

AI AGENT TO SHIPPING

SOFTWARE ENGINEERING FOR SHIPPING VIBE-CODED PRODUCTS

A Solo Builder's System for Planning, Architecture, TDD, Review, Deployment, and Operations with AI Agents

CHANGE CONTRACT

TEST EVIDENCE

FAIL-CLOSED

RELEASE GATES

Contents

Opening	7
The Promise of This Book	7
Who This Book Is For	7
Notes on Examples, Counts, and Translation	8
Copyright and Permitted Use	8
Part I. Turning Vibes into Engineering	9
1. Writing Code Quickly Is Not the Same as Shipping a Product Quickly	10
“It Works” Has Several Layers	10
How Prototype Debt Appears	11
The Goal Is Less Rework, Not Slower Work	12
Exercise 1: Separate the Layers of Completion	12
2. AI Is Neither a Co-Founder nor an Autopilot	13
Four Modes of Work	13
1. Exploration Mode	13
2. Diagnosis Mode	13
3. Change Mode	13
4. Operations Mode	14
Authority and Verification Are Not Opposites	14
Three Illusions AI Commonly Creates	14
Rules for Promoting a Claim into a Fact	15
Exercise 2: Write a Role Declaration	15
3. Write the Product Contract Before the First Prompt	16
Eight Elements of a One-Page Product Contract	16
1. Customer	16
2. Current Behavior	16
3. Promised Change	16
4. Critical Path	16
5. Revenue Event	17
6. Non-Goals	17
7. Failure Default	17
8. Release Evidence	17

How Vincent's Product Contract Chose Its Structure	17
Write Requirements as Observable Differences, Not Solutions	18
Put ROI in the Product Contract	18
Product Contract Template	19
Part I Checkpoint	19
Part II. Make the Repository a System Where AI Can Work Safely	20
4. Let the Repository Remember, Not the Conversation	21
Three Layers of Useful Repository Memory	21
1. Root Operating Rules	21
2. Domain Contracts	21
3. Change History and Evidence	21
Operating Rules Should Be Short and Concrete	22
The LVRS Example: Document the Contract Behind One Command	22
Failure Caused by Documentation Drift	23
Repository Memory Checklist	23
5. Give AI a Change Contract, Not a Task Name	24
Basic Structure of a Change Contract	24
File Count Does Not Define the Size of a Change	24
Assemble the Evidence Packet First	24
Turn a Weak Prompt into a Change Contract	25
Change One Kind of Risk at a Time	25
Ask the AI to Falsify Its Work Before Completion	26
6. Evaluate External Dependencies First and Implement Only the Unique Domain	27
Five Questions Before Adoption	27
1. Maintenance State	27
2. License	27
3. Dependency Size	27
4. Interface Stability	27
5. Failure Meaning	28
When Direct Implementation Is Appropriate	28
Dependency Review Template	28
7. Do Not Predict the Future; Contain Change Cost with Boundaries	29
Find Boundaries by Reasons for Modification	29
Boundaries among Vincent, LVRS, and iiPaintEngine	29

Keep Interfaces Small	30
Signals That Justify an Abstraction	30
Dependency Direction	31
Part II Checkpoint	31
Part III. Promote Generated Changes to Product Quality	32
8. TDD Narrows the Search Space Instead of Slowing AI Down	33
Red-Green-Refactor with AI	33
Red: Own the Failure First	33
Green: Pass with the Smallest Change	33
Refactor: Reduce Modification Cost while Preserving Behavior	33
Think in a Risk Ladder, Not a Universal Test Pyramid	33
Why Contract Tests Are Strong in the AI Era	34
Testing Nondeterministic AI Features	34
Do Not Let the Test Copy the Implementation	34
The Temporary Exception for an Untested UI Prototype	35
9. The Verification Ladder: From Code to Customer Result	36
Default Verification Order	36
1. Inspect Changed Files	36
2. Run Relevant Unit and Contract Tests	36
3. Run Full Static Verification	36
4. Run the Full Test Suite	36
5. Build the Deployment Form	36
6. Inspect the Artifact	36
7. Inspect the Real Surface	36
Separate “Build Green” from “Production Live”	36
Do Not Complete a UI from DOM or QML Source Alone	37
Documents and eBooks Must Be Rendered Too	37
Format for Reporting Verification	38
10. Design Security and Cost Around the Failure Default	39
Separate Public Coordinates from Secrets	39
Distinguish Fail-Closed Behavior from Graceful Fallback	39
Put Cost Ceilings in Code, Not in a Prompt	40
Assume Raw Bodies and Duplicate Webhook Delivery	40
Data Minimization Is Structural Defense	41

