on many things, but they share one assumption: the primary constraint is producing code.

1.1.2 The Inversion

In 2023, that assumption began to collapse. AI coding assistants crossed a threshold from "occasionally useful autocomplete" to "genuine code generation." By 2025, the shift was undeniable.

Consider what changed:

- A developer can describe a function in plain English and receive working code in seconds
- Entire modules can be scaffolded from a description
- Tests can be generated automatically
- Documentation can be written by describing what to document
- Boilerplate that took hours now takes minutes

The raw act of producing code is no longer the bottleneck.

But something else happened that the optimists didn't predict: **the problems didn't go away, they moved**.

Code that generates in seconds still needs to: - Actually solve the intended problem (not just a similar one) - Handle edge cases correctly - Be secure against adversarial input - Perform acceptably under load - Integrate with existing systems without breaking them - Be maintainable by humans who didn't write it - Not infringe on someone else's intellectual property - Comply with applicable regulations

AI can write code fast. AI cannot *guarantee* any of these things.

The bottleneck inverted. We went from:

```
[Hard] Writing Code → [Easy] Verifying Code
```

To:

```
[Easy] Writing Code → [Hard] Verifying Code
```

Every methodology designed for the old world optimizes for the wrong constraint.

1.1.3 The Danger Zone

This inversion creates a dangerous period. Teams can generate code faster than ever, but their verification capacity hasn't increased. The result is predictable:

Technical debt accumulates faster. Code is generated without full understanding, creating maintenance burdens that compound.

Security vulnerabilities multiply. AI generates code that works but doesn't anticipate adversarial conditions. Studies show AI-generated code has higher vulnerability rates than human-written code in security-sensitive contexts.

Quality becomes invisible. When code appears instantly, it *feels* free. Teams skip verification because the generation was so easy. They discover the costs later, in production.

Understanding erodes. Developers accept code they don't fully understand. When problems arise, they lack the context to debug effectively.

This isn't an argument against AI-assisted development. The productivity gains are real and substantial. This is an argument for **methodology that fits the new reality**.

1.1.4 What VID Is (and Isn't)

Before explaining what VID offers, it's important to understand what it is—and what it is not.

VID is NOT:

A replacement for Agile, Scrum, or other methodologies — VID is complementary. You can practice VID within Agile sprints, Scrum ceremonies, or any development process. VID addresses a specific gap: how to handle AI-generated code safely.

An AI-first development framework — VID isn't about maximizing AI usage or treating AI as the primary developer. It's about verification regardless of code source. (That said, AI-generated code often needs more verification.)

A prompt engineering guide — VID doesn't teach you how to write better prompts or get more from AI. It teaches you what to do *after* AI generates code. Prompt engineering is orthogonal to VID.

A productivity framework — VID's goal isn't shipping faster; it's shipping *safely* while using AI. Sometimes VID makes you slower (verification takes time). That's acceptable—the alternative is shipping bugs.

Test-Driven Development (TDD) — While VID encourages test-first thinking, TDD focuses on using tests to drive design. VID focuses on verification across multiple dimensions: functional correctness, security, maintainability, and provenance—not just passing tests.

A development environment or tool — VID is methodology, not software. You don't install VID. You practice it. (Though tools can help—see Appendix D for integration examples.)

VID IS:

A verification-first methodology for AI-augmented development — When code generation is cheap, verification becomes the core skill. VID systematizes verification.

A framework for calibrating trust — Not all code deserves equal scrutiny. VID provides principles for matching verification depth to risk level.

A mental model for the inverted bottleneck — VID helps developers think clearly about what changed, why it matters, and how to adapt.

Complementary to existing processes — VID fits into your current workflow. It answers the question "How should we handle AI-generated code?" that other methodologies don't address.

Applicable regardless of AI usage — Even if you write all code by hand, VID's principles (intent specification, graduated trust, understanding over acceptance) improve quality. But VID becomes essential when AI generates code.

The Core Distinction:

Most methodologies optimize for *code production*. VID optimizes for *code verification*. In the AI era, this is the critical shift.

1.1.5 What VID Offers

Verified Intent Development is designed for this inverted world. Its core insight is simple:

When generation is cheap and fast, verification becomes the valuable skill.

VID doesn't slow down generation. It builds systematic verification into the development process so that speed doesn't come at the cost of quality, security, or maintainability.

The developer's role transforms. Instead of spending 80% of time writing code and 20% reviewing, VID practitioners spend 20% specifying intent and 80% verifying output. This isn't a step backward — it's a recognition of where human judgment adds value.

VID provides:

1. **A mental model** for thinking about AI-augmented development
2. **Principles** that guide decision-making in novel situations
3. **Practices** that build verification into daily work
4. **A vocabulary** for discussing AI-related development decisions
5. **A framework** for adapting as AI capabilities evolve

The rest of this book teaches all five.

1.2 Chapter 2: Why Existing Approaches Fall Short

Before presenting VID, we should understand why you can't just apply existing methodologies to AI-augmented development. The failures aren't superficial — they're structural.

1.2.1 Agile's Blind Spot

Agile's core insight is that requirements change, so we should plan in short iterations and adapt based on feedback. This remains valid. But Agile assumes the expensive operation is writing code, so its ceremonies optimize for coordination and prioritization of coding effort.

Standups ask: "What did you code yesterday? What will you code today?"

Sprint planning asks: "How much can we code this sprint?"

Retrospectives ask: "How can we code more effectively?"

Notice what's missing: systematic attention to verification.

Agile has testing, of course. But testing in Agile is typically focused on "does the feature work?" not "is this code trustworthy?" When code is written by humans who understand what they wrote, this is often sufficient. The developer's understanding provides implicit verification.

When code is generated by AI, that implicit verification disappears. The developer may not fully understand the generated code. "It passes tests" is not the same as "I understand why it's correct."

Agile provides no framework for: - Deciding how much verification a piece of code needs - Tracking where code came from (human vs. AI) - Adjusting process based on code provenance - Building verification skills as a core competency

1.2.2 The "Just Add AI" Fallacy

Many teams try to integrate AI by simply adding it to their existing workflow:

"We do Scrum, but now developers use Copilot."

This approach fails because it treats AI as a faster typewriter rather than a fundamental shift in how code comes into existence. The methodology remains optimized for the old constraint while the actual constraint has changed.

Symptoms of this failure: - Developers generate more code but defect rates increase - Sprint velocity appears higher but production incidents rise - Technical debt accumulates faster than before - Code reviews become rubber stamps because there's too much to review carefully - No one knows which parts of the codebase were AI-generated

1.2.3 The Vibe Coding Trap

On the opposite extreme, some developers abandon methodology entirely. "Vibe coding" — generating code through natural language prompts with minimal structure — produces impressive demos but dangerous production systems.

Vibe coding fails because:

No verification criteria. Without explicit criteria for correctness, there's no way to know if generated code is right. "It seems to work" is not verification.

No risk awareness. All code is treated equally, whether it's a utility function or an authentication handler.

No learning loop. Without systematic tracking, teams can't learn which patterns produce good outcomes.

No accountability. When everyone is prompting AI, no one is responsible for understanding the result.

Vibe coding works for prototypes and experiments. It's actively dangerous for production systems.

1.2.4 Emerging AIDD Frameworks

Several "AI-Driven Development" frameworks have emerged attempting to address these gaps. They typically share these characteristics:

- Replace sprints with shorter cycles ("bolts")
- Emphasize specification as input to AI
- Add AI agents for various development tasks
- Provide prompt templates and workflows

These frameworks improve on naive AI adoption, but most share a fundamental flaw: **they optimize for generation velocity, not verification confidence**.

Questions they don't adequately answer: - How do you know AI-generated code is correct? - When should humans verify vs. trust automation? - How do you build verification skills in developers? - How do you adapt verification to risk level? - How do you maintain understanding as AI writes more code?

VID addresses these questions directly.

1.3 Chapter 3: The Core Insight

1.3.1 Verification as the Skill

VID is built on a single core insight:

In AI-augmented development, verification is the core professional skill.

This might seem obvious, but its implications are profound. Consider what it means for:

Education: We should teach developers to verify code, not just write it. Reading code critically becomes more important than writing code fluently.

Hiring: Verification ability matters more than generation speed. A developer who can reliably identify when AI-generated code is wrong is more valuable than one who can prompt faster.

Process: Development process should optimize for verification throughput, not generation throughput.

Tools: We need tools that help verify, not just tools that help generate.

Career development: Senior developers are distinguished by judgment about what to verify and how deeply, not by typing speed or language knowledge.

1.3.2 The Verification Spectrum

Not all verification is equal. VID recognizes a spectrum from shallow to deep:

Syntactic verification: Does it compile/parse? Does it pass the linter? This is fully automatable and should always happen.

Functional verification: Does it produce correct output for known inputs? Automated tests cover this, but test quality matters enormously.

Semantic verification: Does it do what was *intended*, not just what was *specified*? This requires human judgment to bridge the gap between specification and intent.

Robustness verification: Does it handle edge cases, malformed input, resource constraints, and concurrent access correctly? Requires thinking about what could go wrong.

Security verification: Does it resist adversarial input and protect sensitive data? Requires threat modeling and security expertise.

Maintainability verification: Can future developers understand and modify this code? Requires projecting into the future.

Provenance verification: Where did this code come from? Does it infringe on others' rights? Requires understanding AI training and licensing.

AI-generated code needs all of these. Vibe coding provides none. Traditional Agile provides the first two, partially. VID provides a framework for all of them, calibrated to risk.

1.3.3 The Understanding Requirement

There's a deeper issue than verification mechanics: **understanding**.

When a human writes code, they understand what it does (at least at the moment of writing). This understanding enables: - Effective debugging when problems arise - Confident modification when requirements change - Accurate estimation of impact when changes are proposed - Reliable integration with other code

When AI generates code, this understanding doesn't automatically transfer to the human. The developer might accept code they don't fully understand. This creates a time bomb:

- Today: Code works, developer is happy
- Next month: Bug appears, developer struggles to debug code they don't understand
- Next quarter: Modification needed, developer afraid to change code they don't understand
- Next year: Code is legacy that no one will touch

VID addresses this through a principle we'll explore in detail: **No code without understanding.** This doesn't mean understanding every character — it means understanding at a level appropriate to the code's importance.

1.4 Chapter 4: Principle One — Intent Before Generation

1.4.1 The Principle

Never generate code without first articulating what you intend to build and how you will verify correctness.

This principle seems simple but violates common practice. Most developers using AI start with a vague idea, prompt the AI, look at the result, and decide if it seems right. This is backwards.

1.4.2 Why Intent First?

Without explicit intent, you cannot verify. You can only react. You look at generated code and ask "does this seem okay?" rather than "does this match what I needed?"

The difference is profound:

Reactive verification: "This function seems to do something with dates. It looks reasonable. I'll accept it."

Intent-driven verification: "I needed a function that calculates business days between two dates, excluding weekends and holidays from a provided list. Let me check if this implementation handles holidays, excludes weekends, and correctly counts the days without double-counting boundaries."

Reactive verification catches obvious errors. Intent-driven verification catches subtle ones — the ones that matter in production.

1.4.3 Articulating Intent

Intent should include:

Functional intent: What should this code do? What are the inputs, outputs, and transformation?

Quality intent: What are the performance requirements? Reliability requirements? What trade-offs are acceptable?

Boundary intent: What inputs are valid? What should happen with invalid inputs? What are the edge cases?

Integration intent: How does this interact with existing code? What contracts must it honor?

You don't need a formal document for every function. But you should be able to articulate these before generating code. If you can't, you're not ready to generate.

1.4.4 The Verification Criteria

Intent leads to verification criteria. Before generating, ask:

"How will I verify this is correct?"

If you can't answer, you can't verify. And if you can't verify, you shouldn't generate.

Verification criteria might be: - Specific test cases with expected outputs - Properties that must hold (e.g., "output is always sorted") - Security requirements (e.g., "rejects SQL injection attempts") - Performance requirements (e.g., "responds in under 100ms for 1000 items")

Write these down, even informally. They guide your verification after generation.

1.4.5 Practice: The Intent Moment

Before every generation, pause for an "intent moment." Ask yourself:

1. What exactly should this code do?
2. What are the edge cases?
3. How will I verify it's correct?

This pause takes seconds. It saves hours of debugging wrong code.

1.5 Chapter 5: Principle Two — Graduated Trust

1.5.1 The Principle

The level of verification should match the level of risk. Not all code deserves equal scrutiny.

This principle provides efficiency. Verifying everything deeply is impractical. Verifying nothing is dangerous. VID graduates verification based on risk.

1.5.2 Understanding Risk

Risk in software has multiple dimensions:

Impact if wrong: What happens if this code has a bug? - A typo in a log message: minimal impact - A bug in payment processing: financial loss, customer trust damage - A security flaw in authentication: potential breach

Reversibility: How easily can we recover from problems? - A bug in a stateless API: fix and redeploy - Data corruption: may require complex recovery - Deleted customer data: potentially unrecoverable

Exposure: How widely is this code used? - Internal tool for one team: limited exposure - Core library used everywhere: bugs affect everything - Public API: bugs affect external parties

Regulatory sensitivity: What compliance requirements apply? - Internal tooling: minimal regulatory concern - Healthcare data: HIPAA requirements - Financial transactions: PCI-DSS requirements

1.5.3 The Trust Spectrum

VID defines a spectrum of trust levels:

High Trust (minimal verification): AI output accepted with automated checks only. Appropriate for low-risk, easily reversible code.

Moderate Trust (standard verification): AI output reviewed for functional correctness. Appropriate for typical production code.

Guarded Trust (thorough verification): AI output reviewed in depth for correctness, security, and maintainability. Appropriate for important code.

Minimal Trust (intensive verification): AI used only for suggestions. Human writes or extensively rewrites all code. Multiple reviewers. Appropriate for high-risk code.

1.5.4 Calibrating Trust

For any code generation, ask:

1. What's the worst case if this code is wrong?
2. How easily can we recover from problems?
3. How widely is this code exposed?
4. What compliance requirements apply?

The highest-risk answer determines your trust level.

Example calibrations:

Utility function for formatting dates: - Worst case: Dates display incorrectly - Recovery: Easy redeploy - Exposure: Internal dashboard - Compliance: None - **Calibration: High Trust**

Input validation for a public API: - Worst case: Security vulnerability - Recovery: Depends on exploitation - Exposure: Internet-facing - Compliance: Potential - **Calibration: Minimal Trust**

1.5.5 The Trap of Uniform Treatment

Teams often fall into treating all code the same:

Uniform high scrutiny: Every function gets intensive review. This is unsustainable. Reviews become cursory because there's too much to review carefully.

Uniform low scrutiny: Everything gets quick acceptance. Risk accumulates invisibly until it manifests as incidents.

VID's graduated approach focuses human attention where it matters. It's not about less verification overall — it's about appropriate verification everywhere.

1.6 Chapter 6: Principle Three — Understanding Over Acceptance

1.6.1 The Principle

Never accept code you don't understand at an appropriate level of depth.

This is perhaps VID's most challenging principle. It requires discipline when AI makes acceptance frictionless.

1.6.2 The Appropriate Level

"Understanding" doesn't mean comprehending every character. It means understanding at a level appropriate to:

Your role: A senior engineer reviewing should understand more deeply than a junior engineer using a well-tested utility.

The code's risk: High-risk code requires deeper understanding. Low-risk code can be understood more superficially.

The code's stability: Code that will be modified requires deeper understanding than code that will be called but not changed.

1.6.3 Levels of Understanding

Surface understanding: "This function takes X and returns Y. I can use it correctly."

Appropriate for: Low-risk library functions, well-tested utilities, code you call but won't modify.

Functional understanding: "I understand the algorithm and why it produces correct results for various inputs."

Appropriate for: Most production code, code you might need to debug or modify.

Deep understanding: "I understand the implementation details, edge cases, performance characteristics, and security implications."

Appropriate for: High-risk code, security-sensitive code, code you'll maintain long-term.

1.6.4 Building Understanding

When AI generates code, actively build understanding:

1. **Read the code.** Not skimming — actually reading. Trace the logic.
2. **Question the choices.** Why this approach? What are the alternatives? What are the trade-offs?
3. **Test your understanding.** Can you explain what this code does without looking at it? Can you predict what happens with edge cases?
4. **Identify gaps.** What parts don't you understand? Are those acceptable gaps or do they need resolution?

1.6.5 The Understanding Debt

Accepting code you don't understand creates "understanding debt" — code that works but no one understands. This debt compounds:

- Debugging becomes harder
- Modifications become scary
- The codebase becomes a minefield of mystery code

Unlike technical debt, understanding debt can't be measured by tools. It only reveals itself when someone needs to modify code and discovers they can't.

VID prevents understanding debt through discipline: if you don't understand it at an appropriate level, you're not done.

1.7 Chapter 7: Principle Four — Provenance Awareness

1.7.1 The Principle

Always know where code came from and what that origin implies.

Code has provenance — a history of where it came from. In traditional development, provenance is straightforward: a human wrote it. In AI-augmented development, provenance becomes complex and important.

1.7.2 Why Provenance Matters

Debugging: When code fails, knowing its origin helps debug. AI-generated code might fail in ways specific to how AI generates code.

Modification: Understanding provenance helps assess modification risk. Code that no one understood when it was generated is harder to modify safely.

Legal protection: AI-generated code might inadvertently derive from copyrighted sources. Understanding provenance helps assess and manage this risk.

Quality assessment: Different origins have different quality characteristics. Knowing the origin helps calibrate verification.

1.7.3 Provenance Categories

VID recognizes several provenance categories:

Human original: Code written entirely by a human developer. Lowest risk — the developer understood what they wrote.

AI generated, human verified: Code generated by AI and carefully reviewed by a human who understood it. Risk depends on verification depth.

AI generated, lightly reviewed: Code generated by AI with superficial review. Higher risk — the human might not have caught subtle issues.

AI generated, unreviewed: Code accepted without meaningful review. Highest risk.

Mixed provenance: Code that combines human-written and AI-generated portions. Risk depends on how well the integration was verified.

