Unified Specification-Driven Development Guide

Component-First, AI-Assisted, Technology-Agnostic

Draft: v0.1.2

Hugo O'Connor, Anuna Research

Introduction: The Paradigm Shift

Specification-Driven Development (SDD) inverts the traditional relationship between specifications and code. Specifications are the primary, executable artifacts that generate working systems; code is their expression. This shift is enabled by AI's ability to understand and implement rigorous specifications while architectural discipline is maintained through constitutional principles.

This guide unifies proven practices from software architecture, requirements engineering, and AI-assisted development into a coherent framework. It incorporates a component-first presentation strategy: specifications define composition using a prebuilt component library and design tokens, enabling fast, consistent, and accessible interfaces with minimal bespoke UI.

Scope and Definitions

Specification

The normative, machine-processable description of behavior, contracts, non-functionals, and presentation mapping.

Implementation Plan

The structured translation of a specification into architecture, data, contracts, and tests.

Generated Code

One possible expression of the specification; human-written code may coexist when justified.

Normative Language

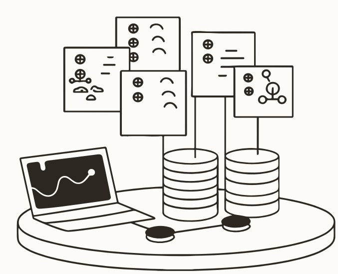
MUST, SHOULD, and MAY are used as in RFC 2119.

What SDD Is and Isn't

What SDD IS

A methodology where specifications are the primary artifact and are validated, versioned, and traced end-to-end.

What SDD ISN'T



A specific tool or CLI. Teams integrate SDD concepts into their existing tools, pipelines, and repositories.

Part I: Core Methodology

The foundation of Specification-Driven Development rests on four key pillars that transform how we approach software development. This methodology represents a fundamental shift from traditional code-first approaches to a specification-centric paradigm.

The SDD Philosophy

Specifications as Lingua Franca

Specifications are the source of truth. Maintaining software means evolving specifications. Code, tests, and docs are generated expressions of that truth.

Power Inversion

Specifications drive generation; code does not drive design. This reduces the gap between intent and implementation.

Intent-Driven Development

Teams express intent in natural language, models, diagrams, and design assets. AI performs mechanical transformation into working systems, with human review.

Composition Over Customization

Presentation is composed from existing, reusable components governed by design tokens. New custom UI is exceptional and governed.

The Constitutional Foundation

An organizational "constitution" governs how specifications become systems. It enforces consistency, quality, and maintainability across implementations.

Core Articles (adapt to your context):

01

Modularity Mandate

Features are composed of cohesive, reusable components.

02

Interface Imperative

All functionality is exposed via well-defined interfaces and contracts.

03

Test-First Principle

Every requirement is verifiable; tests are produced before or alongside implementation.

04

Simplicity Gate

Complexity is capped; deviations require explicit justification.

05

Integration-First Testing

Prefer realistic environments to mocks where feasible.

06

Security by Design

Specify controls early with verifiable criteria.

Constitutional Foundation (Continued)

01

Observability Requirement

Systems MUST be monitorable, diagnosable, and auditable.

02

Composition-First UI

Use existing component libraries and tokens; new components require governance.

03

Accessibility Baseline

Interfaces meet defined standards (e.g., WCAG 2.2 AA) by default.

04

Traceability

Every requirement links to contracts, code, tests, and telemetry.

05

Versioned Change

Specifications, models, and generators are version-controlled with review.

06

AI Accountability

AI assistance is logged, reviewed, and bounded by trust policies.

AI Trust Boundaries (Normative)

- Record for each AI-assisted change: model/version, prompts or templates, parameters, inputs, outputs, reviewer, and decision.
- Define "no-go" areas (e.g., cryptography, auth core, privacy-sensitive transforms) where AI generation requires explicit approval.
- Require deterministic or bounded-variance generation for critical artifacts (e.g., contracts).

The SDD Workflow

Phase 1: Vision to Specification

- Iterative requirements discovery via structured dialogue
- Ambiguity resolution through measurable criteria and examples
- Stakeholder validation and conflict resolution
- Quality assurance against specification standards

Phase 2: Specification to Implementation Plan

- Architecture analysis and component identification
- Presentation mapping to existing components and tokens
- Contract and schema design (APIs, events, data models)
- Testing strategy and acceptance criteria
- Security, performance, privacy, and observability considerations

Phase 3: Plan to Code

- Code generation and/or implementation conforming to architectural and composition principles
- Automated tests derived from acceptance and quality criteria
- Documentation generated from specifications
- Continuous validation against constitutional gates

Phase 4: Feedback and Evolution

- Production metrics and user insights inform specification updates
- Non-functional issues become explicit requirements
- Incidents drive security and reliability enhancements
- Controlled iteration with governance and traceability

Part II: Requirements Specification Excellence

Quality specifications are the cornerstone of successful SDD implementation. This section outlines 24 distinct quality attributes that distinguish exceptional specifications from merely adequate ones.

> "The quality of a specification directly determines the quality of the resulting system. Excellence in specification writing is not optional—it's foundational."

