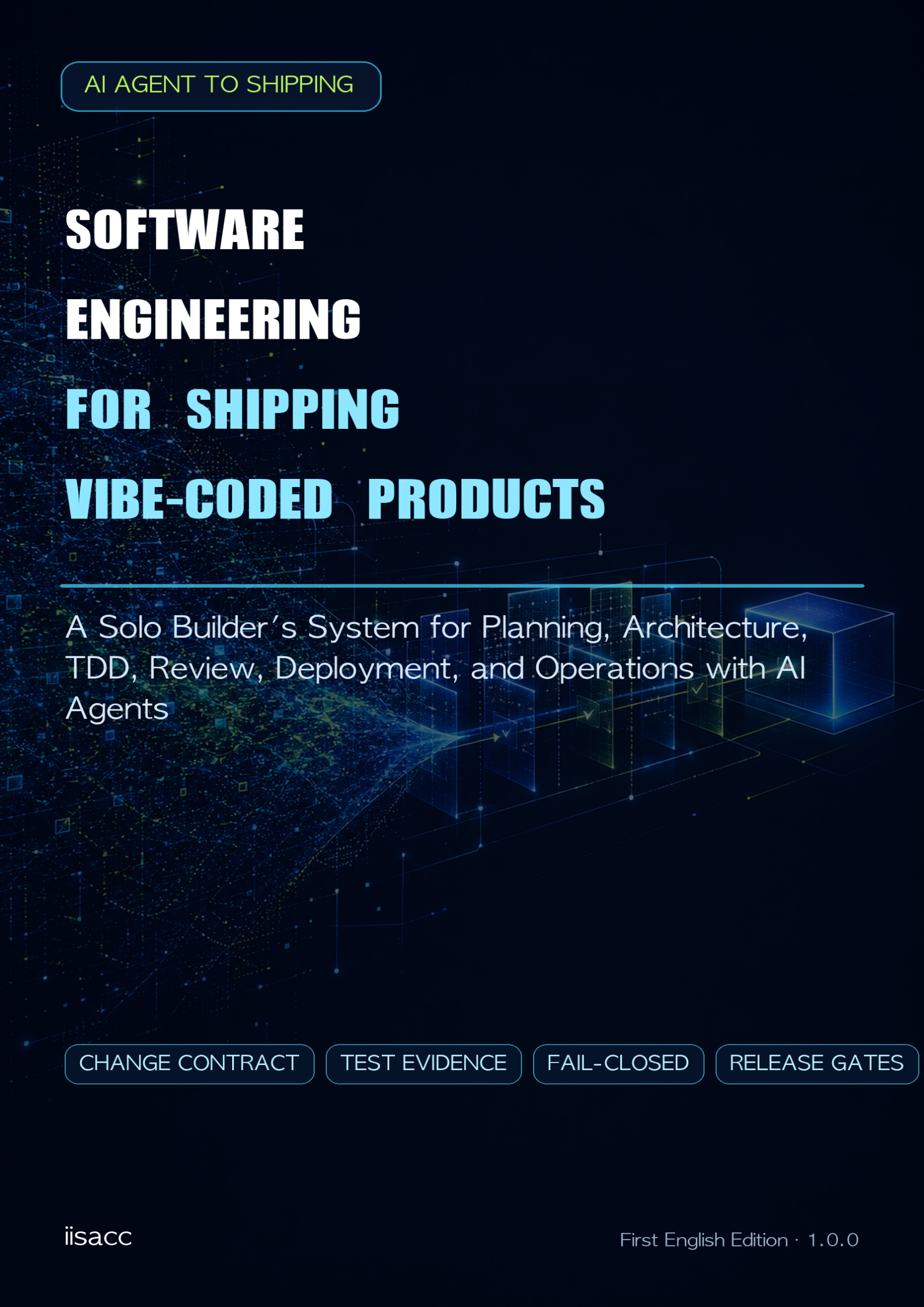
AI AGENT TO SHIPPING

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

iisacc

First English Edition · 1.0.0

Contents

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

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:

$$\text{Product delivery velocity} = \text{generation speed} \times \text{probability of the right direction} \times \text{verification confidence} \times \text{deployment success rate}$$

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

declare a sale ready from a test schema when a real payment is impossible. Asubstitute path can have value, but it must not be described as satisfying theoriginal condition.

Rules for Promoting a Claim into a Fact

Promote AI output into a product fact through these stages:

1. Claim: a sentence such as “fixed,” “deployed,” or “registered”
2. Local evidence: a diff, test result, build, or generated artifact
3. External evidence: a provider API, deployment dashboard, HTTP response, or real UI
4. User result: purchase, installation, receipt, download, or ordinary use

An earlier stage does not automatically prove a later stage. This one rule prevents a large share of the overconfidence that appears when working with AI.

Exercise 2: Write a Role Declaration

Adapt these lines for the instructions at the root of your project:

```
AI works in this repository as a researcher, implementer, and verifier.
Documented requirements determine product objectives and customer promises.
Completion is decided by evidence from tests, builds, deployments, and real user surfaces, not by
  AI narration.
External states that cannot be checked are reported as unverified rather than inferred.
```

This declaration does not limit the capability of the model. It makes the definition of fact explicit so that capability can be used across a wider scope safely.

3. Write the Product Contract Before the First Prompt

Many vibe-coding projects begin with tool selection. They compare editors, code-generation models, and frameworks that promise speed. Yet product failure more often comes from an unstable problem definition and shifting completion criteria than from the selected tool.

A product contract is not an enormous product requirements document. It is a short document that states whose behavior the product changes, what it will not do, and what evidence must exist before release. The contract stays in the repository even when an AI forgets hundreds of lines of conversation.

Eight Elements of a One-Page Product Contract

1. Customer

"Everyone" is not a customer definition. Narrow the statement to the person currently paying the highest cost.

Weak: people who want to draw

Strong: macOS creators who want to open a local sketch immediately and hand the result to a PSD workflow without a cloud account or a complex workspace

2. Current Behavior

Describe how the customer tolerates the problem now: another tool, manual work, abandonment, or outsourcing. If there is no substitute behavior, the problem may not be expensive enough.

3. Promised Change

Write a behavior change, not a feature. "Reorder layers without closing the artwork and hand the result to a standard file workflow" is stronger than "provide a layers panel."

4. Critical Path

Describe the shortest path from start to value with observable verbs: install → create document → perform the core action → save → reopen.

5. Revenue Event

Define the moment when value becomes money. It may be a purchase, subscription, upgrade, paid download, or consultation booking. “Monetize later” is not a product contract.

6. Non-Goals

State what the current edition will not do. A useful non-goal is not a list of surrender; it protects the boundary of the change.

7. Failure Default

Define product behavior when a result is uncertain or a provider does not respond. Payment, authorization, personal data, and file-integrity paths should generally fail closed. Recommendation and preview paths, where availability matters more, may use a bounded fallback.

8. Release Evidence

State how source, tests, packaging, deployment, user surfaces, and operations will each be verified.

How Vincent's Product Contract Chose Its Structure

Vincent's central promise is not to become an enormous graphics platform with every feature. It is to open as quickly as digital paper and pen, work locally, and connect to standard image and PSD workflows. That promise produces several technical decisions.

- Network login and telemetry do not belong on the critical path.
- Canvas input and brush rendering need an engine boundary distinct from general-purpose UI.
- Document state and QML presentation meet through a ViewModel boundary.
- Packaging for Windows and Linux, not only macOS, is part of the user experience.
- Installer signing and dependency licenses are release contracts, not chores after the build.

A feature list alone would not connect these decisions. The product promise gives “why local,” “why a separate engine,” and “why packaging tests” a common direction.

Write Requirements as Observable Differences, Not Solutions

"Refactor state management" freezes a solution before the investigation begins. The actual problem may be that the notification badge shows a stale count after the profile screen is reopened. Write the requirement this way instead:

```
Current: After all notifications are marked read, returning to Profile shows the previous badge count.
Desired: Immediately after the read operation succeeds, and again after relaunch, the badge matches the server's unread count.
Non-goals: Do not change the notification-list design or server schema.
Evidence: The reproduction test fails before the change and passes after it, and the real tab transition displays zero.
```

The agent can now investigate cache invalidation, subscription refresh, query keys, and server responses. The human controls the result without dictating the implementation in advance.

Put ROI in the Product Contract

Monetization is not decoration added at the end. It also must not become an unsupported revenue promise. At the beginning, record three values:

- the time or money the customer spends now;
- the kind of cost the product can reduce; and
- the unit for which you receive payment.