1.7.4 Tracking Provenance

Provenance should be trackable:

- Can you identify which code in your codebase was AI-generated?
- Can you identify who reviewed it and how deeply?
- Can you identify what prompt generated it?

This tracking doesn't need to be elaborate. At minimum: - Commit messages should indicate AI-assisted generation - Code review records should indicate review depth - Teams should maintain awareness of which areas are heavily AI-generated

1.7.5 Using Provenance

Provenance informs decisions:

Incident investigation: "This function is failing. It was AI-generated and received minimal review. The original developer didn't understand the edge cases."

Modification planning: "This module is mostly AI-generated with light review. We should budget extra time for understanding before modification."

Risk assessment: "This area of the codebase has extensive AI generation with minimal documentation. It's higher risk than areas with clear provenance."

1.8 Chapter 8: Principle Five — Continuous Calibration

1.8.1 The Principle

Regularly assess whether your verification practices match actual risk and adjust accordingly.

This principle prevents VID from becoming rigid. No methodology gets calibration right initially. Continuous calibration adapts to reality.

1.8.2 The Feedback Loop

VID requires feedback from outcomes:

Positive feedback: Code that passed verification performs well in production. This validates the verification approach.

Negative feedback: Code that passed verification fails in production. This indicates verification was insufficient.

Both are valuable. The goal is verification that catches problems without excessive overhead — this requires empirical calibration.

1.8.3 Calibration Questions

Regularly ask:

Are we catching problems? - How often does verification reject code? - What kinds of problems are caught? - If verification rarely rejects anything, it might be too lenient.

Are we missing problems? - What problems escape to production? - Were they the kind verification should catch? - If problems consistently escape, verification needs strengthening.

Is verification effort appropriate? - Are we spending appropriate time on high-risk code? - Are we spending excessive time on low-risk code? - Adjust risk calibration if effort is misallocated.

Are our risk assessments accurate? - When problems occur, were they in code we classified as high-risk? - If high-risk problems come from "low-risk" code, our classification is wrong.

1.8.4 Calibration Triggers

Recalibrate when:

- Production incidents occur
- New types of problems appear
- AI capabilities change (new models, new tools)
- Team composition changes
- Regulatory requirements change
- Project characteristics change

1.8.5 The Anti-Pattern: Frozen Methodology

Some teams adopt VID (or any methodology) and never adjust. They follow practices that made sense initially but no longer fit.

Warning signs: - "We've always done it this way" - Verification levels that haven't changed in months - No discussion of whether practices are working - No adaptation to new AI capabilities

VID is a living methodology. It should evolve with your team and the technology.

1.9 Chapter 9: The Intent Specification Practice

1.9.1 From Principle to Practice

Principle One (Intent Before Generation) requires practical implementation. The intent specification practice provides it.

1.9.2 The Practice

Before generating code:

1. **Articulate the functional requirement.** What should this code do? Be specific about inputs, outputs, and transformation.
2. **Identify boundaries.** What inputs are valid? What should happen at boundaries? What should happen with invalid inputs?
3. **State success criteria.** How will you know this code is correct? What tests or checks will verify it?
4. **Assess risk.** What's the impact if this is wrong? This determines verification depth.

1.9.3 Intent Specification Formats

Intent specification can range from informal to formal:

Mental intent (lowest formality): For trivial code, a clear thought is sufficient: "I need a function that capitalizes the first letter of each word."

Comment intent: Write a comment describing the intent before generating:

```
// Function: Capitalize first letter of each word
// Input: String of words separated by spaces
// Output: Same string with each word capitalized
// Edge cases: Empty string returns empty, handles multiple spaces
```

Test-first intent: Write the tests before generating the implementation:

```python
def test_capitalizes_words():
    assert capitalize_words("hello world") == "Hello World"

def test_handles_empty():
    assert capitalize_words("") == ""

def test_handles_single_word():
    assert capitalize_words("hello") == "Hello"
```

Formal specification: For complex or high-risk code, use structured specification documents.

1.9.4 Matching Formality to Risk

- Trivial code: Mental intent is sufficient
- Typical code: Comment intent or test-first intent
- Important code: Test-first intent with edge cases
- Critical code: Formal specification with comprehensive test cases

1.9.5 Anti-Pattern: Retroactive Intent

Writing intent after generation defeats the purpose. If you generate first, you'll rationalize that the output matches your intent rather than critically evaluating it.

The discipline is: intent first, always.

1.10 Chapter 10: The Verification Ritual Practice

1.10.1 The Practice

After every code generation, perform a verification ritual. The depth varies by risk, but the practice is consistent.

1.10.2 The Basic Ritual (High Trust)

For low-risk code:

1. **Read the code.** Actually read it, don't skim.
2. **Verify against intent.** Does this do what you specified?
3. **Run automated checks.** Linting, type checking, tests.
4. **Ask: "What could go wrong?"** Spend 30 seconds considering failure modes.

This takes 2-5 minutes. It's the minimum for any code.

1.10.3 The Standard Ritual (Moderate Trust)

For typical production code:

1. **Complete the basic ritual.**
2. **Trace the logic.** Walk through the code path mentally.
3. **Test edge cases.** Run specific tests for boundary conditions.
4. **Check integration.** How does this interact with existing code?
5. **Document understanding.** Write a brief note on what this code does.

This takes 5-15 minutes depending on code complexity.

1.10.4 The Deep Ritual (Guarded Trust)

For important code:

1. **Complete the standard ritual.**
2. **Adversarial thinking.** Try to break the code. What inputs could cause problems?
3. **Security review.** Are there injection risks? Access control issues? Data exposure?
4. **Performance consideration.** What happens at scale? Are there expensive operations?
5. **Maintainability review.** Can someone else understand this? Should it be refactored for clarity?

This takes 15-30 minutes.

1.10.5 The Intensive Ritual (Minimal Trust)

For high-risk code:

1. **Complete the deep ritual.**
2. **Independent review.** Have another person review independently.
3. **Formal verification.** Apply relevant security tools, static analysis, etc.
4. **Documentation.** Document the code's behavior, limitations, and security considerations.
5. **Provenance documentation.** Record how this code was generated and verified.

This takes 30+ minutes and involves multiple people.

1.10.6 Making It a Ritual

The value of a ritual is consistency. It shouldn't require decision-making about whether to do it. After every generation:

"What's the risk level? Apply that verification ritual."

Make it automatic.

1.11 Chapter 11: The Learning Loop Practice

1.11.1 The Practice

Track what happens to your verified code. Use outcomes to improve verification.

1.11.2 The Tracking

Record:

- **What was generated and when**
- **What risk level was assigned**
- **What verification was performed**
- **What happened in production** (if problems occurred)

This doesn't require elaborate tooling. A simple log or notes suffice.

1.11.3 The Review

Periodically (weekly or monthly) review:

- **Problems that escaped verification:** What problems made it to production? Should verification have caught them? Why didn't it?
- **Verification that seems excessive:** Are you spending significant time verifying code that never has problems? Can that verification be reduced?
- **Risk calibration accuracy:** Do your "high risk" and "low risk" assessments match reality?

1.11.4 The Adjustment

Based on review, adjust:

- **Verification depth:** Increase for areas with escaping problems, decrease for areas with no problems
- **Risk calibration:** Update risk assessment criteria based on actual outcomes
- **Practices:** Add or remove verification steps based on what's working

1.11.5 The Anti-Pattern: No Feedback

Teams that don't track outcomes can't calibrate. They might: - Over-verify some areas while under-verifying others - Miss entire categories of problems - Stick with practices that don't work

The learning loop prevents methodology ossification.

1.12 Chapter 12: The Provenance Hygiene Practice

1.12.1 The Practice

Maintain awareness and documentation of where code comes from.

1.12.2 Commit Hygiene

When committing AI-generated or AI-assisted code:

Indicate AI involvement. Your commit message or description should indicate that AI assisted in generating this code.

Note the verification level. Briefly indicate how this code was verified.

Document significant AI interactions. For complex generations, keep the prompt/response history.

1.12.3 Code Comments

For non-trivial AI-generated code:

Document the intent. What was this code meant to do?

Note limitations. What does the AI-generated version not handle that a more complete implementation might?

Mark uncertainty. If you're uncertain about parts of the code, note that for future maintainers.

1.12.4 Team Awareness

Teams should have shared awareness:

Which areas are heavily AI-generated? This affects modification planning and risk assessment.

What AI tools were used? Different tools have different characteristics.

What verification standards were applied? This affects confidence in different areas of the codebase.

1.12.5 The Anti-Pattern: Invisible AI

When AI-generated code is indistinguishable from human-written code:

- Future maintainers don't know to be cautious
- Problems are harder to diagnose
- Risk assessment is wrong because provenance is unknown

Make provenance visible.

1.13 Chapter 13: The Junior Engineer's Path to Verification Mastery

1.13.1 The Paradox You Face

You're learning to code at a strange moment in history. AI can generate code faster than you can type it. This creates a paradox:

The shortcut that helps you today will cripple you tomorrow.

If you let AI generate code you don't understand, you'll ship features quickly. You'll look productive. But you won't be learning. And when things break—they always break—you won't be able to fix them. You'll become dependent on AI without developing the judgment to know when AI is wrong.

The engineers who will thrive learned verification *first*. They used AI as a learning accelerator, not a learning bypass.

This chapter provides a path to becoming that engineer.

1.13.2 The 12-Week Curriculum

This curriculum assumes you have basic programming knowledge. You can write simple programs. You understand variables, functions, loops, and conditionals. You may have used AI coding assistants casually.

The goal is to transform you from someone who *uses* AI to generate code into someone who can *verify* AI-generated code reliably—and in the process, become a much stronger engineer than you would have been otherwise.

1.14 Phase 1: Foundations (Weeks 1-3)

1.14.1 Week 1: Learning to Read Code Critically

The Goal: Develop the habit of reading code carefully before accepting it.

The Problem You're Solving: Most developers skim code. They look for obvious errors and move on. This works when you wrote the code yourself (you already understand it). It fails catastrophically with AI-generated code.

Daily Practice (30-45 minutes):

Take a small piece of AI-generated code (one function, 10-30 lines) and analyze it completely.

1. **Read it aloud.** Literally speak each line. This forces you to actually read rather than skim.
2. **Trace execution.** Pick a specific input. Walk through line by line. What value does each variable have at each step? Write it down.
3. **Identify assumptions.** What does this code assume about its inputs? What happens if those assumptions are violated?
4. **Find the edges.** What inputs would cause this code to behave unexpectedly? Empty inputs? Very large inputs? Negative numbers? Null values?
5. **Explain it.** Write a 2-3 sentence explanation of what this code does and how. If you can't explain it clearly, you don't understand it.

Exercise 1.1: The Explanation Challenge

Generate 5 simple functions using AI (string manipulation, basic math, list operations). For each: - Write your explanation *before* running any tests - Then test with various inputs - Compare actual behavior to your explanation - Note any surprises

Track: How often did the code do exactly what you expected? Where were you surprised?

Exercise 1.2: Bug Hunting

Ask AI to generate functions with intentional prompts that might produce subtle bugs: - "Write a function to calculate average" (what if the list is empty?) - "Write a function to find the second-

largest number" (what if there are duplicates?) - "Write a function to check if a string is a palindrome" (what about case? spaces?)

Your job: Find the bugs before running the code. Then verify.

Exercise 1.3: The Trace Journal

Keep a journal this week. For every AI-generated function you use: - Write the input/output trace for at least 2 inputs - Note one edge case you identified - Record whether your analysis found any issues

End of Week Assessment:

You should be able to: - Read a 20-line function and explain what it does without running it - Identify at least 2 edge cases for any function - Predict output for specific inputs before testing

Common Mistakes This Week:

- *Skimming instead of reading.* If you can't explain each line, you skimmed.
- *Testing instead of tracing.* Running the code isn't understanding. Trace first, then test.
- *Accepting "it works" as understanding.* Code that passes tests isn't necessarily understood.

1.14.2 Week 2: Intent Specification Practice

The Goal: Build the habit of knowing what you want before you ask AI for it.

The Problem You're Solving: Most AI interactions start with vague prompts. "Write a function to handle dates." Handle how? What dates? What should happen with invalid dates? Vague prompts produce code that might not match your actual need—and you won't notice because you weren't clear about your need.

Daily Practice (30-45 minutes):

Before any code generation:

1. **Write the function signature first.** What's the name? What are the parameters and their types? What's the return type?
2. **Write example calls.** Give 3-5 concrete examples of how you'd call this function and what it should return.
3. **Write the edge cases.** What happens with empty input? Null? Invalid values? Extremely large values?
4. **Write the error cases.** When should this function fail? How should it fail?

Only after writing all this do you prompt the AI.

Exercise 2.1: The Spec-First Challenge

Choose 5 programming problems (sorting, searching, string manipulation, etc.). For each:

1. Write a complete specification (signature, examples, edges, errors) — spend at least 10 minutes
2. Then prompt AI to implement it

3. Verify the implementation against your specification
4. Note: Did the AI match your spec? Where did it differ?

Exercise 2.2: Spec Critique

Look at these vague intents. Rewrite each as a complete specification:

Vague: "Function to validate email" - What makes an email valid? - What should return type be? Boolean? Error message? - What about edge cases like very long emails, unicode characters, multiple @ signs?

Vague: "Function to split a bill" - Split how? Evenly? By item? By percentage? - What about tax? Tip? - What if amounts don't divide evenly? - What's the input format? Output format?

Vague: "Function to check password strength" - What criteria? Length? Character types? Common passwords? - What's the output? Boolean? Score? Specific feedback? - What are the thresholds?

Write complete specs for each. Compare with a peer if possible.

Exercise 2.3: The Retroactive Spec Test

Find code you wrote (or AI generated) last week without a clear spec. Now write the spec that *should* have existed. Then check: does the code actually meet this spec? Often you'll find it doesn't—because you weren't clear about what you needed.

End of Week Assessment:

You should be able to: - Write a complete spec (signature, examples, edges, errors) in 10 minutes - Identify 3+ ambiguities in any vague requirement - Verify implementation against spec systematically

Common Mistakes This Week:

- *Specs that are too vague.* "Handle errors appropriately" isn't a spec. How? Which errors?
- *Forgetting edge cases.* Empty, null, very large, very small, duplicates, negative—always consider these.
- *Skipping to prompting.* The temptation to just ask AI is strong. Resist. Spec first.

1.14.3 Week 3: Calibrating Trust

The Goal: Develop judgment about when to verify deeply vs. when to verify lightly.

The Problem You're Solving: You can't verify everything deeply—there isn't time. But you can't verify everything lightly—too many bugs will escape. You need judgment about what deserves attention.

Daily Practice (30-45 minutes):

For every piece of code you work with:

1. **Assess impact.** What's the worst case if this code is wrong? Categorize: trivial / annoying / problematic / serious / severe

2. **Assess reversibility.** If this breaks in production, how easily can we fix it? Categorize: instant / easy / difficult / very hard / impossible

3. **Assess exposure.** Who/what is affected? Just me / my team / internal users / external users / everyone
4. **Decide verification depth.** Based on assessments, choose: quick check / standard review / deep analysis / intensive verification

Exercise 3.1: Risk Calibration Scenarios

For each scenario, assess impact, reversibility, exposure, and decide verification depth:

1. A logging function that formats debug messages
2. Input validation for a public signup form
3. A calculation for displaying estimated shipping dates
4. Password hashing for user authentication
5. A function that archives old records to cold storage
6. A utility to generate unique IDs for database records
7. An endpoint that deletes user accounts
8. A function that formats currency for display
9. Rate limiting logic for an API
10. A report generator for internal dashboards

After deciding, compare with this guide: - #1, #3, #8, #10: Lower risk (but not zero—verify basics) - #2, #4, #6, #9: Medium to high risk (careful verification) - #5, #7: High risk (intensive verification—destructive actions)

Exercise 3.2: Your Risk History

Think about code you wrote or used recently. For each piece: - What was your implicit risk assessment? - How much verification did you actually do? - In retrospect, was that appropriate?

Many developers discover they over-verify trivial code (it's easy) and under-verify important code (it's harder and they're tired).

Exercise 3.3: The Calibration Journal

This week, for every verification decision: - Record what you verified and how deeply - Record your reasoning - At end of week, review: Do your decisions make sense? Any patterns of over/under verification?

End of Week Assessment:

You should be able to: - Assess risk across impact, reversibility, and exposure quickly - Match verification depth to risk level consistently - Explain why you chose a particular verification depth

Common Mistakes This Week:

- *Everything is high risk.* If everything is critical, nothing is. Differentiate.
- *Familiarity bias.* Unfamiliar code feels risky, familiar code feels safe. Risk is about impact, not familiarity.
- *Effort avoidance.* Calling something "low risk" to avoid verification work. Be honest.

1.15 Phase 2: Building Verification Skills (Weeks 4-7)

1.15.1 Week 4: Functional Verification

The Goal: Learn to verify that code does what it's supposed to do.

The Problem You're Solving: "It works" is not verification. Code that works for one input might fail for others. Functional verification systematically checks that code meets its specification across its intended domain.

Daily Practice (45-60 minutes):

For each function you work with:

1. **Identify categories of inputs.** What are the meaningfully different kinds of inputs? (e.g., for a search: found vs. not found, empty list, one item, many items)
2. **Select representatives.** Pick at least one input from each category.
3. **Determine expected output.** Before running, what should the output be?
4. **Test and compare.** Run the code, compare to expectations.
5. **Investigate surprises.** If output differs from expectation, is your expectation wrong, or is the code wrong?

Exercise 4.1: Category Analysis

For each function type, list input categories that should be tested:

1. A function that finds the maximum value in a list
   - List with one item
   - List with multiple items (max at start, middle, end)
   - List with all same values
   - List with negative numbers
   - Empty list
   - List with very large numbers
2. A function that validates a phone number
   - (You identify the categories)
3. A function that calculates days between two dates
   - (You identify the categories)
4. A function that merges two sorted lists
   - (You identify the categories)

Exercise 4.2: Test Case Design

Take a function you've recently generated with AI. Write test cases that cover: - Normal operation (typical inputs) - Boundary conditions (edges of valid input ranges) - Error conditions (invalid inputs) - Special cases (empty, null, very large)

Aim for 8-12 test cases for a single function. Run them. Did any fail?

