ChatGPT

[Unverified] Discord Multi-Bot Orchestrator Rate Limit Manager Implementation Report and Task Plan

Feasibility and what this does not solve

What Discord and discord.js already do

Discord's REST API has both per-route and global rate limits, and Discord explicitly recommends *not* hard-coding limits; instead, clients should parse response headers and respond to 429s appropriately. [1]

On the client-library side, discord.js (via its REST layer) already implements rate-limit-aware queuing and exposes configuration aligned to Discord's documented global ceiling: `globalRequestsPerSecond` defaults to `50` as "the standard global limit used by Discord," and it provides `offset` to add headroom when timing resets, plus `rejectOnRateLimit` to control whether requests are queued vs. erroring. [2]

discord.js also surfaces observability hooks that matter for your goal: - REST events include `rateLimited`, `invalidRequestWarning`, and `response` exposed on the REST event interface. [3]
- `RateLimitData` emitted on `RESTEvents.RateLimited` includes fields that are helpful for coordination (bucket `hash`, `majorParameter`, `scope`, `retryAfter`, `timeToReset`, etc.). [4]
- If your bots are on v14, the guide documents that the REST rate limit event is named `rateLimited` and is available via `client.rest`. [5]

When an orchestrator is genuinely useful

A centralized (or semi-centralized) orchestrator can add value primarily in these scenarios:

- **Multi-process / multi-service sharing the same Discord token**: each process may maintain its own local view of rate limit buckets; coordination reduces combined burstiness. This is consistent with Discord's model being tied to the authenticated identity ("a bot token for a bot"). [1]
- **Reducing risk of IP-level lockouts from error storms**: Discord documents an "Invalid Request Limit" enforced against IPs: *10,000 invalid requests per 10 minutes*, where invalid includes 401/403/429 (with an exception: 429 with `X-RateLimit-Scope: shared` does not count). If you run multiple bots on the same homelab IP, a single bug-induced retry loop can affect everything. [1]
- **Centralized measurement and defensive controls**: discord.js can emit rate limit signals, but aggregating across many bot processes is still your job; a shared coordinator can provide uniform metrics, queue depth, and "circuit breaker" behaviors when the system is unhealthy. [6]

What problems this will not solve

A “token service” that merely grants permission to attempt requests does **not** inherently: - **Bypass Discord’s limits** (and should not). Discord’s design explicitly expects callers to follow headers and 429 retry signals. [1]
- **Eliminate all 429s**. Some rate-limit behaviors are difficult to predict before observing headers; Discord’s own docs emphasize adaptive parsing rather than static constants. [1]
- **Solve gateway (WebSocket) rate limits or reconnect storms**. Those are separate from REST rate limits; Discord’s developer guidance treats them separately. [7]

Assumptions to proceed (with explicit verification steps)

[Unverified] Assumption: your bots are primarily Node-based and use discord.js v14+ (or the `@discordjs/rest` layer directly).
Verification step: inventory each bot’s `package.json` / lockfile for discord.js / @discordjs/rest version and confirm whether `client.rest` is available for event instrumentation. [8]

[Unverified] Assumption: you care more about “avoid 429 storms / invalid-request ban” than about maximizing throughput to the theoretical limit.
Verification step: capture baseline metrics (429 rate, invalid request count, queue lengths) from discord.js REST events for 24–72 hours before enforcing new throttles. [6]

Discord rate limit model summary

Global vs per-route limits

Discord documents: - **Global rate limit**: “All bots can make up to **50 requests per second** to our API,” independent of per-route limits. [1]
- If there is **no Authorization header**, Discord applies the *global rate limit to the IP address*. [1]

This matters for a multi-bot homelab: the IP-based “invalid request” restriction can be the bigger blast radius than a single token’s global cap. [1]

What “per-route” means in practice

Discord describes: - Per-route limits may include the **HTTP method** (GET/POST/PUT/DELETE). [1]
- Per-route limits can be **shared across similar endpoints**, and Discord exposes `X-RateLimit-Bucket` “as a unique identifier” so clients can group shared limits as they discover them. [1]
- During calculation, per-route rate limits “often” account for **top-level resources** (major identifiers) in the path—currently channels (`channel_id`), guilds (`guild_id`), and webhooks (`webhook_id` or `webhook_id + webhook_token`). Different major IDs can yield independent calculations. [1]