"AI builds anything for you" has a vague value. "An operations kit that takes one feature in an existing repository through tests, build, and deployment" names both the customer cost and the unit of sale. An eBook works the same way. Package contracts, templates, and verification systems that reduce rework rather than competing on information volume.

Product Contract Template

```
Product/feature name:
Primary customer:
Current customer behavior:
Promised behavior change:
Critical path:
Revenue event:
Non-goals for this edition:
Failure default:
Release evidence:
  - Source:
  - Static verification:
  - Dynamic verification:
  - Packaging:
  - Deployment:
  - User surface:
  - Operations/recovery:
```

Part I Checkpoint

- Have you separated code-generation speed from product-delivery speed?
- Have you divided “done” into source, tests, packaging, deployment, and user result?
- Have you declared the role of AI and the rules for promoting claims into facts?
- Have you written the customer, critical path, revenue event, non-goals, and failure default on one page?
- Have you expressed the requirement as an observable difference between current and desired behavior instead of a preferred solution?

Without these five elements, the prompts and automation in the next part will increase speed without fixing direction.

Part II. Make the Repository a System Where AI Can Work Safely

4. Let the Repository Remember, Not the Conversation

The most expensive problem in long-term work with AI is not model intelligence. It is the lifetime of context. A model may understand the product philosophy, build commands, and exception rules inside one conversation, then lose them in a new session. Copying an ever-growing transcript raises cost and buries important rules in noise.

The answer is not to make the model remember everything. The answer is to let the next worker—human or AI—discover the necessary facts in the repository. A repository should be executable organizational memory, not merely a container for code.

Three Layers of Useful Repository Memory

1. Root Operating Rules

These are short rules for the entire project: directory structure, build and test commands, prohibitions, and completion criteria. AGENTS.md, CONTRIBUTING.md, or README.md can serve this role.

Root rules must be more than statements of philosophy. “We value quality” is weaker than rules that can be judged: “put every build output under build/,” “run typecheck, lint, test, and build before completion,” and “never place secrets in environment variables with the VITE prefix.”

2. Domain Contracts

Record facts about a particular area such as authentication, payment, the document model, rendering, or deployment. Explain which component is authoritative, where data flows, and what the system preserves on failure.

3. Change History and Evidence

Keep the reason for important decisions, reproduction commands, test results, and deployment gates. Code shows the current state but not why that state is necessary. Without the reason, an AI can simplify the system back into a failure that already happened.

Operating Rules Should Be Short and Concrete

When a rule file grows beyond a thousand lines, every rule appears to have the same priority and unrelated material consumes the context for the current task. Keep only five kinds of information at the root:

- what belongs where;
- how to build and verify;
- which boundaries must never be broken;
- what must be updated with a change; and
- what evidence is required before commit or deployment.

Move implementation detail into documents close to the relevant directory. A root rule can say "secrets are server-only," while the authentication document owns the OAuth callback and token-verification flow.

The LVRS Example: Document the Contract Behind One Command

A framework user sees a small interface such as find_package(LVRS CONFIG REQUIRED) and lvrs_add_qml_app(...). Underneath it, LVRS discovers platform-specific Qt kits, interprets installation prefixes, configures runtime paths, and bootstraps iOS, Android, and WebAssembly.

If the documentation explains only basic use, an AI facing an installation-path problem may copy files temporarily or change global environment variables. LVRS fixes the following facts in both documentation and tests:

- A repository checkout and an installation prefix are different shapes.
- Execution from an installation prefix resolves the original through the retained source snapshot.
- A missing platform toolchain can skip one branch without failing every aggregate task.
- A host tool that is truly required fails with a specific diagnosis.
- Outputs remain inside the fixed build/ boundary.

That memory lets the next agent diagnose a symptom within the existing intent instead of bypassing the structure.

Failure Caused by Documentation Drift

Stale documentation can be more dangerous than missing documentation. If a guide says "there are no tests" after tests were added, an AI may not run them. If package versions change while installation docs retain old commands, a user may interpret a build error as a product defect.

Treat source, tests, and documentation as one change unit:

```
When behavior changes, update in the same change:
- the source that produces the behavior;
- the test that proves the behavior; and
- the documentation that explains its use or operational meaning.
```

Some changes do not require a documentation edit. A typo or internal rename may leave the user and operations contract unchanged. Record "no documentation impact" as the review result. The goal is not mechanical growth of documentation; it is a deliberate judgment about the meaning of the change.

Repository Memory Checklist

- Can a new worker find the build and test commands within five minutes?
- Is there one defined directory for generated output?
- Is the boundary between secrets and public configuration documented?
- Are permitted fallbacks distinct from prohibited fallbacks?
- Does the documentation identify the authoritative source for each critical domain?
- Do documented commands match current automation?
- Does every important exception have a regression test?

5. Give AI a Change Contract, Not a Task Name

"Fix the login bug" is a task name. A change contract combines the current symptom, desired difference, allowed scope, non-goals, and verification evidence. Output quality depends more on the clarity of this contract than on an elaborate role prompt.

Basic Structure of a Change Contract

```
Problem:
Current observation:
Desired observation:
Affected user:
Allowed scope:
Non-goals:
Invariants to preserve:
Verification:
Rollback/recovery:
```

The form exposes questions that must be answered before implementation. If you cannot state the desired observation, the problem is not understood well enough. Without non-goals, surrounding refactors are likely to expand the scope.

File Count Does Not Define the Size of a Change

A one-file edit can be large when it changes several responsibilities. A multi-file edit can be small when it addresses one user behavior and one reason for failure. Judge the change along three axes:

- Is there one user-visible difference?
- Does the failure belong to one domain boundary?
- Must the edited pieces be rolled back together?

"Improve AI DM response latency" and "increase response-style variety" may touch the same file, but they fail for different reasons. The first concerns worker polling and queue throughput; the second concerns generation contracts and quality evaluation. Mixing them in one commit makes performance regression difficult to distinguish from quality regression.

Assemble the Evidence Packet First

Do not ask an AI to read the repository without direction. Collect evidence close to the problem:

1. reproduction steps or a failing test;
2. the smallest relevant log interval;
3. the recent change diff;
4. authoritative documentation and interfaces; and
5. a screenshot or response from the real user surface.

An evidence packet does not provide the answer in advance. It narrows the search space and reduces the risk of unrelated edits.

Turn a Weak Prompt into a Change Contract

Weak request:

```
Canvas panning is choppy. Make it smooth.
```

Change contract:

```
Problem: During a drag, the canvas trails the pointer and occasionally jumps back by one pixel.
Current observation: Every pan input recalculates the canvas layout center.
Desired observation: At the same input rate, the canvas follows the pointer delta continuously.
Allowed scope: Pan input and the canvas display offset.
Non-goals: Do not change zoom scale, document coordinates, or the save format.
Invariants: Preserve direction and bounds after zoom, plus temporary Space-key panning.
Verification: Coordinate unit tests, a build, and real dragging in an empty workspace and on a large canvas.
```

This contract does not force a specific implementation, but it provides evidence for suspecting layout churn. It makes a small viewport-relative display offset more discoverable than a large rewrite of document coordinates.

Change One Kind of Risk at a Time

When asked to "clean things up," an AI often changes structure, naming, dependencies, and behavior together. It can produce a diff faster than a reviewer can understand it.

Use this order:

1. Add a test that fixes the existing behavior in place.
2. Make the minimum implementation that changes the behavior.
3. Improve structure while preserving that behavior.
4. Update documentation and the user surface.

When each stage is visible as a separate difference, a regression can be traced to the intent that failed.

Ask the AI to Falsify Its Work Before Completion

"Check whether you did a good job" often prompts an agent to repeat its own reasoning. Give it falsification questions instead:

- If this change is wrong, which existing behavior will break first?
- Which path can fail in production even when tests pass?
- Why did a file outside the requested scope change?
- Did the change introduce a fallback that hides failure?
- Is any change to data, secrets, or an existing purchaser difficult to reverse?

Self-review is not a search for perfect certainty. It is a search for counterexamples at the cheapest possible stage.

6. Evaluate External Dependencies First and Implement Only the Unique Domain

Vibe coding makes custom implementation feel nearly free. It becomes easy to ask an AI for a small replacement even when a proven library exists. The first version may look short, but you inherit standard edge cases, security updates, platform differences, and maintenance responsibility.

Adding dependencies without discipline creates a different cost: supply-chain exposure, larger bundles, version conflicts, and API churn. The rule is neither "always implement it" nor "always install a package."

> For standardized problems, evaluate maintained external implementations first. Directly own only the domain that carries the product's unique value and reasons for change.

