

GloriousFlywheel

The build is done once: an $O(1)$ software development lifecycle

Jess Sullivan, xoxd.ai, LLC (dba Tinyland Services)

2026-09-19, second edition

Abstract

Your developer's machine, your CI runner and your release pipeline each execute the same work in an environment of their own, so every change set is executed once per caller. GloriousFlywheel attaches all three to one pool. A result computed once is withdrawn by every later authorized caller instead of being computed again; the action, not the runner, is the unit of compute; and remote-required work fails closed rather than falling back to a laptop. This paper states the thesis in the operator's own words [36], explains how the fabric keeps it (fourteen laws and eleven versioned interfaces, quoted in the appendix), states what you get and what you must do to attach, and places the design beside five products that each optimise one part of the same system.

1 The problem: you pay for the same work three times

Take one change set through an ordinary enterprise delivery system. A developer builds it on a laptop. A CI runner builds it again when the branch is pushed. The release pipeline builds it a third time on the way to a stateful deployment, and the next branch builds it a fourth. Each caller executes the work itself, so the work performed grows with the number of callers: $O(N)$ in callers, and each new caller you add, each coding agent opening branches all night, adds a full execution of work whose answer already exists somewhere in your estate.

Three properties of that shape make the duplication structural.

The reuse boundary is smaller than the estate. A dependency cache keyed by branch and repository cannot serve a sibling branch, another repository, or a developer's machine, so warm work is warm only where it was produced.

The unit of compute is the job. When the thing scheduled is a whole CI job on a whole runner, the parallelism available to a change set is the size of the machine that job rented, and the way past that bound is to rent more machines.

The developer environment is outside the system. A pipeline that starts at a repository event cannot withdraw what a developer already computed, and a developer cannot deposit into what the pipeline will later read.

Runner classes, container images and a pipeline per substrate are how a team pays for one capability three times. Figure 1 draws the shape and the shape this paper replaces it with.

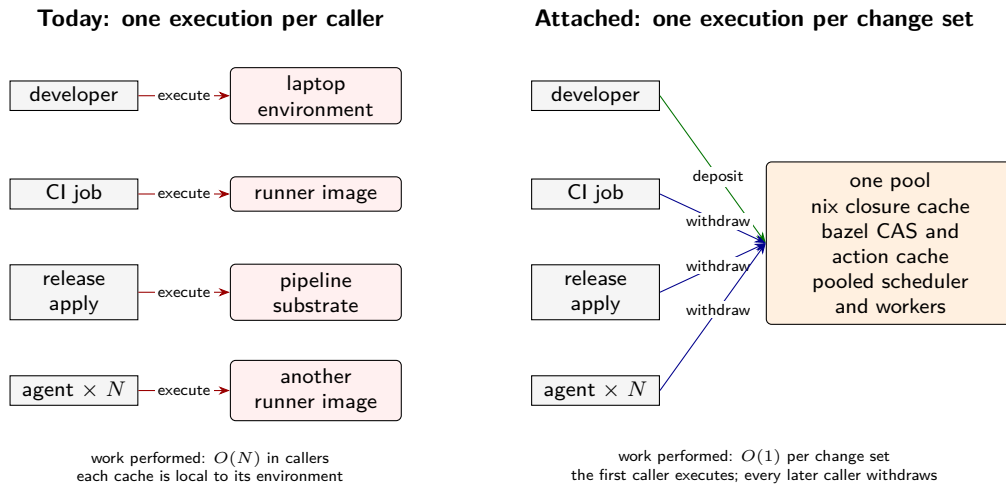


Figure 1: The same change set under the ordinary shape and under one attached path. On the left each caller executes in its own environment and its cache reaches no further than that environment. On the right every caller attaches to one pool: the first execution deposits, every later caller withdraws, and a miss is executed once by a pooled worker.

2 The thesis

Every action, derivation, test and artifact: computed once, withdrawn everywhere [36].

Enterprise software spends its day recomputing answers it already has. GloriousFlywheel makes every build and derivation hermetic at the blob level, so a result computed once is never computed again, anywhere in the lifecycle.

Every engineering organization pays for the same work three times: the developer’s machine builds it, the CI runner builds it again, the stateful environment builds it a third time. GloriousFlywheel replaces those three walled gardens with one attached path. Development, test, build and release run on one pooled cache and scheduler; trusted work deposits its result automatically, and every attached caller withdraws it instead of computing it again. The action, not the runner, is the unit of compute: a granular, hermetic element produced the same way in a dev environment, a remote execution or a test run. Hermeticity at that unit is what lets the whole lifecycle, not one vertical, inherit compute once. Agent fleets made it urgent. We build that fabric [36].

That is a thesis about the software development lifecycle, not a measurement. The rest of this paper says how the fabric keeps it, what it reports, and what it refuses.

3 How it works

The fabric is one Nix and Bazel action fabric that repositories attach to [23]. It is defined by fourteen laws and eleven versioned interface authorities, typechecked as a Dhall model that refuses a core violating any of them [26]. The laws are quoted verbatim in Appendix A; the ids in the margin of this section point at them. What follows is what they mean for you.

