A QUERY FARM TECHNICAL REPORT

Building DuckDB's community extensions, faster and more consistently

The case for treating DuckDB's extension catalogue as a distribution — built, tested, and shipped for every release

Query Farm LLC · 2026-06-18

Summary

DuckDB is superb at being a database engine. But upgrading it can still break things: install a new DuckDB, run `INSTALL airport; LOAD airport;`, and you may be told the extension is not available for your version — because each community extension is a separate binary that must be rebuilt for every engine release. Building the engine and **distributing** the software around it are different problems, and the distribution side has *two*: at an engine release the extensions aren't built yet, and between releases it is unpredictable how long an extension update takes to ship. We argue — with data — that DuckDB's extension catalogue is best treated as a **distribution**, the way Ubuntu is a distribution of Linux. Haybarn is that distribution for DuckDB — installable today with `npx haybarn@rc` or `uvx haybarn-cli` — and the findings below measure what it delivers.

- **Problem 1 — on an engine release, the extensions aren't ready.** A new version is announced when the *engine* ships, but every extension binary is locked to that version and must be rebuilt and re-published first — even a point release requires rebuilding the whole catalogue. Until it catches up, upgrading can mean `INSTALL`/`LOAD` errors (§2).
- **Problem 2 — between releases, extension updates land unpredictably.** Independently of engine releases, fixes and new versions flow as pull requests to `duckdb/community-extensions`; how long they take to merge and publish is rising and increasingly variable (§3) — so a service built on DuckDB cannot reliably tell customers *when* an update will reach them.
- **Haybarn builds the same extensions about 2.4× faster.** On *identical* GitHub-hosted runners, it rebuilds the catalogue about 2.4× faster end-to-end than `duckdb/community-extensions` — 3.1–3.4× on Windows-MinGW and WebAssembly, 1.5× on macOS. The difference is in the build pipeline, not the hardware: a *read-write*, Cloudflare-backed `vcpkg` dependency cache, plus a *fixed pre-built container* on the Linux legs in place of an inline Docker rebuild.
- **The speed-up multiplies across the estate.** One full-catalogue rebuild costs a measured **104 CI-hours** on Haybarn versus a modeled **245** at DuckDB build speed (Appendix A) — roughly **58 %** less. At DuckDB's cadence (10 releases/year) that is on the order of 1,408 CI-hours a year.
- **The cache is cheap and demonstrably effective.** During the v1.5.4 build it served **780,998 reads** (Class B) against **51,035 writes** (Class A) — a 15.3 : 1 ratio, peaking at 268,109 reads/hour. Cloudflare R2 charges nothing for egress; all-in cost is a dollar or two a month.

1. What Haybarn is, and why DuckDB needs a distribution

Haybarn is an independent, derived distribution of DuckDB published by Query Farm LLC (github.com/Query-farm-haybarn). It is not affiliated with or endorsed by the DuckDB Foundation; DuckDB is a trademark of the DuckDB Foundation.

DuckDB ships two things that travel together but are built and released separately. The **engine** — the `duckdb` binary and its embedded libraries — is the equivalent of an operating-system kernel: a fast-moving core that defines an ABI. The **community extensions** — over 250 of them — are the packages: independently-authored capabilities (geospatial, formats, connectors, ML) that must be compiled **against a specific engine version** and distributed as per-platform binaries.

A new engine release does not carry its extensions with it. Every extension binary is keyed to the engine version it was built against; when the engine moves, the entire constellation must be re-compiled and re-published before users on the new version can install any of it. This is the same split Linux has between a **kernel** and a **distribution**. DuckDB Labs has solved the hard part — the engine — brilliantly; what is *not* yet solved is the binary **distribution** and devops around it: compiling, caching, testing, and publishing the entire extension catalogue for every release, on every platform. That is distribution work, and it is exactly what Ubuntu does for Linux — take a fast-moving core and ship a coherent, tested, fully-packaged system on top of it. Few organisations run a raw kernel in production; they run Ubuntu.