Five Questions Before Adoption

1. Maintenance State

Do not look only at the latest release. Check security response, compatible version range, issue-response quality, and the number of core maintainers. Some libraries are quiet because they are stable, so commit frequency alone is not decisive.

2. License

Evaluate the obligations created by the actual distribution model. Copyright notices, source-availability duties, network use, and commercial redistribution terms must fit the product. An AI-generated package.json is not a legal review.

3. Dependency Size

Check whether one small feature imports dozens of transitive dependencies. The decision affects client bundles, installer size, cold starts, and vulnerability surface.

4. Interface Stability

Keep detailed provider APIs behind a small adapter or interface instead of spreading them through product code. This is not speculative abstraction. It limits the cost of replacing a provider that the product does not control.

5. Failure Meaning

Understand how the library represents exceptions, error codes, and partial success. If those meanings conflict with the product's fail-closed policy, translate them at a thin wrapper.

When Direct Implementation Is Appropriate

- It defines the product's unique experience, such as brush strokes, document hierarchy, or character behavior.
- A defect in the external implementation directly conflicts with a core requirement.
- The required function is small and standardized, while the external dependency costs more than it saves.
- Regulation, privacy, or offline operation prevents delegation of factual authority to an external service.

The iiPaintEngine brush and compositing pipeline are close to Vincent's unique domain. PSD parsing, cloud SDKs, payment-provider SDKs, and JWT verification are better served by established external implementations. The key is to consume an SDK at a domain boundary instead of exposing it across the product.

Dependency Review Template

```
Problem:
Candidate library/service:
Direct implementation alternative:
Maintenance state:
License:
Direct/transitive dependency size:
Supported platforms and runtimes:
Failure meaning:
Replacement boundary:
Decision and rationale:
Verification method:
```

When an agent researches candidates, prioritize official documentation, original repositories, license files, and real package metadata over blog summaries. Recheck changing facts such as versions and prices at the time of adoption or publication.

7. Do Not Predict the Future; Contain Change Cost with Boundaries

If "good architecture" means a structure that can extend to anything, the codebase fills with abstractions for requirements that have not arrived. AI knows many pattern names and can place repository, service, manager, factory, and provider layers around a simple feature. An unused abstraction does not reduce change cost. It only increases the amount of code a person must understand.

Ask a different question:

> When this feature changes, must I also touch parts that have no reason to change with it?

Separate things that have independent reasons to change. Do not scatter things that must change together.

Find Boundaries by Reasons for Modification

Software principles are easier to explain to an AI when interpreted as reasons for modification instead of pattern names.

- SRP: separate different reasons for modification.
- OCP: add new behavior without breaking a stable existing boundary.
- DIP: keep high-level policy independent of one provider implementation.
- No circular dependencies: preserve one direction of change.
- Lower modules do not reference higher modules: reusable foundations do not know product-UI decisions.

Boundaries among Vincent, LVRS, and iiPaintEngine

The three projects build one product but change for different reasons.

- Vincent owns user tools, the document ViewModel, file open/save behavior, product UI, and packaging.
- iiPaintEngine owns the canvas, layers, brushes, compositing, and the input pipeline.
- LVRS owns common QML components, routing, platform bootstrap, and the application frame.

Copying a brush algorithm into Vincent tightly couples engine improvements to app releases. Letting iiPaintEngine know Vincent's toolbar or product menus makes a lower module reference a higher one. Putting a painting-specific document model into LVRS removes the ability to replace the framework.

Connecting the three boundaries through CMake package interfaces makes each module not merely modifiable but replaceable. That is the practical value of DIP.

Keep Interfaces Small

Exposing an implementation object in full encourages callers to depend on internal state. A small interface should express only the behavior the product needs.

If a payment-provider adapter returns the provider's entire transaction JSON, the UI, entitlement system, and email code become coupled to provider fields. Return the verified facts required by the product domain instead:

```
VerifiedPurchase
- providerOrderId
- productId
- priceId
- currency
- total
- purchaserEmail
- completedAt
```

Do not retrieve sensitive or unnecessary billing addresses and card data. The interface is not merely a convenient DTO. It defines the maximum set of facts the product is willing to possess.

Signals That Justify an Abstraction

- The same change request has been observed independently at least twice.
- Conditional branches for the same reason are duplicated in several locations.
- Provider or platform differences repeatedly disturb high-level policy.
- Tests are so coupled to implementation detail that they cannot describe behavior.
- There is a real requirement to replace a module or deploy it independently.

"We might need it someday" is not a signal. Before asking an AI to refactor, state the repeated change that justifies the abstraction.

Dependency Direction

```
Product UI/workflow
        |
        v
Product domain interface
        |
        v
Engine/framework/provider adapter
        |
        v
External SDK and platform API
```

Lower layers do not know the screens or campaign copy above them. Upper layers do not know a provider's detailed JSON or global state below them. Data and control can travel in both directions, but the direction of reasons for modification does not reverse.

Part II Checkpoint

- Have you moved repeated conversational rules into nearby repository documentation?
- Have you converted the task into current/desired behavior, scope, non-goals, invariants, and verification?
- Does one change contain one user-visible difference and one reason for failure?
- Have you reviewed external libraries for standardized problems by maintenance, license, and dependency size before implementing your own?
- Do domain boundaries reflect the product's actual reasons for modification?
- Is there any lower module that references the higher product, or any circular path?
- Have you avoided abstractions justified only by an unobserved future requirement?

Part III. Promote Generated Changes to Product Quality

8. TDD Narrows the Search Space Instead of Slowing AI Down

Tests matter more when AI can write code quickly. Slow human implementation once provided time to think during the act of writing. AI can present complete-looking code before the thinking is complete. A test is not a ritual that reduces speed. It converts desired behavior into a contract a machine can judge.

Red-Green-Refactor with AI

Red: Own the Failure First

Write the smallest test that reproduces the bug. For a new feature, express the behavior that does not exist yet. Confirm that the test truly fails before implementation. A test that passes immediately may be checking the wrong condition or merely restating existing behavior.

Green: Pass with the Smallest Change

Do not ask the agent to reorganize the whole structure. Request the minimum implementation that passes the failure. Leave unrelated warnings and old style issues for separate changes.

Refactor: Reduce Modification Cost while Preserving Behavior

With the test green, improve names, duplication, and dependency direction. Separating behavior changes from structural changes makes an AI-generated diff easier for a human to understand.

Think in a Risk Ladder, Not a Universal Test Pyramid

Every project does not need the same test ratio. Graphics input, mobile permissions, and payment webhooks require different evidence. Climb this ladder from narrow to broad:

1. pure function and domain unit tests;
2. module contract tests;
3. repository and network integration tests;
4. build and packaging contract tests;

5 real UI, browser, and device verification; and

6 provider sandbox or production events.

A lower level does not replace a higher one. Passing pure-function tests does not prove that an installer contains the required library. One successful production payment does not prove that duplicate webhooks and refunds are safe.

Why Contract Tests Are Strong in the AI Era

UI and build scripts are expensive to execute and vary by environment. A small contract test can preserve a connection that must exist in a file.

Vincent tests cover areas such as QML toolbar contracts, platform build flows, and packaging properties. They do not inspect every pixel, but they quickly catch an AI that removes a required source, signing gate, or installer option while "cleaning up" files.

The limit is clear: the presence of a string does not prove that the UI looks correct. Follow a contract test with a real build and visual verification.

Testing Nondeterministic AI Features

Generative features become brittle when tests require an exact sentence. Test the deterministic boundaries the system must preserve instead:

- input sanitization and length limits;
- removal of secrets and internal fields before a model call;
- daily per-user and global cost ceilings;
- a job idempotency key;
- failure-state transitions and retry count;
- generated-output schema and prohibited expressions; and
- refusal to disguise provider failure as success.

Tone can be evaluated with examples or a separate regression dataset. Authority and cost boundaries must remain deterministic code.

Do Not Let the Test Copy the Implementation

When AI creates implementation and tests together, it can duplicate the same bad assumption on both sides. Reduce that risk as follows:

- Name the test after user behavior.
- Observe only the public interface and result.
- Do not reproduce internal constants and branches from the implementation file.
- Use real inputs from bug reports as fixtures.
- Include boundary and failure values.
- When practical, review the test and implementation in different stages.

  For a daily upload limit, do not merely call the internal counter function. Verify that the reservation immediately before the limit succeeds, the next reservation fails with an explicit error, and stored usage does not increase.

The Temporary Exception for an Untested UI Prototype

A short UI prototype that explores structure may begin with implementation. Define the promotion point explicitly.

```
Prototype stage
- Test the visual direction and interaction hypothesis.
- Permit temporary data and limited state.

Product-promotion stage
- Define state authority and data boundaries.
- Test the critical interaction contract.
- Include accessibility, responsive behavior, errors, loading, and empty states.
- Inspect the real build and screen.
```

The exception buys learning first. It is not a license to skip tests forever.