11. Debugging Reduces the Space of Facts Before It Changes Code	42
Buy Reproducibility First	42
Divide Observation by Layer	42
Good Logging Is Not More Logging	42
Do Not Revert the Most Recent Change Automatically	43
Separate a Visible Performance Symptom from Its Cause	43
AI Debugging Prompt	43
12. Release Engineering Is the Product's Last Feature	44
A Reproducible Build Boundary	44
Promote Packages Atomically	44
Signing, Notarization, and Store Review Are Separate Gates	44
Separate Existing-Customer Access from the New-Sale Gate	45
Release Checklist	46
Part III Checkpoint	46
Part IV. Engineering Patterns Repeated in Real Products	47
13. Vincent and iiPaintEngine: Separate Fast UI from Slow Physical Models	48
Three Boundaries: Application, Engine, and Framework	48
Case 1: Brush Hardness and Anti-Aliasing	49
Case 2: Pan Jitter Was Not a Document-Coordinate Problem	49
Case 3: Geometry Outside the Canvas and Visual Clipping	49
Case 4: Build and Installation Are Test Subjects	50
System to Take from These Cases	50
14. LVRS: Supporting Several Platforms Does Not Mean Hiding Their Boundaries	51
Keep the Upper Interface Small and the Failure Diagnosis Specific	51
Mistaking an Installation Prefix for a Source Root	51
Distinguish Skip from Fail When a Platform Is Missing	52
Separate State from Placement in a QML Framework	52
Lessons from the Framework	52
15. WeUs/ AISNS: Operating Generative AI as a Product Feature	54
Policies the Model Must Not Own	54
A Queue Turns "AI Replies" into Product Behavior	55
Never Guess What the Model Did Not See	55
A Persona Is a Data Contract, Not a Free-Form Prompt	55
What One Codebase for Web, iOS, and Android Really Means	56

Lessons from a Generative Product	56
16. iisacc.com: Build a Transaction System, Not a Payment Page	57
A Client Completion Event Is Not Payment Authority	57
Separate Checkout Readiness from Delivery Readiness	57
The File Is Part of the Transaction	58
Email Is a Secondary Delivery Route	58
Separate Analytics from Payment	58
Lessons from an Owned Commerce System	58
Part IV Checkpoint	59
Part V. A Repeatable AI Engineering Operating System for a Solo Builder	60
17. An Agent Playbook for Delegating One Task End to End	61
Step 0. Classify the Work	61
Step 1. Freeze the Starting State	61
Step 2. Create an Evidence Packet and Change Contract	61
Step 3. Find External Dependencies and Existing Interfaces	62
Step 4. Create Failure Evidence	62
Step 5. Repeat Minimum Implementation and Narrow Verification	62
Step 6. Climb the Full Verification Ladder	62
Step 7. Update Documentation and Operating Contracts	62
Step 8. Report by Level of Fact	62
Base Prompt for an Engineering Agent	63
18. A Seven-Day Product Promotion Sprint	64
Day 1. Put the Problem and Revenue Event on One Page	64
Day 2. Establish Repository Memory and a Build Baseline	64
Day 3. Create a Failing Test for the Critical Path	65
Day 4. Complete the Minimum Implementation	65
Day 5. Package the Real Form	65
Day 6. Connect Sales, Deployment, and Recovery	66
Day 7. Verify the Real Surface and Launch Documentation	66
Exit Conditions after Seven Days	66
19. Connect Monetization Before Building the Feature, Not After	67
Sell an Outcome, Not a Feature Count	67
Product Ladder for This Book	67
Kindle Book - US\$12.99	67

Book - US\$24.99 / KRW 29,000	67
Operator Bundle - US\$39 / KRW 49,000	68
Team 5 - US\$79 / KRW 99,000	68
Give Each Channel a Role	68
Clarify the Offer Before Lowering the Price	68
Pre-Launch Truthfulness Review	69
A Content-Reuse System	69
20. Manage AI Entropy to Keep the Product Maintainable	70
Signals of Entropy	70
Weekly Maintenance Loop	70
Definition of Done for a Completed Feature	71
What Remains When the Tools Change	71
Part V Checkpoint	71
Appendix A. Thirty Questions to Use Immediately	72
Problem and Scope	72
Repository and Structure	72
Tests and Verification	72
Security and Cost	72
Deployment and Sales	73
Operations and Maintenance	73
Appendix B. Terms	74
Change Contract	74
Fact Promotion	74
Fail-Closed	74
Graceful Fallback	74
Idempotency	74
Entitlement	74
Release Registry	74
Source Aligned	75
User Surface Confirmed	75
AI Entropy	75
Appendix C. Reader Action Plan	76

Opening

The Promise of This Book

This book does not sell a “magic prompt that builds an app from one sentence.” It does not recommend deploying code that AI produced quickly but that its operator does not understand. Instead, it explains how to use AI as a change proposer, researcher, implementer, and verification assistant while a human retains authority over product direction and quality.