This is not a knock on the DuckDB team so much as a matter of scope and capacity. A focused core team has its hands full advancing the engine, and the long tail of commercial devops and packaging concerns is a different, open-ended job — one the engine roadmap should not have to absorb. Several of those concerns are already concrete:

- **Distribution channels.** The official catalogue is served from a single origin. Haybarn additionally serves extensions from Cloudflare's global edge network and publishes them to **npm** (with **PyPI** planned), so they install through the package managers and CDNs teams already rely on — not a single point of distribution.
- **Versioned history.** The official catalogue exposes only the *latest* build of each extension: upgrade and you take whatever is current, with no way back. Haybarn keeps a **full versioned history** of every extension build, so an application can pin — and later reproduce — a specific older version instead of being forced to the latest.
- **Predictable, reproducible builds.** The fast, cache-backed, fixed-container pipeline of §4–§5 rebuilds the whole catalogue for a new engine quickly and identically, closing the availability gap of §2 rather than living with it.

These are the everyday expectations of a distribution — multiple channels, version pinning, reproducible rebuilds — and they are exactly what commercial users need but do not yet get from the engine project alone.

Haybarn is a distribution of DuckDB, in the sense Ubuntu is a distribution of Linux. It tracks each engine release, rebuilds and tests the whole community-extension estate against it across the full platform matrix (Linux, macOS, Windows, WebAssembly; x64 and arm64), and publishes the binaries so that "upgrade the engine" and "all my extensions still work" become

a single, consistent event rather than two events separated by an unpredictable gap of days. Crucially, on the Linux legs each build runs inside a **fixed, pre-published container image** rather than one rebuilt from scratch per job, so the toolchain is identical across the estate — reproducibility a distribution depends on, and, as we show below, a large source of its speed.

2. The extension packaging gap: engine release ≠ extensions available

When DuckDB announces a core release, the community-extension binaries for that version do not yet exist — they are produced afterwards, extension by extension, as each is rebuilt and deployed. That window can stretch from hours to days, and it applies to more than major versions: a point release requires rebuilding every extension too, so each patch must wait for the catalogue to be rebuilt before it is whole. During that window a user who upgrades on the strength of the announcement and then runs `INSTALL airport; LOAD airport;` can be told the extension is unavailable for their version. The engine works; the **distribution** is not yet ready. And because that window varies — sometimes hours, sometimes days — it is increasingly hard to predict *when* it will close, leaving both customers and the developers who serve them wondering when it is actually safe to upgrade.

This is a packaging-readiness problem, and other ecosystems solve it by convention. A Debian release is not declared stable until its package set is built and frozen; the kernel advancing upstream does not, by itself, push a release to users. DuckDB core advances on its own cadence — on the order of 10 releases a year (12 in the trailing twelve months), per the official release calendar[1] — and community extensions are version-locked: each must be compiled against, and published for, the engine versions it targets. A release therefore is not fully usable until its extensions have been rebuilt and published. For commercial users, **predictability** — knowing that upgrading the engine will not silently break the extensions they depend on — is not a nicety but a precondition for adoption. Predictability is key for commercial success, and it is exactly what a coherent "engine and extensions, jointly available" moment provides.

We are not arguing that core releases should block on the entire catalogue; doing so perfectly is genuinely hard. We are observing that the **absence of any consistency window** pushes the integration cost onto every user individually, and that a distribution is the natural place to absorb it.

3. Between engine releases, extension merge times are getting less predictable

The packaging gap above is most visible at engine-release time, but the underlying pipeline runs year-round: extensions don't only change when DuckDB does — authors ship fixes and new versions continuously, each as a pull request to `duckdb/community-extensions`. The predictability of *that everyday, inter-engine pipeline* is what a commercial user ultimately depends on, and it is degrading. If you build services or products on DuckDB, that makes it harder

[1] Release dates and cadence: duckdb.org/release_calendar. Per that page, "larger new features are introduced in minor versions, while patch versions mostly contain bugfixes"; since 1.4.0 every other release is an LTS edition.

to tell customers *when* an extension update will actually be distributed to them. We analysed every PR merged there in the trailing year (1,435 PRs; 1,027 of them extension **release/version** PRs — "add extension", version bumps, and updates) and measured end-to-end time-to-merge (`merged_at − created_at`), bucketed by month.