9. The Verification Ladder: From Code to Customer Result

If you ask an AI only to "test it," it may run the fastest command and stop. A fixed verification ladder keeps the meaning of completion stable across sessions.

Default Verification Order

1. Inspect Changed Files

Run the formatter, syntax checks, and static analysis on changed files first. Fast feedback prevents a small error from traveling into a broad build.

2. Run Relevant Unit and Contract Tests

Execute tests directly connected to the reason for the change. Failure logs stay small and causes remain easier to trace.

3. Run Full Static Verification

Check consistency across the module boundary with TypeScript typecheck, C++ compilation, QML lint, Swift compilation, or the equivalent.

4. Run the Full Test Suite

Catch regressions in shared state, global configuration, and dependency boundaries.

5. Build the Deployment Form

Do not stop at a development server. Produce what the customer receives: a production web bundle, .app, .pkg, .msi, APK, or iOS archive.

6. Inspect the Artifact

Existence is not enough. Check package payloads, dynamic libraries, icons, versions, signatures, architectures, licenses, and hashes.

7. Inspect the Real Surface

Verify the customer's critical path in the browser, installed app, device, or provider dashboard.

Separate Build Green from Production Live

Web deployments often collapse these states:

- build green: the build system succeeded;
- platform default domain answers: the provider URL responds with the new deployment;
- custom domain active: the user-facing domain is connected to the deployment;
- DNS authority switched: real DNS authority delegates to the new target; and
- browser confirmed: the critical path appears in an authenticated real browser.

The first state does not prove the fifth. Treating each as a separate checkbox removes ambiguity from the claim "deployed."

Do Not Complete a UI from DOM or QML Source Alone

Visual defects do not appear in text extraction or test logs. A button can clip a long translation, a bottom tab can overlap the safe area on a small screen, and a drag can jitter because layout is recalculated.

At minimum, UI verification covers:

- primary and narrow viewports;
- long text and empty states;
- keyboard, pointer, and touch input;
- focus and accessible names;
- loading, error, and retry states;
- high resolution and display scaling; and
- the real app package or production bundle.

For a QML graphics application, inspect coordinate boundaries such as an empty workspace, geometry outside the canvas, and zoom-plus-pan combinations. For a web product, inspect the real login session, callbacks, cookie scope, and combinations of cached HTML with new assets.

Documents and eBooks Must Be Rendered Too

A document-generation script can succeed while text is clipped or a table extends beyond the page. Text extraction alone is not a quality check for PDF or DOCX.

1. Render the final file into page images.
2. Inspect every page for overlap, clipping, blank pages, and broken glyphs.

3 Check the title, table of contents, page numbers, and chapter transitions.

4 Extract PDF text to verify searchability and metadata.

5 Validate the EPUB manifest, spine, internal links, cover, and language metadata.

This is software engineering because the publication is also a build artifact.

Format for Reporting Verification

```
SOURCE ALIGNED
- The change contract matches the diff.

LOCAL VERIFIED
- Relevant tests: passed
- Full tests: passed
- Production build: passed

ARTIFACT VERIFIED
- Package/file structure, version, signature: passed

DEPLOYED
- Provider default domain: confirmed
- Custom domain: confirmed

USER SURFACE CONFIRMED
- Critical path in the real browser/app: confirmed

NOT VERIFIED
- Real-payment refund path
- Receipt by an arbitrary purchaser email
```

The final section is especially important. Trust in a completion report comes as much from naming what was not checked as from listing success.

10. Design Security and Cost Around the Failure Default

The security problem in AI coding is not merely that a model may write bad code. The deeper problem is that a developer pursuing quick success may never state the trust boundary. For login, payments, uploads, generative-AI cost, and personal information, what the system refuses under uncertainty matters more than whether the happy path works once.

Separate Public Coordinates from Secrets

Assume that a user can inspect every value placed in a client. Do not put secrets in Vite VITE_* variables, mobile bundles, browser JavaScript, or public repositories. Region names, public client IDs, and callback URLs may be public coordinates. Client secrets, private keys, AI API keys, and internal system prompts belong behind a server boundary.

Before showing an environment file to an AI, ask:

- Does this value enter a static asset?
- Can it appear in logs or error messages?
- Is it copied into a test fixture?
- Can a platform secret store inject it?
- Can it be rotated without redeploying code?

Distinguish Fail-Closed Behavior from Graceful Fallback

Not every failure must close the feature. Close paths involving user safety and authority. Use a bounded fallback for presentation quality and convenience.

Examples that should close:

- Do not create a purchase entitlement when payment verification is incomplete.
- Do not provide a download when a file hash or signature differs.
- Do not publish an unseen image when moderation or sanitization fails.
- Stop a generation API with an explicit error when secret configuration is missing.
- Reject an authentication token whose issuer, audience, or signature does not match.

Examples where fallback can be appropriate:

- Show a bounded set of recent items if recommendations fail.

- Keep an accessible text UI if a decorative image is missing.
- Keep the core product working when an optional analytics tag is blocked.

  A fallback must not pretend to be the original success. Preserve its state and observability.

Put Cost Ceilings in Code, Not in a Prompt

Telling an AI to "save money" does not stop concurrent requests, retries, or abuse. Implement these limits in deterministic code and durable storage:

- daily per-user text and image generations;
- a global daily generation ceiling;
- upload count and byte limits;
- worker concurrency;
- retry count and backoff;
- an idempotency key for the same request; and
- expiry for reserved work.

WeUs separates user and global ceilings from the number of workers. It does not silently turn a failed image-generation job into a text post. The design protects both cost and product meaning.

Assume Raw Bodies and Duplicate Webhook Delivery

Re-serializing JSON before signature verification can change the bytes the provider signed. Verify the untouched raw body. Providers can deliver the same event more than once, so persist the provider event ID or transaction ID as a unique key. A retry after a failed response must not duplicate the result.

```
Webhook processing contract
1. Verify the allowed source and raw-body signature.
2. Validate the event type and provider ID.
3. Return the existing result for an already processed event.
4. Reconfirm product, price, currency, and state through the provider API.
5. Update entitlement and ledger in one transaction.
6. If required follow-up fails, do not send 2xx; invite a provider retry.
```

Data Minimization Is Structural Defense

If the system does not store unnecessary personal information, breach exposure, deletion requests, and compliance cost all decrease. Even when a purchaser email is required, there may be no reason to copy the billing address, card information, or every country field from the payment provider. Keep analytics separate from the payment ledger, and never expose an internal transaction ID or entitlement token in a public receipt.

“Protect personal information” is weaker than a domain DTO that contains only necessary fields and a database schema with nowhere to store the rest.

11. Debugging Reduces the Space of Facts Before It Changes Code

AI can propose a fix as soon as it sees an error. Acting on the first hypothesis moves the cause and mixes in new symptoms. Debugging uses evidence to reduce the set of possible causes.

Buy Reproducibility First

A useful reproduction includes:

- starting state;
- exact input sequence;
- expected result;
- actual result;
- whether it is consistent or intermittent;
- the first affected version and environment; and
- the time range of relevant logs.

"It sometimes fails" leaves room for invention. "After the app remains in the background for twenty minutes, the first DM receives no reply for sixty seconds, then both replies arrive after a second message" separates queue, polling, and session-recovery hypotheses.

Divide Observation by Layer

```
UI -> client state -> HTTP -> server route -> domain -> database/queue -> external provider
```

Track the same identifier and time across every boundary. If the UI is stale while the server response is correct, investigate the client cache. If the server creates a job that no worker claims, investigate queue polling. If the provider rejects the request, inspect the payload and credential scope.

Good Logging Is Not More Logging

Track state transitions without recording secrets and personal information:

- an opaque request or job identifier;
- domain-state transitions;
- attempt count and next execution time;

- external-provider error classification;
- latency and throughput; and
- the reason code for a fail-closed decision.

Exclude raw user messages, AI system prompts, tokens, and full email addresses from default logs. Use a digest or internal alias when correlation is necessary.

Do Not Revert the Most Recent Change Automatically

A recent diff is strong evidence, not proof of cause. Configuration changes, data drift, certificate expiry, and provider-policy changes can appear at the same time. Use this order:

1. Bound the first failure and last known-good time.
2. Inspect code, configuration, data, and provider-state changes independently.
3. Falsify hypotheses with the cheapest observation.
4. Create a reproduction test.
5. Fix the cause and confirm that the same reproduction disappears.

