Delivery standard for coded design deliverables

Branching, gates, pull requests and deploy verification

Version 1.0 · 24 August 2026

Why this exists

A coded design deliverable auto-deploys to a client review site. That one fact drives everything below: **a merge is a publication**. The process therefore has two jobs beyond keeping the code tidy - make an accidental publication impossible, and make a bad deploy detectable rather than merely unlikely.

Nothing here is theoretical. Every warning box is something that went wrong once and cost real time to find.

1. Who does what, and when

Four roles touch a design deliverable, and they arrive at different moments. Only the designer is involved every day; the commonest failure is engaging the other three too late.

Role	Comes in at	What they do
DevOps	**Once, before wave 0** Then only on change	Sets up the CI/CD pipeline - see section 3. After that it should be invisible for months.
Designer	**Every wave, throughout**	Drives the build, reviews the result against the working tree at steps 4-6, and raises both pull requests. **Several designers may run waves in parallel** - one branch per wave is what keeps them from colliding.
PM, senior or design lead	**After a wave has shipped**	Reviews the published result. **Their feedback comes back as a new branch and a new wave**, never as edits to a branch that has already merged.
Code reviewer (dev lead)	**After each wave is pushed** On their own schedule	Reviews and merges the dev PR - see section 7. Never blocks the design branch.

A designer reviewing their own wave is not self-assessment. The design is produced by a tool the designer is directing, so steps 4-6 are a review of that output - checked in a browser, against the specification - rather than someone marking their own homework. It is the same relationship a reviewer has with any other author's work, compressed into one person and one sitting.

Feedback that arrives after a wave has shipped opens a new branch. It cannot be folded into the merged one without rewriting published history, and it should not be: the correction wants its own pull request, its own review and its own deploy verification, exactly like the wave that caused it. Name those branches fix/ rather than feature/ so the history says which is which.

The dividing line is the commit, not the seniority of whoever spoke. Feedback before it lands folds into the working tree at step 6. Feedback after it lands is the next wave.

DevOps is not a per-wave role, which is the point of setting the pipeline up properly once. The exception is a front-end feature that needs a cloud permission - an upload widget, an in-page feedback tool, anything writing to a bucket. **Raise those a wave before you need them**, because the code will otherwise be finished and unmergeable while a policy is attached.

2. Branch model

Branch	What it is	Deploy workflow
The design branch	The review build. Merging here publishes to the client.	Yes - only here
The dev branch	The engineering team's. Receives the same PRs, on their schedule.	Never
main	Where the two working branches were cut from.	No

- **The design branch name is case-sensitive and load-bearing.** GitHub branch matching is case-sensitive, so a workflow watching *Design* will not fire on *design*. Two repositories in one programme may legitimately differ - write the exact name down.
- **Only the design branch carries a deploy workflow.** If the dev branch ever inherits one pointed at the same bucket prefix, the two race and overwrite each other's builds. A dev environment needs its own prefix and base path - a deliberate change, never an inherited file.

3. The CI/CD pipeline - what DevOps sets up, and why

This is the piece the team refers to as 'setting up the CI/CD pipeline'. Concretely, on a project of this shape, it is two halves - and only one of them normally gets built:

	What it means here	Where it runs
CI continuous integration	The four gates - typecheck, lint, format, build - run automatically so nothing broken gets in.	Today: on the author's machine, via a pre-push git hook. **Nothing runs on a pull request.**
CD continuous deployment	Push to the design branch, and the review site rebuilds and republishes itself with no manual step.	A workflow in the repository, on the platform's runners.

Worth knowing before anyone assumes otherwise: the gates are enforced on the author's machine, not on the server. A git hook can be skipped with a flag, and a pull request opened from a browser never runs them at all. That has been sufficient with a small, disciplined team.

If the gates should be enforced for everyone, that is a second workflow and it is a DevOps task - one that runs on every pull request rather than on a push to the design branch. Decide it at setup. Adding it later means going back through open pull requests.

The deployment half, in detail:

- The workflow fires on push to the design branch and only that branch.
- It builds with the project's base path, syncs to one storage prefix, and invalidates the CDN.
- **The sync uses --delete, so every deploy replaces the whole site.** There is one prefix and one distribution per product.

- **Configuration lives in repository variables, not in the code** - region, bucket, distribution id, and any third-party keys. The base path, the bucket or the distribution can change without a commit, and nothing secret sits in the repository.