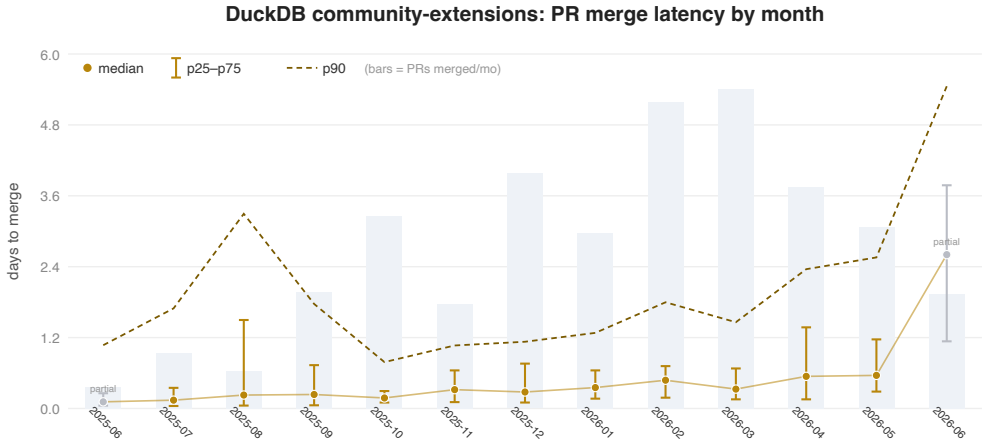

Figure 1: Monthly PR merge latency for `duckdb/community-extensions`. Median (dot) with p25–p75 error bars and a dashed p90; pale bars are PRs merged that month. The first and last months are partial windows (greyed) and are excluded from the trend read.

To be clear, this is not a knock on the maintainers, who are notably responsive: for much of the year a typical PR merged within hours. The point is the **trend**. Across complete months, median time-to-merge rose from about 3.4 h (2025-07) to about 13.5 h (2026-05), with p90 widening from 40.8 h to 61.6 h, as monthly submission volume grew several-fold. (The partial June window, n = 79 and overlapping a release crunch, is greyed and set aside.) Note this is *end-to-end* latency — it includes author and CI round-trips, not only maintainer response-time — so it measures how long a version change takes to land overall.

Release/version PRs are about 72% of all merged PRs and track the same upward trend, so the combined view above represents them; we omit a near-duplicate second chart.

The mechanism matters for consistency. Throughput here is ultimately bounded by maintainer attention, which is finite and batched into working hours and time zones. As submission volume keeps rising, **when** a version lands becomes harder to predict — and that unpredictability, more than any median, is what makes a consistent release experience hard to guarantee. It is also the front of the same pipeline whose tail end is the `INSTALL`/`LOAD` gap of §2: a version is not usable until it has merged, rebuilt, and published.

> **Where Haybarn helps — and where it stops today.** Haybarn already removes the *build-and-distribute* half of this lag: once a change is in the catalogue, it is rebuilt and published across every platform in a fast, predictable window (§4–§5). The *merge* half is still upstream's — today Haybarn tracks `duckdb/community-extensions` by **syncing its contents**, so an extension update reaches Haybarn only once it has merged upstream. Because a distribution is also the natural place to *own* the merge step, Haybarn is positioned to offer faster, predictable merges directly — decoupling a customer's update timeline from upstream merge variance — though that is a future capability, not what it does today.

4. The build-time gap comes down to caching

Community extensions are mostly thin wrappers over heavy C/C++ libraries pulled in through `vcpkg`. The dominant cost of a build is therefore not compiling the extension — it is compiling those third-party dependencies. Whether that cost is paid depends almost entirely on the build cache, and this is where Haybarn and DuckDB differ by design.

DuckDB builds against a read-only cache. `duckdb/community-extensions` points `vcpkg` at the shared, *read-only* cache `vcpkg-cache.duckdb.org`. This is a sound default for a repository that builds untrusted pull requests: a read-only cache means a contributor's PR can never poison the shared store, and it serves popular, already-populated dependencies well. Its limitation is that it cannot *self-populate* — on a miss (a new dependency, compiler, or triple) nothing is written back, so miss-prone targets like Windows-MinGW and the WebAssembly triples recompile the same dependencies from source on every run.