Exercise 4.3: The Mutation Game

This exercise builds intuition for what tests actually verify.

1. Take a working, tested function

2. Introduce a small bug intentionally (change a < to <=, change a + to -, off-by-one error)
3. Run your tests
4. Did any test catch the bug?

If no test caught it, your tests have a gap. What test would have caught it?

This teaches you that tests only verify what they test. Passing tests doesn't mean no bugs—it means no bugs *that the tests check for*.

End of Week Assessment:

You should be able to: - Identify 6+ input categories for any function - Design test cases that cover categories systematically - Recognize when tests are insufficient (gaps in coverage)

Common Mistakes This Week:

- *Happy path only.* Testing only normal operation misses most bugs.
- *Random testing.* Throwing random inputs isn't systematic. Categorize.
- *Trusting test passage.* Tests only verify what they test. Passing tests correct code.

1.15.2 Week 5: Semantic Verification

The Goal: Learn to verify that code does what was *intended*, not just what was *specified*.

The Problem You're Solving: Specifications are imperfect. They describe what we think we want, but we often have implicit expectations we didn't specify. Semantic verification bridges the gap between specification and true intent.

Daily Practice (45-60 minutes):

For each piece of functionality:

1. **Ask "why?"** Why does this code exist? What problem does it solve? What would the user expect?
2. **Check alignment.** Does the code's behavior align with the underlying purpose, not just the written spec?
3. **Consider context.** How will this be used in practice? Does behavior make sense in that context?
4. **Identify implicit expectations.** What behaviors would users/callers assume even if not specified?

Exercise 5.1: Intent vs. Spec Analysis

For each scenario, identify the gap between spec and intent:

Scenario 1: Spec: "Function returns user's age in years" Code: Returns the difference between current year and birth year Intent gap: What about someone born in December asked in January? They'd be "10" when they're actually still 9.

Scenario 2: Spec: "Function splits a string into words" Code: Splits on space character Intent gap: (What about multiple spaces? Tabs? Newlines?)

Scenario 3: Spec: "Function calculates shipping cost based on weight" Code: Returns weight * rate Intent gap: (What implicit expectations might exist?)

Scenario 4: Spec: "Function checks if username is available" Code: Returns true if username not in database Intent gap: (What about case sensitivity? Reserved names? SQL injection?)

Exercise 5.2: The User Story Test

Take a function you're working with. Write a brief user story: "As a [user type], I want to [action] so that [benefit]"

Now review the code through the user's eyes: - Does the behavior make sense for this user? - Would the user be surprised by any behavior? - What would the user assume that might not be true?

Exercise 5.3: The Peer Explanation

Find a peer (or imagine one). Explain what a piece of code is *for*, not just what it *does*. As you explain, you'll often notice gaps:

"This function validates emails… well, actually it just checks for an @ symbol… but for our purposes that's probably… hmm, actually someone could put 'x@x' and it would pass… is that okay?"

The act of explaining intent reveals gaps between code and purpose.

End of Week Assessment:

You should be able to: - Articulate the purpose behind any code you work with - Identify at least 2 implicit expectations for any function - Recognize when code technically meets spec but violates intent

Common Mistakes This Week:

- *Stopping at spec compliance.* "It does what the spec says" isn't enough if the spec is incomplete.
- *Ignoring context.* Code exists in a system. Behavior that's fine in isolation might be wrong in context.
- *Assuming your interpretation is shared.* Your implicit expectations might differ from others'. Surface them.

1.15.3 Week 6: Security Verification

The Goal: Learn to identify security implications in code.

The Problem You're Solving: AI generates code that works. It rarely generates code that's secure against adversarial input. Security thinking requires considering how code could be misused, not just how it's intended to be used.

Prerequisite Understanding:

Before this week, ensure you understand these concepts (research if needed): - Input validation and sanitization - Injection attacks (SQL, command, XSS) - Authentication vs. authorization - Data exposure risks - The principle of least privilege

Daily Practice (45-60 minutes):

For each piece of code:

1. **Identify inputs.** What data enters this code? From where?
2. **Assume hostility.** What if that input comes from an attacker? What's the worst they could do?
3. **Trace sensitive data.** If this code touches sensitive data, where does that data go? Could it be exposed?
4. **Check access control.** Does this code verify the caller has permission to do what they're asking?
5. **Look for trust boundaries.** Where does code assume input is safe? Are those assumptions valid?

Exercise 6.1: Attack Vector Identification

For each code pattern, identify potential attack vectors:

```
# Pattern 1: Database query
def get_user(username):
    query = f"SELECT * FROM users WHERE name = '{username}'"
    return db.execute(query)
```

Attack vectors: (SQL injection - what if username is "'; DROP TABLE users; –"?)

```
# Pattern 2: File access
def read_config(filename):
    with open(f"/app/config/{filename}") as f:
        return f.read()
```

Attack vectors: (What if filename is "../../../etc/passwd"?)

```
# Pattern 3: HTML output
def display_comment(comment):
    return f"<div class='comment'>{comment}</div>"
```

Attack vectors: (What if comment contains

?)

```
# Pattern 4: Command execution
def convert_image(input_path, output_path):
    os.system(f"convert {input_path} {output_path}")
```

Attack vectors: (What if paths contain shell metacharacters?)

Exercise 6.2: Security Review Practice

Take a piece of AI-generated code that handles user input. Perform a security review:

1. List every input to the code
2. For each input, list what an attacker could provide
3. For each attack input, trace what happens
4. Identify which attacks succeed and which are blocked
5. For successful attacks, what's the fix?

Exercise 6.3: The STRIDE Walkthrough

STRIDE is a threat modeling framework. For a function you're reviewing, consider each threat:

- **S**poofing: Could someone pretend to be someone else?
- **T**ampering: Could someone modify data they shouldn't?
- **R**epudiation: Could someone deny they did something?
- **I**nformation disclosure: Could sensitive data be exposed?
- **D**enial of service: Could someone make this unavailable?
- **E**levation of privilege: Could someone gain unauthorized access?

Not every threat applies to every function, but considering all six ensures you don't miss anything.

End of Week Assessment:

You should be able to: - Identify inputs and trust boundaries in any code - Recognize common vulnerability patterns (injection, path traversal, XSS) - Apply threat modeling framework to code review

Common Mistakes This Week:

- *Assuming inputs are safe.* Never trust input. Ever.
- *Security as afterthought.* Security issues are cheaper to fix early. Always consider security.
- *"No one would do that."* Attackers do unexpected things. That's the point.

1.15.4 Week 7: Maintainability Verification

The Goal: Learn to evaluate whether code can be understood and modified in the future.

The Problem You're Solving: Code is read more than it's written. AI generates code that works but is often hard to understand or modify. Maintainability verification ensures code won't become a burden.

Daily Practice (45-60 minutes):

For each piece of code:

1. **Read without context.** Can someone who didn't write this understand what it does?
2. **Predict modification.** What changes are likely in the future? How hard would they be?
3. **Identify complexity.** Where is this code hard to follow? Why?
4. **Check documentation.** Are there comments where needed? Are they accurate? Are they helpful?
5. **Evaluate naming.** Do names convey meaning? Would you understand the purpose from names alone?

Exercise 7.1: The Stranger Test

Take code you generated last week. Pretend you've never seen it before. Time yourself: - How long until you understand what it does? - How long until you could confidently modify it? - What confused you?

If understanding takes more than a few minutes for simple code, maintainability is poor.

Exercise 7.2: Future Modification Prediction

For a piece of code, list 5 changes that might be needed in the future: - New feature addition - Bug fix - Performance improvement - Integration with new system - Behavior change

For each change, assess: - How hard would this change be? - What would you have to understand? - What could go wrong?

Code that makes changes hard has poor maintainability.

Exercise 7.3: Refactoring for Clarity

Take an AI-generated function that works but is unclear. Refactor it purely for readability: - Better variable names - Extract confusing logic into well-named helper functions - Add comments where truly needed (not obvious things) - Simplify complex conditions

Before/after comparison: is it easier to understand?

Exercise 7.4: The Documentation Challenge

Write documentation for a function that has none: - What does it do? - What are the parameters and what do they mean? - What does it return? - What can go wrong? - What are the assumptions/prerequisites?

Now check: did you have to read the code deeply to write this? If the code had been clear, documentation would have been quick.

End of Week Assessment:

You should be able to: - Estimate time-to-understand for code - Identify maintainability problems (complexity, naming, documentation) - Refactor for clarity without changing behavior

Common Mistakes This Week:

- *"I understand it."* You wrote it (or just accepted it). Of course you understand it now. The question is whether future-you or others will.
- *Over-documentation.* Comments that repeat what code clearly says are noise.
- *Under-documentation.* Complex logic with no explanation is a burden.

1.16 Phase 3: Integration and Judgment (Weeks 8-11)

1.16.1 Week 8: Putting It Together

The Goal: Integrate all verification skills into a cohesive practice.

The Problem You're Solving: You've learned functional, semantic, security, and maintainability verification separately. Now you need to apply them together, efficiently.

Daily Practice (60 minutes):

Work through complete development tasks using full VID practice:

1. **Spec first.** Write intent specification before generating
2. **Assess risk.** Determine appropriate verification depth

3. **Generate.** Use AI to generate implementation
4. **Verify functionally.** Does it do what the spec says?
5. **Verify semantically.** Does it do what was intended?
6. **Verify security.** Is it safe against adversarial input?
7. **Verify maintainability.** Can it be understood and modified?
8. **Document decisions.** Record what you verified and why

Exercise 8.1: End-to-End Practice

Complete these tasks using full VID methodology:

Task 1: Build a function that validates and formats phone numbers - Write complete spec - Assess risk (this likely handles user input) - Generate implementation - Verify across all dimensions - Document your verification

Task 2: Build a function that calculates compound interest - Same process

Task 3: Build a function that parses and validates JSON configuration - Same process

For each, time yourself. Where do you spend time? Where could you be more efficient?

Exercise 8.2: The Prioritization Challenge

You have 30 minutes to verify 5 functions. How do you allocate time?

1. A logging utility that formats timestamps
2. A function that authenticates API requests
3. A helper that capitalizes names
4. A function that processes payment amounts
5. A function that generates unique session IDs

Think about: - Which need deep verification? - Which can be verified quickly? - What's your time allocation?

Practice making these decisions quickly. Risk calibration should become instinctive.

Exercise 8.3: Verification Journal

This week, for every verification you perform: - What did you verify? - What depth did you choose? - Why? - What did you find?

Review at end of week: Are your decisions consistent? Are they appropriate?

End of Week Assessment:

You should be able to: - Apply full VID process end-to-end - Allocate verification time based on risk - Verify efficiently without cutting corners

Common Mistakes This Week:

- *Sequential thinking.* You don't have to verify in strict order. Let risk guide attention.
- *Uniform depth.* Not everything needs the same attention.
- *Skipping documentation.* Record your decisions. Future you will thank you.

1.16.2 Week 9: Building Speed Without Sacrificing Depth

The Goal: Become faster at verification without reducing quality.

The Problem You're Solving: Thorough verification takes time. But you're under time pressure. You need to be efficient—doing what matters, skipping what doesn't—without dropping important checks.

Daily Practice (60 minutes):

Focus on verification efficiency:

1. **Time your verifications.** How long does each type of check take?
2. **Identify shortcuts.** What checks can be done together? What patterns indicate you can skip something?
3. **Build checklists.** What do you always check? Make it automatic.
4. **Practice pattern recognition.** What patterns indicate problems? Learn to see them quickly.

Exercise 9.1: Timed Verification

Verify 5 functions with a strict time limit: - 2 minutes each for low-risk code - 5 minutes each for medium-risk code - 15 minutes each for high-risk code

What can you accomplish in these times? What must you skip? What would you need to go back and check given more time?

Exercise 9.2: Pattern Library

Build a personal library of "things to watch for": - Patterns that often indicate bugs - Patterns that often indicate security issues - Patterns that often indicate maintainability problems

Every time you find a bug, add the pattern to your library. Over time, you'll spot these patterns instantly.

Exercise 9.3: Checklist Development

Create verification checklists for common code types:

For any function: - [ ] Understand what it does - [ ] Check edge cases (empty, null, large) - [ ] Verify against spec - [ ] Check error handling

For functions that handle user input: - [ ] Input validation - [ ] Injection risk assessment - [ ] Error message safety (no sensitive info)

For functions that touch data storage: - [ ] Check for data loss scenarios - [ ] Verify transactions/rollbacks - [ ] Check concurrent access safety

Checklists make verification faster by removing decision overhead.

End of Week Assessment:

You should be able to: - Complete appropriate verification within time constraints - Identify which checks are essential vs. optional - Recognize common problem patterns quickly

Common Mistakes This Week:

- *Speed over quality.* Faster is only good if quality is maintained.
- *Rigid checklists.* Checklists help but judgment overrides. Don't skip important checks because they're not on the list.
- *Ignoring intuition.* If something feels wrong, investigate. Speed shouldn't silence intuition.

1.16.3 Week 10: Handling Complexity

The Goal: Learn to verify complex code that exceeds simple function analysis.

The Problem You're Solving: Real code isn't always simple functions. You'll face complex logic, interacting components, and systems that can't be verified by looking at one piece in isolation.

Daily Practice (60 minutes):

Work with increasingly complex code:

1. **Multi-function logic.** Verify code that spans multiple functions
2. **Stateful systems.** Verify code that maintains state
3. **Concurrent code.** Verify code that runs in parallel
4. **Integration points.** Verify code that interacts with external systems

Exercise 10.1: Call Chain Analysis

Take a function that calls other functions. Trace the full execution: - What does each called function do? - How do they compose? - What assumptions does each make about the others? - Where could the chain break?

Exercise 10.2: State Machine Verification

Take code that maintains state (e.g., a user session, a shopping cart, a workflow engine): - What are the possible states? - What transitions are allowed? - What should never happen? - Verify state transitions are correct - Look for invalid state possibilities

Exercise 10.3: Integration Point Analysis

Take code that calls an external API or database: - What happens if the external system is slow? - What happens if it's unavailable? - What happens if it returns unexpected data? - What happens if it returns errors?

Verify that the code handles these scenarios appropriately.

End of Week Assessment:

You should be able to: - Trace and verify multi-function logic - Verify stateful code systematically - Identify integration failure modes

Common Mistakes This Week:

- *Function-level tunnel vision.* Complex systems fail at interactions, not just within functions.
- *Happy path integration.* External systems fail. Verify failure handling.
- *Ignoring state.* Stateful bugs are subtle. State machine thinking helps.

1.16.4 Week 11: Developing Judgment

The Goal: Develop verification judgment that handles novel situations.

The Problem You're Solving: You've learned practices for common situations. But you'll face novel situations where no checklist applies. You need judgment—the ability to figure out what to verify when there's no template.

Daily Practice (60 minutes):

Work with unfamiliar code and domains:

1. **Face novelty.** Work with code types you haven't seen before
2. **Derive approaches.** Figure out verification approaches from first principles
3. **Learn from outcomes.** Track what works and what doesn't
4. **Build intuition.** Develop gut feelings, then verify them

Exercise 11.1: Unknown Domain

Get AI to generate code in a domain you don't know (e.g., graph algorithms, cryptography, signal processing). You can't verify by domain knowledge. How do you approach it?

Options: - Learn enough domain knowledge to verify - Verify structural properties (no crashes, handles edges) - Test against known correct implementations - Consult domain experts

Explore each approach. What can you verify without domain knowledge? What requires expertise?

Exercise 11.2: First Principles Verification

For code in an unfamiliar pattern, ask: - What could go wrong with this type of code? - What would be really bad if it happened? - How would I detect if it went wrong? - What would give me confidence it's right?

Derive your verification approach from these questions.

Exercise 11.3: Judgment Calibration

For code you've verified, predict: - How confident am I this is correct? (percentage) - What's the most likely problem I missed?

Track outcomes. When problems occur, were they things you predicted? Is your confidence calibrated (i.e., code you're 90% confident about should be right 90% of the time)?

End of Week Assessment:

You should be able to: - Approach unfamiliar code types systematically - Derive verification strategies from first principles - Calibrate confidence appropriately

Common Mistakes This Week:

- *Defaulting to patterns.* Novel situations require novel approaches. Don't force familiar patterns.
- *Overconfidence.* Unknown domains have unknown unknowns. Calibrate confidence appropriately.
- *Paralysis.* Don't let unfamiliarity stop you. Do what you can, acknowledge what you can't.

1.17 Phase 4: Mastery and Teaching (Week 12+)

1.17.1 Week 12: Consolidation and Sharing

The Goal: Consolidate your learning and begin sharing with others.

The Problem You're Solving: Mastery isn't just personal skill—it's being able to help others develop skill. Teaching reinforces learning and multiplies your impact.

Activities This Week:

1. **Review your journey.** Look back at your journals and notes. What did you learn? How did you change?
2. **Identify your patterns.** What verification approaches work best for you? What mistakes do you keep making?
3. **Teach someone else.** Explain VID to someone who doesn't know it. Teaching reveals what you don't understand.
4. **Contribute to team practices.** Propose improvements to how your team handles AI-generated code.

Exercise 12.1: The Retrospective

Write a personal retrospective: - What verification skills improved most? - What's still weak? - What was most surprising? - What would you tell past-you starting this journey?

Exercise 12.2: Teaching Practice

Teach a concept from this curriculum to someone else: - Explain risk calibration - Walk through a verification ritual - Demonstrate security verification

Did you explain it clearly? Where did they get confused? What would you explain differently?

Exercise 12.3: Process Proposal

Write a proposal for your team: - What VID practices would benefit the team? - How would you introduce them? - What resistance do you expect? - How would you measure success?

