SDPF

User Manual and Framework Tutorial

From first problem to first verified, evidenced system

Hamza Abdullah · Software Development Prompting Framework · 2026

About This Manual

This manual is the practitioner's guide to SDPF — the Software Development Prompting Framework. It covers everything needed to take a real problem through the complete SDPF process: from the first observation of a gap, through a formally authored specification, to generated, verified, evidenced software.

This manual does not require prior knowledge of SDPF. It does assume you are building software with an AI model and that you have experienced the gap between what you asked for and what you got.

> This manual is the referenced companion to the SDPF Language Specification v1.3.1. The Language Specification is the normative authority. This manual is the practitioner's guide. When this manual describes what to do, the Language Specification defines why it is required.

What This Manual Covers

Part	Contents
Part 1 — Concepts	What SDPF is. Why it exists. The three principles. The key terms you need before you start.
Part 2 — Phase 0	How to define a problem correctly before writing a single requirement. Step-by-step with worked examples.
Part 3 — The Specification	How to author a complete SDPF specification. Style selection. All ten required sections. TVG. Locking.
Part 4 — Generation and Verification	How to generate tests, generate implementation, run the verification gate, and export the evidence package.
Part 5 — Reference	Quick-reference tables, common errors, and the full project checklist.

Part 1 — Concepts

The Problem SDPF Solves

When you ask an AI model to build something, it builds something. Whether it builds what you intended is a different question.

The gap between what you intended and what was built is not a model failure. The model did exactly what it was told. The problem is that what it was told was incomplete, ambiguous, or contained facts about the environment that turned out to be wrong. The model filled the gaps with its best guess. The guess was plausible. It was not what was intended.

> The problem is not the AI model. The problem is the specification. Better models make better guesses. They are still guessing. The correct intervention is a complete specification — one where there is nothing to guess.

SDPF is the formal discipline for writing complete specifications for AI-generated software. A specification written in SDPF gives the AI model nothing to speculate about. The output is not a better guess. It is a mechanical execution of a verified contract.

The Three Core Principles

Principle	What It Means in Practice
Specification First	No tests are generated, no implementation is generated, and no AI model is submitted to until a complete, locked specification exists. The specification is the contract. Everything else executes it.
Facts Before Execution	Every technical fact asserted in the specification — tool versions, API endpoints, file paths, database schemas — must be verified by running a command in the actual target environment before implementation begins. If the fact is wrong, the specification is corrected. Nothing proceeds around a failed fact check.
Verification Always	Every run ends with a verification gate. All eleven structural invariant checks must pass. All tests must pass. The run is not complete until they do. The CLOSURE STATUS must be COMPLETE before evidence is exported.

Key Terms

Term	Plain-Language Definition
Phase 0	The mandatory first step of every SDPF project. Before writing a single requirement, you define the problem: what is the measurable gap, what is the desired state, what is the impact. No specification begins until Phase 0 is complete.

Specification	The complete, locked contract defining what the system must accept, what it must do, what it must return, and what must happen when it fails. Written before any tests or code.
TVG	Technical Verification Gate. Before locking the specification, every technical fact it asserts must be verified by running a command in the live target environment. If a fact is wrong, the specification is corrected. Non-negotiable.
REQ-ID	A unique permanent identifier assigned to every requirement when the specification is locked. Format: R-001, R-002, R-003. Appears in tests, code, and the traceability matrix.
TEST-ID	A unique identifier assigned to every test when the test suite is generated. Format: T-001, T-002, T-003. Referenced in the traceability matrix alongside REQ-IDs.
Style	The specification dialect matched to your system type. 17 normative styles exist. Style selection is the first decision in Phase 1.
Evidence Package	The signed, tamper-evident artifact produced at the end of a verified run. Contains the problem statement, the locked specification, all tests, the implementation hash, the verification record, and a cryptographic provenance signature.
Bounded Stochasticity	The principle that AI output does not need to be identical across runs — it needs to satisfy all structural invariants defined in the specification. Variation in surface form is acceptable. Variation that violates invariants is not.

