resilix — Architecture & Version Plan

A C4 walkthrough of what we are building, why each piece exists, and what ships in each version.

Package `resilix` — npm name verified available 10 Aug 2026 **Repo** github.com/lintdeveloper/resilix · MIT

Author Musa Musa **Date** 10 Aug 2026 **Status** Design locked, pre-implementation

1 · How to read this document

The goal of this document is that **you can explain every box, every formula, and every default to another engineer without looking anything up**. It is deliberately not a scaffold plan. Each version section is ordered the same way:

SUB-SECTION	WHAT IT ANSWERS
Goal	Why this version exists at all. If we cannot say what breaks without it, it should not ship.
What you must understand first	The concepts to internalise *before* writing code. This is the anti-bootstrap section.
C4 delta	Which boxes appear or change at each C4 level.
Algorithms & defaults	Exact formulas and exact numbers, with their provenance.
Data structures	Representation, complexity, memory. This is where most breaker libraries are sloppy.
Public API	The surface we commit to. Everything else stays internal.
Tests	How we prove it. Determinism strategy included.
Done when	Objective exit criteria.

A note on applying C4 to a library

C4 was designed for systems, not packages. The mapping we use, and it is a deliberate choice:

C4 LEVEL	NORMAL MEANING	OUR MEANING
L1 Context	System + users + external systems	The *consumer's service* with our library inside it, and the upstreams/observability it touches
L2 Container	Deployable units	Published packages and subpath exports — the units a consumer `imports`, and the dependency rules between them
L3 Component	Major code groupings	The internals of the core package: executor, classifier, policies, registry, clock
L4 Code	Classes / state machines	State machines, ring buffers, control loops — the parts where correctness actually lives

> **WHY THIS MATTERS**
>
> L2 is the level that keeps the project shippable. The rule *"core has zero runtime dependencies and does no I/O"* is a **container-level constraint**. Every feature that needs a dependency or an API surface outside the core must land in its own container. That single rule is what lets the library grow to a large feature set without becoming unmaintainable, and it is why the version plan below can be paused at any point without leaving the package in a broken shape.

Contents

2 · The thesis

One sentence: **JavaScript's load limiting is locked inside two RPC clients.**

Hedging, retry budgets and adaptive throttling all ship in JavaScript today at roughly 90M downloads a week — welded to `@grpc/grpc-js` and the AWS SDK, usable only for gRPC calls and AWS API calls respectively. There is no general-purpose implementation, and adaptive *concurrency* limiting, slow-call circuit breaking and criticality do not exist anywhere. §2.3 sets out that prior art precisely, because it is the first thing a knowledgeable reader will raise.

Resilience policies split into three families. The split is failsafe-go's and it is the best conceptual frame found in the research, so the library and its documentation are organised around it.

```
PROACTIVE   static configuration; must be tuned to a capacity you guess in advance
             bulkhead · rate limiter · timeout
             JS status: WELL SERVED   p-limit 318M/wk · bottleneck 13M/wk · cockatiel

REACTIVE    detects real overload from live signals; needs NO capacity configuration
             adaptive limiter · adaptive throttler · time-based circuit breaker
             JS status: PROTOCOL-LOCKED   throttling exists only inside the AWS SDK;
                                          hedging + budgets only inside grpc-js.
                                          adaptive CONCURRENCY exists nowhere.   <-- the package

INBOUND     protects your own process rather than your upstream
             system-adaptive protection (load1 / CPU / RT / concurrency)
             JS status: EMPTY        <-- v2.0 territory
```

Evidence that the space is open

FINDING	DETAIL
cockatiel is feature-frozen *by declaration*	Issue #119, 26 Feb 2026, maintainer: “I generally consider this project pretty *complete* and don't plan to take on more maintainers.” Polly-v8 parity request #80 open since Nov 2023. Bus factor 1 (96 commits vs next human contributor's 4).
opossum's backlog is stale-bot rot	#817 count-based window, #818 per-error-type thresholds, #819 half-open herd, #781 serverless — all open since 2023, all answered with “send a PR” then repeated “this issue is stale”.
Effect declined a circuit breaker	26.7M downloads/wk, 15.2k stars. Issue #2843 had 12 reactions; closed by a maintainer as “a stale feature request… not enough recent interest”. **This makes Effect an integration target, not a competitor.**
The only TS adaptive limiter is unusable	`@zingage/adaptive-concurrency` 0.13.0: **no license field, no repository field**, one version, never updated, 2 runtime deps. Cannot legally be adopted or forked in most organisations.
Reactive limiting has proven demand elsewhere	Alibaba Sentinel: 23,141 stars — more than Polly (14,221) or resilience4j (10,738). No JS port exists.

2.2 · Who this is for, concretely

The audience is anyone calling a third-party API, a database, or a queue from JavaScript — which is to say anyone not exclusively calling gRPC or AWS. The headline use case is **LLM and inference APIs**, because they exercise every one of our differentiators at once and because every team has one now.

SYMPTOM TEAMS ACTUALLY HIT	POLICY THAT ANSWERS IT	VER
`429` + `Retry-After` from a model provider, continuously	`overload` verdict → adaptive throttler; retry honours `Retry-After`	0.4
p99 is 10× p50 on the same model, with a flat error rate	slow-call breaker, then the adaptive limiter	0.1 → 0.3
streaming completions hold a connection for 60 s	**concurrency** limiting, not rate limiting — Little's Law	0.3
`400` content-policy rejection	`answered` — never retried, never trips the breaker	0.1
one tenant burns the whole organisation's quota	per-user fairness (`UsageTracker`)	0.5
background summarisation competing with live chat	criticality / prioritisation	0.5
Prisma or pg pool exhausted by a burst	bulkhead + queueing	0.1 → 0.3
payment provider times out; did it charge?	hedging is **unsafe** here — a documented refusal, not a feature	0.5
webhook retries amplify 3× during a partner outage	retry budget	0.4

LLM APIs are the ideal motivating example for one specific reason: **"up but slow at a flat error rate" is their normal operating mode**, and streaming makes in-flight concurrency — not request rate — the binding constraint. That is exactly the gap the AWS rate limiter does not close (§2.3).

2.3 · Prior art in JavaScript — stated plainly

Three of these patterns already exist in JS, at a scale that dwarfs the general-purpose libraries. Verified by reading the shipped source, not the documentation.

PATTERN	WHERE	SCOPE	DL/WK
Hedging	`@grpc/grpc-js` → `retrying-call.ts`: `hedgingPolicy`, `hedgingTimer`, `hedgingDelay`, `nonFatalStatusCodes`	gRPC calls only, configured via service-config JSON	48.9M
Retry budget	same file → `class RetryThrottler`: `maxTokens`, `tokenRatio`, `canRetryCall() = tokens > maxTokens/2`	gRPC calls only	48.9M
Adaptive throttling	`@smithy/core/retry` → `DefaultRateLimiter`, a CUBIC controller	AWS API calls only	41.8M

AWS DefaultRateLimiter — this is not a toy:

```
beta = 0.7   minCapacity = 1   minFillRate = 0.5   scaleConstant = 0.4   smooth = 0.8

cubicThrottle(rate) = rate * beta
cubicSuccess(t)     = scaleConstant * (t - lastThrottleTime - timeWindow)^3 + lastMaxRate
timeWindow          = ((lastMaxRate * (1 - beta)) / scaleConstant) ^ (1/3)
newRate             = min(calculatedRate, 2 * measuredTxRate)
```

What still holds after accounting for it

CLAIM	STATUS
Adaptive **concurrency** limiting — bounding in-flight count from a **latency gradient**	**Holds.** AWS controls *rate* (req/sec) from *throttling errors*. Netflix and Envoy control *concurrency* from *latency*. Different control variable, different signal — that distinction is Little's Law, and it is precisely why Netflix built `concurrency-limits` rather than using a rate limiter
Slow-call-rate circuit breaking	**Holds.** Nothing in JS
Criticality / prioritisation / per-user fairness under load	**Holds.** Nothing found
Any of it as a general-purpose, composable policy	**Holds — and this is the thesis.** You cannot apply grpc-js hedging to a Postgres query, or the AWS limiter to any non-AWS HTTP call

READ THIS AS VALIDATION, NOT REFUTATION

The two most-downloaded RPC clients in the ecosystem each independently decided these patterns were necessary enough to build from scratch. Together that is **~90.7M downloads/week of private, protocol-locked implementations against 3.2M/week of general-purpose ones** (cockatiel 2.0M + opossum 1.2M) — a factor of about 28. The patterns are proven load-bearing in JavaScript at nine figures; they are simply unavailable to anyone outside those two protocols.