You attach once. Your developer’s machine, your CI runners and the pooled workers resolve the same logical tenant, the same three caches (the Nix closure cache, the Bazel content store, the Bazel action cache) and the same toolchain contract. In the typed model the developer lane and the CI-runner lane carry identical withdraw and deposit sets, and a core in which a developer cannot reach the pooled executors is refused outright. GF-A01

Trusted work deposits; everyone withdraws. Successful trusted work publishes its reusable closure or action result before its lane reports success; a manual seed is bootstrap or repair, never steady state. A warm result is reusable by every other authorized caller with the same tenant, digest, platform and policy identity. The cache key carries no branch and no caller identity, which is why the reuse boundary is your estate and not your branch. One developer’s run through the remote-execution surface pre-warms CI’s run of the same suite; that compounding, not any one cache or scheduler, is the flywheel. GF-A04
GF-A05

The action is the unit of compute. The REAPI action, not the CI runner, is what the fabric schedules. Placement is selected by capability (architecture, runtime, privilege, accelerator, bounded resources), never by repository identity, and the scheduler is the fabric’s own: executor-pool admission by capability, per-tenant concurrency quota, worker-pool lease and slot dispatch. Throughput is a property of that pooled scheduler (hit rate, deposited and withdrawn results, admitted actions, worker slots) and never of runner count. A CI runner is a thin admission shell that dispatches actions exactly as a developer’s machine does; a dozen thin runners carrying a hundred-plus concurrently executing actions is the design bar. GF-A12
GF-A02
GF-I11

Misses execute remotely or fail explicitly; nothing falls back to a laptop. A remote-required cache miss is admitted to a compatible pooled worker or terminates with a typed admission or availability failure. Once admitted, work never executes locally: successful execution is validated and deposited; cancellation, deadline, execution failure, worker loss or absent capacity terminates with a typed failure. Backend unavailability, corruption, denial and a genuine miss are four distinct observable states, and a core that cannot tell outage from miss is not admissible at all. The control-plane cell does not execute actions; the typed model asserts that a core with an executing control plane is refused. GF-A06
GF-A07

Tenancy and identity are server-side. Writes, leases and quota charges bind an authenticated tenant and subject; a developer invocation acquires no action-cache write authority of its own. A served object matches its digest, and every output-affecting input, command, environment, platform and toolchain fact participates in the Nix or REAPI identity. Physical deduplication grants no logical visibility. An artifact reference, its verification and its decoded semantics enter a controller only as one opaque accepted value that a caller cannot construct. GF-A08
GF-A03
GF-A14

Evidence is external; supply is measured. Product status comes from protocol responses, build-event data, service metrics, failure injection and restore tests, never from a producer’s summary of itself. Live capacity is read from the running system with a timestamp, never inferred from a declared number: a declared cap is an admission bound, not a measurement. GF-A09
GF-A13

Implementations are replaceable; core holds no installation facts. REAPI Protobuf and the laws are authoritative; the Go and Rust services, storage providers, schedulers and control-plane applications behind the eleven interfaces (Appendix B) are replaceable components. Core defines capability classes, interfaces and laws; your installation binds your organization, your App and your repositories; your overlay binds admitted subjects, trusted workflows, write policy and abstract capability demand; the provider binds routes, endpoints, workers and capacity. A hostname or a provider class is never a core placement primitive, so your demand names no cluster, no worker and no runner label. The projected core, drawn from the typed model, is Appendix C. GF-A10
GF-A11

4 What you get

GloriousFlywheel guarantees and reports three operative product objectives [29]. Exact-head request accepted to routable PRE ready: p95 at most 600 seconds. Protected-main merge to exact merged digest served: p95 at most 600 seconds. Admit 110 simultaneous valid action submissions without using GitHub Actions jobs or ARC pods as the execution unit. Remote ActionCache hits and active cold misses are reported separately, and attribution covers tenant, action, queue, cache, worker lease, latency, refusal, and error class.

The dashboard is metrics and a protected API, and it reports: trusted-deposit coverage and deposit lag; Nix substitutions and bytes avoided; Bazel content and action-cache hit rates; local executions avoided and remote-action count; queue p50 and p95 and available worker slots by capability; worker failures, sandbox failures and no-fallback violations; tenant quota usage and admission rejection; corruption, digest mismatch and quarantined objects; durable-store restore objectives and the last successful restore drill; credential freshness and rollback completion time [23].

Those two lists are the measurement plane. This paper reports no timing, percentage or hit rate of its own: under the evidence law a claim is measured by the fabric, not by the paper.

What the fabric does when something breaks is as much of the product as the objectives, because it is what keeps a degraded pool from looking warm. Table 1 is the failure vocabulary [23].

5 How you attach

An organization attaches in five steps [28]:

1. Create and install your organization’s own all-repositories GF GitHub App.

Table 1: Failure semantics.

Failure	Required behavior
Cache miss	Admit execution to compatible remote supply or return an explicit availability or admission failure; no local or cache-only fallback.
Cache unavailable	Emit an unavailable state and fail closed; never translate it to a miss.
Digest mismatch	Refuse the object, quarantine it, and emit a poison signal.
Unauthorized write	Reject without making content-store or action-cache state visible.
Partial upload	Partial content MAY remain for later collection; action cache MUST NOT become visible until every referenced output is durable.
Worker loss	Revoke and fence the lease; do not claim completion or local fallback.
Storage loss	Stop acknowledging durable writes; recover from independently tested backup.
Rollout failure	Return to the prior release while retaining acknowledged data.

2. Publish signed `OwnerInstallation/v1`, `TenantOverlay/v1` and `RevocationSet/v1` operands from your organization's `-infra` repository.
3. Check an exact `ActionPlan` into each application repository and call the immutable ci-templates v4 workflow.
4. Let the controller join your demand to provider supply and publish the immutable catalog that `gf-action-client` reads.
5. Prove one remote `Execute` or one authenticated same-tenant cache hit with measurement-plane attribution.

