[Draft] AI-Native Software Delivery

A text-first companion for sharing this delivery model with an LLM.

Introduction

Two things can be true: AI is over-hyped and AI is a fundamentally transformative technology.

As a technology, AI allows humans to do things they couldn't do previously. "AI-native" refers to the new approaches and methodologies that wouldn't be possible without the use of AI.

In the software development lifecycle:

1. Humans using AI to take on work that was previously out of reach. A refactor that would take months rarely fits within the business's appetite. If AI can finish it in days, it does.
2. AI autonomously running various steps in the software delivery process.
3. AI orchestrating the delivery process itself, and suggesting or driving improvements to it.

To make these possible, keep these principles at the forefront:

- Make the entire delivery system visible. The system should be legible to both humans and LLMs so they can understand where work is, how it moves, and what context governs it. What may be obvious to a human isn't obvious to AI unless it has access to the appropriate context.
- Articulate the software's contract. Moving faster without losing quality requires being able to work a higher level of sophistication on what the software is meant to do.
- Manage to the constraint. Work piles up at the bottlenecks, so increasing throughput is an exercise in managing to those bottlenecks. AI efficiency easily moves the bottleneck to different parts of the system.
- Consider the human element. AI can generate work far faster than people can thoughtfully review it, and changing human systems requires change management. Make it easy for humans to say "wow, this is amazing!"
- Be deliberate about the final solution. Sometimes AI-based intelligence is the right piece in the system, while other times it's better for AI to just write a deterministic script.

This page is heavily inspired by Eliyahu Goldratt's *The Goal* (https://en.wikipedia.org/wiki/The_Goal_(novel)), a book which judges a business by three measures:

- Throughput: the rate at which the system makes money through sales.

- Inventory: money tied up in things the system intends to sell.
- Operational expense: money spent turning inventory into throughput.

High-level stages

At a high level, the software delivery system is anchored by these questions:

1. Identify: Which opportunities are worth acting on?
2. Define: What is the appropriate product solution to the business need?
3. Execute: What is the best technical solution to address the product requirements?
4. Verify: Is the implementation correct, secure, understandable, and altruistic?
5. Release: Did the solution successfully reach customers?
6. Learn: Based on how customers use the solution, what should be done next?

Select a stage in the flow chart to explore where human judgment is required, the typical constraints, and signals to watch.

IDENTIFY

Which opportunities are worth acting on?

Collect and compare signals from customers, support, sales, product, operations, incidents, and strategy before committing delivery capacity. The goal is to distinguish important opportunities from noise and create enough shared context to make a deliberate priority decision.

Human judgement required:

- Decide which signals represent meaningful customer or business opportunities.
- Compare opportunities with different levels of evidence, urgency, and upside.
- Make tradeoffs when organizational capacity is limited.
- Decide when there is enough evidence to act versus when more discovery is needed.

Typical constraints:

- Too many signals competing for limited attention.
- Incomplete or conflicting evidence.
- Weak connection between incoming requests and broader strategy.
- Slow decision-making because ownership or decision rights are unclear.

Signals to watch:

- Time from signal to priority decision
- Age of high-value untriaged signals
- Percentage of prioritized work later abandoned or substantially reprioritized
- Percentage of shipped priorities that achieve their intended customer or business outcome

---

DEFINE

What is the appropriate product solution to the business need?

Turn a prioritized opportunity into a bounded, sufficiently solved product approach. Clarify the problem, set an appetite, resolve the major rabbit holes, and define enough of the solution that the team can begin without removing their ability to exercise judgment during execution.

Human judgement required:

- Clarify the underlying problem before committing to a solution.
- Decide the appetite: how much time and capacity the opportunity deserves.
- Choose the product approach and make the important scope tradeoffs.
- Identify rabbit holes, no-gos, and unresolved risks.
- Decide when the work is sufficiently shaped to begin without over-specifying it.

Typical constraints:

- Ambiguous or poorly understood customer problems.
- Premature convergence on a solution.
- Too much specification, which removes execution-team judgment.
- Too little definition, which pushes major product decisions into implementation.
- Unresolved rabbit holes that make execution unpredictable.
- Stakeholder pressure to expand scope beyond the appetite.

Signals to watch:

- Time from prioritized opportunity to sufficiently shaped work
- Scope churn after execution begins
- Frequency of major product or architecture decisions discovered during execution
- Work exceeding its original appetite

- Projects that remain on the uphill uncertainty slope too long

---

EXECUTE

What is the best technical solution to address the product requirements?

