SDPF

Styles Playbook

A Practical Guide to Selecting and Using
the 17 SDPF Prompt Styles

Hamza Abdullah | Based on SDPF v2.1

Quick Reference: Style Selection Matrix

Answer: what does your project primarily involve? Match to the style below.

If your project involves…	Choose Style
REST APIs, data pipelines, deterministic systems	#1 Technical Specification
Novel algorithms, research, optimization problems	#2 Exploratory Problem-Solving
UI, interactive apps, frontend systems	#3 User Experience-Centered
Games, generative apps, content tools	#4 Creative Scaffolding
Agile work, prototypes, evolving requirements	#5 Iterative Refinement
Distributed systems, microservices, scalable services	#6 System Architecture
Libraries, APIs, critical infrastructure	#7 Test-Driven Specification
Embedded systems, performance-critical code	#8 Constraint-Based
Legacy code, technical debt reduction	#9 Maintenance / Refactor
Regulated domains, security-critical systems	#10 Compliance-Driven
Universal recursive computation engines	#11 Meta-Hypercomputer
Audit ledgers, chain-of-custody systems	#12 Sovereign Structural Ledger
Multi-module, multi-team architectures	#13 Hybrid System Composition
Risk-aware, safety-critical systems	#14 Dynamic Criticality Extension
Proof pipelines, structural verification	#15 Cross-System Verification
Automated docs, onboarding systems	#16 Prompt-Driven Documentation
Clinical, legal, financial, infrastructure domains	#17 Domain-Specific Template

Detailed Playbook

#1 Technical Specification

Best for: APIs, data pipelines, deterministic systems	*Key differentiator: Contract-first with complete input/output behaviour definitions*

When to Use

- Building REST/gRPC APIs
- Creating data transformation pipelines
- Any system where inputs → outputs must be completely predictable

How to Apply

1	Define EXTERNAL CONTRACT first (interface type, data format, versioning)
2	Write INPUT CONTRACTS with strict schemas (JSON Schema or equivalent)
3	Specify PROCESSING RULES with explicit logic — no ambiguity
4	Define OUTPUT CONTRACTS for success and error scenarios
5	Create VERIFICATION REQUIREMENTS mapping to specific tests

Example Sections

- EXTERNAL CONTRACT: REST API, JSON, v1
- INPUT: UserInput: { username: string, email: string }
- PROCESSING: If username exists → reject (HTTP 400)
- OUTPUT: POST → HTTP 201 + user object

Required Elements

+ Complete input/output schema definitions

+ Error model with structured JSON format

+ Traceability matrix mapping REQ → code → TEST

#2 Exploratory Problem-Solving

Best for: Novel algorithms, research, optimization	*Key differentiator: Explicit unknown handling, exploratory hypotheses, and abort criteria*

When to Use

- Solving problems without known solutions
- Algorithm research and development
- Optimisation problems where the answer is unknown

How to Apply

1	Define the PROBLEM SPACE without assuming a solution exists
2	List EXPLORATORY HYPOTHESES to test
3	Specify ABORT CRITERIA (when to stop searching)
4	Define SUCCESS METRICS for evaluation
5	Outline BOUNDS on the search space

Required Elements

+ Hypothesis list with testable conditions
+ Time/resource limits
+ Fallback strategies if exploration fails

#3 User Experience-Centered

Best for: UI, interactive apps, frontend systems	*Key differentiator: User journey, usability, and accessibility defined before implementation*

When to Use

- Web or mobile applications
- Interactive dashboards
- Any system where user interaction is primary

How to Apply

1	Map USER JOURNEYS end-to-end
2	Define ACCESSIBILITY requirements (WCAG level, screen readers)
3	Specify USABILITY METRICS (task completion time, error rates)
4	List EDGE CASES for user behaviour
5	Define ERROR STATES and recovery flows

Required Elements

+ User journey diagrams
+ Accessibility conformance targets
+ Usability benchmarks