Haybarn builds against a read-write cache it operates. Query Farm rewired its extension-CI toolchain[2] so that `vcpkg`'s *binary* cache (compiled dependencies) and *asset* cache (source tarballs) both point at a **read-write** store fronted by a **Cloudflare Worker backed by R2 object storage**. Reads are anonymous; writes are authenticated. A dependency is compiled once, written back on success, and reused by every later build across the whole estate. The asset cache additionally acts as a transparent mirror with origin fallback, so transient source-download outages no longer fail builds — improving build *reliability* alongside speed. (A `ccache` hashing fix for the WebAssembly toolchain compounds the effect.)

The result, measured on the **same** GitHub-hosted runner types via the same `build.yml` → `_extension_distribution.yml` matrix, is a per-platform speed-up that is largest exactly where the caching matters most. We compared 14 widely-used extensions built on both CIs, combining the Haybarn webhook-collector snapshot with timings pulled from the DuckDB GitHub Actions API.

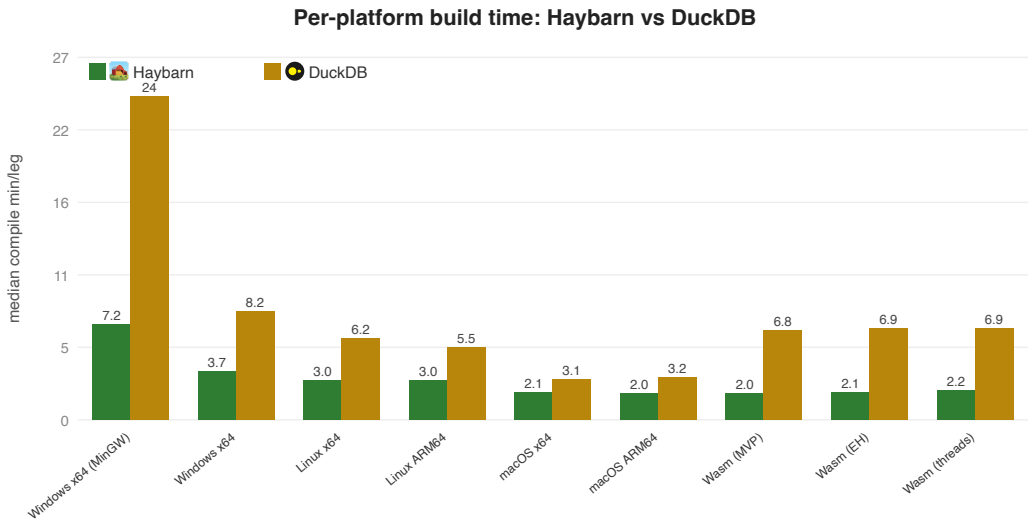

Figure 2: Median compile minutes per platform leg, pooled across 14 extensions (1,200 Haybarn and 1,373 DuckDB legs). Lower is faster; the gap is largest on Windows-MinGW and the WebAssembly triples.

[2] `Query-farm-haybarn/haybarn-extension-ci-tools`, the Haybarn fork of `duckdb/extension-ci-tools`.

Platform	Haybarn (min)	DuckDB (min)	Ratio (DuckDB÷HB)
Windows x64 (MinGW)	7.2	24.5	**3.4×**
Windows x64	3.7	8.2	**2.2×**
Linux x64	3	6.2	**2.1×**
Linux ARM64	3	5.5	**1.8×**
macOS x64	2.1	3.1	**1.5×**
macOS ARM64	2	3.2	**1.6×**
Wasm (MVP)	2	6.8	**3.4×**
Wasm (EH)	2.1	6.9	**3.3×**
Wasm (threads)	2.2	6.9	**3.1×**

Table 1: Per-platform median compile time, pooled across the comparison set.

A full-catalogue rebuild is about **2.4× faster** on Haybarn end-to-end (§5), and the per-platform legs explain why: ratios range from 1.5× on macOS to **3.1–3.4× on Windows-MinGW and the WebAssembly triples** — the dependency-heavy, cache-miss-prone targets. That is the signature of a cache that is populated and reused versus one that cannot self-populate. It is not a hardware advantage — both run on standard GitHub-hosted runners — but a build-pipeline one: Haybarn operates a writable, estate-global dependency cache (and takes on authenticating its writes) where a public PR repository is, reasonably, limited to a read-only one.

