SDPF

Architect Thinking

How a master practitioner thinks, works, and delivers with SDPF

Hamza Abdullah

Software Development Prompting Framework · Version 2.2 · 2026

I The Architect Who Uses SDPF for Everything

An architect who uses SDPF for every project does not simply become more organised. They become a different kind of architect — one who thinks in contracts, works with AI at a level most practitioners cannot match, and produces auditable, traceable software as a matter of course rather than as an afterthought.

This document describes how that architect thinks. It is not a tutorial on the SDPF tools. It is a description of the mental model, the disciplines, the habits, and the professional posture that characterise genuine mastery of the framework.

The core shift is this: most architects think about what the system will do. A SDPF architect thinks about what the system must do, exactly, before anyone touches a keyboard. That single shift changes everything downstream.

RULE *Specification first. Code second. Verification always. This is not a preference. It is the invariant that governs every decision.*

1 Contract-First Thinking

The Fundamental Mental Shift

The most important thing to understand about SDPF architect thinking is that it is not primarily about tools or processes. It is about the order in which thinking happens.

A conventional architect begins with the system. They sketch components, define services, draw data flows. Requirements emerge through the design process — they are discovered during implementation rather than specified before it. This is the natural creative approach, and it produces systems that work. It also produces systems where nobody can clearly articulate what requirement any given piece of code satisfies, because the requirements were never formally established before the code existed.

A SDPF architect begins with the contract. Before any component is designed, before any service is named, before any data flow is sketched, the contract is defined. What does this system accept as input? What does it do with that input? What does it return? What happens when it fails? These questions must be answered completely and precisely before any design work begins.

The Six Contract Questions

Every SDPF architect, approaching any system, asks these six questions in order. They do not proceed to design until all six are answered completely.

Question	What It Establishes
What must this system accept as input, exactly?	The input contract — field names, types, constraints, validation rules, required vs optional
What must it do with that input, exactly?	The processing rules — deterministic logic, no ambiguity, no implicit behaviour
What must it return, exactly?	The output contract — response shapes, status codes, field definitions
What must happen when it fails, exactly?	The exception model — every error state named, coded, and shaped
Is this regulated, compliance-driven, or safety-critical?	The style selection — governs verification requirements and evidence expectations
What are the hard constraints?	The constraint model — performance limits, resource limits, regulatory clauses

A question that cannot be answered completely is not a gap in the architect's knowledge — it is a gap in the requirements. The SDPF architect's response to an unanswerable contract question is not to make an assumption and proceed. It is to stop, identify who can answer it, get the answer, and only then proceed. This discipline prevents the ambiguity that causes AI outputs to drift, implementations to miss requirements, and projects to fail.

Precision in Language

The quality of a SDPF specification is almost entirely determined by the precision of its language. Two architects can follow the same process, use the same tool, and select the same style — and one will produce a specification that generates reliable AI output on the first submission, while the other will produce a specification that requires three rounds of correction.

The difference is precision. Consider the difference between these two requirement statements:

WEAK *Validate the input and return an appropriate error if something is wrong.*

STRONG *[CRITICAL] Validate that username is a non-empty string between 3 and 50 characters containing only alphanumeric characters and underscores. Reject all other values with HTTP 422 and error code INVALID_USERNAME, returning { "error": { "code": "INVALID_USERNAME", "message": string, "field": "username" } }.*

The weak requirement is technically a requirement. It tells a developer something must happen. It tells an AI nothing useful. The AI must invent what "appropriate" means, what "something is wrong" covers, what the error looks like. Every AI will invent something different. None of them will invent exactly what the architect intended.

The strong requirement leaves nothing to invention. The AI executes it mechanically. The result is correct on the first submission because there was nothing to misinterpret.

A SDPF architect writes every requirement at this level of precision. It takes longer to write. It takes far less time to implement correctly.

2 Mastery of the 17 Styles

Style Is Not Cosmetic

A SDPF practitioner who does not master the 17 styles is like a carpenter who only owns a hammer. The framework functions — but not optimally. A master selects the right style for the right problem instinctively, and understands what changes across the entire run as a result: requirement emphasis, test generation strategy, the expert framing the AI receives, verification rules, and evidence expectations.