Essential Quality Attributes

Clarity and Content

1. Unambiguous

Single interpretation possible

2. Correct

Reflects actual needs and intent

3. Complete

No missing functionality or constraints

4. Understandable

Clear to all stakeholders

5. Achievable

Feasible within constraints

6. Concise

No unnecessary detail

7. Design-Independent

States what, not how (except constitutional constraints)

8. Prioritized

Importance and urgency captured

Precision and Testability

9. Verifiable

Objectively testable

10. Precise

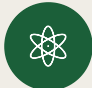
Quantified metrics and exact language

11. Consistent (Internal)

No internal conflicts

12. Consistent (External)

Aligns with policies, standards, and related docs

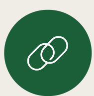


13. Atomic

Each requirement expresses one obligation

14. Non-Redundant

No duplication



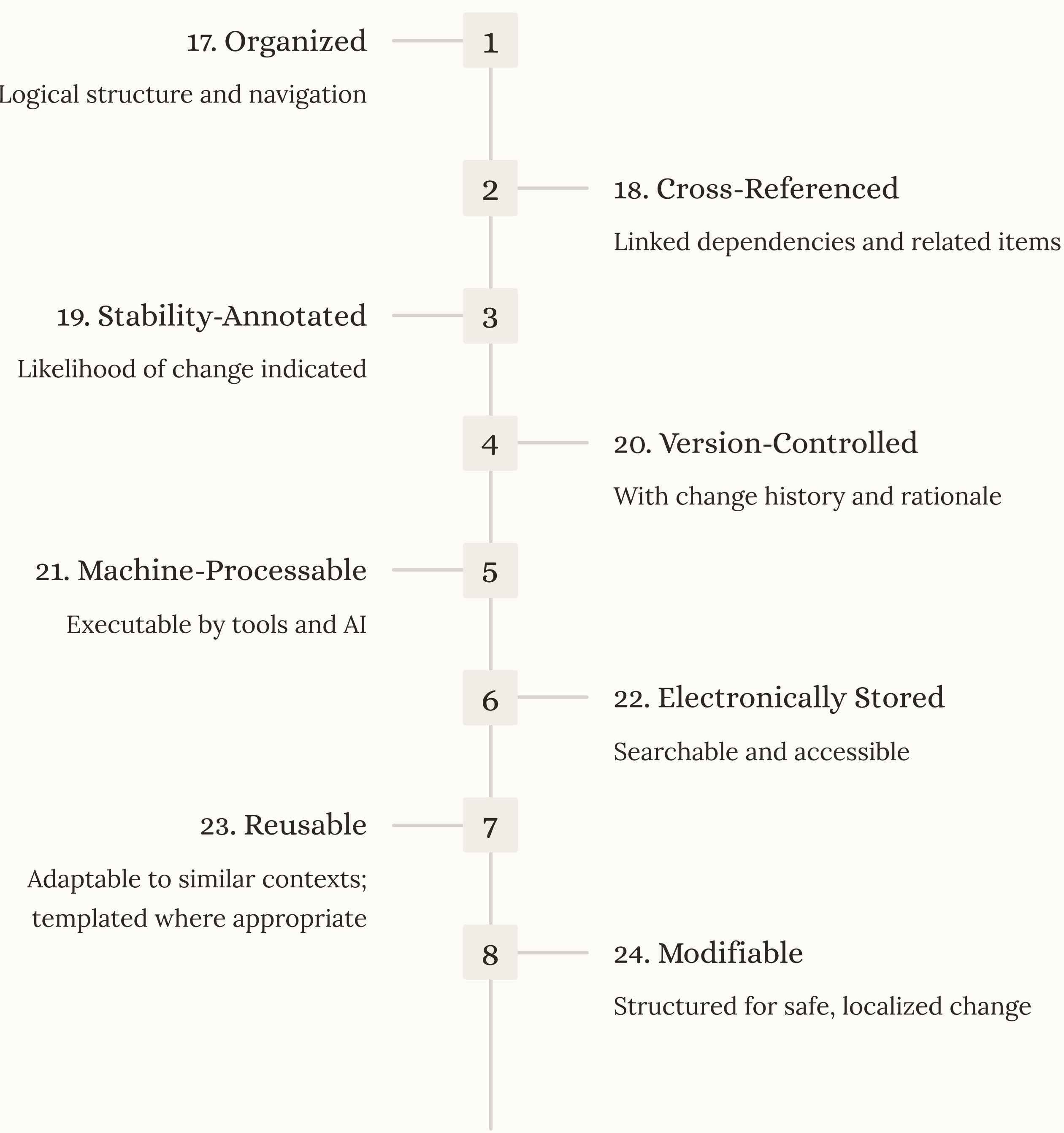
15. Traceable End-to-End

Linked to sources, contracts, tests, and telemetry

16. Error-Aware

Includes failure modes and expected behaviors

Governance and Lifecycle



IDs and Traceability Conventions

- Use stable IDs:

REQ-###, NFR-###, CON-### (contracts), ADR-###, TEST-###, OBS-### (telemetry)

- Every REQ links to at least one TEST and, post-release, to at least one OBS metric or log/tracing signal

Systematic Requirements Discovery

Requirements discovery is a structured, iterative process that ensures comprehensive coverage of all system needs. This systematic approach prevents critical gaps and reduces costly rework.

Phase 1: Context Discovery

- Primary problems, outcomes, and stakeholders
- Business processes and value flows
- Integration requirements and constraints
- Budget, timeline, and risk appetite

Phase 2: Functional Requirements

- Core capabilities, workflows, user interactions
- Data processing and transformations
- Interfaces and contracts (internal/external)
- Reporting, analytics, audit needs

Requirements Discovery (Continued)

Phase 3: Non-Functional Requirements

- Performance (latency, throughput, capacity)
- Scalability (growth, elasticity, limits)
- Security and compliance (policies, controls, attestations)
- Reliability and availability (SLOs, RTO/RPO)
- Usability and accessibility (standards, success criteria)
- Maintainability and operability (support, diagnostics)
- Privacy and data handling (classification, retention, minimization)

Phase 4: Domain-Specific Requirements

- Standards, regulations, certifications
- Domain vocabulary and workflows
- Toolchain and ecosystem integrations

Presentation (Component-First) Addendum

Component Mapping

Map each UI need to an existing component/variant; justify exceptions.