THE RISK THIS EXPOSES — NAMED RATHER THAN HIDDEN

If these patterns are so necessary, why did both teams build in-house instead of taking a dependency? Three reasons, and two are permanent: **(1)** SDKs structurally refuse third-party dependencies — those teams are not addressable, ever; **(2)** both needed protocol-specific integration (gRPC status codes and service config; AWS throttling-error classification) — also permanent; **(3)** nothing general-purpose existed when they built it — the only one we can act on. So the market is not AWS or gRPC teams. It is everyone calling a third-party HTTP API, a database, or a queue, who today has a circuit breaker and nothing else.

2.4 · Corroborating production evidence

OPTIONAL, NOT LOAD-BEARING

We separately hold first-hand production evidence for the exact failure mode this policy family exists to handle. A national identity provider degraded to **p50 10.4 s / p95 15.3 s against a healthy baseline of p50 0.35–0.40 s / p95 0.8–1.0 s — roughly 25–30× slower with zero additional errors**. A failure-rate breaker is blind to that. Separately, **13–18% of calls returned 4xx on a perfectly healthy day**, so any breaker whose failure predicate is "the promise rejected" trips on ordinary bad user input. Both facts drive core design decisions (§15 ADR-3, ADR-6).

Because §2.2 carries the argument on its own, this data is **corroboration rather than the foundation** — so publishing it stays an independent decision, and the provider is referred to generically throughout.

3 · L1 · System context

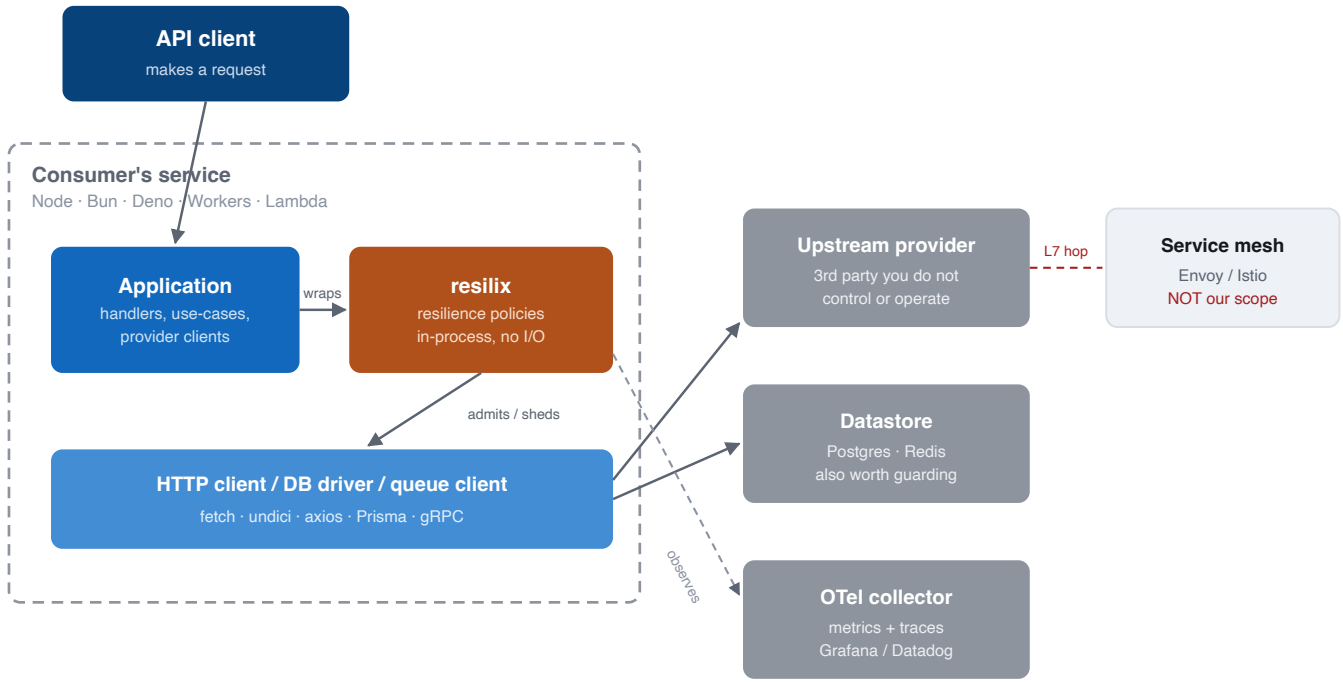


Figure 1 — L1 System Context. The library is entirely in-process and synchronous with the call it guards. It never opens a socket of its own. The mesh box is drawn only to mark it out of scope: cross-instance ejection is the mesh's job, which is why we do not build distributed breaker state (§15 ADR-2).

What the context diagram commits us to

COMMITMENT	CONSEQUENCE
In-process only	Decisions cost microseconds and never add a network hop to the failure path. A policy decision cannot itself fail.
Transport-agnostic	The core cannot import `axios`, `fetch` types, or a Node module. Guarding a Prisma query must be as natural as guarding an HTTP call.
Runtime-agnostic	No `setInterval` at module scope, no eager `AbortController`, no `process`. This is what breaks cockatiel on Cloudflare Workers (their #105, declined).
Observability is one-way	Telemetry is a fire-and-forget observer. A broken exporter must never affect an admission decision.

4 · L2 · Containers

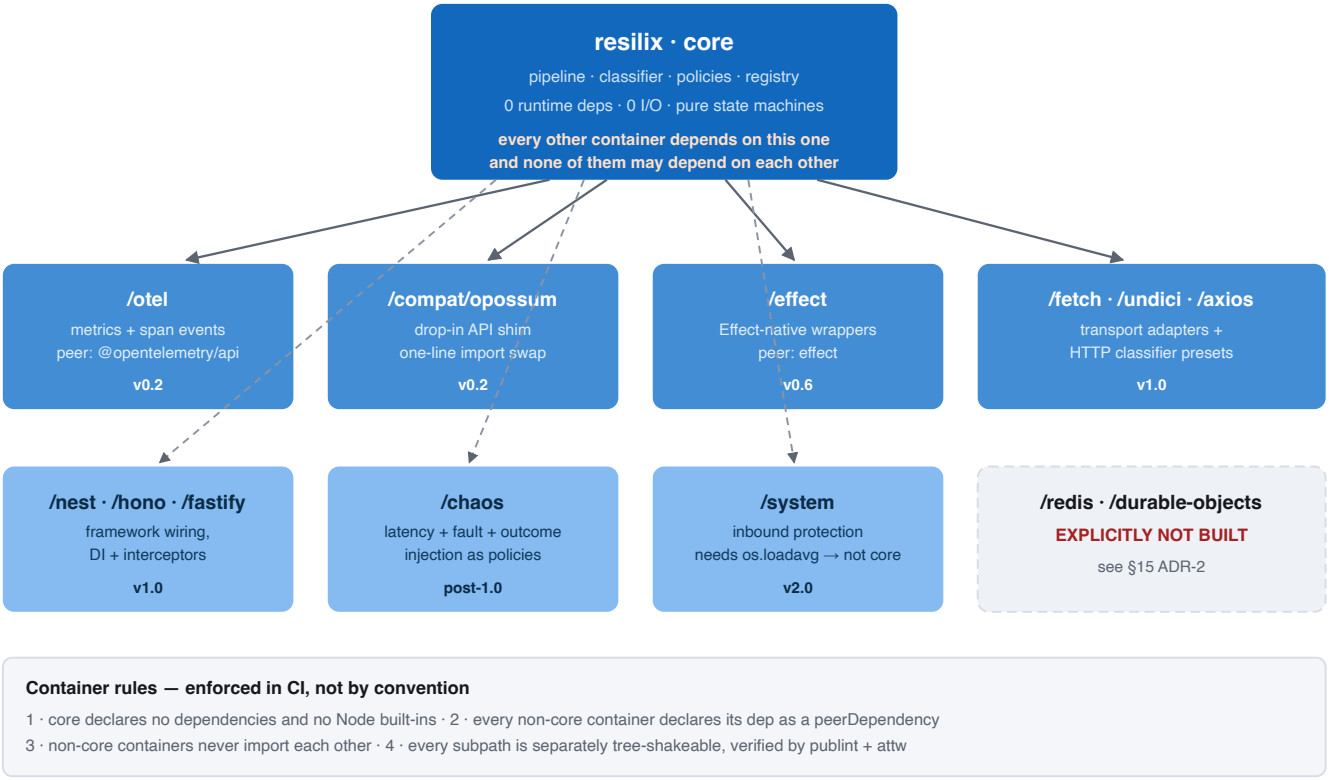


Figure 2 — L2 Containers. One published package, many subpath exports. Solid arrows are "depends on core"; dashed arrows are the same relationship for later versions. This is the `naija-id` pattern (`.` / `./zod` / `./standard` / `./redact`) applied at a larger scale — the mechanism that lets a broad feature set keep a zero-dependency core.