End of Week Assessment:

You should be able to: - Articulate your verification approach clearly - Teach VID concepts to others - Propose practical improvements to team practices

---

1.17.2 Beyond Week 12: Continuous Development

This curriculum builds foundational verification skills. Mastery continues forever.

Ongoing Practices:

- **Keep learning patterns.** Every bug you find teaches you a pattern. Build your library.
- **Keep calibrating.** Track outcomes. Adjust practices based on reality.
- **Keep teaching.** Explaining to others deepens your own understanding.

- **Keep adapting.** AI capabilities change. Verification needs change with them.

Signs of Growing Mastery:

- Verification becomes faster without losing depth
- You spot problems others miss
- Your confidence calibration improves (your predictions match outcomes)
- Others seek your review
- You can explain why, not just what

Signs You Need More Work:

- Problems consistently escape your review
- You can't explain why you check what you check
- Verification feels like checklist compliance rather than understanding
- You struggle with novel situations

1.18 Chapter 14: For Senior Engineers

1.18.1 The Challenge

You've spent years building skills in writing code. AI has commoditized much of that skill. Your role is changing whether you like it or not.

1.18.2 The Opportunity

Your experience becomes more valuable, not less, when properly applied:

Judgment. You've seen what goes wrong. You know the failure modes. This judgment is essential for calibrating trust and verification depth.

Debugging. When AI-generated code fails in subtle ways, your experience debugging helps diagnose problems that junior engineers would struggle with.

Architecture. AI generates functions and modules. Knowing how to compose them into coherent systems is architecture — still a human skill.

Mentorship. Junior engineers need guidance on verification. Your experience informs what to check and how deeply.

1.18.3 The Shift

Your role shifts from producing code to:

Specifying intent. You know what needs to be built. AI generates it. Your understanding of requirements is the input.

Verifying output. Your experience identifies what AI gets wrong. You're the quality gate.

Calibrating trust. You assess risk and determine appropriate verification levels.

Building team judgment. You help junior engineers develop the verification skills you've built over years.

1.18.4 The Danger

Some senior engineers resist AI-augmented development entirely. This resistance leads to irrelevance as teams that embrace AI move faster.

Other senior engineers over-embrace AI, accepting output without applying their judgment. This throws away the experience advantage.

The VID approach: use your experience to verify better, not to write more.

1.19 Chapter 15: For Teams and Organizations

1.19.1 The Team Challenge

Individual VID adoption helps individual developers. Team adoption provides multiplicative benefits.

1.19.2 Building a Verification Culture

Normalize verification time. Verification isn't overhead — it's the work. Build it into estimates and expectations.

Celebrate catches. When someone's verification catches a problem, that's a win. Make it visible.

Discuss failures. When problems escape verification, discuss what could have caught them. No blame, just learning.

Share patterns. When someone finds effective verification approaches, share them with the team.

1.19.3 Team Calibration

Align on risk levels. The team should share understanding of what constitutes low, medium, and high risk for your context.

Calibrate verification standards. What does "thorough verification" mean for your team? Agree on concrete practices.

Review together. Periodically review as a team: Are our verification practices working? What problems are escaping? What should we adjust?

1.19.4 Organizational Support

Organizations adopting VID should:

Train on verification. Invest in building verification skills, not just tool access.

Allow verification time. If velocity metrics don't account for verification time, people will skip verification.

Track the right metrics. Measure verified outputs, not raw outputs. Measure problems caught, not just problems shipped.

Support calibration. Provide time for teams to review and adjust their practices.

1.20 Chapter 16: Adopting VID

1.20.1 Starting Points

You can begin VID adoption at different scales:

Individual adoption. Start applying VID principles personally. No permission needed. Build skills, see benefits.

Team adoption. Agree as a team to apply VID practices. Align on risk calibration and verification standards.

Organizational adoption. Formalize VID as development methodology. Provide training and tooling support.

1.20.2 Starting Small

Don't try to implement everything at once.

Week 1-2: Practice intent specification. Before generating, articulate what you need and how you'll verify it.

Week 3-4: Add verification rituals. Apply basic rituals consistently. Build the habit.

Week 5-6: Start calibrating. Notice which code needs more verification, which needs less.

Ongoing: Build the learning loop. Track outcomes, adjust practices.

1.20.3 Common Obstacles

"It slows us down." Yes, verification takes time. But debugging, rework, and incidents take more time. VID front-loads small costs to avoid large costs.

"AI is good enough." AI is good at generating code that looks correct. It's not good at guaranteeing correctness, especially for edge cases, security, and maintainability.

"We don't have time." You don't have time to not verify. The time you save by skipping verification you spend on debugging and incidents.

1.21 Chapter 17: The Continuing Evolution

1.21.1 AI Will Keep Improving

AI capabilities are improving rapidly. Some predictions:

- AI will get better at generating correct code
- AI will get better at self-verification
- AI will get better at explaining generated code
- New failure modes will emerge that we can't predict today

1.21.2 VID Will Evolve

The principles endure: - Intent before generation - Graduated trust based on risk - Understanding over mere acceptance - Provenance awareness - Continuous calibration

The practices will evolve. The specific verification rituals appropriate for 2025's AI will differ from those for 2027's AI. Calibration questions will change as AI capabilities change.

1.21.3 Your Role in the Evolution

VID isn't complete. It's a starting point — a framework for thinking about AI-augmented development that you can adapt.

As you practice VID:

- **Notice what works and what doesn't** in your context
- **Develop new practices** that address gaps
- **Share what you learn** with others facing similar challenges
- **Adjust as AI evolves** rather than freezing your approach

The developers who navigate the AI transition successfully will be those who develop judgment about verification. VID provides a framework for building that judgment.

1.22 Chapter 18: Summary

1.22.1 The Core Message

Code generation is no longer the bottleneck. Verification is.

The methodology that fits this reality focuses on: - Building verification skills - Calibrating verification to risk - Maintaining understanding of generated code - Learning from outcomes

1.22.2 The Five Principles

1. **Intent Before Generation:** Never generate without articulating what you need and how you'll verify it.
2. **Graduated Trust:** Match verification depth to risk level.
3. **Understanding Over Acceptance:** Never accept code you don't understand at an appropriate level.
4. **Provenance Awareness:** Know where code comes from and what that implies.
5. **Continuous Calibration:** Adjust practices based on outcomes.

1.22.3 The Core Practices

1. **Intent Specification:** Articulate functional requirements, boundaries, and success criteria before generating.
2. **Verification Rituals:** Apply consistent verification practices after every generation, scaled to risk.
3. **Learning Loop:** Track outcomes and adjust practices based on what you learn.
4. **Provenance Hygiene:** Document and maintain awareness of code origins.

1.22.4 The Path Forward

Start with principles. Apply practices consistently. Calibrate based on outcomes. Build the judgment that verification requires.

The developers who thrive in the AI-augmented future will be those who master verification, not just generation.

VID provides a framework for becoming that developer.

1.23 Chapter 19: Patterns and Anti-Patterns

1.23.1 Why Patterns Matter

Principles tell you what to value. Practices tell you what to do. But patterns show you what it looks like in reality—both when it works and when it fails.

This chapter presents concrete scenarios. Some show VID done well. Others show common failures. Study both. The failures are often more instructive.

1.23.2 Pattern 1: The Specification-First Win

The Scenario:

Maya needs a function to calculate prorated subscription charges. A user upgrades mid-billing-cycle; the function determines how much to charge for the remaining days.

The Anti-Pattern (how it usually goes wrong):

Maya opens her AI assistant and types: "Write a function to calculate prorated subscription charges."

The AI generates something. It looks reasonable. Maya runs a quick test—upgrade on day 15 of a 30-day month. The result seems right. She commits.

Two weeks later, a customer complains they were overcharged. Finance investigates. The function doesn't handle months with different numbers of days correctly. It assumed 30 days always. February charges were wrong. So were charges in 31-day months.

Maya fixes the bug. But the damage is done—customers were overcharged, refunds are needed, trust is damaged.

The Pattern (VID approach):

Maya pauses before prompting. She writes:

```
Function: calculate_prorated_charge
Inputs:
  - original_price: monthly subscription cost
  - upgrade_date: date of upgrade
  - billing_cycle_start: first day of current billing period

Output: dollar amount to charge (rounded to cents)

Behavior:
  - Calculate days remaining from upgrade_date to end of billing cycle
  - Prorate based on actual days in that specific month
  - Round to nearest cent

Edge cases:
  - upgrade_date equals billing_cycle_start: charge full amount
  - upgrade_date is last day of cycle: charge for 1 day
  - February (28/29 days): must handle correctly
  - 31-day months: must handle correctly

Verification:
  - Test with Feb 15 in non-leap year (14 days remaining of 28)
  - Test with Feb 15 in leap year (14 days remaining of 29)
  - Test with Jan 15 (17 days remaining of 31)
  - Test with Jun 15 (15 days remaining of 30)
  - Test upgrade on day 1 (full charge)
  - Test upgrade on last day (1 day charge)
```

Now Maya prompts the AI with this specification. The AI generates code. Maya runs her pre-planned tests. The February test fails—the AI assumed 30 days. Maya catches it before commit.

She either fixes the code or prompts again with explicit instructions about variable month lengths. The final code handles all cases correctly.

The Lesson:

The time Maya spent writing the specification was less than the time Anti-Pattern Maya spent on customer complaints, investigation, fixes, and refunds. Specification isn't overhead—it's the shortest path to correct code.

1.23.3 Pattern 2: Graduated Trust in Action

The Scenario:

A team is building a new feature. It involves three components: 1. A utility to format currency for display 2. An endpoint to update user profile information 3. A function to process refunds

The Anti-Pattern:

The team treats all three equally. Each gets the same cursory review: glance at the code, run a quick test, approve the PR.

Six months later: - The currency formatter has a bug with negative numbers. It displays "-$-50" instead of "-$50". Embarrassing but minor—it only affects a rarely-used report. - The profile endpoint has a bug. It doesn't validate email format. Users enter invalid emails, then can't receive notifications. Support tickets pile up. - The refund function has a serious bug. Under certain conditions, it processes refunds twice. The company loses $200,000 before anyone notices.

The Pattern:

The team assesses each component:

Currency formatter: - Impact if wrong: Minor display issue - Reversibility: Easy fix, redeploy - Exposure: Internal report - **Verdict: High trust, basic verification**

Profile endpoint: - Impact if wrong: User experience problems, support burden - Reversibility: Can fix but affected users remain in bad state - Exposure: All users - **Verdict: Moderate trust, standard verification including input validation review**

Refund function: - Impact if wrong: Financial loss, potentially significant - Reversibility: Money already gone, hard to recover - Exposure: Every refund transaction - **Verdict: Minimal trust, intensive verification**

They allocate time accordingly: - Currency formatter: 10 minutes, automated tests, quick review - Profile endpoint: 30 minutes, systematic input validation check, edge case testing - Refund function: 2 hours, multiple reviewers, explicit test of idempotency, audit trail verification, sign-off from senior engineer

The refund double-processing bug gets caught in review. The profile email validation gets added. The currency formatter ships with minimal review—and yes, it has the negative number bug, but that's discovered and fixed with minimal impact.

The Lesson:

Equal treatment of unequal risks is wasteful at best, catastrophic at worst. The refund bug could have bankrupted a startup. Catching it was worth the extra two hours.

1.23.4 Pattern 3: The Understanding Trap

The Scenario:

Carlos needs to implement a caching layer. He's not an expert in caching, but the AI generates something that looks sophisticated.

The Anti-Pattern:

Carlos prompts: "Implement an LRU cache with TTL support."

The AI generates 80 lines of code with a doubly-linked list, a hash map, timestamp tracking, and cleanup logic. Carlos reads through it. It looks professional. He doesn't fully understand the linked list manipulation, but the tests pass.

He ships it.

Three weeks later, production experiences memory growth. The cache isn't releasing memory correctly. Carlos investigates. He stares at the linked list code. He doesn't understand it. He can't debug what he doesn't understand.

He spends two days learning how doubly-linked lists work in this context, then another day finding the bug: a reference wasn't being cleared during eviction, preventing garbage collection.

Total cost: three days of debugging code he didn't understand. Plus the production incident.

The Pattern:

Carlos prompts for the cache implementation. The AI generates the same sophisticated code.

Carlos reads it. He doesn't fully understand the linked list manipulation. He stops.

He asks himself: "Do I understand this well enough to debug it when it breaks?"

The answer is no.

Carlos has options:

Option A: Learn enough to understand He spends two hours studying how LRU caches work with doubly-linked lists. He traces through the AI's code. Now he understands it. He can verify it properly and debug it later.

Option B: Simplify He prompts again: "Implement a simpler LRU cache using OrderedDict" (in Python) or equivalent simple approach. The result is less sophisticated but understandable. It might be slightly slower, but Carlos can verify and maintain it.

Option C: Get help He asks a colleague who knows caching to review with him. Together they verify the implementation. Carlos learns enough to maintain it.

Carlos chooses Option B. The simpler cache is 20% slower but completely understandable. It ships without issues. When a bug eventually surfaces (wrong TTL calculation), Carlos fixes it in 20 minutes because he understands the code.

The Lesson:

Sophisticated code you don't understand is a liability, not an asset. The time saved by accepting complex AI output is borrowed from your future self, with interest.

1.23.5 Pattern 4: The Provenance Blind Spot

The Scenario:

A team has been using AI assistance heavily for six months. The codebase has grown substantially.

The Anti-Pattern:

No one tracked what was AI-generated versus human-written. Commit messages just say "Add feature X" or "Fix bug Y."

Now they need to audit their code for potential IP issues. A large enterprise customer requires attestation that code doesn't infringe third-party IP.

The team realizes they have no idea which parts of their codebase were AI-generated. They can't assess risk. They can't provide attestation.

Their options: - Manually review the entire codebase (expensive) - Provide attestation without knowing (risky) - Lose the enterprise customer (costly)

They also discover a security pattern that's been copy-pasted throughout the codebase. It has a vulnerability. They don't know where it came from. Did a human write it? Did AI generate it? Was it in one place originally and spread, or generated multiple times?

Without provenance, they can't trace the origin or assess how it spread.

The Pattern:

From the start, the team maintains provenance awareness:

- Commit messages indicate AI assistance: "Add user authentication (AI-assisted, reviewed by @carlos)"
- A simple convention marks sections of code: `# AI-generated: initial implementation` or `# Human: security-critical logic`
- PR descriptions note what was AI-generated and what review it received

Six months later, they face the same enterprise audit request.

They can identify AI-generated code. They can show it was reviewed. They can provide meaningful attestation: "Our codebase includes AI-assisted code in these areas, which underwent our standard review process including security verification."

When the vulnerability pattern appears, they trace it: originally AI-generated in one file, then copied (by humans) to other files. They know exactly where to look and can assess the scope of the problem.

The Lesson:

Provenance tracking costs almost nothing in the moment. Reconstructing provenance later costs enormously—if it's even possible.

1.23.6 Pattern 5: The Security Review That Wasn't

The Scenario:

A developer needs to build a password reset flow. The AI generates code that: - Generates a reset token - Stores it in the database - Emails it to the user - Accepts the token on a reset page - Updates the password

The Anti-Pattern:

The developer reviews the code. It looks correct. The token is generated. It's stored. It's emailed. The reset works. Ship it.

Three months later, attackers discover: - The token is generated using predictable random numbers - There's no expiration on tokens - There's no rate limiting on the reset request endpoint - The token is passed in the URL (and thus logged everywhere) - The same token can be used multiple times

None of these were obviously visible in the code. The code "worked." It just wasn't secure.

The Pattern:

The developer recognizes password reset as security-critical (authentication-adjacent, affects account access). This triggers intensive verification including security review.

For the security review, they consider:

Token generation: - Is the random number generator cryptographically secure? - Is the token long enough to prevent brute force? - Is there sufficient entropy?

Token lifecycle: - Does the token expire? - Is expiration enforced server-side (not just client-side)? - Is the token invalidated after use? - Is it invalidated if the user requests another reset?

Attack surface: - Can attackers enumerate valid tokens? - Is there rate limiting to prevent brute force? - Are failed attempts logged?

Token handling: - Is the token transmitted securely? - Is it stored securely? - Does it appear in logs?

The developer checks each of these against the AI-generated code. They find: - Token generation uses basic random, not cryptographic random → fix - No expiration → add expiration - Token appears in URL query parameters → move to POST body - No rate limiting → add rate limiting - Token works multiple times → add single-use enforcement

The fixed implementation ships without the vulnerabilities.

The Lesson:

Security review isn't checking if code looks right. It's systematically considering how code could be attacked and whether it resists those attacks.

1.23.7 Pattern 6: The Maintenance Nightmare

The Scenario:

A feature was built quickly using heavy AI assistance. It works. It ships. Everyone moves on.

The Anti-Pattern:

A year later, the feature needs modification. The original developer has left. A new developer opens the code.

They find: - 500 lines in a single function - Variables named `temp`, `data`, `result`, `x` - No comments explaining the business logic - Deeply nested conditionals (if inside if inside if inside if) - Magic numbers throughout (why 86400? why 0.035?) - Duplicated code blocks with slight variations

The new developer spends three days just understanding what the code does. They're afraid to change it because they might break something they don't understand.

Eventually, they give up on modifying the existing code. They rewrite the feature from scratch. A month of work to add what should have been a two-day feature.

The Pattern:

During original development, the developer applies maintainability verification:

They ask: "Could someone else understand this in a year?"

The AI-generated code has the same issues: one long function, poor names, no comments.

Before accepting it, the developer refactors: - Breaks the long function into smaller, named functions (`calculate_shipping_cost`, `apply_discount`, `validate_address`) - Renames variables to be meaningful (`seconds_per_day` instead of 86400, `commission_rate` instead of 0.035) - Adds comments explaining *why*, not *what* (the what is in the code; the why is the business logic) - Eliminates duplication by extracting common patterns - Reduces nesting by early returns and guard clauses

This takes an extra hour.

A year later, the new developer opens the code. They understand the structure in 15 minutes. They find the function that needs modification. They make the change in two days as expected.

The Lesson:

An hour of maintainability work saves months of future confusion. AI generates working code, but working code isn't necessarily maintainable code.

