T2D2 — Test² Telemetry² Driven Development

Version 1.0

A shop manual, not a sales brochure. This is the current, versioned state of the methodology — future revisions belong in new files (`v1.1`, `v2.0`), not silent edits to this one. `t2d2-field-guide.md` remains the working document behind this: the deliberation, the cut ideas, the naming history. This document is what you'd hand someone on day one.

Foreword: the journey

This started from a narrow, practical irritation: vibecoding works fine on a small script and falls apart on a large, complex codebase. Not because the model gets dumber — because nothing in the workflow tells it, or you, when the shape of what's being built has quietly drifted from what was asked for. That gap doesn't announce itself. It shows up weeks later as a bug nobody can explain, or never shows up at all and just becomes technical debt with no paper trail.

The instinct that follows from that is old, not new — it's Deming's, from a Toyota factory floor seventy years before any of this: catch the mistake by the shape of the process, not by inspecting the output afterward. Applying that to code meant asking what the equivalent of a factory's physical jig or gauge would be for software, and the answer turned out to be a graph — a structural map of what the code actually looks like, that could exist *before* the code does, be checked *against* the code once it's built, and keep being checked *after* it ships, against what the system actually does at runtime.

A second, personal influence shaped how that graph gets used day to day, borrowed from Heinlein's *Waldo* — a story about a man who builds tools that extend what he can physically do, and whose invented word for those tools became a real engineering term. The practical version of that idea here: build tools to build tools, and — the part that's less obvious — give an agent its constraints before its instructions, not after, and make it show its reasoning before it acts. That ordering, tried repeatedly, produced better results than the more natural instinct of stating the goal first and hoping the constraints get respected.

What follows is the methodology that came out of iterating on both of those instincts against a real, if young, tool. It is not a finished doctrine. It's a shop manual for a discipline still being built, written honestly about which parts are load-bearing and which parts are still just scaffolding.

What T2D2 is

T2D2 — Test² Telemetry² Driven Development — is a discipline for keeping a shared, checkable contract between what a human asked for and what a human-plus-agent pair actually builds, from before the first line of code to whatever the system does in production, through however long it keeps running after that. It extends Test-Driven Development's core move — write the check before the thing it checks — to the level of architecture and intent, not just function-level correctness.

The contract has three deliberately separate parts. A **graph** — a structural representation of the codebase, call shape and data shape — says what should *exist*. A **Test Manifest** — behavioral, Given/When/Then — says what should *happen*. An **Execution Trace**, once the system is live, says what actually *did* happen. The graph

and the trace correlate to the same nodes by stable, deterministic IDs; the Test Manifest is a separate, ordinary contract, not wired into that same ID scheme.

That graph does three jobs across a project's life: *before* code exists, it's proposed, for a human to review; *after* code is built, it's regenerated from the real thing and checked against the proposal; *once the system is live*, it becomes the map that production traces get plotted onto — which is what turns telemetry into something navigable, correlated all the way back to the exact node and call chain that produced it, rather than a firehose of logs. Debugging, refactoring, and onboarding all run on that same graph-plus-trace pairing, not as separate, unstructured "ops work" outside the discipline's scope.

The practice this produces — proposing a change's shape before writing it, checking the result against the proposal, and feeding what's learned in production back into the next proposal — is described ceremony by ceremony in Part I. What follows immediately below are the principles that give those ceremonies their spirit; Part I is where they become practice.

Guiding principles

These set the spirit the ceremonies are conducted in. They are not the ceremonies themselves — treat them as the terrain, not the map.

Build quality in; don't inspect it in after the fact. (Deming, Point 3.) The whole discipline exists to move the check earlier than a finished diff.

Improve the system, forever — not the output of any one change. (Deming's Plan-Do-Check-Act, the shape the entire loop below is built from.)

No quotas, no management by the numbers. (Deming, Point 11.) Every diagnostic signal this discipline produces is a signal for the people doing the work, never a ranking tool.

Drive out fear. (Deming, Point 8.) A detected mismatch is useful information, not a verdict.