Part 2 — Phase 0: Define the Problem

> Phase 0 is mandatory. No specification may be written until Phase 0 produces a validated problem statement. This is not a recommendation. It is a hard gate. A specification written without a validated problem statement is solving an undefined problem.

Phase 0 is entirely a human activity. No tool and no AI model does this for you. You observe, measure, and define the problem. The output is a single validated problem statement.

The worked example throughout Part 2 uses a library book loan system — a small, simple problem. Follow the example step by step and apply the same structure to your own problem.

Step 1 Observe the Current State

Describe what is happening that should not be, or what is not happening that should be. Do not name a cause. Do not name a solution. State only what you can observe and measure.

Rule	Why It Matters
Observable	The gap must be visible or measurable in the real world. Not an opinion or assumption.
Present tense	Describe what exists now, not what happened once or what might happen.
No cause	Do not say why the problem exists. That is analysis. This step is observation only.
No solution	Do not say how you will fix it. You have not written the specification yet.

WORKED EXAMPLE — Current State

Library book loan records are created manually by staff on paper and entered into the digital system at end of day.

> Common mistake: 'The system is slow because the database is not indexed.' This names a cause (unindexed database) and implies a solution (add an index). Rewrite as: 'Query response time averages 8 seconds under normal load.'

Step 2 Define the Desired State

State the specific, measurable target condition. This becomes the basis for your Output Guarantees and Verification Requirements. You will know when it is achieved because it is measurable.

WORKED EXAMPLE — Desired State

Every book loan is recorded in the digital system at the moment of loan. Zero loans exist in a paper-only state at any point after the loan act is complete.

Step 3 Quantify the Gap

Express the difference between current and desired state as a number, frequency, percentage, or time measure. A qualitative gap does not satisfy this requirement.

WORKED EXAMPLE — Quantified Gap

Current: average 6–8 hour delay between loan and digital record creation. Target: 0 seconds delay. Frequency: approximately 400 loan events per month across the library system. Incident rate: 23 duplicate loan incidents in the 60-day period March–April 2026.

> 'Significant delay' is not a quantified gap. 'Average 6–8 hour delay, 400 events per month' is a quantified gap. If you cannot put a number on it, keep observing until you can.

Step 4 State the Impact

State the business consequence of the gap in cost, risk, time, or compliance terms. Not a technical description. Not a feature request. The impact justifies why this problem is worth solving.

WORKED EXAMPLE — Impact

23 duplicate loan incidents in 60 days where a book was loaned twice because the first loan had no digital record. Staff time lost to manual reconciliation estimated at 4 hours per week.

Step 5 Write and Validate the Problem Statement

Combine steps 1–4 into a single statement using this structure:

> [Current state] against a target of [desired state], resulting in [impact].

WORKED EXAMPLE — Problem Statement

Library book loan records are created manually by staff on paper and entered into the digital system at end of day, against a target of zero delay at point of loan, resulting in 23 duplicate loan incidents in the 60-day period March–April 2026 where a book was loaned twice because the first loan had no digital record.

Apply the Four Validation Tests

Every problem statement must pass all four tests. If any test fails, rewrite the statement and test again. Phase 0 is not complete until all four pass.

Test	Pass Condition
T-1 Observable	The gap can be seen or measured in the real world. Not an opinion.
T-2 Bounded	The problem has a clear scope. Not 'everything is slow.'
T-3 Cause-Free	No language implying why the problem exists. No 'because', 'due to', 'caused by.'
T-4 Solution-Free	No language implying how the problem will be solved. No 'by implementing', 'using', 'through.'

Applying the Tests to the Worked Example

Test	Assessment
T-1 Observable	PASS — 23 incidents in 60 days. Measurable.
T-2 Bounded	PASS — Scoped to book loan recording at point of loan.
T-3 Cause-Free	PASS — No cause named in the statement.
T-4 Solution-Free	PASS — No solution named. 'Zero delay' is a target, not an implementation.