1.23.8 Anti-Pattern Catalog

Here are common VID anti-patterns in brief:

"It compiled, so it's correct" Code that compiles can be completely wrong. Compilation is syntax checking, not semantic verification.

"The tests pass" Tests only verify what they test. If the tests don't cover the bug, they won't catch it. Passing tests are necessary but not sufficient.

"I'll understand it later" You won't. Understanding erodes quickly. If you don't understand it now, you'll understand it less in a month.

"This is just like last time" Similar-looking code can have very different behavior. Verify each instance; don't assume patterns transfer.

"AI knows what I meant" AI guesses what you meant based on your words. It often guesses wrong. Specify explicitly.

"It's just internal tooling" Internal tools become critical infrastructure. Today's quick script is tomorrow's system dependency.

"We're moving fast" Fast without verification is fast toward failure. Real speed includes time to verify.

"No one will misuse it" Someone will. Assume adversarial use and verify accordingly.

"The senior dev approved it" Approval isn't verification. Did they actually verify, or did they trust without checking?

"We've always done it this way" Past success doesn't guarantee future success, especially as AI capabilities and risks change.

1.24 Chapter 20: The Verification Toolkit

1.24.1 What This Chapter Is

The previous chapter showed patterns in context. This chapter goes deeper into verification techniques—what to actually check and how.

This is not a checklist to follow mindlessly. It's a toolkit to draw from intelligently. Different code needs different verification. Your job is to select the right tools for the situation.

1.24.2 Part 1: Functional Verification

Functional verification answers: "Does this code do what it's supposed to do?"

1.24.2.1 Technique 1: Input Space Partitioning

The input space is everything a function could receive. Testing everything is impossible. Partitioning makes it tractable.

The Approach:

Divide inputs into categories that should behave the same way. Test at least one representative from each category.

How to Partition:

For numeric inputs: - Negative numbers - Zero - Positive numbers - Very large numbers (near overflow) - Very small numbers (precision issues) - Special values (NaN, Infinity if applicable)

For string inputs: - Empty string - Single character - Typical string - Very long string - Unicode characters - Special characters (quotes, backslashes, newlines) - Null (if the language allows)

For collections: - Empty collection - Single element - Multiple elements - Very large collection - Collection with duplicates - Collection with special values (nulls, etc.)

For dates: - Normal date - Leap year date (Feb 29) - Month boundaries - Year boundaries - Time zone edge cases - DST transitions

Example:

For a function `get_element_at_index(list, index)`:

List partitions: - Empty list → should handle gracefully - Single-element list → index 0 valid, others invalid - Multi-element list → various valid and invalid indices

Index partitions: - Negative index → might be valid (Python) or invalid - Zero → valid if list non-empty - Middle valid index → normal case - Last valid index → boundary - One past end → invalid - Very large → invalid

Combined test cases: 1. Empty list, index 0 → error handling 2. Single element, index 0 → returns element 3. Single element, index 1 → error handling 4. Multi-element, index 0 → returns first 5. Multi-element, middle index → returns middle 6. Multi-element, last index → returns last 7. Multi-element, past end → error handling 8. Multi-element, negative → depends on language semantics

1.24.2.2 Technique 2: Boundary Value Analysis Bugs cluster at boundaries. If something works for 5 and 100, it probably works for 50. But it might fail for 0, 1, or the maximum.

The Approach:

For every boundary, test: - The boundary value itself - Just below the boundary (boundary - 1) - Just above the boundary (boundary + 1)

Common Boundaries:

- Array indices: 0, length-1, length
- Numeric ranges: min, max
- String lengths: 0, 1, max
- Date ranges: start of range, end of range
- Pagination: first page, last page

Example:

For a function `is_valid_percentage(value)` that should accept 0-100:

Boundaries: 0 and 100

Test cases: - -1 → should be invalid - 0 → should be valid - 1 → should be valid - 99 → should be valid - 100 → should be valid - 101 → should be invalid

Also consider: - What about 50.5? Are decimals valid? - What about null/undefined?

1.24.2.3 Technique 3: State Transition Testing For code with state, verify that state transitions are correct.

The Approach:

1. Identify all possible states
2. Identify all events that cause transitions
3. Map which transitions are valid
4. Test that valid transitions work
5. Test that invalid transitions are rejected

Example:

For an order state machine:

States: Created, Paid, Shipped, Delivered, Cancelled, Refunded

Valid transitions: - Created → Paid (payment received) - Created → Cancelled (user cancels) - Paid → Shipped (fulfillment ships) - Paid → Refunded (refund before ship) - Shipped → Delivered (delivery confirmed) - Shipped → Refunded (refund after ship)

Invalid transitions: - Delivered → Cancelled (can't cancel delivered order) - Cancelled → anything (cancelled is terminal) - Shipped → Paid (can't go backward)

Test each valid transition. Test key invalid transitions to ensure they're rejected.

1.24.2.4 Technique 4: Property-Based Reasoning Instead of specific test cases, verify properties that should always hold.

The Approach:

Identify invariants—things that should always be true—and verify them.

Common Properties:

- **Round-trip:** encode then decode should return original
- **Idempotency:** doing something twice should be same as once (when applicable)
- **Symmetry:** reverse(reverse(x)) should equal x
- **Ordering:** sort(x) should be ordered
- **Size preservation:** map over a list should preserve length
- **Bounds:** output should be within expected range

Example:

For a serialization function:

Property: For all valid inputs, `deserialize(serialize(input)) == input`

For a sort function:

Properties: - Output length equals input length - Every element in input exists in output (same count) - Output is ordered (each element next element)

For an idempotent API endpoint:

Property: `POST /resource` twice with same data should have same result as once

---

1.24.3 Part 2: Security Verification

Security verification answers: "Can this code be exploited?"

1.24.3.1 Technique 1: Input Vector Enumeration The Approach:

List every input to the code. For each, ask: "What if this came from an attacker?"

Input Vectors Include:

- URL parameters
- Form fields
- HTTP headers
- Cookies

- File uploads
- Database contents (might be poisoned)
- Environment variables (in some contexts)
- File system contents
- Data from external APIs

For Each Input:

- What's the expected format?
- What happens with unexpected format?
- Can special characters cause problems?
- Can excessive length cause problems?
- Can the input influence SQL, commands, HTML, or other interpreted contexts?

Example:

For a user profile update endpoint:

Inputs: - Username (from form) - Email (from form) - Bio (from form) - Profile picture (file upload) - Session token (cookie) - User ID (might be in URL or derived from session)

For each: - Username: Could contain script tags (XSS if displayed), SQL injection characters, or path traversal characters - Email: Could be malformed, excessively long, contain injection characters - Bio: Prime XSS target, could contain huge content (DoS) - Profile picture: Could be disguised executable, excessively large, specially crafted to exploit image processors - Session token: Could be forged, stolen, expired - User ID: Could be manipulated to update someone else's profile

1.24.3.2 Technique 2: Injection Point Analysis **The Approach:**

Find every place where input data is combined with commands, queries, or markup. These are injection points.

Common Injection Contexts:

SQL:

```
# DANGEROUS
query = f"SELECT * FROM users WHERE name = '{user_input}'"

# Input: "'; DROP TABLE users; --"
# Result: SQL injection
```

Shell commands:

```
# DANGEROUS
os.system(f"convert {user_filename} output.png")

# Input: "file.jpg; rm -rf /"
# Result: Command injection
```

HTML:

```
# DANGEROUS
html = f"<div>Welcome, {username}</div>"
```

```
# Input: "<script>steal_cookies()</script>"
# Result: XSS
```

File paths:

```
# DANGEROUS
path = f"/app/data/{user_filename}"

# Input: "../../../etc/passwd"
# Result: Path traversal
```

Verification Questions:

- Is user input ever concatenated into SQL? → Use parameterized queries
- Is user input ever passed to shell? → Avoid if possible; whitelist if necessary
- Is user input ever rendered as HTML? → Escape output
- Is user input ever used in file paths? → Validate against whitelist

1.24.3.3 Technique 3: Authentication and Authorization Audit **The Approach:**

For every action the code enables, verify: 1. Does it require authentication? Should it? 2. Does it verify authorization? Should it? 3. Can authentication be bypassed? 4. Can authorization be bypassed?

Common Vulnerabilities:

Broken authentication: - Weak password requirements - No brute force protection - Predictable session tokens - Session doesn't expire - Password in URL (logged)

Broken authorization: - Checking if user is logged in, but not if they own the resource - Client-side authorization checks only - Assuming sequential IDs can't be guessed - Not re-checking permissions on sensitive actions

Verification Questions:

- What proves this user is who they claim?
- What proves this user can perform this action on this resource?
- Where are these checks? Are they before the action?
- Can they be bypassed by modifying the request?

Example:

For an endpoint `DELETE /api/documents/{id}`:

Questions: - Does it require authentication? (Should: yes) - Does it verify the user owns document {id}? (Should: yes) - What if I'm logged in as User A and request delete of User B's document? - What if the ID is guessed or enumerated?

1.24.3.4 Technique 4: Data Exposure Analysis **The Approach:**

Trace every piece of sensitive data through the code. Where does it go? Could it leak?

Sensitive Data Includes:

- Passwords (should never be visible)
- API keys and secrets
- Personal information (PII)
- Financial information
- Session tokens
- Internal system information

Exposure Points:

- Log files (are sensitive values logged?)
- Error messages (do they reveal internal information?)
- API responses (is too much data returned?)
- URLs (sensitive data in query strings is logged)
- Client-side storage (localStorage is readable by scripts)
- Source code (hardcoded secrets?)

Verification Questions:

- Is this sensitive data logged anywhere?
- Does any error message reveal this data?
- Is this data sent to the client when it shouldn't be?
- Is this data stored securely at rest?
- Is this data transmitted securely in transit?

Example:

For a login function:

Sensitive data: password, session token

Trace: - Password received → is it ever logged? (Should: no) - Password compared → is the comparison constant-time? (Prevents timing attacks) - Password stored → is it hashed properly? (bcrypt, argon2, not MD5/SHA1) - Session token generated → is it cryptographically random? (Should: yes) - Session token stored → is it accessible to JavaScript? (httpOnly flag) - Session token transmitted → is it HTTPS only? (secure flag)

1.24.4 Part 3: Maintainability Verification

Maintainability verification answers: "Can this code be understood and safely modified in the future?"

1.24.4.1 Technique 1: The Stranger Test The Approach:

Pretend you've never seen the code. Time how long it takes to understand: 1. What does this code do? (purpose) 2. How does it do it? (mechanism) 3. Why does it do it this way? (rationale)

Interpretation:

- < 5 minutes for simple code: Good
- 5-15 minutes: Acceptable for complex logic
- 15+ minutes for code that should be simple: Problem

What Makes Understanding Hard:

- Poor naming (what is `x`? what is `process`?)
- Long functions (> 30 lines needs scrutiny, > 100 lines is a problem)
- Deep nesting (more than 3 levels needs refactoring)
- Magic values (what is 86400? what is 0.035?)
- Missing context (why is this special case here?)
- Clever tricks (clever code is hard to debug)

1.24.4.2 Technique 2: Change Impact Assessment The Approach:

Consider likely future changes. How hard would they be?

Common Changes:

- Adding a new case/variant to existing logic
- Changing a business rule or threshold
- Integrating with a new external system
- Adding logging or monitoring
- Fixing a bug in one scenario without breaking others

For Each Change:

- What code would need to be modified?
- What else might be affected?
- How confident are you that you could make the change without breaking something?

Red Flags:

- "I'd need to change this in many places" → duplication problem
- "I'm not sure what else this would affect" → coupling problem
- "I'd need to understand all of this to change any of it" → cohesion problem
- "I'd be afraid to touch this" → clarity problem

1.24.4.3 Technique 3: Naming Audit The Approach:

Read just the names—function names, variable names, parameter names. Do they tell a story?

Good Names:

- Functions: verbs that describe what they do (`calculate_shipping_cost`, `validate_email`, `send_notification`)
- Variables: nouns that describe what they hold (`user_email`, `total_price`, `is_active`)
- Booleans: questions that make sense in conditions (`is_valid`, `has_permission`, `should_retry`)
- Constants: describe what the value means (`MAX_RETRY_COUNT`, `SECONDS_PER_DAY`)

Bad Names:

- Single letters (except loop counters): `x`, `d`, `t`
- Generic names: `data`, `temp`, `result`, `value`, `item`
- Misleading names: `list` that's actually a dictionary, `count` that's actually a sum
- Abbreviated names: `usr`, `pwd`, `cfg` (use `user`, `password`, `config`)

Verification:

For each name, can you understand its purpose without reading the implementation?

1.24.4.4 Technique 4: Complexity Assessment **The Approach:**

Identify complexity and decide if it's justified.

Complexity Indicators:

- **Cyclomatic complexity:** Number of decision points (if, while, for, case). Higher = more complex. Over 10 warrants scrutiny.
- **Nesting depth:** How deep do control structures nest? More than 3 levels is hard to follow.
- **Function length:** Longer functions are harder to understand. Over 30 lines warrants scrutiny.
- **Parameter count:** Functions with many parameters are hard to use correctly. Over 4 parameters warrants scrutiny.
- **Dependencies:** How many other modules does this code depend on? More dependencies = more fragile.

Complexity Isn't Always Bad:

Some problems are inherently complex. The question is whether the code's complexity matches the problem's complexity.

- Simple problem, complex code → refactor
- Complex problem, complex code → acceptable but document
- Complex problem, simple code → ideal (if it's actually correct)

1.24.5 Part 4: Provenance Verification

Provenance verification answers: "Where did this code come from, and what does that imply?"

1.24.5.1 Technique 1: Origin Documentation **The Approach:**

For every significant piece of code, document: - Who/what created it (human, AI, copied from somewhere) - When it was created - What verification it received

For AI-Generated Code:

- Which model/tool generated it
- What prompt produced it (summary if long)
- What verification was performed after generation
- Who verified it

Why It Matters:

- AI-generated code might have training data issues
- Code that was lightly reviewed is higher risk for modification
- Knowing origin helps assess appropriate trust level

1.24.5.2 Technique 2: License Compatibility Check The Approach:

Verify that code origins are compatible with your project's licensing.

Check:

- If code was copied from external sources, what's the license?
- If AI-generated, what are the terms of service? Any IP concerns?
- Are all dependencies' licenses compatible with your project?

Common Issues:

- Copying GPL code into proprietary project
- Using code from Stack Overflow without attribution
- AI generating code that closely matches training data (potential copyright issue)

1.24.5.3 Technique 3: Pattern Origin Tracking The Approach:

When you see a pattern repeated, trace its origin.

Questions:

- Where did this pattern start?
- Who introduced it?
- Was it verified when introduced?
- Has it been modified since?

Why It Matters:

- A bad pattern spreads through copy-paste
- Fixing the pattern requires finding all copies
- Understanding origin helps assess risk

1.24.6 Using the Toolkit

These techniques form a toolkit, not a checklist. Selection depends on context.

For Low-Risk Code: - Quick functional verification (key test cases) - Naming audit (basic clarity) - Skip intensive security analysis

For Medium-Risk Code: - Systematic functional verification (input partitioning, boundary analysis) - Basic security review (input vectors, obvious injection points) - Maintainability review (stranger test, naming audit) - Provenance documentation

For High-Risk Code: - Comprehensive functional verification (all techniques) - Intensive security review (all techniques) - Full maintainability assessment (all techniques) - Formal provenance documentation - Multiple reviewers

The skill is matching technique to need. Over time, you develop intuition for what each situation requires. The techniques become second nature—you apply them without consciously thinking through lists.

That's mastery: not following a checklist, but having internalized the toolkit so deeply that you automatically apply what's needed.

---

2 Chapter 21: Test Verification Framework

2.1 The Problem with AI-Generated Tests

When AI generates your code, you verify it with tests. But when AI also generates your tests, who verifies the tests?

This isn't academic. AI-generated tests can: - **Pass while missing critical bugs** — Tests check what AI thought to check, not necessarily what matters - **Test the wrong thing** — Verifying implementation details instead of behavior - **Contain the same logical errors as the code** — AI makes correlated mistakes - **Provide false confidence** — "100% coverage" means nothing if tests are weak

This chapter provides a framework for verifying tests themselves—whether AI-generated or human-written.

2.2 Core Principle: Tests Are Code

Tests require verification, just like production code.

The difference: tests verify production code, so we need *meta-verification* strategies to verify tests.

Meta-verification means using techniques that validate tests without requiring another layer of tests (avoiding infinite regression).

2.3 Part 1: Test Provenance

2.3.1 Why Test Provenance Matters

The origin of a test affects how we verify it:

Human-written test - Likely reflects human understanding of requirements - May miss edge cases humans didn't consider - Usually tests behavior, not implementation

AI-generated from specification - Only as good as the specification - Might test what was specified, not what was intended - May miss implicit requirements

AI-generated from code - Likely tests implementation, not behavior - May duplicate bugs in the code logic - Provides weak verification

AI-generated from examples - Tests known cases well - May miss untested variations - Brittle to code changes

2.3.2 Tracking Test Provenance

Mark tests with their origin:

```python
def test_calculate_prorated_charge():
    """
    Test prorated subscription charge calculation.

    Provenance: AI-generated from specification
    Verified: 2025-12-07 by @engineer
    Review: Expanded edge cases for leap years
    """
    # Test cases...
```

Minimum provenance documentation: - How was this test created? - Who verified it? - What verification was performed?

2.4 Part 2: Meta-Verification Strategies

2.4.1 Strategy 1: Mutation Testing

Concept: Introduce bugs in the code. If tests don't fail, the tests are weak.

How it works: 1. Take working code with passing tests 2. Make small changes (mutations): - Change `<` to `<=` - Change `+` to `-` - Remove a conditional - Change a constant 3. Run tests 4. If tests still pass, they didn't catch the mutation → weak tests

Example:

```python
# Original code
def is_valid_age(age):
    return 0 <= age < 120

# Mutation 1: Change <= to <
def is_valid_age(age):
    return 0 < age < 120  # Should reject age=0

# Mutation 2: Change 120 to 121
def is_valid_age(age):
    return 0 <= age < 121  # Should reject age=120
```

Question: Do your tests catch these mutations?