4.1. Where the time goes: a5, step by step

Drilling into one extension makes the mechanism concrete. `a5` is the *floor case*: a small Rust extension with almost no native dependencies, so the dependency cache saves little here and what remains is mostly *fixed per-build overhead*. The chart breaks its `linux_amd64` leg into steps on both CIs.

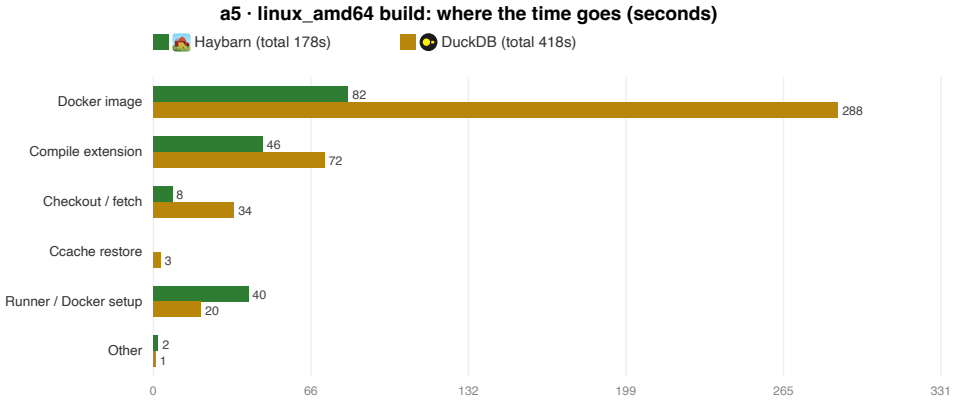

Figure 3: a5 `linux_amd64` build, seconds per step-group. The build-environment container dominates: rebuilt inline on the DuckDB Linux leg vs pulled pre-built on Haybarn.

The single largest difference is the **build-environment container**. On its Linux legs DuckDB rebuilds the Docker image inline (288 s for a5); Haybarn pulls a pre-published image from GHCR (82 s) — saving about 3–4 minutes on this leg alone. Because that is pure fixed overhead, it is paid on every Linux leg of every extension; Haybarn's CI notes estimate 5–8 minutes per leg for heavier images, on the order of 70 CI-hours across a 150-extension sweep. Even compiling a5 itself is 46 s versus 72 s — and this is the telling part: a5 has almost no dependencies, yet it still links against **DuckDB core and its static libraries**, which Haybarn reuses from cache

rather than recompiling. A no-dependency extension already shows a gap; dependency-heavy extensions show far more.

Two engineering details enable this. First, Haybarn ships a newer **ccache** (4.13.6, versus 4.11.3 in the stock toolchain) specifically because older versions lack the bearer-token authentication needed to talk to a remote HTTP cache — without it, ccache cannot use a shared backend at all. Second, DuckDB's compiler cache is GitHub Actions' built-in `actions/cache`: per-repository, branch-scoped, and LRU-evicted around 10 GB, so it cannot share compiled objects effectively across hundreds of independent extension builds. Haybarn's cache is *global to the estate* — an object compiled for one extension is available to every other. Together with the fixed container, this also makes builds more **reliable**: no per-run package/Docker-layer drift, and the asset-cache mirror removes a common class of transient download failures.

5. Multiplying across the estate

A per-extension speed-up is interesting; the **estate-wide** effect is the point. Of DuckDB's roughly 250-extension catalogue, Haybarn rebuilt 210 in its v1.5.4-rc1 sweep; the 14 extensions of §4 are the subset we could time on both CIs. Shipping the estate for a new engine version means rebuilding all of them, on every platform. We *measured* one such Haybarn sweep and *modelled* the DuckDB-equivalent cost by scaling each platform's measured Haybarn compute by that platform's observed ratio — a projection of §4, not a second measurement (Appendix A).

