

Architecture

Microservices Roadmap (monolith → modular → services)

🕒 6 min read · Updated 13 October 2026 · Verified against [a661527](#) on 2026-09-26

How the stdlib monolith is factored into bounded contexts, and the spikes and gates on the road to services.

<https://docs.gurbanisoul.com/architecture/microservices-roadmap/>

`webapp/serve.py` is a single stdlib module today. The path to services is **incremental and proven by the golden vectors** — no behaviour changes as seams are extracted.

Step 1 — modular monolith (in-repo package `webapp/sggs/`)

Split the single dispatch ladder into modules behind an app-context object: `db.py` (connection factory, `_int`, `rows_to_list`, degradation convention), `search/` (waterfall + `romannorm`), `reader.py`, `analytics.py`, `timing.py`, `verify.py`, `static.py`, `http.py` (router). `serve.py` becomes a thin entrypoint. Extraction order by coupling (cleanest first):

`verify`, then `timing` (today's knowledge context), `analytics` (today's insights), `search`, and `reader` last.

Progress. 2026-09-24 — `api()` is a route table (`serve.ROUTES`): 26 handlers, each tagged with its bounded context (`reader`, `search`, `verify`, `insights`, `knowledge`). Behaviour proven identical (golden vectors in-process and over HTTP, a differential old-vs-new run, all three harnesses). 2026-09-24 — the handlers and their helpers live in bounded-context modules under `webapp/sggs/` (`core · search · reader · verification · insights · knowledge`). `core` owns the one DB handle and all shared mutable state; context modules reach it as `core.NAME` and never import `serve`. `serve.py` is the composition root (build identity, route table, HTTP layer) and proxies `serve.DB = ...` to `core`. Behaviour proven identical again (80/80 differential, seeded Hukam, HTTP contract against python `serve.py`, harnesses). 2026-09-24 — each context declares the tables it reads (`TABLES`; `serve.CONTEXT_TABLES`), enforced by an authorizer-scoped test over every golden suite and OpenAPI sample. Finding: no API route reads `fts_tri`. 2026-09-24 — `SGGS_MODULES` runs any subset of contexts from one image (default `all` = the monolith); `SGGS_DB` selects a database slice; split mode refuses to start on a slice missing a declared table; `/healthz` + `/readyz` (the Docker HEALTHCHECK now uses `/readyz`). Phase 2 (modular monolith) complete.

Step 2 — service split (later)

Each module becomes a service behind a gateway that keeps the **single /api/* origin** (so no CORS is introduced), each with a shared read-only DB image. The `contract/*.ndjson` vectors become cross-service tests (`tools/contract_http.py` replays them over HTTP against any running API); `contract/openapi.json` (generated by `tools/gen_openapi.py`) is the contract. Verify and Timing extract first (own tables, own graceful-degradation path).

Guardrails: `/api/meta` gains a per-module health block; every response stays same-origin; golden vectors + `webapp/tests/` prove byte-identical behaviour before and after each move.

Spike S3 — database slices (measured 2026-09-24, dataset `cb6775ff...`, full DB 108.4 MB)

`tools/slice_db.py --modules <ctx> --out build/slices/<ctx>.sqlite` (or make slices) cuts a service's database from the pinned one: the context's declared tables plus their closure (foreign-key targets, FTS5 shadow tables and external content tables), `VACUUMed`, every kept table proven equal in content to the full database and every kept FTS index passing integrity-check, installed atomically. Each slice was then served on its own (`SGGS_MODULES=<ctx> SGGS_DB=<slice>`):

Service	Tables	Slice	Ready (/readyz)	Key routes
search	24	74.0 MB (68 %)	90 ms	<code>/api/search</code> 10–200 ms, <code>/api/word</code> 8 ms
insights	17	65.7 MB (61 %)	91 ms	<code>/api/themes/network</code> 6 ms, <code>/api/analytics/*</code> ≤ 2 ms
reader	17	55.7 MB (51 %)	88 ms	<code>/api/ang/712</code> 22 ms, <code>/api/health</code> all-true 45 ms, <code>/api/random</code> 258 ms
verify	6	44.2 MB (41 %)	119 ms	<code>/api/verify</code> 28 ms
knowledge	7	0.7 MB (< 1 %)	94 ms	<code>/api/timing/clock</code> 1 ms