If not, you need tests like:

```python
def test_age_boundary_zero():
    assert is_valid_age(0) == True  # Catches mutation 1

def test_age_boundary_max():
    assert is_valid_age(120) == False  # Catches mutation 2
```

Tools: - Python: `mutmut`, `cosmic-ray` - JavaScript: `Stryker` - Java: `PIT` - General: `universal-mutator`

VID Practice: - Run mutation testing on AI-generated tests - Aim for >80% mutation score (percentage of mutations caught) - If mutations survive, add tests

---

2.4.2 Strategy 2: Property-Based Testing

Concept: Instead of testing specific examples, test properties that must always hold.

Why it helps: Properties are harder for AI to get wrong than specific test cases.

Example:

Weak (example-based):

```python
def test_reverse():
    assert reverse([1, 2, 3]) == [3, 2, 1]
    assert reverse([5]) == [5]
```

Strong (property-based):

```python
@given(lists(integers()))
def test_reverse_properties(lst):
    # Property 1: Reversing twice returns original
    assert reverse(reverse(lst)) == lst

    # Property 2: Length preserved
    assert len(reverse(lst)) == len(lst)

    # Property 3: Elements preserved (same set)
    assert set(reverse(lst)) == set(lst)

    # Property 4: First becomes last
    if lst:
        assert reverse(lst)[0] == lst[-1]
```

Tools: - Python: `hypothesis` - JavaScript: `fast-check` - Haskell: `QuickCheck` - Java: `jqwik`

VID Practice: - For critical code, supplement AI-generated tests with property tests - Properties are human insight—harder for AI to generate - Use properties to validate AI-generated example tests

---

2.4.3 Strategy 3: Test Review Questions

Systematically review tests using these questions:

2.4.3.1 Correctness Questions

- **Does this test actually test what it claims?**
  - Read the test name and docstring
  - Read the test code

  - Do they match?
- **Are the assertions correct?**
  - Is `assert result == expected` the right comparison?
  - Are edge cases handled correctly?
  - Are error cases actually supposed to error?

2.4.3.2 Completeness Questions

- **What's not being tested?**
  - Edge cases: empty, null, min, max
  - Error conditions
  - Integration points
  - Security scenarios
- **What assumptions do the tests make?**
  - "Input is always valid"
  - "Database is empty before test"
  - "System clock doesn't change mid-test"
  - Are these assumptions documented?

2.4.3.3 Quality Questions

- **Do these tests test behavior or implementation?**
  - Good: "User can log in with correct credentials"
  - Bad: "Login function calls check_password and then create_session"
- **Are the tests brittle?**
  - Will they break if we refactor without changing behavior?
  - Do they depend on implementation details?
- **Are the tests readable?**
  - Can someone else understand what's being tested?
  - Are test names descriptive?

2.4.4 Strategy 4: Coverage Analysis (Used Correctly)

Coverage is necessary but not sufficient.

What coverage tells you: - Which lines of code are executed by tests - Which branches are tested

What coverage doesn't tell you: - Whether assertions are correct - Whether edge cases are tested - Whether tests would catch real bugs

VID Approach to Coverage:

1. **Use coverage to find untested code**

```
# Find what's not covered
pytest --cov --cov-report=html
# Review uncovered lines: are they testable? Should they be tested?
```

2. **Don't worship 100% coverage**

- Some code is hard to test and low-risk (fine to skip)
- 100% coverage with weak tests = false security
- 80% coverage with strong tests > 100% coverage with weak tests

3. **Check coverage of critical paths**
   - Payment processing: 100% coverage required + mutation testing
   - Error handling: explicitly test error paths
   - Security logic: not just covered, but tested adversarially

---

2.4.5 Strategy 5: Adversarial Test Review

For security-critical code, review tests adversarially.

Questions: - What attack vectors are NOT tested? - SQL injection attempts - XSS payloads - Path traversal - Integer overflow - Race conditions

- **Do tests verify security properties?**
  - Authentication required
  - Authorization checked
  - Input sanitized
  - Output escaped
  - Secrets not logged

Example:

Weak test:

```
def test_delete_document():
    response = client.delete('/api/documents/123')
    assert response.status_code == 200
```

Strong test:

```
def test_delete_document_authorization():
    # Create document owned by user A
    doc_id = create_document(user_a)

    # Try to delete as user B
    response = client.delete(f'/api/documents/{doc_id}',
                             auth=user_b_token)
    assert response.status_code == 403  # Forbidden

    # Verify document still exists
    assert document_exists(doc_id)

    # Delete as user A should succeed
    response = client.delete(f'/api/documents/{doc_id}',
                             auth=user_a_token)
    assert response.status_code == 200
    assert not document_exists(doc_id)
```

2.5 Part 3: AI in Test Verification

2.5.1 Where AI Can Help

AI is not just for generation—it can assist verification:

AI can: - Generate additional test cases from specifications - Suggest edge cases humans might miss - Identify untested code paths - Generate adversarial inputs - Check for common vulnerability patterns

AI cannot: - Guarantee tests are correct - Understand semantic intent - Assess business rule compliance - Make trust calibration decisions

2.5.2 Human-AI Division of Labor in Testing

AI generates: - Initial test suite from specifications - Additional test cases for edge cases - Property-based test generators - Mutation testing variants

Humans verify: - Tests match requirements (not just specifications) - Assertions are correct - Coverage is appropriate to risk level - Security properties are tested - Tests will catch real bugs

Together: - AI generates 100 test cases - Human reviews with mutation testing - Mutation testing shows gaps - AI generates additional tests to fill gaps - Human validates the additions

---

2.6 Part 4: Test Quality Checklist

Use this checklist when reviewing tests (AI-generated or human-written):

2.6.1 Before Accepting Tests

- ☐ **Provenance documented** — How were these tests created?
- ☐ **Test names describe behavior** — Not implementation details
- ☐ **Assertions are correct** — Verified, not assumed
- ☐ **Edge cases covered** — Empty, null, boundaries, max values
- ☐ **Error cases tested** — Not just happy paths
- ☐ **Coverage appropriate to risk** — Critical code has thorough tests

2.6.2 For Important Code (High Trust Level)

- ☐ **Mutation testing run** — Mutations mostly caught (>80%)
- ☐ **Property tests added** — Key invariants tested
- ☐ **Security scenarios tested** — Adversarial inputs tried
- ☐ **Integration tested** — Not just unit tests
- ☐ **Tests are readable** — Others can understand them
- ☐ **Tests are maintainable** — Won't break on refactoring

2.6.3 For Critical Code (Minimal Trust Level)

- ☐ **Independent test review** — Second person reviewed
- ☐ **Formal specifications** — Properties formally stated
- ☐ **Comprehensive mutation testing** — >90% mutation score
- ☐ **Adversarial testing** — Security team reviewed
- ☐ **Documentation** — Test strategy documented
- ☐ **Regression suite** — Tests will catch known issues

2.7 Part 5: Common Test Anti-Patterns

2.7.1 Anti-Pattern 1: "Tests That Always Pass"

Problem: Test passes regardless of code correctness.

Example:

```
def test_calculate_total():
    result = calculate_total([10, 20, 30])
    assert result  # Just checks it's not None/0
```

Why it's wrong: Test doesn't verify correctness, just that something was returned.

Fix:

```
def test_calculate_total():
    result = calculate_total([10, 20, 30])
    assert result == 60  # Actual expected value
```

2.7.2 Anti-Pattern 2: "Tautological Tests"

Problem: Test verifies code against itself.

Example:

```
def test_process_data():
    data = [1, 2, 3]
    result = process_data(data)
    expected = process_data(data)  # Calling same function!
    assert result == expected
```

Why it's wrong: If `process_data` is buggy, both calls are buggy. Test always passes.

Fix: Define expected value independently:

```
def test_process_data():
    data = [1, 2, 3]
    result = process_data(data)
    expected = [2, 4, 6]  # Manually calculated
    assert result == expected
```

2.7.3 Anti-Pattern 3: "Tests That Test Implementation"

Problem: Tests break when code is refactored (even if behavior unchanged).

Example:

```
def test_user_login():
    # Bad: Testing implementation details
    user = User("alice", "password123")
    user._hash_password()  # Testing private method
    assert user.password_hash is not None
    assert user._check_password("password123")  # Internal method
```

Why it's wrong: Refactoring internals breaks test, even if login still works.

Fix: Test public behavior:

```
def test_user_login():
    # Good: Testing behavior
    user = User.create("alice", "password123")
    assert user.login("password123") == True
    assert user.login("wrongpass") == False
```

2.7.4 Anti-Pattern 4: "Correlation Bugs"

Problem: AI generates code and tests with same logical error.

Example:

AI-generated code:

```
def days_in_month(month, year):
    if month == 2:
        return 28  # BUG: Ignores leap years
    elif month in [4, 6, 9, 11]:
        return 30
    else:
        return 31
```

AI-generated test (from same context):

```
def test_days_in_month_february():
    assert days_in_month(2, 2024) == 28  # BUG: 2024 is leap year
```

Why it's wrong: AI made same mistake in test and code. Test passes, bug ships.

Detection: Human review or property testing:

```
def test_leap_year_property():
    # 2024 is leap year (divisible by 4, not by 100)
```

```python
    assert days_in_month(2, 2024) == 29
    # 2000 is leap year (divisible by 400)
    assert days_in_month(2, 2000) == 29
    # 1900 is not leap year (divisible by 100, not 400)
    assert days_in_month(2, 1900) == 28
```

2.8 Part 6: Test Verification Workflow

2.8.1 For AI-Generated Tests (Low-Risk Code)

1. **Read tests** — Understand what they claim to test
2. **Run tests** — Verify they pass with correct code
3. **Spot-check assertions** — Pick 2-3 tests, verify assertions are right
4. **Coverage check** — Are critical paths covered?
5. **Add 1-2 edge cases** — What did AI miss?
6. **Accept if sufficient**

Time: 5-10 minutes

2.8.2 For AI-Generated Tests (Important Code)

1. **All steps from low-risk, plus:**
2. **Mutation testing** — Run mutator, check survival rate
3. **Review security tests** — Are attack vectors tested?
4. **Add property tests** — Define key invariants
5. **Independent review** — Second person reviews tests
6. **Document test strategy** — What's our testing approach?

Time: 30-60 minutes

2.8.3 For AI-Generated Tests (Critical Code)

1. **All steps from important code, plus:**
2. **Comprehensive mutation testing** — Achieve >90% mutation score
3. **Adversarial review** — Security specialist reviews
4. **Formal specification** — Write formal properties
5. **Multiple reviewers** — 2+ people review independently
6. **Regression verification** — Do tests catch known past bugs?
7. **Document thoroughly** — Test plan, rationale, gaps

Time: 2-4 hours

2.9 Part 7: Tools & Techniques Summary

Technique	Purpose	When to Use	Tools
Mutation Testing	Verify tests catch bugs	Important+ code	mutmut, Stryker, PIT
Property Testing	Test invariants	Complex logic	Hypothesis, fast-check
Coverage Analysis	Find untested code	All code	pytest-cov, Istanbul
Adversarial Testing	Find security gaps	Security-critical	Manual + fuzzing
Test Review	Human judgment	All AI-generated tests	Checklist
AI-Assisted Generation	Create more test cases	When gaps found	GPT-4, Copilot

---

2.10 Part 8: Integration with VID Principles

2.10.1 Principle 1: Intent Before Generation

For tests: - Specify what should be tested before generating tests - List expected behaviors, edge cases, error conditions - Define properties that must hold

2.10.2 Principle 2: Graduated Trust

For tests: - Low-risk code: Basic test review - Important code: Mutation testing - Critical code: Comprehensive verification with multiple techniques

2.10.3 Principle 3: Understanding Over Acceptance

For tests: - Never accept tests you don't understand - If a test is unclear, rewrite it - Understand what each test verifies and why

2.10.4 Principle 4: Provenance Awareness

For tests: - Track how tests were created - Note what verification was performed - Distinguish AI-generated from human-written

2.10.5 Principle 5: Continuous Calibration

For tests: - When bugs escape to production, check: did tests miss them? - Update test strategy based on what you learn - Add regression tests for production bugs

---

2.11 Conclusion

Tests are not automatically trustworthy just because they pass.

In the AI era, we must verify tests with the same rigor we verify code. Meta-verification strategies (mutation testing, property testing, systematic review) let us validate tests without infinite regression.

Key Takeaways:

1. **Test provenance matters** — Track how tests were created
2. **Mutation testing validates test quality** — Weak tests let mutations survive
3. **Properties complement examples** — Invariants are human insight
4. **AI can assist test verification** — But humans must arbitrate
5. **Coverage quality** — Use coverage to find gaps, not to declare victory
6. **Review tests systematically** — Use checklists appropriate to risk level

The meta-verification loop:

```
Code Generated → Tests Generated → Mutations Created
     ↑                                       ↓
Confidence Gained ← Tests Improved ← Gaps Identified
```

With this framework, tests become verification tools we can trust—not just rituals we perform.

Return to Chapter 20: The Verification Toolkit | Table of Contents | Next: Appendix A: Quick Reference

---

3 Appendix A: Quick Reference

3.1 The Five Principles

Principle	Core Question
Intent Before Generation	What do I need and how will I verify it?
Graduated Trust	What's the risk and how much verification does that warrant?
Understanding Over Acceptance	Do I understand this well enough for its importance?
Provenance Awareness	Where did this come from and what does that imply?
Continuous Calibration	Are my verification practices working based on outcomes?

3.2 Verification Depth Quick Guide

Risk Level	Characteristics	Verification
Low	Trivial impact, easily reversible, internal	Basic: Read, check intent, run automated tests
Medium	Moderate impact, standard production code	Standard: + trace logic, test edges, check integration
High	Significant impact, important systems	Deep: + adversarial testing, security review, peer review

Risk Level	Characteristics	Verification
Critical	Severe impact, security/financial/safety	Intensive: + formal verification, multiple reviewers, documentation

3.3 Intent Specification Template

```
What: [Functional requirement - what should this code do?]
Inputs: [Valid inputs and their types/constraints]
Outputs: [Expected outputs]
Edges: [Edge cases and how they should be handled]
Verify: [How I will verify correctness]
Risk: [Low/Medium/High/Critical and why]
```

---

4 Appendix B: Discussion Questions

For teams adopting VID, discuss:

1. What's our current approach to AI-generated code? What's working and what isn't?
2. How do we currently calibrate verification depth? Is it working?
3. What problems have escaped to production that verification should have caught?
4. What's our risk tolerance? How do we currently assess risk?
5. How do we build verification skills in junior engineers?
6. What would we need to change about our process to adopt VID?
7. How would we know if VID is working for us?

---

5 Appendix C: Glossary

Graduated Trust: The practice of calibrating verification depth to risk level.

Intent Specification: Articulation of what code should do and how correctness will be verified, created before code generation.

Provenance: The origin and history of code, including whether it was human-written or AI-generated and how it was verified.

Trust Level: A classification (High/Moderate/Guarded/Minimal) indicating how much verification is needed for code.

Verification Ritual: A consistent set of checks performed after code generation, scaled by risk level.

VID: Verified Intent Development — a methodology for AI-augmented software development focused on verification rather than generation.

End of Document

---

6 Appendix D: Risk Scoring Rubric

6.1 Purpose

This rubric provides a systematic method for assessing code risk and determining appropriate verification depth. It transforms the conceptual framework from Chapter 5 (Graduated Trust) into an actionable decision tool.

Use this when: - Deciding verification depth for AI-generated code - Calibrating team practices - Discussing edge cases in code review - Training new team members on risk assessment

---

6.2 The Four Risk Dimensions

6.2.1 1. Impact (I)

Question: What's the worst-case outcome if this code has a bug?

Score	Severity	Description	Examples
1	Trivial	Cosmetic issue, no functional impact	Log message typo, comment formatting, internal debug output
2	Minor	Annoyance, easy workaround exists	UI alignment issue, suboptimal sort order, redundant API call
3	Moderate	User-facing problem, no data loss	Feature doesn't work, incorrect calculation in non-critical context, slow performance
4	Serious	Data corruption, security issue, or financial impact	Payment calculation error, data loss, authentication bypass, PII exposure
5	Critical	System-wide failure, safety issue, legal liability	Complete system outage, financial fraud, regulatory violation, safety system failure

6.2.2 2. Reversibility (R)

Question: How easily can we recover if this breaks in production?

Score	Recovery Difficulty	Description	Examples
1	Instant	Fix and redeploy, no persistent effects	Stateless API endpoint, display logic, formatting function
2	Easy	Redeploy + minor cleanup (< 1 hour)	Cache invalidation needed, restart required, temporary config change
3	Moderate	Significant manual recovery (hours)	Database migration rollback, batch job rerun, customer notifications
4	Hard	Complex recovery, possible data loss (days)	Manual data correction, multi-system coordination, partial data unrecoverable
5	Impossible/Severe	Permanent data loss or irreversible actions	Deleted production data, sent incorrect payments, compliance breach recorded

6.2.3 3. Exposure (E)

Question: How many users/systems are affected?

Score	Reach	Description	Examples
1	Developer Only	Only affects development environment	Local dev tool, personal script, debug utility
2	Team Internal	Affects only your immediate team	Team dashboard, internal CI pipeline, developer tooling
3	Company Internal	Affects internal users across organization	Employee portal, internal API, admin tools

Score	Reach	Description	Examples
4	External Limited	Affects external users in controlled way	Beta features, specific customer segment, partner API
5	Public/Universal	Affects all users or public-facing systems	Public website, main product API, critical service

6.2.4 4. Compliance (C)

Question: What regulatory or legal requirements apply?

