

Switchyard

Jul 06, 2026 • spencerburleigh.com/switchyard

A coding agent can finish its assignment while leaving the project further from shipping. Its change might depend on a branch that hasn't landed, conflict with another agent's work, or have been edited since its last review. More agents give you more of these coordination problems.

CrossCheck^[a] was my first attempt to put structural constraints around AI development: independent model review, scoped permissions, and repository protections. Switchyard extends that approach across the work itself, from deciding what should run to integrating the result.

The biggest change is moving from a do-work folder of markdown tasks to using **Linear as a state machine for the project**. I'm writing a bunch of middleware around it to keep agents using that structure consistently and make progress depend on the right evidence.

This has been an internal tool I'm now piloting with a few friends' startups. The **landing page**^[b] is the best visual explainer; its animation follows work through the whole system.

From a folder of tasks to a state machine

CrossCheck's do-work queue was a useful initial way to track agent work: each .md file described an assignment and an agent could pick one up, create a branch, implement it, and submit a pull request through the repository's checks.

The awkward part was representing everything around the assignment. One task blocked another. Several findings described the same bug. A small fix belonged to a larger project whose priorities had changed. Files could hold that information, but each relationship required another convention for agents to interpret. Agents were also bad at keeping track of which tasks were claimed or done. Seeing an item's history meant digging through Git rather than opening its work record.

Linear already has a model for this. Issues have states, priorities, size estimates, and labels. They have first-class relationships to blockers, parent issues, duplicates, and related work. Projects and milestones put individual assignments in the context of a larger outcome.

These fields do different jobs. Priority expresses urgency. Size estimates effort. Labels classify the work. Dependencies constrain ordering. Projects and milestones organize what the work contributes

toward. Keeping those dimensions separate matters: a small task can be dangerous, and an urgent task can still be blocked.

Switchyard uses that structure to decide how work moves. Linear holds the shared record; the middleware defines the allowed transitions and the evidence required for them. An implementation being finished, its review being complete, and its code reaching main are distinct states with different requirements.

For example, an urgent issue can wait on a prerequisite recorded as a blocking relationship. Once that prerequisite lands and its state is reconciled, the scheduler can reconsider the issue for dispatch. It doesn't need a new prompt explaining the dependency. A replacement agent can pick up the same record after the original session ends.

GitHub holds the code, checks, and merge history. The middleware reconciles that evidence with Linear. The middleware keeps everything clean: a dead process should not leave an issue permanently "in progress," and an old approval should not certify code that has since changed.

Rails around a flexible API

Linear exposes a broad API for changing issues, metadata, and relationships. It leaves me free to define my own development process. But an update the API accepts can still leave the work in a state that makes no sense for that process.

An agent can describe a dependency without recording a blocking relationship. It can create a plausible issue without saying how anyone will know the work is finished. Changing a status to "ready" doesn't establish that the prerequisites for starting are actually satisfied.

The middleware puts rails around that flexibility. I'm building supported ways for agents to submit findings, attach evidence, and request transitions, with checks that guide those requests into a consistent form. The aim is to make the conforming path the easiest path to follow.

Useful feedback matters as much as rejection. If a request is incomplete, the response should identify what is missing. If a transition needs evidence, the agent should learn what evidence to supply. That gives it a way to correct the request and continue, without guessing at the workflow's conventions.

LLMs and deterministic tools have complementary jobs here. A model can interpret a vague request, recognize two differently worded reports of the same problem, or propose a smaller assignment. Deterministic checks can validate required fields, relationship rules, and whether the evidence for a transition is present. Model judgment can help form a proposal; explicit checks govern what happens next.

Coverage is still evolving. Some paths enforce these checks today; metadata audits can also propose corrections for review. As recurring mistakes become clearer, I can turn more conventions into checks. Better models can improve interpretation without changing where the project's state lives.

That is the useful progression from CrossCheck: the same idea of structural constraints now applies to organizing the work, as well as checking the code. Permissions support that process. An agent can contribute evidence without acquiring the authority to approve its own contribution or redefine the rules.

Five stages of work

Switchyard follows five stages: intake, triage, dispatch, integrate, and follow up. Each handoff has to leave the next stage with enough information to do its job.