Symmetrical grounding. Neither party in a human-agent pair is the infallible one. Humans carry outdated mental models, misremember how a library actually behaves, and project last year's architecture onto this year's codebase. Agents invent APIs that don't exist and miss context outside their window. The codebase itself — read directly, not recalled from memory or training — is the arbiter for both. The agent's job includes correcting the human's stale assumptions, not just executing the human's stated intent; the human's job includes noticing when an agent's confidence has outrun its evidence. This is not a hedge against agent failure alone — it's a hedge against the quieter, more common failure of a human confidently directing an agent to build the wrong thing.

A brief word on where this sits philosophically, since it's a genuine influence and worth naming rather than leaving implicit: there's a Sun Tzu instinct running underneath symmetrical grounding — that battles are won or lost in the accuracy of the assessment before the first move is made, not in cleverness during the fight. And there's a Taoist instinct in the discipline's insistence on speaking the codebase's own language rather than imposing an external one on it — working with the grain of what already exists instead of against it. Neither is load-bearing on its own; both are why the ceremonies below open with empirical investigation before they open with a plan.

Tools to build tools, constraints before requirements. (Reverse Polish Waldo.) The leverage is in the tooling, not any single piece of code it produces — and an agent given its guardrails first reasons better than one given them as an afterthought.

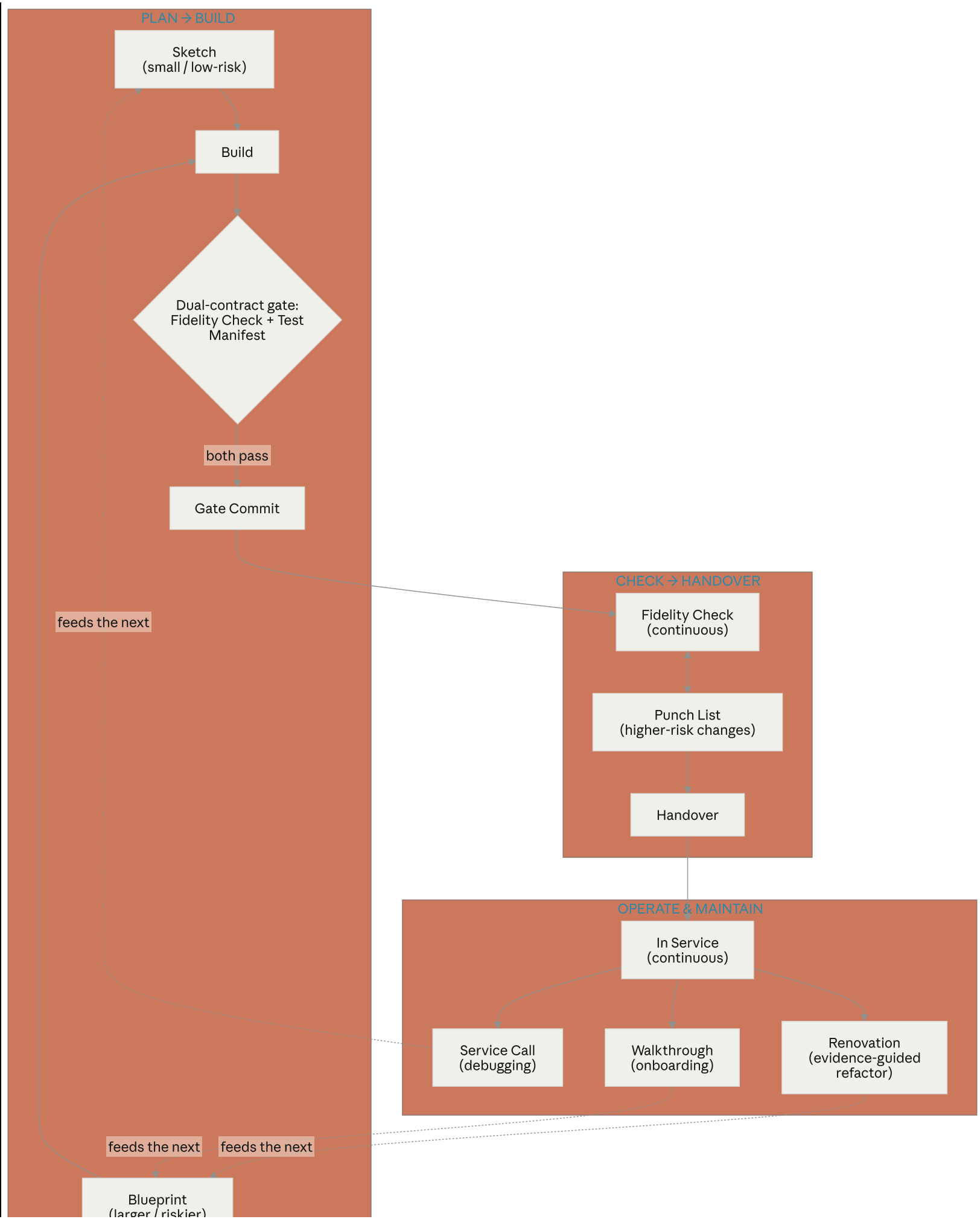
Keep the discipline open. The methodology is free and will stay that way; any commercial tooling built on it sits alongside it, never in front of it.

