Commitment Design System (CDS)

A system model for discovering meaning, refining intent, and formalizing commitments.

Viktor Jevdokimov, Vilnius, Lithuania

Table of contents

1. INTRODUCTION & QUICK START

1.1 CDF Quick Start – From Context to Practice

“CDF is not another delivery method — it’s the commitment layer that makes methods work.”

> ⚠️ **ATTENTION: You may be viewing a downloaded version.**
> The living, latest version of this documentation is always available online: CDS Official Documentation

A commitment-design system for turning **meaning** into **decision-grade intent** and **formal commitments** — without drifting into delivery theater.

> ⏱️ **30-Second Summary**
> The **Commitment Design System (CDS)** defines a simple lifecycle:
> **Meaning Discovery → Intent Refinement → Commitment Formalization**.
> It produces three canonical artifacts — **Meaning Handshake**, **Intent Package**, **Commitment Envelope** — plus quality checks and re-entry rules that prevent teams from “starting work” without a real commitment.

1.1.1 What Is CDS

CDS is a **commitment formation system**.

Where most frameworks focus on *how teams execute work*, CDS focuses on the step before execution becomes expensive:
making sure the commitment is real, inspectable, and governable.

CDS exists because many failures attributed to delivery (Agile, Scrum, “bad requirements”, “slow engineering”) are actually **upstream commitment problems**:

- meaning was never aligned
- intent was never decision-grade
- commitment was never formalized (accountability, change protocol, acceptance evidence)

CDS replaces “just start and iterate” with a more disciplined stance:

- **Discover meaning**
- **Refine intent**
- **Formalize commitment**
- then run delivery with fewer surprises and less escalation

1.1.2 Where CDS sits in the 3in3.dev stack

CDS is designed to connect two layers:

- **HCS (Human Cooperation System)** explains cooperation stability and breakdowns at a universal level.
- **3SF (3-in-3 SDLC Framework)** runs software delivery commitments across Client–Vendor–Product coherence.

CDS sits between them as the commitment-design layer:

- narrower than HCS (focused on a specific commitment)
- upstream of 3SF (prepares a commitment that delivery can actually run)

1.1.3 Who It's For

CDS is designed for practitioners who routinely translate between:

- business meaning ↔ product intent ↔ delivery commitments
- multiple stakeholders with different stakes and language
- uncertainty that cannot be "estimated away"

Typical roles:

- Product leaders and product-facing discovery practitioners
- Project / delivery leads (internal or vendor-side)
- Business analysts and solutioning practitioners (RFI/RFP)
- Engineering leaders who keep inheriting ambiguous commitments

1.1.4 What CDS Is — and What It Is Not

What CDS Is

CDS is a **system model and artifact discipline** for:

- discovering and aligning meaning
- shaping decision-grade intent
- formalizing commitments that can be governed and inspected over time

It is designed to prevent drift and escalation by making:

- tradeoffs explicit
- decision rights explicit
- change routable
- acceptance evidence explicit

What CDS Is Not

CDS is not:

- ❌ a delivery methodology (Scrum/Kanban replacement)
- ❌ a requirements format or ticketing standard
- ❌ a strategy framework
- ❌ a tool stack recommendation
- ❌ a substitute for leadership or decision-making

CDS ends when the **Commitment Envelope** is formalized.
Execution belongs to a runtime (e.g., 3SF, Agile/Kanban, or your org's delivery system).

1.1.5 The CDS Core Model (Meaning → Intent → Commitment)

CDS is intentionally simple:

- **Meaning Discovery**
  Align on conditions, needs, frictions, stakes, evidence, and uncertainty.
  Output: Meaning Handshake.
- **Intent Refinement**
  Convert meaning into decision-grade intent: outcomes, signals, boundaries, constraints, tradeoffs, assumptions, decision rights.
  Output: Intent Package.
- **Commitment Formalization**
  Freeze intent into a governable commitment: accountability, governance cadence, change protocol, acceptance evidence, revisit triggers.
  Output: Commitment Envelope.

1.1.6 CDS User Onboarding Checklist

> *Start here before you apply any delivery method.*

Step	Action	Why This Step Is Necessary
1. Orient	Read **Vision, Principles, and Beliefs**.	Aligns stance: CDS is a commitment system, not a method war.
2. Choose profile	Start with **Core Model**, then apply **Software Delivery Profile** only if your runtime is delivery.	Keeps CDS universal, profiles additive.
3. Discover meaning	Produce a **Meaning Handshake**.	Prevents solution-first commitments.
4. Refine intent	Produce an **Intent Package**.	Makes intent decision-grade: signals, boundaries, constraints, tradeoffs.
5. Formalize commitment	Produce a **Commitment Envelope**.	Makes commitment executable and governable.
6. Run quality checks	Use **Core Quality Checks** (Must / Should / Smells + re-entry rules).	Prevents "progress by assumption."
7. Hand off to runtime	Hand the Commitment Envelope into your runtime (e.g., 3SF).	CDS prepares the input contract; runtime executes.

1.1.7 How to Navigate CDS

CDS documentation is structured for fast, non-linear use:

Section	Purpose
Core Model	The universal Meaning → Intent → Commitment lifecycle, artifacts, and quality checks.
Software Delivery Profile	Adds delivery-grade fields and gates; maps Commitment Envelope into 3SF-style runtime coherence.
Reference	Glossary, schemas, sources, practices map, versioning/licensing.

1.1.8 Where to Start

- **Vision, Principles, and Beliefs**
  The stance and purpose of CDS.
- **Core Model**
  The overview page that links Meaning Discovery → Intent Refinement → Commitment Formalization.

- **Software Delivery Profile (if applicable)**
  Start with the overview, then use the mapping and stage extensions.

1.1.9 When You’re Ready

If you want to apply CDS immediately, start with:

- **Meaning Handshake (template)**
- **Intent Package (template)**
- **Commitment Envelope (template)**

Then use **Core Quality Checks** to decide whether you can progress — or should re-enter an earlier stage.

> *CDF begins with meaning, not method. Make the commitment real — then choose how to run it.*

2. CORE MODEL

2.1 Vision, Principles, and Beliefs

The **Commitment Design System (CDS)** defines the layer of discipline that turns **meaning** into **decision-grade intent**, and then into **formal commitments** that can be executed responsibly. CDS sits between broad cooperation dynamics (HCS) and execution runtimes (like 3SF), narrowing "how we work together" into "what we are actually committing to do."

CDS exists because most breakdowns in complex work are not implementation failures — they are **commitment failures**: unclear intent, hidden tradeoffs, ambiguous decision rights, and commitments formed under pressure rather than understanding.

CDS makes the upstream commitment work explicit — so execution doesn't become the place where meaning and intent are discovered too late.

2.1.1 Vision

To help individuals, teams, and organizations **form commitments intentionally** — grounded in shared meaning, explicit tradeoffs, and clear decision responsibility.

> **Vision Statement**
> *A world where commitments are designed with clarity — not negotiated through confusion.*

2.1.2 Principles

These principles describe the "physics" of commitment design: how CDS maintains coherence as work moves from meaning to intent to commitment.

Principle	Description	Anti-Pattern (Violation Signal)
Meaning before Intent	Intent must be rooted in the real conditions, needs, and frictions of the people and system involved.	"We already know what to build" before meaning is aligned.
Intent before Commitment	No commitment is valid unless intent is decision-grade: bounded, testable, and tradeoff-aware.	"Let's commit and figure it out later" (and rework becomes the plan).
Clarity before Speed	Speed without shared understanding produces waste downstream.	"We don't have time to align" followed by weeks of rework.
Evidence over Interpretation	Commitments should be anchored in signals and observable outcomes, not assumptions or narratives.	"Everyone agrees" but nobody can point to evidence.
Tradeoffs must be explicit	Every commitment sacrifices something; CDS makes the sacrifice visible and owned.	Hidden tradeoffs later reappear as conflict ("why is this slow/ugly/ manual?").
Decision rights are part of the commitment	A commitment without decision ownership is a future escalation.	"We agreed" but nobody can approve changes or say "no."
Transparency enables coherence under change	Commitments remain stable when reality shifts only if information is shared and symmetrical.	Selective reporting, surprises, and late constraint discovery.
Reversibility is explicit	Commitments should state what is reversible, costly to reverse, and effectively irreversible — and what triggers reconsideration.	"We didn't know this was irreversible" after the system is already locked in.
Learning closes the loop	Under uncertainty, commitments must include learning gates and revisit triggers.	Blame-driven postmortems instead of system learning.

These principles intentionally echo the foundation beneath HCS and the delivery principles in 3SF, but focus specifically on the **formation of commitments**.

2.1.3 Beliefs

CDS is grounded in a set of beliefs about how complex work succeeds or fails:

1. **Commitments are socio-technical contracts.**
   They bind people, money, risk, and time — not just scope.
2. **Most delivery problems are upstream commitment problems.**
   Execution usually fails “correctly” against unclear or unstable intent.
3. **Shared meaning is a prerequisite for sustainable commitment.**
   When meaning is not aligned, commitments become political artifacts.
4. **Intent is a state, not a document.**
   It must be refined through explicit decisions, not captured once and archived.
5. **Uncertainty is normal; hidden uncertainty is expensive.**
   CDS does not eliminate uncertainty — it makes it governable through learning gates and revisit triggers.
6. **Commitments should remain inspectable over time.**
   Future teams should be able to reconstruct why a commitment was reasonable at the time — its conditions, tradeoffs, and decision rights.
7. **Disagreement is a signal, not a blockage.**
   When meaning is contested, CDS treats disagreement as input to refine intent — not something to suppress in order to “move faster.”

2.1.4 Scope and Non-Goals

CDS defines how commitments should be formed, not how work should be executed.

- **Not a delivery methodology**
  CDS does not replace Agile, Lean, or any execution approach. It designs the commitment that execution methods must serve.
- **Not a strategy framework**
  CDS can support strategy formation, but it focuses on the moment strategy becomes **commitment** (accountability + governance).
- **Not a replacement for cooperation systems**
  CDS assumes cooperation dynamics exist and may be supported by HCS; CDS narrows that cooperation into specific, formal commitments.
- **Not a legal contract template**
  CDS can be expressed through contracts (e.g., SoW/RFP outputs), but the system is about **coherent commitment formation**, not legal wording.
- **Not documentation-first**
  CDS uses lightweight artifacts, but its primary output is shared commitment clarity, not documents.

2.2 Core Model

The **Commitment Design System (CDS)** is a system model for **discovering meaning, refining intent, and formalizing commitments**.

CDS exists to narrow broad cooperation into a **specific, accountable commitment** — before execution begins. It focuses on the quality of upstream translation so delivery does not become the place where meaning and intent are discovered too late.

2.2.1 What CDS is

CDS is an upper-layer discipline for turning human reality into commitment-ready clarity:

- It helps teams **discover and align on meaning** (conditions, needs, frictions, stakes).
- It helps teams **refine intent** into a decision-grade form (outcomes, boundaries, constraints, tradeoffs, assumptions).
- It helps teams **formalize commitments** so execution can be governed, adapted, and evaluated responsibly.

CDS is compatible with different delivery approaches and operating models. It does not prescribe how work must be executed; it designs what execution is accountable to deliver.

2.2.2 What CDS is not

- **Not a delivery methodology** (it does not replace Agile/Lean/Scrum/Kanban).
- **Not requirements writing** (it is commitment formation, not ticket production).
- **Not a strategy framework** (it can support strategy, but focuses on when strategy becomes commitment).
- **Not a legal template** (it can be expressed via contracts, but is not contract language).
- **Not a substitute for cooperation systems** (it assumes cooperation dynamics exist and may be supported by broader systems such as HCS).

2.2.3 CDS lifecycle

CDS uses a simple lifecycle. These are steps, not phases of a project plan. Depending on stakes, they can be run lightly or deeply.

Step 1: Meaning Discovery

Discover and align on what is true, what is needed, and what is blocking progress — without jumping to solutions.

Step 2: Intent Refinement

Turn meaning into decision-grade intent by making outcomes, boundaries, constraints, tradeoffs, assumptions, evidence, and decision rights explicit.

Step 3: Commitment Formalization

Freeze refined intent into an accountable commitment with governance, change protocol, evidence expectations, and revisit triggers.

2.2.4 Core interfaces

CDS is designed to connect to other systems. Two interfaces matter most.

Meaning Handshake

The interface between Meaning Discovery and Intent Refinement.

Purpose: ensure meaning is captured in a usable, inspectable form — so intent refinement starts from shared reality rather than assumptions.

Primary output: Meaning Handshake artifact.

Commitment Envelope

The interface between Commitment Formalization and execution runtimes.

Purpose: provide a stable, explicit commitment object that delivery can execute and govern without repeatedly re-discovering intent.

Primary output: Commitment Envelope artifact.

2.2.5 Core artifacts

CDS keeps artifacts lightweight and inspectable. The canonical outputs are:

- **Meaning Handshake** (Meaning Discovery output)
- **Intent Package** (Intent Refinement output)
- **Commitment Envelope** (Commitment Formalization output)

Each lifecycle stage page defines:

- purpose and boundaries
- process and substages
- artifact schema
- core quality checks ("what good looks like")
- common failure patterns and recovery moves

2.2.6 Core roles

CDS is role-agnostic, but it requires these functions to be present (even if performed by the same person in small settings):

- **Sponsor** — holds the why, sets priorities, accepts tradeoffs.
- **Decider** — has authority to commit and resolve conflicts when needed.
- **Facilitator / Translator** — runs the CDS process, surfaces assumptions, maintains clarity.
- **Domain voices** — people closest to the work and constraints (business, technical, operational, risk/compliance as needed).
- **Delivery representative** — ensures intent is plausible and commitment is executable.