The scope does not end when an idea appears on a screen. It continues through writing requirements as changeable contracts, turning the repository into the agent’s working memory, evaluating external dependencies, fixing small changes with tests, inspecting real builds and user surfaces, closing payment, security, and deployment failures safely, and reducing operating cost after launch.

Who This Book Is For

- People who can prototype with AI coding tools but see their existing codebase deteriorate with every change
- Solo business owners who must plan, build, design, deploy, and sell a product themselves
- Developers who want to delegate work safely to AI agents inside an existing repository
- Teams that need a product-level definition of done covering tests, deployment, security, and payments
- Builders who want an operating system that survives changes in models and tools

You can understand the central principles without deep programming knowledge. This book does not promise that “anyone can succeed without learning development.” In the age of vibe coding, development skill includes the ability to investigate what you do not know, divide risky changes into small units, and refuse to ship unverified results.

Notes on Examples, Counts, and Translation

The examples come from the July 31, 2026 source snapshots of Vincent, iiPaintEngine, LVRS, WeUs/ AISNS, iisacc.com, and Time Scopes, all developed by iisacc. Each repository uses a different language and testing system, so a test declaration count in one project is not treated as equivalent to a test file count in another. Only structures and decisions that can be disclosed are used. API keys, authentication material, private prompts, and customer data are excluded.

This English edition was produced through an AI-generated translation workflow and reviewed and edited by iisacc. The Korean first edition remains the source work. Technical names, commands, product facts, and rights statements were checked against that source rather than inferred from the translation.

Copyright and Permitted Use

Copyright 2026 iisacc. All rights reserved.

One purchaser may use this book and its individual templates for personal learning and for internal work within the purchaser's organization. The text, images, and templates may not be copied for sale, redistributed through a public repository, shared drive, or community, or supplied as source material for most of another course, book, or dataset. A team license permits only the number of seats purchased.

Part I. Turning Vibes into Engineering

1. Writing Code Quickly Is Not the Same as Shipping a Product Quickly

The first experience of vibe coding can feel like the disappearance of a bottleneck. Tasks that once required hours—creating files, reading framework documentation, looking up APIs, and writing repetitive code—can now be described in natural language and turned into dozens of files in minutes. A screen appears. The buttons move. At that moment it is easy to confuse code-generation speed with product-development speed.

Product speed is not typing speed. It is the total time required to choose the right problem, understand the scope of a change, preserve existing behavior, deliver the result somewhere a user can actually reach, and recover when something goes wrong. Code created in ten minutes does not make a fast product if it breaks only in production and takes two days to diagnose. An implementation that takes two hours can still be fast when its test and deployment contracts are clear enough to ship it once.

Remember the difference with this expression:

Product delivery velocity = generation speed × probability of the right direction × verification confidence × deployment success rate

Each factor is between zero and one. Even if generation becomes ten times faster, actual delivery will not rise as expected when the other three factors fall by half. AI makes plausible code in volume, so it amplifies the cost of a wrong direction as well as the speed of a correct one.

It Works Has Several Layers

When an AI says that a task is complete, decompose what was actually completed.

- Source aligned: the code structurally matches the requirement.
- Static verification: type checking, linting, and compilation pass.
- Dynamic verification: the relevant tests actually pass.
- Packaging verified: the bundle, installer, or web build that the user receives is produced.
- Deployment verified: the production environment serves the new artifact.

- User surface verified: the result appears through the real browser, app, payment, or download path.
- Operations verified: logging, retry, cancellation, and recovery behave as expected.

Once these layers are separated, “the button works locally” and “a customer can pay and receive the file safely” are clearly different states. Vibe coding becomes dangerous when an AI can cover the gaps between those states with smooth prose. Engineering names each gap as a state and requires evidence for every state.

How Prototype Debt Appears

Prototypes are not bad. The problem begins when a prototype is promoted to an operating product without changing its contract. Temporary data, hard-coded URLs, secrets in the client, an installation process that succeeds only once, and manually tuned UI remain because they “work for now.” AI is highly effective at satisfying the visible success condition. If you do not explain the future reasons for change, it will often bind several responsibilities together.

A request for a payment button, for example, can touch the UI, price display, provider configuration, checkout session, webhook verification, purchase entitlement, download, refund, and cancellation. If the screen is the only acceptance criterion, an AI may treat the client-side checkout.completed event as payment success. That event is a navigation signal, not the authority for payment. The server must not issue an entitlement until it verifies the transaction through the provider API or a signed webhook.

The answer is not a longer prompt. The answer is to identify who owns each product fact and to define the default state on failure.