Design Tokens

Define brand colors, typography, spacing, radii, elevations; forbid hard-coded literals in code.

Component States

Default, loading, error, disabled; hover/focus/pressed where applicable.

Accessibility

Roles, labels, keyboard/focus behavior, contrast targets; commit to WCAG 2.2 AA.

Content Constraints

Length limits, localization, truncation/wrapping rules.

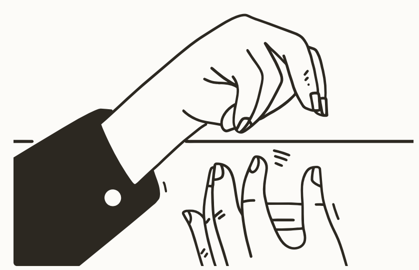
Ambiguity Resolution Techniques

Common Sources

- Vague quantifiers ("fast", "easy", "secure")
- Implicit assumptions and undefined terms
- Pronouns without clear antecedents
- Incomplete conditionals or exception handling

Resolution Strategies

- Replace adjectives with measurable criteria
- Provide examples and counter-examples
- Use structured templates for consistency
- Validate via paraphrasing and scenario walkthroughs



Template Patterns

Functional: The system SHALL [action] [object] [condition] WITHIN [timeframe] FOR [user role] WITH [success criteria].

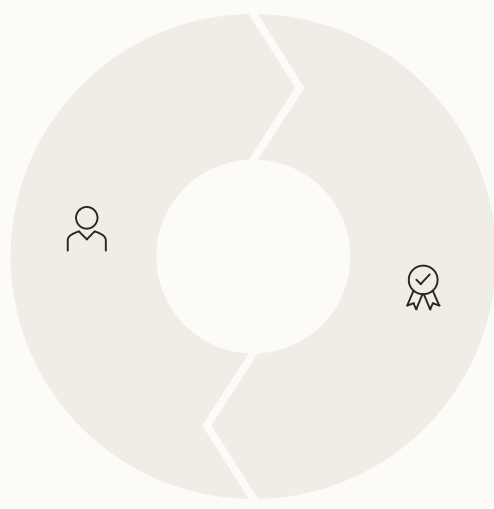
Non-Functional: [Attribute] SHALL be [metric] UNDER [conditions] WITH [confidence/percentile].

Presentation Mapping: The interface SHALL use [ComponentName] variant=[name] size=[name] tokens=[list] content=[spec] states=[list] accessibility=[spec].

Validation and Quality Assurance

Multi-Perspective Validation

- Stakeholder matrix: Validate per role and concern
- Conflict resolution: Document trade-offs and decisions
- Binary validation: Agree/Disagree with rationale
- Completeness checklists: Functional, non-functional, presentation, security, privacy



Quality Gates (per requirement)

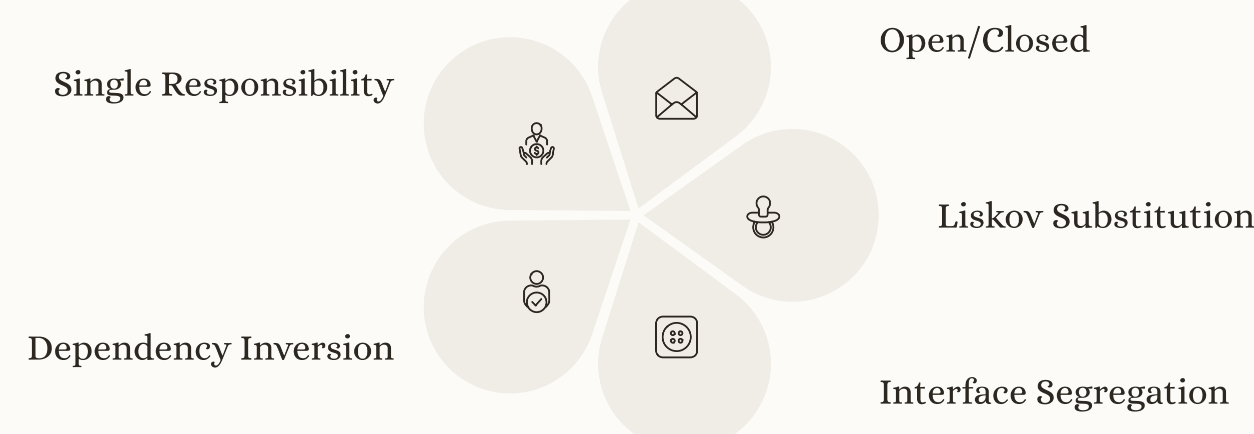
- Testable? Necessary? Non-conflicting? Feasible?
- Meets applicable quality attributes and has trace links?

Part III: Architectural Excellence

Architectural excellence provides the structural foundation that enables specifications to become robust, maintainable systems. This section establishes the principles and frameworks that guide architectural decisions within the SDD methodology.

> "Architecture is the art of how to waste space beautifully. In software, it's the art of how to organize complexity elegantly."

Core Architectural Principles



Design System Integration (Presentation)

Component Library as Platform

Compose UIs from known building blocks.

Tokenization

Style via tokens; no hard-coded presentation.

Variant Discipline

Prefer variants and composition to bespoke components.

Accessibility by Default

Semantics and interactions built-in.

Architectural Decision Framework

01

Context Analysis

Problem, constraints, drivers

02

Alternatives

Feasible options with evidence

03

Trade-offs

Impact on quality attributes

04

Decision Rationale

Why this choice?

