Gaurav Agrawal, Raptors Fellow, Senior Distinguished Engineer

Building eCommerce Systems That Survive Their Own Success

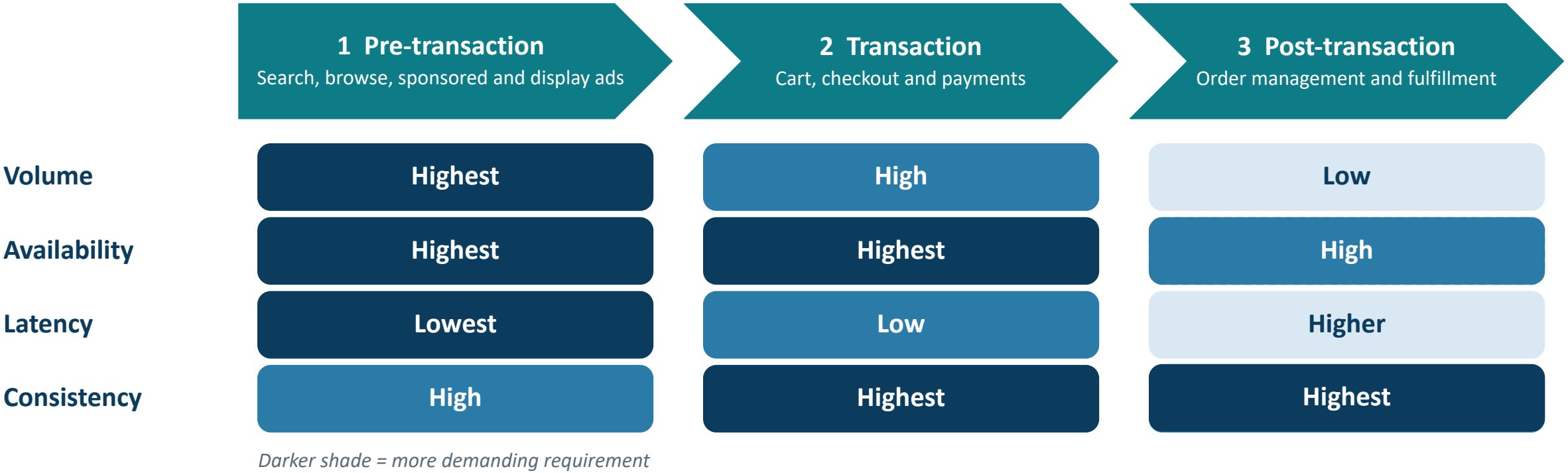
Low-latency systems that scale horizontally — "Infinite" scale, distributed architecture, data consistency, pre-computation and caching strategy, non-blocking programming and memory & networking optimizations along with resiliency that keeps systems dependable as demand grows.

eCOMMERCE SYSTEMS • DISTRIBUTED ARCHITECTURE • OPERATIONAL RESILIENCE

3 domains of eCommerce User Interaction

eCommerce touches our lives daily: grocery, food delivery, shopping, rides and travel booking. Its systems must scale to serve all users, 24x7.