Translate the shaped product approach into a maintainable technical implementation. Decompose the work into coherent changes, preserve system context, and integrate parallel work without creating more downstream review and rework than the organization can absorb.

Human judgement required:

- Choose the technical approach and make implementation tradeoffs.
- Decide how to decompose the work into coherent, reviewable changes.
- Recognize when new information requires changing the original plan.
- Resolve integration conflicts and architectural ambiguity.
- Decide when parallelization helps versus when it creates excess coordination cost.

Typical constraints:

- Poor decomposition that creates oversized or tangled changes.
- Loss of system context while engineers or agents work in parallel.
- Too much work in progress relative to review and integration capacity.
- Integration conflicts between concurrent changes.
- Rework caused by implementation discovering unresolved product or architecture questions.

Signals to watch:

- Time from shaped work to review-ready change
- Active work in progress
- Average change size
- Percentage of changes requiring substantial rework before review
- Blocked time caused by dependencies or integration conflicts
- Ratio of generated work to work actually absorbed downstream

---

VERIFY

Is the implementation correct, secure, understandable, and altruistic?

Build enough confidence to release the change safely. Combine deterministic checks, agentic review, and human judgment to catch defects, assess risk, and decide whether the implementation is good enough to ship.

Verification asks whether the change is correct, secure, understandable, maintainable, and beneficial to the surrounding system-not merely whether the tests pass.

Human judgement required:

- Judge maintainability, architectural fit, and implementation tradeoffs.
- Decide whether product and design intent have been preserved.
- Evaluate ambiguous risks and unusual edge cases.
- Determine when automated evidence is sufficient and when deeper human review is warranted.
- Decide whether a change is safe to merge or requires revision.

Typical constraints:

- Human review capacity cannot keep pace with implementation throughput.
- Large or poorly explained changes create excessive cognitive load.
- Automated checks provide coverage without enough confidence in overall correctness.
- Review happens too late to cheaply change the approach.
- False positives from automated or agentic review consume scarce human attention.

Signals to watch:

- Queue time before human review begins
- Time from review-ready to merge-ready
- Number of revision cycles per change
- Escaped defects and post-merge regressions
- Percentage of review findings caught deterministically, agentically, and by humans
- Human review time per change
- False-positive rate from automated and agentic checks

RELEASE

Did the solution successfully reach customers?

Move a verified change into production and expose it to customers through the appropriate release path. Deployment and release may be separate events, using feature flags, staged rollout, or coordinated release windows to manage risk.

Human judgement required:

- Release confidence, operational judgment, and incident response when needed.
- Decide when to use feature flags, staged rollout, or targeted exposure.

Typical constraints:

- Operational confidence and release coordination.
- Poor feature-flag hygiene can leave stale flags, hidden complexity, and ambiguous product state.

Signals to watch:

- Time from merge to production
- Deployment frequency
- Failed deployment, rollback, or remediation rate

LEARN

Based on how customers use the solution, what should be done next?

Interpret product usage, customer feedback, support, observability, and business outcomes to decide what should change next. Learning should improve both the product and the delivery system that produced it.

Human judgement required:

- Customer and product interpretation; prioritization of the next response.

Typical constraints:

- Evidence quality and the capacity to act on learning.

Signals to watch:

- Customer feedback and support trends
- Product adoption and retention
- Incidents, errors, and performance regressions
- Recurring operational anomalies
- Time from meaningful signal to explicit decision

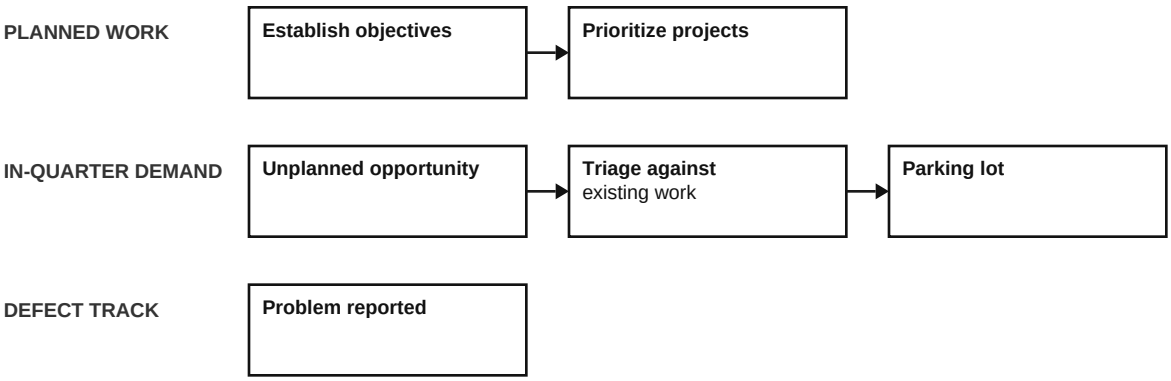
Step-by-step flow

Within each stage, the software delivery system has a series of steps which produce artifacts and make decisions.