A key implementation implication is that a stable “route key” is not just a string like `/channels/:id/messages`: major parameters matter, and the `X-RateLimit-Bucket` value is “non-inclusive of top-level resources,” so you need both a bucket identifier and a major parameter partition if you want to be precise. [1]

It is also publicly documented (via a `discord/discord-api-docs` issue) that the exact algorithm for "per-resource / shared" limits can be hard to infer without encountering 429 responses, and that implementers may hit ambiguity around what "top-level" means. This supports designing the MVP to be *adaptive and conservative* rather than "perfect out of the box." [9]

Headers and 429 handling

Discord's rate-limit headers and 429 payload model include: [1]
- `X-RateLimit-Limit`, `X-RateLimit-Remaining`
- `X-RateLimit-Reset` (epoch seconds) and `X-RateLimit-Reset-After` (seconds, can be decimals)
- `X-RateLimit-Bucket` (bucket identifier)
- On 429: `X-RateLimit-Global` if it's a global limit
- On 429: `X-RateLimit-Scope` with values: `user` (per bot/user), `global` (global per bot/user), `shared` (per resource)
- 429 response body: `retry_after` and `global` boolean. Discord recommends relying on `Retry-After` header or the `retry_after` field. [1]

[Inference] For defensive engineering, prefer using **the maximum of** `Retry-After` and body `retry_after` when both are present, because there is at least one documented case in the Discord API docs issue tracker where the header was reported incorrect while the body had a much larger `retry_after`. [10]

Invalid request limit and Cloudflare bans

Discord documents an IP-based "Invalid Request Limit":
- **10,000 invalid HTTP requests per 10 minutes**, applied to IP addresses.
- Invalid requests include responses with **401, 403, or 429** statuses, but **429 with** `X-RateLimit-Scope: shared` **does not count**. [1]

Discord's developer support article also highlights the same set of rate limit types (global, per-route, resource-specific, invalid request) and points to `X-RateLimit-Scope` as the identification mechanism. [7]

Architecture options ranked

Option C: rely on discord.js plus a local limiter only

Summary: Keep discord.js in full control. Add a local limiter (per bot process) for burst smoothing on specific high-traffic operations, plus observability by subscribing to REST events (`rateLimited`, `invalidRequestWarning`, `response`). [11]

Pros - Lowest complexity and lowest added latency (no extra network hop). - Uses discord.js's native understanding of `hash`, `majorParameter`, `scope`, and sublimits (where present). [4] - Aligns with Discord's guidance to parse headers rather than hard-code. [1]

Cons - No cross-process coordination: multiple services (or bots sharing an IP) can still collectively create an error storm, and you lose centralized queue depth unless you implement a separate telemetry aggregator. [12] - Cross-bot fairness is ad hoc.

Latent failure modes - If a bot has a retry-loop bug, it can still contribute to the IP-level invalid request limit affecting the entire homelab. [10]

Operational complexity: low.

Option B: sidecar per-bot + shared Redis

Summary: Each bot runs a local "rate limit sidecar" (could be a small local HTTP service or even an in-process module) that talks to shared Redis for distributed tokens and shared counters (IP-wide or group-wide).

Pros - Removes a single orchestrator bottleneck: if one sidecar dies, other bots continue. - Can keep "token check" local (loopback), reducing latency vs. a centralized service if you have bots on multiple hosts. - Shared Redis can still centralize counters and allow aggregated rate limiting (e.g., cap total 429/invalid counts across the homelab). Redis supports millisecond-resolution TTL and key expiration suitable for rolling windows. [13]

Cons - Harder to operate than Option C: more moving parts (N sidecars + Redis). - Still requires a careful atomic update strategy in Redis to avoid race conditions under load; Redis scripting can execute with atomic/blocking semantics to implement conditional updates. [14] - Harder to provide a single "queue depth" number unless you define it carefully (e.g., total waiting requests across sidecars).

Failure modes - Redis outage becomes the shared failure point; sidecars must fall back to local limiting.

Operational complexity: medium.

Option A: centralized token service over HTTP + Redis

Summary: A single Rust (axum) service exposes `/request_token` (and, for correctness, an observation endpoint) and uses Redis for shared/distributed counters with TTL.