Completion contract

1. Use the browser completion event only for navigation.
2. Create purchase access only after the server verifies the provider transaction.
3. Fail closed unless price, currency, product ID, and transaction state all match.
4. Repeated delivery of the same webhook must create only one entitlement.
5. Prove tests, the build, and the real callback path independently.

These five lines are shorter and stronger than “implement payments.” They establish invariants without freezing a particular implementation.

The Goal Is Less Rework, Not Slower Work

Writing a contract before a change, adding a small test, and checking a build can look slow. In reality, they move the hidden cost of post-generation rework to the beginning. A good vibe-coding workflow does not maximize the amount of code an AI writes. It shortens this loop:

- 1 Choose one observable problem.
- 2 Record the current and desired behavior as evidence.
- 3 Define the smallest change boundary.
- 4 Create or reproduce a failing verification.
- 5 Implement the change.
- 6 Pass narrow verification first, then broad verification.
- 7 Inspect the real user surface.
- 8 Update documentation and operating contracts.

When this loop is small, a misunderstanding by the AI produces limited damage. Bad assumptions surface quickly, and the human can review a bounded difference instead of rereading the whole codebase. Generation stays fast while promotion of a change remains deliberate.

Exercise 1: Separate the Layers of Completion

Choose one feature you are currently building. Avoid broad labels such as “finish signup,” “deploy the app,” or “integrate payments.” Record its state with this form:

Feature: _____
Source-alignment evidence: _____
Static-verification evidence: _____
Dynamic-verification evidence: _____
Packaging evidence: _____
Deployment evidence: _____
User-surface evidence: _____
Operations/recovery evidence: _____
Not yet verified: _____

An empty field is not a failure. Hiding that field behind the single word “done” is the failure.

2. AI Is Neither a Co-Founder nor an Autopilot

Using conversational language with an AI agent is convenient. “Investigate this,” “fix it,” and “take it all the way to launch” feel like collaboration. Product responsibility, however, cannot be delegated with the work. AI is a powerful operator, but it does not own the business objective, legal responsibility, customer promise, or decision to accept risk.

The most practical role definition is this:

AI gathers evidence, proposes changes, and acts within an authorized boundary. A human owns the objective, priorities, authority boundary, and promotion of claims into facts.

This definition does not reduce AI to passive autocomplete. It permits wide delegation across research, implementation, testing, documentation, and deployment verification. The deciding evidence simply comes from the external world, not from the AI’s sentence.

Four Modes of Work

AI work becomes safer when divided into four modes.

1. Exploration Mode

Read the repository structure, relevant files, official external documentation, and current deployment state. Change no files. Produce hypotheses and supporting evidence. The less clear it is where to make a change, the more important exploration becomes.

2. Diagnosis Mode

Separate symptoms from causes. Compare reproduction conditions, logs, recent changes, and data boundaries. A diagnosis request does not grant modification authority by default. Changing code while looking for the cause moves the evidence and can make the investigation slower.

3. Change Mode

Modify code, tests, and documentation together inside the authorized scope. The completion contract matters more than a fixed list of target files. An agent may discover a relevant file that no one anticipated.

4. Operations Mode

Change external state through builds, deployments, product registration, monitoring, or customer delivery. Before acting, ask whether the action is reversible, whether repetition is safe, and whether failure can damage existing customers.

One request can require all four modes. “Fix the bug and deploy it” includes exploration, diagnosis, change, and operations. Require the agent to retain evidence from each stage.

Authority and Verification Are Not Opposites

Giving an AI autonomy is not the same as skipping verification. Autonomy is the ability to choose the next safe action inside the objective without waiting for approval before every click. Verification is the ability to prove that the action succeeded in the external world.

A useful delegation contract looks like this:

```
Objective: Keep the notification badge consistent with the read state.  
Authority: Modify relevant source, tests, and documentation, and run local verification.  
Constraints: Do not change secrets or production data. Do not refactor unrelated features.  
Done: Prove the reproduction test, full typecheck/lint/test/build, and the real screen state.  
Report: Separate changed facts, evidence, and external states that still require outside authority.
```

Inside that contract, an AI may locate files, modify them, and repeat tests. Results outside its control—such as App Store approval, a real card payment, or receipt by another person’s mailbox—must remain “unverified.”

Three Illusions AI Commonly Creates

The first is the narrative illusion. A long explanation and a confident summary can look like evidence. Counter it by requiring command output, file locations, test results, and the state of the real URL.

The second is the near-success illusion. The system infers that production must work because the configuration looks correct. A DNS record existing does not prove that a custom domain is active. A successful build, a response from a platform default domain, a connected custom domain, and transfer of DNS authority are separate states.

The third is the substitute-success illusion. When the original path is blocked, the system points to the success of a similar path. It may replace the required authenticated Chrome session with an unauthenticated embedded browser, or