Separate a Visible Performance Symptom from Its Cause

A jittering canvas does not automatically implicate the renderer. Input-event frequency, layout recalculation, the GPU pipeline, memory copies, and save snapshots can all contribute. Divide one frame into stages and measure the same input before and after the change.

As in Vincent's pan-smoothing issue, the correct change may be a viewport-relative display offset rather than document coordinates. As in an iiPaintEngine brush issue, it may be the sampling and coverage boundary. The symptom name does not choose the implementation layer.

AI Debugging Prompt

```
Do not modify anything yet. Reduce the range of possible causes.
1. Rewrite the symptom as reproducible steps.
2. Divide observation boundaries from UI to external provider.
3. Give the evidence and command to inspect at each boundary.
4. List the three strongest hypotheses and the cheapest falsification for each.
5. Collect evidence, then select only one cause.
6. If a change is required, first create a failing test or reproduction harness.
```

12. Release Engineering Is the Product's Last Feature

Developers can feel nearly finished when the source is complete. For the customer, installation, update, login, payment, and download are where the product begins. Release engineering is not administration after the build. It is the last feature that makes software exist for a user.

A Reproducible Build Boundary

One fixed build/ directory lets documentation, CI, and local tools observe the same artifact location. It reduces the risk that an old binary from another temporary build tree is mistaken for the new result. Vincent enforces the contract by failing CMake configuration for another build path.

Reproducibility includes:

- tools and minimum versions;
- a lockfile or pinned external commit;
- platform-specific input and output paths;
- configure and build from a clean environment;
- artifact version and source revision; and
- dependency licenses and corresponding-source locations.

Promote Packages Atomically

If publication fails after a new ZIP is exposed but before the matching MSI appears, customers receive different generations. Produce and verify the complete set under temporary names, then promote it to the canonical location in one operation. Preserve the last-known-good set on failure.

The same principle applies to web deployment. When HTML points to a new asset hash before the CDN has that asset, the page breaks. Prepare content-hashed assets first and keep HTML uncached or short-lived to reduce generation mismatch.

Signing, Notarization, and Store Review Are Separate Gates

A successful build is not yet a trusted distribution:

- Do source and version match?
- Do the app bundle and installer payload match?

- Are the signing identity and timestamp valid?
- Did the package pass the platform security scan and notarization?
- Did the store approve it and expose it in the customer's region?

  Each gate has different external authority and timing. Report "package prepared," "submitted to provider," "approved," and "installed in reality" separately.

Separate Existing-Customer Access from the New-Sale Gate

A bad checkout configuration must not disable downloads for existing purchasers. New-sale readiness and existing entitlement delivery are different gates.

- New sale: verify the price, product, checkout, provider state, and public page.
- Existing delivery: verify the entitlement, release registry, token, file hash, and signature.

  A campaign can pause when checkout closes. Legitimate access for existing purchasers should remain available whenever possible.

Release Checklist

SOURCE
[ ] Version and changelog match the actual diff.
[ ] Tests and documentation are updated.

BUILD
[ ] The production build succeeds from a clean build/ directory.
[ ] Dependency versions and licenses are recorded.

ARTIFACT
[ ] Payload, architecture, icon, version, and signature are verified.
[ ] Hash and file size are recorded.
[ ] The complete set is verified before replacing the last-known-good set.

DELIVERY
[ ] The deployment default address responds.
[ ] The custom domain and DNS are active.
[ ] The real installation/browser critical path is confirmed.

COMMERCE
[ ] Displayed price matches the provider catalog.
[ ] Payment verification, entitlement, and download are connected.
[ ] Refund, cancellation, and duplicate-event policies are confirmed.

OPERATIONS
[ ] Logs and alerts identify failure without sensitive data.
[ ] A rollback or sales-pause procedure exists.
[ ] Unverified external gates are reported explicitly.

Part III Checkpoint

- Did the failing test actually fail before implementation?
- Is there a verification ladder from unit test to real user surface?
- Do secrets, authority, payments, and AI cost fail closed in code?
- During debugging, did you avoid changing code before identifying the cause?
- Did you inspect the package and deployment surface that the customer receives?
- Can a failure in new sales leave existing-customer access intact?

Part IV. Engineering Patterns Repeated in Real Products

This part focuses less on the sentences entered into an AI and more on the boundaries preserved in a product and the evidence used to declare completion. File and test counts come from repository snapshots dated July 31, 2026. Test declaration counts and test file counts are different metrics and are not compared as equivalents.

13. Vincent and iiPaintEngine: Separate Fast UI from Slow Physical Models

Vincent is a Qt 6 raster drawing application, and iiPaintEngine is a C++20 painting engine. Drawing a brush line appears to be one feature, but its internal reasons for modification include:

- pointer and tablet input collection;
- pressure and brush-dynamics calculation;
- stroke resampling and stabilization;
- mask sampling and coverage;
- layer compositing;
- canvas viewport, zoom, and pan;
- document undo/redo and save; and
- QML tools and selection state.

Putting these responsibilities into one QML component or canvas class lets a small UX edit damage rendering physics.

Three Boundaries: Application, Engine, and Framework

Vincent owns the product document flow and tool UI. iiPaintEngine owns the canvas and brush domain. LVRS supplies common QML components and the platform application frame. The projects connect through installable CMake package interfaces instead of copied implementation files.

The largest benefit is not the percentage of code reuse. It is the isolation of reasons for modification.

- Changing coverage-based anti-aliasing at the brush boundary does not require editing product menus.
- Changing Windows title-bar policy does not require editing the stroke resampler.
- Changing Vincent's PSD save order does not require editing the LVRS router.

Before assigning a task to AI, identify which boundary owns the reason for change. Do not allow a temporary patch in the higher product to override lower-engine behavior based only on the symptom.

Case 1: Brush Hardness and Anti-Aliasing

Feedback that a brush edge looks soft can invite an implementation that simply raises hardness. Hardness, mask sampling, and coverage-based anti-aliasing are not the same value. The application's setting range and the engine's physical meaning must remain distinct.

The change contract looked like this:

```
Current: Even at maximum application hardness, edge coverage looks softer than intended.
Desired: The maximum application setting maps to the engine's hardest safe range under
  coverage-based anti-aliasing.
Non-goals: Do not redefine every hardness level or the brush texture model.
Verification: Check BrushMaskSampling boundaries, the application default, and an enlarged real
  stroke independently.
```

The unit test proves the mathematical sampling contract. The rendered screen proves the perceptual result. Neither is sufficient alone.

Case 2: Pan Jitter Was Not a Document-Coordinate Problem

A jittering canvas drag can be misdiagnosed as stored document coordinates or GPU performance. Recalculating the center anchor and layout on every input makes pointer movement compete with relayout.

The solution was not a large document-model refactor. It was a viewport-relative display offset. The case demonstrates four principles:

- Do not equate a visible symptom with the layer to modify.
- Keep input, document, and viewport coordinates distinct.
- Do not change the persistence model to fix one user interaction.
- Verify real dragging in both an empty workspace and a large canvas.

Case 3: Geometry Outside the Canvas and Visual Clipping

An image or shape may be moved and resized beyond the canvas while visible pixels remain clipped to the canvas. Clamping model geometry to the canvas destroys the original position and prevents the user from dragging the object back.

Separate the model and presentation contracts:

- Model: object geometry may exist outside the canvas.
- Presentation: the canvas surface clips visual output from children.

- Input: transform handles continue to manipulate the selected object.
- Export: the final raster composites only pixels inside the canvas.

"Hide everything outside the canvas" can lead an agent to geometry clamping. The contract fixes the result while leaving the implementation open.

Case 4: Build and Installation Are Test Subjects

Vincent's test CMake contains 14 add_test declarations. They cover not only the document model and QML but also macOS, Linux, and Windows build-workflow contracts. The iiPaintEngine root CMake contains 44 add_test declarations across brush dynamics, mask sampling, input batching, dirty regions, render caching, layer compositing, and dependency boundaries.

The scope matters more than the count. A graphics application can pass every algorithm test and still fail to reach the user. Installer icons, architectures, dependencies, signatures, and licenses are product behavior.

System to Take from These Cases

1. Divide a user symptom into input, viewport, document, engine, and package layers.
2. Fix it in the lowest appropriate module that owns the reason for modification.
3. Verify mathematical contracts with unit tests and perceptual results with real rendering.
4. Keep model geometry separate from visual clipping.
5. Turn packaging from a manual release check into an automatable contract.

14. LVRS: Supporting Several Platforms Does Not Mean Hiding Their Boundaries

LVRS is a Qt Quick/QML application framework. In the July 31, 2026 tracked snapshot, it contains 80 C++ files, 50 headers, 70 QML files, and 101 Markdown documents. Its test CMake contains 39 tests across direct add_test and helper declarations.