Why one package with subpaths rather than a monorepo of packages

CONCERN	SUBPATH EXPORTS	SEPARATE PACKAGES
Version skew between core and adapter	impossible	a permanent support cost
Release process	one changeset, one tag	coordinated multi-publish
Install footprint	tree-shaken; unused subpaths cost nothing	smaller nominally
Peer dep hygiene	optional peers, per subpath	cleaner in theory
Discoverability	one README, one name	fragmented

Decision: subpaths. Revisit only if a single adapter grows its own release cadence.

5 · L3 · Components of the core

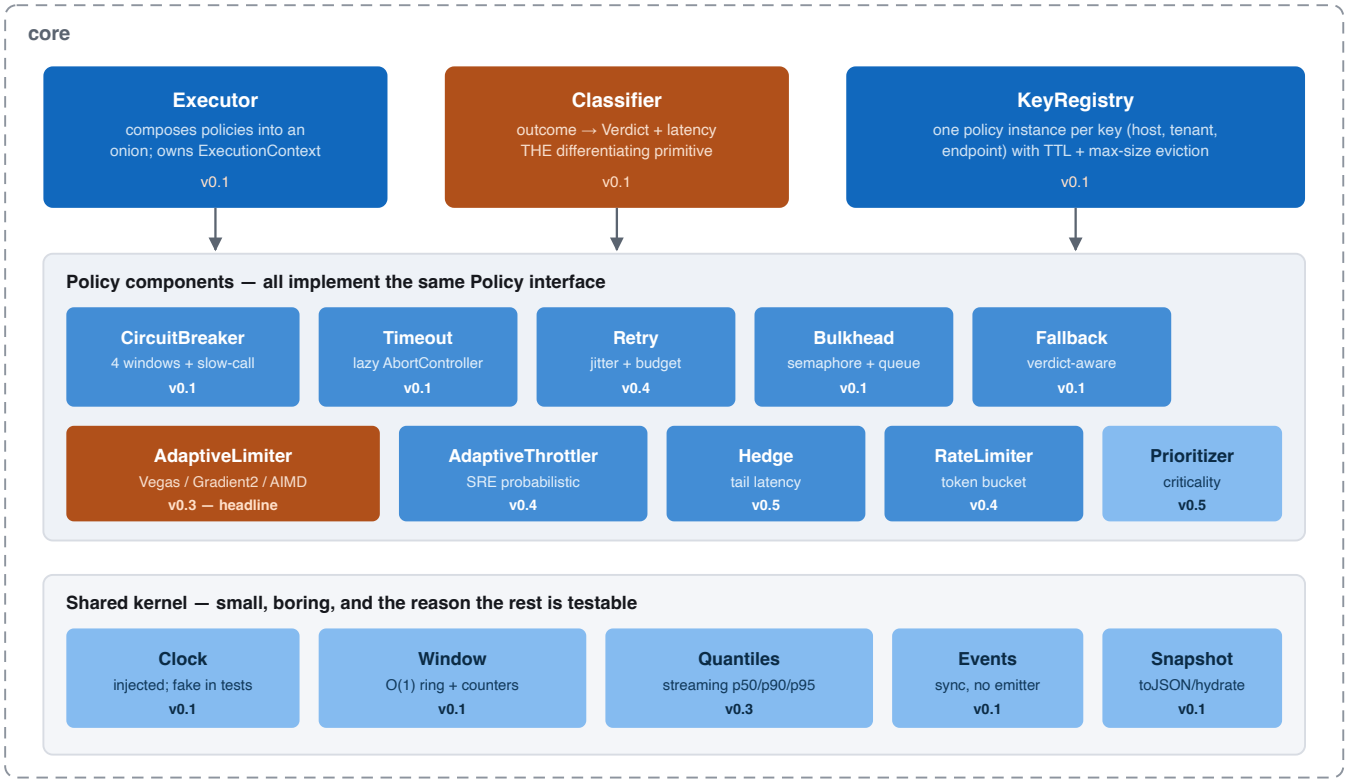


Figure 3 — L3 Components. Orange marks the two components that are genuinely novel in the JS ecosystem: the **Classifier** and the **AdaptiveLimiter**. Everything blue exists in some form in cockatiel or opossum; those are table stakes, not the reason the project exists.

The two interfaces everything hangs off

```
// Every policy is this shape. Nothing else. If a feature does not fit,
// it is either a Classifier concern or it belongs in a subpath container.
interface Policy<S = unknown> {
  readonly name: string;

  // Gate: may this execution proceed right now?
  admit(ctx: ExecutionContext): Admission;   // { ok: true } | { ok: false, reason, retryAfterMs? }

  // Feedback: an execution that we admitted has settled.
  settle(obs: Observation, ctx: ExecutionContext): void;

  // Serialisable state, for serverless hydration and for tests.
  snapshot(): S;
  hydrate(s: S): void;
}

interface Observation {
  verdict:   Verdict;     // classified outcome — not a boolean
  latencyMs: number;      // always present; the signal breakers throw away
  at:        number;      // clock.now(), never Date.now() inside a policy
}
```

WHY `ADMIT` / `SETTLE` RATHER THAN `EXECUTE(FN)`

opossum and cockatiel both own the invocation: you hand them a function. That couples the policy to the call mechanics and makes composition the library's problem rather than yours. Splitting the gate from the feedback means (a) policies are pure state machines with no async surface at all, (b) they are trivially unit-testable with no promises involved, (c) the Executor is the *only* component that knows about `async`, and (d) a consumer who wants to drive the state machine by hand — from a Kafka consumer, a cron job, a stream — simply can. The ergonomic `execute(fn)` form still exists; it is roughly thirty lines on top of this.

6 · L4 · The execution model

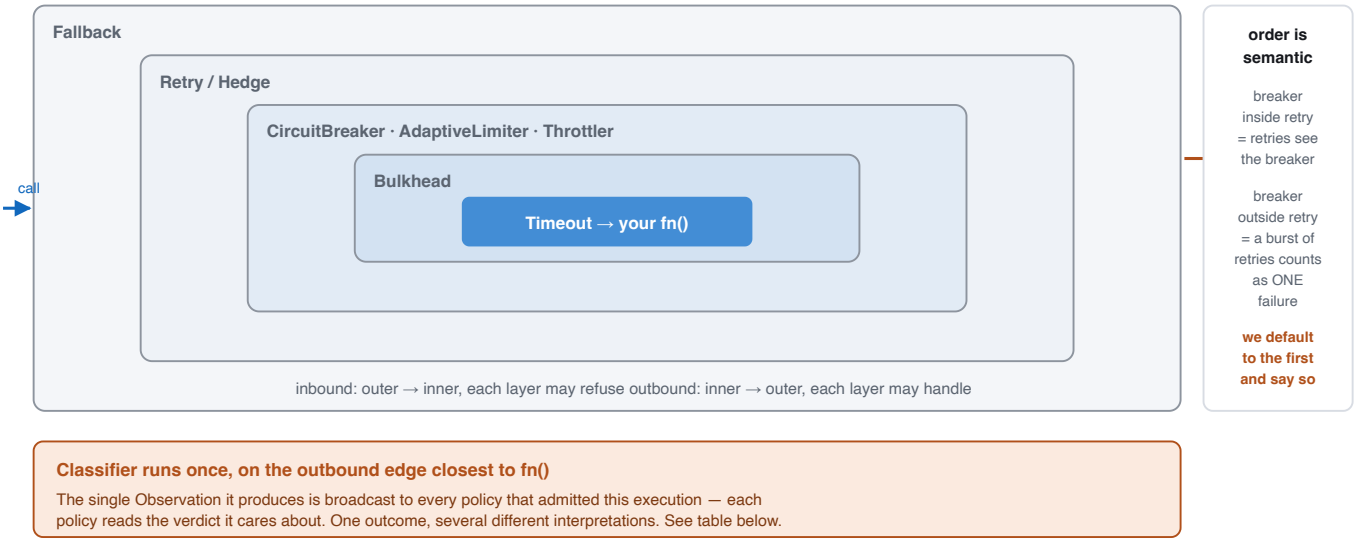


Figure 4 — L4 Execution model. Policies compose left-to-right into an onion, outermost first — the same model as failsafe-go (`Fallback(Retry(Breaker(Timeout(fn))))`) and cockatiel's `wrap()`. Composition is not our differentiator; it is the well-understood part, so we adopt the convention rather than invent one.