Enrollment is org-shaped, one-way and immediate. You enroll in the fabric; the fabric does not enumerate, register or approve you, and no consumer list lives in core. Any repository your App installation selects, bound by the five independently verified inputs plus exact `ActionPlan` authority, attaches to your organization's pooled REAPI backend out of the box, from an underpowered developer machine exactly as from CI [23]. Enrollment and invocation are separate transactions: the durable owner-supply catalog binds your policy to abstract capability supply and contains no repository, action-plan or source instance, while each call supplies one exact reviewed `ActionPlan` and receives one short-lived `ResolvedActionBinding/v1`. Your demand is provider-blind: it never names a backend cluster, endpoint, worker, node, taint, storage class or runner label [28].

6 Where it stands

Five products each optimise one element of the system this paper changes. Each does what it does well. The one-paragraph comparisons below are drawn from the corrected record in Appendix D, which quotes each product's own documentation; no number here was measured by the comparison [24].

Blacksmith optimises the CI runner: a drop-in host that boots in seconds and, for developers, a second CI-identical microVM per agent. Its reuse boundary is the

repository, and the developer’s own machine never attaches to the cache. Here the developer, the runner and the pooled workers attach to one tenant, cache and toolchain contract, and a miss on a remote-required lane becomes every authorized caller’s hit (GF-A01, GF-A05, GF-A06). Blacksmith’s one-line adoption is the better onboarding story.

Buck2 optimises the build tool and is remote-execution first by design. Whether an action runs locally, remotely or races both, and whether a remote failure falls back to local, are fields a project’s rule authors choose, and it ships no scheduler of its own. Here the local shape of a remote-dispatching lane is refused by the typecheck and throughput is a product surface owned by the fabric’s scheduler (GF-A06, GF-A07, GF-A12). Buck2’s documented developer-to-CI sharing through sandboxed remote execution is genuine and to its credit.

BuildBuddy optimises the Bazel remote-execution and cache backend, with a pooled executor and warm whole-machine snapshots for its runners. Its cache is documented as a Bazel cache whose separation between CI and developers is offered as opt-in keys, uploading local results is a client flag, and local execution remains a reachable strategy in the same invocation. Here deposit is a lane obligation performed by the executing cell, writer authority is server-side, and the runner is typed as a thin admission shell with nothing to snapshot (GF-A04, GF-A08, GF-A12). Whether this design is immune to the blob-eviction case BuildBuddy documents is not established and is not asserted.

Buildbarn optimises the remote-execution backend behind a build tool and is deployed at scale. Its contract ends at the worker: attachment is a client setting each build tool carries and can omit, and how a caller treats an unavailable backend is the client’s choice. Here local execution of remote-required work is inadmissible at the lane, at admission and as evidence, and the instance name a caller presents resolves to a tenant bound server-side rather than being trusted as the tenant (GF-A07, GF-I02, GF-I04). Buildbarn’s read-time completeness check against eviction is a real advantage and is recorded as one.

GitHub Actions optimises the CI runner as a disposable, autoscaled job host: one job per runner, scaling arithmetic on jobs, caches shared by branch. Here throughput is bounded by admitted actions and attested worker slots, never by runner count; the cache key carries no branch; and the developer is an attached caller of the same pool rather than outside the system (GF-A05, GF-A12, GF-A13). Hosted machines with zero maintenance remain its strength.

7 Who runs it, and what is next

The producer runs it on its own estate, and the flagship external consumer is `tinyland-inc/rockies`, the Rocky-10 XR workstation OS program, named as the flagship vertical slice for the real remote-build product [27]: an immutable Rocky 10 builder and toolchain contract, a bounded composition graph, and a distributed worker fleet, each its own authority. Two organizations beyond the producer are adopting organizations in the sense section 5 defines: Great Falls Tool Bus and MMS, whose installations belong to their own `-infra` repositories and whose demand stays provider-blind [23]. Adding an organization requires no source edit in core: the adopter’s

verifier constructs its own demand and the tenant-blind provider aggregator joins it to independently verified supply.

The roadmap has five outcomes, each with its own completion boundary: owner and provider attachment (an installed catalog that admits the exact owner, repository, action-plan and source tuple, with no producer-held consumer rows); remote execution and pooled reuse (an exact remote miss consumes a worker leaf and a fresh caller then receives the same action’s qualified cache hit without one); application publication (qualified action output enters the release publisher, preview binds the exact head, production binds protected main); stateful preview with main equal to production (typed reconciliation in the owner controller, a routable exact-head preview, expiry and rollback, no manual post-merge apply); and throughput and latency (the objectives of section 4 measured against the real action fabric, with hits, queued misses and executing misses accounted separately). Source merged, artifact published, controller installed, action admitted and content independently served are distinct outcomes, and the fabric reports each one on its own evidence.

The record this paper cites is the product’s own typed model and architecture documents, pinned at the commit on the receipt. Attaching costs you five steps and one repository. If you run a fleet that already outgrew its pipeline, we would like twenty minutes.

A The fourteen axioms, verbatim

The product laws as `docs/architecture/recursive-dogfooding.md` states them, one paragraph each, with the typed model’s contribution where `Core.dhall` adds a shape law [23, 26].