1. Intake establishes the assignment.

A request might come from a person, a bug report, monitoring, or another agent. It needs a clear objective, bounded scope, and a way to recognize success before it becomes executable work. That assignment remains the reference for implementation and review. Discovering adjacent work is useful; quietly absorbing it into the current task makes both scheduling and review harder.

2. Triage makes the backlog usable.

Triage clarifies requests, consolidates duplicates, records dependencies, and chooses an execution lane. Several agents can discover the same underlying bug. Preserving their evidence in one place is more useful than dispatching competing fixes.

The dependency graph also preserves useful parallelism. Some changes need a prerequisite to land; others can proceed independently. The scheduler needs to distinguish the two.

3. Dispatch plans and supervises a wave.

Switchyard plans around declared scope, dependencies, task difficulty, and available capacity. Agents work in isolated worktrees, so they can edit separate branches without trampling each other's checkouts. They still share a destination codebase, so isolation alone cannot prevent incompatible changes.

Within each session, an implementation agent writes code and, where required, submits it to a reviewer from another lab. The agents work through review findings before the PR moves to integration.

Supervision continues this entire time. The system tracks progress and surfaces stalled or failed sessions for recovery or escalation. An abandoned session needs a next action, rather than an issue that stays reserved forever.

4. Integration checks the combined result.

Every PR gets its own fast CI run before being grouped into a bigger integration branch for the full test run. If checks fail, the likely failing PR is kicked out and the failing test is re-run.

5. Follow up preserves what the process learns.

Implementation, review, and validation uncover more work, which agents are prompted to submit as new Linear issues. Those issues return to triage for deduplication, prioritization, and checks that they conform to the issue schema.

Difficulty and risk are separate decisions

A difficult refactor might have a narrow, reversible effect. A one-line edit to a sensitive control might be easy to write and dangerous to apply. Effort and consequence need different treatment.

The lite, standard, and deep execution lanes match implementation effort to the assignment. Human work has its own route. Risk determines the review and approval requirements, with repository policy and explicit issue requirements supplying the applicable gates.

Switchyard has five risk tiers:

Tier	Review route
r0	Required policy and CI checks without model review
r1	Lightweight model review plus required checks
r2	Strong independent review with the expected checklist evidence
r3	Release review with strong model reviewers and supporting evidence
r4	Recorded human approval and human control of application or merge

Triage sets the initial risk level, which is reassessed against the actual changes when a pull request is submitted.

Adjusting throughput to budget, conflicts, and CI

As implementation gets faster, more of the project's time goes into deciding what should run, reviewing changes, and getting them onto main. An agent can finish quickly while increasing the unfinished work around it. The useful measure is how much reviewed, integrated work the system can complete.

Switchyard manages three constraints around that flow.

Budget accounts for remaining capacity on my Claude, Codex, and Grok plans. It leaves headroom for both implementation and review so a wave can finish the work it starts.

Flow slows or pauses dispatch when review and integration fall behind. Waiting branches age as main changes, so starting more work can increase the cost of finishing the work already open.

Cleanup reclaims completed workspaces under the configured policy while protecting active work and uncommitted changes. Failed sessions need enough reconciliation to determine what can be recovered.

These constraints interact. Stalled integration leaves more branches waiting, more workspaces occupied, and more changes to revisit. Keeping every agent busy can make the overall system slower.

People retain distinct decision points. Human-only assignments stay out of automatic dispatch. Agents can prepare other work that still needs a recorded human decision before it lands. The highest risk tier also keeps application or merge under human control.

Evolving Switchyard

CrossCheck made me think about the conditions under which I would accept an agent's code. Switchyard extends that question to the project: what is ready to start, what can run together, and what has actually earned its way onto main? Linear provides the shared structure. The middleware is where I'm making those answers consistent enough for agents to act on.

I'm piloting Switchyard with a few friends' startups to see whether this workflow stays understandable and recoverable across different projects. If you're interested in trying Switchyard please [reach out](#) ^[c].

Links

- a. spencerburleigh.com/crosscheck
- b. switchyard.tools
- c. <mailto:spencer.burleigh@gmail.com>