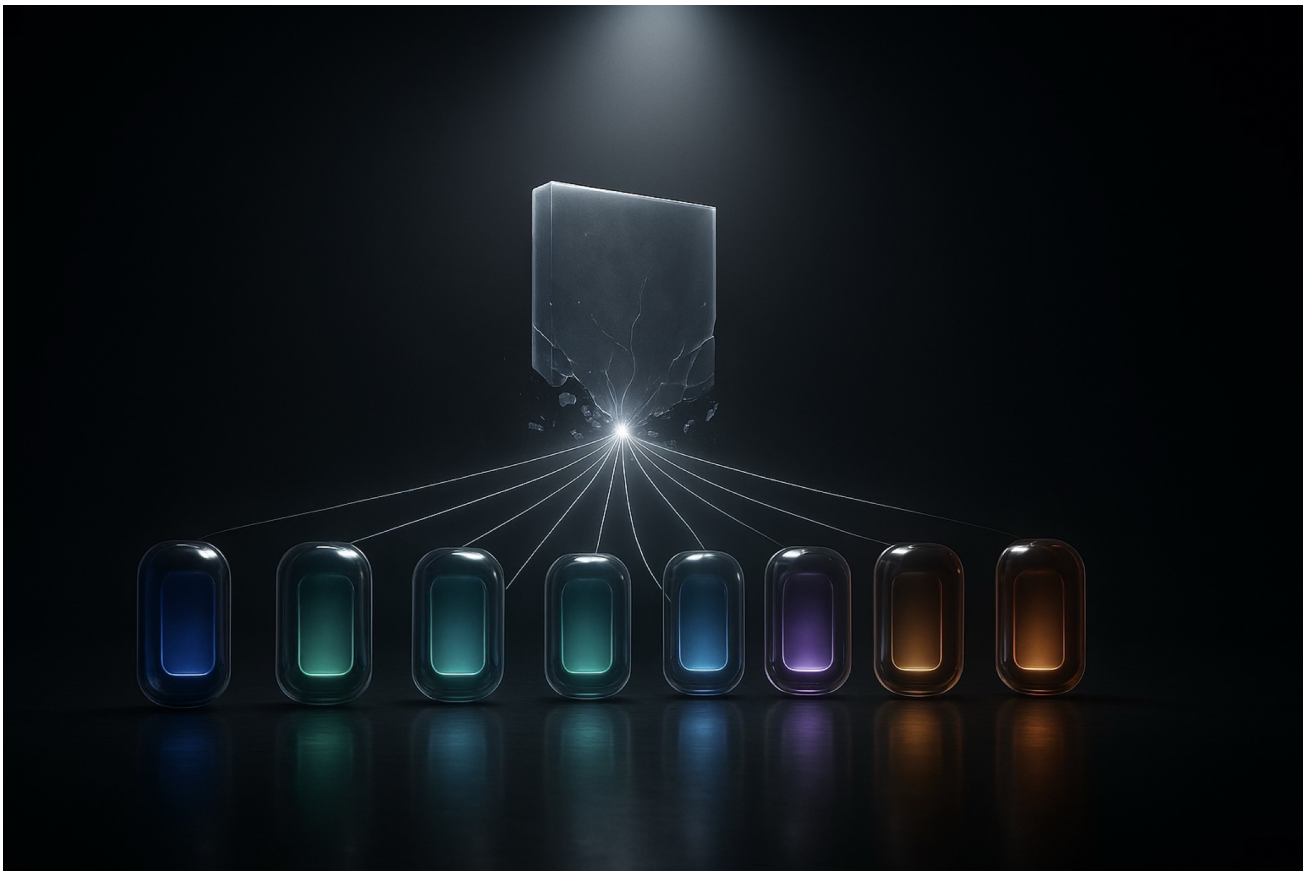

cupertino v1.3.0: The Big Refactor

This release started with a number that looked too small to matter.

Back in v1.2.0 I had added deployment-aware search: ask cupertino for an API and tell it which OS you actually ship, and it filters out the answers that would not compile for you.

```
cupertino search swiftui --source samples --min-ios 16
```

The flag parsed. The database carried availability metadata. The command looked completely right. And yet, in the dev corpus, `--min-ios` handed back the same 88 results whether I asked for iOS 13 or iOS 18. A filter that returns the same set for every value is not filtering. It is decoration.

That is the kind of bug that looks like a one-line fix until you ask why it was possible at all.

A filter that does not filter is worse than no filter. It lets an assistant claim it respected your deployment target while quietly handing you an iOS 18 API for an app that still ships iOS 16. That is not an answer. It is a future compile error wearing a green checkmark.

I could have found the missing parameter, threaded it through the one path that dropped it, added a test, and shipped. One path fixed. The reason that path could go missing in the first place: untouched.