This is much less of a linear process. Some work can proceed in parallel, some steps require a return path for rework, and work-in-progress can get blocked by missing requirements, dependencies, or new discoveries.

Select a node to inspect its purpose, activities, gate, and next possible move.

Identify flow



IDENTIFY PLANNED WORK - ESTABLISH OBJECTIVES

Translate company and product strategy into a small set of objectives that guide work during a given period.

- Gather relevant customer, market, product, and business evidence
- Identify the most important objectives for the period
- Make strategic tradeoffs between competing priorities
- Define clear measures of success
- Check objectives for ambiguity, overlap, and premature solutioning
- Connect objectives to the evidence and prior learning that motivated them

IDENTIFY PLANNED WORK - PRIORITIZE PROJECTS

Decide which opportunities deserve delivery capacity during the period.

- Compare candidate work against the current objectives
- Evaluate customer impact, business value, urgency, risk, and opportunity cost
- Make explicit tradeoffs between competing opportunities
- Decide what will be pursued now, deferred, or declined
- Keep the amount of committed work within realistic organizational capacity

IDENTIFY IN-QUARTER DEMAND - UNPLANNED OPPORTUNITY

Capture an unplanned opportunity with enough context to compare it against existing work.

- Capture the source, context, urgency, and expected value
- Clarify the customer or business impact
- Distinguish real deadlines from perceived urgency
- Connect the opportunity to related work, prior decisions, and existing evidence
- Identify missing information or duplicate requests

IDENTIFY IN-QUARTER DEMAND - TRIAGE AGAINST EXISTING WORK

Compare the unplanned opportunity with existing work and decide whether it should be shaped or retained for later consideration.

- Compare the opportunity with current commitments and objectives
- Evaluate urgency, impact, opportunity cost, and switching cost
- Identify what existing work would be delayed or displaced
- Decide whether the opportunity should move into shaping or remain parked

IDENTIFY IN-QUARTER DEMAND - PARKING LOT

Retain the opportunity with enough context to revisit it without consuming active delivery capacity.

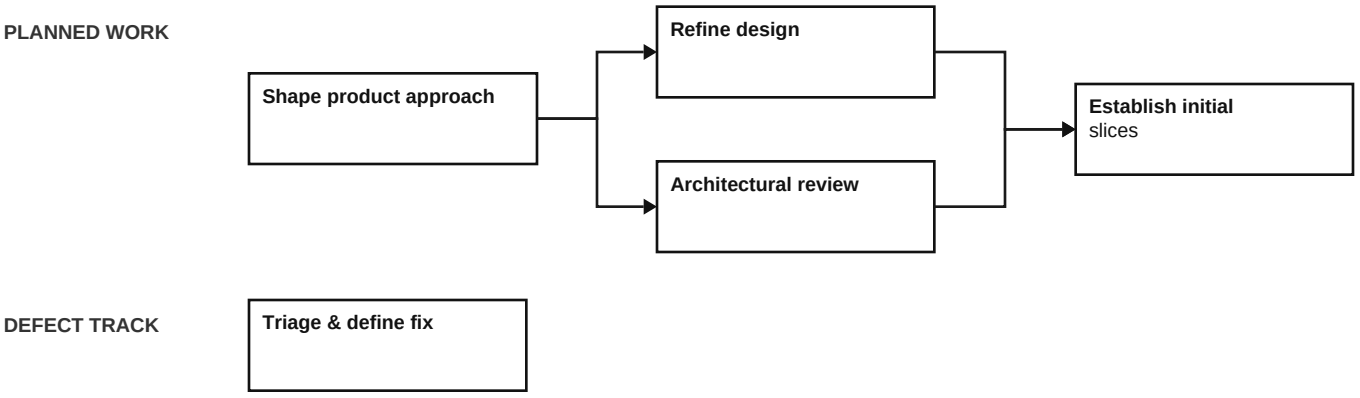
- Record the opportunity, evidence, and triage rationale
- Capture the conditions that would make reconsideration appropriate
- Link related work, customers, and known constraints
- Review it during future prioritization when capacity or context changes

IDENTIFY DEFECT TRACK - PROBLEM REPORTED

Capture evidence of something behaving incorrectly and establish enough context to determine its impact and whether investigation is warranted.