Pros - Central policy point with uniform metrics, queue depth, and audit logs. - Simplifies client upgrades: bots call one service, no per-bot sidecar deployment. - Can implement server-side waiting (optional long-poll) to create a real queue whose depth is measurable.

Cons - Adds a network hop on every "permission check" unless you implement client-side caching or batch permits. - A centralized token service, if it does **not** observe real Discord response headers, can diverge from actual per-bucket behavior (Discord explicitly discourages hard-coded assumptions). [15] - Single point of failure (mitigated by the required "orchestrator-down fallback").

Failure modes - Orchestrator down: bots must fall back locally (your requirement). - Redis down: orchestrator either becomes "fail open" (risk) or "fail closed" (outage). A design choice must be explicit.

Operational complexity: medium (central service) to medium-high (if you add full adaptive bucket tracking).

Ranking recommendation

[Inference] For a 2-week evening MVP that aims to produce measurable benefit while minimizing correctness risk, the best ranking is:

C → A → B

- Start with **C** as the baseline that you can deploy quickly and that leverages discord.js's native rate limiter and telemetry surfaces. [16]
- If you can demonstrate cross-process bursts or IP-wide invalid-request risk, move to **A**, but keep the MVP "advisory + defensive" rather than pretending to perfectly model every Discord bucket. This aligns with Discord's own "don't hard-code" warning and the real-world ambiguity documented in the API issue tracker. [15]
- **B** is a good longer-term path if you plan multi-host deployment or want to avoid a centralized bottleneck, but it's more operational surface area than you need for an MVP.

Recommended MVP design

MVP scope statement

This MVP coordinates scheduling to reduce 429 storms and smooth bursts, without attempting to bypass Discord's limits (your non-goal). Discord's documented model expects clients to parse headers and handle 429 retry signals. [1]

Recommended architecture for the MVP

[Inference] Choose **Option A** (central token service) but implement it as a **two-signal system**:

- **Permit path**: `/request_token` decides when a bot is allowed to *attempt* a request (coarse gating + queue).
- **Observation path**: `/report_result` lets bots report back Discord rate-limit headers and 429 events to continuously calibrate Redis state.

This prevents the orchestrator from relying on fixed guesses in a world where Discord warns against hard-coded limits and where per-resource logic can be ambiguous until observed. [15]

API contract

`POST /request_token`

Request JSON

```
{
  "client_id": "bot-1",
  "group_id": "homelab-ip",
  "discord_identity": "sha256-of-token-or-app-id",
  "method": "POST",
  "route": "/channels/:channel_id/messages",
  "major_parameter": "123456789012345678",
  "priority": "normal",
  "max_wait_ms": 2000,
  "request_id": "uuid-v7"
}
```

Response JSON (granted immediately)

```
{
  "granted": true,
  "not_before_unix_ms": 1739325600123,
  "lease_id": "opaque",
  "reason": "ok"
}
```

Response JSON (not granted, client should wait and retry)

```
{
  "granted": false,
  "retry_after_ms": 150,
  "not_before_unix_ms": 1739325600273,
  "reason": "global_bucket_exhausted"
}
```

Optional behavior: if `max_wait_ms > 0`, the server may hold the request until `not_before_unix_ms` (server-side queue). That provides a real “queue depth” metric (see observability).

Notes: - `group_id` enables *IP-wide* coordination (e.g., cap invalid-request spikes) even when bots use different tokens; Discord’s invalid-request limit is IP-based. [1]
- `discord_identity` is a stable grouping key so you can enforce “per token” vs “per homelab” independently. Discord’s global rate limit is per bot/application identity. [17]

`POST /report_result`

Request JSON

```
{
  "request_id": "uuid-v7",
  "lease_id": "opaque",
  "discord_identity": "sha256-of-token-or-app-id",
  "method": "POST",
  "route": "/channels/:channel_id/messages",
  "major_parameter": "123456789012345678",
  "status_code": 429,
  "x_ratelimit_bucket": "abcd1234",
  "x_ratelimit_limit": 5,
  "x_ratelimit_remaining": 0,
  "x_ratelimit_reset_after_s": 1.234,
  "x_ratelimit_scope": "user",
  "retry_after_ms": 1234,
  "observed_at_unix_ms": 1739325600456
}
```