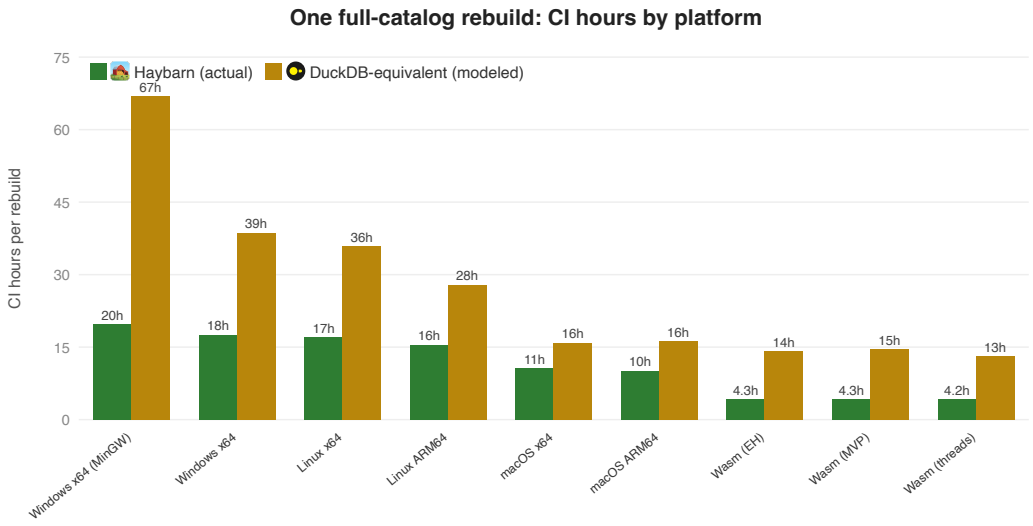

Figure 4: CI-hours for one full-catalogue rebuild, by platform: Haybarn *measured* vs the *modelled* DuckDB-equivalent. Windows and Linux native targets dominate cost and savings.

Platform	Legs	Haybarn h	DuckDB-eq h	Ratio
Windows x64 (MinGW)	128	19.8	67.4	3.4×
Windows x64	160	17.7	38.9	2.2×
Linux x64	202	17.2	36.1	2.1×
Linux ARM64	201	15.6	28.1	1.8×
macOS x64	189	10.7	16	1.5×
macOS ARM64	192	10.2	16.3	1.6×
Wasm (EH)	122	4.3	14.3	3.3×
Wasm (MVP)	121	4.3	14.6	3.4×
Wasm (threads)	122	4.2	13.2	3.1×

Table 2: One full-catalog rebuild. “DuckDB-eq” applies each platform’s measured ratio to Haybarn’s actual compute — a modeled counterfactual, not an observed DuckDB sweep.

That measured sweep cost **104 CI-hours** on Haybarn. At DuckDB build speed the same sweep is a modelled **245 CI-hours** (scaled from the per-platform ratios of 14 extensions; indicative) — roughly **58 %** more, or about 141 CI-hours per release. At DuckDB’s cadence (10 releases/year) that is on the order of 1,408 CI-hours a year, and it grows with every extension added. Just as important as the total is the **predictability**: because cache hits dominate, build times are stable and failures from source-download flakiness are largely eliminated, so a full-estate rebuild lands in a known, short window. The faster and more reliably the estate builds, the shorter the packaging gap of §2, and the more affordable it is to rebuild promptly and often.

Estate-completion *latency* compounds the same overheads. Both CIs run their platform legs in parallel — each waits only on matrix generation, not on the Linux leg — so the binding constraint is how quickly individual legs clear and how long they queue for scarce runners (macOS most acutely). Two things hurt here, and Haybarn fixes both: every leg pays its fixed container and setup cost *before* any useful compilation, so eliminating the inline Docker rebuild shortens the critical path of the whole sweep, not just its compute; and cache-warm legs finish sooner, returning runners to the shared pool and pulling the entire estate’s completion time forward.

6. The economics: shared concurrency, not dollars

It is tempting to say “CI compute isn’t free”, but for public repositories GitHub Actions *is* monetarily free. The real constraint is different and more binding: Actions **concurrency is a fixed, shared pool** per organization. Every job a full-catalog rebuild queues competes with every other piece of work the org needs to run. A rebuild is one large, indivisible task — the engine release is not “done” until the last extension lands — so halving its compute (§5) means the org clears that task in roughly half the runner-time and frees concurrency for everything else sooner. Speed here buys **throughput and a shorter packaging gap**, which is exactly the resource that is scarce.

