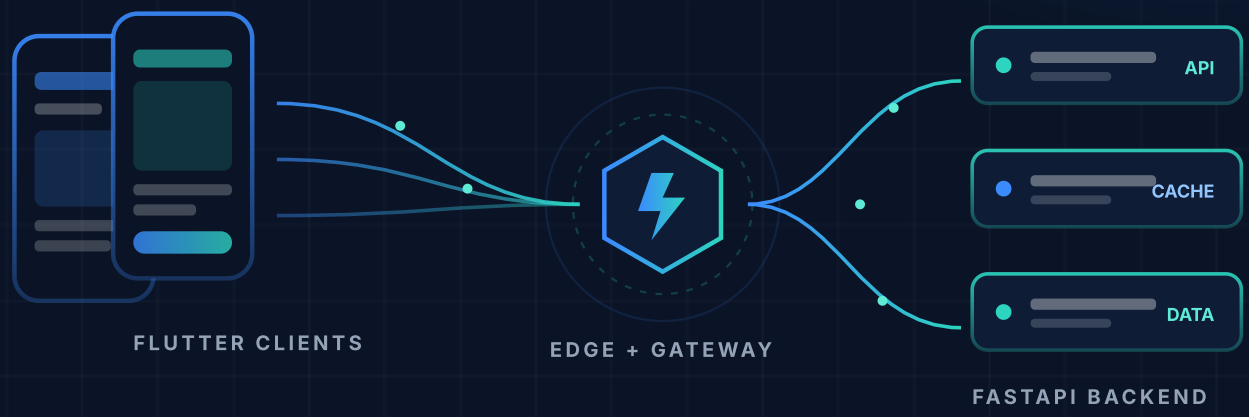
Steigern Tech

RESOURCE KIT · 2026 EDITION

ARCHITECTURE · PERFORMANCE · SCALE

The Enterprise Guide to Flutter & FastAPI Scalability

How to engineer cross-platform mobile apps that handle **100k+ concurrent users** with **sub-100 ms API response times** — the architecture, the math, the playbooks and the launch checklist.



100k+
concurrent users, designed for from day one

<100 ms
p95 server-side API latency target

1 → 6
one Dart codebase, six target platforms

99.95%
availability SLO (~22 min downtime / month)

Published by Steigern Tech LLP · steigerntech.com

14 pages · Infographic edition

INSIDE THIS KIT

From first commit to 100k concurrent users

A field guide for CTOs, product owners and engineering leads deciding whether Flutter and FastAPI can carry an enterprise mobile product — and exactly how to build it so they do.

03 **Should you choose this stack?**
Decision matrix vs. React Native + Node and fully native

04 **Reference architecture**
The production blueprint, layer by layer

05 **Capacity math**
Turning "100k users" into pods, connections and QPS

06 **The 100 ms latency budget**
Where every millisecond of a request goes

07 **FastAPI performance playbook**
8 rules + production-ready setup code

08 **Flutter client playbook**
Frame budgets, isolates, offline-first networking

09 **Caching & data layer**
The cache pyramid and PostgreSQL at scale

10 **Real-time & async work**
WebSockets, push, queues and the outbox pattern

11 **Scaling roadmap**
Three stages, and the metrics that tell you to move

12 **Observability & load testing**
SLOs, golden signals and a load-test plan

13 **Launch-readiness checklist**
40 checks across security, reliability and performance

14 **Work with Steigern Tech**
Architecture reviews and build teams

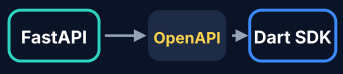
THE STACK AT A GLANCE

Flutter · client — DART

- **One codebase** for iOS, Android, Web, Windows, macOS and Linux.
- **Compiled ahead-of-time** to native machine code on mobile — no JS bridge at runtime.
- **Own rendering engine (Impeller)** draws every pixel: identical UI on every device, built for 60/120 fps.
- **Hot reload** and a mature package ecosystem shorten feature cycles.

FastAPI · backend — PYTHON

- **Async-native (ASGI)** on Starlette + Uvicorn — thousands of concurrent I/O waits per process.
- **Pydantic v2** validation with a Rust core: strict, fast request/response models.
- **OpenAPI generated automatically** from type hints — docs and contracts for free.
- **Python's AI/ML ecosystem** sits one import away from your API.



The combined advantage: an end-to-end typed contract

FastAPI emits an OpenAPI spec from your Python types; a generator (e.g. **openapi-generator dart-dio**) turns it into a typed Dart client. A renamed field breaks the build in CI — not the app in production.

01 · DECISION FRAMEWORK

Should you choose this stack?