05

Consequences

Risks and mitigations

06

Review

Stakeholder validation

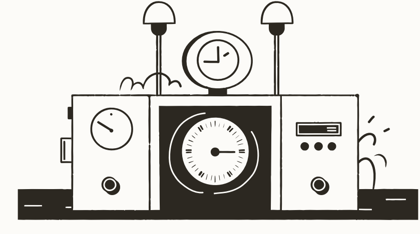
Architecture Decision Records (ADRs)

Title, Status, Context, Decision, Consequences, Alternatives, Links (REQ/NFR/CON/TEST)

Quality Attributes Framework

Performance Architecture

- Scalability strategies (horizontal/vertical)
- Latency and throughput optimization
- Resource efficiency



Design Considerations

- Caching and invalidation; load distribution
- Async and streaming patterns
- Data indexing and access paths
- Efficient asset delivery

Reliability Architecture

Core Principles

- Availability targets; graceful degradation
- Recovery strategies and backups
- Monitoring and alerting

Design Considerations

- Redundancy and failover
- Circuit breakers and bulkheads
- Self-healing and auto-remediation
- Chaos testing

Security Architecture

1	Threat Modeling
2	Data Protection
3	Authorization
4	Authentication

Security architecture encompasses authentication and identity assurance, authorization and least privilege, data protection (in transit/at rest), and comprehensive threat modeling and mitigation.

Design Considerations

- Secure defaults and fail-safe design
- Input validation and output encoding
- Secrets management and rotation
- Audit trails, tamper evidence, and SBOM targets

Observability Standards

SLIs/SLOs per Capability

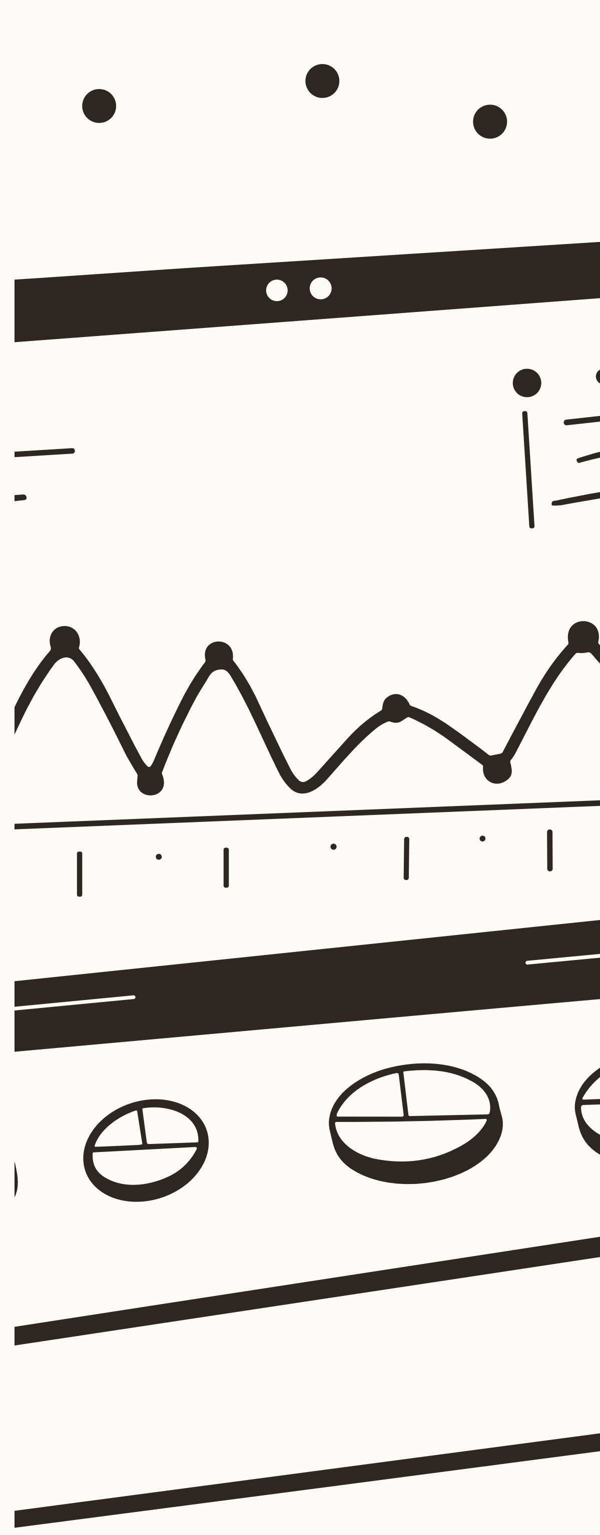
Error budget policy with clear service level indicators and objectives for each system capability.

Required Signals

Metrics, structured logs, and traces with correlation IDs for comprehensive system visibility.