The classifier, concretely

This is the part to understand most deeply, because every other component reads its output.

```
type Verdict =
  | 'success'     // healthy completion
  | 'answered'    // 4xx — the upstream worked; the CALLER was wrong
  | 'transient'   // 5xx, ECONNRESET, socket hang up, unlabelled transport error
  | 'overload'    // 429, 503 + Retry-After — upstream is explicitly shedding
  | 'timeout'     // our deadline, not theirs
  | 'rejected';   // WE shed it. never counts as upstream evidence.
```

One settled call, read differently by each policy. This table is the design:

OUTCOME	VERDICT	BREAKER	ADAPTIVE LIMITER	RETRY	THROTTLER
200 in 120 ms	`success`	healthy sample	latency sample, may raise limit	done	accept
200 in 9 s	`success`	**counts to slow-rate**	**strong overload signal**	done	accept
404 invalid record	`answered`	**healthy**	latency sample only	**never retry**	accept
422 restricted	`answered`	**healthy**	latency sample only	**never retry**	accept
429	`overload`	**healthy**	**reduce limit**	retry after delay	**reject-ward**
500	`transient`	failure	reduce limit	retry	reject-ward
ECONNRESET / no status	`transient`	failure	reduce limit	retry	reject-ward
our 15 s deadline hit	`timeout`	failure	**strong overload signal**	retry once	reject-ward
we refused it	`rejected`	**ignored**	**ignored**	no retry	**ignored**

THE TWO ROWS THAT JUSTIFY THE WHOLE DESIGN

Row 3–4 (`answered`). With 13–18% 4xx on a healthy day, treating a rejected promise as a failure means a customer submitting bad input can open your circuit. cockatiel's `handleWhen` is a boolean predicate, so you can only choose “failure” or “invisible” — you cannot say “not a breaker failure, but still a load signal”, which is exactly what `overload` needs to be. opossum #818 is three years of people asking for this.

Row 9 (`rejected`). Our own shedding must never feed back as upstream evidence. Without this verdict a breaker that opens observes its own rejections and stays open — a self-sustaining outage. cockatiel #115, “Idle wrapped policy fires onFailure when circuit breaker opens”, is this bug.

7 · v0.1 — Pipeline, classifier, breaker, timeout PARITY BEACHHEAD

Establish the primitives everything later depends on, and be immediately better than both incumbents on the breaker itself.

SHIPS	Executor · Classifier · CircuitBreaker · Timeout · Bulkhead · Fallback · KeyRegistry · Clock · Window · Snapshot
SIZE	~1,100–1,400 LOC source, ~2,500 LOC tests
DEPS	zero
BEATS	slow-call tripping, dual-bound window, consecutive backstop, single-probe half-open, `rejected` verdict — none present in cockatiel or opossum

What you must understand first

1. **Why a rate-based breaker alone is not enough.** At 8 req/min a "last 100 calls" window spans ~12 minutes, so the decision reflects the upstream twelve minutes ago. At 100 req/min the same window spans one minute. A window bounded by *only* count is stale at low traffic; bounded by *only* time it is unbounded at high traffic. This is the substance of opossum #817.
2. **Why a minimum-sample rule creates a hole.** Every rate breaker refuses to evaluate below a minimum sample (cockatiel's `minimumRps` / `minimumNumberOfCalls`, resilience4j's `minimumNumberOfCalls`). A completely dead host at 8 req/min never reaches that minimum inside a 5-minute window, so it **never trips** and every caller pays the full timeout. The fix is a window-independent consecutive-failure backstop.
3. **Why half-open must admit exactly one call.** On transition, an unguarded half-open admits every waiting caller at once — a thundering herd onto a provider that is by definition fragile. And the faster your reset timeout, the more often you do it. opossum #819 is this discussion.
4. **Why the window must be O(1) per call.** Filtering an array on every request is O(n). At n=100 it is invisible; as a library default with a configurable window it is a latency bug in the hot path of every guarded call.

Breaker state machine

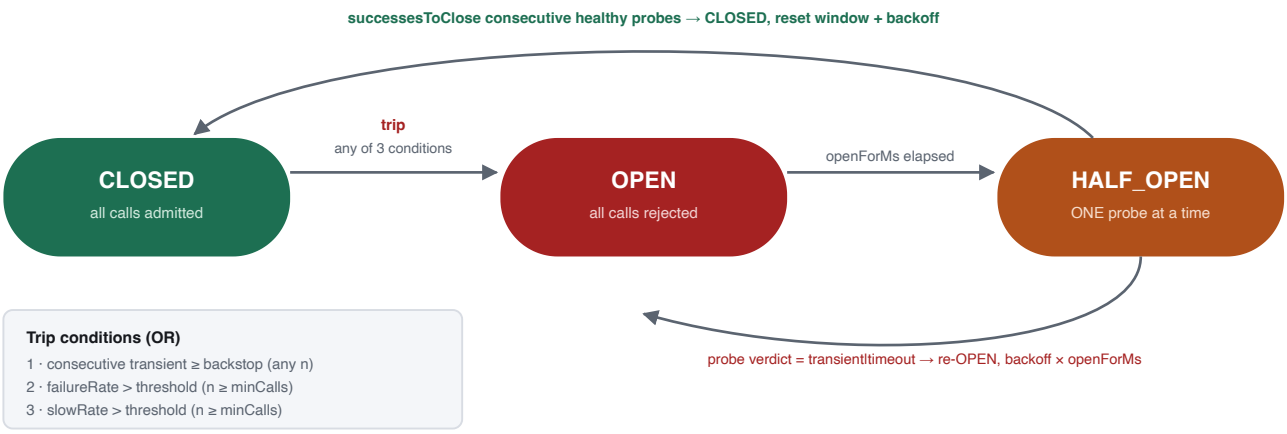


Figure 5 — Breaker FSM. Two additions over every JS implementation: condition 3 (slow-rate, from resilience4j) and condition 1 (the window-independent backstop, from our ADR-015). Verdict `rejected` never enters the window at all.

Window: the data structure

```
// Dual-bound ring buffer with running counters. O(1) amortised per call.
class Window {
  private ts:   Float64Array;   // timestamps, capacity = maxCalls
  private dur:  Float64Array;   // latency per sample
  private vd:   Uint8Array;     // verdict enum, 1 byte
  private head = 0; private tail = 0; private size = 0;

  // running counters — this is why evaluation is O(1) rather than O(n)
  private failures = 0;
  private slow     = 0;

  push(v: Verdict, latencyMs: number, now: number) {
    this.evictOlderThan(now - this.maxAgeMs);      // amortised O(1)
    if (this.size === this.capacity) this.dropTail();
    // write at head; increment counters if this sample is a failure / slow
  }
  // dropTail() and evictOlderThan() DECREMENT the counters as they remove
  // samples. failureRate is then simply failures/size — no iteration, ever.
}
```

PROPERTY	VALUE	NOTE
push	`O(1) amortised`	eviction is a tail advance, never a scan
failureRate / slowRate	`O(1)`	read two counters and divide
memory per breaker	`~17 B × capacity`	1.7 KB at capacity 100; 100 hosts ≈ 170 KB
allocation on the hot path	`zero`	typed arrays preallocated at construction

Defaults, and where each comes from

KNOB	DEFAULT	PROVENANCE
`window.calls`	`100`	ADR-015; matches cockatiel `CountBreaker.size`
`window.maxAgeMs`	`300_000`	ADR-015; caps lookback at 5 min so low-traffic decisions stay current
`window.minCalls`	`20`	ADR-015; resilience4j uses the same concept
`failureRate`	`0.5`	**changed from ADR-015's 0.8.** 0.8 was chosen for one very specific provider; 0.5 matches resilience4j/Polly convention and is the defensible library default
`slowCallRate`	`0.5`	resilience4j `slowCallRateThreshold`
`slowCallMs`	`none — required`	**deliberately has no default.** "Slow" is meaningless without your baseline. Docs teach ≈3× healthy p95
`consecutiveBackstop`	`10`	ADR-015
`openForMs`	`15_000`	ADR-015; cockatiel 10 s, opossum 30 s, failsafe-go 60 s
`halfOpen.probes`	`1`	ADR-015 single-probe admission
`halfOpen.successesToClose`	`3`	ADR-015
`openBackoff`	`×1.0 (off)`	opt-in exponential re-open delay; cockatiel allows a backoff for `halfOpenAfter`

> **DELIBERATE DEVIATION FROM OUR OWN ADR**
>
> `failureRate` ships at **0.5**, not the 0.80 in ADR-015. 0.80 was tuned to one provider that was routinely 20–70% failed while still worth calling. Shipping it as a library default would ship that provider's pathology to everyone. The *capability* transfers; the *tuning* does not. `slowCallMs` takes this further and refuses to guess at all — a default there would be actively harmful.