Attach once. One logical tenant, cache, and toolchain contract is resolved by local development, CI, ARC runners, and REAPI workers alike. In the typed model the Developer lane and the ArcRunner lane carry identical withdraw and deposit sets over all three cache kinds, and a core in which the developer cannot reach the Linux executor pool is refused [26]. GF-A01

Capabilities, not repositories. Placement is selected by architecture, runtime, privilege, accelerator, and bounded resource capability; project identity is not a runner class [23]. GF-A02

Content identity is verified. A served object MUST match its digest, and every output-affecting input, command, environment, platform, and toolchain fact MUST participate in the corresponding Nix or REAPI identity [23]. GF-A03

Trusted work deposits automatically. Publication of the reusable closure or action result precedes the lane reporting success. A manual seed is bootstrap or repair, never steady-state acceptance. The typed model contributes the shape law that a lane declaring deposits must set `depositOnSuccess = True`, and that a Bazel-result lane must deposit both the CAS and the action cache [26]. GF-A04

Every attached path withdraws from shared authority. A warm result is reusable by another authorized caller with the same tenant, digest, platform, and policy identity. The cache key carries no branch and no caller identity [23]. GF-A05

Remote misses execute remotely or fail explicitly. A remote-required cache miss MUST either be admitted to a compatible worker pool or receive a typed admission or availability failure. The control-plane cell is not a compatible worker and MUST NOT execute actions. Once admitted, work MUST never execute locally: successful execution is validated and deposited; cancellation, deadline, execution failure, worker loss, or absent capacity terminates with a typed failure. Rollback selects a prior worker release or route snapshot, or leaves execution unavailable; it never restores control-plane execution [23].	GF-A06
No silent fallback or disguised outage. Remote-required work fails closed when remote execution is unavailable, and backend unavailability, corruption, denial, and a genuine cache miss remain four distinct observable states. The core must declare outage distinct from miss to be admissible at all [26].	GF-A07
Tenancy is server-side. Writes, leases, and quota charges bind an authenticated tenant and subject. Every audit event records either that identity, an explicitly anonymous public reader, or a denied unauthenticated principal. Public-read Nix closures require explicit policy plus signature and content verification. Physical deduplication MUST NOT grant logical visibility [23].	GF-A08
Evidence is external to the claim. Product status comes from protocol responses, build-event data, service metrics, failure injection, and restore tests. A producer cannot be the sole verifier of its own summary [23].	GF-A09
Implementations are replaceable. REAPI Protobuf and these behavioral laws are authoritative. Go, Rust, storage providers, schedulers, and control-plane applications are replaceable components [23].	GF-A10
Core contains no installation facts. Core defines capability classes, interfaces, and laws. A consumer installation binds one organization, its App installation, and all-repository selection; overlays bind organization-wide admitted subjects, trusted workflows, write policy, and abstract capability and concurrency demand. Repository and action-plan facts enter only with a fresh invocation. Provider supply binds the instance authority, routes, endpoints, workers, and capacity. Cluster substrate binds physical nodes, taints, labels, and provider storage classes. A hostname or provider class is not a core placement primitive [23].	GF-A11
The action is the unit of compute. Throughput lives in the pooled scheduler, not runner count, and a runner-count delta is never a product win. Capacity and capability remain evidence-gated, with ARC job capacity and REAPI worker capacity scaling independently within declared bounds [23].	GF-A12
Supply is measured, not remembered. Live provisioned capacity is read from the running system, never inferred from a declared or last-known number. A declared cap is an admission bound, not a measurement; any capacity claim cites a measured reading with its timestamp, never a remembered value [23].	GF-A13
Verified operands are constructor-protected. An artifact reference, a verification observation, canonical payload bytes, and decoded semantics MUST NOT enter a controller as independent caller inputs. An independent verifier fetches the exact subject, verifies its external signature or bundle and producer policy, bounds and canonically decodes those same payload bytes, and returns one opaque accepted value whose	GF-A14

constructor is unavailable to callers. A caller-constructed structurally similar record remains untrusted [23].

B The eleven interface authorities

Each product plane is replaceable only behind the interface authority that binds it. Upstream wire specifications remain authoritative; the product maps its policy onto them and does not fork their semantics [23].

ID	Plane	What it binds
GF-I01/v4	Attachment	A consumer-owned OwnerInstallation/v1 , TenantOverlay/v1 , consumer RevocationSet/v1 , and a bounded all-repositories App observation, resolved to canonical VerifiedOwnerDemandContribution/v1 bytes under a projected pod-bound service-account identity.
GF-I02/v4	Identity and admission	An explicit closed identity distinguishing genuine Actions OIDC from a Developer authenticated through the adopter's Product App; exact repository, source, action-plan, lock, and Bazel identities joined into the atomic ResolvedOwnerSupplyCatalog/v1 ; and the short-lived ResolvedActionBinding/v1 that supplies tenant, route, platform, policy, and scopes.
GF-I03/v1	Nix shared data	The Nix binary-cache NAR and .narinfo protocol, store-path identity, signatures, and the complete recursive closure. Attic is an adapter, not the protocol authority.
GF-I04/v3	Bazel shared data	REAPI v2 content-addressable storage, ActionCache , and ByteStream , tenant-routed by instance_name , plus one provider-selected invocation-scoped ExternalInputClosureBinding/v1 that addresses a complete target-scoped vendor tree and admits no catalog, roster, distdir, network, or fallback authority.
GF-I05/v2	Remote execution	REAPI v2 Execution and Operations plus declared platform properties, typed admission and failure status, worker identity, lease fencing, and no-local-fallback semantics; WorkerRouteSnapshot/v1 binds a monotonic generation to the exact ready and draining workers, and WorkerRouteActivation/v1 binds that snapshot to one selected ready worker and its workload identities.
GF-I06/v1	Runner capacity	The GitHub Actions runner protocol through an ARC AutoscalingRunnerSet , selected by a capability tuple rather than repository identity. ARC job capacity and REAPI worker capacity remain distinct.
GF-I07/v5	Qualified action output	The exact canonical action-plan digest and its ResolvedActionBinding/v1 , joined to a bounded, CAS-present, non-authoritative ActionOutputSet/v1 and an independent ActionCacheObservation/v1 . Only a remote cache hit or a remote-execution deposit qualifies; local execution is representable only so the law can refuse it.
GF-I08/v1	Operations	OpenMetrics and Prometheus exposition for deposit, substitution, cache, queue, worker, quota, corruption, failure, and restore state.