Standard Attributes

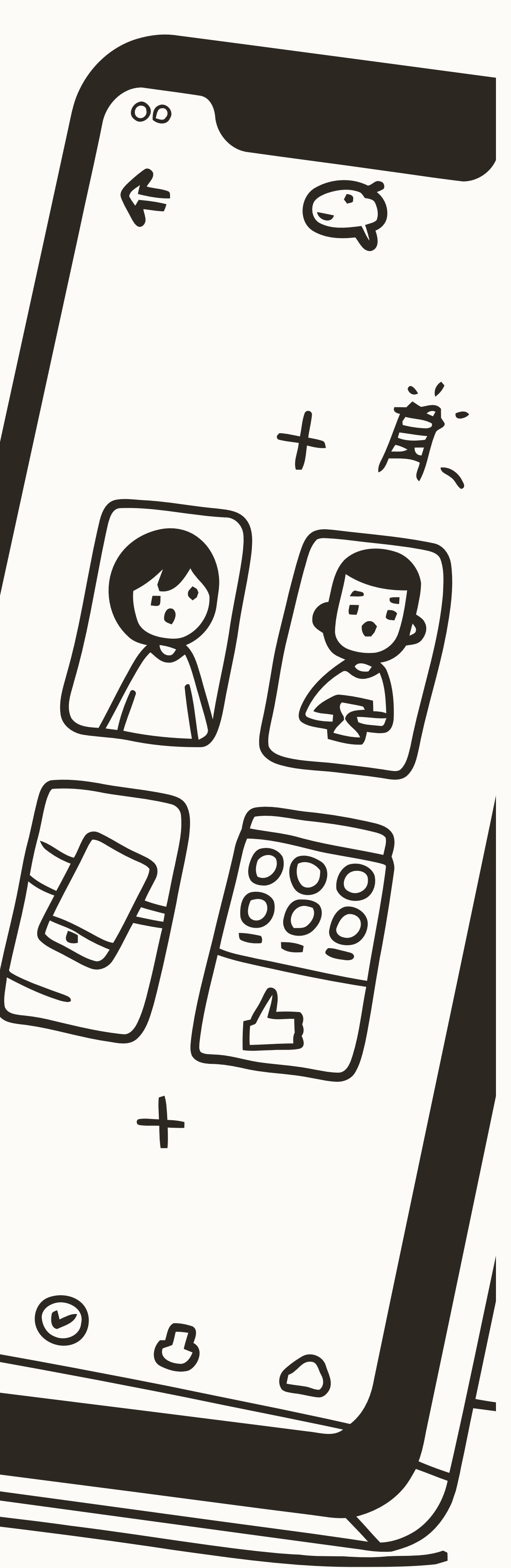
Consistent attributes like trace_id, user_id hash, request_class across all observability signals.



Part IV: Implementation Templates

Implementation templates provide standardized structures that ensure consistency and completeness across all development efforts. These templates embody SDD principles and constitutional requirements.

> "Templates are not constraints—they are accelerators. They capture institutional knowledge and prevent reinventing the wheel."



Feature Specification Template

Feature Overview

Feature Name: [kebab-case-name]
Purpose: [Business value in one sentence]
User Story: As a [user role], I want [goal] so that [benefit].
Business KPI Impact: [metric + target]
Telemetry Spec (SLIs): [name, unit, target, percentile]

Acceptance Criteria:
- [ ] [Specific, testable condition]
- [ ] [Measurable success metric]
- [ ] [Error and empty-state behavior]
- [ ] [A11y success criteria]

Data Classification: [e.g., Public/Internal/PII/PHI]
Privacy Notes: [collection, minimization, retention]

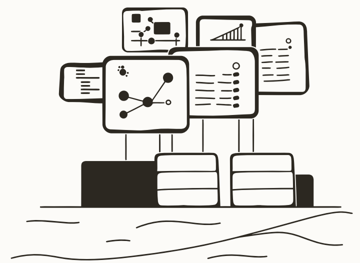
Requirements Analysis Template

Functional Requirements

- REQ-###: [Detailed requirement]
- REQ-###: [Detailed requirement]

Non-Functional Requirements

- NFR-###: Performance — [metric, threshold, condition]
- NFR-###: Security — [control, criteria, verification]
- NFR-###: Reliability — [SLO/SLA and guarantees]



Constraints

- Technical: [limits, dependencies]
- Business: [policies, approvals]
- Operational: [support, schedules]

Architecture Analysis Template

1

Affected Components

- Data: [entities, schemas, migrations]
- Services: [business logic, integrations]
- Contracts: [endpoints/interfaces, validation]
- Presentation: [screens/views, component composition]

2

New Components

[Name]: [purpose, responsibility, public interface]

3

Dependencies

- Internal: [modules/services]
- External: [providers/adapters]

4

Compatibility & Versioning

[contract versions, deprecation window, migration path]

Technical Implementation Plan Template

System Design

Architecture Decisions

[Decision]: [Rationale, trade-offs]
[Decision]: [Alternatives considered]

Runtime Topology

[services, data stores, queues, regions]

Data Model

[entities, relationships, constraints, retention]

Contracts

[interface definitions, formats, pre/post-conditions, error model]

Security Considerations

- Authentication: [method and assurance level]
- Authorization: [policy and scope]
- Data Protection: [classification, encryption, retention]

Resiliency Budget

[error budget, retry/backoff, timeouts, idempotency]

Implementation Phases

Phase 1: Foundation

- Data model and migrations
- Core services and contracts
- Unit tests and static analysis

Phase 2: Business Logic

- Rules and workflows
- Integration tests
- Error handling and recovery

Phase 3: Interfaces

- Contract implementation
- Request/response validation
- Contract tests (consumer/provider)

Phase 4: Presentation (Component-First)

- Screen specs (component mapping + tokens)
- Composition using existing components
- Accessibility verification
- End-to-end tests