Response JSON

```
{ "ok": true }
```

Rationale: - Discord's docs explicitly encourage you to parse headers and rely on 429 retry signals. [1]
- discord.js surfaces a comparable semantic model (`hash`, `majorParameter`, `scope`, `retryAfter`) when rate limited; your client library can translate to this schema. [4]

Redis state model and TTL strategy (shared/distributed counters)

Key-space goals

- Support **two orthogonal enforcement scopes**:
- **Per Discord identity** ("token/app") global: target ~50 req/s, aligned to Discord's documented global cap. [18]

- **Per group_id (homelab IP)** invalid-request protection: cap "bad events" and implement a circuit breaker if 401/403/429 spike toward the IP ban threshold. [17]

- Support **observed bucket state**:

- `X-RateLimit-Bucket` is the canonical bucket identifier for shared per-route limits. [1]
- Bucket state is keyed by `(bucket_hash, major_parameter)` because Discord's bucket hash excludes top-level resources, and major parameters influence calculation. [19]

Proposed keys

1) **Global token-bucket (per Discord identity)** - `rl:global:{discord_identity}` → integer counter or token state
TTL: ~2–5 seconds (short), because global limit is "per second" and you mostly need short windows. [20]

2) **Coarse per-route limiter (safety net, not perfect)** - `rl:route:{discord_identity}:{method}:{route}`
TTL: window size (e.g., 1s) + small slack.

3) **Observed bucket mapping** - `rl:bucket_map:{method}:{route}` → last seen `x_ratelimit_bucket`
TTL: 24h (mirrors discord.js having a "hashLifetime" concept; discord.js defaults hash lifetime to 24h). [21]

4) **Observed bucket state** - `rl:bucket_state:{discord_identity}:{bucket_hash}:{major_parameter}` → a small JSON blob or hash fields such as: - `limit`, `remaining`, `reset_at_unix_ms`, `scope` TTL: until reset time + slack; Redis expiration supports millisecond precision, and expiry metadata is stored as absolute timestamps (so "time passes" even if Redis restarts). [22]

5) **Invalid-request guardrail (per group_id)** - `rl:invalid:{group_id}` → counter in a 10-minute rolling window
TTL: 10 minutes. This aligns with Discord's documented IP-level invalid request window. [17]

Atomic update strategy

Use Redis server-side scripting for "check + update" in one step (for example: compute whether a permit is available, and if so decrement / reserve and update expiry). Redis guarantees scripts' atomic/blocking execution semantics: while a script executes, "all server activities are blocked," and effects are observed as "either have yet to happen or had already happened." [14]

This is the simplest reliable tool for distributed token issuance without race conditions under contention.

Local caching strategy to reduce round trips

[Inference] Keep the MVP simple and safe; avoid large "leases" initially.

Recommended caching layers: - **Client-side micro-cache (per process)**: when `/request_token` returns `not_before_unix_ms` in the future, the client should sleep locally and avoid hammering the orchestrator with rapid retries. - **Single-flight per route key**: if multiple in-process tasks want the same route at once, let one task call `/request_token` and share the result. - **Optional batch permits (phase 2)**: extend `/request_token` to allow `permits: N` for the global bucket only, with a short TTL (e.g., ≤250ms) to reduce calls while keeping fairness. This stays aligned with Discord's 50 req/s global cap. [23]

Fallback behavior when Redis or orchestrator is unavailable

Orchestrator down (required behavior)

- Bot-side client switches to **local limiter** + discord.js built-in queuing.
- This is consistent with discord.js already implementing global request pacing and rate limit handling; the library default is explicitly aligned with 50 req/s global. [24]

Redis down

You have two viable stances:

- **Fail open (availability-first)**: orchestrator grants permits using an in-memory limiter only (per orchestrator instance).
- Risk: if you run multiple orchestrator instances, limits can be exceeded.
- **Fail closed (safety-first)**: orchestrator returns "not granted" with a short retry and asks clients to fall back locally.

[Inference] For a homelab single-instance orchestrator, "fail open with conservative in-memory limits" is practical *if* you also cap orchestrator concurrency and add client fallback.