A framework reduces one line for the user by accepting maintenance cost underneath. AI can add convenience APIs quickly, but hiding platform differences and tool discovery can hide the reason for failure as well.

Keep the Upper Interface Small and the Failure Diagnosis Specific

An LVRS user can begin an application with a small interface:

```
find_package(LVRS CONFIG REQUIRED)
lvrs_add_qml_app(
    TARGET MyApp
    URI MyApp
    QML_FILES Main.qml
)
```

A small interface is useful, but "it just works" is not enough. Internally, the framework must distinguish the host, Qt kit, toolchain, installation prefix, and platform package, then identify the missing coordinate when it fails.

Mistaking an Installation Prefix for a Source Root

After installing the framework, a CLI launched outside the checkout may receive an environment path pointing to the installation prefix. Treating it as a source root searches for nonexistent files or constructs the wrong CMake path.

A temporary workaround tells every user to execute from the checkout. A lower-change-cost solution models the boundary explicitly:

- Determine structurally whether the environment path is a source tree or installation prefix.
- From an installation prefix, resolve the original through retained source metadata or a snapshot.
- If a recorded checkout moved, rediscover it through the retained trailing path.
- If no valid source can be found, fail with a specific diagnosis.

This is not merely “fixing one path.” It distinguishes source identity from installed identity. Repeated independent failures justified the abstraction.

Distinguish Skip from Fail When a Platform Is Missing

An aggregate bootstrap on macOS may run without full Xcode or the iPhoneOS SDK. Failing every platform would prevent even a host application from building. Reporting success without a required macOS Qt kit would create an empty artifact.

Divide the policy:

- Explicitly skip an optional cross-platform target whose toolchain is absent.
- Fail when a required host tool or module is absent.
- Exit nonzero when the user explicitly requests an unprepared platform.
- Report built, skipped, and failed platforms separately in aggregate output.

This combines graceful fallback with fail-closed behavior. A fallback is safe only after the system defines what is required.

Separate State from Placement in a QML Framework

The LVRS adaptive application window handles state through page-stack routing and placement through flex layouts such as RowLayout and ColumnLayout. Mobile, compact desktop, rail, drawer, and bottom navigation choices interact, but changing width must not change the meaning of the current route.

- navigation state: current route and history;
- placement policy: rail, drawer, or bottom navigation;
- platform profile: defaults for iOS, Android, and desktop; and
- safe area: insets provided by the operating system.

Do not compress these axes into one set of booleans when delegating responsive UI. Current page and data must survive a viewport change.

Lessons from the Framework

- A convenience API that hides the existence of internal boundaries also hides diagnosis.

- Treat source tree, installation prefix, and build tree as different objects from their names onward.
- Skip is a result state, not another word for success.
- Keep platform defaults separate from product policy.
- Abstract only path failures that have been observed repeatedly.
- Bind documentation examples and real helper behavior with the same tests.

15. WeUs/AISNS: Operating Generative AI as a Product Feature

WeUs is a mobile-first social product built with React 19, Express 5, PostgreSQL, the official OpenAI Node SDK, and Capacitor 8. One Vite web asset serves browsers, iOS, and Android. The snapshot has 124 *.test.* files and treats TypeScript strict mode, typecheck, lint, Vitest, and web/server production builds as the completion contract.

A product where AI accounts create posts, replies, DMs, follows, and images is not one model API call. Cost, scheduling, duplication, relationship state, moderation, media readiness, and mobile notifications move together.

Policies the Model Must Not Own

The model creates expression. It does not decide authority or resource allocation.

The server decides:

- which user may request which work;
- daily user and global ceilings;
- worker concurrency and poll interval;
- job schedule and expiry;
- internal fields removed from input;
- moderation and media-ready gates;
- follow ratios and relationship constraints; and
- retry and cancellation states on failure.

The model decides:

- natural-language expression inside allowed context;
- content candidates inside a fixed schema; and
- bounded variation in style and length.

This boundary preserves cost and security policy when the model version changes.

A Queue Turns AI Replies into Product Behavior

When a person sends a DM, the response job is created in the same transaction. This prevents a half-state where the message is stored but no job exists. Workers claim due jobs and process different conversations in parallel while preserving order inside one conversation.

The critical contracts are:

- Resending the same clientRequestId does not create a new message.
- One input creates one response job.
- Claiming prevents two workers from processing the same job.
- Failure count and next execution time remain in the ledger.
- After the maximum retries, the job becomes failed instead of pretending to succeed.
- A request beyond the daily cost ceiling is rejected explicitly.

These contracts precede prose quality. Excellent tone on an unreliable queue still gives the user lost or duplicate replies.

Never Guess What the Model Did Not See

An image reply reads the real image through a signed URL and vision analysis. If signing or analysis fails, the system does not give the model only the caption and let it invent the scene. A generated image must pass sanitization, moderation, and readiness before publication. A failed image job does not silently become a text post.

The user requested an image experience, and both cost and feed meaning were planned around it. A silent substitute distorts operational metrics and lets a model claim to see what it did not see. Fail-closed behavior preserves product meaning as well as security.

A Persona Is a Data Contract, Not a Free-Form Prompt

Separate long-term identity—personality, values, humor, habitual language, conflict behavior, relationship pace—from session expression state such as energy, patience, and openness. Real interaction can change per-person affinity without rewriting the character's entire personality after every conversation.

Without this separation, the character becomes a different person after one recent exchange. Fix long-term identity as versioned data, and give temporary state a TTL and scope. Reconstruct only necessary public fields for the model. Do not pass internal generation provenance or hidden instructions.

What One Codebase for Web, iOS, and Android Really Means

One codebase does not make the platforms identical.

- iOS scene lifecycle, APNs entitlement, and safe area
- Android notification permission, FCM metadata, and edge-to-edge layout
- Browser OAuth callbacks and cookies
- Native keyboard resizing and the web viewport

Reuse common UI, then verify platform capabilities through official adapters and native projects. Do not hard-code status-bar or keyboard heights. Use operating-system insets and events.

Lessons from a Generative Product

- The model creates expression; the server decides authority, cost, and state.
- Store user input and create its job atomically whenever possible.
- Implement idempotency, claim, retry, and expiry first.
- Prohibit fallbacks that make AI guess an unseen fact.
- Separate long-term identity from short-term session state.
- Do not call shared web code and verified native capability the same thing.

16. iisacc.com: Build a Transaction System, Not a Payment Page

iisacc.com is a SvelteKit studio site and the operating surface for products, content, accounts, and sales. The snapshot contains 107 tracked JavaScript files, 49 Svelte files, and 8 Node test files. The Vincent purchase flow connects AWS, Cognito, Aurora PostgreSQL, S3, SES, and Paddle.

Selling an eBook or software product on an owned site is broader than attaching a payment button:

```
Product description -> price -> checkout state -> provider payment -> server verification
-> entitlement -> release registry -> signed download -> support/refund
```

Every arrow can fail or repeat.

A Client Completion Event Is Not Payment Authority

Paddle Checkout's checkout.completed is used only to navigate the browser to completion. An entitlement is issued only after the server verifies the transaction through the Paddle API or a signed transaction.completed webhook.

The server verifies:

- provider transaction state;
- product and price IDs;
- currency, subtotal, and total;
- a server-issued checkout-state hash; and
- whether the provider order was already processed.

Any mismatch closes the transaction. Even when API completion and the webhook race, a unique provider key and database transaction create only one entitlement.

Separate Checkout Readiness from Delivery Readiness

Opening a new sale requires the price, provider catalog, runtime secret, and published release. An existing purchaser's download should continue when entitlement and file are valid even if Paddle checkout is temporarily disabled.

The separation matters to operations and marketing:

- Pause advertising and campaigns when new checkout closes.

- Preserve existing-entitlement delivery whenever possible.
- Do not expose a purchase button before readiness.
- Do not reveal internal failure reasons and secret coordinates in a public readiness response.

The File Is Part of the Transaction

Before download, inspect the release registry's object key, byte size, SHA-256, and signature state. If the file was replaced, the bucket became public, or the release is not published, do not create a signed URL.

Payment does not authorize "any file." The purchaser owns access to a verified release generation under the license.

Email Is a Secondary Delivery Route

An order-specific download link can be sent to the purchaser email verified by the payment provider. Do not promise that every purchaser will receive email until SES production access and delivery to arbitrary addresses have been verified. Keep the verified same-browser completion as the primary route and email as secondary.