2.2.7 Core quality checks

CDS quality checks prevent premature commitment. At a high level:

- Meaning is sufficient when **conditions, needs, frictions, and stakes** are explicit and not mutually surprising.
- Intent is decision-grade when **outcomes, boundaries, constraints, tradeoffs, assumptions, evidence, and decision rights** are explicit.
- Commitment is formalized when **ownership, governance, change protocol, evidence expectations, and revisit triggers** are explicit.

Each lifecycle stage page defines concrete checks and "smells" (early warning signals).

2.2.8 How to read the next pages

Use the lifecycle pages as building blocks:

- **Meaning Discovery** — what CDS must learn and align on first.
- **Intent Refinement** — how CDS turns meaning into decision-grade intent.
- **Commitment Formalization** — how CDS freezes intent into an accountable commitment ready for execution.

2.3 Meaning Discovery

Meaning Discovery is the first step of the Commitment Design System (CDS). Its purpose is to **discover and align on shared reality** — so intent refinement begins from truthfully stated conditions, needs, and frictions rather than assumptions or solution narratives.

Meaning Discovery does **not** produce solutions. It produces **meaning** in a form that can be reliably translated into intent.

2.3.1 Purpose

Meaning Discovery exists to:

- Establish a **shared understanding** of the situation (what is true, and for whom).
- Identify **needs** (what must be true for the situation to improve).
- Name **problems/frictions** (what prevents needs from being met).
- Make **stakes** visible (who benefits, who pays, who can block, who must operate the outcome).
- Capture **evidence** and **uncertainty** so later stages remain inspectable.

2.3.2 Boundaries

Meaning Discovery is complete when the group can say:

- “We agree on the *conditions* we are operating under.”
- “We agree on the *needs* that matter most right now.”
- “We can name the *frictions* blocking those needs.”
- “We know who is affected and what is at stake.”
- “We can point to evidence and unknowns — not just opinions.”

Meaning Discovery is **not** complete when:

- The conversation is still dominated by solution proposals.
- Stakeholders are missing and meaning is being inferred on their behalf.
- The “problem” is just a list of features not yet built.

2.3.3 Core outputs

Meaning Discovery produces the **Meaning Handshake** artifact. It is the canonical output schema for meaning in CDS.

The Meaning Handshake is designed to be:

- lightweight
- inspectable
- usable as direct input to Intent Refinement

2.3.4 Process

Meaning Discovery can be run as a short alignment session or a deeper discovery effort. Regardless of duration, it follows the same substages.

Step 1: Establish the situation

Align on what prompted the work now:

- What changed?
- What broke?
- What opportunity appeared?
- Why is this being discussed today (and not six months ago)?

Step 2: Capture conditions

Document the key conditions shaping the situation:

- environmental constraints (market, regulation, deadlines)
- operational constraints (systems, process bottlenecks, access)
- organizational constraints (ownership, incentives, capacity)
- climate conditions (trust level, fatigue, conflict temperature)

Step 3: Identify needs

Translate the situation into needs:

- What must be true for this to feel meaningfully improved?
- What must be protected (non-negotiables at the human/system level)?
- What needs are in tension (speed vs safety, autonomy vs alignment)?

Step 4: Name frictions and failure patterns

Describe what blocks needs today:

- bottlenecks and delays
- rework loops
- unclear decision rights
- semantic confusion
- dependency traps (approvals, access, procurement, other teams)

Step 5: Map stakeholders and stakes

Make meaning multi-perspective:

- who experiences the pain directly
- who benefits most if solved
- who pays (money, risk, reputation)
- who can block or slow change
- who must operate the outcome long-term

Step 6: Gather evidence and signals

Anchor meaning in signals:

- qualitative evidence (observations, quotes, support tickets)
- quantitative evidence (cycle time, errors, churn, cost)
- boundary cases (when it does not happen, when it is worse)

Step 7: Make uncertainty explicit

List assumptions and unknowns:

- what we believe but haven't verified
- what we disagree about
- what would change the framing if proven false

2.3.5 Meaning Handshake artifact

A minimal Meaning Handshake includes:

- **Situation** (why now)
- **Conditions**
- **Needs**
- **Frictions / Problems**
- **Stakeholders & Stakes**
- **Evidence & Signals**
- **Assumptions & Unknowns**

This artifact is intentionally not a requirements document. It is a meaning capture that preserves what is usually lost: context, stakes, and uncertainty.

2.3.6 Core quality checks

Use these checks before moving to Intent Refinement.

Meaning coherence checks

- Conditions are stated in a way others can recognize (not private interpretations).
- Needs are expressed as "must be true" statements, not solutions.
- Frictions clearly explain why needs are not met.

Multi-perspective checks

- At least one stakeholder group is represented or their input is captured.
- Stakes are not assumed to be aligned; tensions are named.

Evidence checks

- At least one concrete signal supports the narrative.
- The team can say what would disconfirm the framing.

Uncertainty checks

- Unknowns are explicit rather than silently embedded.
- Disagreements are captured as differences in meaning, not personality conflicts.

2.3.7 Common failure patterns

- **Solution-first discovery**: jumping to implementation before meaning is aligned.
- **Single-perspective meaning**: meaning defined by one role without stakeholder input.
- **Myth framing**: strong narrative with no evidence or falsification path.

- **Hidden stakes**: unspoken fears, incentives, or status threats surface later as resistance.

2.3.8 Transition to Intent Refinement

Meaning Discovery hands off via the **Meaning Handshake**. The next step, **Intent Refinement**, uses it to shape decision-grade intent: outcomes, boundaries, constraints, tradeoffs, assumptions, evidence expectations, and decision rights.

2.4 Intent Refinement

Intent Refinement is the second step of the Commitment Design System (CDS). Its purpose is to turn **meaning** into **decision-grade intent** — intent that is clear enough to commit to, and robust enough to guide execution without constant re-interpretation.

Intent Refinement does not finalize delivery plans. It **refines what must be achieved and under what rules**, so a commitment can be formalized responsibly.

2.4.1 Purpose

Intent Refinement exists to:

- Translate Meaning Handshake inputs into a **coherent intent statement**.
- Define **outcomes** and **success signals** (what "better" means).
- Set **boundaries** (what is in, out, and uncertain).
- Make **constraints** and **tradeoffs** explicit and owned.
- Identify **assumptions and unknowns** that affect viability.
- Define **decision rights** and **evidence expectations**.
- Produce an intent package that can be **formalized into a commitment**.

2.4.2 Boundaries

Intent Refinement is complete when the group can say:

- "We can state the intended outcome without describing a solution."
- "We know how we will recognize success."
- "We agree on the boundaries and constraints."
- "We have named the tradeoffs we are accepting."
- "We know what we still don't know — and how we will reduce that uncertainty."
- "We know who decides if reality changes."

Intent Refinement is **not** complete when:

- The intent is still a feature list or implementation proposal.
- Success is described vaguely ("improve", "optimize", "modernize") without signals.
- Constraints are discovered late ("security will review later").
- Decision ownership is missing ("we'll align later").

2.4.3 Core outputs

Intent Refinement produces the **Intent Package** artifact. It is the canonical output schema for decision-grade intent in CDS.

The Intent Package is designed to be:

- testable (via signals and evidence)
- bounded (via constraints and scope edges)
- inspectable (via explicit tradeoffs and assumptions)
- commit-ready (via decision rights and revisit triggers)

2.4.4 Process

Intent Refinement can be run in one session or several. Regardless of format, it follows the same substages.

Step 1: State the outcome (as a change)

Convert meaning into an outcome statement:

- For whom will the situation improve?
- What will be different when this succeeds?
- What will people be able to do that they cannot do today?

Avoid solution verbs (“build”, “implement”, “migrate”) in the first pass. Use change verbs (“reduce”, “enable”, “prevent”, “increase”, “make predictable”).

Step 2: Define success signals

Specify how success will be recognized:

- What measurable signals should move?
- What qualitative evidence will count?
- What would indicate “no improvement” or “worse”?

Separate:

- **leading signals** (early indicators)
- **lagging signals** (outcome confirmation)

Step 3: Set scope boundaries

Make the boundaries explicit:

- What is **in scope**?
- What is **out of scope**?
- What is **unknown / to be discovered**?
- What dependencies could reshape scope?

Step 4: Declare constraints

Turn implicit constraints into explicit rules:

- must / must-not constraints (compliance, security, privacy, brand, budget)
- non-negotiables vs preferences
- constraints that apply now vs later

Step 5: Surface and own tradeoffs

Every intent implies tradeoffs. Make them visible:

- What are we optimizing for?
- What are we sacrificing?
- Who is accepting the sacrifice?
- What risk are we consciously taking on?

Step 6: Make assumptions and unknowns explicit

List:

- assumptions that must be true for success
- disputed assumptions (where meaning differs)
- unknowns that could invalidate the intent

This is where intent becomes inspectable later.

Step 7: Define a learning plan (when uncertainty is material)

Translate uncertainty into learning moves:

- What is the smallest learning action that reduces risk?
- What decision will that learning enable?
- What is the timebox for learning before committing further?

Examples:

- prototype, spike, experiment, user test, technical validation

Step 8: Define decision rights and revisit triggers

Clarify:

- who decides acceptance of tradeoffs
- who can approve changes to intent
- what triggers a revisit (signals, events, constraint changes)
- what happens when a revisit trigger is hit

2.4.5 Intent Package artifact

A minimal Intent Package includes:

- **Outcome statement**
- **Success signals & evidence**
- **Scope boundaries** (in / out / unknown)
- **Constraints**
- **Tradeoffs accepted**
- **Assumptions & unknowns**
- **Learning plan** (if needed)
- **Decision rights**
- **Revisit triggers**

This artifact is intentionally not a delivery plan. It is the “commitment-ready” shape of intent.

2.4.6 Core quality checks

Use these checks before moving to Commitment Formalization.

Outcome quality checks

- Outcome is expressed as a change in the world, not a solution.

- Outcome is specific enough that people can disagree meaningfully.

Signal quality checks

- Success signals exist beyond “we shipped it.”
- At least one signal can be observed within the commitment horizon.

Boundary quality checks

- In/out boundaries are explicit.
- Unknowns are labeled as unknowns (not buried).

Constraint quality checks

- Non-negotiables are explicit and not contradictory.
- Constraints have owners (who enforces/validates them).

Tradeoff quality checks

- At least one explicit sacrifice is named.
- The tradeoff owner is identifiable.

Assumption quality checks

- Key assumptions are written down.
- There is a way to disconfirm them (even if not yet executed).

Decision quality checks

- Decision rights are explicit.
- Revisit triggers exist and are plausible.

2.4.7 Common failure patterns

- **Solution-shaped intent**: outcome is replaced by a preferred implementation.
- **Vague outcomes**: “improve” with no signals or boundaries.
- **Constraint denial**: constraints treated as “later problems”.
- **Tradeoff amnesia**: sacrifices are implicit until conflict erupts.
- **Unknowns disguised as certainty**: assumptions presented as facts.
- **Decision fog**: nobody can approve changes when reality shifts.

2.4.8 Transition to Commitment Formalization

Intent Refinement hands off via the **Intent Package**. The next step, **Commitment Formalization**, freezes refined intent into an accountable commitment: roles, governance, change protocol, evidence expectations, and revisit triggers — ready for execution.

2.5 Commitment Formalization

Commitment Formalization is the third step of the Commitment Design System (CDS). Its purpose is to **freeze refined intent into an accountable commitment** — a commitment that can be executed, governed, and evaluated without repeatedly re-discovering meaning or re-negotiating intent.

Commitment Formalization does not guarantee outcomes. It ensures the commitment is **explicit, owned, and governable**.

2.5.1 Purpose

Commitment Formalization exists to:

- Convert the Intent Package into a **binding commitment** (internal or external).
- Make **ownership and decision rights** explicit.
- Define **governance and change protocol** so adaptation is not escalation-by-default.
- Align **deliverables and evidence** (what will be produced vs how value will be validated).
- Set expectations for **risk posture**, **revisit triggers**, and **accountability**.
- Produce a commitment object that execution runtimes can run without inventing missing intent.

2.5.2 Boundaries

Commitment Formalization is complete when the group can say:

- “We know who is committing, and who is accountable for what.”
- “We know what is being committed — and what is explicitly not.”
- “We know how change will be handled.”
- “We know how value will be evaluated (not just what will be delivered).”
- “We know what conditions force us to revisit or re-formalize.”

Commitment Formalization is **not** complete when:

- The commitment depends on implied understanding (“we all know what we mean”).
- Ownership is distributed but accountability is unclear.
- Change is handled through informal escalation and surprise.
- Evidence of value is missing or defined as “shipping”.

2.5.3 Core outputs

Commitment Formalization produces the **Commitment Envelope** artifact. It is the canonical output schema for commitments in CDS and the handoff object into execution.

The Commitment Envelope is designed to be:

- explicit (what is committed and by whom)
- governable (how change and decisions work)
- inspectable (why this commitment was reasonable at the time)

2.5.4 Process

Commitment Formalization can be lightweight (internal alignment) or formal (RFP/SoW/contract). Regardless of format, it follows the same substages.