Against that, Haybarn’s marginal cost is strikingly small. The cache is an R2 bucket behind a Cloudflare Worker; it holds **75.68 GB**, which at R2’s $0.015/GB-month is on the order of $1.14/month of storage at list price — and R2 charges **nothing for egress**, so the cache is read on every build with no bandwidth bill. A bounded object TTL keeps the bucket from growing

without limit as triples and dependency versions churn. All in — storage plus the operations below — the cache costs about a dollar or two a month.

The read/write mix confirms the cache is working as intended. Over the v1.5.4 build window the bucket served **780,998 reads** (Class B — `GetObject` + `HeadObject`) against **51,035 writes** (Class A — `PutObject` and lists), a **15.3 : 1** ratio: dependencies are compiled and written once, then served from cache many times. The chart traces R2 requests per hour through that build, where reads peaked at **268,109 per hour** while writes stayed flat — the cache absorbing the sweep almost entirely from already-compiled artifacts.

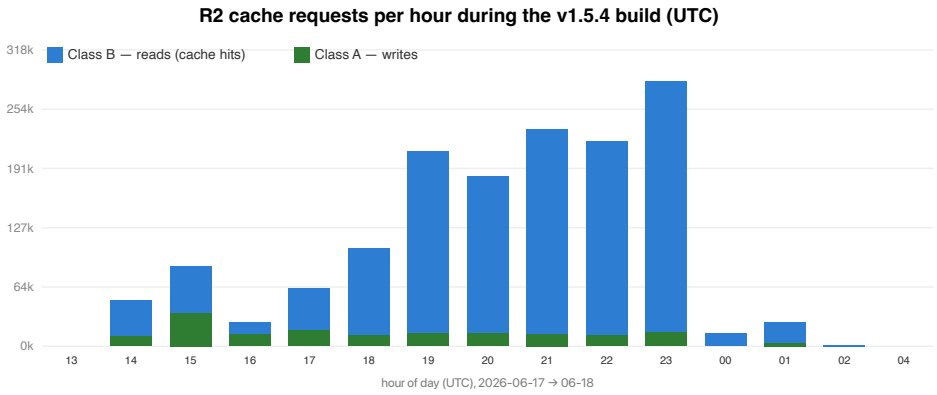

Figure 5: R2 cache requests per hour during the v1.5.4 full-catalogue build (UTC); empty hours omitted. Reads (cache hits) dominate writes throughout — the cache, not recompilation, carries the sweep.

R2 bills Class B at $0.36/million and Class A at $4.50/million, so a read-dominated cache is also the inexpensive direction to be busy in.

There is a broader point: dependency caching may be the single highest-leverage use of object storage in this pipeline. Every extension build re-compiles DuckDB core and a stack of shared C/C++ libraries; storing those compiled artifacts once and serving them back trades a few dollars of storage for hundreds of runner-hours. The per-repository GitHub Actions cache cannot capture that value because it is not shared across extensions — which is exactly why the remote, estate-global cache pays for itself many times over.

The displaced compute also carries a small environmental cost — an estimated 7–76 $kgCO_2e$ a year avoided (Appendix A), modest in absolute terms but scaling with the catalogue. The larger point is direction: efficient caching turns "rebuild everything for every release" from a cost that discourages frequent, consistent packaging into one cheap enough to do routinely.

7. Discussion

Three findings reinforce one conclusion. (1) Engine releases outpace extension availability, leaving a gap that surfaces to users as installation errors. (2) The PR pipeline that feeds extension releases is bounded by maintainer attention and shows rising, less-predictable latency as volume grows. (3) Rebuilding the whole estate is expensive without estate-global caching, which makes "just rebuild everything promptly" costly to do.

A distribution addresses all three at the point of integration rather than pushing them onto every user: it rebuilds and tests the whole catalogue against each engine release, fast and cheaply enough to do consistently, and presents a coherent "engine + extensions are jointly ready" moment. That is the role Haybarn fills. None of this is a criticism of DuckDB Labs, whose

engine, public vcpkg cache, and CI tooling are what make a distribution possible in the first place — we build directly on them. We are not proposing that DuckDB gate its releases on the full catalogue; we are surfacing the gap, quantifying its cost, and noting that — as Ubuntu is for Linux — a distribution is the established way ecosystems give users a consistent experience. Where the techniques here are useful upstream — the read-write cache design, the pre-built build images, the `ccache` upgrade — we would be glad to contribute them back.