No stack wins on every axis. Here is an honest comparison against the two alternatives teams most often weigh, scored for an enterprise app that must ship on several platforms and scale.

CRITERION	FLUTTER + FASTAPI	REACT NATIVE + NODE.JS	NATIVE (SWIFT/KOTLIN) + GO
Time-to-market across platforms	●●●●●	●●●●●	●●○○○
Pixel-consistent UI on every device	●●●●●	●●●○○	●●○○○
Native look & day-one OS APIs	●●●○○	●●●●○	●●●●●
Backend throughput per CPU core	●●●○○	●●●●○	●●●●●
AI / ML & data-science integration	●●●●●	●●●○○	●●○○○
Typed API contract, client ↔ server	●●●●●	●●●●○	●●●○○
Hiring pool	●●●●○	●●●●●	●●●●○
Long-term maintenance cost	●●●●○	●●●●○	●●○○○

Scores are Steigern Tech's editorial judgement for a typical multi-platform business app; weight them for your own priorities.

✓ Choose it when…

- You need iOS + Android (and often web/desktop) from **one team**.
- Brand-consistent, custom UI matters more than stock OS widgets.
- The backend is mostly **I/O-bound**: databases, caches, third-party APIs.
- AI features — recommendations, search, LLM calls, analytics — are on the roadmap.
- You want typed contracts and auto-generated docs to keep teams in sync.

⚠ Think twice when…

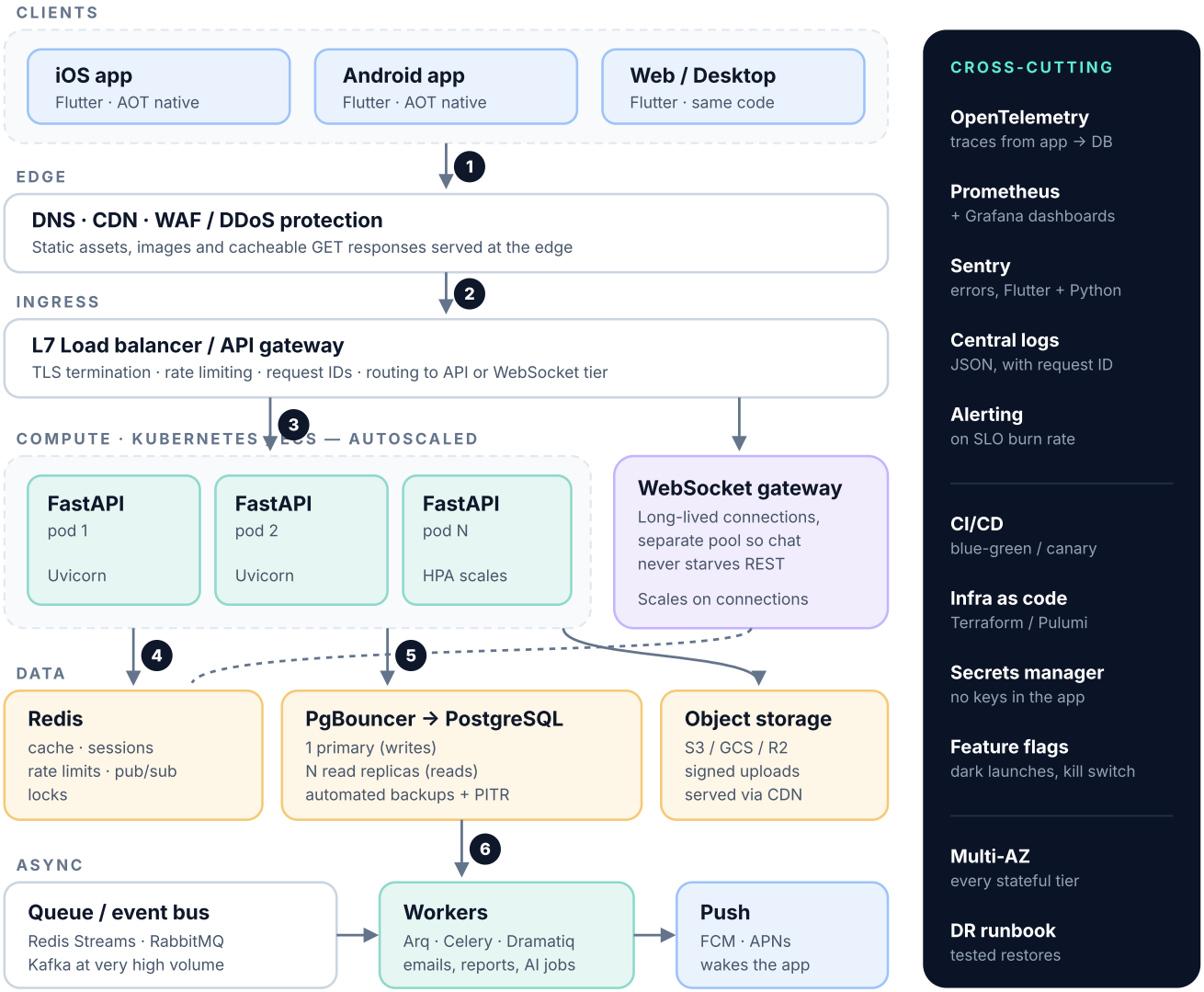
- Hot paths are **CPU-heavy** (video encoding, big number-crunching). Python's GIL means scale-out by processes — or move that service to Go/Rust.
- You depend on brand-new OS features on release day (widgets, AR, watch).
- Binary size is critical — Flutter adds a few MB of engine to each app.
- Your team is deeply invested in a JavaScript/TypeScript monorepo.