ID	Plane	What it binds
GF-I09/v1	Owner-overlay handoff	A published <code>ApplicationRelease</code> , a published reusable <code>OverlayBundle</code> , and a separately signed <code>OwnerOverlaySelection</code> that binds both complete artifact references and the installation image, driven to a deterministic accepted or refused decision by verifier-owned source identity, payload and descriptor cryptography, and publication evidence.
GF-I10/v1	Nix remote build	Nix evaluation, dependency, and output semantics with derivation-granular execution on the shared REAPI fabric: a typed request binds tenant, source, lock, installable, system, registered builder, exposure policy, deadline, bounded capacity, and destination, and success requires qualified output custody, recursive publication of the complete closure, and independent substitution.
GF-I11/v1	Execution scheduling	GF’s own orchestrator, the <code>gf-reapi-cell</code> scheduler: executor-pool admission by the <code>gf.platform</code> capability property, per-tenant concurrency quota, worker-pool lease and slot dispatch, and the enqueued, started, completed, queued, inflight, and slot metrics that carry product throughput.

C The projected core

Figures 2 and 3 draw the component projection of the ratified core, transcribed from the generated projection at `docs/architecture/generated/flywheel-components.puml`, itself generated from `spec/flywheel/ComponentDiagram.dhall` [25]. Every node and edge below is one the projection itself names: three cache databases, five lane components, the withdraw and deposit edges each lane declares, the submission edges from the two caller lanes, and the projected invariants.

The control plane is a separate component of the same model and carries one field: it does not execute actions. The typed model asserts both that the canonical core is admissible and that the same core with an executing control plane is not [26]. Within the remote path, workers upload output content to the shared store and the cell validates the result before publishing action-cache state; workers hold no action-cache write authority [23].

D The corrected comparison, in full

The positioning record as the first edition carried it: every statement about another product is quoted from that product’s own documentation, read on 2026-09-17 and recorded in the competitive-position record [24]; every statement about GloriousFlywheel cites an axiom. No number here was measured by the comparison.

Blacksmith

Blacksmith optimises the CI runner. It is, in its own words, “A dead simple, drop-in replacement that costs 60% less than GitHub runners” [2], running jobs “in ephemeral Firecracker microVMs that boot in under 3 seconds”, with adoption as cheap as it

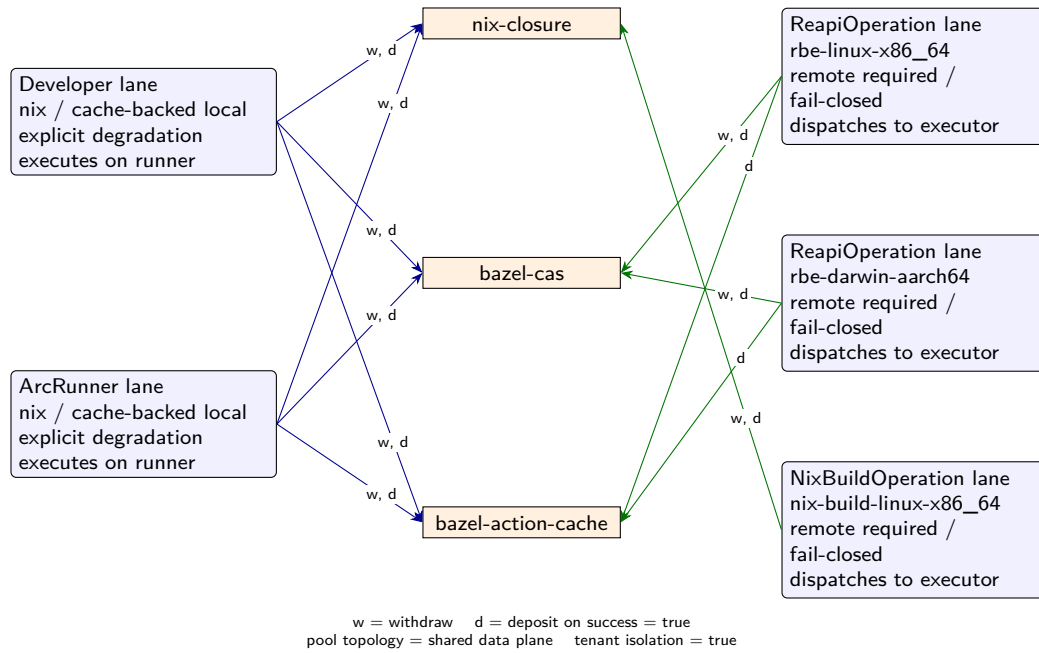


Figure 2: The five lane components of the ratified core and the three cache databases of the shared data plane, with the withdraw and deposit edges each lane declares. Every caller lane reads and writes all three caches; the two remote-required REAPI lanes carry the Bazel content store and the Bazel action cache; the Nix build lane carries the closure cache.