> **The one line in the workflow that is easiest to lose, and worst to lose.** Built assets are content-hashed and the sync runs with --delete, so yesterday's files are gone. If the HTML shell is cached by the CDN, a returning reviewer gets an old shell pointing at asset names that no longer exist - a blank page, on a site that deployed successfully.
>
> So the HTML is synced **separately, with no-cache headers**, after the hashed assets. Hashed assets can be cached forever; the shell can never be cached at all. Losing this produces a fault that looks like a broken build and is not one.

What DevOps needs before the first wave:

- A storage prefix and a CDN distribution **per product**, not per branch.
- An IAM user scoped to that prefix and that distribution, with its secret in repository secrets. Decide deliberately which values are variables and which are secrets.
- The deploy workflow, watching exactly one branch, with the base path matching the prefix.
- The access gate on the review URL, if the client's work is not to be publicly readable. Note that a gated URL cannot be fetched by a script, which is why deploys are verified from the build log instead.

4. The wave loop

One feature, one branch, one commit, two PRs. Ten steps, in order:

1. Cut a branch from the design branch, after pulling. Name it feature/, fix/ or docs/.
2. **Read the specification section first.** Section numbers are load-bearing. Check whether the spec answers a question somewhere else before recording it as blocked.
3. **Read whatever prior art exists** - a previous build, a reference design, the client's current product - **report what ports and what does not, then WAIT.** The report is the design proposal.
4. **Build. Verify in the browser.** Run the four gates.
5. **Stop and review it, uncommitted.** In a browser, against the specification. Never commit mid-build - a half-built wave in the history is a wave nobody can review.
6. **Feedback rounds against the working tree**, as many as it takes.
7. **Only once it is approved, update the documentation.** Revise stale entries; do not append beside them.
8. **One commit** (two if the reasoning genuinely splits), push, raise both PRs.
9. **Merge the design PR deliberately - it publishes.** Leave the dev PR alone.
10. **Verify the deploy by matching asset hashes**, then move on.

> **Four things about this loop that are easy to under-weight.**
>
> Step 3's WAIT applies even when there is no prior art at all - the report becomes the proposal either way.
>
> Step 6 rewrites waves; it is not a polish pass. Expect a feature to gain a whole dimension after a reviewer looks at it.
>
> Step 4 means the browser, not reading the diff. Defects that pass typecheck, lint and build while being plainly wrong on screen are the normal case, not the exception.
>
> Step 7's **revise, do not append** is what stops documentation from accumulating two contradictory answers, both of which look current.

5. The four gates

```
npm run typecheck     # tsc --noEmit -p tsconfig.app.json
npm run lint          # 0 errors; a known warning count is fine
npx prettier --write .
npm run build
```

- **Never a bare tsc --noEmit** where the root tsconfig is a solution file holding only references - the bare form exits 0 on any error in the source tree. It has reported clean while four files held real type errors.
- **The pre-push hook runs a subset and fails the push** if they fail. Run them yourself first; do not reach for --no-verify.
- **Read the chunk sizes whenever one module gains a dependency on another.** A bundle regression is invisible in every other gate - typecheck, lint and the browser all stay green while first load doubles. Measure the baseline by stashing, not by trusting a number written down.

6. The two pull requests

Two PRs per wave: same branch, same body, different bases.

```
git push -u origin feature/<name>
gh pr create --base <design-branch> --head feature/<name> --body-file pr.md
gh pr create --base <dev-branch>    --head feature/<name> --body-file pr.md
```

- **The design PR is a publication.** Merging it is a decision, not a formality. If a second pair of eyes should see it first, that is what branch protection is for - section 7.
- **The dev PR belongs to the engineering team.** Never merge it, never chase it.
- **Merge commits only** - gh pr merge <n> --merge. Squash and rebase lose the branch's shape.
- **Never pass --delete-branch.** Branch cleanup is a separate, deliberate act, and a branch must stay on the remote while any PR against it is open.
- **Check nothing is already deploying before you merge.** Two merges to the design branch race each other.
- **No reviewer is requested on the platform** unless the team works that way - the ask goes through whatever channel they actually read.

> **The dev PR will show a superset, and that is expected.** While the dev branch lags, a branch cut from the design branch contains every wave the dev branch is missing, so its PR shows all of them. Say so at the top of the PR body and name the merge order. It shrinks by itself as the queue lands. **Do not fix it by stacking branches** - that trades a cosmetic problem for a real one.

7. Code review - two reviews, at two different moments

The two PRs exist because **there are two different reviews to do, and they cannot be done by the same person at the same moment**.