Verdict: for most enterprise products — marketplaces, fintech dashboards, field-force, health, logistics and SaaS companion apps — Flutter + FastAPI gives the best balance of speed, UI quality and AI-readiness. Scale problems are almost always **architecture** problems, not language problems; the next pages show how to solve them.

02 · REFERENCE ARCHITECTURE

The 100k-user blueprint

Every tier is stateless or horizontally scalable, and each request is answered from the cheapest layer that can answer it. Follow the numbered path.



1. **Edge first.** Cacheable responses and media never touch your servers.
2. **Gateway** terminates TLS, rate-limits and stamps a request ID for tracing.
3. **Stateless pods** — any pod can serve any user, so autoscaling is trivial.
4. **Redis** answers 80–95% of hot reads in under a millisecond.
5. **PgBouncer** multiplexes thousands of app connections onto a few hundred DB ones.
6. **Slow work goes async** — the API replies in ms; workers finish the job.

WHAT EACH TIER SCALES ON

TIER	SCALE SIGNAL	SCALES BY
FastAPI pods	CPU > 60% or RPS per pod above tested limit	Horizontal Pod Autoscaler — add pods in seconds
WebSocket gateway	Open connections per node	Add nodes; pub/sub keeps any node able to reach any user
PostgreSQL	Read CPU · write CPU · replica lag	Replicas for reads, bigger primary or partitioning for writes

03 · CAPACITY PLANNING

100k concurrent users ≠ 100k requests/sec

Concurrency is people with the app open. Load is what they actually send. Size infrastructure from the second number — here is a worked example you can rerun with your own analytics.

THE CORE FORMULA

Peak RPS = Concurrent users × Requests per user per minute ÷ 60 × Peak factor

100,000	concurrent users with the app open	input
6,667 RPS	average · each user makes ~4 API calls / minute	×4 ÷ 60
20,000 RPS	peak · launches, pushes, prime time	×3 peak
3,000 RPS	reach the database	85% cache hits
≈ 6,000 QPS	~2 queries per miss	

34 pods

API tier. 20,000 peak RPS ÷ 1,000 RPS per pod ÷ 60% target CPU = ~34 FastAPI pods (plus headroom for a zone failure).

1,360 → 200

DB connections. 34 pods × 4 workers × 10-conn pool = 1,360 client connections. PgBouncer funnels them into ~200 real Postgres connections.

6–10 nodes

WebSockets. If all 100k users hold a socket, at ~10–20k connections per tuned node you need 6–10 gateway nodes, plus spare.

Where your inputs come from

- **Requests / user / min** — product analytics or API logs from a pilot.
- **Peak factor** — ratio of busiest 5-minute window to the daily average (2–5× is common; flash sales and live events go higher).
- **RPS per pod** — measured by load-testing *your* endpoints, never borrowed from a benchmark.

Common sizing mistakes

- Sizing for the average and falling over at the first push notification.
- Scaling API pods while the **database** is the real ceiling.
- Forgetting that every new pod opens new DB connections.
- Ignoring **retry storms**: clients retrying without jitter multiply load during an incident.