Millions
requests per minute

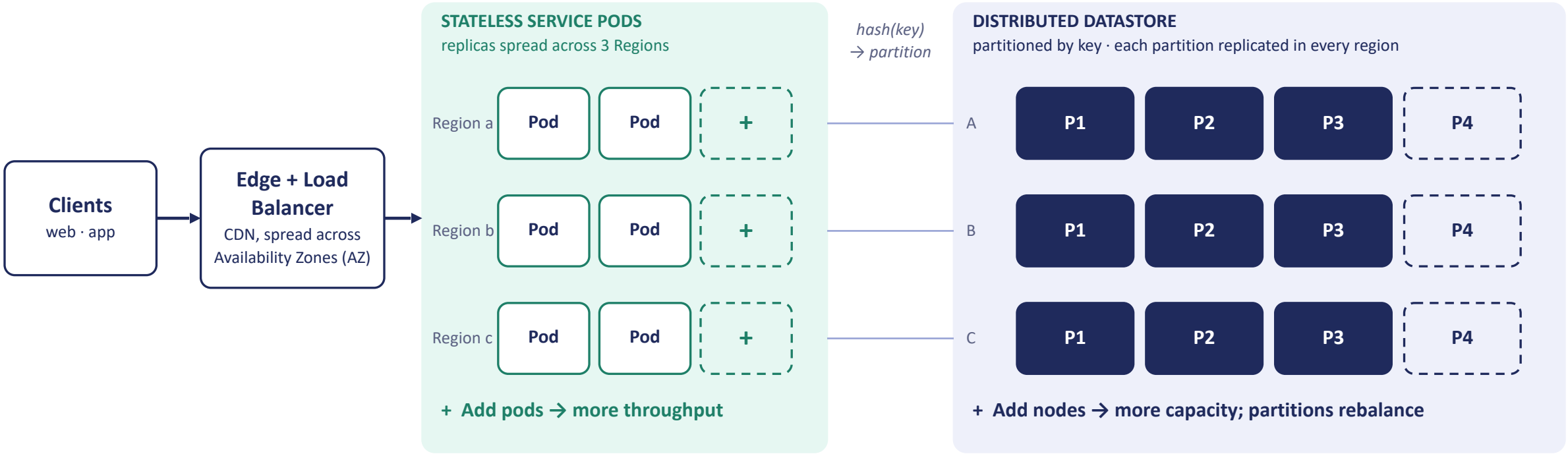


Darker shade = more demanding requirement

Why it matters Underperformance of any system degrades user experience and hinders adoption, so each domain must be designed to function and scale appropriately.

Scope of this paper Designing systems that withstand the highest user traffic with the lowest latencies, very high availability and data freshness.

Distributed Architecture



Stateless service pods

No local state or sticky sessions: any pod serves any request, so pods can be added, lost or replaced freely.

Partitioned, replicated data

A high-cardinality key spreads data and traffic evenly; replicas in every region keep it available.

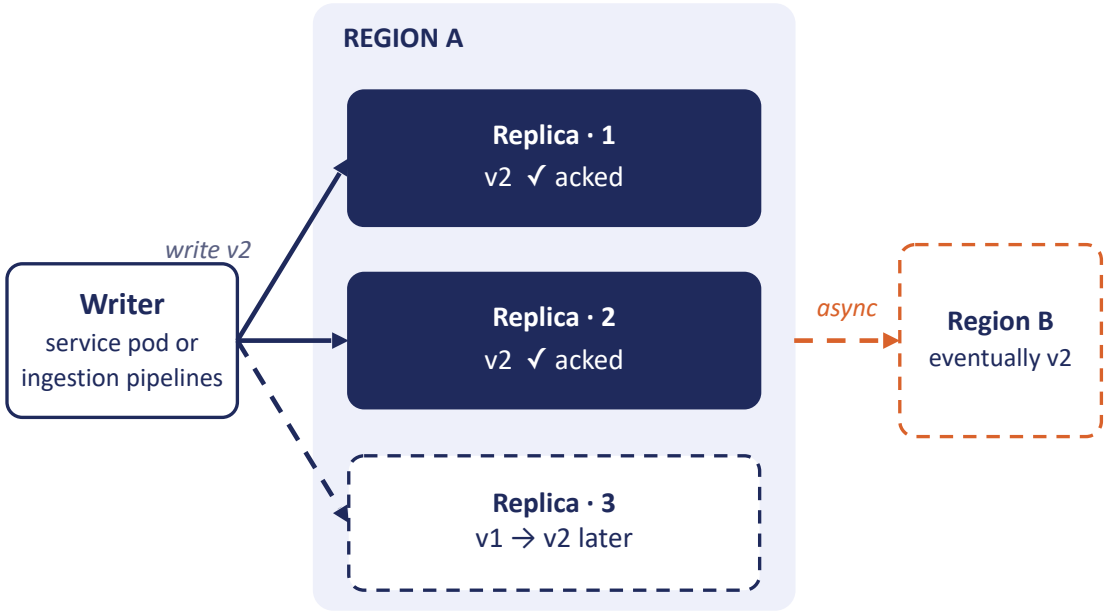
No shared bottleneck

Single-partition operations; no global locks, counters or cross-partition transactions; eventual consistency by default.

Scale horizontally → "infinite" scale Capacity grows with every pod and node you add, as long as no shared bottleneck remains.