Part I: The ceremonies

How to read this section

Each ceremony below follows the same structure: **Purpose**, **Participants**, **Trigger**, **Inputs**, **What happens**, **Output**, **Exit criteria**. Participants are described by role (human, agent) rather than by a specific multi-agent architecture — nothing here requires or assumes a team of specialized agents versus one generalist agent handling a change end to end. That's a choice each team makes for itself; the artifacts are the interoperability layer, not a prescribed org chart.

The lifecycle, at a glance


PLAN → BUILD
Sketch
(small / low-risk)
Build
Dual-contract gate:
Fidelity Check + Test
Manifest
both pass
Gate Commit
feeds the next
CHECK → HANDOVER
Fidelity Check
(continuous)
Punch List
(higher-risk changes)
Handover
OPERATE & MAINTAIN
In Service
(continuous)
Service Call
(debugging)
Walkthrough
(onboarding)
Renovation
(evidence-guided
refactor)
feeds the next
feeds the next
Blueprint
(larger / riskier)

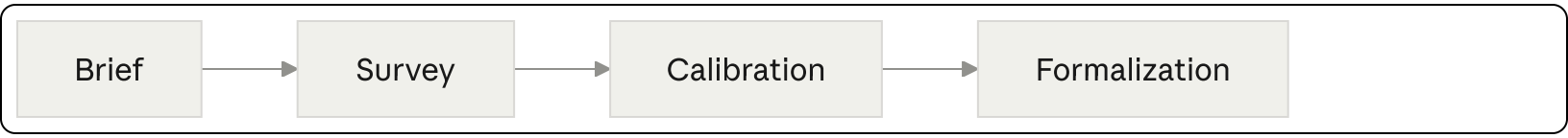


Sketch

Purpose: a fast, cheap comprehension check for small or low-risk changes — catch a misunderstanding while it costs nothing to fix.
Participants: human + agent.
Trigger: a change assessed as small/low-risk under Shop Rules.
Inputs: a brief statement of intent; the current graph.
What happens: the agent edits a lightweight duplicate of the current graph to represent the proposed change, applied through an MCP tool call that runs basic structural validation before accepting the edit (see Part III for what that validation does and doesn't catch). No formal Survey or Calibration stage — this is the lightweight path on purpose.
Output: a lightweight proposed graph.
Exit criteria: informal human go/no-go. Proceeds to Build.

Blueprint

Purpose: the full comprehension check for larger or higher-risk changes, closing with a real sign-off before any code exists.
Participants: human + agent, iterating.
Trigger: a change assessed as larger/higher-risk under Shop Rules, or a Sketch that surfaced enough ambiguity to warrant escalating.
Inputs: the human's initial Brief; the current graph and codebase.
What happens — four stages within one ceremony:



Brief. The human states intent, scope, and goals in prose. Explicitly includes room for "I might be wrong about X" — treated as a normal starting condition, not a weakness.
Survey. The agent performs deep, read-only research: the actual source files involved, the current graph, third-party library usage as it actually exists in the project (not as remembered), and any references the human provided. This produces a working plan that includes **Assumption Corrections** — places where the human's stated intent conflicts with what the agent actually found (e.g., "the Brief mentions the retry library, but it's not in the classpath — the project uses a different one here"). This is symmetrical grounding in practice: the agent's research is not just service to its own planning, it's the human's check against their own outdated mental model.
Calibration. Human and agent iterate until they share one accurate picture of both the current state and the target state. The human can — and should — interrogate the agent directly ("show me where that's actually used, trace the call chain"), and the agent answers with targeted graph queries, not recall. Each side corrects the other: the human's assumptions against the agent's evidence, the agent's proposed direction against the human's clarified intent.
Formalization. The agreed change is written down as linked artifacts: the **proposed graph**, the **Test Manifest** (Given/When/Then descriptions of intended behavior — a normal, separate contract from the graph, not correlated to graph nodes by any shared ID scheme), a **Formalization Matrix** (a simple mapping showing that

every piece of stated intent has a corresponding graph edit or test, so nothing agreed to in the Brief silently fails to show up in the plan), and, if the change surfaces new vocabulary, updates to the **Lexicon** (see Shop Rules, below).
Output: a proposed graph, a Test Manifest, a Formalization Matrix, and human sign-off.
Exit criteria (Definition of Ready for Build): the Brief is addressed, Assumption Corrections have been acknowledged, the Lexicon is current, and the human has explicitly signed off. No code is written before this point.

Build

Purpose: implement the agreed change.
Participants: human + agent.
Trigger: Blueprint sign-off (or a Sketch go-ahead).
Inputs: the proposed graph; the Test Manifest.
What happens: implementation proceeds against the plan. On completion, two independent gates run: **Fidelity Check** (does the regenerated graph's structure match the proposal) and the **Test Manifest** (does the behavior match). These are separate contracts on purpose — conflating structural and behavioral correctness into one check would make each less precise.
Output: working code; if both gates pass, a **Gate Commit** — a commit specially marked for coinciding with a verified checkpoint. This does not replace ordinary incremental commits along the way; it's a marker on top of normal git hygiene, not a restriction on it.
Exit criteria (Definition of Done): Fidelity Check passes, the Test Manifest passes, the code is merged. If either gate fails and the cause traces back to the plan rather than the implementation, work returns to Blueprint, not just to another Build attempt.

Fidelity Check

Purpose: continuous structural verification.
Participants: automated; findings surface to human + agent.
Trigger: every Build attempt, and on an ongoing basis afterward.
Inputs: the proposed graph; the regenerated graph from the real, current code.
What happens: a structural diff — call shape and data shape only. The graph's scope stays strict on purpose: behavioral correctness belongs to the Test Manifest.
Output: a diagnostic signal — pass, or a specific structural mismatch. Never a ranking or performance metric (Deming, Point 11).
Exit criteria: none — this is a standing check, not a one-time gate, though it must be green for a Gate Commit specifically.

Punch List

Purpose: a heavier, staged review for higher-risk changes, beyond what continuous Fidelity Check alone covers.
Participants: human + agent.
Trigger: a change flagged higher-risk under Shop Rules.
Inputs: the current state of the change; any open Fidelity Check findings.
What happens: remaining discrepancies between proposed and actual are enumerated and worked through one at a time.

Output: a cleared list.
Exit criteria: the punch list is empty. Proceeds to Handover.

Handover

Purpose: deploy.
Participants: human + agent, or deploy tooling.
Trigger: a clear Punch List (or, for lower-risk changes, a passing Fidelity Check alone).
Inputs: the built, checked code.
What happens: deterministic IDs are injected into the running system.
Output: a live system now capable of emitting Execution Traces; the graph becomes the map those traces get plotted onto.
Exit criteria: deployment complete. In Service begins immediately.

In Service

Purpose: ongoing production monitoring — is the live system still doing what was proposed.
Participants: automated collection; human + agent review as needed.
Trigger: continuous, from Handover onward.
Inputs: Execution Traces streaming from the live system; the graph as the map they're plotted onto.
What happens: traces correlate to specific graph nodes by deterministic ID (this correlation is real and established — it's how the graph becomes a map at all) and can be queried for performance, staging comparisons, or KPI capture.
Output: an ongoing, queryable record of real production behavior, feeding Service Call, Renovation, and future Sketches/Blueprints.
Exit criteria: none — standing state for as long as the system runs.