So I pulled on it instead.

The problem under the bug

cupertino had outgrown the shape it was born in.

Early on, the whole thing fit one mental model: a big Apple documentation index with a few extra sources bolted on. Apple Developer Documentation, the HIG, Apple Archive, Swift Evolution, swift.org, and *The Swift Programming Language* all lived inside a single SQLite file. Samples and packages had their own databases, but most of the command surface still thought in three buckets:

```
search.db
samples.db
packages.db
```

That shape was fine when the project was small. It quietly stopped being fine the moment cupertino became a genuine multi-source tool, and the symptoms were everywhere if you knew to look. A platform filter honored in one source and silently dropped in another. A doctor probe pointed at a database filename that a per-source build no longer produced. A release script that knew the names of its databases by heart.

Those were not five unrelated bugs. They were one disease with five rashes. Source identity was scattered across CLI switches, MCP handlers, setup checks, release manifests, and tests, each carrying its own hand-copied list of names. Change the world in one place and the other four kept believing the old map.

The cure was already half-built. The Source Independence Day effort had been pushing toward a single source of truth in code: a registry every part of the system reads from, so adding a content source becomes one composition-root change instead of a scavenger hunt. The code was moving that way. The shipped artifact was not. The bundle users actually installed still wore the old three-bucket shape.

So v1.3.0 became the release that drags the artifact up to meet the architecture.

Not because retiring a database filename is exciting. It is not. Nobody installs cupertino to admire the shape of its SQLite files. It matters because that old shape was a tax. Every new source paid it. Every cross-source feature paid it. v1.3.0 is the refactor that stops collecting it.

One registry, eight databases

In v1.3.0 a source is no longer a string in a `source` column.

It is a registered provider that declares everything about itself: where its data lives, how it fetches, how it reads, how it searches, which metadata it carries, and which filters it can honestly apply. One registry holds them all, and the rest of the system asks the registry instead of remembering.

The release bundle now matches that model exactly. Instead of one giant docs database plus two siblings, v1.3.0 ships eight per-source databases:

```
apple-documentation.db
hig.db
apple-archive.db
swift-evolution.db
swift-org.db
swift-book.db
apple-sample-code.db
packages.db
```

`cupertino setup` downloads `cupertino-databases-v1.3.0.zip`, a 742 MB bundle that expands to about 3.9 GB. Most of that is one file: `apple-documentation.db` weighs 2.6 GB and

holds 351,505 documents and 240,543 symbols across 398 frameworks. Next is `packages.db` at 1.0 GB with 185 packages. The HIG, archive, Swift Evolution, swift.org, and Swift book databases are the small ones, a few megabytes each.

The split is not cosmetic. It changes what can go wrong independently. If the HIG corpus is stale, the HIG database is stale and Apple API search is not dragged down with it. If packages need a schema bump, packages get one without reindexing everything else. And when a new source lands, the release tool derives its database straight from the registry. No human goes hunting for the one more filename list that forgot to grow.

Users never see any of this. The fan-out search surface stays exactly as simple as it was. Underneath, SmartQuery dispatches across the independent source databases and fuses the results with reciprocal rank fusion, so one source can degrade without taking the whole query down with it. That is the trade the refactor buys: invisible to the person typing a query, decisive for the person adding the next source.

The old public shape had to break

Some flags described a world that no longer exists, so they had to go.

The legacy `--search-db` override is gone from all six commands that carried it: `search`, `read`, `save`, `doctor`, `list-frameworks`, and `inheritance`. It made sense when there was one docs database to point at. It is meaningless when every source resolves through the registry to its own file, and keeping it around would only preserve the wrong mental model. Scripts that still pass `--search-db` now get a clear error instead of a silent misread.

The same honesty went inward. Roughly 230 `searchDB` and `searchDb` identifiers across the CLI, services, indexer, and search layers became the generic `dbURL` and `dbPath`, because there is no longer a single "the db" to name. The release tool stopped bundling a hardcoded trio of filenames and now asks the production registry for every enabled source's destination database. And `databaseVersion` moved to `1.3.0`, so a v1.3.0 binary pulls the per-source bundle while a v1.2.x binary keeps pulling the old unified one. Upgrades and downgrades both land on a bundle they understand.

The one place the literal `search.db` name survives is the upgrade shim, which has to recognize a real pre-v1.3.0 file by its old name in order to migrate it. Everywhere else, the old name is finally just history.

Then SQLite made the refactor earn it

Splitting the bundle uncovered a second bug, the kind that only appears once you change how the file is born.