Findings: every slice boots in about a tenth of a second and serves its routes; routes of other contexts return 404; the reader slice passes every `/api/health` check (FTS5, Mool Mantar, 568

ੴ

, live verify). No service needs more than ~75 MB of data, so the smallest paid instance class is sufficient for each; the total slice footprint (240 MB) is about twice the single database because `lines` + `fts` are shared by four contexts. Measured on a MacBook Air (Apple silicon) with warm file caches — cold-start on the host is Spike S2's job.

Spike S1 — CDN caching of the proxied API (observed 2026-09-24 on production)

Vercel caches responses of the external `/api/*` rewrite by the API's own headers: `/api/ang/*` (`s-maxage=3600`) goes MISS then HIT, and `s-maxage` is stripped before the browser; `no-store` routes (`/api/search`, `/api/random`, `/api/health`) are never cached. Vercel's documentation does not promise that a new deployment clears cached external-rewrite responses, so both deploy workflows now purge the CDN cache right after the web goes live (`vercel cache purge --type cdn`, available in the pinned CLI), and the data canary checks through the CDN as well as at the origin.

Performance baseline (measured 2026-09-24, before the split)

`tools/perf_baseline.py` measures a fixed request mix per context against one origin and records p50/p95 (`docs/perf/baseline-2026-09.json`, 60 requests per route, 2 in flight, API origin, from Sydney): reader p95 568 ms, insights 524, search 453, knowledge 375, verify 365 — p50 about 300 ms everywhere, i.e. network round-trip dominates (server-side work is 1–200 ms, see Spike S3). After the split, `--compare docs/perf/baseline-2026-09.json` from the same vantage point fails if any context's p95 is more than 10 % worse.

Hosting: Vercel functions in the web project (ADR-0011)

The contexts first ran on staging as five free Render services. They now run as Python functions in the web's own Vercel project — one per context plus `all` — generated at deploy time by `tools/build_api_functions.py` and routed internally, so production can move with no new hosting (the web is already on Vercel Pro; functions are billed by use from the plan's credit). The trial on a separate Vercel project passed the whole golden contract (266 records at the time) on Vercel's Python 3.12 / SQLite 3.40.0, with every function ready at its commit and holding exactly its context; it also showed why the functions belong in the web project (one edge and one firewall instead of two, and no framework-less static output that would publish the databases).

The gateway (staging, Phase 4 step 3)

`frontend/vercel.json`'s rewrites are generated by `tools/gen_gateway.py` from `serve.ROUTES` (every route is tagged with its context, so a prefix can only go to the service that serves it) and from `gateway/routes.json`, which lists per environment the contexts that have their own service. A test fails if `vercel.json` drifts. Staging routes contexts one at a time — first knowledge (`/api/timing/*`, `/api/forms`, and their `/api/v1` twins) — and the `deploy-staging` verify job proves each routed context answers through the real gateway with its own X-Service. Production (v1.3.10, and 1.4.0) still sends everything to `all`; the change that routes all five contexts for production is held back (owner decision 2026-10-11), and they go live together in a later release, after the app is on the App Store ([services-production](#), step 3).

dataset_stats is not needed. The plan proposed a precomputed table so the reader slice could answer `/api/meta`'s cross-context counts. Measured on staging, the reader service's `/api/meta` is identical to the single API's (all 17 fields): its declared tables already cover everything `/meta` reports, which the authorizer test also enforces.

Sri Guru Granth Sahib Ji — Knowledge Base wiki. Generated from commit c3d9360. Updated 13 October 2026. Verified against a661527 on 2026-09-26.
This copy may be out of date; the page is at <https://docs.gurbanisoul.com/architecture/microservices-roadmap/>