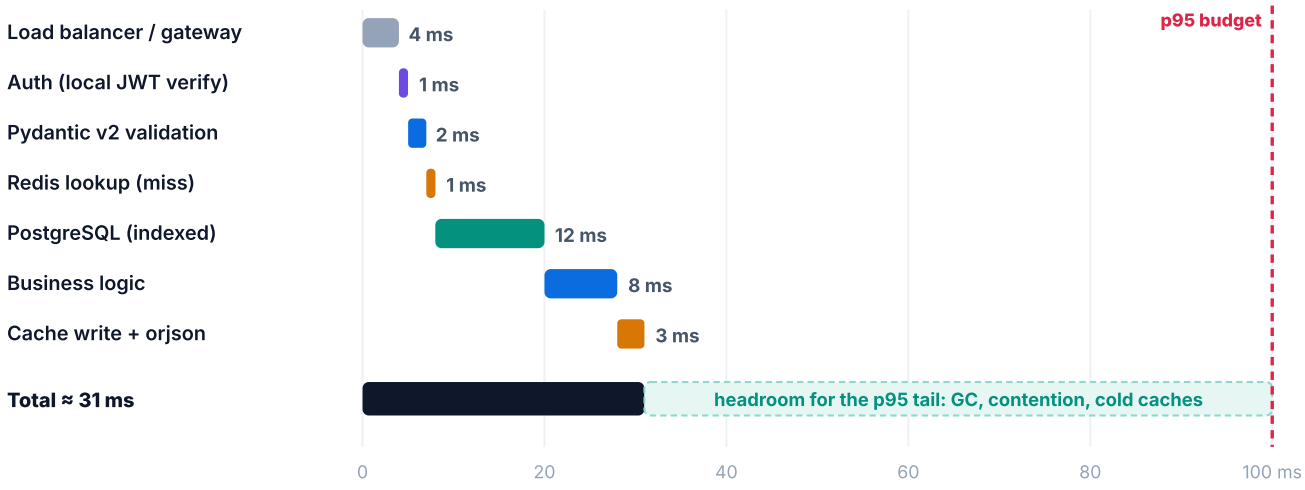
Illustrative assumptions: 4 req/user/min, 3× peak, 1,000 RPS per 2-vCPU pod for I/O-bound cached endpoints, 85% cache hit rate. Measure and replace with your numbers.

04 · PERFORMANCE BUDGET

Spending the 100 ms budget

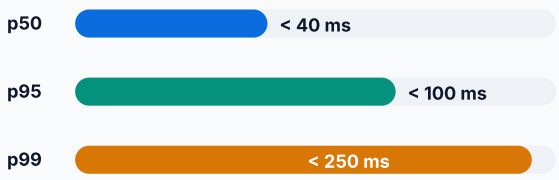
"Sub-100 ms" only means something with a percentile and a boundary. Our target: **p95 below 100 ms, measured at the load balancer**, so the network to the phone is excluded and the tail is included.

Anatomy of a typical read request (p50, cache miss)



Cache hit path: steps 5–7 disappear and the request completes in ~8–10 ms.

Percentile targets



Averages hide pain: at 20k RPS, 1% is 200 slow requests every second.

Top latency killers we see in audits

- **N+1 queries** — one list endpoint firing 50 queries.
- **Blocking calls in `async def`** (requests, sync DB drivers, file I/O) freezing the event loop.
- **Chatty screens** — 8 sequential API calls to render one view.
- **Unbounded payloads** — no pagination, full objects where 5 fields would do.
- **Cross-region hops** — API in one region, DB in another.
- **Cold caches** after deploys — no warm-up step.

Client-side perception: users feel total time, not server time. Add RTT (20–150 ms on mobile networks) and render time — then hide it with optimistic UI, skeleton screens, prefetching the next screen and serving cached data instantly while refreshing in the background.

05 · BACKEND PLAYBOOK

8 rules for a high-throughput FastAPI

1 Async end-to-end

asyncpg / SQLAlchemy async, `httpx.AsyncClient`, `redis.asyncio`. One blocking call in an `async def` stalls every request on that worker. If a library is sync, use a plain `def` route (FastAPI runs it in a threadpool).

2 Create pools once

DB engine, Redis and HTTP clients live in the `lifespan` handler and are reused. Never open a connection per request.

3 Serialize less, faster

Pydantic v2 response models that return only the fields the screen needs; `orjson` for encoding; gzip/br for large lists.

4 Cache-aside with jittered TTLs

Read Redis first, fall back to DB, write back with TTL ± random jitter so keys don't expire together. Add ETags so clients get cheap 304s.

5 Keyset pagination

`WHERE id < :cursor ORDER BY id DESC LIMIT 30` stays fast at page 10,000; `OFFSET` gets slower every page.

6 Offload anything slow

Emails, PDFs, image processing and LLM calls go to a queue. The API returns 202 + job ID; the app polls or gets a push.