Step 1: Name the parties and roles

Make explicit:

- who is committing (teams/orgs)
- who is accountable (by dimension)
- who decides changes (decision rights)
- who must be consulted (domain voices, risk/compliance)

Step 2: Write the commitment statement

Produce a short statement that includes:

- intended outcome (from the Intent Package)
- scope boundary (in/out)
- horizon (timebox or checkpoints)
- quality bar (what "done" must include beyond shipping)

Step 3: Separate deliverables from evidence

Define:

- deliverables (what will be produced)
- evidence (how value/acceptance will be demonstrated)
- acceptance approach (who accepts and on what basis)

Step 4: Confirm constraints and tradeoffs

Carry forward:

- non-negotiable constraints (and their owners)
- accepted tradeoffs and sacrifices (and their owners)
- any "do not regress" conditions that must remain protected

Step 5: Define governance cadence

Decide:

- checkpoints (frequency)
- decision forum (who attends)
- what is reviewed (signals, risks, drift)
- how decisions are recorded

Step 6: Define the change protocol

Make change explicit:

- what counts as a change (scope, constraints, signals, timeline)
- how change is proposed and evaluated
- who approves
- what happens when change is rejected
- "fast path" for urgent change

Step 7: Define risk posture and escalation paths

List:

- top risks and early signals
- mitigation/rollback posture (conceptual level)
- escalation path when risks materialize

Step 8: Set revisit triggers and expiry

Specify:

- events/signals that trigger re-entry to Meaning Discovery or Intent Refinement
- commitment expiry or renewal point (if relevant)
- what must be re-confirmed when revisiting

Step 9: Formalize the agreement

Choose the appropriate formalization mechanism:

- internal agreement (recorded commitment)
- delivery agreement (engagement charter, working agreement)
- procurement/legal agreement (RFP response, SoW, contract)

The mechanism can differ; the commitment envelope content stays consistent.

2.5.5 Commitment Envelope artifact

A minimal Commitment Envelope includes:

- **Parties, roles, and decision rights**
- **Commitment statement** (outcome + boundary + horizon + quality bar)
- **Deliverables**
- **Evidence & acceptance**
- **Constraints & non-negotiables**
- **Tradeoffs accepted**
- **Governance cadence**
- **Change protocol**
- **Risk posture & escalation**
- **Revisit triggers & expiry**

This artifact is intentionally not a detailed delivery plan. It is the binding reference that execution uses to stay coherent.

2.5.6 Core quality checks

Use these checks before execution begins.

Accountability checks

- Accountability is explicit (not “shared” by default).
- Decision rights exist for changes, acceptance, and tradeoffs.

Commitment clarity checks

- Commitment statement is short, specific, and boundary-aware.
- In/out scope is explicit and does not rely on implied understanding.

Value and acceptance checks

- Evidence of value is defined separately from deliverables.
- Acceptance criteria and acceptance owner exist.

Governance checks

- Cadence is defined and includes review of signals, risk, and drift.
- Decision recording exists (even lightweight).

Change protocol checks

- Change is defined and routable.
- There is a non-escalation path for normal change.

Risk and revisit checks

- Top risks and early signals are named.
- Revisit triggers are explicit and plausible.

2.5.7 Common failure patterns

- **Commitment by implication**: “we all know” replaces explicit boundaries and roles.
- **Accountability diffusion**: many contributors, no accountable decider.
- **Deliverables-as-value**: shipping is treated as acceptance without evidence.
- **Governance theater**: meetings exist but decisions and changes are unmanaged.
- **Escalation-as-change-control**: every change becomes a conflict.
- **Frozen intent in a moving world**: no revisit triggers, so drift accumulates silently.

2.5.8 Transition to execution

The Commitment Envelope is the handoff object into execution runtimes. Execution should not need to re-invent intent; it should implement and validate what the commitment envelope defines.

When CDS is used with a delivery runtime (e.g., a software delivery runtime), the Commitment Envelope becomes the runtime’s input contract.

2.6 Core Quality Checks

This page defines the **core quality checks** of the Commitment Design System (CDS). These checks provide simple, repeatable criteria for deciding whether CDS can safely progress from:

Meaning Discovery → Intent Refinement → Commitment Formalization

They also define when CDS should **re-enter** an earlier step instead of pushing forward with ambiguity.

2.6.1 How to use these checks

Use these checks as:

- **Stage exit criteria** (before moving to the next step)
- **Ongoing drift detectors** (during longer engagements)
- **Re-entry triggers** (when reality changes or new information appears)

The checks are intentionally lightweight. They aim to prevent **premature commitment**, not to enforce bureaucracy.

2.6.2 Check types

CDS uses three types of checks:

- **Must** — required for moving forward. If missing, CDS should not progress.
- **Should** — strongly recommended. If missing, progress is possible but risky.
- **Smells** — warning signals. If present, pause and investigate.

2.6.3 Meaning Discovery quality checks

Must

- **Conditions are explicit**: key constraints and context are written down and not mutually surprising.
- **Needs are stated as needs**: “must be true” statements, not solution proposals.
- **Frictions are causal**: the team can explain how frictions prevent needs from being met.
- **Stakes are visible**: at least the primary affected groups and their stakes are identified.
- **Unknowns are declared**: assumptions and unknowns are captured rather than embedded.

Should

- **Evidence exists**: at least one concrete signal supports the framing (qualitative or quantitative).
- **Tensions are named**: incompatible needs (speed vs safety, autonomy vs alignment, etc.) are explicit.
- **Language is aligned**: obvious semantic conflicts are identified (even if not resolved yet).

Smells

- Solution debate dominates and meaning cannot be summarized without feature talk.
- One perspective defines meaning “for everyone”.
- “We all agree” appears, but stakeholders are absent or evidence is missing.
- The “problem” is actually a label for a preferred project (“modernize”, “replatform”, “AI”).

2.6.4 Intent Refinement quality checks

Must

- **Outcome is stated as change**: what will be different if successful (without prescribing implementation).
- **Success signals exist**: at least one observable signal beyond “we shipped it”, and:
    - at least one signal is **falsifiable** (it could show “no improvement” or “worse”)
    - at least one signal/evidence source is **independently observable** by the acceptance owner (not solely produced/curated by a single party)
- **Boundaries are explicit**: in-scope, out-of-scope, and unknown areas are identified.
- **Constraints are explicit**: true non-negotiables are stated, with an owner where relevant.
- **Tradeoffs are owned**: at least one explicit sacrifice is named, and someone accepts it.
- **Decision rights exist**: someone can approve changes and resolve conflicts.
- **Decision rights are operational**: deciders are named individuals (not just roles), are empowered in practice, and can reliably exercise the decision (attend the decision forum or have a standing delegate).
  If decision rights are expected to rotate, continuity rules are defined and role-change triggers re-formalization.
- **Revisit triggers exist**: clear conditions that force re-evaluation are stated.

Should

- **Assumptions have disconfirmation paths**: the team knows how key assumptions could be proven false.
- **Learning plan exists when uncertainty is material**: a small plan to reduce uncertainty is defined.
- **Stakeholder acceptance is understood**: who will accept value/evidence is identified.

Smells

- Intent is still a feature list or a solution mandate.
- “Improve/optimize/modernize” with no boundary or signal.
- Constraints are deferred to later (“security will review at the end”).
- Tradeoffs are denied (“we can have all of it”).
- Decision rights are ambiguous (“we’ll align later”).

2.6.5 Commitment Formalization quality checks

Must

- **Parties and accountability are explicit**: who commits and who is accountable for what.
- **Commitment statement is clear**: outcome + boundary + horizon/checkpoints + quality bar.
- **Deliverables and evidence are separated**: what is produced vs how value is evaluated.
- **Governance cadence exists**: a forum/cadence for reviewing progress, signals, and risks.
- **Change protocol exists**: what counts as change, how it is proposed, who decides, what happens next.
- **Risk posture is stated**: key risks and escalation paths are identified.
- **Revisit/expiry is defined**: what triggers re-entry, and when commitment is renewed or re-confirmed.

Should

- **Acceptance is operationalized**: acceptance owner, acceptance method, and evidence format are clear.
- **Dependency obligations are stated**: external dependencies have named owners and expectations.
- **Decision recording exists**: decisions and changes are captured in a lightweight log.

Smells

- “Everyone is accountable” (which means no one is).
- Acceptance is “we delivered it” rather than evidence of value.
- Change control is escalation-by-default.
- Governance meetings exist, but decisions are not recorded.
- The commitment relies on implied shared understanding instead of explicit envelope fields.

2.6.6 Re-entry rules

Re-entry is not failure. It is the normal behavior of CDS under uncertainty.

Use these rules:

Re-enter Meaning Discovery when

- stakeholder stakes were misunderstood or newly revealed
- constraints changed materially (regulatory, org, budget, deadline, trust climate)
- disagreement shows meaning was not aligned
- the problem framing no longer fits observed reality

Re-enter Intent Refinement when

- success signals are disputed or no longer measurable
- scope boundaries shift materially
- new constraints appear
- tradeoffs become unacceptable
- assumptions are disproven
- decision rights change

Re-enter Commitment Formalization when

- governance or change protocol is not working in practice
- accountability changes (new sponsor/decider)
- commitments must be renegotiated due to material changes
- commitment expiry/renewal is reached

2.6.7 Minimum viable CDS (v0)

For low-stakes work, CDS can be run with a minimal set:

- **Meaning Handshake (v0)**: situation, top conditions, top needs, top frictions, key stakeholders, unknowns
- **Intent Package (v0)**: outcome, one success signal, boundary, one constraint, one tradeoff, decision right
- **Commitment Envelope (v0)**: accountable owner, commitment statement, acceptance evidence, change protocol

Use v0 only when commitments are reversible and risk is low.

2.6.8 Outcome of “failing” a check

When a **Must** check fails, the correct action is:

- **pause**
- **name what is missing**
- **choose re-entry**
- **reduce uncertainty**
- then proceed

CDS is designed to prevent teams from “progressing” by silently converting ambiguity into downstream rework.

3. SOFTWARE DELIVERY PROFILE

3.1 Overview

The **Software Delivery Profile** is an extension of the CDS Core Model for commitments whose execution runtime is **software delivery**.

CDS Core is universal: it defines how to move from **Meaning Discovery → Intent Refinement → Commitment Formalization**. The Software Delivery Profile adds the minimum additional structure needed to ensure commitments are **executable in real software delivery environments** — where constraints surface late, dependencies are indirect, and “delivery” can accidentally become the place where intent is discovered too late.

3.1.1 What this profile is

This profile is:

- A set of **software-specific additions** to CDS artifacts (Meaning Handshake, Intent Package, Commitment Envelope).
- A set of **software-specific quality checks** (“profile gates”) that prevent predictable delivery failures.
- A mapping from the **Commitment Envelope** into a software delivery runtime (e.g., a delivery framework such as 3SF).

It keeps CDS Core intact. It does not introduce a new lifecycle.

3.1.2 Why software delivery needs a profile

Software delivery commitments fail in characteristic ways:

- **Hidden constraints** appear late (security, compliance, data, privacy, architecture limits).
- **Dependencies** are often indirect (access, environments, other internal teams, procurement workflows).
- **Semantic drift** accumulates (business terms evolve while systems fossilize).
- **Evidence is vague** (shipping is mistaken for value).
- **Decision rights are unclear** (changes become escalation-by-default).
- **Work starts before commitment is real** (execution begins with undefined intent).

The Software Delivery Profile pulls these realities upstream so they become part of intent and commitment formation — rather than surprises paid for during execution.

3.1.3 When to use this profile

Use the Software Delivery Profile when:

- the commitment involves building, changing, integrating, migrating, or operating software systems
- delivery spans multiple teams, organizations, or vendors
- constraints and approvals are likely to be significant
- the work is high-stakes (production risk, compliance risk, reputational risk)
- you expect uncertainty and change, and need governance that can handle it

You may not need this profile for:

- low-risk internal tooling
- small reversible changes with single-team ownership
- situations where constraints and dependencies are already fully known and stable

3.1.4 Relationship to 3SF (runtime mapping)

The Software Delivery Profile is designed to hand off into a software delivery runtime. If you use **3SF**, the mapping is direct:

- CDS **Commitment Envelope** becomes a **runtime input contract**
- 3SF operates as the **execution runtime**, maintaining coherence across:
    - **Client ↔ Vendor ↔ Product**
    - **Engagement ↔ Delivery ↔ Value**

This profile ensures the Commitment Envelope contains what a delivery runtime actually needs: decision rights, constraints, evidence expectations, dependency obligations, and change protocol — so delivery can execute rather than continuously renegotiate.

3.1.5 What changes vs CDS Core

What stays the same

- The CDS lifecycle and its three steps
- The purpose and boundaries of each step
- The core artifacts (same names, same intent)

What gets extended

- **Meaning Discovery** gains software-context fields (dependencies, semantics, operational reality).
- **Intent Refinement** gains delivery-grade intent elements (NFRs, reversibility, feasibility probes).
- **Commitment Formalization** gains runtime-ready commitment elements (governance, acceptance evidence, operational posture).

3.1.6 What you will find in this section

This section is structured as extensions to the CDS lifecycle:

- **Meaning Discovery (Software Delivery additions)**
- **Intent Refinement (Software Delivery additions)**
- **Commitment Formalization (Software Delivery additions)**

Each profile page includes:

- software-specific purpose and focus
- additions to the core artifact schema
- profile-specific quality checks
- common failure patterns and recovery moves

3.1.7 How to apply the profile

Start with CDS Core pages. Then apply this profile as a lens:

1. Run **Meaning Discovery** and produce a Meaning Handshake.
2. Apply the Software Delivery additions to capture delivery-relevant reality.
3. Run **Intent Refinement** and produce an Intent Package.
4. Apply the Software Delivery additions to make intent delivery-grade.
5. Run **Commitment Formalization** and produce a Commitment Envelope.
6. Apply the Software Delivery additions so the envelope is runtime-ready.
7. Hand off the Commitment Envelope into your delivery runtime (e.g., 3SF).

3.2 Mapping to 3SF

This page explains how the **CDS Commitment Envelope** maps into **3SF** as a delivery runtime.

CDS produces a commitment object. 3SF runs that commitment through execution while maintaining coherence across the triangle (**Client–Vendor–Product**) and across the three lines (**Engagement–Delivery–Value**). The mapping below is designed to make that handoff explicit and repeatable.