Score	Regulatory Burden	Description	Examples
0	None	No special compliance requirements	Internal tooling, general business logic
2	Basic	Industry best practices apply	Standard web security, data retention policies
4	Moderate	Specific regulations apply	GDPR (personal data), accessibility requirements, SOC2 controls
6	Significant	Strict regulatory oversight	HIPAA (healthcare), PCI-DSS (payment cards), SOX (financial reporting)
8	Critical	Severe penalties for violations	FDA (medical devices), financial trading regulations, safety-critical systems
10	Existential	Violations could end business	Critical infrastructure, life-safety systems, national security

---

6.3 Risk Score Calculation

6.3.1 Formula

```
Risk Score = (I × 3) + (R × 2) + (E × 2) + C
```

Why weighted? - **Impact (3×)**: Most important—severity of what goes wrong - **Reversibility (2×)**: Affects recovery cost and learning opportunity - **Exposure (2×)**: Determines blast radius - **Compliance (1×)**: Already scaled 0-10 for appropriate weight

Range: 0 (minimal risk) to 47 (maximum risk)

6.4 Trust Level Mapping

Risk Score	Trust Level	Verification Depth	Time Investment
0-10	**High Trust**	Basic verification	5-10 minutes
11-20	**Moderate Trust**	Standard verification	15-30 minutes
21-30	**Guarded Trust**	Thorough verification	30-60 minutes
31-47	**Minimal Trust**	Intensive verification	1-3 hours + peer review

---

6.5 Verification Requirements by Trust Level

6.5.1 High Trust (Score 0-10)

- Read and understand the code
- Verify against intent specification
- Run automated tests (linting, type checking)
- Spot-check one edge case
- Brief documentation of what code does

Skip: Deep security analysis, extensive edge case testing, peer review

6.5.2 Moderate Trust (Score 11-20)

Everything in High Trust, plus: - Systematic edge case testing (input partitioning) - Security review for user inputs - Check error handling - Review integration points - Document edge cases and assumptions

Skip: Formal verification, multiple reviewers, extensive security testing

6.5.3 Guarded Trust (Score 21-30)

Everything in Moderate Trust, plus: - Comprehensive security review (STRIDE analysis) - Adversarial testing (try to break it) - Performance/scalability consideration - Maintainability refactoring if needed - Peer review recommended - Document security considerations and trade-offs

Skip: Formal methods, external audit (unless compliance requires)

6.5.4 Minimal Trust (Score 31-47)

Everything in Guarded Trust, plus: - **Mandatory** peer review (minimum 1, prefer 2+ reviewers) - Security specialist review if available - Formal threat modeling - Comprehensive test coverage

with mutation testing - Consider formal verification for critical logic - Document thoroughly (architecture decisions, security model, failure modes) - Sign-off from tech lead or senior engineer - Incident response plan consideration

---

6.6 Worked Examples

6.6.1 Example 1: Log Message Formatter

Code: Formats timestamps in application logs

Assessment: - **Impact (I):** 1 (cosmetic, doesn't affect functionality) - **Reversibility (R):** 1 (instant fix if needed) - **Exposure (E):** 2 (internal team uses logs) - **Compliance (C):** 0 (no special requirements)

Calculation: (1×3) + (1×2) + (2×2) + 0 = **9**

Result: High Trust — 5-10 minute verification - Read code, verify format makes sense - Test with a few timestamps - Check it handles null/undefined - Ship it

---

6.6.2 Example 2: User Profile Update Endpoint

Code: API endpoint that updates user profile information

Assessment: - **Impact (I):** 3 (user-facing, could break profiles) - **Reversibility (R):** 3 (need to identify and fix affected users) - **Exposure (E):** 4 (all external users) - **Compliance (C):** 4 (GDPR applies to personal data)

Calculation: (3×3) + (3×2) + (4×2) + 4 = **27**

Result: Guarded Trust — 45-60 minute verification - Comprehensive input validation review - Check authorization (can user update this profile?) - SQL injection / XSS vulnerability testing - Verify GDPR consent and audit logging - Test edge cases (empty fields, special characters, unicode) - Get peer review - Document security model

---

6.6.3 Example 3: Payment Processing Function

Code: Calculates and processes customer refunds

Assessment: - **Impact (I):** 5 (financial loss if wrong) - **Reversibility (R):** 4 (money already sent, hard to recover) - **Exposure (E):** 4 (affects all customers who request refunds) - **Compliance (C):** 6 (PCI-DSS, financial regulations)

Calculation: (5×3) + (4×2) + (4×2) + 6 = **37**

Result: Minimal Trust — 2-3 hours + multiple reviewers - **Do NOT accept AI output without deep understanding** - Verify refund calculation logic step by step - Test idempotency (refund not processed twice) - Verify amount limits and fraud checks - Check audit trail and logging - Test with real scenarios from finance team - Security review for manipulation attempts -

Peer review from 2+ engineers - Tech lead sign-off required - Document refund policy, edge cases, rollback procedures

---

6.6.4 Example 4: Cache Implementation

Code: LRU cache with TTL support for API responses

Assessment: - **Impact (I):** 3 (incorrect caching could serve stale data) - **Reversibility (R):** 2 (clear cache and redeploy) - **Exposure (E):** 4 (affects all API users) - **Compliance (C):** 2 (standard data handling)

Calculation: (3×3) + (2×2) + (4×2) + 2 = **24**

Result: Guarded Trust — 45-60 minutes - Understand cache eviction algorithm deeply - Test TTL expiration works correctly - Test cache hit/miss scenarios - Check for memory leaks (cache grows unbounded?) - Verify thread-safety if concurrent access - Performance test with realistic load - Document cache size limits and behavior

---

6.7 Special Cases & Escalation

6.7.1 When Scores Disagree

Scenario: Low overall score but ONE dimension is very high

Example: Internal tool (E=2) with potential data loss (R=5)

Rule: Escalate to next trust level when: - Any single dimension 4, regardless of total score - Compliance score 6 - Impact = 5

Rationale: A single high-risk dimension can cause catastrophic failure.

6.7.2 When in Doubt

If you're uncertain about scoring:

1. **Score conservatively** (round up, not down)
2. **Discuss with team** — calibration improves with practice
3. **Document the edge case** for future reference
4. **Default to higher verification** — overconfidence is riskier than over-verification

Red flags that suggest higher score: - "This is just a quick fix" - "We can patch it later if there's a problem" - "No one will probably use this edge case" - "The AI seems confident"

6.8 Team Calibration Exercise

Purpose: Align team understanding of risk levels

Process:

1. **Select 10 code scenarios** from recent work
2. **Individually score** each scenario using this rubric
3. **Compare scores** — Where do you disagree?
4. **Discuss discrepancies:**
   - Why did you score Impact as 3 vs. 4?
   - What assumptions led to different Exposure scores?
5. **Agree on team norms:**
   - Are we consistently conservative or aggressive?
   - Which dimension do we struggle with most?
6. **Repeat quarterly** — Keep calibration fresh

Sample Scenarios for Calibration:

Scenario	I	R	E	C	Score	Trust Level
Sort algorithm for internal admin list	?	?	?	?	?	?
Password reset token generation	?	?	?	?	?	?
CSS styling for marketing page	?	?	?	?	?	?
Database migration to add column	?	?	?	?	?	?
API rate limiting implementation	?	?	?	?	?	?

6.9 Integration with Development Workflow

6.9.1 In Code Review

PR Template Addition:

```
## VID Risk Assessment
- Impact: [1-5]
- Reversibility: [1-5]
- Exposure: [1-5]
- Compliance: [0-10]
- **Total Risk Score:** [calculated]
- **Trust Level:** [High/Moderate/Guarded/Minimal]

**Verification performed:**
- [ ] [List specific verification steps for this trust level]
```

6.9.2 In Task Planning

Before starting work, assess: 1. What trust level will this require? 2. Do I have time/expertise for that level of verification? 3. Do I need to involve others (peer review, security team)? 4. Should this be broken into smaller, lower-risk pieces?

6.9.3 In Incident Retrospectives

After production incidents: 1. What was the original risk score? 2. Was it accurate? 3. If inaccurate, which dimension was misjudged? 4. How can we improve risk assessment?

6.10 Customization Guidelines

This rubric is a starting point. Adapt it to your context:

6.10.1 Adjust Weights

If compliance is critical for your domain (healthcare, finance):

```
Risk Score = (I × 3) + (R × 2) + (E × 2) + (C × 1.5)
```

6.10.2 Adjust Thresholds

If your team is new to VID, start conservative:

```
0-8:   High Trust
9-16:  Moderate Trust
17-25: Guarded Trust
26+:   Minimal Trust
```

6.10.3 Add Dimensions

Some teams add: - **Complexity (Co):** How hard is this code to understand? (1-5) - **Novelty (N):** How unfamiliar is this domain/technology? (1-5)

6.10.4 Document Your Modifications

If you customize: - Document why - Share with team - Revisit quarterly - Keep the core concept: objective risk → appropriate verification

---

6.11 Common Pitfalls

Scoring to justify desired verification level - "I want to ship quickly, so I'll call this Impact=2" - Fix: Have someone else review your risk assessment

Forgetting about compliance - "It's just a feature" (but touches PII = GDPR) - Fix: Always ask "What data does this touch?"

Averaging away risk - "Most dimensions are low, so overall it's low risk" (but Reversibility=5) - Fix: Use the escalation rules for high single dimensions

Analysis paralysis - Spending 20 minutes scoring a 5-line function - Fix: For obviously trivial code, intuitive assessment is fine

Never re-scoring - Initial assessment was wrong, but never revisited - Fix: Re-assess during retrospectives

---

6.12 Quick Reference Card

Print this for your desk:

```
         VID RISK SCORING QUICK REFERENCE

 IMPACT (I): 1=trivial → 5=critical
 REVERSIBILITY (R): 1=instant → 5=impossible
 EXPOSURE (E): 1=dev only → 5=public
 COMPLIANCE (C): 0=none → 10=existential

 SCORE = (I×3) + (R×2) + (E×2) + C

  0-10: High Trust      (5-10 min)
 11-20: Moderate Trust  (15-30 min)
 21-30: Guarded Trust   (30-60 min)
 31-47: Minimal Trust   (1-3 hr + review)

 ESCALATE if ANY dimension   4 or Compliance   6
 WHEN UNSURE: Round up, ask team, verify more
```

6.13 Conclusion

This rubric transforms "graduated trust" from a concept into a practice. Use it to: - Make consistent risk decisions - Communicate verification rationale - Calibrate team expectations - Improve over time through retrospectives

Remember: The rubric is a tool for judgment, not a replacement for it. When the score doesn't match your intuition, investigate why. That's where learning happens.

Return to Appendix A: Quick Reference | Table of Contents

---

7 Appendix E: Decision Trees

7.1 Purpose

These decision trees guide you through common VID scenarios. Use them when you're uncertain about the right approach.

7.2 Decision Tree 1: Should I Use AI for This Task?

```
START: Need to write code
```

```
→ Is this security-critical code?
  (authentication, authorization, crypto, payment, PII handling)

   → YES → Use AI for reference only
           Write code yourself
           Apply Minimal Trust verification
           Peer review mandatory

   → NO → Continue

          → Do I have clear intent specification?

              → YES → Safe to use AI
                      Generate code
                      Apply appropriate verification

              → NO → Stop!
                     Write intent specification first
                     Then use AI
```

Key insight: AI is a tool, not a shortcut. Intent first, generation second.

7.3 Decision Tree 2: What Trust Level Should I Apply?

```
START: Code generated (by AI or human)

→ Calculate Risk Score:
  Risk = (Impact × 3) + (Reversibility × 2) + (Exposure × 2) + Compliance

   → Score 0-10
      → HIGH TRUST
         Verification: 5-10 minutes
         - Read and understand
         - Verify against intent
         - Run automated tests
         - Spot-check one edge case

   → Score 11-20
      → MODERATE TRUST
         Verification: 15-30 minutes
         - All High Trust checks
         - Systematic edge case testing
         - Security review for inputs
         - Check error handling
         - Review integration points
```

```
    → Score 21-30
        → GUARDED TRUST
           Verification: 30-60 minutes
           - All Moderate Trust checks
           - Comprehensive security review
           - Adversarial testing
           - Performance consideration
           - Peer review recommended

    → Score 31-47
        → MINIMAL TRUST
           Verification: 1-3 hours + peer review
           - All Guarded Trust checks
           - Mandatory peer review (2+ people)
           - Security specialist if available
           - Formal threat modeling
           - Comprehensive test coverage
           - Tech lead sign-off


 → ESCALATION CHECK:

    → Is ANY dimension   4?
       OR Is Compliance   6?

        → YES → Escalate to next trust level
                Even if total score is low

        → NO → Use calculated trust level
```

Key insight: A single high-risk dimension can be catastrophic. Escalate accordingly.

7.4 Decision Tree 3: How Do I Verify AI-Generated Tests?

```
START: AI generated tests

 → Step 1: Check Test Provenance

    → How were tests generated?

        → From specification
            → Moderate confidence
               Apply standard verification

        → From code
            → Low confidence
               Tests may duplicate code bugs
```

```
            Apply intensive verification

        → From examples
            → Medium confidence
               May miss edge cases
               Apply thorough verification

→ Step 2: Review Test Quality

    → Read test names and assertions
    → Do they test behavior or implementation?

        → Implementation → RED FLAG
          Tests are brittle
          Rewrite to test behavior

        → Behavior → GOOD
           Continue verification

    → Are assertions correct?

        → Uncertain → Manually verify expected values
        → Confident → Continue

→ Step 3: Check Coverage

    → What's NOT tested?
      - Edge cases (empty, null, max, min)
      - Error conditions
      - Security scenarios
      - Integration points

    → Add missing tests

→ Step 4: Meta-Verification

    → For Important+ code:
      Run mutation testing

        → Mutation score < 80%
            → Tests are weak
               Add stronger tests

        → Mutation score   80%
            → Tests are adequate
               Proceed

    → For Critical code:
```

```
        Add property-based tests
        Independent test review
        Mutation score   90%

→ ACCEPT or REJECT

    → ACCEPT if:
       - Tests are behavioral
       - Assertions are correct
       - Coverage is appropriate
       - Mutation score is adequate

    → REJECT if:
       - Tests test implementation
       - Assertions are wrong/unknown
       - Major gaps in coverage
       - Low mutation score
```

Key insight: Tests need verification just like production code.

7.5 Decision Tree 4: Should I Accept This AI-Generated Code?

```
START: AI generated code

→ Question 1: Do I understand what this code does?

    → NO → STOP

            → Can I learn enough to understand?

                → YES → Spend time learning
                        Then re-evaluate

                → NO → Options:
                        A) Simplify the code
                        B) Get help from expert
                        C) Reject and write manually

            → DON'T ACCEPT CODE YOU DON'T UNDERSTAND

    → YES → Continue to Question 2

→ Question 2: Does it match my intent specification?

    → NO → Reject or modify
           AI solved a different problem
```

```
    → YES → Continue to Question 3

→ Question 3: Did verification pass?

    → NO → What failed?

            → Tests failed → Fix or regenerate
            → Edge cases wrong → Fix or regenerate
            → Security issue → Fix or regenerate
            → Can't be fixed → Reject, write manually

    → YES → Continue to Question 4

→ Question 4: Is verification depth appropriate for risk?

    → NO → Do I need deeper verification?

            → YES → Perform additional verification
                    Then re-evaluate

            → Verification was too shallow
               Return to Decision Tree 2

    → YES → Continue to Question 5

→ Question 5: Can I maintain this code in 6 months?

    → NO → Issues:
           - Poor naming?
           - Too complex?
           - Unclear logic?

            → Refactor for maintainability
               Then re-evaluate

    → YES → Continue to Final Check

→ FINAL CHECK: All conditions met?

    → I understand the code
    → It matches my intent
    → It passed appropriate verification
    → Risk-appropriate verification depth
    → I can maintain it

    → ALL YES → ACCEPT
                Document provenance
                Commit with verification notes
```

```
    → ANY NO → REJECT or FIX
               Don't compromise on quality
```

Key insight: Acceptance is not a yes/no decision. It's a series of quality gates.

7.6 Decision Tree 5: What Do I Do When Production Breaks?

```
START: Production incident

 → Question 1: Is this code AI-generated?

    → Don't know → PROBLEM
                   Provenance tracking failure
                   (Fix this for future)

    → NO (human-written) → Standard incident response

    → YES (AI-generated) → Continue to Question 2

 → Question 2: What verification was performed?

    → Check commit message / PR notes
      What trust level?
      What checks were done?

    → Unknown → PROBLEM
                Verification documentation failure
                (Fix this for future)

    → Known → Continue to Question 3

 → Question 3: Should verification have caught this?

    → YES → Verification failure

            → Root cause analysis:
               - Was risk score too low?
               - Was verification incomplete?
               - Was verification technique wrong?

                → Update verification practices
                  Prevent similar failures

    → NO → Acceptable miss
               (edge case beyond reasonable verification)
```

```
                → Add regression test
                  Update risk calibration if needed

→ Question 4: Can the original developer fix it?

    → YES → Have them fix it
            (they understand the code)

    → NO → Understanding gap

              → Code was not understood when accepted
                  → PROBLEM: Violated Understanding principle
                     (Fix this for future)

              → Developer is unavailable
                  → Hope someone can figure it out
                     (This is why understanding matters)

→ RESOLUTION PATH:

    → Immediate: Fix the bug

    → Short-term: Add regression test
                  Update documentation

    → Long-term: Calibrate verification practices
                 If verification should have caught it:
                 - Update risk scoring
                 - Update verification rituals
                 - Share learning with team
```

Key insight: Production incidents are learning opportunities. Update practices accordingly.