The Style Selection Decision

Style selection is the first architectural decision on every project. It is made before the specification is written, because the style governs how the specification is structured and what the AI is told about its role.

The SDPF architect answers these questions in order to select the style:

- Are hardware constraints or resource limits the primary design driver? → Style 8 (Constraint-Based)
- Is the system regulated, compliance-driven, or security-critical? → Style 10 (Compliance-Driven)
- Are the requirements unknown and must emerge through exploration? → Style 2 (Exploratory Problem-Solving)
- Will requirements change frequently and predictably? → Style 5 (Iterative Refinement)
- Is the primary deliverable a user interface or interactive experience? → Style 3 (User Experience-Centered)
- Is the system a library or component others will depend on? → Style 7 (Test-Driven Specification)
- Is the system distributed across multiple services or teams? → Style 6 (System Architecture)
- Is the goal to improve or preserve existing working code? → Style 9 (Maintenance / Refactor)
- Is creativity, novelty, or generative behaviour the primary goal? → Style 4 (Creative Scaffolding)
- None of the above — deterministic system with defined I/O? → Style 1 (Technical Specification)

The extended seven styles (11–17) address advanced structural, sovereign, and domain-specific systems. A master knows when to reach for them and is not intimidated by their scope.

What Style Changes in Practice

Dimension	How Style Governs It
Requirement emphasis	Style 10 requires regulatory mapping on every CRITICAL requirement. Style 8 requires hard constraint thresholds. Style 7 requires test definitions before any implementation.
AI preamble	The exported prompt frames the AI differently per style. Style 12 tells the AI it is a ledger systems expert where any mutation is a specification violation. Style 14 tells it escalation rules are bound to runtime state.
Semantic policy checks	Each style has keyword rules checked against the specification. Style 6 checks for topology and interface language. Style 2 checks for hypothesis and abort criteria.
Verification focus	Style 10 checks audit mapping completeness. Style 7 checks that tests precede implementation. Style 15 checks invariance and closure at all boundaries.
Evidence expectations	Compliance, constraint, and documentation profiles produce different evidence shapes. Style 10 evidence includes regulatory mapping. Style 16 evidence requires documentation artifacts.

3 AI-Assisted Development Mastery

The Core Insight

The insight that separates a SDPF master from everyone else in the field of AI-assisted development is this: the bottleneck is not the model. It has never been the model. The bottleneck is the specification.

When an AI produces poor output, the instinct is to blame the model — it hallucinated, it misunderstood, it chose the wrong approach. In almost every case the real cause is that the prompt was ambiguous, incomplete, or underspecified. The model filled the gaps with its best guess, and the guess did not match what the developer intended.

SDPF eliminates the gaps. A complete, locked specification — with every input defined, every processing rule explicit, every output guaranteed, every exception named — leaves nothing for the model to guess. Any capable model executes it correctly on the first submission. This is not a claim about model capability. It is a claim about specification quality.

KEY INSIGHT

A SDPF master's specifications are so complete and unambiguous that the AI has nothing to misinterpret. The quality of the output is determined before the prompt is submitted.

The Prompt Export Discipline

A SDPF architect does not manually write prompts for AI models. The specification is the prompt. The Prompt Export feature packages the locked specification with a style-specific expert preamble and submits it to any model.

The preamble matters. A Style 10 specification submitted with the compliance engineer preamble produces different output than the same specification submitted with a generic developer preamble — because the AI is told what role to inhabit before it begins. A SDPF master understands this and ensures the style selection and preamble are appropriate for the system being built.

The prompt works identically on Claude, GPT-4o, Gemini, Mistral, Llama, and every other capable model. This is not an accident — it is a direct consequence of specification completeness. The model is not the variable. The specification is the constant. When the constant is correct, every model gives a correct answer.

