A PRACTICAL GUIDE · 2026 EDITION

# Web Product Delivery Flow

How to take a web product from an unresolved idea to a working, measurable, maintainable system — and keep it that way.

**Barton Anderson** · Founder, Product Builder & Fractional CTO · bartonlanderson.com

## Why a delivery flow still matters

I wrote the original version of this guide years ago, drawing on the web development lifecycle I standardized for a game studio that had no established web practice. The tools have changed almost completely since then. The underlying problem has not: teams still lose months building the wrong thing well, or the right thing without a way to ship, measure, and operate it.

A delivery flow is not bureaucracy. It is a shared agreement about what has to be true before work moves forward, so decisions are made once, by the right people, with the evidence in front of them.

The phases below are written in order, but modern delivery is iterative. Small teams run them in tight loops — sometimes in a single week — and revisit earlier phases whenever the evidence changes. The point is not to finish a phase forever; it is to never skip one.

| PHASE | THE QUESTION IT ANSWERS |
|---|---|
| **1. Discovery** | What problem is worth solving, for whom, and how will we know it worked? |
| **2. Kickoff** | Does everyone share the same picture of the product and the constraints? |
| **3. Specification** | What exactly are we building, and what does “done” mean? |
| **4. Prototype** | Does the idea hold up when real people use it? |
| **5. Content** | What will the product say, and can people and search systems understand it? |
| **6. Build** | How do we turn the specification into reliable, reviewable software? |
| **7. Quality** | What evidence proves it works before real users depend on it? |
| **8. Launch** | How do we release safely, and how do we undo it if needed? |
| **9. Operate** | Is it available, correct, secure, and producing results? |
| **10. Improve** | What did we learn, and what changed in the world around us? |

## 1 Discovery and problem framing

Every cycle starts by understanding how the product serves the vision, the business objectives, and the people who will use it. Discovery should be fast and realistic, not exhaustive. Its job is to replace assumptions with the smallest amount of evidence needed to commit.

- **Users and jobs.** Who has the problem, what are they trying to get done, and what do they do today instead?
- **Outcomes and measures.** Define success as a change you can measure — leads, activation, retention, time saved — not a list of features.
- **Constraints.** Budget, timeline, regulation, privacy, existing systems, data, and the team you actually have.
- **Risks.** Name the riskiest assumption and decide how to test it cheaply before building around it.
- **Build, buy, or integrate.** Many components — authentication, payments, email, scheduling — are solved problems. Spend custom engineering only where it differentiates.

Involve design, engineering, operations, and whoever will support the product from the start. A plan that only one discipline has seen is not yet a realistic plan.

**INPUTS**

- Vision and business objectives
- User research, support data, analytics
- Existing systems, data models, and platforms

**OUTPUTS**

- Problem statement and success measures
- Constraints and risk list
- Recommended scope and build/buy decisions

## 2 Kickoff and shared vision

The kickoff is where the team leaves with the same picture of the product. It covers audience, positioning, brand, information hierarchy, navigation, the core user journeys, and what the product will deliberately *not* do.

Brainstorming is welcome, but the meeting should end with a written record of decisions, open questions, and owners. Decisions that live only in someone's memory get relitigated later at a much higher cost.

**INPUTS**

- Discovery outputs
- Team and stakeholder availability

**OUTPUTS**

- Decision log and open questions
- Roles and owners
- Preliminary requirements

## 3 Specification

A good specification is a living, versioned description of what will be built and how everyone will know it is done. It should be detailed enough to remove ambiguity and short enough that people actually read it.

### Product and experience

- User flows for each core journey, including error, empty, and edge states.
- Wireframes that define layout, hierarchy, navigation, and where content lives — before visual design and final content exist.
- Acceptance criteria written as observable behavior, so “done” is testable rather than debatable.

### System and data

- An architecture and data-flow diagram showing how systems talk to one another, where authentication happens, and where data is stored.
- Integrations, APIs, and the contracts between them.
- What will be measured: analytics events tied to real business outcomes, and how acquisition will be attributed.

### Non-functional requirements

These are where projects quietly fail, so write them down explicitly:

- **Security:** authentication, authorization, secrets handling, and exposure of admin or internal paths.
- **Privacy:** what personal data is collected, why, how consent works, and how long it is kept.
- **Accessibility:** target WCAG 2.2 AA; keyboard navigation, contrast, and screen-reader support are requirements, not polish.
- **Performance:** page-speed and Core Web Vitals budgets on real mobile devices and networks.
- **Discoverability:** canonical URLs, sitemaps, structured data, and content that search engines and AI assistants can read and cite.
- **Operability:** logging, monitoring, backups, and recovery expectations.

After review, the specification becomes the basis of a written proposal: scope, responsibilities, milestones, and what is explicitly out of scope.

**INPUTS**

- Preliminary requirements
- Decision log

**OUTPUTS**

- Flows, wireframes, architecture
- Acceptance criteria and non-functional requirements
- Scope, milestones, and proposal

## 4 Prototype and validate

Prototypes exist to learn quickly and cheaply. Depending on the risk being tested, that may be a clickable design prototype, a working slice of the real system, or a technical spike that proves an integration or performance assumption.