8. Try Haybarn

Haybarn's distribution of the DuckDB community extensions is installable today:

```
npx haybarn@rc   or   uvx haybarn-cli
```

It tracks each DuckDB engine release and ships the whole extension catalogue — rebuilt and tested across every platform. Source, issues, and the build tooling are public at github.com/Query-farm-haybarn more at query.farm and we welcome questions and collaboration at hello@query.farm.

9. Appendix A — Methodology and assumptions

9.1. Data sources

- **Haybarn build timings:** a Cloudflare Durable Object that records every GitHub organization webhook for the Haybarn org, snapshotted locally to DuckDB. Tables `workflow_runs` and `workflow_jobs` carry per-job `started_at`/`completed_at`.
- **DuckDB build timings:** the GitHub Actions REST API for `duckdb/community-extensions` (we do not collect its webhooks). For each comparison extension we mapped recent commits touching `extensions/<name>/description.yml` to their `build.yml` run and read per-platform job durations.
- **Merge latency:** every PR merged to `duckdb/community-extensions` in the trailing year, via the GitHub API.
- **Release cadence:** the official DuckDB release calendar (duckdb.org/release_calendar).
- **Caching / container / ccache architecture:** read directly from `haybarn-extension-ci-tools` (`_extension_distribution.yml`, `docker/*/Dockerfile`) and the DuckDB `extension-ci-tools` defaults — the differences (read-only vs read-write `vcpkg` cache; inline `docker build` vs pre-published GHCR image; `actions/cache` ccache vs remote HTTP cache; ccache 4.11.3 vs 4.13.6) are documented configuration differences, not inferences.
- **a5 step timings:** the GitHub Actions jobs API, `linux_amd64` build leg, for the a5 run on each CI; raw step durations bucketed into the categories shown.
- **Cloudflare R2 metrics:** the Cloudflare GraphQL Analytics API (`r2OperationsAdaptiveGroups`) over the trailing 24 h for the `haybarn-build-cache` bucket, with `actionType` mapped to R2's Class A / Class B / free billing classes. Storage and pricing per R2 pricing ($0.015/GB-month; egress free).

9.2. Definitions

- **Build leg / compile time:** one platform's compile job, `completed_at − started_at`. We compare compile time only and exclude queue wait (`created_at → started_at`), which reflects each org's CI concurrency contention, not build speed.
- **Extension attribution (Haybarn):** the run title 💫 `<ext> (engine)`; platform from the job-name matrix parameters.
- **Release/version PR:** title beginning "add"/"update", or containing "bump", "version", or "release". A heuristic; it over- and under-counts at the margins.
- **Merge latency:** `merged_at − created_at`. It does not isolate review iteration or the moment a PR became review-ready, so it is an upper bound on pure maintainer wait.

9.3. Estate counterfactual

Haybarn's full-catalog rebuild compute is **measured** (the rc1 sweep). The "DuckDB-equivalent" is **modeled** by multiplying each platform's measured Haybarn compute by that platform's observed median slowdown ratio. DuckDB does not perform single-shot full-catalog rebuilds the same way, so this is a like-for-like cost model, not an observed DuckDB sweep. The comparison set is 14 widely-used extensions; DuckDB samples are sparser per extension (single-CI builds), so per-platform ratios are indicative.

9.4. Carbon model

Energy = runner-hours $\times$ P $\times$ PUE; carbon = energy $\times$ I. Reported as a range:

Parameter	Low	Central	High
Avg active runner power P (kW)	0.04	0.06	0.09
Grid carbon I (gCO_2e/kWh)	100	400	500
Data-centre PUE	1.2	1.2	1.2

Table 3: Carbon-model parameters. Runners are heterogeneous (cloud VMs for Linux/Windows, hosted Macs for macOS); a single blended power figure is a deliberate simplification, hence the wide range.

9.5. Caveats

The trailing-year window's final month is partial and overlaps a core-release crunch. Cancelled and in-progress jobs are excluded. Carbon figures are order-of-magnitude. All input data, queries, and chart code are reproducible from the analysis scripts (`fetch_data.py`, `fetch_upstream.py`, `prep_report.py`).