3.2.1 Why the mapping matters

Most delivery breakdowns happen when a commitment is *formally "agreed"* but is not **runtime-ready**:

- roles and decision rights are unclear
- value evidence is undefined
- constraints are discovered late
- dependencies are implicit
- change becomes escalation-by-default

3SF can detect and manage drift during execution — but only if the commitment envelope provides the necessary "input contract."

3.2.2 Mapping overview

The mapping is done in two passes:

- **Pass 1 — Triangle coherence:** Client ↔ Vendor ↔ Product
- **Pass 2 — Line coherence:** Engagement ↔ Delivery ↔ Value

You can treat this as a checklist during Commitment Formalization.

3.2.3 Pass 1: Client–Vendor–Product coherence

Client (why / authority / acceptance)

Commitment Envelope fields that must map clearly:

- **Sponsor / accountable buyer**
- **Decision rights and escalation path**
- **Acceptance owner** (who decides "this is acceptable")
- **Value evidence expectations** (what counts as proof)
- **Constraints owned by client** (policy, compliance, access, internal approvals)

3SF runtime implication: If these are missing, delivery will drift into "busy work," or stall waiting for decisions and approvals.

Vendor (capability / responsibility / execution integrity)

Commitment Envelope fields that must map clearly:

- **Accountable delivery owner**
- **Delivery responsibilities** (what the vendor owns vs supports)
- **Risk posture** (what risks vendor will actively manage)
- **Change handling** (how vendor proposes changes and what happens next)
- **Dependency obligations** (what vendor needs from client and by when)

3SF runtime implication:
If these are missing, vendor execution becomes reactive, and accountability diffuses.

Product (scope boundary / constraints / operability)

Commitment Envelope fields that must map clearly:

- **Commitment statement boundary** (in/out/unknown)
- **Quality bar** (including operational constraints)
- **Non-negotiables** (security, privacy, performance, uptime, data)
- **Revisit triggers** (what invalidates the commitment)
- **Evidence instrumentation expectations** (how product value is observed)

3SF runtime implication:
If these are missing, “done” becomes subjective and value becomes unprovable.

3.2.4 Pass 2: Engagement–Delivery–Value coherence

Engagement line (Client ↔ Vendor)

This is the working relationship contract.

Map from Commitment Envelope:

- **Governance cadence** (how often, who attends, what is reviewed)
- **Decision forum** (where decisions are made and recorded)
- **Communication norms** (channels, response expectations where needed)
- **Escalation rules** (what escalates, how, and to whom)
- **Change protocol** (normal vs urgent changes)

3SF runtime implication: This prevents “relationship drift” where delivery is blocked by absent leadership or unclear collaboration mechanics.

Delivery line (Vendor ↔ Product)

This is the execution integrity contract.

Map from Commitment Envelope:

- **Definition of done / quality bar**
- **Constraints enforcement** (who validates security, compliance, performance)
- **Dependency access** (environments, data, credentials, approvals)
- **Risk posture** (rollback, mitigation, cutover stance where relevant)
- **Decision rights for technical tradeoffs** (who can accept debt, performance tradeoffs, scope cuts)

3SF runtime implication: This prevents “delivery drift” where teams build the wrong thing, build it unsafely, or are blocked by invisible dependencies.

Value line (Product ↔ Client)

This is the value validation contract.

Map from Commitment Envelope:

- **Evidence & acceptance** (what counts as value, who verifies, how often)
- **Success signals** (leading/lagging indicators)
- **Adoption expectations** (if relevant: training, rollout, comms ownership)
- **Post-release responsibilities** (who owns operations, support, incident response)
- **Revisit triggers tied to value** (signals that force re-evaluation)

3SF runtime implication:
This prevents “value drift” where deliverables ship but outcomes remain unclear or unmeasured.

3.2.5 Practical mapping table

Use this as a compact handoff guide.

CDS Commitment Envelope element	3SF coherence target
Parties, roles, decision rights	Client ↔ Vendor alignment (Engagement)
Commitment statement (outcome + boundary)	Product scope coherence (Product)
Deliverables	Delivery coherence (Vendor ↔ Product)
Evidence & acceptance	Value coherence (Product ↔ Client)
Constraints & non-negotiables	Delivery + Product integrity
Tradeoffs accepted	All three lines (prevents conflict later)
Governance cadence	Engagement runtime stability
Change protocol	Engagement + Delivery drift control
Risk posture & escalation	Delivery safety + recoverability
Revisit triggers & expiry	Runtime re-entry conditions

3.2.6 Mapping quality checks (profile gates)

Before handing the Commitment Envelope into 3SF, confirm:

- Every triangle vertex has an **accountable owner** (Client/Vendor/Product).
- Every line has a defined **operating contract** (Engagement/Delivery/Value).
- Acceptance is defined as **evidence**, not “we shipped it.”
- Dependencies are visible and owned (especially client-side access/approvals).
- Change protocol exists and is routable without escalation-by-default.

3.2.7 Common mapping failures

- **Engagement undefined:** delivery starts without decision forums and escalation rules.
- **Delivery undefined:** constraints and dependencies are discovered late.
- **Value undefined:** acceptance is subjective and metrics are absent.
- **Triangle imbalance:** one vertex dominates (e.g., vendor blamed for client decision vacuum).
- **Tradeoffs denied:** later conflict appears as “quality vs speed” arguments.

3.2.8 What comes next

The next profile pages extend the CDS lifecycle artifacts so the mapping becomes natural:

- Meaning Discovery (Software Delivery additions)
- Intent Refinement (Software Delivery additions)
- Commitment Formalization (Software Delivery additions)

3.3 Meaning Discovery for Software Delivery

This page extends **CDS Meaning Discovery** for commitments executed through **software delivery**.

Software delivery introduces predictable “meaning traps”: constraints appear late, dependencies are indirect, and stakeholders assume shared understanding while using different language. This profile ensures Meaning Discovery captures the additional reality needed to refine intent and formalize a commitment that is actually executable.

3.3.1 Purpose (software delivery focus)

In software delivery, Meaning Discovery must capture not only *why change is needed*, but also the **delivery reality** that will shape or block execution.

This profile extension exists to:

- Reveal **delivery constraints early** (access, environments, security/compliance gates).
- Identify **dependency systems** (other teams, workflows, approvals, vendors).
- Capture **domain semantics** (shared language and meaning boundaries).
- Ground the situation in **signals** from product and operations (not only narratives).
- Make **operational stakes** explicit (who will run/support the result).

3.3.2 Boundaries

Software-delivery Meaning Discovery is complete when the group can say:

- “We understand the situation and needs *and* the delivery conditions that shape feasibility.”
- “We know where approvals, access, or internal dependencies can block us.”
- “We have identified the key domain terms and where language diverges.”
- “We can reference operational/product signals supporting the framing.”

It is not complete when:

- constraints are deferred (“security will check later”)
- stakeholders closest to operations/support are absent
- dependencies are treated as “someone else will handle it”
- “modernization/migration” is used as the problem statement

3.3.3 Additions to the Meaning Handshake (software profile)

Use the standard Meaning Handshake fields, and add the following software-delivery fields.

Delivery conditions

Capture conditions that impact software execution:

- **Access reality** (accounts, RBAC, approvals, lead times)
- **Environment reality** (dev/test/prod availability, parity, release windows)
- **Data reality** (sources, quality, governance, residency, privacy constraints)
- **Security/compliance gates** (reviews, policies, evidence requirements)
- **Operational reality** (monitoring, incident response, on-call constraints)

Dependency landscape

Explicitly name indirect dependencies:

- internal platform teams (IAM, network, security, SRE, data)
- ticketing/approval workflows (e.g., ServiceNow-style queues)
- external vendors and tool owners
- release management / CAB processes
- procurement/legal constraints (if applicable)

For each dependency, capture:

- owner/team
- expected lead time
- success condition (what “done” means)
- what blocks them from acting

Domain language (semantic alignment)

Capture the minimal domain vocabulary needed to avoid drift:

- key terms (customer, account, order, incident, “done”, “active”, etc.)
- conflicting definitions
- terms that are overloaded or politically charged
- where the system language disagrees with business language

This is not DDD design yet. It’s **meaning alignment** so intent refinement doesn’t hardcode the wrong semantics.

Operational stakes

Make explicit who will live with the result:

- who operates the system after change
- who supports incidents
- who owns SLOs/SLAs (if any)
- what failure looks like operationally (risk posture in meaning terms)

Evidence sources (software signals)

Anchor meaning in signals such as:

- incident reports, postmortems
- support ticket categories and volumes
- latency/error trends
- deployment frequency / lead time / change failure rates (where available)
- usage/adoption funnel signals
- cost signals (cloud spend, license cost, operational load)

You do not need perfect metrics. You need at least *some* signal beyond opinion.

3.3.4 Process (software-delivery emphasis)

Run the core Meaning Discovery process, with these additional prompts.

Step: Expose delivery constraints early

Ask:

- "What approvals or gates can stop us?"
- "What access will we need and how long does it take?"
- "Which environment limitations shape what's possible?"

Step: Make dependency queues visible

Ask:

- "Which teams must act for us to progress?"
- "What is their incentive and priority relative to ours?"
- "What's the typical lead time and failure mode?"

Step: Align domain language before intent refinement

Ask:

- "Which terms do we use that might not mean the same thing to others?"
- "Which term disagreements caused past rework?"
- "Where does the current system's language disagree with business meaning?"

Step: Include operations/support meaning

Ask:

- "Who gets paged when it fails?"
- "What risks are unacceptable operationally?"
- "What would make support load worse?"

3.3.5 Software profile quality checks (Meaning)

Must

- Delivery conditions include at least **access + security/compliance + environments** (even at a high level).
- At least one **dependency system** is identified, with an owner and expected lead time.
- At least one **operational stakeholder** perspective is represented or captured.
- At least one **software signal** supports the framing.

Should

- Key domain terms are captured with at least one known ambiguity.
- Top operational risks are stated in meaning terms (what must be protected).

Smells

- "We'll get access later."
- "Security will review when we're done."
- Dependencies are invisible until execution stalls.
- Domain terms are used confidently but mean different things across groups.
- Operations/support is absent from the conversation.

3.3.6 Common software-delivery failure patterns

- **Constraint ambush:** security/privacy/data restrictions discovered late.
- **Dependency paralysis:** work stalls in ticket queues or indirect teams.
- **Semantic drift:** teams build the “right thing” with the wrong meaning.
- **Invisible operability:** support and reliability needs surface only after release.
- **Meaning collapse into solution:** “modernize/migrate” replaces the actual need.

3.3.7 Transition to Intent Refinement (software profile)

The extended Meaning Handshake becomes input to software-delivery Intent Refinement, where intent must explicitly include:

- quality attributes (NFRs)
- feasibility probes (spikes/validations)
- dependency obligations
- evidence expectations and instrumentation needs

3.4 Intent Refinement for Software Delivery

This page extends **CDS Intent Refinement** for commitments executed through **software delivery**.

In software delivery, intent must survive a reality where constraints surface late, dependencies are indirect, and “done” can be mistaken for value. This profile adds the minimum intent structure needed to make commitments executable and governable in delivery runtimes (e.g., 3SF).

3.4.1 Purpose (software delivery focus)

Software-delivery Intent Refinement exists to turn meaning into **delivery-grade intent**:

- Outcomes and success signals that are observable in software realities
- Boundaries and constraints that reflect architecture, data, security, and operations
- Explicit treatment of **quality attributes** (NFRs) as first-class intent
- Explicit dependency obligations (access, environments, approvals, platform teams)
- A feasibility and learning plan that reduces technical and organizational unknowns
- Clear decision rights for inevitable tradeoffs during implementation

3.4.2 Boundaries

Software-delivery Intent Refinement is complete when:

- Intent includes **quality attributes** and operational expectations (not only features).
- Constraints include security/privacy/data realities and their validation ownership.
- Dependencies are explicit, owned, and time-aware.
- At least one feasibility path exists for major unknowns (spike/prototype/validation).
- Evidence expectations include instrumentation/measurement approach where relevant.
- Decision rights exist for scope changes, tradeoffs, and acceptance evidence.

It is not complete when:

- intent is primarily a backlog of features
- non-functional requirements are “later”
- feasibility is assumed rather than validated
- dependency work is invisible or treated as “someone else’s problem”

3.4.3 Additions to the Intent Package (software profile)

Use the standard Intent Package fields, and add the following software-delivery fields.

Quality attributes (NFRs) as first-class intent

Define the quality bar in intent terms:

- reliability / availability expectations (if applicable)
- performance expectations (latency, throughput, batch windows)
- security and privacy requirements (incl. threat posture where relevant)
- observability expectations (logs/metrics/traces, alerting)
- maintainability expectations (ownership boundaries, upgrade posture)
- operability expectations (on-call, runbooks, incident response posture)

Keep this lightweight: a few “must be true” statements + owners.

Reversibility classification

Classify key parts of intent by reversibility:

- **easy to reverse** (low cost)
- **costly to reverse**
- **effectively irreversible** (locks architecture/data/contracts/user behavior)

For irreversible elements, require:

- explicit tradeoff ownership
- revisit triggers
- feasibility validation before commitment deepens

Feasibility probes (delivery-grade learning plan)

When uncertainty is material, define probes such as:

- architecture spike / proof-of-concept
- data profiling / migration rehearsal
- integration test with external systems
- security review pre-check (policy fit, evidence requirements)
- performance baseline measurement

Each probe should specify:

- timebox
- decision it enables
- pass/fail signals

Dependency obligations

Make dependencies part of intent, not a delivery surprise:

- required access (roles, lead times, approvals)
- environment readiness (test data, parity, release windows)
- platform team work requests (what, when, who owns follow-up)
- procurement/legal gates (if any)
- stakeholder availability constraints (SME time, acceptance cadence)

Include:

- owner
- expected lead time
- what “ready” means

Evidence and instrumentation expectations

If value is expected to be measured, define:

- what signals will be captured
- where they come from (analytics, logs, business reports)
- who owns instrumentation
- when measurement begins

Avoid perfection. Aim for “measurable enough to decide.”