Consistency Trade-offs

HOW LOCAL_QUORUM WORKS (ReplicationFactor = 3)



Write locally at LOCAL_QUORUM; the datastores replicate asynchronously to all regions.
Versioned, idempotent writes: skip stale or duplicate updates; retries are safe.
Concurrent cross-region updates to a key resolve by version or timestamp.
Adaptive rate limits and priority tiers keep ingestion from starving reads.

CHOOSE THE LEVEL PER DATA TYPE

Eventual consistency: replicas converge, and a read may briefly return older data. Set a staleness budget per data type and track replication lag as an SLI.

← faster, more available, may be stale *fresher, slower, less available →*

Level	Reads see	Latency / availability	Use for
LOCAL_ONE	May be stale	Lowest latency, highest availability	Browse, search, catalog reads
LOCAL_QUORUM	Consistent in region (read-your-writes)	Low latency; survives 1 of 3 replicas down	Price and Inventory display
EACH_QUORUM	Strong consistency across regions	Cross-region latency; lowest availability	Checkout and Inventory reservations

Staleness budget (example)

Catalog, search minutes	**Price, stock display** seconds	**Checkout, reservation** strong, no staleness

Takeaway use the weakest consistency each data type can tolerate; reserve strong guarantees for checkout and inventory.

Read-Optimized Architecture

Response-ready documents can be shipped directly to clients of your services.

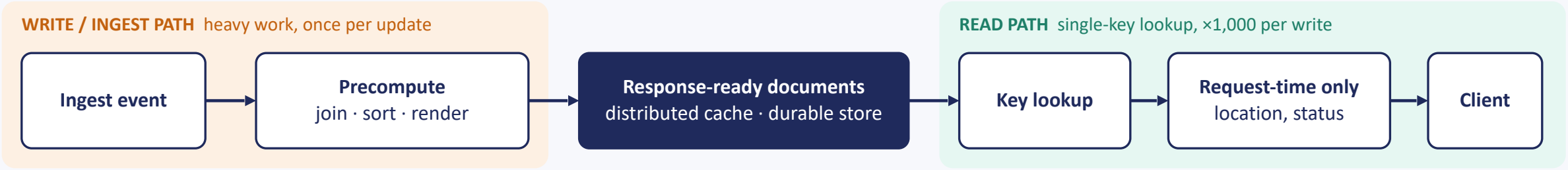
Core idea: move work from the read path to the write/ingest path. Reads become single-key lookups that do only the unavoidable computation at request time, with distributed caches serving hot documents.
Why it works: in high-volume services, reads often outnumber writes 1000:1 or more, so repeating resource-intensive work on every read wastes resources and adds latency.
Example: keeping a list ordered makes each write costlier (20 steps vs 1) but each read far cheaper (20 steps vs 1M), which is why precomputation is central to scaling.

Why pre-computation wins

Operation	Count	Unordered list	Ordered (precomputed)
Writes	1	1 × 1 = 1	20 × 1 = 20
Reads	1,000	1M × 1,000 = 1B	20 × 1,000 = 20K
Total steps		**≈ 1B**	**≈ 20K**

1M entries, 1,000 reads per write; cells show steps × count (worst case). Ordered ≈ log2(1M) = 20, using a balanced tree or skip list.

≈50,000× fewer steps overall for 1M item list.
Imagine the steps saved when we scale to Billions of items in a list



Isolate write pressure
Keep the write path narrow and separate so read scaling never competes with ingestion.