- Capture observed behavior and expected behavior
- Record customer, operational, security, or data impact
- Correlate related reports, incidents, support tickets, logs, or recent changes
- Deduplicate related reports
- Establish severity and urgency without prematurely prescribing the fix

Define flow



DEFINE PLANNED WORK - SHAPE PRODUCT APPROACH

Turn a prioritized opportunity into a bounded, sufficiently solved product approach.

- Clarify the underlying customer or business problem
- Set an appetite for how much time and capacity the opportunity deserves
- Explore and choose the product approach
- Define the important scope boundaries
- Identify rabbit holes, no-gos, and unresolved risks
- Capture enough context for design and architecture decisions without over-specifying implementation

DEFINE PLANNED WORK - REFINE DESIGN

Resolve the important interaction, usability, and product-design questions that would otherwise create uncertainty during execution.

- Define the key user flows and interactions
- Resolve important states, edge cases, and behavioral details
- Test assumptions with lightweight prototypes or examples where useful
- Make explicit product and design tradeoffs
- Identify remaining questions that can safely be deferred until execution

DEFINE PLANNED WORK - ARCHITECTURAL REVIEW

Resolve the architectural decisions and technical risks that materially affect the shape of the implementation.

- Identify the major systems and boundaries affected
- Choose the high-level technical approach
- Evaluate important tradeoffs and alternatives
- Surface migration, compatibility, security, performance, or operational risks
- Record architectural decisions that future implementation work needs to understand
- Distinguish decisions that must be made now from implementation details that can remain flexible

DEFINE PLANNED WORK - ESTABLISH INITIAL SLICES

Break the shaped approach into the first coherent slices of work needed to begin execution.

- Identify vertical slices that deliver coherent progress
- Capture the intent, relevant context, and important constraints for each slice
- Order work where dependencies genuinely require sequencing
- Keep slices small enough to implement and review without fragmenting the overall design
- Create only enough initial work to begin; do not exhaustively ticket the entire project

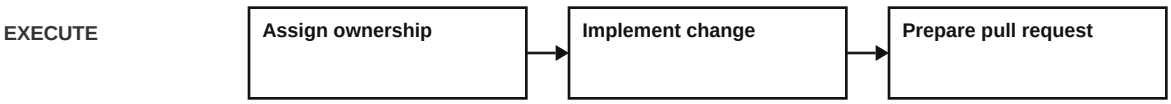
DEFINE DEFECT TRACK - TRIAGE & DEFINE FIX

Understand the failure well enough to define a safe, appropriately scoped response.

- Reproduce or characterize the failure
- Determine severity, scope, and affected surfaces

- Identify likely causes and validate them where possible
- Decide whether the appropriate response is a fix, rollback, mitigation, or further investigation
- Define the smallest safe change that addresses the problem
- Identify regression coverage needed before execution

Execute flow



EXECUTE - ASSIGN OWNERSHIP

Establish clear responsibility for delivering a bounded slice of work.

- Pull in the issue, relevant product context, architectural decisions, and dependencies
- Confirm the intended outcome and important constraints
- Identify ambiguities that need to be resolved before implementation
- Establish a working branch, change set, or isolated execution environment
- Coordinate ownership when multiple contributors or agents will work in parallel

EXECUTE - IMPLEMENT CHANGE

Produce the smallest coherent implementation that satisfies the defined intent while preserving the surrounding system.