7 Timeouts & limits everywhere

Per-call timeouts on DB and outbound HTTP, per-user rate limits in Redis, circuit breakers on third parties. Fail fast beats queueing forever.

8 Stateless pods

JWT auth, sessions in Redis, files in object storage. A pod can die at any second and no user notices.

APP/MAIN.PY — PRODUCTION BASELINE

```python
from contextlib import asynccontextmanager
from fastapi import FastAPI, Response
from fastapi.responses import ORJSONResponse
from redis.asyncio import Redis
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker
import orjson, random

engine = create_async_engine(DB_URL, pool_size=10, max_overflow=5, pool_pre_ping=True)
Session = async_sessionmaker(engine, expire_on_commit=False)
redis = Redis.from_url(REDIS_URL)

@asynccontextmanager
async def lifespan(app: FastAPI):
    yield                                  # pools are ready before the first request
    await engine.dispose(); await redis.aclose()

app = FastAPI(lifespan=lifespan, default_response_class=ORJSONResponse)

@app.get("/v1/products/{pid}", response_model=ProductOut)
async def get_product(pid: int):
    key = f"product:{pid}"
    if cached := await redis.get(key):           # ~1 ms hit path
        return Response(cached, media_type="application/json")
    async with Session() as db:
        product = await repo.get_product(db, pid)     # indexed, selects only needed columns
    body = orjson.dumps(ProductOut.model_validate(product).model_dump(mode="json"))
    await redis.set(key, body, ex=300 + random.randint(0, 60))  # TTL jitter
    return Response(body, media_type="application/json")
```

Running it

```
uvicorn app.main:app --host 0.0.0.0 --workers 2 \
  --loop uvloop --http httptools --proxy-headers
```

On Kubernetes, keep 1–2 workers per pod and scale **pods** with the HPA — cleaner health checks and failure isolation than one giant process.

Recommended libraries

DB SQLAlchemy 2 async / asyncpg · **Migrations** Alembic · **Cache** redis-py asyncio · **Jobs** Arq / Celery · **HTTP** httpx · **Auth** PyJWT / Authlib · **Settings** pydantic-settings · **Tests** pytest + httpx

06 · CLIENT PLAYBOOK

A Flutter app that stays fast at scale

Scalability is a client concern too: a well-built app sends fewer, smaller, smarter requests — and never drops a frame while it waits.

16.7 ms
frame budget at **60 Hz** — build, layout and paint must all fit

8.3 ms
frame budget at **120 Hz** on ProMotion and high-refresh Android

0
heavy JSON parsing or crypto on the UI isolate. Ever.

1. **Move heavy work off the UI isolate**
   Decode large responses with `Isolate.run()` / `compute()`. Parsing a 1 MB list on the main isolate is a guaranteed jank spike.

2. **One resilient HTTP layer**
   Dio with interceptors for auth refresh, request IDs, timeouts and **retry with exponential backoff + jitter** (only for idempotent calls).

3. **Offline-first, cache-then-network**
   Persist screens in SQLite (Drift) and show them instantly; refresh in the background. Users see data in <50 ms even on a bad network.

4. **Lazy everything**
   `ListView.builder` / slivers with cursor pagination; request page N+1 as the user nears the end.

5. **Rebuild only what changed**
   `const` widgets, granular state (Riverpod or Bloc), `select` to watch single fields, `RepaintBoundary` for animated areas.

6. **Ship right-sized images**
   CDN-resized WebP/AVIF for each device width, with `cached_network_image` and `cacheWidth` to cap decoded memory.

7. **Debounce & coalesce**
   Search-as-you-type waits ~300 ms; duplicate in-flight requests share one future. Fewer requests = cheaper backend.

8. **Measure in profile mode**
   Flutter DevTools timeline on real low-end devices; track slow and frozen frames in production with Firebase Performance or Sentry.