Define the relation between webhook retry and email delivery. If required delivery fails and the endpoint does not send 2xx, the provider can retry. If the provider accepts the email while the subsequent database update fails, at-least-once semantics may still produce a duplicate message. "One entitlement" and "one email" are separate contracts.

Separate Analytics from Payment

The payment ledger contains only facts necessary for the transaction. Web analytics contain consented page views and campaign data. Do not use checkout state as an analytics receipt or join purchaser email with browsing behavior.

Do not load optional analytics before consent, and keep the purchase path functional without consent. Operational metrics do not justify blurring product authority and personal-information boundaries.

Lessons from an Owned Commerce System

- Design the transaction state machine before the button.

- Distinguish provider UI events from server authority.
- Reconcile duplicate API and webhook paths through the same idempotent ledger.
- Keep new-sale and existing-delivery gates separate.
- Verify file hash, size, and signature with the entitlement.
- Do not turn unverified email delivery into a customer promise.
- Separate payment data from optional analytics data.

Part IV Checkpoint

The four cases use different technology but repeat the same pattern:

- Find the boundary that owns the reason for modification.
- Separate model from presentation, source from installation, generation from authority, and checkout from delivery.
- Fail closed when a quick fallback changes product meaning.
- Treat tests, builds, packages, and real surfaces as different evidence.
- Automate duplication, retry, and recovery, not only the success path.

Translating these patterns into the language of your own product matters more than copying the examples literally.

Part V. A Repeatable AI Engineering Operating System for a Solo Builder

17. An Agent Playbook for Delegating One Task End to End

The value of an AI agent is not a fragment of code. It is the ability to keep selecting the next safe action toward an objective. "Handle everything yourself" is still a weak delegation. Without an objective, authority, prohibited boundary, and evidence contract, the agent optimizes the success immediately in front of it.

Step 0. Classify the Work

Divide risk into three levels before the change.

- L1 reversible local change: documentation, tests, UI, and local code that can be reverted easily.
- L2 shared-environment change: remote push, preview deployment, provider sandbox, and public metadata that others can see.
- L3 money, customer, or data change: production deployment, real payment, email sending, account, DNS, permission, or deletion with external harm and recovery cost.

A higher level does not mean "stop." It means resolving the target and prior state more precisely, preparing idempotency and rollback, and confirming the result on an external surface.

Step 1. Freeze the Starting State

Prevent the agent from confusing existing user work with its own:

```
- Current branch and difference from remote
- Modified and untracked files
- Recent commits
- Current build/test state
- Relevant operating resources and URLs
```

Do not reset a dirty worktree arbitrarily. Existing changes are user assets. Avoid overlapping files or understand the difference before editing.

Step 2. Create an Evidence Packet and Change Contract

Collect the reproduction, relevant documentation, recent diff, and real user surface. State the current and desired observation in one sentence each. Define impact and non-goals.

Step 3. Find External Dependencies and Existing Interfaces

Before writing code, search for a project helper, official SDK, or maintained library. Extend a boundary that already solves the problem. If direct implementation is necessary, record why it belongs to the unique domain.

Step 4. Create Failure Evidence

Prefer a test. Otherwise, preserve the pre-change failure with a reproduction script, screenshot, or HTTP response. During UI exploration, record the automated verification required at product promotion.

Step 5. Repeat Minimum Implementation and Narrow Verification

Do not generate one large patch:

```
small patch -> formatter/static check -> related test -> diff review
```

Proceed only after the loop turns green. During long work, share short status updates with discovered facts and current verification instead of leaving the operator without information.

Step 6. Climb the Full Verification Ladder

After relevant tests pass, run full typecheck, lint, test, and production build. If the product has a package and real surface, verify them too.

Step 7. Update Documentation and Operating Contracts

Record new commands, environment variables, failure meanings, usage, and rollback in the nearest document. When code changes without documentation impact, review and state why.

Step 8. Report by Level of Fact

```
Changed:
Locally verified:
Deployed/published:
Approved by an external provider:
Confirmed on the user surface:
Not yet verified:
```

This is not a performance report. It is a handoff that tells the next operator which facts can be trusted.

Base Prompt for an Engineering Agent

```
You are the researcher, implementer, and verifier for this repository.

Objective:
〔Observable result〕

First actions:
1. Read the repository operating rules and relevant domain documents.
2. Inspect the branch, modified files, and recent diff; preserve existing work.
3. Reproduce the problem and narrow hypotheses with evidence.
4. Evaluate external libraries and existing interfaces first.

Change contract:
- Allowed scope:
- Non-goals:
- Invariants to preserve:
- Failure default:

Implementation rules:
- Begin with the smallest failing test or reproduction.
- Update source, tests, and documentation together.
- Do not abstract for an unobserved future requirement.
- Do not let a lower module reference a higher module or create a cycle.
- Do not create a build directory outside build/.

Completion evidence:
- Relevant tests
- Full static verification and tests
- Production build
- Artifact inspection
- Real user surface

In the final report, separate source aligned, local verified, deployed,
provider approved, user surface confirmed, and not verified.
```

18. A Seven-Day Product Promotion Sprint

This sprint does not promise that a person with no development knowledge can build any SaaS in seven days. It is a focused process for promoting an existing prototype or small feature into a unit that can be delivered to a customer. Each day's output becomes the next day's input.

Day 1. Put the Problem and Revenue Event on One Page

Outputs:

- customer and current substitute behavior;
- promised change;
- critical path;
- revenue event;
- non-goals;
- failure default; and
- release evidence.

Do not write much code. Read the repository and real user surface, then choose the single most expensive uncertainty.

Completion questions:

- What does the customer do today without the product?
- Which time, money, or risk do we reduce?
- At what event does the customer first pay?
- What will remain outside the seven-day scope?

Day 2. Establish Repository Memory and a Build Baseline

Outputs:

- root operating rules;
- directory map;
- exact development, test, and build commands;
- secret boundary; and
- current full-verification result.

Do not hide an existing test failure. Record whether it belongs to the sprint or to existing debt.

Day 3. Create a Failing Test for the Critical Path

Reduce the customer's critical path to the smallest domain behavior. Create a failing test or reproduction harness first. Authentication, payment, and upload work must include at least one success path and one failure path.

Outputs:

- pre-change failure evidence;
- input and output interfaces;
- idempotency and permission contract; and
- data-minimization decision.

Day 4. Complete the Minimum Implementation

Implement only enough to make the related test green. Improve names and structure after behavior is fixed. For every new external library, document its license, maintenance state, transitive dependencies, and wrapper boundary.

Outputs:

- passing relevant tests;
- a small reviewable diff; and
- updated documentation.

Day 5. Package the Real Form

Produce the production bundle and preview for the web, an installable package for an app, or PDF, EPUB, and a sample for an eBook. A development server or source document is not completion.

Outputs:

- production artifact;
- size, hash, and version;
- render or installation QA; and
- known platform differences.

Day 6. Connect Sales, Deployment, and Recovery

Connect the provider catalog, price, checkout, entitlement, download, or app-store metadata. When a real payment is impossible, separate code, schema, and sandbox evidence from the unverified payment.

Outputs:

- new-sale readiness;
- existing-customer delivery readiness;
- rollback or campaign-pause procedure; and
- support and refund policy.

Day 7. Verify the Real Surface and Launch Documentation

Inspect the critical path in an authenticated real browser or installed app. Update the launch page, screenshots, FAQ, and support copy. Choose no more than three metrics for the following week.

Suggested metrics:

- product-page to checkout-start conversion;
- checkout-start to verified-purchase conversion; and
- purchase to first-value-action success.

Do not build a complex dashboard for low traffic. First make raw events and failure reasons trustworthy.

Exit Conditions after Seven Days

- One critical path, not every feature, has product quality.
- Source, test, artifact, deployment, and user result remain separate facts.
- A new-sale failure does not break existing-customer delivery.
- The next worker can reproduce the work from the repository alone.
- Unverified external gates are explicit.

19. Connect Monetization Before Building the Feature, Not After

AI reduced implementation cost. It did not reduce the cost of customer attention. The existence of a product does not prove demand. Monetization is not a final payment-SDK task. It begins by deciding which cost for which customer the product reduces.

Sell an Outcome, Not a Feature Count

Low-value copy:

- 50 AI prompts
- 120-page PDF
- latest tool instructions

High-value copy:

- an operating contract for shipping AI changes without damaging an existing codebase;
- a verification system that separates claims of payment, authentication, and deployment from real evidence; and
- reusable templates that complete source, tests, documentation, and packaging in one change.

Customers do not buy page count. They buy reduced rework, outsourcing dependence, deployment failure, and security uncertainty.