- Put prototypes in front of real users and stakeholders, and record what changed as a result.
- Use the prototype to confirm the riskiest assumption from discovery before committing the full build.
- Agree when design is stable enough to build against. Changes after that point are still possible, but they are tracked as changes with a visible cost.
- Begin test planning and data modeling in parallel, including realistic sample data.

**INPUTS**

- Specification
- Riskiest assumptions

**OUTPUTS**

- Validated design and prototype
- Revised scope where evidence required it
- Initial test plan and data model

## 5 Content and information architecture

Content is part of the design, not something poured in at the end. Real copy changes layouts, reveals missing states, and exposes unclear positioning early.

- Write for the reader’s actual questions. Clear, specific, factual pages are what convert visitors and what search engines and AI assistants choose to cite.
- Make claims you can support. Invented metrics, reviews, or results damage trust faster than thin content.
- Give each important topic its own page with a clear title, description, and heading structure, rather than burying everything on one page.
- Plan alt text, captions, and transcripts alongside images and video.

**INPUTS**

- Wireframes and design templates
- Positioning and audience

**OUTPUTS**

- Edited, accessible content
- Page metadata and structured data

## 6 Build

Engineering turns the specification into working software without compromising the agreed experience. Developers need to understand the design and navigation, not just the tickets, and should work directly with design when something is unclear.

### Engineering foundations

- **Version control for everything** — code, configuration, infrastructure, and documentation — with small changes reviewed before they merge.
- **Automated pipelines** that build, test, and package every change the same way, so “it worked on my machine” stops being a release strategy.
- **Separate environments** for development, staging, and production, with production secrets kept out of source control.
- **Documentation that lives with the code**: setup, architecture decisions, deployment, and recovery steps.

### Where AI fits

AI coding agents have changed the economics of building software. Used well, they multiply the research, implementation, testing, and documentation that can happen around a problem. They do not remove the need for judgment.

- Keep humans responsible for architecture, product decisions, security, and what ships to production.
- Review AI-generated changes with the same rigor as any other change, and verify behavior with tests rather than trusting explanations.
- Give agents written project context — conventions, environments, decisions already made — so work continues from verified state instead of being rediscovered.

**INPUTS**

- Specification and validated design
- Content and data model

**OUTPUTS**

- Reviewed, versioned application
- Automated build and deploy pipeline
- Technical documentation

## 7 Quality and verification

Web products are multi-user systems running on unpredictable devices and networks, so they need evidence, not optimism. Automate everything that can be proven by a machine, and reserve human review for judgment: visual design, tone, and usability.

- **Automated tests** at the unit, integration, and end-to-end levels, with real-browser tests for critical journeys.
- **Cross-browser and device testing**, especially on mid-range mobile phones.
- **Accessibility checks**, combining automated scanning with keyboard and screen-reader testing.
- **Performance and load testing** against the budgets set in the specification.
- **Security testing**: dependency scanning, input validation, and checks that sensitive paths are not exposed.

- **Negative testing**: confirm invalid input is rejected cleanly and that test activity never pollutes production data or analytics.

Every failure should be classified before anything is changed: is it a product defect, a test defect, or an environment problem? Fixing the wrong layer creates new bugs.

**INPUTS**

- Application and acceptance criteria
- Non-functional requirements

**OUTPUTS**

- Test results and evidence
- Known issues with severity
- Release-candidate approval

## 8 Launch

A launch is a controlled change to production, with a way back.

- Have a written release checklist and a tested rollback point before deploying.
- Prefer releases that are small and reversible; use staged rollouts or feature flags for higher-risk changes.
- Confirm canonical domains, redirects from old URLs, sitemaps, and search-engine registration so existing visibility is preserved.
- Confirm analytics and consent behave correctly in production, not just in testing.
- Run a post-release smoke test from outside your own network.

**INPUTS**

- Approved release candidate
- Release checklist and rollback plan

**OUTPUTS**

- Production release
- Post-release verification evidence

## 9 Operate and measure

Launch is the beginning of the product's life, not the end of the project.

- **Monitor what users experience.** A server responding is not the same as the right site being served. Check availability from outside your infrastructure, including DNS, certificates, and that the expected content is actually returned.
- **Watch errors and performance** with logs and alerts tuned to be actionable, so quiet periods stay quiet.
- **Measure outcomes** against the success measures from discovery, with attribution that shows which channels lead to real conversations or customers.
- **Keep dependencies and security patches current**, and keep backups you have actually restored.
- **Record incidents** with their cause and the change that prevents a repeat.

## 10 Continuous improvement

The final phase keeps every other phase improving. Review what was learned after each cycle, retire what no longer earns its place, and feed real usage back into discovery.

It also means looking outward on a regular schedule: platform and browser changes, security advisories, how search engines and AI assistants are finding and citing content, and what competitors and substitutes are doing. The goal is to keep exceeding user expectations rather than discovering too late that the ground moved.

> **The pattern behind all of it:** identify a practical gap, learn every part of the problem, and build the system needed to solve it — then prove it works, and keep it working.

---

Barton Anderson is a founder, product builder, and fractional CTO who turns practical gaps into working systems through product strategy, software, and AI. To discuss a product or platform, visit bartonlanderson.com.

© 2026 Barton Anderson. You are welcome to share this guide unaltered with attribution.