Testing Strategy

1. End-to-End
2. Contract Tests
3. Integration Tests
4. Unit Tests

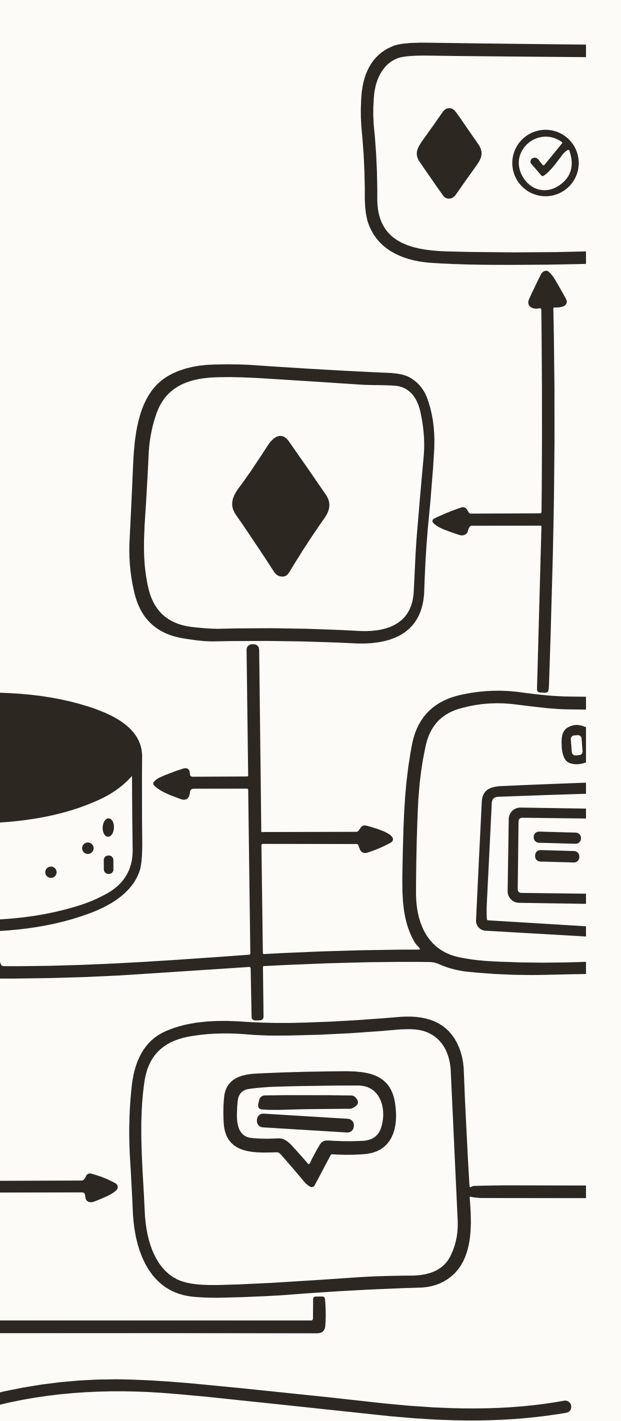
Additional Testing

- Property-based tests (core logic)
- Fuzzing (inputs/parsers)
- Mutation testing budget
- Test data management (synthetic, PII-safe)

Test Scenarios

- Happy path; edge cases and boundary conditions
- Failure modes and recovery
- Performance and load
- Accessibility (presentation)

Presentation (Component-First) Specification Templates



Screen Specification

```
Screen: [Name]
Identifier: [route-or-id]

Components (existing library only):
- [ComponentName]
 - variant: [name]
 - size: [name]
 - props: [key=value, ...]
 - content: [text/slots]
 - state(s): [default|loading|error|disabled]
 - automationId: [identifier]
 - accessibility: [role|label|keyboard behavior]

Layout:
- [Composition using layout components]

Validation (if inputs):
- [FieldName]: [rule, message]

Navigation:
- onSuccess: [destination]
- onBack: [destination]
```

Design Tokens and Annotations

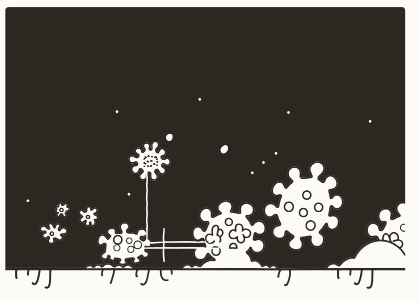
Design Tokens

```
Colors:
- primary|accent|destructive|neutral:
  [token refs only; no literals]

Typography:
- fontFamily: [name]
- scale: [xs/sm/md/lg/...]
- lineHeights: [values]

Spacing and Radii:
- spacing: [scale]
- radius: [scale]

Shadows/Elevation (if applicable):
- [levels and semantics]
```



Design-File Annotations

Layer naming examples:

- Button.primary.lg
- Input.outline
- ProgressIndicator

Required metadata:

- component, variant, size, content, state
- automationId (stable; not derived from labels)
- accessibility (label/role/hints)

Constitutional Gates Template

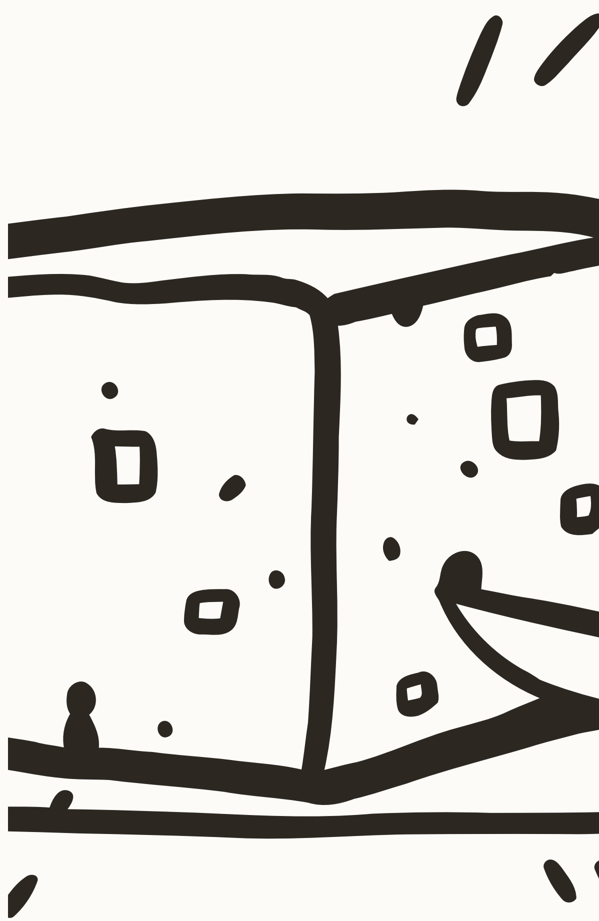
Each gate MUST record status and evidence (artifacts, links, metrics).