Multi-Model Strategy

A SDPF master does not commit to a single AI model for a project. They submit the same locked specification to two or three models, capture the results, and compare them objectively using the Model Comparison feature.

This produces data that most teams argue about subjectively: which model is better for this type of system? The SDPF architect answers it empirically — here are the requirement coverage percentages, here are the output lengths, here are which models covered all CRITICAL requirements. The decision is made on evidence, not preference.

Over many projects a SDPF architect builds a personal data set of model performance by project type and style. They know from experience which models perform best for compliance-driven systems, which perform best for constraint-based systems, and which are most reliable for complex hybrid architectures. This knowledge compounds with every project.

The Coverage Check Discipline

When an AI output is captured, SDPF checks which REQ-IDs were referenced in the output. A SDPF master understands what this check means and what it does not mean.

It means the AI acknowledged and traced the requirements. It does not mean the implementation is correct. A SDPF master always reviews AI-generated code before treating it as production-ready. The evidence package proves process discipline. Engineering judgment proves correctness. Both are required. Neither replaces the other.

4 The Evidence Discipline

Evidence as a Natural Output

For most developers, documentation and audit trails are obligations — things produced reluctantly when someone demands them. For a SDPF architect, evidence is a natural output of the development process, as routine as committing code.

Every verified run produces a signed JSON evidence package automatically. The architect exports it, archives it, and moves on. There is no extra work. The evidence is a by-product of following the process correctly.

After five years of using SDPF for every project, a SDPF architect has a complete, searchable, signed archive of every system they have ever built — what was specified, how it was verified, which AI model produced the output, and what the traceability looked like. This archive is a professional asset of genuine value.

What the Evidence Proves

Evidence Proves	Evidence Does Not Prove
Specification existed before any code was written	That the implementation logic is functionally correct
Tests were generated from and locked against the specification	That the AI-generated code has no bugs
Implementation was generated after tests were locked	That the requirements themselves were the right requirements
All 11 verification checks passed at a specific timestamp	That edge cases not in the specification were handled
Every requirement traces to at least one test	That the system performs correctly under production load
Specific AI models were used and their outputs were checked	That security vulnerabilities were addressed beyond what was specified
The package has not been modified since it was signed	

A SDPF master is honest about this distinction with clients and stakeholders. The evidence proves process discipline, which is what most audits and compliance frameworks require. It does not prove correctness, which is the architect's ongoing professional responsibility.

The Signature and What It Means

Every evidence package is signed with HMAC-SHA256. The signature is computed over the entire canonical payload. If any field is changed after signing — a requirement reworded, a check result altered, an AI result added retroactively — the signature fails verification.

This means the evidence package is not just a record of what happened. It is tamper-evident proof of what happened. A client, auditor, or regulator who receives the evidence package can verify its integrity independently using the procedure documented inside the package itself.

A SDPF architect manages the signing key with the same discipline they apply to any production credential — stored securely, rotated periodically, never committed to source control, tracked in the key rotation registry. An evidence package signed with a compromised key is not reliable evidence. The integrity of the archive depends on the integrity of the key management.

5 The Template Library as Professional Capital

Why the Template Library Matters

Every SDPF specification a master produces is better than the last. Requirements are more precise. Edge cases are more completely covered. Error models are more thoroughly defined. Style contexts are more tightly scoped. This improvement compounds — but only if it is captured.

The prompt library is how a SDPF architect captures and compounds their specification quality. Every time they complete a project and produce a specification that worked well — that generated reliable AI output, that passed verification cleanly, that satisfied the client — they save it as a template. The template embodies everything they learned on that project.

After ten projects a SDPF architect's template library is a knowledge base of proven specification patterns. After twenty it is a competitive advantage. A new project in a domain they have worked in before begins with a template that is 70% complete — representing everything they have learned about how to specify that type of system precisely. The remaining 30% is where the real project-specific thinking happens.