Public API

```
import { pipeline, breaker, timeout, classifyHttp } from 'resilix';

const nimc = pipeline({
  key: (req) => new URL(req.url).host,        // per-host isolation via KeyRegistry
  classify: classifyHttp(),
  policies: [
    breaker({ failureRate: .5, slowCallRate: .5, slowCallMs: 3000,
              window: { calls: 100, maxAgeMs: 300_000, minCalls: 20 },
              consecutiveBackstop: 10, openForMs: 15_000,
              halfOpen: { probes: 1, successesToClose: 3 } }),
    timeout(15_000),
  ],
});

// ergonomic form
const res = await nimc.execute(req, (ctx) => fetch(req.url, { signal: ctx.signal }));

// manual form — same state machine, you own the call
const gate = nimc.gate(req);
if (!gate.ok) throw new Rejected(gate.reason);
try   { const r = await fetch(url); gate.settle(classifyHttp(r), elapsed); }
catch (e) { gate.settle(classifyHttp(e), elapsed); }
```

Tests

AREA	APPROACH
Determinism	Injected `Clock`. Zero real timers, zero `await sleep`. Every temporal assertion is `clock.advance(ms)`.
Classifier	Table-driven over ~60 fixtures: every status class, every Node error code, axios/fetch/undici error shapes, and a bare `Error`.
FSM	Explicit transition table test — every (state × verdict) pair asserted, including the impossible ones.
Window	Property-based: after any push/evict sequence, `failures` and `slow` must equal a brute-force recount. This is the invariant that catches counter drift.
Herd	Fire 50 concurrent admissions at a half-open breaker; assert exactly one is admitted.
Snapshot	Round-trip `hydrate(snapshot())` must be behaviourally identical, including window contents.
Edge runtimes	`import` the package at global scope under `wrangler dev` in CI. This is cockatiel #105 as a regression test.

Done when

- A dead host at 8 req/min opens inside 10 calls (backstop).
- A host at 0% errors and p50 10 s opens on slow-rate alone.
- A host at 15% 4xx and 0% 5xx never opens, at any traffic level.
- 50 concurrent half-open admissions ⇒ exactly 1 upstream call.
- `attw --pack --profile node16` and `publint` clean; imports cleanly on Node 18/20/22, Bun, Deno, Workers.
- Benchmark: `admit()+settle()` overhead < 1 µs, zero steady-state allocation.

8 · v0.2 — Telemetry and the opossum compatibility layer DISTRIBUTION

Make the library observable without a plugin, and make adoption a one-line diff.

What you must understand first

Polly v8's biggest practical win over v7 was not a new policy — it was **built-in telemetry**. The moment resilience decisions appear on the same dashboards as everything else, the library stops being invisible infrastructure and starts being operable. In JS today, opossum needs `opossum-prometheus` (7.8k/wk against opossum's 1.19M — under 1% of users instrument their breakers) and cockatiel offers event callbacks and nothing else. Instrumentation-by-default is a genuine differentiator.

Metrics

INSTRUMENT	TYPE	ATTRIBUTES
`resilience.executions`	counter	key, policy, verdict, outcome=`admitted|rejected`
`resilience.execution.duration`	histogram (ms)	key, verdict
`resilience.breaker.state`	gauge (0/1/2)	key
`resilience.breaker.transitions`	counter	key, from, to, reason=`failure-rate|slow-rate|consecutive|probe-failed|probe-success`
`resilience.breaker.failure_rate` / `.slow_rate`	gauge	key — the two numbers you tune against
`resilience.limiter.limit` / `.inflight` / `.queued`	gauge	key (v0.3)

> CONSTRAINT
> Telemetry is an **observer**, registered through the same `Events` hook consumers use. It cannot influence `admit()`. It must be sync and non-throwing — a wrapped, swallowing dispatch. A metrics exporter that starts throwing must not be able to take down the policy that is protecting your provider.

The compat layer

```
// the entire migration
- const CircuitBreaker = require('opossum');
+ const CircuitBreaker = require('resilix/compat/opossum');
```

OPOSSUM OPTION	MAPPING
`timeout`	→ `timeout()` policy
`errorThresholdPercentage`	→ `failureRate` (÷100)
`resetTimeout`	→ `openForMs`
`rollingCountTimeout` / `rollingCountBuckets`	→ `window.maxAgeMs`; bucket count is accepted and ignored (our window is not bucketed)
`volumeThreshold`	→ `window.minCalls`
`errorFilter`	→ wrapped into a Classifier: `true` ⇒ `answered`
`capacity`	→ `bulkhead()`
`cache` / `coalesce`	→ deferred; documented as unsupported, throws on use
events `open/close/halfOpen/reject/success/failure/timeout/fallback`	→ emitted with the same names and payload shapes

Compat is validated by running opossum's own test suite against the shim in CI. That is the honest bar for a drop-in claim, and it doubles as a strong README line.

Done when

- Zero-config OTel: one `otel()` observer produces all instruments above.
- Opossum's public test suite passes against `/compat/opossum`, with any skips enumerated in the docs.
- A Grafana dashboard JSON ships in the repo.

9 · v0.3 — Adaptive limiter HEADLINE RELEASE

Ship the only real adaptive concurrency limiter in JavaScript. This is the version that makes the project unignorable.

What you must understand first

1. **Latency rises before errors do.** When an upstream saturates, work queues internally and response times climb while the error rate stays flat. Our own production case: p50 0.35 s → 10.4 s with *zero* additional errors. A failure-rate breaker sees nothing until calls hit the timeout. A latency-driven limiter sees it immediately. This is why failsafe-go's own documentation says to *"prefer adaptive limiters over circuit breakers since they respond more quickly to indications of overload"* — and why failsafe-go's breaker deliberately has no latency dimension at all.
2. **Absolute latency is not a signal; a latency *ratio* is.** "800 ms is slow" is untrue for one upstream and absurd for another. Every serious implementation compares recent latency against a learned baseline. Envoy calls it `minRTT`, Vegas calls it `rtt_noload`, failsafe-go calls it the baseline window. Same idea.
3. **You must periodically re-measure the baseline, and it is expensive.** Once the limiter clamps down, observed latency is latency *under restriction* — it can no longer tell you the unloaded truth. Envoy solves this by pinning concurrency to 3 for a measurement window every 60 s. This *intentionally causes 503s*: Envoy's own docs say it "may cause noticeable increases in 503 responses". Understand this trade before implementing it, and document it as loudly as Envoy does.
4. **Rejection is a blunt instrument; queueing is better.** A hard limit converts a small overshoot into errors. failsafe-go queues into a multiple of the limit and only then rejects, and it rejects *gradually*.
5. **A ratio needs a percentile, and a percentile needs a streaming estimator.** You cannot sort samples per request. This is a real data-structure decision, covered below.
6. **Limiting a *rate* is not limiting *concurrency*, and this is the whole reason this version exists.** The AWS SDK already ships an adaptive controller in JS (§2.3) — but it adapts a **rate** (requests/second) in response to **throttling errors**. We adapt **concurrency** (in-flight count) in response to **latency**. By Little's Law, `L = λW`: hold the rate λ fixed and let time-in-system W triple during a slowdown, and concurrency L triples with it — a rate limiter cannot see that, and a streaming LLM call with a 60 s connection is exactly the case where W dominates. This is precisely why Netflix built `concurrency-limits` instead of using a rate limiter, and it is the sentence to have ready when someone asks how this differs from what AWS already does.