#4 Creative Scaffolding

Best for: Games, generative apps, content tools

Key differentiator: Innovation space is bounded before operational constraints are locked

When to Use

- Game development
- Generative AI applications
- Creative tools and content creation systems

How to Apply

1	Define INNOVATION BOUNDS (what can be creative)
2	Specify OPERATIONAL CONSTRAINTS (hard limits)
3	List CREATIVE FREEDOM ZONES
4	Define BOUNDARY BEHAVIOURS (what happens at edges)
5	Set OUTPUT VALIDATION criteria

Required Elements

+ Clear boundaries between creative and constrained areas
+ Validation of outputs within bounds

#5 Iterative Refinement

Best for: Agile work, prototypes, evolving requirements

Key differentiator: Change tolerance and revision paths are contract-level controls

When to Use

- MVP development
- Projects with changing requirements
- Sprint-based development

How to Apply

1	Define CHANGE CLASSIFICATION (BREAKING, SIGNIFICANT, MINOR)
2	Specify REVISION PROCEDURES for each change type
3	List VERSIONING strategy
4	Define ROLLBACK procedures
5	Set CHANGE IMPACT assessment rules

Required Elements

+ Change request template
+ Impact analysis procedure
+ Revision history requirements

#6 System Architecture

Best for: Distributed systems, scalable services

Key differentiator: Topology, boundaries, and interface contracts are explicit

When to Use

- Microservices architecture
- Cloud-native systems
- Multi-node distributed systems

How to Apply

1	Define SYSTEM TOPOLOGY (components, relationships)
2	Specify INTERFACE CONTRACTS between services
3	Define BOUNDARY RULES (what crosses service lines)
4	List SCALABILITY requirements
5	Specify FAULT BOUNDARY behaviours

Required Elements

+ Architecture diagram (logical)
+ Service interface contracts
+ Failure mode analysis

#7 Test-Driven Specification

Best for: Libraries, APIs, critical infrastructure

Key differentiator: Tests are derived and locked before implementation generation

When to Use

- Building reusable libraries
- API development for external consumers
- Safety-critical systems

How to Apply

1	Write ALL TESTS first (before any implementation code)
2	Define ACCEPTANCE CRITERIA for each test
3	Lock TEST SUITE before implementation begins
4	Implement to pass locked tests
5	Add IMPLEMENTATION-SPECIFIC tests after

Required Elements

+ Complete test suite
+ Test execution requirements
+ Pass criteria before implementation

#8 Constraint-Based

Best for: Embedded systems, performance-critical code

Key differentiator: Hard constraints are primary requirements and verification criteria

When to Use

- Real-time systems
- Resource-constrained environments
- Performance-sensitive applications

How to Apply

1	List ALL HARD CONSTRAINTS (memory, CPU, latency, etc.)
2	Define CONSTRAINT PRIORITIES
3	Specify MEASUREMENT methods for each constraint
4	Define VIOLATION HANDLING
5	Set CONSTRAINT VERIFICATION procedure

Required Elements

+ Constraint specification with measurable thresholds
+ Verification test suite for constraints
+ Violation handling procedures

#9 Maintenance / Refactor

Best for: Legacy code, technical debt reduction

Key differentiator: Preserve-behaviour commitments are mandatory

When to Use

- Updating existing systems
- Reducing technical debt
- Modernising legacy applications

How to Apply

1	Document CURRENT BEHAVIOUR (what must be preserved)
2	Define PRESERVE-BEHAVIOURS list (critical)
3	Specify BEHAVIOUR VERIFICATION tests
4	Define SCOPE of refactoring
5	Create ROLLBACK procedures

Required Elements

+ Behaviour preservation requirements

+ Regression test suite

+ Incremental change procedure

#10 Compliance-Driven

Best for: Regulated domains, security-critical systems

Key differentiator: Regulatory mapping and audit evidence are built-in outputs

When to Use

- Healthcare systems (HIPAA)
- Financial systems (SOX, PCI-DSS)
- Government systems