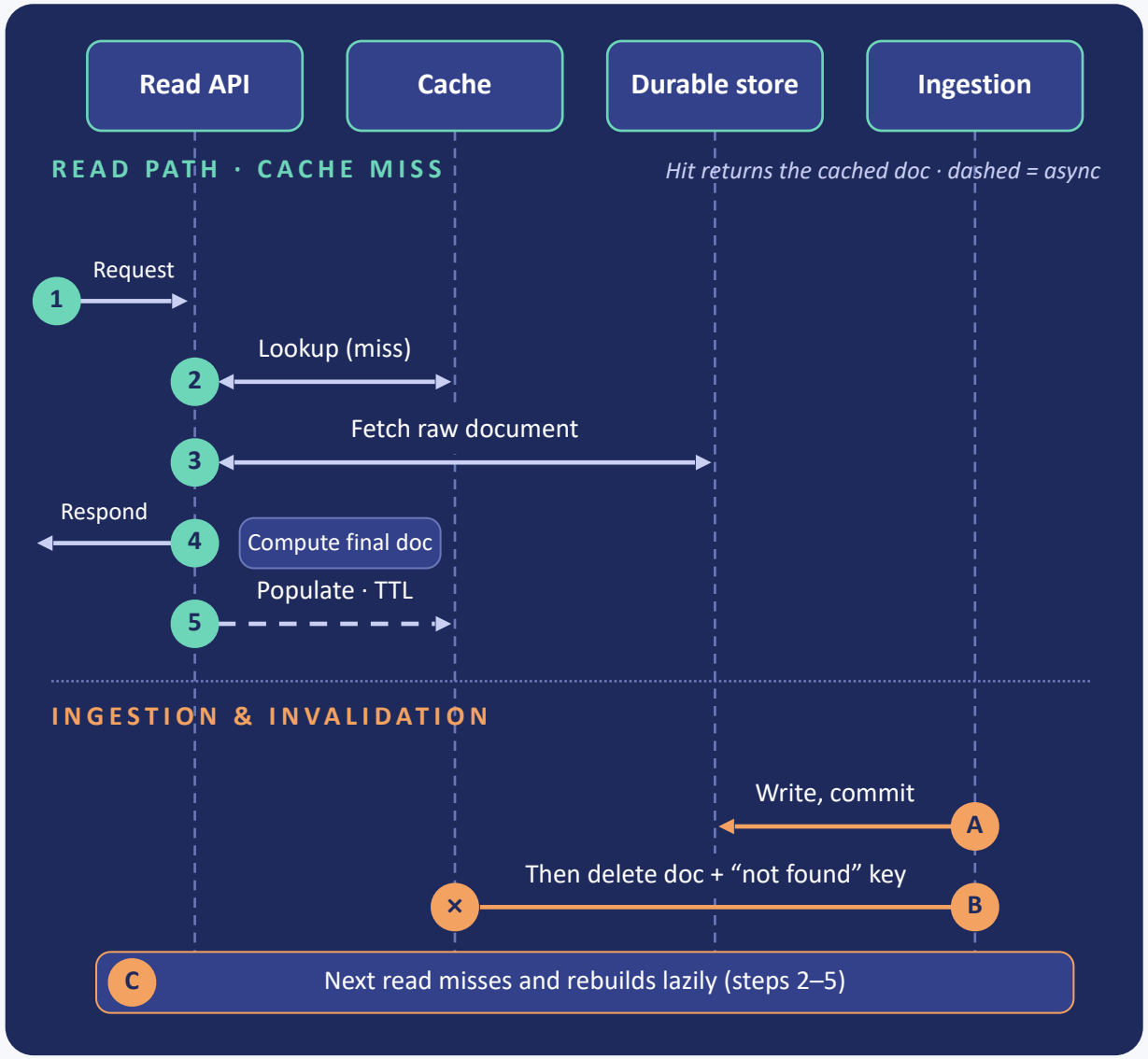
Denormalize for reads
Precompute joins into read-shaped documents so the hot path never assembles data at request time.

Serve from cache
Cache aggressively close to the client; treat the primary store as a fallback, not the first stop.

Read-through Caching & Invalidation

A read-optimized architecture needs a caching strategy that keeps precomputed documents in fast caches, ready to serve on request.

- Invalidate on the ingestion event only after the durable-store write commits; TTL backstops any residual staleness.
- Read-through caches a document only when first requested, then serves it precomputed; unlike write-through, the cache need not hold every document.
- Briefly cache “not found” results so repeated lookups of nonexistent keys, such as bot traffic, don’t hit the DB.
- Trade-off: heavy ingestion invalidating hot documents can cause cache-miss storms and stampedes, so the rebuild path needs protection. One common way is to write the new version of known-hot documents in cache instead of deleting



Non-blocking code for concurrency

When requests mostly wait for I/O, blocked threads waste memory and CPU on scheduling. Non-blocking event-loop threads instead multiplex thousands of in-flight requests. Java has Reactor and RxJava frameworks; all major languages support non-blocking constructs

1 Never block the event loop

One blocked thread can stall thousands of requests. Catch violations in tests - Thread.sleep; blocking JDBC; block() on event-loop thread

```
// fails the test if a blocking call runs on an event-loop thread
BlockHound.install();
```