Technical decision rights

Explicitly define who can approve:

- scope reduction to meet constraints
- acceptance of technical debt (and under what conditions)
- performance/security tradeoffs
- architecture changes that affect reversibility
- release/cutover decisions (if applicable)

This prevents “engineering is blocked by unclear authority.”

3.4.4 Process (software-delivery emphasis)

Run the core Intent Refinement process, with these additional substages.

Step: Make quality attributes explicit

Ask:

- “What must be true operationally for this to be acceptable?”
- “What failure modes are unacceptable?”
- “What will support/on-call refuse to inherit?”

Step: Classify reversibility early

Ask:

- “Which choices lock us in?”
- “What becomes hard to undo once we start?”
- “What do we need to validate before we cross that line?”

Step: Convert unknowns into feasibility probes

Ask:

- “What do we need to learn first to avoid expensive rework?”
- “What would disconfirm our current intent?”
- “What is the smallest test that clarifies feasibility?”

Step: Pull dependencies into intent

Ask:

- “What work must other teams do for us to succeed?”
- “What lead times can block delivery?”
- “Who owns the dependency outcomes?”

Step: Define evidence capture

Ask:

- “How will we prove value beyond shipping?”
- “Do we need instrumentation changes, and who owns them?”

Step: Confirm decision rights

Ask:

- “When constraints collide with scope, who decides?”
- “Who can approve a meaningful tradeoff quickly?”

3.4.5 Software profile quality checks (Intent)

Must

- At least one **quality attribute** is explicitly stated (usually reliability/security/operability).
- Any major **irreversible** element is identified and owned.
- At least one **feasibility probe** exists for the biggest unknown.
- Key **dependencies** are listed with owners and lead times (even rough).
- Acceptance evidence is more than “shipped”; instrumentation expectations are stated where relevant.
- Technical decision rights exist for tradeoffs that are likely to occur.

Should

- A minimal operational posture is defined (who runs it, what “safe” means).
- Constraint validation ownership is explicit (security/compliance/architecture sign-off).
- Dependency obligations include “definition of ready” per dependency.

Smells

- “NFRs later.”
- “Security review at the end.”
- “We’ll figure environments/access as we go.”
- “We can change architecture later” without reversibility awareness.
- “Analytics/measurement isn’t necessary” while claiming outcome improvement.

3.4.6 Common software-delivery failure patterns

- **Feature-only intent:** quality attributes missing, causing late conflict.
- **Irreversible surprise:** lock-in discovered after work begins.
- **Probe avoidance:** uncertainty treated as confidence; spikes seen as waste.

- **Dependency denial:** external teams become bottlenecks mid-flight.
- **Unmeasurable value:** outcomes claimed, but no evidence plan exists.

3.4.7 Transition to Commitment Formalization (software profile)

The extended Intent Package becomes input to software-delivery Commitment Formalization, where commitments must include:

- governance cadence and decision forums
- change protocol that matches delivery reality
- operational ownership and acceptance evidence
- dependency obligations as part of the commitment envelope

3.5 Commitment Formalization for Software Delivery

This page extends **CDS Commitment Formalization** for commitments executed through **software delivery**.

In software delivery, commitments fail when they are formal on paper but not **runtime-ready**: access and dependencies block work, constraints arrive late, “done” is confused with value, and change becomes escalation. This profile adds the minimum formalization structure needed to hand off a commitment into a delivery runtime (e.g., 3SF) with clarity and governance.

3.5.1 Purpose (software delivery focus)

Software-delivery Commitment Formalization exists to:

- Produce a **runtime-ready Commitment Envelope** for delivery.
- Formalize **governance, decision forums, and change protocol** suited to delivery reality.
- Formalize **acceptance evidence** (value proof) distinct from deliverables.
- Bind **dependency obligations** (especially client-side access/approvals) into the commitment.
- Define **operational posture** (who runs it, what “safe” means, release/cutover stance).
- Prevent execution from re-discovering meaning and re-negotiating intent mid-flight.

3.5.2 Boundaries

Software-delivery Commitment Formalization is complete when:

- The commitment envelope maps cleanly into 3SF coherence targets:
  - Client/Vendor/Product owners exist
  - Engagement/Delivery/Value contracts exist
- “Done” and “valuable” are not conflated; acceptance evidence is explicit.
- Dependencies (access, environments, indirect teams) are explicit, owned, and time-aware.
- Change protocol is routable without escalation-by-default.
- Operational ownership and risk posture are explicit.

It is not complete when:

- governance is “we’ll meet weekly” with no decision mechanism
- acceptance is “we delivered X”
- dependencies remain implied (“client will provide access”)
- security/compliance constraints are acknowledged but not operationalized

3.5.3 Additions to the Commitment Envelope (software profile)

Use the standard Commitment Envelope fields, and add the following software-delivery fields.

Runtime mapping fields (3SF-ready)

Make the triangle and lines explicit:

- **Client owner(s):** sponsor + acceptance owner + policy gate owners
- **Vendor owner(s):** accountable delivery owner + escalation path
- **Product owner(s):** domain/product authority + operational authority

And the three line contracts:

- **Engagement contract:** governance forum, cadence, decision recording, escalation
- **Delivery contract:** quality bar, dependency readiness, technical decision rights
- **Value contract:** success signals, evidence cadence, acceptance method, adoption ownership

Acceptance evidence (value contract)

Formalize acceptance as evidence:

- what evidence proves progress/value (signals + qualitative proof)
- cadence of evidence review
- who accepts and how acceptance is recorded
- what happens when evidence is inconclusive

This prevents “acceptance by shipment.”

Dependency obligations (explicit commitments)

Bring dependencies into the envelope:

- access provisioning obligations (roles, lead times, owners)
- environment readiness obligations (test data, parity, release windows)
- internal dependency team obligations (what they must deliver, by when)
- procurement/legal obligations (if relevant)
- SME availability commitments (hours/cadence)

For each obligation, define:

- owner
- expected lead time
- “definition of ready”
- escalation path

This prevents delivery stalling in hidden queues.

Operational posture and ownership

Define the operational reality of the result:

- who runs it post-change (team/function)
- SLO/SLA expectations (if applicable)
- monitoring/alerting/runbook expectations (minimal)
- support model (handoff, on-call, incident ownership)
- release/cutover stance (where relevant: canary, phased rollout, maintenance windows)

This prevents “deliver and disappear.”

Technical tradeoff authority

Formalize who can approve:

- accepting technical debt and under what limits
- performance/security tradeoffs
- scope reductions due to constraints
- architecture changes that impact reversibility
- release/cutover go/no-go decisions

This prevents slow motion decision paralysis.

3.5.4 Process (software-delivery emphasis)

Run the core Commitment Formalization process, with these additional substages.

Step: Build the runtime contract explicitly

Before “signing,” confirm:

- triangle owners (Client/Vendor/Product)
- line contracts (Engagement/Delivery/Value)
- mapping completeness (nothing critical is implicit)

Step: Convert intent into acceptance evidence

Ask:

- “What evidence will we review to confirm value?”
- “How frequently will we review it?”
- “Who can accept, reject, or request change based on evidence?”

Step: Bind dependency obligations

Ask:

- “What must the client/platform teams provide, by when?”
- “What is the lead time and what is ‘ready’?”
- “What happens if this obligation is not met?”

Step: Decide operational posture

Ask:

- “Who owns operations after release?”
- “What makes this safe to run?”
- “What is our release/cutover posture?”

Step: Formalize change protocol for delivery reality

Ask:

- “What changes are routine vs escalated?”
- “Who decides within 24–48 hours when reality shifts?”
- “How will scope/constraints be renegotiated without conflict?”

3.5.5 Software profile quality checks (Commitment)

Must

- Triangle owners are named (Client/Vendor/Product) with accountability and decision rights.
- Engagement/Delivery/Value contracts exist (at least minimally).
- Acceptance evidence is defined and not equivalent to shipping.
- Key dependencies include owner + lead time + definition of ready + escalation.
- Change protocol is routable and includes decision forum + decision recording.
- Operational ownership is explicit (who runs/supports the result).

Should

- Release/cutover posture is stated when production risk exists.
- Constraint validation ownership is explicit (security/compliance/architecture).
- Technical tradeoff authority is explicit for likely tradeoffs.

Smells

- “Client will provide access” with no owner, lead time, or definition of ready.
- “We’ll align with security later.”
- Governance exists as meetings, but decisions and changes are unmanaged.
- Delivery starts while acceptance evidence is still vague.
- Operational ownership is assumed (“ops will handle it”) with no agreement.

3.5.6 Common software-delivery failure patterns

- **Runtime gap:** formal commitment exists, but delivery cannot execute due to missing owners/decision rights.
- **Dependency stall:** execution blocked by access, environments, indirect teams.
- **Acceptance theater:** demos replace evidence; value is never validated.
- **Change escalation:** every change becomes a conflict or leadership escalation.
- **Operational surprise:** production readiness and support obligations appear late.

3.5.7 Transition to 3SF runtime

The software-delivery Commitment Envelope is now **3SF-ready**:

- It defines triangle ownership (Client/Vendor/Product).
- It defines line contracts (Engagement/Delivery/Value).
- It makes acceptance evidence, dependencies, and change protocol explicit.

From this point, 3SF operates as the runtime system that executes the commitment, detects drift, and governs adaptation without losing coherence.

3.6 Playbooks by Scenario

This page provides lightweight **scenario playbooks** for applying the CDS Software Delivery Profile. Each playbook describes:

- when to use it
- what to emphasize in Meaning Discovery, Intent Refinement, and Commitment Formalization
- the minimum outputs to produce
- common failure signals to watch for

These are not “full methodologies.” They are repeatable patterns for shaping commitments that survive software delivery reality.

3.6.1 RFI/RFP initiation

Use when

- a client requests a proposal, estimate, or plan
- scope is unclear or solution is assumed
- procurement expects fixed language early

Emphasize

Meaning Discovery

- client-side stakes and decision rights (who truly owns outcomes)
- constraint discovery early (security, compliance, data residency)
- dependency landscape (access, environments, internal teams)

Intent Refinement

- outcome and success signals (avoid feature-only proposals)
- explicit assumptions (what the proposal depends on)
- feasibility probes (what must be validated post-award)
- tradeoffs (speed vs certainty, fixed scope vs learning)

Commitment Formalization

- commitment envelope “procurement mode” (translate into SoW/RFP terms)
- change protocol (how discovery changes scope without conflict)
- dependency obligations (client access, SMEs, approvals)

Minimum outputs

- Meaning Handshake (v0–v1)
- Intent Package with explicit assumptions + probes
- Commitment Envelope with change protocol + dependency obligations

Failure signals

- client cannot name decision owner or acceptance owner
- constraints are “unknown until after award”
- proposal is forced into fixed scope without an explicit uncertainty strategy

3.6.2 Legacy modernization / migration

Use when

- “modernization” is the headline but meaning is unclear
- there is production risk and hidden constraints
- dependencies and legacy knowledge are fragmented

Emphasize

Meaning Discovery

- operational pain and failure modes (incidents, costs, stability)
- domain semantics and “why the system exists”
- constraint ambush prevention (security, data, compliance)

Intent Refinement

- reversibility classification (what becomes irreversible)
- feasibility probes (data profiling, migration rehearsal, cutover simulation)
- explicit “do not regress” needs (availability, support load, compliance)
- success signals that reflect reality (lead time, change failure rate, incident rate, cost)

Commitment Formalization

- release/cutover posture (phased, canary, parallel run)
- operational ownership (who runs it, support model)
- change protocol tuned for uncertainty (learning gates)

Minimum outputs

- Meaning Handshake with operational signals + dependency landscape
- Intent Package with NFRs + reversibility + probes
- Commitment Envelope with cutover stance + revisit triggers

Failure signals

- modernization defined as tech replacement with no operational outcome
- no plan for data realities and cutover risk
- operations/support absent from commitment

3.6.3 New product build (discovery → build)

Use when

- a new product or major capability is being created
- value assumptions are high and evidence is needed early
- scope is likely to evolve based on learning

Emphasize

Meaning Discovery

- user/customer needs and progress (JTBD lens is useful here)
- stakeholder stakes (who funds, who benefits, who can block)
- evidence baseline (what signals exist today)

Intent Refinement

- define success signals early (activation, retention, task success)
- learning plan as first-class (experiments, prototypes, user tests)
- scope boundaries that protect learning (avoid premature full build)
- tradeoffs: speed-to-learn vs completeness

Commitment Formalization

- commitments framed as learning milestones + value evidence cadence
- change protocol that supports iteration without political escalation
- acceptance based on evidence, not feature completeness

Minimum outputs

- Meaning Handshake with user needs + stakeholder stakes
- Intent Package with clear signals + learning plan
- Commitment Envelope with evidence cadence + decision forum

Failure signals

- pressure to commit to full scope before learning
- “MVP” defined as a small scope, but with no evidence plan
- acceptance defined as “features delivered”

3.6.4 Team augmentation engagement

Use when

- client requests people/capacity rather than outcomes
- client leadership may be weak or fragmented
- delivery depends on client-side access, priorities, and decisions

Emphasize

Meaning Discovery

- client intent clarity: what outcomes they actually need (even if they say “resources”)
- dependency reality: access, environments, onboarding, approvals
- decision vacuum detection: who prioritizes and accepts work