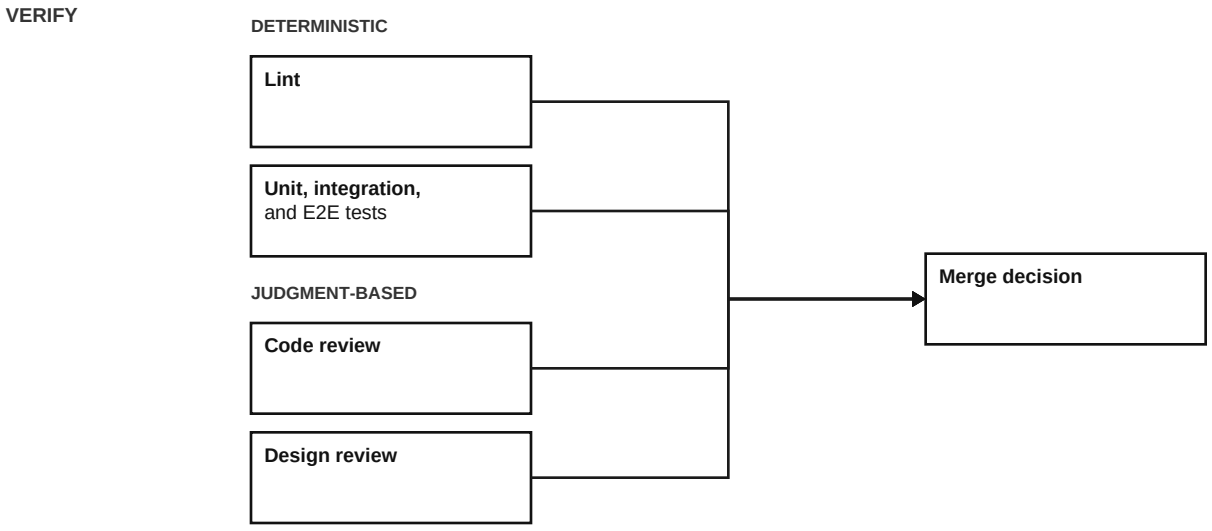
- Implement the required product and technical behavior
- Add or update tests alongside the implementation
- Follow existing architecture, conventions, and system boundaries
- Continuously run fast feedback checks during development
- Re-evaluate the approach when implementation reveals new information
- Keep the change focused and separate unrelated improvements
- Update relevant documentation or system context as decisions are made

EXECUTE - PREPARE PULL REQUEST

Package the implementation so downstream verification can evaluate it quickly and accurately.

- Produce a clear summary of what changed and why
- Connect the implementation back to the original intent and relevant decisions
- Highlight important tradeoffs, risks, migrations, and known limitations
- Include evidence of testing and expected behavior
- Keep the change small and comprehensible enough to review
- Remove unrelated changes and obvious noise before requesting review

Verify flow



VERIFY - LINT

Catch mechanically detectable issues before they consume broader review attention.

- Run linting and formatting checks
- Run type checking and static analysis
- Enforce repository conventions and policy rules
- Detect applicable dependency, schema, or compatibility violations

VERIFY - UNIT, INTEGRATION, AND E2E TESTS

Verify that the change behaves correctly at the relevant levels of the system.

- Run unit tests for local behavior
- Run integration tests across system boundaries
- Run E2E tests for critical user workflows

- Validate migrations, contracts, and compatibility where relevant
- Investigate unexplained failures rather than simply retrying them

VERIFY - CODE REVIEW

Evaluate whether the implementation is coherent, maintainable, appropriately scoped, and suitable for the surrounding system.

- Evaluate implementation tradeoffs and architectural fit
- Assess readability and maintainability
- Check whether the change is appropriately scoped
- Look for regression risks and missing coverage
- Challenge questionable assumptions or unnecessary complexity
- Compare the implementation with the original intent and relevant decisions

VERIFY - DESIGN REVIEW

Evaluate whether the implemented experience preserves the intended product and design behavior.

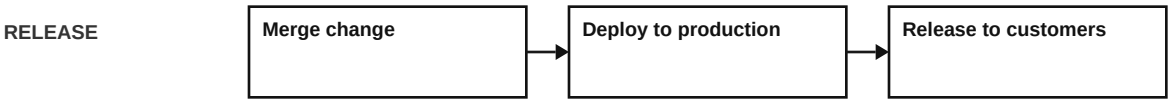
- Review important user flows and interaction states
- Compare implementation with the shaped product intent
- Check edge cases and unintended behavioral changes
- Evaluate UX quality, consistency, and accessibility where relevant
- Confirm that implementation compromises remain acceptable

VERIFY - MERGE DECISION

Decide whether the available verification evidence is sufficient to integrate the change.

- Evaluate the results of required deterministic and judgment-based verification
- Confirm that the required verification for the change’s risk profile is complete
- Decide whether unresolved findings require revision
- Determine whether additional review is warranted
- Record the evidence supporting the decision where useful

Release flow



RELEASE - MERGE CHANGE

Integrate a verified change into the releasable codebase.

- Confirm required verification has completed
- Merge the approved change
- Resolve final integration conflicts if necessary
- Trigger the appropriate build and deployment pipeline
- Preserve traceability between the change, its verification evidence, and resulting release

RELEASE - DEPLOY TO PRODUCTION

Make the change operationally available in the production environment.