GATE	**Phase 0 is complete when the problem statement passes all four tests AND a problem owner is named. Phase 1 cannot begin until this gate is passed.**

Name the Problem Owner

The problem owner is the person who experiences the gap, is accountable for closing it, and is the primary stakeholder. Record their name. Phase 0 is not complete without one.

WORKED EXAMPLE — Problem Owner
Head Librarian

Part 3 — The Specification

The specification is the contract. It defines exactly what the system must accept, what it must do, what it must return, and what must happen when it fails. Every requirement traces to the problem statement. Nothing may be added that does not derive from the validated problem.

> The specification is written before any tests or code. Tests are generated from the locked specification. Code is generated from the locked tests. This order is not optional.

Step 6 — Select a Style

Style selection is the first decision in Phase 1. The style governs how the specification is structured, what additional sections are required, how the AI model is framed, and what the verification gate checks. Evaluate the conditions below in order. Stop at the first match.

Condition — first match wins	Style
Hardware or real-time performance constraints are the primary design driver	Style 8 — Constraint-Based
System is regulated, compliance-driven, or security-critical	Style 10 — Compliance-Driven
Requirements must emerge through exploration	Style 2 — Exploratory Problem-Solving
Requirements will change frequently	Style 5 — Iterative Refinement
Primary deliverable is a user interface	Style 3 — User Experience-Centered
System is a library or shared component	Style 7 — Test-Driven Specification
System is distributed across multiple services	Style 6 — System Architecture
Goal is to improve existing working code	Style 9 — Maintenance / Refactor
Audit ledger or chain-of-custody system	Style 12 — Sovereign Structural Ledger
Multi-module, multi-team architecture	Style 13 — Hybrid System Composition

None of the above — deterministic system with defined input/output	Style 1 — Technical Specification

WORKED EXAMPLE — Style Selection

The library loan system is a deterministic REST API with defined input/output. No hardware constraints, no regulation, no exploration needed. Style 1 — Technical Specification.

Step 7 — Write the Ten Required Sections

Every SDPF specification must contain all ten sections. Write them in order. Do not skip any section. Do not lock the specification until all ten are complete.

Section	What to Write
S-01 Style Declaration	State the style ID and name. Example: Style 1 — Technical Specification.
S-02 Style Context	State the domain scope and system boundary derived from the problem statement. What does this system do and where does its responsibility start and end?
S-03 External Contract	State the protocol (REST, gRPC, CLI, etc.), the data format (JSON, XML, etc.), and the interface version.
S-04 Input Contract	Define every valid input. For each field: name, type, constraints (length, format, range), required or optional.
S-05 Processing Rules	Define the complete, deterministic logic for every valid input. Every conditional branch has an action. No undefined states. No ambiguous language.
S-06 Output Guarantees	Define the complete structure of every success response. Every field, every status code.
S-07 Exception Handling	Name every reachable error condition. For each: a machine-readable error code, a human-readable description, the HTTP status code (or equivalent), and a recovery or escalation path for CRITICAL errors.
S-08 Technical Verification Gate	List every technical fact the specification asserts. For each: the asset name, the asserted value, a command to verify it in the live environment, and the pass condition. See Step 8.
S-09 Verification Requirements	List all tests with pass conditions. Every CRITICAL and REQUIRED requirement must have at least one test.
S-10 Traceability Matrix	Map every REQ-ID to the TEST-IDs that verify it. Format: R-001 → T-001, T-002 → implementation-artifact

Writing Requirements

Every requirement begins with a priority tag. No exceptions.

Tag	When to Use
[CRITICAL]	Directly addresses the problem gap. Failure to satisfy this requirement means the system does not close the problem. At least one CRITICAL requirement must address the gap from the problem statement.
[REQUIRED]	Necessary for the system to function correctly. Failure to satisfy this requirement makes the system incomplete.
[OPTIONAL]	Desirable but not essential. Failure to satisfy this requirement does not block delivery.