LIB/DATA/API_CLIENT.DART — RESILIENT NETWORKING

```dart
final dio = Dio(BaseOptions(
  baseUrl: 'https://api.example.com/v1',
  connectTimeout: const Duration(seconds: 5),
  receiveTimeout: const Duration(seconds: 10),
));

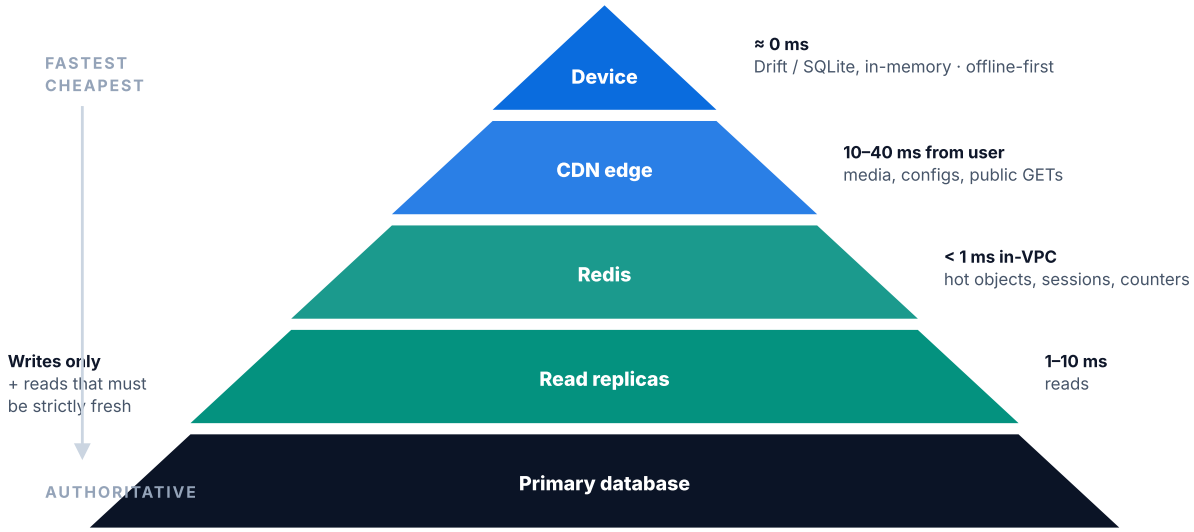
void configure() {
  dio.interceptors.addAll([
    AuthInterceptor(),                          // attaches + refreshes JWT
    RetryInterceptor(dio: dio, retries: 3, retryDelays: const [  // dio_smart_retry
      Duration(milliseconds: 300), Duration(seconds: 1), Duration(seconds: 3),
    ]),
  ]);
}

Future<List<Product>> fetchProducts({String? cursor}) async {
  final res = await dio.get<String>('/products',
      queryParameters: {'cursor': cursor, 'limit': 30});
  // Decode on a background isolate so the UI keeps its 8–16 ms frame budget
  return Isolate.run(() => parseProducts(res.data!));
}
```

07 · CACHING & DATA LAYER

The cache pyramid

The fastest request is the one that never reaches your database. Each layer below absorbs traffic so the one beneath it only sees what it must.



CACHE-INVALIDATION STRATEGIES

STRATEGY	HOW IT WORKS	USE FOR
TTL + jitter	Keys expire after N seconds ± random offset	Catalogues, feeds, anything tolerating seconds of staleness
Write-through / delete-on-write	Service updates DB, then deletes or rewrites the key	Profiles, carts, settings
Event-driven	DB change events (outbox / CDC) trigger invalidation across services	Multi-service systems, search indexes
Stampede protection	Single-flight lock or early probabilistic refresh so only one request rebuilds a hot key	Home screens, trending lists, prices

PostgreSQL at 100k users

- **PgBouncer** in transaction mode in front of every database.
- **EXPLAIN ANALYZE** every hot query; index for its exact filters & sort.
- **Route reads** to replicas; read-your-own-writes from primary.
- **Partition** event/log tables by time; archive cold rows.
- `statement_timeout` and short transactions — no long locks.

Targets to monitor

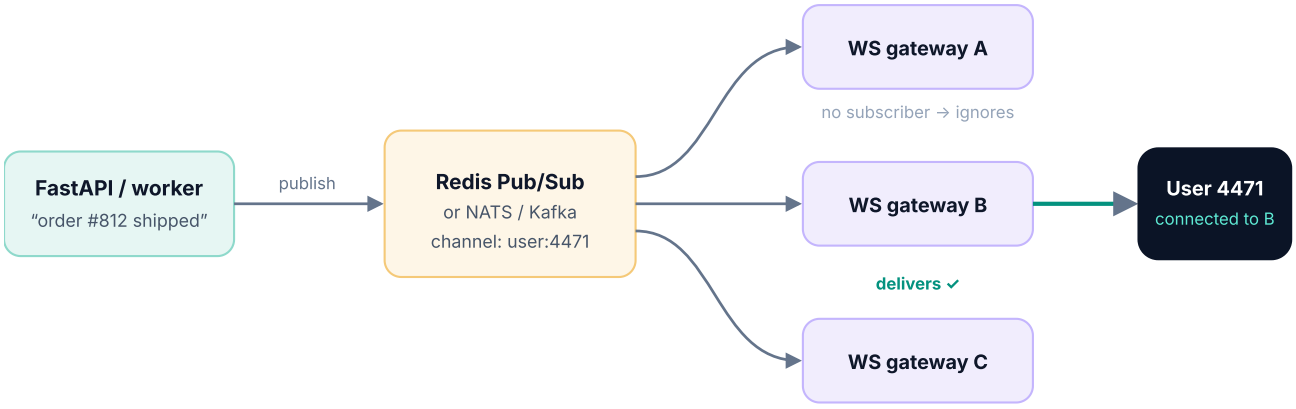
- Redis hit rate **≥ 85%** on hot endpoints
- DB CPU **< 60%** at peak (headroom for failover)
- Replica lag **< 1 s**
- Connection pool wait time **≈ 0**
- Slowest 10 queries reviewed **every sprint**