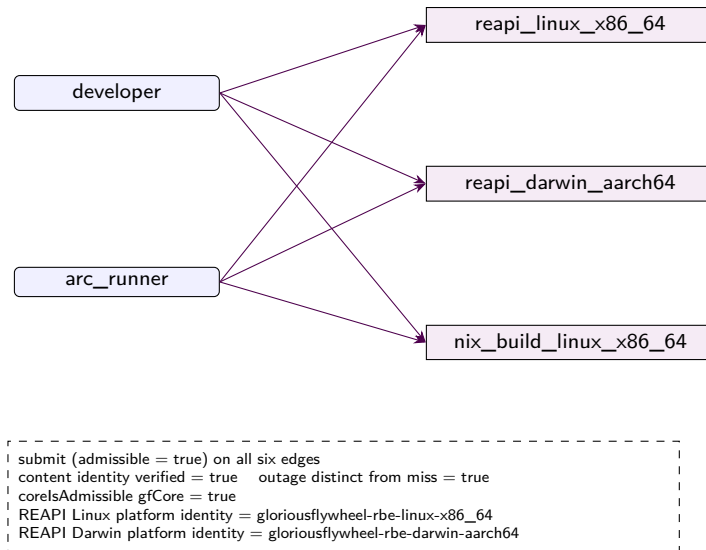


Figure 3: The submission edges of the ratified core. Both caller lanes submit to each of the three remote capabilities, and the projection reports every submission admissible. A core in which a developer cannot reach a remote capability is refused by the typecheck.

sounds: “Transitioning to Blacksmith is as simple as changing a single line in your workflow file” [3].

Corrected, the comparison is this. A CTO adopting Blacksmith changes the CI job’s host with a single workflow line and, for developers and coding agents, rents a second CI-identical microVM per agent that mirrors the working tree: “Testboxes sync your local changes to a Blacksmith microVM and run commands inside a real GitHub Actions job” [4]. The developer’s own machine never attaches to the cache. Blacksmith’s fixed reuse boundary is the repository across all four of its caches, with the Bazel cache further separated by CPU architecture: “Each repository has its own Bazel cache” [1]. On a Bazel cache miss the action executes inside the requesting job’s VM and the result is saved to that repository’s cache. Under this design the developer’s machine, the CI runner, and the pooled workers attach to one tenant, cache, and toolchain contract (GF-A01); the reuse boundary is tenant, digest, platform, and policy identity rather than the repository (GF-A05), with deposit rights still bound to a trusted identity and physical deduplication granting no logical visibility (GF-A08); and a miss on a remote-required lane is admitted to a compatible pooled worker or terminates with a typed failure (GF-A06, GF-A07), so one caller’s miss becomes every authorized caller’s hit. A warm developer iteration is a cache withdrawal, not an admitted action, so it consumes no pooled worker slot. What that costs in any billing unit is a measurement, and this paper reports none.

The parallelism claim holds as written: the parallelism of a single job’s work is bounded by the vCPUs of the VM that job rents, and the documented way past that bound is more jobs, each renting its own VM, whereas here the runner’s core count is not the bound on concurrent actions (GF-A12).

Buck2

Buck2 optimises the build tool. It describes itself as “A large-scale build tool. The successor to Buck.” [33] and, on design, “Buck2 is remote execution first - local execution is considered a special case of remote execution” [35]. It is a client of someone else’s remote-execution service, and it is a good one.

Corrected, the comparison is about where the decisions live. Whether an action may run locally, remotely, or race both, and whether a remote infrastructure failure falls back to local, are execution-platform fields a project’s rule authors choose [31], and the glossary offers to “race local and remote execution and get whichever finishes first” [32], so the same action can legitimately complete locally in one invocation and remotely in the next. Buck2 documents genuine developer-to-CI sharing when actions execute remotely, and that is to its credit: “everyone writes to the cache through sandboxed remote execution” [30]. Results of locally executed actions reach the shared cache only when the project’s execution platform opts in, subject to an upload size cap [31], and the open-source binding “uses Buck2 without remote execution” [35]. Buck2 ships no scheduler of its own: remote execution is consumed from a separately operated service named in the project’s configuration [34] and described as “scaling the nodes available for parallel actions, and by caching action outputs across Buck2 users” [32], with queue behaviour left to that service.

Under this design the local shape of a remote-dispatching lane is refused by the typecheck, local execution is not a qualifying origin for evidence at all, and the core must declare outage distinct from miss (GF-A06, GF-A07): there is no vocabulary in which a local alternative for remote-required work is admissible. Write authority is a server-side binding of authenticated identity to policy rather than a client-side switch, and a developer invocation acquires no action-cache write at all (GF-A08, GF-A11). Throughput is a product surface with objectives attached, owned by the fabric's own scheduler (GF-A12, GF-I11), rather than a question delegated to whichever backend a project attaches.

BuildBuddy

BuildBuddy optimises the Bazel remote-execution and cache backend, and then extends that element toward CI by wrapping the Bazel client itself in a warm machine. In its own words: “BuildBuddy is the open-core developer productivity platform built for Bazel.” [12]. Its executors are pooled: “By default, all BuildBuddy executors are placed in a single pool - and any task can run on any executor (running the same operating system and cpu architecture).” [14]. Its remote runners carry warmth as a whole-machine snapshot: “Running workloads in persistent execution environments using microVM snapshotting (on Linux) and persistent runners (on macOS).” [16].