The control loop

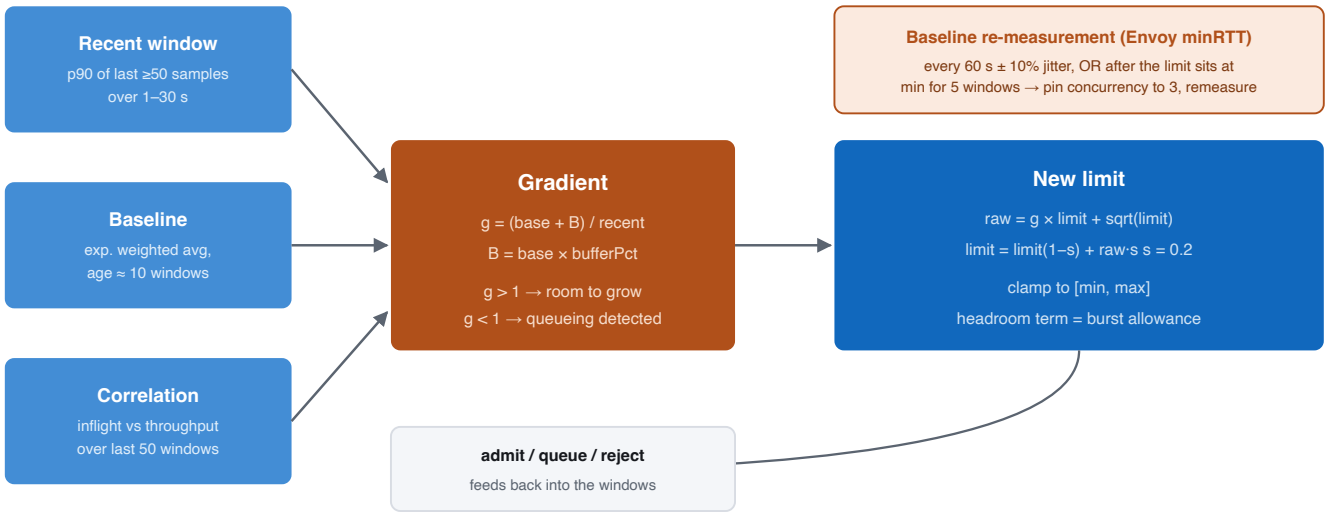


Figure 6 — Adaptive limiter control loop. Formula and defaults from Envoy's gradient controller and Netflix `Gradient2Limit`; queueing behaviour and the correlation signal from failsafe-go. The `sqrt(limit)` headroom term is unconfigurable in Envoy and fixed at 4 in Netflix's Gradient2 — we follow Envoy.

The three algorithms we ship, and their real defaults

KNOB	GRADIENT2 NETFLIX	VEGAS NETFLIX	AIMD NETFLIX	OURS
initialLimit	20	20	20	20
minLimit	20	–	20	4
maxLimit	200	1000	200	200
smoothing	0.2	1.0	–	0.2
queueSize / headroom	4	–	–	sqrt(limit)
longWindow	600	–	–	600
rttTolerance	1.5	–	–	1.5
alpha / beta	–	$3 \cdot \log_{10}(L)$ $6 \cdot \log_{10}(L)$	–	same
probeMultiplier	–	30	–	30
backoffRatio	–	–	0.9 asserted [0.5,1.0)	0.9
sample percentile	–	–	–	p90 (Envoy)
update interval	–	–	–	100 ms (Envoy)
baseline recalc	–	–	–	60 s ± 10% (Envoy)
concurrency during recalc	–	–	–	3 (Envoy)

```
Gradient2 (Netflix, verbatim from source):
  newLimit = gradient * currentLimit + queueSize
  newLimit = currentLimit * (1 - smoothing) + newLimit * smoothing

Envoy gradient controller (verbatim):
  gradient   = (minRTT + B) / sampleRTT          B = minRTT * bufferPct
  limit_new  = gradient * limit_old + headroom   headroom = sqrt(limit)   [not configurable]

Vegas (Netflix, verbatim from source comment):
  "limit increases by alpha if the queue_use is small (< alpha) and decreases
   by alpha if the queue_use is large (> beta)"
  alpha = max(3, 10% of limit) -> implemented as 3*log10(limit)
  beta  = max(6, 20% of limit) -> implemented as 6*log10(limit)
```

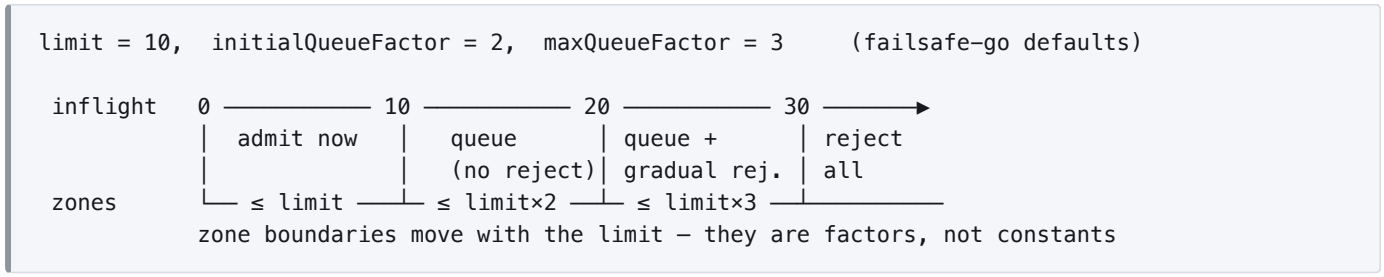
THIS AUDITS OUR OWN PRODUCTION LIMITER

`adaptive-concurrency.ts` in the gateway is AIMD, and it is misconfigured at both ends against Netflix's reference:

KNOB	GATEWAY	NETFLIX AIMD	EFFECT
initialLimit	1000 (= maxLimit)	20	starts fully open — no protection until the first failure
minLimit	1	20	sustained failure walks 1000→500→…→1; recovery is +1 per success, so ~1000 serialised calls to climb back
backoffRatio	0.5	0.9	0.5 is the most aggressive value Netflix's builder will even accept
latency signal	>5000 ms → ×0.9, hardcoded	none (AIMD is error-only)	a hand-rolled reach toward Vegas

Worth a separate ticket on the gateway, independent of this library: the `min 1` floor plus `+1` recovery is a real production hazard. The breaker likely masks it by opening first.

Queueing zones



Streaming quantiles — a real decision

OPTION	MEMORY	PER-SAMPLE	ACCURACY	VERDICT
Sort the window	O(n)	O(n log n)	exact	no — hot path
t-digest	~100s of centroids	O(log n)	excellent, all quantiles	overkill; large code for one number
HDR histogram	fixed buckets, KBs	O(1)	bounded relative error	good, but memory per key × many keys
P² (Jain & Chlamtac)	**5 markers**	**O(1)**	good for a single fixed quantile	**chosen** — we need exactly p90, per key, cheaply
Bounded sorted-insert ring (n≤64)	64 slots	O(log n) search + memmove	exact within window	fallback if P² proves unstable on bursty traffic

P² is ~60 lines, needs 5 numbers per key, and is exactly the "one fixed quantile, streaming" case it was designed for. We ship it behind a `Quantile` interface so the sorted-ring fallback is a one-line swap, and we validate P² against exact quantiles in property tests.

Done when

- Simulation harness: a synthetic upstream whose latency is a function of concurrency. The limiter must converge on its capacity within N windows and hold, without oscillating more than ±20%.
- Injected step change in capacity (halve it) ⇒ limit re-converges; recovery is observed after baseline re-measurement.
- The 25–30× slowdown scenario from production, replayed: the limiter must shed before the breaker would have opened.
- P² p90 within 5% of exact p90 across bursty, skewed, and bimodal latency distributions.
- Baseline re-measurement documented, including the 503 cost, in Envoy's own candid language.

10 · v0.4 — Adaptive throttler and execution budgets RETRY-STORM SAFETY

Stop the client itself from being the cause of the outage.

What you must understand first

A retry is an amplifier. Three attempts per request turns a degraded upstream into a 3× load spike exactly when it can least absorb one. Google's SRE book measures it: with a per-client retry budget capping retries at 10% of requests, worst-case growth drops from **3×** to **1.1×**.

Exactly one JavaScript implementation exists, and it is protocol-locked: `@grpc/grpc-js` ships `class RetryThrottler` — a token bucket with `maxTokens` / `tokenRatio` where `canRetryCall()` returns `tokens > maxTokens/2` — usable only for gRPC calls. No general-purpose retry library has one: not `p-retry` at 46.6M/wk, not `async-retry` at 29.7M/wk, not cockatiel. Every Node service retrying plain HTTP is running unbudgeted.

Second idea: a breaker is binary and a throttler is probabilistic. A breaker rejects *everything* while open, which means it also stops learning. A throttler rejects a computed *fraction*, so traffic keeps flowing, recovery is detected quickly, and there is no cliff.