Service Call

Purpose: debugging, grounded in what actually happened rather than a guess from a stack trace alone.
Participants: human + agent, whoever's responding.
Trigger: an incident, a bug report, an alert.
Inputs: the relevant Execution Trace(s), by journey ID or node ID.
What happens: the call chain is reconstructed via parent span references; the exact node and source location are identified; a fix is proposed grounded in the trace, not a hypothesis.
Output: a diagnosed root cause and a proposed fix, which typically re-enters the loop as a new Sketch or Blueprint.
Exit criteria: issue resolved; any resulting code change goes through its own Sketch/Blueprint → Build cycle rather than being patched outside the loop.

Renovation

Purpose: refactoring guided by real usage rather than the original Blueprint alone.
Participants: human + agent.
Trigger: a decision to refactor — prompted by In Service findings, accumulated Punch List debt, or a standing team decision.
Inputs: historical Execution Traces for the area being refactored.
What happens: traces are queried for actual call frequency and real contexts; the refactor is then planned through a normal Blueprint, with that evidence in hand instead of assumptions.

Output: a refactor plan grounded in production evidence.
Exit criteria: the refactor passes its own Fidelity Check and Test Manifest, same as any other Build.

Walkthrough

Purpose: onboarding.
Participants: whoever's onboarding, human or agent.
Trigger: a new team member or newly-instantiated agent joining work on a codebase.
Inputs: the current graph; historical Execution Traces.
What happens: the graph is read for structure, the traces for real behavior — together, rather than reading source files sequentially.
Output: a working understanding of the codebase's actual structure and behavior.
Exit criteria: informal — done when the person or agent can operate independently.

Shop Rules (cross-cutting, not a ceremony with its own trigger or exit)

The standing policy governing which ceremony a change requires (Sketch vs. Blueprint), where an agent may propose changes unsupervised versus needs a human in the loop, and how the handoff protocol works day to day. Also where **human stop-and-redirect authority** lives — exercisable at any ceremony, always, regardless of how much of the loop becomes automatable over time.
Shop Rules also govern the **Lexicon**: a living glossary of the terms, module names, and patterns the codebase actually uses, extracted during Survey and adopted during Calibration. If a human's own vocabulary drifts from the Lexicon mid-conversation, that's treated as a real signal worth pausing for — not pedantry. A team speaking past its own codebase's names for things is an early symptom of the same drift this whole discipline exists to catch.

The Prototype (a place, not a ceremony)

The shared surface where Sketching and Blueprinting happen. Always an edited duplicate of the real graph, never a blank canvas — which is the reason it isn't called one.

Guardrails

No gate-skipping. Blueprint needs sign-off before Build starts; Build needs both Fidelity Check and the Test Manifest to pass before a Gate Commit.
The graph's scope stays strict. Structure only — call edges, containment, inheritance, data shape — never behavior.
A diagnostic signal is never a quota. Fidelity Score and anything like it exists for the pair, not for ranking people.
A detected mismatch is signal, not blame — in the tooling's language and behavior, not just in this document.
The Lexicon is real, not decorative. A human overriding it with private vocabulary mid-Blueprint is treated as a symptom worth stopping for.
The human keeps stop-and-redirect authority at every ceremony, regardless of how much becomes automatable over time.
T2D2 checks whether the build matches the proposal — not whether the proposal itself was right. A deliberately deferred scope boundary, not an oversight.
The methodology stays free. Any commercial tooling sits alongside it, never gates it.

Part II: Supporting research

The claims below are drawn from research compiled and summarized during this project's development, not independently re-verified paper-by-paper here — treat citations as pointers to follow up on directly, not as this document's own primary research. Each entry states the finding and, separately, why it matters for T2D2 specifically, since those are different claims and worth keeping distinct.

On reviewing the actual codebase before planning

Spec Kit Agents (arXiv:2604.05278) found that agents operating on large, evolving repositories go "context blind," producing hallucinated APIs and architectural violations — and that read-only research hooks grounding each planning stage in actual repository evidence improved judged output quality by +0.15 on a 1–5 scale ($p < 0.05$), while maintaining 99.7–100% repository-level test compatibility. *Why it matters here:* this is close to a direct empirical case for the Survey stage inside Blueprint — the agent's research phase isn't ceremony for its own sake, it's the specific intervention this research found effective.