Corrected, the comparison is not that BuildBuddy lacks a pooled executor; it has one, and a good one. It is that the cache is documented as a Bazel cache, shared across an organization by default with per-context keys and instance-name namespaces offered as opt-in separation, its own guidance being keys “in different contexts (i.e. one for CI, one for developers)” [10] and “separate remote caches, whether it’s for CI builds vs local builds” [15]; that uploading locally executed results is a client configuration flag [11] while cache-write permission travels with the key [10]; that the client’s default strategy list is “remote,worker,sandboxed,local” [15], so local execution remains a reachable strategy within the same invocation; that eviction of a blob the client believes present is documented as happening “without Bazel’s knowledge” with detection and recovery left to client flags [17]; and that selecting a specific pool per platform or target is written into build files as execution properties [13, 14].

Under this design a developer’s remote execution and the CI runner’s execution deposit into and withdraw from the same three caches on one shared data plane, so a Nix closure or Bazel result a developer produced remotely is withdrawn by CI under the same tenant, digest, platform, and policy identity (GF-A01, GF-A05). Deposit is a lane obligation performed by the executing cell, not a client flag, and writer authority is server-side (GF-A04, GF-A08). The ARC runner is typed as a thin admission shell, so there is no warm client artifact to snapshot, bill, or match by branch (GF-A12). An application repository’s checked-in ActionPlan names only an abstract capability class and never an executor pool, runner label, or provider endpoint (GF-A11); the controller joins that demand to provider supply. Whether this design is immune to the eviction case BuildBuddy documents is not established here and is not asserted.

Buildbarn

Buildbarn optimises the remote-execution backend behind a build tool. It is “an implementation of the Remote Execution protocol. This protocol is used by tools such as Bazel, Buck2, BuildStream and recc to cache and optionally execute build actions remotely” [7], assembled from single-purpose daemons; “Most setups will run a single instance of `bb_scheduler` and a large number of pairs of `bb_worker`/`bb_runner` processes” [8]. It is deployed at scale and it is well built.

Corrected, three of the five contrasts hold and two need narrowing. Attachment is a client-side setting each build tool carries and can omit; Buildbarn’s documentation stops at the server, and with only the storage daemon deployed, “build actions will still be executed on the local system” [9]. Both systems place action parallelism in the executor pool, so the honest contrast is not that Buildbarn leaves the farm unoptimised but that its documented contract ends at the worker and says nothing about the CI runner that calls it, whereas here the thin-runner shape is a typed contract of the CI lane: a lane that dispatches its work may not claim the runner’s node-local scratch, and recipes carry no local alternative (GF-A12). On fallback, Buildbarn’s storage layer distinguishes unavailable from not-found internally, but its contract ends at the returned status: whether the calling build tool retries, treats it as a miss, or executes locally is the client’s setting, and one documented decorator reports a size-limit refusal as not-found [5], which GF-A07 forbids. Here local execution of remote-required work is inadmissible at the lane, at admission, and as evidence, and the lane rather than the caller owns the reaction to unavailability (GF-A06, GF-A07). On tenancy, this design also routes Bazel data by `instance_name` on the wire (GF-I04), so the precise contrast is that the instance name a caller presents resolves to a tenant bound server-side through `ResolvedActionBinding/v1` (GF-I02, GF-A11) rather than being trusted as the tenant. Buildbarn’s authorizer is evaluated against authentication metadata from client certificates or bearer tokens [6], so an operator can bind instance names to an authenticated identity; the difference is that there this is operator-written policy whose documented examples default to allow-all and prefix matching, while here the binding is a typed invariant the core cannot omit. Buildbarn’s read-time completeness check also defends against eviction after action-cache publication, which a write-time gate alone does not; that is a real advantage and is recorded as one.

GitHub Actions

GitHub Actions optimises the CI runner as a disposable, autoscaled job host. It is “a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline”, and its unit of work is the job: “Each runner can run a single job at a time” [22]. Hosted execution is fresh per job: “With the exception of single-CPU runners, each GitHub-hosted runner is a new virtual machine (VM) hosted by GitHub” [19]. Scaling is arithmetic on jobs: “The target number of runners created will be calculated as a sum of `minRunners` and the number of jobs assigned to the scale set” [21]. It does that job well, with zero-maintenance hosted machines and a plan-backed concurrency entitlement.

Corrected, the comparison is this. Throughput here is bounded by admitted actions and attested worker slots in the shared exec-cache pool, never by how many ARC runners exist, and a dozen thin runners are designed to carry a hundred-plus concurrently executing actions (GF-A12); on GitHub Actions a runner runs one job at a time, ARC scales runner pods one per assigned job up to an operator-typed maximum, and hosted concurrency is a plan table with support able to raise it [20], so a CTO there sizes job concurrency because the job is the unit. A result deposited once here is withdrawable by any authorized caller with the same tenant, digest, platform, and policy identity, and the key carries no branch or caller identity (GF-A05), whereas “Caches are shared based on the branch or tag a workflow run uses, not on the identity of the workflow or job” [18], so sibling branches and developer machines cannot read it. The developer is an attached caller of the same pool here, and CI’s execution of a suite the developer already ran through the remote-execution surface withdraws the deposited results rather than recomputing them (GF-A01, GF-A05); on GitHub Actions the developer machine is outside the system, since the system is workflows triggered by repository events. Deposit here is a property of the lane type, so the pool warms as a side effect of every trusted success (GF-A04), while there the cache is an action a workflow author adds with an author-written key and no documented rule ties a job’s success to publishing anything. Finally, admission here reads a worker’s own measured attestation and admits a fail-closed one-slot default otherwise, and the supply axiom obliges every capacity claim to cite a measured reading with its timestamp (GF-A13); the typed attestation carries the measurement but not the timestamp, which remains an axiom-level obligation rather than a typed field. GitHub’s ceilings are a plan table, an operator-typed maximum concurrency, and an operator-typed runner maximum: computed from live assigned-job count, but declared rather than measured.