How to Apply

1	Identify APPLICABLE REGULATIONS
2	Map REQUIREMENTS to regulatory clauses
3	Define AUDIT TRAIL requirements
4	Specify COMPLIANCE EVIDENCE needed
5	Set VERIFICATION and reporting procedures

Required Elements

+ Regulation mapping table

+ Evidence generation requirements

+ Audit trail specifications

#11 Meta-Hypercomputer Specification

Best for: Universal recursive computation engines

Key differentiator: Operator families, closure, and absorption constraints are explicit

When to Use

- Recursive computation systems
- Self-modifying systems
- Hypercomputation research

How to Apply

1. Define OPERATOR FAMILIES
2. Specify CLOSURE properties
3. Define ABSORPTION constraints
4. List RECURSION BOUNDS
5. Set VERIFICATION criteria for infinite processes

Required Elements

+ Operator specification

+ Termination/finiteness proofs

+ Resource bounds

#12 Sovereign Structural Ledger

Best for: Immutable audit ledgers, chain-of-custody systems

Key differentiator: Tamper-evidence and provenance are CRITICAL requirements

When to Use

- Audit logging systems
- Chain-of-custody tracking
- Blockchain-like ledgers

How to Apply

1	Define LEDGER STRUCTURE
2	Specify IMMUTABILITY mechanisms
3	Define PROVENANCE tracking
4	Set TAMPER EVIDENCE requirements
5	Define VERIFICATION procedures

Required Elements

+ Cryptographic integrity verification
+ Audit trail requirements
+ Tamper detection mechanisms

#13 Hybrid System Composition

Best for: Multi-module, multi-team architectures

Key differentiator: Inter-prompt dependencies and boundary rules are enforced

When to Use

- Large-scale systems with multiple teams
- Systems requiring module integration
- Platform engineering

How to Apply

1	Define MODULE BOUNDARIES
2	Specify INTER-MODULE CONTRACTS
3	Define DEPENDENCY RULES
4	Set INTEGRATION VERIFICATION procedures
5	Define CROSS-MODULE CHANGE procedures

Required Elements

+ Module interface specifications
+ Dependency graph
+ Integration test suite

#14 Dynamic Criticality Extension

Best for: Risk-aware, safety-critical systems

Key differentiator: Escalation behaviour is bound to runtime state changes

When to Use

- Systems with varying criticality levels
- Adaptive safety systems
- Risk-managed applications

How to Apply

1	Define CRITICALITY LEVELS
2	Specify STATE TRIGGERS for escalation
3	Define ESCALATION BEHAVIOURS
4	Set DE-ESCALATION procedures
5	Define MONITORING requirements

Required Elements

+ State-to-criticality mapping

+ Escalation logic specification

+ Runtime monitoring requirements

#15 Cross-System Verification

Best for: Proof pipelines, structural verification

Key differentiator: Invariance and closure checks are required at all boundaries

When to Use

- Formal verification systems
- Proof-generating pipelines
- Multi-system validation

How to Apply

1	Define INVARIANTS to maintain
2	Specify BOUNDARY CHECK requirements
3	Define CLOSURE properties
4	Set VERIFICATION FREQUENCY
5	Define FAILURE HANDLING

Required Elements

+ Invariant specifications

+ Boundary verification procedures

+ Proof generation requirements

#16 Prompt-Driven Documentation

Best for: Automated docs, onboarding systems

Key differentiator: Documentation is a CRITICAL deliverable and verification output

When to Use

- Auto-generating documentation
- Onboarding systems
- Self-documenting codebases

How to Apply

1	Define DOCUMENTATION STRUCTURE
2	Specify REQUIRED SECTIONS
3	Define DOCUMENTATION VERIFICATION
4	Set GENERATION TRIGGERS
5	Define REVIEW procedures