Intent Refinement

- boundaries: what augmented team owns vs does not own
- success signals: flow metrics + client satisfaction + value evidence
- constraints: client process constraints (ticket queues, CAB, security gates)
- tradeoffs: utilization vs effectiveness, speed vs coordination

Commitment Formalization

- engagement contract (cadence, escalation, how priorities are set)
- explicit dependency obligations (client must provide access, decisions, SMEs)
- definition of “ready” for work intake (to prevent idle time and churn)

Minimum outputs

- Meaning Handshake focused on leadership, access, and bottlenecks
- Intent Package with boundaries + decision rights
- Commitment Envelope with engagement contract + intake readiness rules

Failure signals

- “We’ll give you tasks” without decision forum or backlog ownership
- access/onboarding lead times are unknown
- team becomes idle due to client bottlenecks

3.6.5 Rescue / reset engagement

Use when

- execution has started but outcomes are unclear
- scope is thrashing and trust is low
- hidden constraints and dependencies have already surfaced painfully

Emphasize

Meaning Discovery

- reality reset: what is actually true right now
- stakeholder alignment: what each party believes is happening
- evidence audit: what signals show drift (value drift, delivery drift, engagement drift)

Intent Refinement

- shrink to decision-grade intent: clarify outcomes, boundaries, constraints, tradeoffs
- identify what assumptions were false
- define a learning plan to unblock confidence quickly
- re-establish decision rights

Commitment Formalization

- renegotiate commitment envelope (especially change protocol and governance)
- define immediate stabilization commitments (stop-the-bleeding)
- explicit revisit triggers and escalation paths

Minimum outputs

- Meaning Handshake updated from current reality
- Intent Package that narrows scope and redefines evidence
- New Commitment Envelope (renewed commitment, not a patch)

Failure signals

- “Just execute harder” without re-entering meaning/intent
- blame cycles prevent reality statement
- governance exists but nobody can decide

3.6.6 How to pick the right playbook

Choose the playbook based on the dominant risk:

- **Procurement compression risk** → RFI/RFP initiation
- **Production/legacy risk** → modernization/migration
- **Value uncertainty risk** → new product build
- **Decision vacuum risk** → team augmentation
- **Trust + drift risk** → rescue/reset

3.6.7 Re-entry reminder

If a scenario gets stuck, don’t “push through.” Re-enter the correct stage:

- meaning conflict or missing stakeholders → **Meaning Discovery**
- unclear signals/boundaries/tradeoffs → **Intent Refinement**
- unclear accountability/change control → **Commitment Formalization**

3.7 Failure Patterns and Recovery Moves

This page lists common **software delivery failure patterns** and the recommended **CDS recovery moves**. Each pattern includes:

- early signals (what you notice first)
- typical root cause (which CDS stage was skipped or under-specified)
- recovery move (which stage to re-enter and what to produce)

These patterns are intentionally practical. The goal is not blame—it is fast re-alignment.

3.7.1 Pattern: Constraint ambush (security/compliance/data arrives late)

Early signals

- “Security will review later” becomes “Security blocked release”
- new requirements appear late (data residency, PII handling, audit evidence)
- architecture must change after implementation starts

Typical root cause

- **Meaning Discovery** did not capture delivery conditions and gate ownership
- **Intent Refinement** treated constraints as future validation, not intent

Recovery move

Re-enter **Meaning Discovery (Software Delivery)**:

- capture gate owners and evidence requirements
- document data/security constraints as conditions and non-negotiables

Then re-enter **Intent Refinement (Software Delivery)**:

- elevate constraints to first-class intent
- add feasibility probe (policy fit check, threat posture review)
- update tradeoffs and decision rights

3.7.2 Pattern: Dependency paralysis (blocked in indirect teams / ticket queues)

Early signals

- delivery team is idle waiting for access, environments, approvals
- “it’s in ServiceNow” becomes the dominant status update
- lead times are unknown; no one can escalate effectively

Typical root cause

- **Meaning Discovery** missed dependency landscape (owners, lead times, definitions of ready)
- commitment did not bind dependency obligations

Recovery move

Re-enter **Meaning Discovery (Software Delivery)**:

- map dependencies with owner + lead time + definition of ready
- identify incentive misalignment and escalation paths

Then re-enter **Commitment Formalization (Software Delivery)**:

- formalize dependency obligations and “definition of ready”
- add governance cadence item for dependency review
- define escalation path when obligations are not met

3.7.3 Pattern: Semantic drift (building the “right thing” with the wrong meaning)

Early signals

- repeated rework due to “misunderstood requirements”
- stakeholders approve wording but disagree during demos
- business terms are overloaded (“account”, “active”, “done”, “customer”)

Typical root cause

- **Meaning Discovery** did not capture domain language and ambiguities
- intent was refined as features without semantic alignment

Recovery move

Re-enter **Meaning Discovery (Software Delivery)**:

- capture key terms, conflicting definitions, and boundary cases
- collect examples (real data/events) that illustrate meaning

Then re-enter **Intent Refinement**:

- restate outcomes and boundaries using agreed terms
- add acceptance evidence that relies on shared semantics (examples/tests)

3.7.4 Pattern: Feature-only intent (NFRs and operability appear late)

Early signals

- “works on my machine” → production readiness conflict
- performance/reliability debates start late in the cycle
- operations/support push back near release

Typical root cause

- **Intent Refinement** did not include quality attributes as first-class intent
- operational stakeholders were absent from meaning

Recovery move

Re-enter **Meaning Discovery (Software Delivery)**:

- include operational stakes and failure modes (“what must be protected”)

Then re-enter **Intent Refinement (Software Delivery)**:

- define NFRs/quality attributes (minimal “must be true” set)
- define observability and support posture expectations

Then re-enter **Commitment Formalization** if acceptance criteria must change.

3.7.5 Pattern: Decision vacuum (no one can say “yes/no”)

Early signals

- work stalls waiting for approval
- conflicting stakeholder requests with no resolution mechanism
- changes are negotiated via escalation, not decisions

Typical root cause

- **Intent Refinement** did not define decision rights
- **Commitment Formalization** did not define governance and change protocol

Recovery move

Re-enter **Intent Refinement**:

- name decision rights per category (scope, tradeoffs, constraints, acceptance)

Then re-enter **Commitment Formalization**:

- define decision forum, cadence, and decision recording
- define change protocol with a routable path

3.7.6 Pattern: Value drift (deliverables ship, outcomes remain unclear)

Early signals

- demos feel “busy” but impact is unknown
- “done” means shipped, not improved outcomes
- stakeholders argue about whether it’s working

Typical root cause

- **Intent Refinement** did not define success signals and evidence expectations
- commitment treats deliverables as value

Recovery move

Re-enter **Intent Refinement**:

- define success signals (leading/lagging)
- define evidence review cadence and acceptance owner

Then re-enter **Commitment Formalization**:

- formalize acceptance evidence and what happens if evidence is inconclusive

3.7.7 Pattern: Scope thrash (constant change without control)

Early signals

- backlog churn dominates planning
- "just one more thing" becomes the default mode
- delivery feels chaotic; estimates collapse

Typical root cause

- boundaries and tradeoffs were not explicit in intent
- change protocol was missing or unused

Recovery move

Re-enter **Intent Refinement**:

- restate boundaries (in/out/unknown)
- name explicit tradeoffs and what is being sacrificed

Then re-enter **Commitment Formalization**:

- formalize change protocol (what counts as change, who decides, how recorded)
- add revisit triggers rather than silent scope creep

3.7.8 Pattern: Probe avoidance (unknowns treated as confidence)

Early signals

- major technical questions are postponed
- risks are discussed but not tested
- rework appears late and expensive

Typical root cause

- learning plan (feasibility probes) was missing from intent

Recovery move

Re-enter **Intent Refinement (Software Delivery)**:

- convert top unknowns into timeboxed probes
- define pass/fail signals and the decision each probe unlocks
- update reversibility classification and tradeoffs accordingly

3.7.9 Pattern: Governance theater (meetings exist, decisions don't)

Early signals

- recurring status meetings with no clear decisions
- same issues repeat; nothing is resolved
- change and risk are discussed but not managed

Typical root cause

- governance cadence exists without decision mechanism and recording
- commitment formalization did not specify how governance produces decisions

Recovery move

Re-enter **Commitment Formalization**:

- define the decision forum (agenda, decision types, attendance)
- introduce lightweight decision recording
- connect governance cadence to change protocol and revisit triggers

3.7.10 Pattern: Operational surprise (handoff to ops/support fails)

Early signals

- “we delivered it” but support refuses ownership
- incidents increase; no runbooks/alerts exist
- unclear on-call responsibilities and escalation paths

Typical root cause

- operational posture and ownership were not formalized
- meaning did not include operational stakes

Recovery move

Re-enter **Meaning Discovery (Software Delivery)**:

- capture operational stakes, unacceptable failure modes

Then re-enter **Commitment Formalization (Software Delivery)**:

- formalize operational ownership, support model, minimal observability/runbooks
- define release/cutover posture if needed

3.7.11 Fast diagnosis: which stage to re-enter?

Use this quick mapping:

- Conflicting realities, missing stakeholders, hidden constraints → **Meaning Discovery**
- Vague outcomes, missing signals, unclear boundaries, denied tradeoffs → **Intent Refinement**
- Unclear accountability, change chaos, acceptance disputes → **Commitment Formalization**

3.7.12 Recovery principle

When a failure pattern appears, do not “push execution harder.”

Instead:

1. identify the pattern
2. re-enter the right CDS stage
3. update the artifact (Handshake / Intent Package / Envelope)
4. re-formalize what changed

That is how CDS prevents repeated drift.

3.8 Templates (Profile Versions)

This page provides **Software Delivery Profile** versions of the three CDS artifacts. These templates are intentionally lightweight. They are designed to be:

- fast to fill
- easy to review
- stable enough to support handoffs (especially into 3SF)

Use them as:

- working documents during workshops
- handoff artifacts between roles/teams
- alignment checkpoints during delivery

3.8.1 Meaning Handshake (Software Delivery)

Situation

- Why now? What triggered this work?

Conditions (include delivery reality)

- Business/organizational conditions:
- Delivery conditions:
    - Access/RBAC reality:
    - Environment reality:
    - Security/compliance gates:
    - Data realities (sources, privacy, residency):
    - Operational realities (support, monitoring, incident posture):

Needs (must be true)

- Need 1:
- Need 2:
- Needs in tension (if any):

Frictions / Problems (what blocks needs)

- Friction 1:
- Friction 2:

Stakeholders & Stakes

- Primary beneficiaries:
- Primary payers (money/risk/reputation):
- Blockers / gate owners:
- Operators/support owners:

Dependency landscape

For each dependency:

- Dependency:
- Owner/team:
- Lead time (typical):
- Definition of ready:
- Escalation path:

Domain language (semantics)

- Key terms and meanings:
- Known ambiguous/contested terms:
- Examples/boundary cases:

Evidence & Signals

- Qualitative signals:
- Quantitative/operational signals:

Assumptions & Unknowns

- Assumption:
- Unknown:
- Disagreement (if any):

3.8.2 Intent Package (Software Delivery)

Outcome statement (change in the world)

- For whom:
- What changes:
- What stays protected:

Success signals & evidence

- Leading signals:
- Lagging signals:
- Evidence review cadence:
- Acceptance owner (for evidence):

Scope boundaries

- In scope:
- Out of scope:
- Unknown / to be discovered:
- Key dependencies that can reshape scope:

Constraints (non-negotiables)

- Security/privacy:

- Compliance/regulatory:
- Data constraints:
- Budget/time constraints:
- Other constraints:
- Constraint validation owner(s):

Tradeoffs accepted

- Optimizing for:
- Sacrificing:
- Tradeoff owner(s):

Quality attributes (NFRs) — must be true

- Reliability/availability:
- Performance:
- Observability:
- Operability/support:
- Maintainability:
- Other:

Reversibility classification

- Easy to reverse:
- Costly to reverse:
- Effectively irreversible (requires explicit ownership):

Assumptions & Unknowns (with disconfirmation paths where possible)

- Assumption:
    - How it could be disproven:
- Unknown:
    - How we reduce it:

Feasibility probes / learning plan (timeboxed)

Probe template:

- Probe:
- Timebox:
- Pass/fail signal:
- Decision it enables:

Decision rights

- Who decides scope changes:
- Who decides constraint exceptions (if any):
- Who approves tradeoffs/debt:
- Who approves acceptance evidence:

Revisit triggers

- Trigger:
- What happens when triggered:

3.8.3 Commitment Envelope (Software Delivery / 3SF-ready)

Parties, roles, decision rights (triangle owners)

- Client sponsor:
- Client acceptance owner:
- Client gate owners (security/compliance/access):
- Vendor accountable delivery owner:
- Product/domain authority:
- Operational authority (who runs/supports it):

Commitment statement

- Outcome:
- Boundary (in/out):
- Horizon (timebox/checkpoints):
- Quality bar (must be true):

Deliverables

- Deliverable 1:
- Deliverable 2:

Evidence & acceptance (value contract)

- Evidence required:
- Evidence cadence:
- Acceptance method:
- What happens if evidence is inconclusive:

Constraints & non-negotiables

- Constraint:
- Validation owner:
- Evidence needed (if relevant):

Tradeoffs accepted

- Tradeoff:
- Tradeoff owner:

Dependency obligations (explicit commitments)

Obligation template:

- Obligation:
- Owner:
- Lead time:
- Definition of ready:
- Escalation path:
- What happens if not met:

Governance cadence (engagement contract)

- Forum/cadence:
- Required attendees:
- Agenda minimum (signals, risks, decisions, dependencies):
- Decision recording method:

Change protocol

- What counts as change:
- How change is proposed:
- Who approves:
- Urgent change path:
- Change recording method:

Risk posture & escalation (delivery contract)

- Top risks:
- Early signals:
- Mitigation/rollback stance (conceptual):
- Escalation path:

Operational posture (product runs in reality)

- Support model / on-call:
- Monitoring/alerting expectations:
- Runbook expectations:
- Release/cutover posture (if relevant):

Revisit triggers & expiry

- Revisit triggers:
- Expiry / renewal point:
- What must be re-confirmed on renewal:

3.8.4 Notes on usage

- Start with **v0** versions (fill only the bold essentials) when stakes are low and reversibility is high.