WORKED EXAMPLE — Requirements

[CRITICAL] Record the loan in the loans table within 1 second of request receipt. If the write does not complete within 1 second, rollback and return HTTP 503 with error code WRITE_TIMEOUT. [CRITICAL] Validate that book_id is not currently on loan (status = ACTIVE in loans table). If already on loan, reject with HTTP 409 and error code BOOK_ALREADY_ON_LOAN. [REQUIRED] Return confirmed loan_id and due_date (loan_date + 14 days) in the success response. HTTP 201.

> Never write: 'Handle errors appropriately.' This is ambiguous. Name every error condition. Give it a code. Give it a status. Define what the system does when it occurs.

Step 8 — Complete the Technical Verification Gate

Before the specification can be locked, every technical fact it asserts must be verified against the live target environment. This is not optional. It is the most important enforcement mechanism in SDPF.

> A TVG failure is a specification error, not a build problem. The specification asserted a fact that is not true in the environment where the software will run. Correct the specification. Then proceed.

For Each Technical Fact

TVG Field	What to Write
Asset	The name of the tool, library, endpoint, or environment variable.
Asserted	The value the specification claims. E.g. Python 3.11, PostgreSQL 15.
Command	A command that can be run in the target environment to check the fact. E.g. python --version
Pass Condition	What the output must look like to confirm the assertion. E.g. output begins with 'Python 3.11'
HALT Rule	If the output does not match the pass condition, work stops. Correct the

	specification. Re-run the check. Nothing continues around a failed TVG entry.

WORKED EXAMPLE — TVG Entries

Asset: Python | Asserted: 3.11.x | Command: python --version | Pass: output begins with 'Python 3.11' Asset: PostgreSQL | Asserted: 15.x | Command: psql --version | Pass: output begins with 'psql (PostgreSQL) 15' Asset: FastAPI | Asserted: 0.110.x | Command: pip show fastapi | grep Version | Pass: output contains 'Version: 0.110'

Run every command in the actual target environment before proceeding. Record the actual output. Compare to the pass condition. If any entry fails, update the specification to match reality — or fix the environment to match the specification — then re-run.

GATE	**Lock the specification only after: all ten sections are complete, all TVG entries have passed against the live environment, and all requirements have been reviewed for CRITICAL-CRITICAL conflicts. Locking is irreversible within a run.**

On Locking

When the specification is locked, every requirement is assigned a permanent REQ-ID beginning at R-001. These identifiers never change. They follow the requirement through every downstream artifact — tests, code, the traceability matrix, and the evidence package.

Part 4 — Generation and Verification

Step 9 — Generate and Lock Tests

Tests are generated from the locked specification. Not from the code. Not after the code. From the specification, before any implementation exists.

Every test references the REQ-ID it tests using a trace comment in the test body. Example: # Trace: R-001. Every test is assigned a TEST-ID in T-NNN format beginning at T-001.

WORKED EXAMPLE — Test Trace Comment

```
def test_write_within_1_second():     # Trace: R-001     # CRITICAL: loan write must complete within 1 second     response = client.post('/api/v1/loans', json=payload)     assert response.elapsed.total_seconds() < 1.0     assert response.status_code == 201
```

Once the test suite is reviewed and correct, lock it. No tests may be added, removed, or modified after locking. Implementation begins after the test suite is locked.

GATE	**Test suite locked. Implementation gate open. No implementation may be generated before this gate.**

Step 10 — Generate the Implementation

Submit the locked specification to the AI model. The model generates the implementation. Every function in the generated code references the REQ-ID it implements using an annotation or comment.

WORKED EXAMPLE — REQ-ID Annotation in Generated Code

```
async def create_loan(payload: LoanRequest, db: AsyncSession):     """Trace: R-002 | CRITICAL"""     # Requirement: book must not already be on loan     existing = await db.execute( select(Loan).where(Loan.book_id == payload.book_id,                 Loan.status == 'ACTIVE'))     if existing.scalar():         raise HTTPException(409, 'BOOK_ALREADY_ON_LOAN')  # R-002
```

> Any function in the generated code that has no REQ-ID annotation is untraced. It was generated outside the specification-governed contract. It should be removed or traced to a requirement before verification proceeds.