Adaptive throttler — Google SRE client-side throttling

```
Track over a rolling window (2 min in the SRE book):
  requests = attempts made by the application layer
  accepts  = attempts accepted by the backend

  rejectionProbability = max(0, (requests − K * accepts) / (requests + 1))

K = 2 is the SRE book's aggressiveness knob. K > 1 means the client tolerates
some failure before self-regulating; smaller K throttles sooner.

failsafe-go's equivalent framing:
  WithFailureRateThreshold(.1, 10, time.Minute)   // start rejecting above 10% failures,
                                                  // min 10 executions
  WithMaxRejectionRate(.9)                        // never reject more than 90% —
                                                  // MUST be < 1.0 so recovery is detectable
```

> **WHY MAXREJECTIONRATE MUST BE BELOW 1.0**
> If the throttler can reach 100% rejection it stops sending any traffic, so it never observes that the upstream recovered — the same trap a breaker avoids with half-open probes. Capping at 0.9 keeps a trickle of real traffic as a continuous probe. This is a small detail with outage-shaped consequences.

Execution budgets

KNOB	DEFAULT	SOURCE
`maxRate`	`0.25`	failsafe-go — max fraction of in-flight executions that may be retries/hedges
`minConcurrency`	5	failsafe-go — floor so low-traffic services can still retry at all
SRE reference ratio	`0.10`	Google SRE book per-client retry budget

Budgets are **shared objects passed into multiple policies**, not per-policy config — the point is a system-wide cap. One budget instance across every retry and hedge policy in the process.

Also in v0.4

- **Retry** with the four jitter strategies (none / full / half / decorrelated), `abortOn` by verdict — `answered` is never retried — and `Retry-After` honouring on `overload`.
- **RateLimiter**, token bucket, composable as a policy. The docs page explains via Little's Law why a rate limiter does *not* bound concurrency and a bulkhead does — failsafe-go teaches this well and it is the most common confusion in the space.

Done when

- Simulated retry storm: unbudgeted amplification ≈3×, budgeted ≈1.1×. Reproduce the SRE book's numbers in a test.
- Throttler holds its computed rejection rate within ±5% of the analytic value across a failure-rate sweep.
- A single shared budget demonstrably caps retries across three independent pipelines.

11 · v0.5 — Hedging and prioritisation TAIL LATENCY + CRITICALITY

Two policies that exist in Polly and failsafe-go, and in JavaScript only inside gRPC.

Hedge — what it is, and how it differs from retry

```
RETRY   attempt 1 ──────────✗
                  attempt 2 ────────✓          sequential; pays the full failure first

HEDGE   attempt 1 ──────────────────────────────✓
             └── delay ──▶ attempt 2 ─────✓ (first wins, other cancelled)
                                                concurrent; pays extra load, saves tail lat
```

KNOB	DEFAULT	NOTE
`maxHedges`	1	failsafe-go default: one extra attempt
`delay`	required	fixed, or a function — the useful form is `() => p95Latency()`, so you only hedge calls already in the tail
cancellation	cancel all on first result	refinable: `cancelOnResult`, `cancelOnError`, `cancelIf`
`budget`	strongly recommended	hedging is a load amplifier exactly like retry — same budget mechanism

PRIOR ART, AND WHAT WE ADD

`@grpc/grpc-js` implements hedging properly — `hedgingPolicy` with `maxAttempts`, `hedgingDelay`, `nonFatalStatusCodes`, cancellation of losers, and integration with its `RetryThrottler`. It is a good implementation and we should say so. Its limits are that it works **only for gRPC calls**, is configured through **service-config JSON** rather than code, and takes a **fixed** hedging delay. Our additions are: any operation (HTTP, SQL, queue, an arbitrary function), a **dynamic** delay so you can hedge at measured p95 rather than a guessed constant, and composition with the rest of the pipeline.

CORRECTNESS WARNING THAT BELONGS IN THE DOCS, IN BOLD

Hedging sends the same request more than once. It is **only safe for idempotent operations**. Hedging a payment or a non-idempotent write duplicates it. Polly and failsafe-go both document this; ours should refuse to be subtle about it, and the docs page should open with it.

Prioritisation — Google SRE criticality

```
5 priority classes: VeryHigh · High · Medium · Low · VeryLow
internally expanded to 100 granular levels per class  (failsafe-go)

A shared Prioritizer watches load across every limiter it is attached to,
calibrates a rejection threshold on an interval (1 s), and rejects everything
below the threshold level. Under overload, VeryLow dies first; VeryHigh is
protected until last.

Optional UsageTracker adds per-user fairness: a heavy user's requests are
throttled before a light user's at the SAME priority. This is what stops one
tenant monopolising capacity during a squeeze.

"Prioritizers can and should be shared across multiple limiters" — failsafe-go
```

Why this matters for the kind of system we run: a health check, a background reconciliation job, and a customer-facing verification are not equally important, but a plain limiter treats them identically. Under overload you want the reconciliation to be shed and the customer request to survive. Nothing in JS offers this. It also composes with multi-tenancy — the `UsageTracker` is exactly the “one org is eating the whole provider quota” problem.

Done when

- Hedge reduces simulated p99 with a bimodal latency upstream, and the reduction is measured in the test, not asserted qualitatively.
- Under a saturating load test, VeryHigh throughput is preserved while VeryLow is shed, with a documented crossover point.
- `UsageTracker` keeps a light user's success rate above a heavy user's during overload.
- Docs page opens with the idempotency warning.

12 · v0.6 — Effect adapter 26.7M DOWNLOADS/WK

Serve a large, well-defined audience whose own maintainers declined to serve it.

Effect issue #2843, “Add Circuit Breaker”: opened May 2024, 12 reactions, closed by a maintainer as “a stale feature request… not enough recent interest or maintainer action to keep it actionable.” Effect has `Schedule` and `retry`; it has no breaker, no limiter, no throttler, no budget.

The adapter's job is to be idiomatic, not merely functional: policies as `Effect` combinators that thread `Cause` and interruption correctly, a `Layer` for pipeline construction, and classifiers that map Effect's tagged errors to verdicts rather than sniffing exceptions. If it looks like a wrapper, Effect users will not adopt it.

13 · v1.0 — Transport adapters, runtime matrix, docs site

Make the library the obvious default rather than the interesting option.

Adapters

SUBPATH	CONTENTS
`/fetch`	guarded `fetch`, per-host keying, `Retry-After` parsing, WHATWG-only so it works on every runtime
`/undici`	an undici interceptor — meets Node users where the platform is going
`/axios`	request/response interceptor pair; the migration path for the existing installed base
`/nest`	module + DI providers + a method decorator; per-host and per-provider pipelines
`/hono`, `/fastify`	middleware, and inbound guards once v2.0 lands

Runtime matrix — a CI badge, and a competitive statement

```
Node 18 · Node 20 · Node 22 · Node 24 · Bun · Deno · Cloudflare Workers (wrangler) · Vercel Edge

Every runtime runs the FULL suite, plus a global-scope import smoke test.
This is the exact thing cockatiel declined to support (#105: "low motivation
to cater to capricious choices of random runtimes"). It is cheap for us
because the zero-I/O, zero-dep core makes it true by construction.
```

Docs site — structure copied from failsafe-go, deliberately

PAGE	WHY IT EXISTS
One page per policy	Reference. Same shape every time: what, when, knobs, defaults, events, gotchas.
`/load-limiting`	**The most important page.** Teaches proactive vs reactive, and says plainly which to reach for. This concept is why failsafe-go reads as authoritative at 2.2k stars.
`/policy-composition`	Ordering is semantic, not stylistic. Breaker-inside-retry vs breaker-outside-retry changes what a burst of retries means to the breaker.
`/comparisons`	Honest, specific, and linked to issue numbers. Where cockatiel or opossum is the better choice, say so. **Must include grpc-js and the AWS SDK** (§2.3) and explain rate-vs-concurrency — if a reader discovers that prior art on their own, we look either ignorant or evasive.
`/production-story`	The 25–30× slowdown at 0% error rate, and the 13–18% healthy 4xx rate. The motivating example no competitor can write.
Little's Law explainer	Rate limiter vs bulkhead — the single most common confusion in this space.

> **BUDGET THIS HONESTLY**
>
> Polly won .NET on documentation as much as on code. Plan for docs to take **as long as `src/` did**. It is not overhead; at v1.0 it is the product.

14 · v2.0 — Inbound (system-adaptive) protection

Open a second axis: protect your own process, not just your upstream.

Every JS resilience library, including this one through v1.0, is **outbound**: it guards calls you make. Alibaba Sentinel — 23,141 stars, more than Polly or resilience4j, and with no JS port — pioneered the **inbound** axis: shed incoming work based on the health of the machine you are running on.

```
Signals (Sentinel):  load1 · CPU · inbound QPS · global avg RT · global max concurrency

Block inbound work when either holds:
   load1 > highestSystemLoad
   concurrentThreads > minRt * maxQps        <- BBR-style capacity estimate

Stated goal, verbatim: "increase the throughput of the system within appropriate
system load, rather than the load must be restricted below a threshold"
```

The distinction matters: an outbound limiter cannot save you from your own event loop being saturated by 1,000 concurrent inbound requests. A JS-native equivalent would read `os.loadavg()`, event-loop delay/utilisation (`perf_hooks`), and heap pressure. That is Node-specific I/O, so it lives in `/system`, never in core.