- Use full templates when:
    - multiple teams/orgs are involved
    - access/approvals are significant
    - production risk or compliance risk exists
    - acceptance and value evidence matter

These templates are intended to be copied into your working tools (docs, tickets, proposals) and adapted to your org context without changing the underlying CDS structure.

4. REFERENCE

4.1 Glossary

This glossary defines key CDS terms and common acronyms used across the documentation. Definitions are intentionally practical and optimized for consistent usage.

4.1.1 Acronyms

CDS — Commitment Design System
A system model for discovering meaning, refining intent, and formalizing commitments.

HCS — Human Cooperation System
A system model for designing, sustaining, and diagnosing human cooperation in complex work.

3SF — 3-in-3 SDLC Framework
A meta-framework for software delivery that maintains coherence across the **Client–Vendor–Product** triangle and the **Engagement–Delivery–Value** lines.

JTBD — Jobs To Be Done
A lens for expressing needs as desired progress in a context, including obstacles and success criteria.

DDD — Domain-Driven Design
An approach to designing software around domain language, boundaries, and models, emphasizing shared understanding and explicit context boundaries.

RFI — Request for Information
A pre-procurement request used to gather information from vendors/providers to shape options and assess fit.

RFP — Request for Proposal
A procurement request asking vendors/providers to propose a solution, scope, pricing, and approach against stated needs and constraints.

SoW — Statement of Work
A formal document defining deliverables, scope boundaries, responsibilities, and commercial terms for a commitment.

NFR — Non-Functional Requirement
A requirement describing quality attributes and constraints (e.g., reliability, security, performance), not feature behavior.

SLO / SLA — Service Level Objective / Service Level Agreement
Targets (SLOs) and contractual commitments (SLAs) for service performance and reliability.

RBAC — Role-Based Access Control
A mechanism for managing system access based on roles and permissions.

4.1.2 Core CDS Concepts

Meaning
Shared understanding of the situation: conditions, needs, frictions, stakes, evidence, and uncertainty. Meaning is not solutions.

Meaning Discovery
The CDS step that discovers and aligns meaning without jumping to solutions. Output: Meaning Handshake.

Meaning Handshake
The canonical CDS artifact that captures meaning in an inspectable form so intent refinement starts from shared reality.

Intent
Decision-grade clarity about what should change and under what rules: outcomes, success signals, boundaries, constraints, tradeoffs, assumptions, decision rights, and revisit triggers.

Intent Refinement
The CDS step that turns meaning into decision-grade intent. Output: Intent Package.

Intent Package
The canonical CDS artifact that captures refined intent in a commitment-ready form.

Commitment
A binding agreement to pursue an intent under stated rules: ownership, governance, change protocol, acceptance evidence, and revisit triggers.

Commitment Formalization
The CDS step that freezes intent into an accountable commitment. Output: Commitment Envelope.

Commitment Envelope
The canonical CDS artifact that defines the commitment object and serves as the handoff contract into an execution runtime.

4.1.3 Quality, Change, and Governance

Decision-grade
Clear enough that reasonable people can commit, disagree, or trade off explicitly. Not perfect certainty—sufficient clarity.

Decision rights
Explicit authority to decide specific categories of decisions (scope, tradeoffs, constraints, acceptance, change).

Governance cadence
A recurring forum and rhythm for reviewing signals, risks, dependencies, and making/recording decisions.

Change protocol
The defined mechanism for proposing, evaluating, approving, and recording changes to a commitment (including an urgent path).

Revisit trigger
A defined signal or event that forces re-entry into Meaning Discovery or Intent Refinement rather than silently drifting.

Expiry / renewal
A defined point where a commitment must be re-confirmed, renewed, or replaced.

4.1.4 Common CDS Quality Terms

Must / Should / Smell
CDS check types: “Must” blocks progress if missing; “Should” is recommended; “Smell” is a warning signal requiring investigation.

Inspectable
Structured so future teams can reconstruct why a decision or commitment was reasonable at the time (conditions, tradeoffs, decision rights, evidence).

Tradeoff
A consciously accepted sacrifice made to commit under constraints (e.g., speed over completeness, cost over performance).

Constraint
A non-negotiable rule that shapes what is acceptable (e.g., security policies, data residency, budget ceiling). Distinct from a preference.

Evidence
Signals and artifacts used to evaluate whether a commitment is producing value (distinct from shipping deliverables).

4.1.5 Software Delivery Profile Terms

Software Delivery Profile
A CDS extension that adds software-specific fields and checks so commitments are executable in real delivery environments.

Quality attributes
Operational and technical properties that define “acceptable” beyond features (reliability, security, performance, observability, operability).

Reversibility
How costly it is to undo a decision or change. Often categorized as easy, costly, or effectively irreversible.

Feasibility probe
A timeboxed learning action (spike/prototype/validation) designed to reduce a major unknown and unlock a decision.

Dependency obligation
An explicit commitment from a party or team to provide something required for delivery (access, environment readiness, approvals).

Operational posture
Agreed expectations for running and supporting the delivered system: support model, monitoring, runbooks, incident response posture.

4.1.6 3SF Mapping Terms

Client–Vendor–Product triangle
A 3SF coherence model describing the three parties that must remain aligned for delivery to succeed.

Engagement–Delivery–Value lines
A 3SF coherence model describing the three contracts that must remain stable during execution: relationship mechanics (Engagement), execution integrity (Delivery), and value validation (Value).

Runtime
The execution system that runs a commitment (methods, governance, teams, tools). CDS produces the commitment; the runtime executes it.

4.2 Artifact Schemas

This page defines the **canonical schemas** for CDS artifacts. These schemas are stable across CDS Core and can be **extended** by profiles (e.g., Software Delivery Profile) without changing the underlying structure.

Each field includes:

- **Purpose** (why it exists)
- **Guidance** (how to write it well)
- **Common smells** (early warning signals)

4.2.1 Meaning Handshake schema

Situation

Purpose: Anchor why the work exists now.
Guidance: One short paragraph or 3–5 bullets: trigger, urgency, why now.
Smells: “Because leadership said so” with no observable trigger; pure solution framing.

Conditions

Purpose: Capture reality that shapes what is possible and what matters.
Guidance: List the few conditions that *must be true* for the framing to hold (constraints, context, climate).
Smells: Too generic; missing constraints; stated as opinions rather than observable conditions.

Needs

Purpose: Define what must become true for the situation to improve.
Guidance: Write needs as “must be true” statements; include tensions where relevant.
Smells: Needs written as features/solutions; vague “improve/optimize” language.

Frictions / Problems

Purpose: Explain what prevents needs from being met today.
Guidance: Describe causal blockers (bottlenecks, decision gaps, rework loops, unclear ownership).
Smells: A feature backlog disguised as problems; blaming individuals instead of describing system friction.

Stakeholders & Stakes

Purpose: Make meaning multi-perspective and politically real.
Guidance: Identify performers/beneficiaries/payers/blockers/operators and what each risks or protects.
Smells: “Stakeholders = everyone”; missing acceptance owner; stakes assumed aligned.

Evidence & Signals

Purpose: Ground meaning in signals instead of narrative.
Guidance: Include at least one qualitative and/or quantitative signal that supports the framing.
Smells: No evidence; only anecdotes; metrics without interpretation context.

Assumptions & Unknowns

Purpose: Preserve uncertainty explicitly so later stages stay inspectable.
Guidance: Separate “assumptions we believe” from “unknowns we must learn”; include disputes.
Smells: Assumptions presented as facts; unknowns hidden inside confident statements.

4.2.2 Intent Package schema

Outcome statement

Purpose: Define the intended change in the world (not the solution).
Guidance: Express as a change for a specific group; avoid implementation verbs on first pass.
Smells: Outcome is a project (“build X”, “migrate Y”); outcome is too broad to disagree with.

Success signals & evidence

Purpose: Define how success will be recognized and validated.
Guidance: Provide leading and/or lagging signals; state how evidence will be collected/reviewed.
Smells: “Success = delivered”; signals are undefined or unobservable.

Scope boundaries (in / out / unknown)

Purpose: Prevent silent scope creep and misalignment.
Guidance: Explicitly name what is included, excluded, and not yet known.
Smells: Everything is “in”; unknowns buried; boundaries depend on implied understanding.

Constraints (non-negotiables)

Purpose: Define what must not be violated.
Guidance: Separate true non-negotiables from preferences; name constraint owner where relevant.
Smells: Constraints deferred (“later”); contradictions; no ownership for enforcement/validation.

Tradeoffs accepted

Purpose: Make sacrifices visible and owned.
Guidance: State what is being optimized and what is being sacrificed; name who accepts it.
Smells: “No tradeoffs”; tradeoffs discovered only when conflict erupts.

Assumptions & Unknowns (decision impact)

Purpose: Identify what could invalidate intent or reshape scope.
Guidance: Prioritize assumptions by impact; note disconfirmation paths where possible.
Smells: Assumptions unprioritized; high-risk unknowns ignored.

Learning plan (if uncertainty is material)

Purpose: Turn uncertainty into timeboxed learning that unlocks decisions.
Guidance: Define smallest learning actions + what decision they enable.
Smells: Learning treated as optional; no timebox; no decision linkage.

Decision rights

Purpose: Prevent decision vacuum during inevitable tradeoffs and change.
Guidance: Define who decides scope changes, constraint exceptions (if any), tradeoffs, acceptance evidence.
Smells: “We’ll align later”; committee-by-default; unclear veto paths.

Revisit triggers

Purpose: Define when intent must be revisited instead of silently drifting.
Guidance: Tie triggers to signals/events/constraint shifts; define what happens on trigger.
Smells: No triggers; triggers are unrealistic; triggers exist but no action is defined.

4.2.3 Commitment Envelope schema

Parties, roles, and accountability

Purpose: Define who is committing and who is accountable for what.
Guidance: Name accountable owners; distinguish contributors from deciders.
Smells: “Everyone accountable”; accountability split without decision rights.

Commitment statement

Purpose: Freeze intent into a short binding statement.
Guidance: Outcome + boundary + horizon/checkpoints + quality bar.
Smells: Too long; ambiguous; depends on implied knowledge.

Deliverables

Purpose: Define what will be produced.
Guidance: Keep to key deliverables; avoid turning this into a full plan.
Smells: Deliverables explode into backlog; deliverables replace outcomes.

Evidence & acceptance

Purpose: Define how value is evaluated and who accepts it.
Guidance: Separate “delivered” from “valuable”; define acceptance owner and evidence cadence.
Smells: Acceptance = demo; acceptance owner missing; value evidence undefined.

Constraints & non-negotiables

Purpose: Carry forward constraints as commitment rails.
Guidance: Include validation ownership and evidence expectations where relevant.
Smells: Constraints acknowledged but not enforceable; validation is undefined.

Tradeoffs accepted

Purpose: Preserve the intentional sacrifices behind the commitment.
Guidance: Restate the key tradeoffs and owners; keep short.
Smells: Tradeoffs disappear at commitment time; later disputes repeat old decisions.

Governance cadence

Purpose: Provide a stable forum to review progress, risks, and decisions.
Guidance: Define cadence, attendees, minimum agenda, and decision recording.
Smells: Meetings without decisions; attendance unclear; no record of decisions.

Change protocol

Purpose: Ensure changes are routable without escalation-by-default.
Guidance: Define what counts as change, proposal path, approval path, urgent path, recording.
Smells: Change = chaos; approvals unclear; urgent path abused.

Risk posture & escalation

Purpose: Make risk explicit and manageable.
Guidance: Name top risks, early signals, mitigation stance, escalation path.
Smells: Risks listed as generic boilerplate; escalation path is “raise to leadership”.

Revisit triggers & expiry

Purpose: Define when the commitment must be re-confirmed or re-formed.
Guidance: Tie triggers to meaningful events/signals; define renewal points if relevant.
Smells: Frozen commitment in moving reality; no renewal point for long engagements.

4.2.4 Profile extension rules

Profiles extend schemas without breaking them.

- **CDS Core fields remain stable.**
- Profile fields should be:
- additive (not renaming core fields)
- minimal (only what the runtime requires)
- traceable (clearly mapped to a runtime need or failure pattern)

Example: the Software Delivery Profile adds fields such as **quality attributes**, **dependency obligations**, and **operational posture** — but keeps the underlying Meaning/Intent/Commitment structure unchanged.

4.2.5 Minimum viable versions (v0)

When stakes are low and reversibility is high, you can use compressed versions:

- **Meaning Handshake v0:** situation, top conditions, top needs, top frictions, key stakeholders, unknowns
- **Intent Package v0:** outcome, one success signal, boundary, one constraint, one tradeoff, decision right
- **Commitment Envelope v0:** accountable owner, commitment statement, acceptance evidence, change protocol

Use v0 intentionally. If a v0 artifact starts growing, switch to the full schema.

4.3 Sources and Frameworks

CDS is not built from a single discipline. It's a synthesis layer: it pulls stable functions from multiple fields and arranges them around one job — **turning meaning into decision-grade intent and formal commitments**.

This page is not a bibliography. It's a map of intellectual neighborhoods CDS draws from, and the specific kinds of problems each neighborhood helps address.