Required Elements

+ Documentation template

+ Generation automation

+ Review workflow

#17 Domain-Specific Template

Best for: Clinical, legal, financial, infrastructure domains

Key differentiator: Jurisdiction and compliance mappings are embedded in the contract

When to Use

- Industry-specific applications
- Regulated domain systems
- Specialised knowledge systems

How to Apply

1	Identify TARGET DOMAIN
2	Define JURISDICTION requirements
3	Specify COMPLIANCE MAPPINGS
4	Define DOMAIN-SPECIFIC vocabulary
5	Set VERIFICATION procedures for domain rules

Required Elements

+ Domain vocabulary specification

+ Jurisdiction mapping

+ Compliance requirement matrix

General Process: Applying Any SDPF Style

Regardless of which style is selected, every SDPF project follows the same nine-step process. The style determines what goes inside the specification — the process is invariant.

Step 1 — Select Style

Use the Quick Reference Matrix. Ask: what type of system am I building? What are the critical concerns? Who are the stakeholders?

Step 2 — Create the Technical Specification Prompt (TSP)

For ALL styles, create a TSP with: EXTERNAL CONTRACT · REQUIREMENTS (with IDs) · INPUT CONTRACT · PROCESSING RULES · OUTPUT CONTRACTS · ERROR MODEL · STATE MODEL · VERIFICATION REQUIREMENTS.

Step 3 — Follow Style-Specific Requirements

Apply the applicable style's required elements as documented in the playbook above.

Step 4 — Lock Specification

Once the TSP is complete, lock it. No changes without change control. Classify any subsequent changes as BREAKING, SIGNIFICANT, or MINOR.

Step 5 — Generate Tests

Generate the complete test suite based on the locked specification. Tests must be derived from requirements — not from code.

Step 6 — Lock Tests

Lock the test suite before any implementation begins. Tests written after implementation are not SDPF-compliant.

Step 7 — Implement

Write code to pass the locked tests. Every function references a requirement ID. No code is written that does not trace to a specification requirement.

Step 8 — Verify

Run the verification gate. All five gates must pass in order: TVG → Unit → Integration → Clean-Machine → Closure Record.

Step 9 — Export Evidence

Generate the audit trail: Prompt + Traceability Matrix + Verification Closure Record. This is the complete, auditable chain from intent to output.

Common Mistakes to Avoid

1	**Wrong Style Selection**	Choosing matrix que
2	**Incomplete Specifications**	Skipping th section is
3	**Locking Too Early**	Implement spec will re
4	**Locking Too Late**	Changing change mu
5	**Missing Traceability**	Not mappi code.
6	**Ignoring the Verification Gate**	Skipping a Record mu
7	**No Evidence Export**	Failing to p cannot be

Quick Start Checklist

Use this checklist for every SDPF project. Every item must be completed before a system can be considered fully SDPF-compliant.

- [ ] **Identify project type**
  Select the appropriate SDPF style using the Quick Reference Matrix
- [ ] **Read the style section**
  Review the applicable style entry in this playbook
- [ ] **Create the TSP**
  Write the Technical Specification Prompt with all required sections
- [ ] **Add style-specific elements**
  Include the required elements specific to your chosen style
- [ ] **Lock specification**
  Freeze the spec and apply change control to all subsequent modifications
- [ ] **Generate tests**
  Derive the complete test suite from the locked specification
- [ ] **Lock tests**
  Freeze the test suite before writing any implementation code
- [ ] **Implement code**
  Write code to pass locked tests, annotating each function with its requirement ID
- [ ] **Run verification gate**
  Execute all five gates: TVG → Unit → Integration → Clean-Machine → Closure
- [ ] **Export evidence package**
  Produce Prompt + Traceability Matrix + Verification Closure Record
- [ ] **Verify signature on evidence**
  Confirm CLOSURE STATUS = COMPLETE before shipment

Specification first. Code second. Verification always.