2 Use non-blocking clients end to end

WebClient, R2DBC, Lettuce, async drivers. One blocking driver defeats the benefit.

```
Mono<Product> product = webClient.get()
    .uri("/products/{id}", id)
    .retrieve().bodyToMono(Product.class);
```

3 Run independent calls in parallel

Latency is the slowest call, not the sum of all calls.

```
Mono.zip(productClient.get(id), priceClient.get(id),
         stockClient.get(id), (p, pr, s) -> render(p, pr, s));
```

4 Fan out with bounded concurrency

At most 32 calls in flight; unbounded fan-out risks OOM and overload.

```
Flux.fromIterable(ids)
    .flatMap(id -> productClient.get(id), 32)
    .collectList();
```

5 Isolate blocking and CPU-heavy work

Blocking calls on boundedElastic(); CPU-heavy work on parallel(), sized to cores.

```
Mono.fromCallable(() -> dao.find(id))   // blocking
    .subscribeOn(Schedulers.boundedElastic());
flux.publishOn(Schedulers.parallel()).map(this::render); // CPU
```

Examples use Reactor. RxJava equivalents: Single.zip, flatMap with maxConcurrency, subscribeOn(Schedulers.io()).

Memory and networking optimizations

During heavy load, a single service instance may receive multiple GBs over network calls within a minute. Memory and network efficiency is therefore essential for availability.

1 Pass bytes through, stream large payloads

Compact formats (protobuf, compression); skip deserializing where you can. Stream instead of buffering.

```
// stored pre-serialized and pre-compressed: send the bytes as-is
ServerResponse.ok().header("Content-Encoding", "gzip")
    .bodyValue(storedGzipBytes);
```

2 Bound memory, concurrency and time

Capped buffers and in-flight limits prevent OOM; timeouts free slow calls.

```
source.onBackpressureBuffer(1_000, BufferOverflowStrategy.DROP_OLDEST)
      .flatMap(id -> call(id).timeout(Duration.ofMillis(200)), 32);
```

3 Keep allocation low, use a low-pause GC

Don’t retain references past the request. ZGC or tuned G1 cut pauses; leave heap headroom.

```
-XX:+UseZGC -Xms8g -Xmx8g        # fixed heap with headroom
-Xlog:gc*:file=gc.log:time,uptime
```

4 Reuse connections

Pool datastore and HTTP/2 or gRPC connections; new connections on each request cannot scale.

```
ConnectionProvider.builder("api").maxConnections(200)
    .maxLifeTime(Duration.ofMinutes(5))   // rebalance across backends
    .build();
```

5 Batch calls, with limits

Fewer round trips; cap batch size and handle per-item failures.

```
Flux.fromIterable(ids)
    .buffer(100)                          // 100 ids per call
    .flatMap(batch -> client.getMany(batch), 4);
```

6 Route calls locally

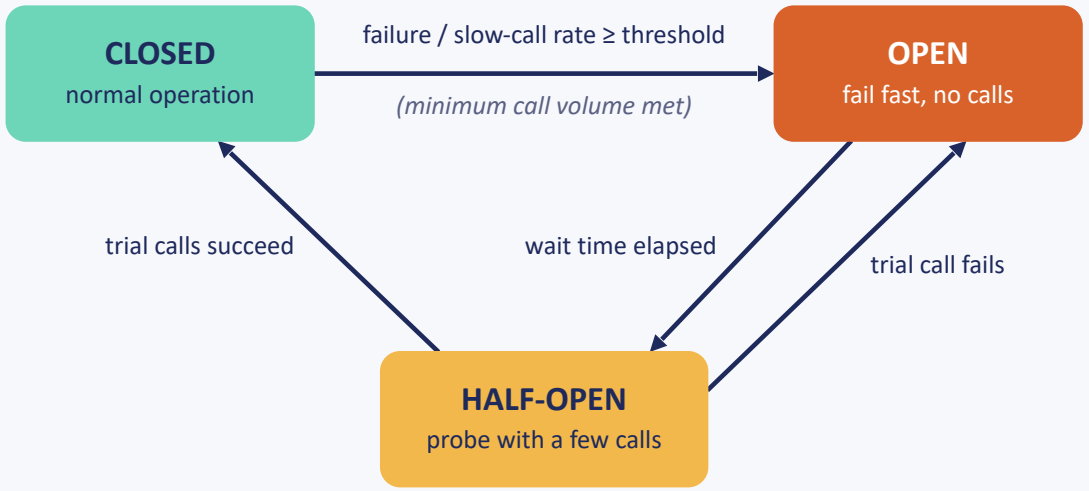
Prefer same-region, ideally same-AZ endpoints; cross-region calls add latency and cost.

```
CqlSession.builder()
    .withLocalDatacenter("us-west-2");   // prefer local replicas
```

Examples use Spring WebFlux, Reactor Netty and the DataStax Cassandra driver; adapt them to your stack.