4.3.1 How to read this page

Use it to:

- understand what CDS is compatible with (and why)
- recognize familiar ideas without assuming CDS "is just" that thing
- find good upstream reading when deepening a specific CDS capability

4.3.2 Systems thinking and cybernetics

What it contributes: feedback loops, control under uncertainty, drift, governance as regulation, system boundaries.
CDS uses it for: revisit triggers, change protocol, "inspectability", and re-entry behavior.

Common neighbors:

- systems thinking (boundaries, leverage points)
- cybernetics (control loops, sensing/actuation)
- second-order thinking (observer effects, narrative risk)

4.3.3 Socio-technical systems and organizational design

What it contributes: work as coordination across people + tools + incentives; failure modes from misaligned structures.
CDS uses it for: decision rights, dependency obligations, "delivery reality" capture, escalation prevention.

Common neighbors:

- socio-technical design
- Team Topologies / Conway's Law dynamics
- organizational incentives and principal–agent problems

4.3.4 Communication, facilitation, and negotiation

What it contributes: alignment mechanics, surfacing disagreement safely, shared language formation, decision hygiene.
CDS uses it for: Meaning Handshake facilitation, tradeoff ownership, commitment ceremonies without theater.

Common neighbors:

- facilitation practices (workshop design, alignment prompts)
- negotiation (interests vs positions)
- conflict-aware collaboration (psychological safety, escalation design)

4.3.5 Product discovery and problem framing

What it contributes: meaning as user/customer progress; separating needs from solutions; evidence-driven learning.
CDS uses it for: Meaning Discovery and early Intent shaping without collapsing into feature lists.

Common neighbors:

- JTBD and related discovery approaches
- design research and qualitative inquiry
- hypothesis-driven discovery

4.3.6 Strategy and sensemaking

What it contributes: framing choices, option spaces, constraints as strategic posture, “why now” clarity.
CDS uses it for: situation framing, tradeoff explicitness, scope boundaries, and commitment coherence.

Common neighbors:

- Wardley Mapping (context + evolution + strategic moves)
- VMOSA-style clarity (vision/mission/objectives)
- systems sensemaking practices

4.3.7 Requirements, intent, and decision quality

What it contributes: structured articulation of intent, ambiguity management, traceability, acceptance thinking.
CDS uses it for: “decision-grade intent”, boundaries, assumptions/unknowns, and acceptance evidence.

Common neighbors:

- requirements engineering (without treating it as ticket-writing)
- architecture decision records (ADRs) as “decision memory”
- testability and acceptance thinking

4.3.8 Governance, contracts, and procurement

What it contributes: commitments as agreements with obligations; change control; accountability; risk allocation.
CDS uses it for: Commitment Formalization, Commitment Envelope, procurement-friendly “compatibility mode”.

Common neighbors:

- SoW / RFP / RFI structures
- contract change control patterns
- accountability models (RACI-like clarity where useful)

4.3.9 Delivery methods and execution runtimes

What it contributes: ways to run work once a commitment exists (cadence, coordination, flow).
CDS uses it for: runtime handoff, governance cadence realism, avoiding “method wars”.

Common neighbors:

- Agile / Scrum / Kanban (execution mechanics)
- Lean delivery (flow, WIP, feedback)
- DevOps and continuous delivery (release posture, operability expectations)

CDS position: **above** delivery methods. CDS produces the commitment those methods should execute.

4.3.10 Architecture, quality attributes, and operability

What it contributes: constraints and quality attributes as first-class; reversibility; production reality.
CDS uses it for (software profile): NFR intent, reversibility classification, feasibility probes, operational posture.

Common neighbors:

- quality attributes (reliability, performance, security, maintainability)
- observability practices
- release/cutover strategies and production risk management

4.3.11 Risk, safety, and reliability thinking

What it contributes: risk as posture, not a checklist; failure modes; escalation design; safe-to-fail learning.
CDS uses it for: risk posture, early signals, revisit triggers, escalation paths that aren't political.

Common neighbors:

- SRE-style reliability thinking
- incident learning culture
- risk framing and mitigation planning

4.3.12 AI-assisted work and tooling

What it contributes: acceleration of drafting, summarization, and option exploration—plus new failure modes (confidence without grounding).
CDS uses it for: better artifact production and review, while defending against "coherent nonsense".

CDS stance: AI can help generate artifacts, but **CDS quality checks** must still be satisfied (signals, ownership, decision rights, tradeoffs).

4.3.13 What CDS contributes (the synthesis)

Many frameworks describe parts of the space. CDS contributes a specific integration:

- A stable **Meaning → Intent → Commitment** model as a translation pipeline
- Three canonical artifacts with clear boundaries:
    - **Meaning Handshake**
    - **Intent Package**
    - **Commitment Envelope**
- A consistent system of **quality checks** and **re-entry rules**
- A profile mechanism that keeps the core universal while enabling runtime-specific extensions (e.g., Software Delivery → 3SF)

In short: CDS is a **commitment formation system**, designed to work *with* your existing methods—by making the upstream commitment explicit, inspectable, and governable.

4.4 Practices Map

This page is a practical **bridge** between CDS and the wider world of industry practices and tools. It answers a simple question:

“If I already use X (or my org expects X), where does it fit inside CDS?”

The intent is not to rank practices. It’s to show how CDS can **host** them, **compose** them, and **diagnose gaps** when they are used without the upstream clarity CDS provides.

4.4.1 How to use this map

Use it in three modes:

- **Orientation:** understand where a practice contributes (Meaning, Intent, or Commitment).
- **Composition:** combine practices without turning CDS into a methodology soup.
- **Diagnosis:** when things go wrong, identify which CDS function is missing.

4.4.2 CDS lifecycle placement map

Meaning Discovery practices

These help discover and align meaning (needs, conditions, frictions, stakes, evidence).

- **JTBD interviews / story gathering** → needs and progress framing
- **Problem framing workshops** (Lean-style) → shared reality alignment
- **Stakeholder mapping** → stakes, blockers, operators
- **Service blueprinting / journey mapping** → friction visibility
- **Wardley mapping (early)** → situational context, constraints, option awareness
- **Incident review / postmortems** → operational meaning, failure modes
- **Support ticket analysis** → evidence of pain and friction

Common misuse: jumping from these directly to backlog items without intent refinement.

Intent Refinement practices

These help turn meaning into decision-grade intent (outcomes, signals, boundaries, constraints, tradeoffs, assumptions, decision rights).

- **OKRs / outcome framing** → outcome and success signals
- **User story mapping** → scope boundaries and sequencing hypotheses
- **Hypothesis and experiment design** → learning plan tied to decisions
- **DDD (strategic)** → domain language and boundaries (prevents semantic drift)
- **Architecture optioning** (lightweight) → constraints, reversibility awareness
- **Risk framing** (top risks + early signals) → intent inspectability
- **RACI-style decision clarification** → decision rights for tradeoffs and change

Common misuse: treating intent as “requirements” without signals, tradeoffs, or decision rights.

Commitment Formalization practices

These help freeze intent into an accountable commitment (governance, change protocol, acceptance evidence, obligations).

- **SoW / contract structures** → formal obligation scaffolding
- **RFP response packaging** → commitment expression in procurement language
- **Working agreements / team charters** → engagement contract
- **Governance cadence design** (forums, decision logs) → decision system
- **Definition of Done / acceptance criteria** → acceptance clarity (must include evidence)
- **Change control practices** → routable change without escalation
- **Risk registers** (lightweight) → explicit risk posture and escalation paths

Common misuse: formalizing deliverables while leaving acceptance evidence and change protocol vague.

4.4.3 CDS functions map (what CDS “hosts”)

Instead of mapping by stage, you can map by **function**. This tends to be more practical when designing your own playbooks.

Discover meaning

Common tools/practices:

- interviews, observation, ticket mining, incident reviews, journey mapping

Align meaning

Common tools/practices:

- facilitation, stakeholder mapping, shared vocabulary work, conflict surfacing

Shape decision-grade intent

Common tools/practices:

- outcome framing, OKRs, user story mapping, DDD boundaries, assumption mapping

Reduce uncertainty (feasibility)

Common tools/practices:

- spikes, prototypes, data profiling, architecture validation, security pre-checks

Formalize obligations

Common tools/practices:

- SoW/RFP structures, responsibility mapping, dependency obligations

Govern change

Common tools/practices:

- decision forums, decision logs, change protocol, escalation design

Validate value

Common tools/practices:

- metrics, analytics, acceptance evidence cadence, feedback loops

4.4.4 Software Delivery Profile: where practices attach

This is a practical view of which practices usually matter most in software delivery commitments.

Meaning Discovery (SD)

- incident / ops signal review (SRE-adjacent)
- dependency mapping (access, environments, internal teams)
- domain vocabulary capture (DDD precursor)

Intent Refinement (SD)

- NFR/quality attribute framing
- reversibility classification
- feasibility probes (spikes/prototypes)
- instrumentation planning (evidence capture)

Commitment Formalization (SD)

- governance cadence + decision forum
- change protocol (normal vs urgent)
- operational posture (support model, cutover stance)
- dependency obligations (definition of ready + lead times)

4.4.5 Pattern: practice without CDS function (gap diagnosis)

When a practice is present but a CDS function is missing, the system fails predictably.

- **Agile without decision-grade intent** → sprinting in circles
- **DDD without meaning alignment** → elegant models solving the wrong need
- **OKRs without commitment formalization** → goals with no accountability
- **RFP/SoW without evidence expectations** → deliverables-as-value
- **Workshops without decision rights** → alignment theater

Use the CDS Core Quality Checks page to locate which stage should be re-entered.

4.4.6 A note on tools

Tools (Jira, Azure DevOps, Miro, Confluence, ServiceNow, etc.) are not practices. They can support practices — but cannot replace:

- meaning alignment
- decision-grade intent
- commitment formalization

CDS treats tooling as implementation detail below the commitment layer.

4.5 Version and Licensing

This section documents the current version, license, and attribution principles for the **Commitment Design System (CDS)**.
It ensures transparency, traceability, and consistency across all CDS documentation and derivative work.

4.5.1 Version Information

Attribute	Description
System Name	Commitment Design System (CDS)
Version	**V1.0**
Status	Release Candidate
Release Date	January 2026
Maintained by	**3in3.dev**
Latest Online Documentation	Always find the latest version and web source here.

Version 1.0 Summary (Current)

CDS V1.0 introduces the core commitment-design lifecycle:

- **Meaning Discovery → Intent Refinement → Commitment Formalization**
- Canonical artifacts:
- **Meaning Handshake**
- **Intent Package**
- **Commitment Envelope**
- Core quality discipline:
- **Must / Should / Smells**
- **Re-entry rules** (Meaning / Intent / Commitment)
- A **Software Delivery Profile** that extends CDS for software commitments and runtime mapping into delivery frameworks (e.g., 3SF).

4.5.2 License

The Commitment Design System (CDS) and all related documentation are licensed under the:

> **License Type:** Creative Commons Attribution 4.0 International (CC BY 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format.
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

4.5.3 Attribution & Citation

To attribute this work, please use the following reference:

> *Commitment Design System (CDS) by 3in3.dev — licensed under CC BY 4.0 via 3in3.dev.*

4.5.4 Versioning Policy

- **Major versions (V2, V3, …)** introduce structural or conceptual extensions to the model.
- **Minor revisions (e.g., V1.1)** include refinements, clarifications, terminology alignment, and documentation improvements.
- All published versions will remain **permanently available** for reference and citation.
- Future releases will aim to maintain **backward compatibility** with the foundational definitions, rules, and models of CDS.

4.5.5 Attribution Guidelines

If reusing or adapting CDS content:

1. Include a visible credit line referencing *3in3.dev* and the license type.
2. Retain section structure where possible to preserve internal navigation and cross-references.
3. When combining CDS content with other frameworks, clearly separate attribution and derived materials.
4. For translations or derivative works, add a note identifying what changed and what version you based it on.

4.6 About the Author

Viktor Jevdokimov, Vilnius, Lithuania — Creator of 3in3.dev, HCS, and 3SF

Viktor Jevdokimov is a software engineering leader, systems thinker, and framework designer with over 30 years of experience in software product delivery, modernization, and team alignment.

He is the creator of the **Human Cooperation System (HCS)** and the **3-in-3 SDLC Framework (3SF)**, and founder of the **3in3.dev** initiative — an independent platform dedicated to advancing collaboration and alignment between **Client**, **Vendor**, and **Product** ecosystems.

Professional Background

- Began career supporting distributed banking software on DOS and Windows, developing a deep appreciation for troubleshooting and system design.
- Progressed through roles of **developer**, **architect**, **delivery lead**, and **practice lead**, working with international clients on modernization and cloud migration initiatives.
- Specializes in **Client–Vendor relationship design**, **project leadership**, and **delivery system diagnostics**.
- Advocates for *"Context before Method"* and *"Trust before Control"* as guiding principles of effective collaboration.

Creative and Personal Work

Beyond software, Viktor is an **active musician and live sound engineer**, performing and mixing with the *Great Things* cover band.
He approaches both sound and systems with the same mindset: striving for **clarity, balance, and authenticity**.

About 3in3.dev

3in3.dev is an independent research and publishing initiative founded by Viktor Jevdokimov.
It consolidates his experience and experimentation into open frameworks that help organizations improve how they **engage, deliver, and measure value** across collaborative ecosystems.

3in3.dev publishes:

- The **Human Cooperation System (HCS)** — theoretical foundation for cooperative system design.
- The **3-in-3 SDLC Framework (3SF)** — practical application of HCS principles in software delivery.
- Supporting tools, templates, and learning materials under an open license.

> "These systems aren't about control — they're about clarity, trust, and the shared intent that makes collaboration work."
> — Viktor J., Creator of 3in3.dev