08 · REAL-TIME & ASYNC

Live updates without melting servers

Chat, tracking, notifications and live dashboards are where naïve designs break first. The fix: decouple who *produces* an event from which server *holds* the user's connection.

WebSocket fan-out across many servers



Each gateway subscribes only to channels of the users connected to it. Producers never need to know which server a user is on — so you can add gateways freely.

PICK THE RIGHT TRANSPORT

TRANSPORT	BEST FOR	WATCH OUT FOR
Polling + ETag	Data changing every few minutes; simplest and cache-friendly	Wasted calls if interval is too short
Server-Sent Events	One-way live feeds, scores, progress bars	Server → client only
WebSockets	Chat, collaboration, live tracking (two-way, low latency)	Stateful: separate tier, heartbeat, reconnect with backoff
Push (FCM / APNs)	Anything when the app is backgrounded or closed	OS kills background sockets — push is the only reliable wake-up

The transactional outbox

Write the business change **and** an `outbox` row in one DB transaction; a relay publishes outbox rows to the queue. No more "order saved but notification lost" when a pod dies mid-request.

Worker hygiene

- Jobs are **idempotent** — safe to run twice.
- Retries with backoff, then a **dead-letter queue**.
- Separate queues for fast vs. slow jobs (e.g. OTP vs. reports).
- Autoscale workers on **queue depth**, not CPU.

09 · SCALING ROADMAP

Scale in stages — not all on day one

Over-engineering burns budget; under-engineering burns users. Design for 100k from the start (stateless, cached, async), but only *deploy* the infrastructure each stage needs.

1 — 2 — 3

STAGE 1 · LAUNCH

≤ 5k CCU

- Single region, 2–3 API containers behind a managed LB
- Managed PostgreSQL with automated backups
- Managed Redis for cache + rate limits
- CDN for media; Sentry from day one
- CI/CD with automated tests and migrations
- Offline-first Flutter app with generated API client

Move on when:
p95 > 80 ms at peak, or DB CPU regularly > 50%

STAGE 2 · GROWTH

5k–30k CCU

- Kubernetes / ECS with autoscaling on CPU + RPS
- PgBouncer + 1–2 read replicas; read/write split
- Queue + workers for all slow work
- OpenTelemetry tracing, SLO dashboards, on-call
- Load tests in CI against a staging copy
- Feature flags and canary releases

Move on when:
Primary write load > 50%, or socket count drives node count

STAGE 3 · SCALE

100k+ CCU

- Multi-AZ everything; tested zone-failure drills
- Dedicated WebSocket gateway tier + pub/sub
- Table partitioning; split hot domains into services
- Multi-region CDN; regional read replicas
- Chaos tests, DR restores, capacity reviews quarterly
- Cost dashboards per service and per feature

Next frontier:
Active-active multi-region, sharding, edge compute

Design-for-scale decisions that are cheap now and expensive later

- API versioning (`/v1`) and a forced-update mechanism in the app
- UUID / ULID primary keys, not guessable integers
- Cursor pagination on every list endpoint
- Request IDs propagated from Flutter to the database
- All config via environment; zero state on disk
- Idempotency keys on payments and order creation

TYPICAL TEAM SHAPE PER STAGE

Launch
2 Flutter · 2 FastAPI · 1 QA · shared DevOps · 1 designer

Growth
3–4 Flutter · 3–4 backend · 1 DevOps/SRE · 2 QA (incl. automation)