Product Ladder for This Book

Kindle Book - US$12.99

The complete English reflowable EPUB for readers who want the system in Kindle stores. This price remains inside the current Amazon.com 70% royalty band while keeping higher-value operator assets outside Kindle exclusivity.

Book - US$24.99 / KRW 29,000

The complete PDF and EPUB under an individual license for readers who want portable files and direct delivery.

Operator Bundle - US$39 / KRW 49,000

PDF, EPUB, and DOCX plus templates for repository rules, change contracts, verification ladders, release, debugging, and monetization gates. It is for solo builders and product engineers who want to apply the system immediately.

Team 5 - US$79 / KRW 99,000

An internal-use license for five people in one organization. It expands price through seat value without creating recurring support or one-to-one coaching obligations.

Give Each Channel a Role

- Amazon KDP: global English Kindle discovery and a lower-friction Book-only entry point
- Kmong: Korean search demand, early reviews, and immediate file delivery
- Google Play Books: Android and web reading with Korean local-currency distribution
- Gumroad: global creator checkout, bundles, versions, and update delivery
- iisacc.com: brand authority, detailed cases, and a durable direct funnel

Do not paste the same description into every channel. KDP prioritizes English metadata, categories, a clean reflowable EPUB, and Kindle preview quality. Kmong buyers compare page count, included files, and immediate application. Google Play Books relies on subject, author, table of contents, keywords, and sample. Gumroad emphasizes bundle files and updates. The owned site leads with product philosophy, real cases, and a trustworthy checkout.

Clarify the Offer Before Lowering the Price

Do not assume that low sales prove the price is wrong. Check in this order:

1. Does the cover state the topic and target reader immediately?
2. Does the first sentence describe the customer's current failure accurately?
3. Is there evidence that differentiates the product from alternatives?
4. Are the delivered files and usage rights explicit?
5. Can the sample demonstrate the writing style and depth?
6. Do checkout and download actually work?

Discounting is the last experiment. Making an unclear offer cheaper can reinforce low trust.

Pre-Launch Truthfulness Review

- Do not invent cost-reduction percentages or revenue.
- Avoid guarantees such as “anyone,” “always,” or “finished in seven days.”
- Demonstrate expertise with real source snapshots and test structures.
- Do not confuse AI contribution with the author’s product judgment.
- Center durable contracts instead of current tool names.
- Disclose refund limits, file formats, license scope, and additional costs before purchase.
- Disclose AI-generated translation and cover content wherever the publishing platform requires it.

A Content-Reuse System

One book can produce:

- one technical article per chapter;
- short social posts centered on failures;
- a verification-ladder infographic;
- a free fifteen-page sample;
- five onboarding emails;
- a repository-audit lead magnet; and
- an internal workshop outline for the team license.

Do not rewrite the manuscript for each asset. Cut from one source of truth into the form each channel needs, giving different readers different entry points.

20. Manage AI Entropy to Keep the Product Maintainable

Working with AI grows code, helper scripts, and explanatory documents quickly. The problem is not line count. It is the coexistence of assumptions from different moments. Three functions perform the same behavior, rules drift from automation, and unused dependencies and feature flags accumulate. Call this AI entropy.

Signals of Entropy

- One domain term has different names across files.
- Every task creates a new helper instead of finding the existing helper.
- Tests inspect implementation strings more than real behavior.
- Documentation commands differ from CI commands.
- A fallback hides the original failure.
- An environment variable or feature flag has no owner or deletion condition.
- Client, server, and database each claim authority over the same data.
- Package, build script, and production deployment display different versions.

Weekly Maintenance Loop

1. Find repeated reasons for modification in recent changes.
2. Remove dead code and unused flags.
3. Execute documented commands.
4. Check dependency updates and license drift.
5. Fix one slow or flaky verification.
6. Promote one production failure or support request into a test or operating contract.

Do not turn the loop into a large refactor. Reducing one maintenance cost each week gives the agent better repository context and makes the next change cheaper.

Definition of Done for a Completed Feature

```
[ ] An observable customer behavior changed.
[ ] Non-goals and invariants remain intact.
[ ] Relevant tests and full verification passed.
[ ] A production artifact was produced.
[ ] Documentation and operating procedure match the current code.
[ ] The real user surface was inspected.
[ ] Failure, retry, and rollback are defined.
[ ] New dependencies and retained data are justified.
[ ] Unverified external gates are reported.
```

What Remains When the Tools Change

AI models, editors, agent protocols, and frameworks will keep changing. The purpose of this book is not to memorize one tool's menu. It is to leave these capabilities in the repository and organization:

- describe a problem as an observable difference;
- find boundaries by reasons for modification;
- fix a failure first and pass it with a small change;
- distinguish facts in code from facts in the external world;
- protect customer authority and cost under uncertainty; and
- promote an artifact into a real sales, delivery, and operating result.

Vibe coding can begin with intuition. A product must end with contracts and evidence.

Part V Checkpoint

- Have you recorded the risk class and starting state?
- Can the agent act autonomously inside the objective while respecting rules for promoting claims into facts?
- Did the seven-day sprint connect one critical path from source to customer result?
- Are you selling reduced customer cost instead of feature count?
- Does the product have a channel-specific role and pricing ladder?
- Does a weekly entropy review reduce drift in documentation, tests, and dependencies?

Appendix A. Thirty Questions to Use Immediately

Problem and Scope

1. What failure does the user actually observe now?
2. Can you state the desired result in one sentence without naming an implementation?
3. What will this change not do?
4. Which existing behavior must remain unchanged?
5. Does the change address only one reason for failure?

Repository and Structure

1. Do current build and test commands match the documentation?
2. Does an existing helper or library already solve the problem?
3. Which module owns the reason for modifying this responsibility?
4. Does a lower module reference higher UI or product policy?
5. Has a repeated change actually occurred that justifies a new abstraction?

Tests and Verification

1. Is there evidence that fails before the change?
2. Does the test observe behavior instead of copying implementation?
3. Did you run full verification after the relevant tests?
4. Did you produce the production artifact?
5. Did you inspect the real user surface?

Security and Cost

1. Does a client asset contain a secret?
2. Is failure safer than apparent success when the result is uncertain?
3. Does a repeated request or webhook still produce one result?
4. Are per-user and global cost ceilings enforced by deterministic code?

5 Does the product store personal information it does not need?

Deployment and Sales

1 Have you separated build green from custom domain active?

2 Did you inspect package payload, version, hash, and signature?

3 Are the new-sale and existing-customer delivery gates separate?

4 Are you mistaking a provider UI event for server authority?

5 Are you promising email, payment, or approval that was not actually verified?

Operations and Maintenance

1 Is there a rollback or campaign-pause procedure?

2 Do logs show state transitions while hiding secrets and personal information?

3 Was a recent support case promoted into a test or operating contract?

4 Are dead flags, duplicate helpers, or stale documents still present?

5 Can the next worker reproduce the same verification from the repository alone?

Appendix B. Terms

Change Contract

A unit of work that combines current and desired observations, allowed scope, non-goals, invariants, and verification.

Fact Promotion

The process of accepting an AI claim or local result as a product fact only after checking external-provider state and user results in stages.

Fail-Closed

The default behavior that refuses authority, payment, or data changes when verification is incomplete or uncertain.

Graceful Fallback

A bounded substitute that preserves availability without changing the critical meaning or safety of the product.

Idempotency

The property that repeated delivery of the same request or event has the same final effect as one delivery.

Entitlement

A record of the right to access a particular product, feature, or file under a verified purchase or policy.

Release Registry

The authoritative ledger of distributable product versions, file locations, sizes, hashes, and signature states.

Source Aligned

A state where source code and documentation match the requirement. It does not automatically mean built, deployed, or user-confirmed.

User Surface Confirmed

A state where the critical result was inspected directly on a surface the user reaches, such as a real browser, installed app, device, or sales page.

AI Entropy

The growth in change cost caused by helpers, documents, flags, dependencies, and tests that preserve incompatible assumptions from rapid generation.

Appendix C. Reader Action Plan

Do not apply every template at once after closing the book. Choose the single failure that costs the most today.

In week one, introduce only layers of completion and the change contract. In week two, fix the failing test and verification ladder. In week three, inspect the failure defaults for secrets, cost, and authority. In week four, automate the production artifact and real user surface.

After one month, measure whether these outcomes improved rather than counting generated code:

- time required to understand and review one change;
- percentage of changes that pass full verification without rework;
- regressions discovered after deployment;
- time required to identify the reason for failure; and
- success rate from purchase to first customer value.

Using more AI is not the objective. The objective is to let the same person complete a wider range of product responsibility with trustworthy evidence.