References

- [1] Blacksmith. Blacksmith caching: Bazel build caching, 2026. documentation page, read 2026-09-17.
- [2] Blacksmith. Blacksmith, 2026. product home page, read 2026-09-17.
- [3] Blacksmith. Blacksmith runners: Overview, 2026. documentation page, read 2026-09-17.
- [4] Blacksmith. Blacksmith Testboxes, 2026. documentation page, read 2026-09-17.
- [5] Buildbarn. Blobstore configuration protocol definition, 2026. bb-storage configuration proto comments, read 2026-09-17.
- [6] Buildbarn. gRPC configuration protocol definition, 2026. bb-storage configuration proto comments, read 2026-09-17.
- [7] Buildbarn. Buildbarn organization profile, 2026. project profile README, read 2026-09-17.
- [8] Buildbarn. bb-remote-execution README, 2026. project README, read 2026-09-17.

- [9] Buildbarn. bb-storage README, 2026. project README, read 2026-09-17.
- [10] BuildBuddy. Authentication guide, 2026. BuildBuddy documentation page, read 2026-09-17.
- [11] BuildBuddy. RBE with GitHub Actions, 2026. BuildBuddy documentation page, read 2026-09-17.
- [12] BuildBuddy. BuildBuddy introduction, 2026. documentation page, read 2026-09-17.
- [13] BuildBuddy. RBE platforms, 2026. BuildBuddy documentation page, read 2026-09-17.
- [14] BuildBuddy. RBE executor pools, 2026. BuildBuddy documentation page, read 2026-09-17.
- [15] BuildBuddy. RBE setup, 2026. BuildBuddy documentation page, read 2026-09-17.
- [16] BuildBuddy. Remote runner introduction, 2026. BuildBuddy documentation page, read 2026-09-17.
- [17] BuildBuddy. Troubleshooting RBE failures, 2026. BuildBuddy documentation page, read 2026-09-17.
- [18] GitHub. Dependency caching, 2026. GitHub Actions documentation page, read 2026-09-17.
- [19] GitHub. GitHub-hosted runners, 2026. GitHub Actions documentation page, read 2026-09-17.
- [20] GitHub. Limits for GitHub Actions, 2026. GitHub Actions documentation page, read 2026-09-17.
- [21] GitHub. Deploy runner scale sets, 2026. GitHub Actions documentation page, read 2026-09-17.
- [22] GitHub. Understand GitHub Actions, 2026. GitHub Actions documentation page, read 2026-09-17.
- [23] GloriousFlywheel. GloriousFlywheel core architecture, 2026. docs/architecture/recursive-dogfooding.md, revision 9ac83ff176f3c63c1bd2b2c137a830ffd305d108.
- [24] GloriousFlywheel. Competitive position: GF as a pooled $O(1)$ system, 2026. docs/agent-notes/2026-09-17-competitive-position.md in the glorious.build repository, revision cfa99160d3c7b03f8ff4fc292f8735fba5c6a564; competitor documentation read 2026-09-17.
- [25] GloriousFlywheel. Component projection of the ratified core, 2026. spec/flywheel/ComponentDiagram.dhall and its generated output docs/architecture/generated/flywheel-components.puml, revision 9ac83ff176f3c63c1bd2b2c137a830ffd305d108.

- [26] GloriousFlywheel. Structural architecture authority, 2026. `spec/flywheel/Core.dhall`, revision `9ac83ff176f3c63c1bd2b2c137a830ffd305d108`.
- [27] GloriousFlywheel. Golden objective: The refusal boundary, 2026. `docs/product/golden-objective.md`, revision `9ac83ff176f3c63c1bd2b2c137a830ffd305d108`.
- [28] GloriousFlywheel. Platform architecture, 2026. `docs/architecture/platform-layers.md`, revision `9ac83ff176f3c63c1bd2b2c137a830ffd305d108`.
- [29] GloriousFlywheel. RBE, cache and runner SLO definitions, 2026. `docs/build-system/slo.md`, operative v4 product objectives block, revision `9ac83ff176f3c63c1bd2b2c137a830ffd305d108`.
- [30] Meta Platforms. Benefits when compared to Buck1, 2026. Buck2 documentation page, read 2026-09-17.
- [31] Meta Platforms. CommandExecutorConfig, 2026. Buck2 build API documentation page, read 2026-09-17.
- [32] Meta Platforms. Buck2 glossary, 2026. Buck2 documentation page, read 2026-09-17.
- [33] Meta Platforms. Buck2, 2026. product home page, read 2026-09-17.
- [34] Meta Platforms. Buck2 remote execution, 2026. Buck2 documentation page, read 2026-09-17.
- [35] Meta Platforms. Why Buck2?, 2026. Buck2 documentation page, read 2026-09-17.
- [36] Jess Sullivan. Decision: the pitch thesis, as the operator ratified it. GloriousFlywheel repository, `docs/decisions/pitch-thesis-2026-09.md`, 2026. Ratified 2026-09-19; the O(1) software development lifecycle thesis.