Scale
Squads per domain · platform/SRE team · dedicated data & security owners

10 · OBSERVABILITY & LOAD TESTING

You can't scale what you can't see

LATENCY
p95
per endpoint, success and error paths tracked separately

TRAFFIC
RPS
per route, plus active WebSocket connections

ERRORS
5xx %
server errors, timeouts and app crash-free sessions

SATURATION
CPU · pool
CPU, memory, DB pool wait, queue depth, replica lag

EXAMPLE SERVICE-LEVEL OBJECTIVES

SLO	TARGET	ERROR BUDGET / 30 DAYS	ALERT WHEN
API availability	99.95% of requests succeed	~21.6 minutes	Budget burning 14× faster than plan for 1 h
API latency	95% of requests < 100 ms	5% may be slower	p95 > 100 ms for 10 min
App stability	≥ 99.8% crash-free sessions	0.2% of sessions	New release drops below target
Async jobs	99% of notifications sent < 5 s	1% of jobs	Queue depth rising 15 min

A load-test plan that finds the ceiling before your users do



Tooling we recommend

Load: k6 or Locust (Python, shares models with FastAPI) · **Tracing:** OpenTelemetry → Grafana Tempo / Jaeger · **Metrics:** Prometheus + Grafana · **Errors:** Sentry (Flutter + Python SDKs) · **Mobile perf:** Firebase Performance / Crashlytics

Rules for realistic tests

- Replay real traffic mix, not one endpoint
- Production-sized data, not an empty DB
- Generate load from outside the cluster

11 · LAUNCH READINESS

The 40-point go-live checklist

Print it, tick it, and don't launch until every box in the first two columns is checked.

Security — MUST

- [ ] TLS everywhere; HSTS; certificate pinning for high-risk apps
- [ ] Short-lived access tokens + rotating refresh tokens
- [ ] Tokens in Keychain / Keystore (flutter_secure_storage)
- [ ] No secrets or API keys compiled into the app
- [ ] Per-user and per-IP rate limits; WAF rules on auth routes
- [ ] Object-level authorization checked on every endpoint
- [ ] Input validated by Pydantic; queries parameterized
- [ ] Dependencies scanned in CI (pip-audit, OSV)
- [ ] Dart code obfuscated in release builds
- [ ] OWASP API & MASVS review completed

Reliability — MUST

- [ ] Liveness + readiness probes on every pod
- [ ] Graceful shutdown drains in-flight requests
- [ ] Multi-AZ database with automatic failover tested
- [ ] Backups restored successfully in the last 30 days
- [ ] Timeouts on every DB, cache and outbound call
- [ ] Circuit breakers on third-party dependencies
- [ ] Idempotency keys on payments and order creation
- [ ] Zero-downtime, backward-compatible DB migrations
- [ ] Rollback in one click; canary on new releases
- [ ] Remote kill switch / forced-update in the app

Performance — SHOULD

- [ ] Load test passed at 1.5× expected peak
- [ ] p95 < 100 ms on top 20 endpoints
- [ ] No N+1 queries (checked with query logging)
- [ ] Indexes cover every hot query plan
- [ ] Cache hit rate ≥ 85% on hot reads
- [ ] Cursor pagination on all list endpoints
- [ ] Images resized + served via CDN
- [ ] Large JSON parsed off the UI isolate
- [ ] No janky frames on a low-end Android device
- [ ] Autoscaling verified under a spike test

Operations — SHOULD

- [ ] SLOs defined; burn-rate alerts routed to on-call
- [ ] Dashboards for the four golden signals
- [ ] Distributed tracing app → API → DB
- [ ] Structured logs with request ID correlation
- [ ] Crash & ANR reporting in the Flutter app
- [ ] Runbooks for the top 5 failure modes
- [ ] Infrastructure defined as code
- [ ] Cost alerts and per-service cost tags
- [ ] Store listing, privacy labels and data-safety forms
- [ ] Status page and incident comms template

Steigern Tech

STEIGERNTECH.COM

BUILD IT WITH US

Planning a Flutter + FastAPI product? Let's architect it for scale.

Steigern Tech designs, builds and scales cross-platform apps and AI-ready backends — from a first MVP to systems serving tens of thousands of daily users.

Architecture & scalability review

We audit your app, API and database against this kit and hand you a prioritized fix list with latency and cost impact.

Flutter app development

iOS, Android, web and desktop from one codebase — offline-first, accessible and tuned for 60/120 fps.

FastAPI backends & cloud

Async APIs, PostgreSQL, Redis, queues, Kubernetes, CI/CD and observability — production-ready from day one.

AI implementation

LLM features, search, recommendations and automation wired directly into your Python backend.

WHATSAPP

+91 94275 72571

EMAIL

info@steigerntech.com

WEBSITE

steigerntech.com

© 2026 Steigern Tech LLP. Figures are illustrative engineering targets — validate with your own load tests.

Flutter is a trademark of Google LLC. FastAPI is an open-source project by Sebastián Ramírez.