Built-In Templates as a Starting Point

SDPF Desktop Studio ships with six built-in templates covering the most common system types:

- REST API — User Directory: CRUD operations, HTTP status codes, error contracts
- Python Desktop Application: Tkinter UI, persistent state, atomic writes, background threading
- Progressive Web Application: Service worker, offline capability, WCAG 2.1 AA, installability
- CLI Tool: Subcommands, argument parsing, exit codes, stdin/stdout/stderr contracts
- Data Pipeline: Schema contracts at each stage, dead-letter handling, idempotency
- Event-Driven Microservice: Topic contracts, idempotency, dead-letter topics, circuit breakers

A SDPF master treats these as starting points, not endpoints. They load a template, improve it for the specific project, complete the run, and if the result was good, save the improved version as a custom template. The library grows with every project.

Template Quality Indicators

A well-crafted template exhibits these characteristics:

- Every requirement is tagged [CRITICAL], [REQUIRED], or [OPTIONAL] — nothing is untagged

- Every error state is named, coded, and shaped — no generic "return an error" requirements
- The external contract is precise — protocol, format, version, authentication method
- The style context is specific to the domain — not "API" but "payment processing API, PCI-DSS scope, all write operations audited"
- The exception handling covers not just the happy path errors but edge cases — rate limits, malformed inputs, downstream failures
- The output guarantees specify exact shapes — not "return the user" but "return { id: UUID, username: string, email: string, created_at: ISO-8601 }"

6 How the SDPF Architect Works With Teams

The Cultural Effect

An architect who uses SDPF for every project gradually shifts the culture of the teams around them without mandating it. The shift happens because the experience of working with complete specifications is so different from the experience of working with vague requirements that developers notice and adapt.

Developers stop being asked to implement ambiguous requirements and start receiving complete contracts. They spend less time asking clarifying questions and less time reworking code because the requirements changed or were misunderstood. They begin to associate specification completeness with smooth delivery. They start asking for specifications before they begin work.

QA stops chasing requirements after the fact. The locked test suite exists before implementation begins. The traceability matrix shows exactly which tests cover which requirements. Coverage gaps are visible before code is written, not discovered after testing.

Project managers stop wondering whether the implementation matches what the client asked for. The traceability matrix answers that question continuously throughout the project. The evidence package answers it definitively at the end.

Consistency Across Projects

Every project a SDPF architect delivers has the same structural fingerprint. The specification format is the same. The requirement tagging is the same. The traceability structure is the same. The evidence package format is the same.

This consistency is not about uniformity for its own sake. It is about reducing cognitive load. A developer who has worked on one SDPF project can pick up any other SDPF project and immediately understand its structure. They know where to find the requirements. They know what the traceability matrix shows. They know how to read the evidence package. They can be productive immediately.

This consistency also makes onboarding dramatically faster. A new team member who receives the evidence package for a system they are being asked to maintain has everything they need to understand what the system was supposed to do, how it was verified, and what the implementation traces back to.

The Shared Template Library

When a SDPF architect leads a team, the template library becomes a team asset. Approved specification templates for the organisation's common system types are shared across all projects. Every new project starts from a template that embodies the team's accumulated specification knowledge, not from scratch.

New team members learn what good specifications look like by reading the templates. Senior members improve the templates as they encounter new edge cases or learn better ways to

specify common patterns. The library becomes the organisation's institutional knowledge of how to build software correctly.

7 The Ten Principles of SDPF Architect Thinking

These are the principles that characterise a SDPF architect's thinking, distilled from the methodology and from practice. They are not rules imposed from outside. They are the natural conclusions of someone who has internalised the framework and seen what it produces.

1

Specify completely before doing anything else

No design, no implementation, no AI submission until the contract is fully defined. Ambiguity is not a starting point — it is a problem to be solved before work begins.

2

Assert nothing you have not verified

Every technical fact in a specification — version numbers, API endpoints, flag behaviours — must be verified against live output before implementation begins. The TVG is not overhead. It is prevention.

3

Select the style that fits the problem

The wrong style produces the wrong emphasis and the wrong AI framing. Style selection is the first architectural decision, not an afterthought.

4

Write requirements that leave nothing to inference

If the AI or a developer can interpret a requirement differently from what you intended, the requirement is not finished. Rewrite it until only one interpretation is possible.