	The design review	The code review
When	Steps 5-6, before a commit exists	After the wave is committed and pushed
Against	The working tree, in a browser	The dev PR's diff
By	Whoever faces the client	The engineering team who will take this to production
Looking for	Does it do the right thing? Is the copy true? Is the scope right?	Architecture, reuse, naming, whether the mock seam is findable at handoff
Blocks	The commit	Nothing on the design branch

- **The design review is the one that changes the product**, and it happens before there is anything to review on the platform. That is deliberate: reviewing code before the design has settled reviews something that is about to be rewritten.
- **The code review never blocks the client.** If it did, a review comment about naming would hold up a client demo. The design branch ships; the dev branch absorbs the same commits on the engineering team's own schedule.
- **Because the two reviews are separated, neither is optional.** A wave that skips the design review ships the wrong thing quickly. A wave that skips the code review leaves the engineering team a deliverable nobody on their side has read.

> **An approval on the platform is pinned to a commit, and by default it survives new commits.** Unless **dismiss stale approvals** is enabled on the repository, a pull request can show an approval for code nobody has read - the reviewer approved an earlier revision and the branch moved underneath them. **Enable it per repository at setup.** It is one checkbox and it is the difference between a review and the appearance of one.

> **Requesting a review on the platform does not produce one.** A review request is easy to raise and easy to leave unread, and it is common to find every wave discussed in detail in chat while the pull request record stays completely empty. That is not a failure - it is where the team actually works. **Decide which channel the request goes through, and treat the platform record as bookkeeping rather than as the review itself.** If that record matters for audit, that is a reason to change the habit deliberately, not to assume it is already happening.

- **Decide whether the design branch requires an approval.** Merging it publishes to the client, so there is a real argument for protecting it - and a real argument against, if one person owns both the build and the client relationship. **Either is defensible; defaulting to neither is not.** Where a second pair of eyes is wanted, code owners on the paths that matter is lighter than protecting the whole branch.

8. Commits

- One commit per wave. Two only when the reasoning genuinely splits into two changes.

- Conventional format: type(scope): lowercase subject. Scope is mandatory where commitlint is configured that way; body lines wrapped at 100 characters.
- **Write the reasoning, not the file list.** The diff already says what changed. The message is the only place that can say why, and what was rejected.
- An interactive commit hook will block automated commits. Bypass it explicitly and write a compliant message by hand rather than disabling the hook for everyone.

9. Verifying a deploy

The green tick is not the check. It says the job ran. It has run green while shipping the wrong build.

```
npm run build -- --base=/<path>/ 2>&1 \
  | grep -oE "dist/assets/[A-Za-z0-9_-]+\.(js|css)" | sort -u > local.txt

gh run view <run-id> --log \
  | grep -oE "dist/assets/[A-Za-z0-9_-]+\.(js|css)" | sort -u > deployed.txt

diff local.txt deployed.txt      # empty = deployed exactly what you built
```

- **Count the assets from your own build every time.** On one project the count went 8, 12, 14, 17, 21, then back to 19.
- **A decrease is not automatically a lost code-split** - a bundler may legitimately merge two chunks. **Explain any change in either direction before merging.** An unexplained count is the only signal you get.
- If the review site sits behind an access gate, fetching the URL proves nothing. The build log is the source of truth.

10. The rules that cost the most

Delivery-level lessons, each learned the expensive way:

- **A rule recorded in the design system is not a rule followed.** Rules have been written down, cited to justify an exception, and never actually applied to the surface that needed them. Audit against the specification, not against your own notes.
- **A blocked item stops being re-examined.** On one project, five entries sat in a 'do not start' table after the thing blocking them had been answered elsewhere. Re-read the blocked list whenever a decision lands.
- **Copy is a claim, and an unbuilt claim is a defect.** Interface text that describes behaviour the product does not have is a bug, and the fix is usually the missing behaviour rather than softer words. Sweep the copy for capability claims.
- **Seed data is copy.** It is read by the same people and carries the same authority. Never put a price, a plan name or realistic legal prose in a mock.
- **A comment is not a measurement.** Two boxes described as identical differed by four pixels. Measure the thing before you assert it, and before you adjust it.
- **Auditing a model is not auditing a user interface.** When one product consumes another's data, read the other product's screens, not just its types. Requirements live in rendering code that no schema mentions.

This document generalises what is written in each repository's own agent guide. The repository copy stays authoritative for that project, because it names the exact branch, base path and gate commands. Keep this one for starting the next project.

Two things are deliberately left blank because they are organisational rather than technical: who holds the DevOps role

on a given engagement, and whether the code review is a named person or the team. Fill those in per project; everything else here transfers unchanged.