Step 11 — Run the Verification Gate

The verification gate runs eleven structural invariant checks. All eleven must pass. A single failure holds the stage at CODE_GENERATED and blocks evidence export. Run the verification gate every time.

Check (VCR-INV ID)	What It Confirms
VCR-INV-1 spec_exists	A specification was formally authored before execution began.
VCR-INV-2 style_defined	A style was declared in the specification.
VCR-INV-3 tests_exist	Tests were generated from the specification.
VCR-INV-4 tests_locked	Tests were locked before implementation.
VCR-INV-5 implementation_exists	An implementation was generated.
VCR-INV-6 traceability_complete	Every CRITICAL and REQUIRED requirement has at least one test.
VCR-INV-7 style_constraints_present	Style-specific structural markers are present in the specification.
VCR-INV-8 style_N_engine_verified	Style-specific engine tags appear in tests and code.
VCR-INV-9 policy_engine_passed	Specification content satisfies the semantic rules for the declared style.
VCR-INV-10 ci_gate_ready	A CI workflow was generated.
VCR-INV-11 provenance_signing_ready	A signing key is configured for evidence tamper-proofing.

After all eleven checks pass, run the full test suite. All tests must pass. CLOSURE STATUS must be COMPLETE before the evidence package can be exported.

GATE	**CLOSURE STATUS = COMPLETE. All eleven structural invariant checks passed. All tests passed. Evidence package may now be exported.**

Step 12 — Export the Evidence Package

The evidence package is the signed, tamper-evident record of the entire run. It is produced automatically at the end of every verified run. It is not optional. It is the deliverable that proves the SDPF discipline was followed.

Evidence Package Field	Contents
problem_statement	The validated Phase 0 problem statement with all four components and the problem owner.
specification	The complete locked specification with all REQ-IDs.
tests	All test cases with TEST-IDs and REQ-ID trace comments.
implementation_sha256	SHA-256 hash of the implementation code.
verification	Verification Closure Record with results for all eleven VCR-INV checks.
traceability	One row per requirement: REQ-ID → TEST-IDs → implementation artifact.
provenance	HMAC-SHA256 signature over the complete package. Tamper-evident.

Archive the evidence package. It is the authoritative proof that the problem was defined, the specification was complete, all facts were verified, all tests passed, and all invariants were satisfied. In regulated environments, it is the audit trail.

Part 5 — Reference

Complete Project Checklist

Use this checklist for every SDPF project. Every item must be complete before the run is considered closed.

Phase 0

☐	Current state observed and described. Observable and measurable.
☐	Desired state defined. Specific and measurable target condition.
☐	Gap quantified. Number, frequency, percentage, or time measure.
☐	Impact stated. Cost, risk, time, or compliance terms.
☐	Problem statement written combining all four components.
☐	T-1 Observable — PASS
☐	T-2 Bounded — PASS
☐	T-3 Cause-Free — PASS
☐	T-4 Solution-Free — PASS
☐	Problem owner named.

Specification — Phase 1

☐	Style selected. Conditions evaluated in order. First match applied.
☐	S-01 Style Declaration — complete
☐	S-02 Style Context — complete
☐	S-03 External Contract — complete
☐	S-04 Input Contract — complete. All fields named, typed, constrained, classified.
☐	S-05 Processing Rules — complete. All branches defined. No undefined states.
☐	S-06 Output Guarantees — complete. All success responses defined.
☐	S-07 Exception Handling — complete. Every error named, coded, and shaped.
☐	S-08 TVG — complete. Every technical fact has a command and pass condition.
☐	S-09 Verification Requirements — complete. Tests listed with pass conditions.
☐	S-10 Traceability Matrix — complete. Every CRITICAL and REQUIRED requirement mapped.
☐	Every requirement begins with a priority tag.
☐	At least one CRITICAL requirement addresses the problem gap directly.
☐	All TVG entries run against live environment. All pass.
☐	No unresolved CRITICAL-CRITICAL conflicts.