Implementation task ledger and decision gates

Repo structure proposal (monorepo)

[Inference] A single repo makes iteration easier for a 2-week evening MVP:

- `orchestrator/` (Rust, axum service)
- HTTP API, Redis integration, metrics `/metrics`, health `/healthz`
- `client-js/` (Node package)
- wrapper module that bots import; integrates with discord.js REST events and/or REST `makeRequest` hook
- `sim/` (test harness)
- "bot simulators" generating burst patterns and collecting 429 metrics
- `deploy/`
- docker-compose, systemd units, example `.env`
- `docs/`
- API contract, Redis schema, runbook, dashboards

Task ledger

ID	Task	Acceptance criteria	Test method
ENG-001	Define API spec for `/request_token` and `/report_result`	Spec includes all fields needed for (a) per-token global, (b) per-route, (c) feedback loop; includes error semantics and time units	Spec review against Discord header model and discord.js RateLimitData fields [19]
ENG-002	Implement Redis schema + atomic permit issuance	Redis keys and TTL strategy documented; permit issuance uses atomic check+update consistent with Redis scripting semantics	Load test with concurrent permit requests; validate no double-grants beyond configured caps [25]
ENG-003	Implement orchestrator HTTP service (axum)	Exposes `/healthz`, `/metrics`, `/request_token`, `/report_result`; supports graceful shutdown	Run service and terminate; verify graceful shutdown behavior is wired using axum's graceful shutdown mechanism [26]
ENG-004	Implement metrics and queue depth	`/metrics` exposes counters and gauges listed in observability plan; queue depth is observable (in-flight or queued)	Scrape with Prometheus; verify correct `Content-Type` and stable scrape; validate queue depth changes under load [27]
ENG-005	Implement JS client: permit + fallback	If orchestrator reachable, calls `/request_token`; if not, falls back to local limiter without crashing	Kill orchestrator; run bot simulator; verify continued operation and recovery on orchestrator restart (required acceptance test)
ENG-006	Hook JS client to discord.js telemetry	Client subscribes to REST events and reports 429/limit headers to `/report_result`	Induce rate limiting in simulator; confirm `/report_result` receives events (validate via orchestrator logs + counters) [28]
ENG-007	Implement "invalid request guardrail"	Orchestrator detects spikes in 401/403/429 toward IP ban threshold and throttles / rejects new permits for a cooldown window	Simulator generates invalid requests; verify throttling triggers well before 10,000/10min threshold [17]

ID	Task	Acceptance criteria	Test method
ENG-008	Deployment artifacts for Pi	docker-compose and/or systemd unit(s) created; includes restart/backoff and health checks	Deploy on Pi; reboot; confirm services return and endpoints respond [29]
ENG-009	Write runbook + dashboards	Docs cover config, failure modes, and troubleshooting	Tabletop exercise: Redis down / orchestrator down / Discord 429 storm and confirm steps
ENG-010	Acceptance test harness	Harness can simulate 2–3 bots with burst patterns and generates summary metrics	Run acceptance tests below; store output artifacts

Acceptance tests (must include)

Burst simulation test (required)

Goal: simulate 2–3 bots sending burst traffic through the orchestrator and show: - orchestrator metrics, and - no sustained 429s.

Definition (explicit)
[Inference] Define “sustained 429s” as:
- more than **3** HTTP 429 responses with `X-RateLimit-Scope` in `{user, global}` within any **30-second** sliding window, per `discord_identity`, after the first 60 seconds of warm-up.

Rationale: - Discord distinguishes `scope` including `global`, `user`, and `shared`. [19]
- 429s can occur transiently during discovery; the goal is to avoid repeated 429 loops that risk the invalid-request threshold. [10]

Test method - Start orchestrator + Redis. - Run 2–3 simulators generating bursts targeting a small set of routes/major parameters. - Observe orchestrator `/metrics` counters for: - permits granted/denied - queue depth (if using server-side wait) - 429 observed by scope - Confirm the defined “sustained 429” threshold is not crossed.

Orchestrator-down test (required)

Test method - Start simulators with orchestrator enabled. - Stop orchestrator process. - Confirm simulators do not crash and continue (local limiter + discord.js). - Restart orchestrator. - Confirm clients reconnect and resume orchestrator-assisted mode.