- Build and deploy the production artifact
- Apply required migrations or configuration changes
- Verify deployment health
- Confirm observability and rollback mechanisms are functioning
- Leave customer exposure disabled where release is intentionally separate from deployment

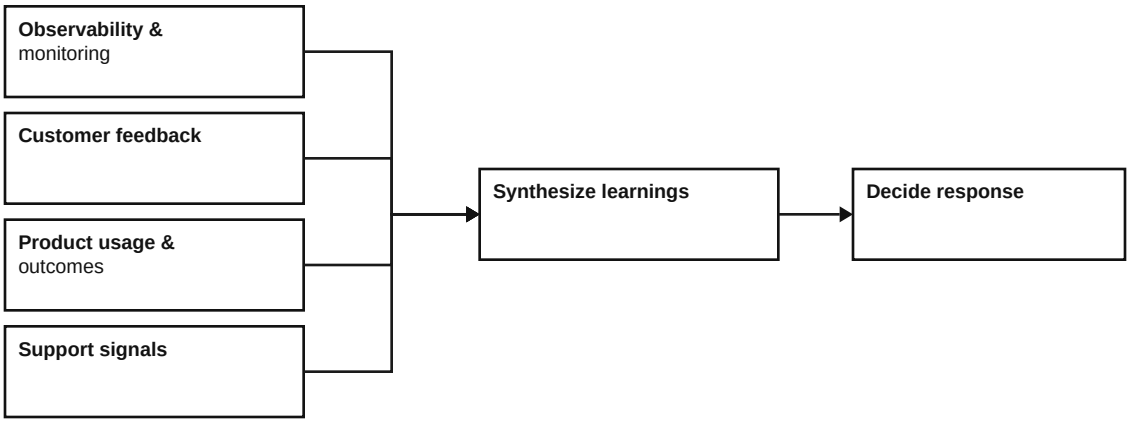
RELEASE - RELEASE TO CUSTOMERS

Expose the production change to the intended customers through the appropriate rollout strategy.

- Enable immediately when deployment and release are the same event
- Enable a feature flag when deployment intentionally precedes release
- Roll out progressively by percentage, cohort, account, geography, or another appropriate boundary
- Coordinate customer communication, migrations, or enablement where required
- Watch early rollout signals and pause or reverse exposure when necessary
- Remove temporary rollout controls when the release is complete

Learn flow

LEARN



LEARN - OBSERVABILITY & MONITORING

Understand whether the production system is behaving correctly and reliably.

- Monitor errors, latency, availability, and resource behavior
- Detect incidents, regressions, and operational anomalies
- Correlate changes in system behavior with recent releases
- Capture logs, metrics, traces, and other diagnostic evidence
- Surface recurring operational failure modes

LEARN - CUSTOMER FEEDBACK

Understand what customers say, experience, expect, and struggle with.

- Gather direct customer conversations and research
- Capture qualitative reactions to shipped changes
- Collect surveys, interviews, and other explicit feedback
- Identify recurring needs, confusion, and unmet expectations
- Preserve relevant customer context rather than reducing everything to a feature request

LEARN - PRODUCT USAGE & OUTCOMES

Understand what customers actually do and whether shipped work produces its intended outcomes.

- Measure adoption and ongoing usage
- Observe important product behaviors and funnels

- Compare expected behavior with actual behavior
- Evaluate outcome measures defined when the work was shaped
- Segment behavior by relevant cohort, account, or customer type
- Identify unexpected or unintended usage patterns

LEARN - SUPPORT SIGNALS

Surface recurring customer friction, defects, questions, and unmet needs appearing through support.

- Aggregate recurring support issues and requests
- Identify sudden changes in ticket volume or topic
- Connect support issues to product areas and recent releases
- Distinguish isolated cases from recurring patterns
- Preserve customer impact and context when summarizing support evidence

LEARN - SYNTHESIZE LEARNINGS

Turn production, product, and customer evidence into a coherent understanding of what happened and why it matters.

- Combine evidence across different sources
- Identify recurring patterns, anomalies, and contradictions
- Compare actual outcomes with the intent of shipped work
- Distinguish isolated anecdotes from broader signals
- Surface unexpected behavior and unintended consequences
- Identify where evidence remains insufficient or conflicting

LEARN - DECIDE RESPONSE

Determine what, if anything, should change based on what was learned.

- Decide whether the evidence warrants action
- Determine whether the response is a new opportunity, defect, operational improvement, further investigation, or no action
- Capture the relevant evidence and reasoning behind the decision
- Feed actionable learning into the appropriate part of the delivery system
- Identify lessons that should change the delivery system itself

Modalities

Fundamentally, AI is a blob of computation that takes whatever information you put into it and spits out a computed result. Every time a human is interfacing with AI, they are just running a chunk of information through a processor.

The "seam", as Claude likes to say, is the boundary between humans and AI. It is continually evolving as AI becomes more capable and humans work at a higher level of abstraction.