The indexers build databases in WAL mode, which is great for local write workloads and awkward for a static read-only artifact. A freshly extracted WAL database usually arrives with no `-shm` sidecar, and a plain `SQLITE_OPEN_READONLY` cannot open it without conjuring that shared-memory state first. So a shipped database could be byte-for-byte intact, pass an integrity check, and still refuse to open on a clean install.

`packages.db` made it impossible to ignore. On a fresh setup, package search would report the database as unopenable while the file sat right there, perfectly valid.

v1.3.0 fixes the artifact instead of teaching every reader a workaround. The release tool converts each bundled database to rollback journal mode before zipping, and rollback-mode databases open read-only with no `-shm` or `-wal` sidecar required. Then every query, read, and serve path opens through one shared helper, `SQLiteSupport.openReadOnly(at:)`, using

`SQLITE_OPEN_READONLY`. `SELECT` works. `INSERT`, `UPDATE`, `DELETE`, and any DDL fail with `SQLITE_READONLY`. The indexer and the setup migrator are the only writers; ordinary search and MCP reads physically cannot mutate the cache they read from.

That is the invariant I want under a local documentation tool. The thing you query is the thing you shipped, and nothing in the read path can quietly rewrite it.

I did not trust any of this without batteries

The frightening part of a refactor this wide is never the code you remember touching. It is the assumption asleep in a path you forgot existed.

So v1.3.0 leans on tests that behave more like release audits than unit checks.

`DatabaseBundleManifestTests` proves the bundle is derived from the production registry and excludes the retired `search.db`. `searchDBFlagRejectedEverywhere` proves the removed flag errors on every one of the six commands that used to take it. `ConvertToRollbackJournalTests` reproduces the WAL no-sidecar failure, then proves rollback conversion cures it. `SQLiteSupportReadOnlyTests` proves the shared helper can read and cannot write. `Issue1190PackageQueryReadOnlyOpenTests` proves a shipped rollback-mode `packages.db` answers a query the instant setup finishes.

Then come the snapshot batteries, which run the real CLI against the real shipped databases.

`ReadOnlyReadBatteryTests` drives the actual binary across all eight databases through the read-only path, searching every source and then performing at least twenty reads from each. The assertions check shape, not whether the output was simply long enough to look alive.

`ExhaustiveEnrichmentBatteryTests` covers all 24 enrichments from the project's enrichment inventory. Sixteen of them are proven end to end through the real CLI: lexical search, AST symbols, generic constraints, framework aliasing, conformances, inheritance, rank fusion, exact-title reranking, deployment floors for every `--min` platform, structured reads, and code examples. The other eight are internal columns the CLI never surfaces, so they are proven directly against the database with row-count probes rather than pretended through a command that cannot show them. The run writes the actual output into an HTML report, because a feature is not real until the artifact users install can demonstrate it.

That is the whole point of a battery. The local corpus is the product, so the product has to prove itself against the corpus, not against a mock that always agrees.

One more footgun, while I was in there

The refactor also exposed a package-fetch default that had quietly stopped making sense.

Before v1.3.0:

```
cupertino fetch --source packages
```

refreshed Swift Package Index metadata and star counts for 10,995 packages and *then* downloaded the curated archive closure the indexer actually consumes. The metadata refresh can take around four hours without a `GITHUB_TOKEN`. The archive stage took about 97 seconds for the 185-archive closure on a clean base. The four-hour stage was not the one doing the load-bearing work.

So the default flipped. Now `cupertino fetch --source packages` just downloads the archives. If you want the metadata refresh, you ask for it:

```
cupertino fetch --source packages --refresh-metadata
```

The old `--skip-metadata` flag is gone, because skipping metadata is no longer the exception. It is the default.

The boundary work continues

v1.3.0 also documents three public Swift packages factored out of cupertino: `SwiftMCPCore`, `SwiftMCPClient`, and `CupertinoDataKit`.

`CupertinoDataKit` is the one that belongs to this release's theme. It is the public read contract: the documentation and sample-code read protocols plus the value types they return. cupertino re-exports it through `SearchModels`, so the in-repo names stay put while the boundary becomes something real and external rather than implied.

It is the same move as the database split, applied to code instead of files. Name the boundary out loud. Make the system derive from it. Stop asking every caller to carry the old map in its head.

Upgrade

```
brew upgrade cupertino
cupertino setup
```

`setup` pulls the v1.3.0 per-source bundle. After that, search, read, and MCP serve all open the shipped databases read-only.

Here is the short version. Source independence has always been a code goal, and it is still a work in progress there. v1.3.0 is the release where it stops being only a code goal and starts being true of the artifact you actually install: eight databases, one registry, nothing in the read path that can rewrite them, and no unified `search.db` left to leak the old shape back in.

And the ninth database, whenever a new source earns one, is now a registry entry away instead of a refactor. That door is open now. I will walk through it soon.

That is why this one is The Big Refactor.