Pre-Execution Codebase Exploration research reported that agent failures correlate strongly with underspecified problem statements, and that a lightweight exploration phase before the main task — adding reproduction steps and targeted exploration hints — improved resolution rates by 20% on SWE-bench. Claude Code's own documented best practice ("explore first, then plan, then code") reflects the same finding independently. *Why it matters here:* independent convergence on "look before you plan" from a different angle than the Spec Kit Agents work — two separate lines of evidence for the same design choice.

De-Hallucinator (arXiv:2401.01701) found that LLMs are frequently unaware of the code that already exists in a specific project, leading them to invent or misuse APIs — and that iterative grounding against the actual project improved edit distance by 23.3–50.6% and correct API usage recall by 23.9–61.0%. *Why it matters here:* directly supports treating the graph (and the Survey's direct file reads) as the grounding mechanism, rather than trusting a model's general training knowledge of "how a library like this usually works."

Structured API-contract grounding research reported pass rates improving from 84.8% to 98.4% when agents worked against compact, explicit API contracts rather than unstructured context. *Why it matters here:* the closest direct precedent for the Formalization Matrix's whole premise — that making the contract between intent and implementation explicit and checkable, rather than implicit, measurably changes outcomes.

On commit discipline

Research on agent reliability found that trajectory divergence tends to happen early — often within the first 1–1.5% of generated tokens — and that once an agent goes off-track, that deviation compounds into a substantially different outcome rather than self-correcting. Separately, research on human oversight found that supervision concentrated at the *output verification* point is more effective than constant supervision, with frequency beyond that yielding diminishing returns. *Why it matters here:* together, these are the case for Gate Commits over time-based commits — if the meaningful failure happens early and compounds, catching it at a verified checkpoint matters more than how often you commit in between.

On multi-agent architectures — the evidence is genuinely mixed, not a case either way

This matters enough to state plainly rather than cite selectively: the research does not clearly favor single-agent or multi-agent approaches in general, and T2D2 takes no position on which a team should use (see "a note on agentic architecture" in the field guide). Findings pointing toward multi-agent value: a 2025 study on

specialized multi-agent systems reported a 65.5-percentage-point task-success improvement over a single agent on complex tasks, and the SEMAP protocol reported up to a 69.6% reduction in total failures for function-level development using explicit behavioral contracts and structured handoffs between agents. Findings pointing the other way: a controlled study across 260 configurations and six benchmarks found single-agent baseline capability to be the strongest predictor of whether adding coordination helps or hurts, with capable single agents often outgrowing any benefit from collaboration; another study found multi-agent collaboration yielding only marginal gains (0–5 points) while costing 3.4–5.3× the latency and 4.2–4.6× the tokens. *Why it matters here:* the honest reading is that multi-agent coordination helps on complex, multi-step, role-diverse tasks and actively hurts on simpler ones — which is itself the argument for T2D2 staying agent-architecture-agnostic rather than prescribing a specific setup. Shop Rules, not the methodology, is where a team should encode its own answer to this question for its own tasks.

On drift and self-correction risk

The same trajectory-divergence research found that each "off-canonical" step an agent takes raises the probability of the next step also being off-canonical by roughly 22.7 percentage points — a self-reinforcing pattern, not a random one. Separately, reported production experience with multi-agent systems attributes most real-world failures to looping supervisors, runaway token costs from error-recovery loops, and intermittent, hard-to-reproduce failures rather than to model capability limits. *Why it matters here:* this is the empirical case against letting an agent "self-heal" indefinitely without a checkpoint — it's also the case for the graph functioning as a canonical reference path, since the research explicitly frames adherence to a canonical path as what distinguishes successful runs from failed ones.