1	**Simplicity Gate** • Complexity within threshold (e.g., avg cyclomatic ≤ N; public API size ≤ M) • No premature optimization (ADR required for exceptions) • Minimal dependencies (new deps justified) • Clear separation of concerns (no layer violations)
2	**Quality Gate** • Requirements are testable; tests authored or generated • Security & privacy addressed (threat model, data classification) • Performance budgets defined (e.g., p95 latency ≤ X under Y RPS) • Error handling specified (timeouts/retries/idempotency) • Observability requirements captured (metrics, logs, traces)
3	**Architecture Gate** • Adheres to patterns and interfaces; arch-lint clean • Composable and extensible; boundaries respected • ADRs updated and linked • Dependencies justified; SBOM updated
4	**Presentation Composition Gate** • Uses approved components; exceptions approved • Tokens applied; no hard-coded styles • States and accessibility verified (0 critical A11y violations) • Layer names and metadata conform
5	**Delivery Gate** • Acceptance criteria traceable to tests (REQ ↔️ TEST) • Coverage plan complete (critical path ≥ threshold) • Deployment, rollback, and progressive rollout defined • Monitoring and alerting in place; runbook linked

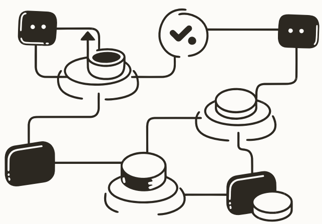
Part V: Advanced Practices

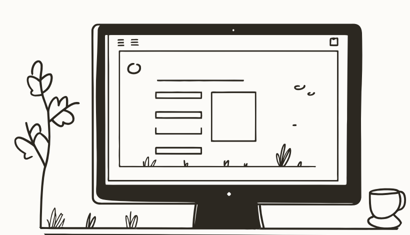
Advanced practices extend the core SDD methodology with sophisticated techniques for automation, evolution management, and multi-implementation strategies. These practices enable organizations to scale SDD across complex, enterprise-level initiatives.

> "Advanced practices are not about complexity—they're about sophisticated simplicity that scales."



Automated Specification Assistance





AI-Enhanced Discovery and Validation

- Semantic analysis: Extract implicit requirements and assumptions
- Completeness checking: Compare specs to domain models and checklists
- Consistency validation: Detect contradictions and ambiguities
- Quality assessment: Score against the attribute framework

Template-Driven Generation

- Structured prompts/templates: Enforce format and terminology
- Quality constraints: Auto-apply gates and thresholds
- Traceability: Link requirements → tests → implementation → telemetry
- Version control: Track evolution and rationale for AI-assisted changes

Specification Evolution Management

Change Management Process

1. Impact analysis across components, contracts, and screens
2. Stakeholder review and approval
3. Update propagation to dependent specifications
4. Regenerate or revise implementation and tests
5. Validation testing and release

Continuous Improvement

- Feedback integration from operations and users
- Pattern extraction and reuse
- Quality metrics for specification health
- Process refinement and tool automation

Drift Detection

- Contract diffs (e.g., OpenAPI change budgets)
- Component usage audits and token reference checks
- Golden tests for critical behaviors

Multi-Implementation Strategies

Parallel Implementations

- A/B testing using the same specification
- Platform variations with consistent contracts
- Performance-tuned variants
- Risk mitigation via alternate approaches

Evolutionary Architectures

- Incremental modernization
- Strangler-style replacement
- Feature toggles and staged rollouts
- Backward compatibility and migration paths

Part VI: Quality Assurance and Governance

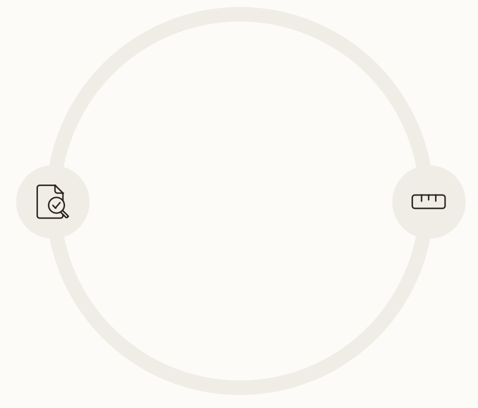
Quality assurance and governance provide the oversight mechanisms that ensure SDD implementations maintain high standards and organizational alignment. This comprehensive framework balances automation with human judgment.

> "Governance is not about control—it's about enabling excellence at scale while maintaining coherence."