7.7 Decision Tree 6: How Do I Onboard a New Developer to VID?

```
START: New developer joins team

→ Week 1: Foundation

    → Assign reading:
      - VID One-Page Overview
      - Chapter 1: The Inversion
      - Chapter 4: Intent Before Generation

    → 1-hour 1-on-1 discussion:
      - Why does VID matter?
      - How does it work?
```

```
      - What are our team norms?

    → Shadow experienced developer:
       Watch full VID cycle:
       - Intent specification
       - Generation
       - Verification
       - Documentation

→ Week 2: Supervised Practice

    → Pair program with mentor
       New dev drives, mentor guides
       Focus on intent specification

    → Low-risk task
       New dev applies VID
       Mentor reviews every step

    → Debrief:
       - What went well?
       - What was confusing?
       - Clarify principles

→ Week 3: Guided Independence

    → Medium-risk task
       New dev works independently
       Mentor reviews verification notes

    → Join team calibration session
       Participate in risk scoring
       Learn team norms

    → Check understanding:
       Can they explain:
       - The 5 principles?
       - How to score risk?
       - When to escalate verification?

→ Week 4: Independent with Oversight

    → Full VID cycle independently
       Mentor spot-checks

    → Can take on any risk level
       With appropriate oversight
```

```
    → Ready to mentor others
      After 3 months of practice
```

Key insight: VID is learned by doing, not just reading. Pair learning accelerates adoption.

7.8 Decision Tree 7: When Should I Update Our VID Practices?

```
START: Considering practice changes

 → Trigger: Why are we considering changes?

    → Problems escaping to production

       → Root cause analysis:

          → Risk scores too low?
             → Recalibrate risk scoring
                Update team norms

          → Verification incomplete?
             → Update verification rituals
                Add missing checks

          → Verification technique ineffective?
              → Adopt better techniques
                 Train team

    → Verification too time-consuming

       → Analysis:

          → Verifying low-risk code too deeply?
             → Recalibrate downward
                Trust level adjustments

          → Inefficient verification workflow?
             → Streamline process
                Remove redundant steps

          → Team inexperienced?
              → More training
                 Practice builds speed

    → New technology or AI capability

       → Evaluate impact:
```

```
            → AI generates better code?
                → Don't reduce verification yet
                   Wait for data
                   Recalibrate after 3 months

            → AI can assist verification?
                → Experiment carefully
                   Human still arbitrates

            → New code patterns?
                → Assess new risks
                   Update risk framework

    → Team growth or composition change

        → Ensure consistency:
           - Onboard new members
           - Recalibrate as team
           - Document team norms

→ Decision: Should we change practices?

    → YES → Process:

            → Propose changes
            → Discuss with team
            → Try for 2 weeks
            → Measure impact
            → Keep, modify, or revert

    → NO → Continue current practices
           Next review in 1 month

→ WHEN TO REVIEW:

    → Monthly retrospectives (lightweight)
    → Quarterly deep reviews (comprehensive)
    → After major incidents (root cause)
    → When team composition changes (recalibrate)
```

Key insight: VID is continuous calibration. Adjust based on evidence, not opinion.

7.9 How to Use These Decision Trees

7.9.1 In Daily Work

Before generating code: Use Decision Tree 1 (Should I use AI?)

After generating code: Use Decision Tree 2 (What trust level?) and Tree 4 (Should I accept?)

When verifying tests: Use Decision Tree 3 (How to verify tests?)

7.9.2 In Incident Response

After production issues: Use Decision Tree 5 (What went wrong?)

7.9.3 In Team Development

Onboarding: Use Decision Tree 6 (How to onboard?)

Process improvement: Use Decision Tree 7 (When to update?)

7.9.4 Customization

These decision trees are starting points. Your team should: - Adapt them to your context - Add team-specific branches - Document your variations - Share improvements

---

7.10 Quick Reference

Print these and keep at your desk:

Pre-Generation: Intent first, always **Post-Generation:** Verify appropriately **In Incidents:** Learn and calibrate **In Onboarding:** Pair and practice **In Retrospectives:** Measure and adjust

---

Return to Table of Contents

---

8 Appendix F: VID Checklists

8.1 Purpose

These checklists ensure you don't skip critical verification steps. Print and use daily until they become second nature.

8.2 Checklist 1: Pre-Generation (Intent Specification)

Use before every AI code generation

Functional Requirements: - [ ] I can describe what this code should do in 2-3 sentences - [ ] I have identified the inputs and their types - [ ] I have identified the expected outputs - [ ] I have listed the main use cases (at least 2-3 examples)

Boundary Cases: - [ ] I know what happens with empty/null inputs - [ ] I know what happens at min/max boundaries - [ ] I have identified at least 3 edge cases - [ ] I know how errors should be handled

Success Criteria: - [ ] I have written down how I'll verify this is correct - [ ] I have specific test cases in mind (at least 3) - [ ] I know what properties must always hold - [ ] I can recognize correct vs. incorrect behavior

Risk Assessment: - [ ] I have scored Impact (1-5) - [ ] I have scored Reversibility (1-5) - [ ] I have scored Exposure (1-5) - [ ] I have scored Compliance (0-10) - [ ] I have calculated total risk score - [ ] I know what trust level to apply

Ready to Generate: - [ ] I have documented my intent (at least informally) - [ ] I know what verification I'll perform - [ ] I have allocated appropriate time for verification - [ ] I understand what I'm trying to build

If ANY checkbox is unchecked: STOP. Complete intent specification before generating.

8.3 Checklist 2: Post-Generation Verification (All Trust Levels)

Use immediately after AI generates code

8.3.1 Basic Verification (Required for ALL code)

Understanding: - [ ] I have READ the entire code (not skimmed) - [ ] I can explain what this code does without looking at it - [ ] I understand the approach it takes - [ ] I could explain this to a colleague

Intent Match: - [ ] The code does what I specified - [ ] It handles the inputs I identified - [ ] It produces the outputs I expected - [ ] It solves my actual problem (not a similar one)

Automated Checks: - [ ] Code compiles/parses without errors - [ ] Linting passes - [ ] Type checking passes (if applicable) - [ ] Basic tests pass

Spot Check: - [ ] I tested with at least 1 normal input - [ ] I tested with at least 1 edge case - [ ] Results match my expectations

---

8.4 Checklist 3: Moderate Trust Verification

Use for Risk Score 11-20 (in addition to Basic Verification)

Systematic Testing: - [ ] I have identified input categories (at least 4-5) - [ ] I have tested at least 1 representative from each category - [ ] I have tested boundary values - [ ] I have tested error conditions

Security Review (Basic): - [ ] I have identified all inputs - [ ] User-controlled inputs are validated - [ ] No obvious injection vulnerabilities (SQL, command, XSS) - [ ] Sensitive data is not logged

Error Handling: - [ ] Error cases are handled (not just ignored) - [ ] Error messages don't leak sensitive information - [ ] The code fails safely - [ ] Resources are cleaned up on errors

Integration: - [ ] I understand how this integrates with existing code - [ ] Interfaces/contracts are honored - [ ] Dependencies are appropriate - [ ] Side effects are documented

Documentation: - [ ] Edge cases are documented - [ ] Assumptions are documented - [ ] Purpose is clear from code/comments - [ ] Future maintainers can understand this

---

8.5 Checklist 4: Guarded Trust Verification

Use for Risk Score 21-30 (in addition to Moderate Trust Verification)

Comprehensive Security Review: - [ ] I have performed STRIDE threat analysis - [ ] I have tested adversarial inputs - [ ] Authentication is required (if applicable) - [ ] Authorization is checked (if applicable) - [ ] Sensitive data handling is appropriate - [ ] Rate limiting is considered (if applicable)

Adversarial Testing: - [ ] I tried to break the code - [ ] I tested with malicious inputs - [ ] I tested extreme values - [ ] I tested concurrent access (if applicable) - [ ] I tested failure modes

Performance: - [ ] I considered performance implications - [ ] I tested with realistic data volumes - [ ] No obvious performance bottlenecks - [ ] Resource usage is reasonable

Maintainability: - [ ] Code is readable (not overly clever) - [ ] Complexity is justified - [ ] Naming is clear - [ ] Logic is straightforward - [ ] I would be comfortable modifying this in 6 months

Peer Review: - [ ] Another developer has reviewed this code - [ ] Review feedback has been addressed - [ ] Reviewer understands and approves

---

8.6 Checklist 5: Minimal Trust Verification

Use for Risk Score 31-47 (in addition to Guarded Trust Verification)

Mandatory Reviews: - [ ] At least 2 developers have reviewed this code - [ ] Security specialist has reviewed (if available) - [ ] Tech lead has approved - [ ] All review feedback addressed

Comprehensive Testing: - [ ] Test coverage is comprehensive (>90%) - [ ] Mutation testing performed (score >90%) - [ ] Property-based tests added (where applicable) - [ ] Integration tests pass - [ ] Performance tests pass

Formal Analysis: - [ ] Threat model documented - [ ] Security properties formally stated - [ ] Critical invariants identified and tested - [ ] Failure modes analyzed

Documentation: - [ ] Architecture decisions documented - [ ] Security model documented - [ ] Edge cases and limitations documented - [ ] Incident response considerations documented - [ ] Provenance thoroughly documented

Sign-offs: - [ ] Tech lead sign-off obtained - [ ] Security team sign-off (if applicable) - [ ] Product/business sign-off (for critical business logic)

8.7 Checklist 6: Test Verification

Use when AI generates tests or when reviewing test quality

Test Quality: - [ ] Test names describe behavior (not implementation) - [ ] Assertions check correct values (not just "something returned") - [ ] Tests test behavior (not implementation details) - [ ] Tests are readable and understandable

Coverage: - [ ] Normal/happy paths are tested - [ ] Edge cases are tested (empty, null, boundaries) - [ ] Error conditions are tested - [ ] Integration points are tested (if applicable)

Test Correctness: - [ ] I have verified expected values are actually correct - [ ] Tests would fail if code was wrong - [ ] Tests are not tautological (not testing code against itself) - [ ] No correlated bugs (test and code don't share same logic error)

For Important+ Code: - [ ] Mutation testing performed - [ ] Mutation score >80% (important) or >90% (critical) - [ ] Property-based tests added for key invariants

Provenance: - [ ] Test provenance documented (how tests were created) - [ ] Verification performed is documented - [ ] Tests are marked as AI-generated (if applicable)

8.8 Checklist 7: Code Acceptance

Use before committing code (final gate)

Quality Gates: - [ ] All appropriate verification checklists completed - [ ] Verification depth matches risk level - [ ] All verification passed (or issues resolved) - [ ] No known bugs or issues

Understanding: - [ ] I understand this code at appropriate depth for risk level - [ ] I could debug this if it breaks - [ ] I could modify this if requirements change - [ ] I could explain this to a colleague

Documentation: - [ ] Commit message describes what was done - [ ] Commit message indicates AI assistance (if applicable) - [ ] Commit message notes verification performed - [ ] PR description includes risk score and trust level

Provenance: - [ ] Code provenance is documented - [ ] Verification provenance is documented - [ ] Future maintainers will know where this came from

Team Standards: - [ ] Code follows team conventions - [ ] No shortcuts taken under time pressure - [ ] I'm confident in this code - [ ] I'm willing to be on-call for this code

Final Check: - [ ] I would accept responsibility for this code - [ ] I'm not rushing to ship without proper verification - [ ] The quality is appropriate for the risk level

If ANY critical checkbox is unchecked: DON'T COMMIT. Complete required verification.

8.9 Checklist 8: Incident Response

Use after production incidents involving AI-generated code

Immediate Response: - [ ] Incident is contained - [ ] Impact is assessed - [ ] Customers are notified (if needed) - [ ] Fix is deployed (or incident is mitigated)

Investigation: - [ ] Root cause identified - [ ] Was this code AI-generated? - [ ] What verification was performed? - [ ] What verification should have been performed? - [ ] Should verification have caught this?

Documentation: - [ ] Incident is documented - [ ] Root cause is documented - [ ] Timeline is documented - [ ] Lessons learned are captured

Process Improvement: - [ ] Was risk score accurate? - [ ] Was trust level appropriate? - [ ] Was verification sufficient? - [ ] What verification technique would have caught this? - [ ] Do we need to update our practices?

Follow-up: - [ ] Regression test added - [ ] Similar code patterns identified and reviewed - [ ] Team notified of lessons learned - [ ] Verification practices updated (if needed) - [ ] Risk calibration adjusted (if needed)

8.10 Checklist 9: Team Calibration Session

Use monthly to align team on risk assessment and verification

Preparation: - [ ] Selected 5-10 recent PRs for review - [ ] PRs represent mix of risk levels - [ ] Everyone has reviewed PRs beforehand

During Session: - [ ] Each person independently scores risk for each PR - [ ] Compare scores - [ ] Discuss differences (why did you score higher/lower?) - [ ] Agree on calibrated scores - [ ] Document any changes to team norms

Review Outcomes: - [ ] What verification caught in past month - [ ] What escaped to production in past month - [ ] Were trust levels appropriate? - [ ] Were verification techniques effective?

Action Items: - [ ] Update risk scoring guidelines (if needed) - [ ] Update verification rituals (if needed) - [ ] Schedule training (if knowledge gaps identified) - [ ] Document decisions

---

8.11 Checklist 10: Onboarding New Developer to VID

Use when new developer joins team

Week 1: - [ ] Read VID One-Page Overview - [ ] Read Chapter 1: The Inversion - [ ] Read Chapter 4: Intent Before Generation - [ ] Shadow experienced developer on full VID cycle - [ ] 1-hour 1-on-1 discussion about VID

Week 2: - [ ] Pair program with mentor (new dev drives) - [ ] Complete low-risk task using VID - [ ] Mentor reviews every step - [ ] Debrief session

Week 3: - [ ] Complete medium-risk task independently - [ ] Mentor reviews verification notes - [ ] Attend team calibration session - [ ] Can explain 5 principles and risk scoring

Week 4: - [ ] Complete full VID cycle independently - [ ] Appropriate oversight based on risk level - [ ] Mentor spot-checks work - [ ] Debrief on month 1 experience

Competency Verification: - [ ] Can articulate the 5 VID principles - [ ] Can score risk using the rubric - [ ] Can apply appropriate verification for each trust level - [ ] Knows when to ask for help

- [ ] Ready to work independently (with normal oversight)

8.12 Checklist 11: PR Review (for Reviewers)

Use when reviewing pull requests containing AI-generated code

Before Reviewing: - [ ] PR includes risk score - [ ] PR includes trust level - [ ] PR notes verification performed - [ ] PR indicates AI usage (if applicable)

Code Review: - [ ] I understand what the code does - [ ] Code matches PR description - [ ] Changes are appropriate for the task - [ ] No obvious bugs or issues - [ ] Code is maintainable

Verification Review: - [ ] Was appropriate trust level applied? - [ ] Was verification depth sufficient? - [ ] Are verification notes convincing? - [ ] Would I be comfortable with this code in production?

Risk Assessment Review: - [ ] Do I agree with the risk score? - [ ] If no, what dimension did they miss? - [ ] Should trust level be escalated?

Testing Review: - [ ] Tests are present and appropriate - [ ] Tests cover important cases - [ ] Tests are not just testing implementation - [ ] Test quality is appropriate for risk level

Decision: - [ ] Approve (quality is appropriate for risk) - [ ] Request changes (issues need addressing) - [ ] Escalate (higher trust level needed)

8.13 How to Use These Checklists

8.13.1 Daily Use

1. **Print checklists** 1-7 and keep at your desk
2. **Use checklist 1** before every generation
3. **Use checklist 2** after every generation
4. **Use checklists 3-5** based on risk score
5. **Use checklist 7** before every commit

8.13.2 Periodic Use

- **Checklist 8:** After incidents
- **Checklist 9:** Monthly calibration
- **Checklist 10:** When onboarding
- **Checklist 11:** During PR reviews

8.13.3 Adaptation

These checklists are starting points. Your team should: - Adapt them to your context - Add team-specific items - Remove items that don't apply - Update based on lessons learned - Share improvements

8.13.4 Building Habits

First 2 weeks: Use checklists explicitly (check every box)

Weeks 3-4: Use checklists as reference (verify you didn't miss anything)

Month 2+: Internalized habits (checklists as spot-check)

Goal: Build habits so verification is automatic, not checklist-driven.

8.14 Printable Quick Reference Cards

8.14.1 Card 1: Pre-Generation

```
BEFORE AI GENERATION: INTENT FIRST

  What should this do? (2-3 sentences)
  What are the inputs/outputs?
  What are 3+ edge cases?
  How will I verify it's correct?
  What's the risk score?
  What trust level will I apply?

IF ANY UNCHECKED → STOP
Complete intent specification first
```

8.14.2 Card 2: Post-Generation

```
AFTER AI GENERATION: VERIFY

ALWAYS:
  Read and understand code
  Verify against intent
  Run automated checks
  Test edge cases

THEN by risk score:
 0-10: Basic verification (10 min)
11-20: + Security & integration (30m)
21-30: + Adversarial testing (60m)
31-47: + Peer review & sign-off (2h+)

ANY dimension  4 → Escalate trust
```

8.14.3 Card 3: Before Commit

```
BEFORE COMMIT: FINAL GATE

  Appropriate verification completed
  I understand this code
  I could debug/modify this
  Provenance documented
  Commit message includes:
   - What was done
   - AI assistance noted
   - Verification noted
   - Risk score/trust level

I accept responsibility for this code
```

8.15 Digital Checklist Templates

8.15.1 PR Template Addition

```
## VID Verification

**Risk Assessment:**
- Impact: [1-5]
- Reversibility: [1-5]
- Exposure: [1-5]
- Compliance: [0-10]
- **Total Risk Score:** [calculated]
- **Trust Level:** [High/Moderate/Guarded/Minimal]

**AI Usage:**
- [ ] AI-generated code (specify tool/model)
- [ ] Human-written code
- [ ] Mixed (specify which parts)

**Verification Performed:**
- [ ] Intent specification documented
- [ ] Basic verification completed
- [ ] [Trust-level-specific checks completed]
- [ ] Tests passing
- [ ] Peer review (if required)

**Verification Notes:**
[What did you verify? What did you find? Any concerns?]
```

8.15.2 Commit Message Template

```
[Brief summary of change]

AI-assisted: [Yes/No, tool used]
Risk Score: [0-47] (Trust Level: [High/Moderate/Guarded/Minimal])

Verification:
- [List key verification steps performed]
- [Note any issues found and resolved]

[Detailed description if needed]
```

Print these checklists. Use them daily. Build verification habits.

Return to Table of Contents