When working on the seam, consider these predispositions of AI:

- Steering. AI doesn't require complete instructions before work begins. Sometimes, the fastest path is to give it enough context to get started and then provide incremental feedback as the AI progresses.
- Loops. AI is much better than humans at introspection and identifying improvement opportunities.
- Research. Set the AI loose on your research task; you don't need to know where the answer might be up front. AI is really good at going multiple layers deep and producing a result that will surprise you.

Also consider these predispositions of humans:

- Humans are much faster at reading and reacting, vs. providing their own input.
- Humans are better at recognizing quality than specifying it up front.
- Humans aren't particularly good at multi-tasking, holding a lot of context, or retaining the context for a long period of time.
- Humans can easily be biased by framing, sunk cost, or the context they currently hold. "Critique my work and identify the gaps" is a great loop for humans.
- Humans are much more likely to add things than remove things.
- Humans live within groups of humans. Decisions spend on alignment, incentives, relationships, and a variety of squishy factors.

And, don't forget about compounding. Some things that were impractical are now easily done, and they can provide the basis for a step change in improvement. For example:

- Substantially increasing E2E test coverage.
- Producing very detailed pull request descriptions.
- Prototyping a full solution (user-facing or architectural) before a decision is made on whether to implement it.
- Setting up entire systems (e.g. cloud environments).

Related reading

Self-driving products

Both PostHog (https://posthog.com/blog/self-driving-product) and Amplitude (https://x.com/spenserskates/status/2094496197901987945) are pivoting into "self-driving products", or agents modifying the codebase based on what they observe from the system. TBD whether this produces better outcomes - do PostHog and Amplitude using self-driving themselves produce outsized returns?

Tokens as a proxy for value

> I'm struck by one thing. Imagine for a moment that we understood team funding as a combination of salaries and AI spend. A team might look at itself and say, "Geez, we cost $1.5M a year in salaries plus $1M in tokens. What is our business case here?"
>
> I understand that this is somewhat utopian. In many companies, the goal is explicitly to use AI to eliminate some portion of that team, its managers, or the surrounding organization, based on some substitution math.
>
> But if you think about the problem at the level of the team + AI system, the solution is remarkably similar to the "alternative" described above. What demand are we serving? What causal model connects our work to differentiated value? What durable areas are we investing in? What assumptions have to hold? What evidence would make us increase, decrease, or redirect the investment?

John Cutler, TBM 437: Tokens, Hours, Points, and Other Curious Proxies (https://cutlefish.substack.com/p/tbm-437-tokens-hours-points-and-other)

Articulated as a set of statements:

> 1. We believe using AI for _____ will reduce _____ without materially increasing _____.
> 2. We believe AI will primarily augment/replace _____, resulting in _____.
> 3. We believe spending _____ on AI will be economically preferable to _____ because _____.
> 4. We believe making _____ cheaper/faster will not simply move the bottleneck to _____.
> 5. We believe increased AI-enabled production of _____ can be absorbed without creating excessive _____.

> 6. We believe the primary long-term asset created by this AI investment will be _____, rather than merely cheaper production.
> 7. We believe the human + AI system will outperform _____ on _____, after accounting for _____.
> 8. We believe the downstream costs of AI-generated _____ will remain below _____ over its useful lifetime.
> 9. We believe paying a premium for _____ will produce enough additional _____ to justify the premium.
> 10. We believe this investment works economically only if _____ remains true and _____ does not become the limiting constraint.

Agentic verification

> I personally feel that agentic verification is so important that I would unironically suggest building your own rich debugging tools, or even choosing a different tech stack, in order to have unfair advantages and extreme productivity in building software. As I mentioned earlier, giving agents the ability to verify their own work unlocks everyone in your organization to be able to contribute and validate that their changes actually work. The harder your tech stack is to debug and control, the more difficult it will be to use agents productively.

Lauren Tan, The Complete Guide to pstack Pt. 1 (https://x.com/poteto/article/2094457600259842065)

Anthropic’s AI-native SDLC playbook

Anthropic published an AI-Native SDLC playbook (https://claude.com/blog/the-ai-native-sdlc-playbook) that starts from the same premise as this page: "Code is no longer the bottleneck-the human-speed steps around it are." Its six stages (Plan, Design, Build, Test, Deploy, Maintain) map closely onto Identify, Define, Execute, Verify, Release, and Learn, and both treat delivery as a loop in which humans keep the decisions that require judgment.