5

Lock before proceeding — always

Specification locked before tests. Tests locked before implementation. This order is not a suggestion. It is the mechanism that prevents implementation-first drift.

6

Submit to multiple models and compare objectively

Model preference is a hypothesis. Model comparison with the same locked specification is evidence. Decide on evidence.

7

Capture evidence for every run

The evidence package is not documentation created after delivery. It is a by-product of following the process correctly. Export it every time without exception.

8

Build and refine the template library continuously

Every project that produces a good specification should contribute a template. The library is compounding knowledge. Neglecting it wastes the learning.

9

Review AI output — the evidence proves process, not correctness

The evidence package proves that SDPF controls were followed. It does not prove the implementation is correct. Engineering judgment is always required.

10

Know where the framework's responsibility ends

SDPF is the most rigorous practical methodology for AI-assisted development. It is not a substitute for domain expertise, security review, or production testing. Use it fully, and augment it appropriately.

8 What Mastery Looks Like in Practice

The First Hour of a New Project

A SDPF master begins every project the same way. They open SDPF Desktop Studio. They go to the Prompt Library and check whether a template exists for this type of system. If it does, they load it, review it, and adjust it for the specific project context. If it does not, they open the Help tab, read the sample specification, and use it as a structural reference while writing from scratch.

They select the style before they write a single requirement. They read the style use case and differentiator to confirm the selection is right. If they are uncertain between two styles, they apply the decision matrix until one answer is clear.

They write the specification with the precision of someone who knows the AI will execute it literally. Every requirement is tagged. Every error state is named. Every output is shaped. They do not submit until the specification would give an auditor — or an AI — nothing to guess about.

The Moment of Submission

When the specification is locked, a SDPF master opens the Prompt Export tab and generates the prompt. They review the preamble to confirm it matches the expert framing appropriate for the selected style. They check the token estimate to ensure it fits the target model's context window.

They submit to at least two models. They capture both results. They review the coverage detail for each — not just the summary percentage but the specific requirements that were and were not referenced. They make a judgment about which output is the better foundation, informed by the comparison data and their own review of the code.

This entire workflow — from locked specification to captured, compared AI outputs — takes minutes. Not hours. The hours were spent on the specification. The minutes are spent on everything else.

The Evidence as the Closing Deliverable

When verification passes, a SDPF master exports the evidence package before closing the project. This is not an optional step. The evidence package is a deliverable — to the client, to the archive, to their future self who will maintain this system.

If the project was for a client in a regulated industry, the evidence package goes to the client explicitly as a professional deliverable. The client receives: a signed specification showing what was agreed, a verification record showing it was checked, a traceability matrix showing every requirement was covered, and an AI results record showing which model produced the implementation and what it covered. Most clients have never received anything like this. It establishes the architect as operating at a level of professional discipline that is genuinely uncommon.

THE RESULT

An architect who has operated this way for five years has a signed, searchable archive of every system they have ever built, a template library refined through real projects, objective data on AI model performance by domain, and a professional reputation for delivering exactly what was specified — with proof.

The Architect SDPF Produces

The SDPF architect is not defined by the tools they use. They are defined by the thinking the tools enforce and ultimately internalise. The specification discipline, the contract-first instinct, the precision of language, the evidence habit, the template library — these become how they think, not just what they do.

When that thinking is applied to every project, consistently, over years, the compound effect is a professional who operates at a level that is genuinely difficult for a conventional architect to replicate. Not because they are smarter or more technically capable, but because they have eliminated the ambiguity, drift, and undocumented decisions that accumulate in every project that does not follow this discipline.

The work is traceable. The decisions are documented. The AI output is captured and attributed. The evidence is signed. Every project builds on the last. The template library grows. The model performance data accumulates. The professional reputation compounds.

This is what SDPF architect thinking produces. It starts, as all mastery does, with a single project done correctly.

Specification first. Code second. Verification always.

SDPF — Software Development Prompting Framework · Hamza Abdullah · 2026