SIGNAL	JS SOURCE	NOTE
load1	`os.loadavg()[0]`	meaningless in containers without cgroup awareness — must be documented
event-loop delay	`perf_hooks.monitorEventLoopDelay()`	the *best* JS-native saturation signal; better than CPU%
event-loop utilisation	`performance.eventLoopUtilization()`	directly analogous to CPU utilisation for a single-threaded runtime
heap pressure	`process.memoryUsage()`	leading indicator of GC thrash
inbound concurrency / RT	our own counters	already available from the pipeline

Positioned as v2.0 because it needs the whole reactive machinery from v0.3–v0.5 to be worth anything, and because the framework adapters from v1.0 are how anyone would actually apply it.

15 · Cross-cutting decisions

These are the decisions that constrain every version. They are numbered so the codebase and the docs can cite them, and they should be written up as real ADRs in the repository.

#	DECISION	RATIONALE, AND WHAT WE REJECTED
1	Core has zero runtime dependencies and performs no I/O	Makes the runtime matrix true by construction, keeps the failure path free of anything that can itself fail, and keeps the install cost of the whole library near zero. *Rejected:* a convenience dep like `es-toolkit` — the only TS adaptive limiter that exists took two deps and became unusable for other reasons anyway.
2	No distributed or shared policy state, ever, in core	Cross-instance ejection is the service mesh's job — Envoy outlier detection, Istio, Consul, Kong Mesh all own it, precisely to avoid coordination overhead. cockatiel's maintainer analysed it and declined (#89: read latency in the hot path, distributed locking for half-open probes, and failure handling for the coordination store itself). Demand is also weak: opossum #781 has sat three years, unanswered, with four comments. *We do ship* `snapshot()`/`hydrate()` for serverless, which is the real need behind most of those requests.
3	Outcomes are classified into a verdict, not reduced to a boolean	The single differentiating primitive. A boolean predicate cannot express "not a breaker failure, but still a load signal" (429) or "the upstream is healthy, the caller was wrong" (4xx). With 13–18% healthy 4xx, getting this wrong means customer input can open your circuit. Three years of opossum #818 is people asking for this. *Rejected:* cockatiel's `handleWhen` shape.
4	Policies are `admit`/`settle` state machines; only the Executor is async	Policies become synchronous, allocation-free, and trivially testable; consumers who want to drive them from a non-promise context can. *Rejected:* `execute(fn)` as the primitive, which is what couples opossum and cockatiel to invocation mechanics.
5	Time is injected via a `Clock`	Every temporal behaviour becomes deterministically testable with no timers and no sleeps. Also removes `Date.now()` from hot paths in favour of one read per execution. Non-negotiable, and cheap only if adopted from the first commit.
6	Latency is a first-class signal in every policy	Our production failure mode was a 25–30× slowdown at a 0% error rate. Any design that only records success/failure is blind to it. Consequence: `Observation` always carries `latencyMs`; the breaker gets a slow-rate condition; the adaptive limiter is latency-primary.
7	Our own rejections are never upstream evidence	The `rejected` verdict. Without it, an open breaker observes its own rejections and can never close — a self-sustaining outage. cockatiel #115 is this bug.
8	Prefer queueing and probabilistic shedding over hard rejection	A hard limit converts a small overshoot into errors. failsafe-go queues to a multiple of the limit and then rejects gradually; the SRE throttler rejects a computed fraction. Both keep some traffic flowing, which is also what lets the system detect recovery.
9	Ship capabilities from our production experience; do not ship our tuning	`failureRate` defaults to 0.5, not our ADR-015 value of 0.80, which was tuned to one provider's pathology. `slowCallMs` has *no* default at all, because a wrong guess there is worse than a required argument.
10	Telemetry is a non-throwing observer, built in, not a plugin	Under 1% of opossum users instrument their breakers (`opossum-prometheus` 7.8k/wk vs opossum 1.19M/wk) — a plugin model means nobody has data when it matters. Polly v8's built-in telemetry is its most-cited improvement. Hard constraint: telemetry cannot influence `admit()`.
11	One package, many subpath exports	No version skew between core and adapters, one release, one README, tree-shakeable. The `naija-id` pattern at larger scale. *Rejected:* a monorepo of separately versioned packages.

12	Ship a compatibility shim for the incumbent	Adoption cost is the real barrier, not features. A one-line import swap, validated by running opossum's own test suite in our CI, converts “I'd have to rewrite” into “I'll try it on a branch”. Nobody in this space has done it.

16 · Glossary

TERM	MEANING
AIMD	Additive Increase, Multiplicative Decrease. Raise the limit by a constant on success, cut it by a ratio on failure. TCP's classic congestion response; error-driven, so it reacts *after* damage.
TCP Vegas	Congestion control that infers queueing from RTT rather than from loss. Estimates queue occupancy as `limit × (1 − rtt_noload/rtt)`; grows by α when the queue is small, shrinks by β when large. The intellectual ancestor of every adaptive concurrency limiter.
Gradient / Gradient2	Netflix's refinement: compare a short-window RTT against a long-window exponentially-smoothed RTT, and scale the limit by that ratio. Gradient2 compares two exponential averages rather than short-vs-absolute-minimum, which is more stable on bursty traffic.
minRTT	Envoy's name for the best-case, unloaded round-trip time — the baseline a gradient is measured against. Must be periodically re-measured under artificially low concurrency, because once you clamp, you can no longer observe it.
BBR	Bottleneck Bandwidth and RTT. Models the path's actual capacity instead of reacting to loss. Sentinel's system-adaptive protection is BBR-inspired: estimate what the system can handle, then admit that much.
Little's Law	$L = \lambda W$ — concurrency equals arrival rate × time in system. Why a rate limiter does not bound concurrency: hold λ fixed, let W triple during a slowdown, and L triples. A bulkhead bounds L directly.
Bulkhead	A hard concurrency cap, named after ship compartments: flooding one does not sink the vessel. Proactive — you must know the right number in advance.
Hedging	Issuing a duplicate request after a delay and taking whichever finishes first. Trades extra load for lower tail latency. Only valid for idempotent operations.
Retry budget	A cap on retries as a fraction of total requests. Google SRE uses 10%, which holds worst-case amplification to ~1.1× instead of 3×.
Client-side throttling	The client rejects a computed fraction of its own requests when the accept rate drops: `max(0, (requests − K·accepts)/(requests + 1))`. Probabilistic, so unlike a breaker it never fully blinds itself.
Criticality	A priority attached to a request so that, under overload, low-value work is shed before high-value work. From the SRE book; implemented in failsafe-go as five classes × 100 levels.
Slow-call rate	The fraction of calls exceeding a latency threshold. resilience4j's `slowCallRateThreshold`, and the only breaker dimension that catches an upstream that is up but degraded. Absent from every JS library.
P² algorithm	Jain & Chlamtac's streaming quantile estimator. Maintains five markers and updates them with a piecewise-parabolic prediction — O(1) time and memory for a single target quantile.
Outlier detection	Envoy's mechanism for ejecting a misbehaving endpoint from the load-balancing set. The infrastructure-layer answer to what a distributed circuit breaker would try to do in the application.

Primary sources. failsafe-go.dev (load-limiting, adaptive-limiter, adaptive-throttler, circuit-breaker, policies, execution-budgets, execution-prioritization, hedge, comparisons) · Netflix/concurrency-limits source (`Gradient2Limit.java`, `VegasLimit.java`, `AIMDLimit.java`) · Envoy adaptive concurrency filter documentation · Google SRE Book, *Handling Overload* · resilience4j documentation · Polly v8 documentation and telemetry guide · Alibaba Sentinel, system-adaptive protection · **shipped-source reads:** **`grpc-node/packages/grpc-js/src/retrying-call.ts` (hedging + `RetryThrottler`) and `@smithy/core/retry/DefaultRateLimiter` (CUBIC adaptive rate limiting)** · gRPC proposal A6, client retries · GitHub issues: cockatiel #80, #89, #105, #115, #119; opossum #781, #817, #818, #819; Effect-TS #2843 · npm registry download and metadata APIs, 10 Aug 2026 · internal: ADR-005, ADR-015, `libs/core/src/infrastructure/http/http.service.ts`, `adaptive-concurrency.ts`.