Resiliency Programming

CIRCUIT BREAKER



Always define the fallback: a stale cached doc, a degraded response (skip personalization), or a default. A breaker without a fallback just fails faster.

- **Trip on:** failure rate over a sliding window (e.g. ≥50% of the last 100 calls) or slow-call rate (e.g. >200 ms), once a minimum call volume is met.
- **Scope:** per dependency, ideally per endpoint/partition, so one bad shard doesn't open the breaker for healthy ones.
- **Tools:** Resilience4j (CircuitBreakerOperator for Reactor); Envoy/Istio outlier detection.

OTHER SELF-HEALING MECHANISMS

Timeouts + deadline propagation
Bound every call; pass the deadline downstream so work stops once the client timeouts

Observability
Observability and monitoring using grafana + prometheus metrics and logging. Alerting in case of system issues are critical

Load shedding + concurrency limits
Pod-level adaptive (Vegas or gradient) concurrency to reject at saturation; Cluster level rate limit for consumer throttling

Health checks + remediation
Liveness (restart), readiness (stop traffic), startup probes; autoscale on concurrency, latency or queue lag

Outlier ejection + slow start
Eject hosts with consecutive 5xx or high latency, re-admit after a cool-down; ramp traffic and warm caches before readiness.

Graceful degradation
Automatically switch to graceful degradation by removing nice-to-have features to keep serving core functions where possible.

Region failover
Multi-region deployment with automatic failover to the surviving region; each region must have enough capacity.

Data-level self-healing
Error replay to ensure data accuracy; keep data-backup for full disaster recovery.

Chaos testing + game days
Kill nodes, inject latency and drop dependencies to verify that breakers, fallbacks and shedding behave as designed.

WHAT NOT TO DO

Scaling Anti-patterns

× Computing on every read

Repeating joins and rendering when reads far outnumber writes.

Fix Precompute response-ready documents and cache them.

× Cross region latencies

Cross-region connections with dependent systems leads to system latency

Fix Service pods to connect to local region datastores and caches. No cross-region connections

× Unbounded everything

Queues, threads, retries and data growth turn overload into OOM.

Fix Bound them all; shed load at saturation.

× Blocking the event loop

One blocking call stalls thousands of in-flight requests.

Fix Stay non-blocking end to end; isolate legacy blocking on a bounded pool.

× Chatty call chains

N+1 calls and deep fan-out multiply latency and failure odds.

Fix Batch, parallelize or precompute; cap fan-out; set hard deadlines.

× Retry storms

Retries at every layer, without backoff, amplify an outage.

Fix Let users do the retries on API calls, retries only during critical checkout ops

× Shared bottleneck

One primary DB, lock or counter caps every request.

Fix Partition by a high-cardinality key; generate IDs locally.

× Coordination on the hot path

Cross-partition transactions, locks and strong consistency everywhere.

Fix Single-partition writes, sagas, eventual consistency; strong only for price and inventory.

× Running hot, testing flat

Latency climbs sharply near saturation; uniform tests hide skew and failures.

Fix Track p99 latencies, test skewed load with injected failures.

The common thread shared bottlenecks · unbounded resources · synchronous agreement · untested assumptions

Results

1 SCALE

Handle more than 10Million calls per minute with P99 < 50ms for batched calls of 25+ documents

2 Resources

Extract the maximum efficiency from the CPU and Memory available using the non-blocking techniques.

3 Availability

Services never fail, 99.99% or 4 9’s availability with very no error or downtime

Scale with maximum efficiency and highest availability

“The architecture makes scale possible.
The operational discipline makes it survive.”

Gaurav Agrawal, Raptors Fellow, Sr Distinguished Engineer

Thank you