| Anthropic | This page | How they compare |

| --- | --- | --- |

| Plan | Identify | The playbook captures an idea as intent.md right away and measures time to the first commit. This page starts earlier, with deciding which signals are worth acting on and when there’s enough evidence to commit capacity. |

| Design | Define | The playbook collapses requirements and design into one AI session, constrained by policy skills, which a product owner reviews. This page shapes the work with an appetite, rabbit holes, and no-gos, and deliberately leaves room for judgment during execution. |

| Build | Execute | The playbook requires an accepted plan.md before implementation and scales through parallel sessions per engineer. This page focuses on decomposing work into coherent, reviewable changes and keeping work in progress within review capacity. |

| Test | Verify | The playbook has each session check its own work, with evals in CI. This page asks whether a change is correct, secure, understandable, and altruistic, combines deterministic, agentic, and human checks, and treats human review capacity as the likely constraint. The playbook puts AI code review in Deploy; this page puts it in Verify. |

| Deploy | Release | The playbook enforces approval through hooks and tiered autonomy, where the agent "may act up to the production gate and cannot pass it." This page separates deploying from releasing, using feature flags and staged rollout to manage customer exposure. |

| Maintain | Learn | The playbook has monitoring breaches invoke an agent, which writes its diagnosis back as a new intent.md. This page interprets usage, feedback, and support signals to decide what to do next, and applies the learning to the delivery system as well as the product. |

Beyond the stages, the two diverge in two places.

The prescribed chain of markdown files isn’t durable. In the playbook, every stage commits an artifact the next stage reads: intent.md, then spec.md, then plan.md, with CLAUDE.md and REVIEW.md alongside. That’s a good audit trail for a single change, but it doesn’t hold up as a description of the software:

- Each file is a snapshot of one change at one moment. Once the code merges, nothing keeps spec.md true. The next change to the same behavior writes a new spec, and the old one quietly goes stale. The playbook’s own metrics hint at this, since it tracks how often the merged diff still matches the committed plan.md.
- The files are organized around changes, not around the system. To learn how a feature is supposed to behave, you’d have to reassemble it from every intent, spec, and plan that ever touched it.
- The playbook offers a legacy tracker as an alternative source of truth, which suggests the files aren’t meant to be the system of record either.

What should endure is the software’s contract: what it’s meant to do, kept in a place that tests, reviews, and agents can check against. What that system of record looks like is still an open question on this page.

It doesn’t address how people develop a shared understanding. The playbook’s governance has three layers: skills (advisory), hooks (deterministic), and human review (judgment). Each one is a control on a single change. None of them covers how a team comes to agree on the problem, the architecture, or what "good" looks like.

- A product owner reviews the spec rather than writing it. Reacting is faster than producing, but approving an artifact isn’t the same as a team agreeing on it.
- AI makes code cheap and architecture the hard part. When engineers don’t agree on an approach before implementation, code review turns into nitpicking at best and a restart at worst.
- The playbook measures concurrent sessions per engineer. That’s single-player mode. Multi-player mode, where several people and agents work on the same system, depends on ceremonies that build shared context: design walkthroughs, recorded talks about decisions, and decision records discussed before they’re written down.

Open questions

- Humans are offline for much of the day. What’s the most effective way to run AI during that time?
- Should you allow agentic review to automatically merge PRs?
- In organizations that don’t allow agents to automatically merge PRs, how can AI help a human quickly understand intent, risk, evidence, tradeoffs, and the parts of a change that deserve attention?
- When do you create custom UI or applications for different parts of the SDLC?
- What’s the maximum rate of change an organization can tolerate?
- What are the best visualizations for a human to review AI work?
- What’s the best way to delineate between autonomous bot actions, AI-supported human actions, and human-only actions?
- Right now, "how the software is supposed to work" is a kludge of PRDs (if you’re lucky), various test suites, and institutional knowledge. What might a higher-level system of record look like?
- Single-player vs. multi-player mode?
- What are the best ceremonies for humans to develop shared understandings?
- How do you continually evaluate whether token spend is worth the benefit?
- What’s the optimal approach for allocating token spend?

- Greatness cannot be planned (https://www.goodreads.com/en/book/show/25670869-why-greatness-cannot-be-planned). How does this fit into the model?

Changelog

How this page has evolved over time:

- 2026-10-05 - Expanded the comparison with Anthropic's AI-Native SDLC playbook in Related reading: where the two agree, a stage-by-stage comparison, why its chain of markdown files isn't durable, and why it doesn't address shared understanding.
- 2026-09-06 - Added a Related reading section, and a principle on articulating the software's contract, with an open question about what the higher-level system of record might look like.
- 2026-08-23 - Initial version.