□	Style-specific additional sections present (if required by declared style).
□	Specification locked. REQ-IDs assigned.

Tests

□	Test suite generated from locked specification.
□	Every test references its REQ-ID with a trace comment.
□	Every test assigned a TEST-ID in T-NNN format.
□	Test suite reviewed. All tests are correct derivations of requirements.
□	Test suite locked. No modifications after locking.

Implementation

□	Implementation generated from locked specification.
□	Every function annotated with its REQ-ID.
□	No untraced functions (functions with no REQ-ID annotation).

Verification

□	VCR-INV-1 spec_exists — PASS
□	VCR-INV-2 style_defined — PASS
□	VCR-INV-3 tests_exist — PASS
□	VCR-INV-4 tests_locked — PASS
□	VCR-INV-5 implementation_exists — PASS
□	VCR-INV-6 traceability_complete — PASS
□	VCR-INV-7 style_constraints_present — PASS
□	VCR-INV-8 style_N_engine_verified — PASS
□	VCR-INV-9 policy_engine_passed — PASS
□	VCR-INV-10 ci_gate_ready — PASS
□	VCR-INV-11 provenance_signing_ready — PASS
□	All tests pass.
□	CLOSURE STATUS = COMPLETE

Evidence

□	Evidence package exported.
□	Provenance signature valid.

□	Evidence package archived.
□	Problem owner confirms the gap defined in Phase 0 is closed.

Common Errors and How to Fix Them

Error	Fix
Problem statement contains a cause	Remove the cause. Describe only what is happening. T-3 fails if you can identify why in the statement itself.
Problem statement contains a solution	Remove the solution. 'By building an API' is a solution. T-4 fails. Describe the gap, not how to close it.
TVG entry fails	Correct the specification to match the actual environment — or fix the environment to match the specification. Re-run the check. Never proceed around a failure.
Requirement without a priority tag	Add a priority tag. Every requirement must begin with [CRITICAL], [REQUIRED], or [OPTIONAL]. No exceptions.
Ambiguous requirement text	Rewrite. Remove words like 'appropriate', 'reasonable', 'as needed'. Replace with specific, measurable conditions.
CRITICAL-CRITICAL conflict detected	Two CRITICAL requirements cannot both be satisfied. One must be downgraded to REQUIRED, or one must be rewritten so they are compatible. This cannot be resolved automatically.
traceability_complete fails	A CRITICAL or REQUIRED requirement has no test mapped to it. Add a test. Map it in the traceability matrix. Regenerate if needed.
Untraced function in implementation	Either trace the function to a requirement by adding a REQ-ID annotation, or remove the function. Untraced functions are not governed by the specification.

Where to Go Next

Document	What It Contains
SDPF Language Specification v1.3.1	The normative authority. Defines every rule this manual describes. Read when you need the exact formal definition of any requirement.
SDPF Language Specification v1.3.1 — Addendum A	Stabilisation patches: TEST-ID format (TEST-SYNTAX-1 through TEST-SYNTAX-4), normative Annex B with all eleven VCR-INV check identifiers, foreword correction.
SDPF Styles Playbook v2.1	Detailed guidance on all 17 styles. Read when selecting a style or applying style-specific additional sections.
SDPF Architect Thinking v2.2	How an experienced SDPF practitioner thinks, decides, and works. Read when you are past the basics and want to develop mastery.
SDPF vNext Comprehensive Reference v2	The architecture for the governed execution runtime. Read when you are ready to move beyond manual discipline to automated runtime enforcement.

SDPF Bounded Stochasticity Theorem	The formal justification for why SDPF works. Read when you need to understand or explain the theoretical foundation.

Problem first. Specification second. Facts before execution. Verification always.

Hamza Abdullah · Software Development Prompting Framework · 2026