On human oversight

One study found an unconstrained multi-agent baseline produced critical failures in 72% of runs, dropping to 16% once organized around three commitments: agents restricted to reasoning (not unchecked action), structured verification, and human oversight gates. A separate "dynamic intervention" approach reportedly matched full human-oversight task success (98.2%) while requiring human intervention on only 14.5% of decision steps. *Why it matters here:* this is the strongest evidence found for two design choices together — keeping the human's stop-and-redirect authority (guardrail 6) *and* not requiring it to be exercised constantly. The ceremonies are built to concentrate human attention at Blueprint sign-off and gate checks, not to demand continuous supervision.

On independent validation that the underlying problem is real

Covered in more depth in the industry primer, worth restating briefly here: Meta's own applied research (ARCTIC, arXiv:2607.29516, July 2026) states the motivation for their tool in terms close to this project's own founding problem — that AI coding agents now generate code faster than traditional peer review can absorb. Aviator Verify, Tricentis AI Workspace, and pre-code planning tools like Intent and Kiro are independently converging on pieces of the same problem from different angles, none yet combining a compiler-accurate structural graph, live telemetry correlation, and a pre-code comprehension ceremony in one loop. *Why it matters here:* this is external confirmation, from outside this project, that the problem is real and worth solving at scale — not just an internal hypothesis.

Part III: Partner diligence — hard questions, honestly answered

A capable design partner or investor will ask questions like these before committing anything. The discipline that matters here is the same one this whole document is trying to model: answer what's actually grounded, be explicit about what's a design commitment rather than a measured fact, and say plainly when something is a genuine open gap rather than papering over it with a plausible-sounding answer.

"Show me the false positive/negative rate of your graph generation for my specific language and framework stack." Honestly: this data doesn't exist yet, and claiming it does before real measurement would be the exact kind of overclaiming this project's own guardrails argue against. What can be said is the commitment, not the number — Fidelity Check's value depends entirely on its precision, so validating false-positive/negative rates against a design partner's real stack should be one of the first things measured in a pilot, not asserted in advance. If a vendor answers this question with a confident number and no methodology behind it, that's the more concerning answer, not the honest "we don't have that for your stack yet."

"Can the agent query the graph without dumping the entire thing into the prompt?" Yes — this one is grounded, not aspirational. The semantic layer is built around targeted queries (symbol search, callers/callees lookup, reference finding, type hierarchy, path tracing, item inspection) exposed as individually callable tools, not full-graph serialization into a prompt. This was part of the system's design from early on, specifically because context-window limits make full-graph dumping impractical for any codebase of real size.

"What happens when the agent proposes a graph edit that is syntactically valid but semantically impossible — e.g., a circular dependency? Does validation reject it immediately, or does it pass through to Build and fail later?" Edits are applied through an MCP tool call that validates before applying — this is not an unvalidated write path. That validation is basic and structural, and it deliberately does not attempt to catch something like circular dependencies, because that class of pattern isn't reliably a defect: recursion with a proper exit condition is structurally circular and also completely legitimate code. A check that flagged all structural circularity would produce enough false positives on valid recursive patterns to erode trust in the tool faster than it prevented real problems. This is a deliberately scoped limit, not an oversight — the same category of honest boundary as guardrail 7's stance on spec-correctness adjudication: T2D2 checks structure and behavior against what was proposed; whether a given structural pattern is actually a problem in context is still a human-plus-agent judgment call, not something pushed onto automatic validation.

"How do you handle generated code (protobuf, OpenAPI clients, etc.) that changes the graph without human intent?" The Gate Commit discipline exists, but T2D2 deliberately doesn't prescribe a development-pipeline strategy for this case — it's a Shop Rules decision for each team, not something the methodology mandates centrally. When generated code changes the graph in a way that wasn't part of the stated plan, a team has the same two options the loop already provides for any unexpected mismatch: treat it as a Service Call — collaborative debugging with the agent to understand why the graph moved — or roll back, adjust direction, and re-enter Sketch/Blueprint. In the founder's own practice, these situations get handled by pairing directly with the agent rather than letting it resolve on its own, since the underlying cause is usually either genuine complexity or the agent working from incomplete context — exactly the case where human judgment is least substitutable. This is a good candidate for a team's own documented Shop Rule (for instance, "generated-code-driven graph changes always trigger a Service Call by default") rather than a global methodology mandate.