Specification Quality Control

Continuous Validation

- Automated checks and linting (structure, links, policies)
- Peer review and collaborative refinement
- Stakeholder sign-off at milestones
- Traceability audits from requirement to runtime signals



Quality Metrics

- Completeness score; clarity/ambiguity index
- Consistency rating; testability score
- A11y conformance (presentation); drift incidents
- Error-budget consumption and SLO attainment

Architectural Governance

Standards Enforcement

- Constitutional compliance
- Pattern and interface consistency
- ADR discipline and transparency
- Stack and dependency coherence (technology-agnostic criteria)

Evolution Management

- Change control and approvals
- Impact assessment and risk mitigation
- Migration planning and sunset criteria
- Legacy interoperability and containment

Implementation Assurance & Conclusion

Code Generation and Implementation Quality

- Template verification and drift detection
- Output quality checks (style, structure, security)
- Coverage thresholds and critical path tests
- Security scanning and policy enforcement; SBOM

Delivery Pipeline Integration

- Automated processing of executable specifications within your existing pipeline
- Quality gates aligned with constitutional checks
- Release readiness verification
- Monitoring and observability as first-class deliverables

> ### A Stable Framework for an AI-Accelerated Future
>
> SDD treats specifications as executable assets that generate systems. By coupling rigorous specification quality with a component-first presentation strategy and design tokens, teams achieve speed, consistency, accessibility, and maintainability—without defaulting to bespoke UI.
>
> This unified, technology-agnostic methodology brings together requirements engineering, architecture, and AI-assisted generation. The result is faster cycles, higher quality, and stronger alignment with business intent. Maintain discipline around specification quality, enforce constitutional principles and trust boundaries, and let AI transform clear intent into reliable implementations.

References

1. Ali, S. W., Ahmed, Q. A. & Shafi, I. Process to enhance the quality of software requirement specification document. in 2018 International Conference on Engineering and Emerging Technologies (ICEET) 1–7 (IEEE, Lahore, Pakistan, 2018). doi:10.1109/ICEET1.2018.8338619.
2. Anuar, U., Ahmad, S. & Emran, N. A. A simplified systematic literature review: Improving Software Requirements Specification quality with boilerplates. in 2015 9th Malaysian Software Engineering Conference (MySEC) 99–105 (IEEE, Kuala Lumpur, Malaysia, 2015). doi:10.1109/MySEC.2015.7475203.
3. Davis, A. et al. Identifying and measuring quality in a software requirements specification. in [1993] Proceedings First International Software Metrics Symposium 141–152 (IEEE Comput. Soc. Press, Baltimore, MD, USA, 1993). doi:10.1109/METRIC.1993.263792.
4. Giakoumakis, E. A. & Xylomenos, G. Evaluation and selection criteria for software requirements specification standards. Softw. Eng. J. UK 11, 307 (1996).
5. Nicolás, J. & Toval, A. On the generation of requirements specifications from software engineering models: A systematic literature review. Information and Software Technology 51, 1291–1307 (2009).
6. Osman, M. H. & Zaharin, M. F. Ambiguous Software Requirement Specification Detection: An Automated Approach. (2018).
7. **spec-kit/spec-driven.md at main · github/spec-kit · GitHub**.

Appendix: LLM Operating Guide for Specification-Driven Development (SDD)

This appendix provides a comprehensive and unambiguous instruction set for Large Language Models (LLMs) that assist in Specification-Driven Development (SDD). Its primary purpose is to ensure LLM outputs are consistently aligned with the SDD constitution, presentation rules, and governance framework, while seamlessly integrating with existing repositories and pipelines without assuming a specific SDD Command Line Interface (CLI).

Key Operating Principles for LLMs

Source of Truth

The specification is normative; LLMs must not contradict it. If gaps are identified, the LLM should request clarification rather than inventing details.

Constitution Compliance

All outputs generated by LLMs must be rigorously checked against the established articles and gates of the SDD constitution.

Determinism

LLMs should prioritize low-variance outputs for critical artifacts such as contracts, schemas, and tests, surfacing any inherent non-determinism constraints.

Human-in-the-Loop

Uncertainties must be flagged by the LLM, with options and trade-offs proposed. The LLM must never silently alter the scope of a task.

AI Accountability

For significant decisions or outputs, the LLM should record its model, version, parameters, and a brief justification summary for auditability.

LLM Response Envelope

Every reply from an LLM assisting with SDD should utilize a standardized response envelope to ensure clarity, reviewability, and automation. This includes fields such as `task`, `status`, `summary`, `questions` (if applicable), `assumptions`, `artifacts`, `gate_results`, `risks`, `next_actions`, and `audit` details.

Core Workflows and Expected Outputs

LLMs are guided through four core workflows, each with defined actions, expected outputs, and mandatory gate checks:

01

1. Vision → Specification

LLMs elicit context, stakeholders, value flows, and constraints, converting intent into atomic requirements (REQ/NFR) with acceptance criteria. They also map presentation needs to existing components.

Output: `requirements.md`, `context.md`, `presentation-map.md`

02

2. Specification → Implementation Plan

The LLM identifies architecture, defines contracts, plans tests, and specifies budgets for security, privacy, performance, and observability.

Output: `plan.md`, `contracts/`, `testing-strategy.md`, `resiliency.md`

03

3. Plan → Code (Generation/Guidance)

The LLM maps the plan to code structure, generates or proposes tests first, and drafts documentation directly from specifications.

Output: Module layout proposals, test stubs, contract tests, doc stubs.

04

4. Feedback → Evolution

LLMs propose specification updates based on telemetry and incidents, run drift checks, and produce migration and deprecation notes.

Output: Spec diffs, ADR updates, `migration.md`, deprecation schedule.

LLM-Enforced Quality Attributes and ID Linkage

LLMs are instructed to enforce strict quality attributes for specifications, including clarity, precision, testability, and robust governance. A consistent ID and linkage system (e.g., REQ-###, CON-###, TEST-###) is mandated to ensure end-to-end traceability across all artifacts. LLMs must refuse to generate content in restricted areas without explicit approval and escalate when key inputs are missing or ambiguous, ensuring safe and compliant operations within the SDD framework.