Decision gates

Day-1 validation test

A practical Day-1 gate is “the orchestrator exists, is measurable, and a single bot can use it safely”:

- `/healthz` returns OK and stays OK under light load.
- `/metrics` is scrapeable and includes at least:
- `tokens_granted_total`
- `tokens_denied_total`
- `redis_latency_ms` (summary/histogram)
- `inflight_requests` or `queue_depth`
  (Prometheus targets should return valid `Content-Type` headers; Prometheus 3.0+ requires this.) [27]
- One bot (or simulator) can request permits and send `/report_result` events.

Week-1 go/no-go criteria

Go if all are true: - You can run 2–3 bots with orchestrator enabled and meet the “no sustained 429” definition above. - Redis outage does not cause cascading failures (bots continue with fallback). - Orchestrator outage does not crash bots and auto-recovers on return (required test). - You can demonstrate that metrics identify whether you are hitting `scope: global`, `scope: user`, or `scope: shared` limits (Discord + discord.js both surface this concept). [30]

No-go if: - The orchestrator meaningfully increases end-to-end latency for command handling (because every request blocks on permit acquisition) without a compensating reduction in rate limit incidents. - You must add so much “Discord-bucket emulation” logic that the MVP grows beyond “evening scale,” especially given documented ambiguity around per-resource behaviors. [15]

Observability, security, and deployment on Pi

Observability plan

Metrics to expose

Expose Prometheus-format metrics at `/metrics`. Prometheus requires correct exposition format handling; Prometheus 3.0+ requires a valid `Content-Type` header for successful scrapes. [27]

Recommended core metrics (minimum viable): - `orchestrator_request_token_total{outcome="granted|denied|error"}` - `orchestrator_request_token_wait_ms_bucket` (histogram) and/or summary - `orchestrator_queue_depth` (gauge): number of server-side waiting requests (if you implement waiting) or in-flight permit requests - `orchestrator_429_observed_total{scope="global|user|shared"}` (from `/report_result`)
- Scope definitions align with Discord’s `X-RateLimit-Scope` and discord.js RateLimitData `scope`. [19]
- `orchestrator_invalid_requests_total{status="401|403|429"}` (from `/report_result` /

discord.js warnings) - `redis_roundtrip_ms_bucket` (histogram) - `redis_errors_total` - `orchestrator_fallback_events_total{reason="orchestrator_down|redis_down"}` (emitted by clients if possible)

Log schema and tracing correlation

Use structured JSON logs with: - `timestamp` (UTC ISO8601) - `level` - `service` - `event` - `request_id` (correlation ID carried end-to-end: client → orchestrator → report)
- `route`, `method`, `status_code`, `duration_ms` - `discord_identity` (hashed) and `group_id`

On the Rust side, request ID propagation can be implemented using tower-http request-id utilities (supports setting and propagating request IDs, including `x-request-id`). [31]

Security and safety

Network binding

- Default bind to **loopback only** (e.g., `127.0.0.1`) if all bots run on the same Pi.
- If you must serve the LAN, bind to the LAN interface and firewall the port to your bot hosts only.

This aligns with the OpenMetrics/OpenMetrics 2.0 security guidance that authentication/authorization is often handled "outside of OpenMetrics," and operators should use TLS and/or firewalling/ACLs where feasible. [32]

Auth options (LAN threat model)

- **Shared secret**: simplest (e.g., `Authorization: Bearer <secret>`).
- **mTLS**: strongest, but more operational effort (cert distribution/rotation); likely beyond a 2-week evening MVP.

Abuse scenarios and mitigation: - **Token starvation/DoS**: a compromised device spams `/request_token` to starve bots.
Mitigation: per-client rate limit on orchestrator endpoints + authentication + bind LAN-only.
- **Data disclosure**: don't transmit Discord bot tokens; use a hash-based `discord_identity` label so the orchestrator cannot impersonate bots. - **Retry loop amplification**: if clients retry too frequently after a denial, the orchestrator becomes the hotspot.
Mitigation: enforce minimum retry delay in responses and implement client-side exponential backoff.

Deployment plan on Raspberry Pi

Docker Compose vs native install

Docker Compose - Pros: reproducible packaging (orchestrator + Redis), easy restarts, easy upgrades. - For dependency readiness, Compose supports waiting for a dependency to be "healthy" using `depends_on` long syntax with `condition: service_healthy` (defined via `healthcheck`). [33]

Native + systemd - Pros: lowest runtime overhead on Pi, no container layering, straightforward journald logging. - systemd supports restart policies (`Restart=on-failure` recommended for long-running services) and start-rate limiting via `StartLimitIntervalSec` / `StartLimitBurst`. [34]

[Inference] For a Pi homelab, "native + systemd for orchestrator" plus "Redis either native or container" is often the lowest-footprint approach, but Compose is faster to iterate.

systemd unit considerations

- Use `Restart=on-failure` and `RestartSec` for backoff; systemd's service docs note restart is subject to start rate limiting (`StartLimitIntervalSec=` and `StartLimitBurst=`). [34]
- Use an `EnvironmentFile=` and keep secrets out of the repo.

Health checks

Provide: - `GET /healthz` (liveness/readiness): returns OK only if: - HTTP server is running, and - Redis connectivity is OK (if Redis is required for non-fallback mode)

For Compose, connect a container healthcheck to `/healthz`, enabling dependency ordering via `service_healthy`. [33]

For systemd, use a watchdog strategy or a separate cron/monit check (optional).

Graceful shutdown

For axum, use `with_graceful_shutdown` to stop accepting new requests and allow in-flight work to finish when a shutdown signal completes. [35]
For signal handling, Tokio provides a portable `ctrl_c()` future and a signal module for async signal handling. [36]

Non-blocking questions

1) Do any of your bots share the **same bot token** across multiple processes/services, or is it strictly "one token per process"?
2) Can you commit to a single "group_id" meaning (e.g., "homelab IP") or do you need multiple groups (per token + per homelab)?
3) Should `/request_token` be **blocking** (server-side queue) or **non-blocking** (returns retry time and clients sleep)?
4) What is your preferred metrics stack on the Pi (Prometheus scrape only, or do you also run Grafana)? [27]
5) Are any bots deployed off-Pi (other LAN nodes) such that the orchestrator must bind to LAN rather than loopback?

---

[1] [10] [12] [15] [17] [18] [19] [20] [23] [30] Rate Limits - Documentation - Discord
https://discord.com/developers/docs/topics/rate-limits

2 16 21 24 RESTOptions (rest - main) | discord.js
https://discord.js.org/docs/packages/rest/main/RESTOptions%3AInterface

3 6 11 28 RestEvents (discord.js - 14.19.2) | discord.js
https://discord.js.org/docs/packages/discord.js/14.19.2/RestEvents%3AInterface

4 RateLimitData (rest - main) | discord.js
https://discord.js.org/docs/packages/rest/main/RateLimitData%3AInterface

5 8 Updating to v14 | discord.js
https://discordjs.guide/additional-info/changes-in-v14

7 My Bot is Being Rate Limited! – Developers
https://support-dev.discord.com/hc/en-us/articles/6223003921559-My-Bot-is-Being-Rate-Limited

9 clarify per-resource rate limit algorithm · Issue #5557 · discord/discord-api-docs · GitHub
https://github.com/discord/discord-api-docs/issues/5557

13 Keys and values | Docs
https://redis.io/docs/latest/develop/using-commands/keyspace/?utm_source=chatgpt.com

14 25 Scripting with Lua | Docs
https://redis.io/docs/latest/develop/programmability/eval-intro/?utm_source=chatgpt.com

22 EXPIRE | Docs
https://redis.io/docs/latest/commands/expire/?utm_source=chatgpt.com

26 35 Serve in axum::serve - Rust
https://docs.rs/axum/latest/axum/serve/struct.Serve.html?utm_source=chatgpt.com

27 Exposition formats | Prometheus
https://prometheus.io/docs/instrumenting/exposition_formats/?utm_source=chatgpt.com

29 33 Control startup order | Docker Docs
https://docs.docker.com/compose/how-tos/startup-order/?utm_source=chatgpt.com

31 TraceLayer in tower_http::trace - Rust
https://docs.rs/tower-http/latest/tower_http/trace/struct.TraceLayer.html?utm_source=chatgpt.com

32 OpenMetrics 2.0 | Prometheus
https://prometheus.io/docs/specs/om/open_metrics_spec_2_0/?utm_source=chatgpt.com

34 systemd.service
https://www.freedesktop.org/software/systemd/man/253/systemd.service.html?utm_source=chatgpt.com

36 tokio::signal - Rust
https://docs.rs/tokio/latest/tokio/signal/?utm_source=chatgpt.com