Kyiv School of Economics
Software Engineering & Business Analysis — Capstone Project

MatchUp

Design and Implementation of a Mobile-First Matchmaking Platform for the DanceSport Community

Author: Ivan Solomatin

Supervisor: Ihor Pysmennyi, ipysmennyi@kse.org.ua

Expert Reviewer: _____

Submission date: **June 3, 2026**

Kyiv, 2026

Academic Integrity Statement

I, the undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures, and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received is acknowledged.

Place, date: Kyiv, June 3, 2026

Signature: Ivan Solomatin

Abstract

Competitive and social dance — ballroom, Latin, and DanceSport — is a partner-dependent discipline in which progress, competition entry, and continuity all depend on having a compatible partner, yet the community has no purpose-built infrastructure for finding one. Partner search runs entirely on word of mouth, Instagram and Facebook posts, Telegram communities, trainer referrals, and club bulletin boards, leaving dancers out of the sport for months at a time. This thesis documents the complete design and implementation of MatchUp, a mobile-first matchmaking platform that replaces that fragmented process with a structured, verified, media-rich, and filterable system built around the criteria dancers actually use to evaluate a partner.

The work followed a research-first engineering process. The problem and its scale were established through a market-research phase combining secondary analysis of existing platforms and federation data with primary research across the Ukrainian dance community — semi-structured interviews, an online survey, and field observation at a national competition. That evidence was translated into an explicit set of functional and non-functional requirements, which in turn drove an architecture methodology centred on three principles: a modular-monolith backend that preserves strict domain boundaries while deferring the operational cost of distribution; schema-first development with end-to-end type-safety carried from the database to the API contract; and an observability-first implementation in which metrics, traces, logs, errors, and product analytics are treated as first-class concerns. Implementation proceeded over three month-long sprints — foundations and identity, the core matching loop, and the surrounding ecosystem of clubs, safety, monetisation, and mobile.

The delivered system is a live, internet-accessible product spanning fourteen domain modules. Its technical contributions include a two-sided recommendation engine whose three-tier candidate pipeline guarantees a non-empty feed in a low-density market while collecting the feature data for a future learned ranking model; a single SvelteKit codebase that ships to both web and a native iOS application through Capacitor; a match-triggered in-app messenger; and a complete observability stack (Prometheus, Grafana, Tempo, Loki, Sentry, and PostHog). The principal engineering challenges — selecting a low-technical-debt stack, designing for a cold-start market, and architecting the messenger — are documented with the decisions and trade-offs that resolved them, alongside the full user journey and end-to-end data flow that connect the product experience to the system internals.

Verification was demonstrated through an implemented smoke-testing strategy exercising every core flow against a seeded database, the continuous-integration pipeline, and deployment checks; validation was framed against measurable product metrics — retention, conversion, profile completion, and feed engagement — and a forward experimentation framework, shifting the emphasis from “the software works” to “users want the product.” The business case is viable: at a Ukrainian launch price of five US dollars per month against modest operating costs, break-even is reached at roughly 165 paying subscribers, and the addressable market scales to a substantial international opportunity. The thesis demonstrates that a research-first approach to software engineering can produce both a validated problem case and an architecture engineered for the specific economics of its market.

Keywords: Software Engineering, MatchUp, DanceSport, mobile application, recommendation system, modular monolith, schema-first development, end-to-end type safety, SvelteKit, Capacitor, observability, freemium model.

Contents

Chapter 1. Introduction

Competitive and social dance is a paired discipline practised by children and adults, amateurs and professionals alike. Although dancers can train alone, meaningful progress, competition entry, and skill development all depend on a compatible partner — matched across age, height, skill level, competitive goal, dance programme, club affiliation, and location. In Ukraine the community is large and active, supported by a national federation, hundreds of clubs, and a year-round competition calendar, yet the process of finding that partner has never been digitised. MatchUp addresses this gap directly: a mobile-first matchmaking platform that centralises partner search, applies structured two-sided filtering, presents verified profiles, and supports faster, more reliable matching for a globally active but digitally underserved community. This thesis documents the project end to end — from the research that established and quantified the problem, through the architecture and implementation of a working system, to its verification, validation, and business justification.

1.1 Project Objectives

The primary objectives of this capstone project are:

1. To validate, through primary and secondary research, that partner search is a real, severe, and recurring problem for the DanceSport community, and to translate that research into concrete product requirements.
2. To design and implement a mobile-first matchmaking application built around the decision criteria dancers actually use to evaluate a partner.
3. To implement a recommendation engine that produces high-quality, two-sided matches while remaining useful in a low-density market.
4. To build a maintainable, type-safe, observable system that a lean team can operate and that has a clear path to scale.
5. To deploy the application as a live, internet-accessible product suitable for real user validation and iOS release.

1.2 Relevance

1.2.1 Market Relevance

The broader application market has moved decisively toward niche, interest-based matching — from sports-partner apps to hobby communities — reflecting a user expectation of finding others who share a specific context, skill level, and goal rather than a general interest. DanceSport, a partner-dependent discipline with exactly this structural matching problem, remains conspicuously underserved by any modern, mobile-first tool. Primary research confirms strong, unmet demand within a community that is large, active, and globally connected, and the economic analysis in Chapter 2 establishes that the opportunity is both real and reachable at a viable price point. The market relevance is therefore immediate: a clearly defined audience with a painful, recurring need and no adequate solution.

1.2.2 Technical Relevance

The project demonstrates modern software-engineering practice end to end: a modular-monolith architecture with strict domain boundaries, schema-first development with type-safety carried from the database to the API contract, a single client codebase serving both web and native iOS, deliberately swappable and fail-open third-party integrations, and a production-grade observability stack — all delivered by a single developer within a constrained timeframe and deployed as a live system.

1.2.3 Academic Relevance

The work integrates knowledge from across the Software Engineering and Business Analysis curriculum — software architecture, database design, web and mobile development, DevOps and observability, user-experience design, and market research — and applies it to a single, real-world problem from inception to deployment, producing a documented, evidence-based engineering record.

1.3 Object and Subject of Research

1.3.1 Object of Research

The object of research is the partner-search processes within the DanceSport ecosystem — how dancers, parents, trainers, and clubs currently discover, evaluate, and form partnerships, and the inefficiencies those processes impose.

1.3.2 Subject of Research

The subject of research is the methods, recommendation systems, software architectures, and digital platforms that enable efficient partner discovery and matching — specifically, how a structured, two-sided, mobile-first system can replace the fragmented status quo and remain viable in a low-density market.

1.4 Scientific and Technical Contribution

This thesis makes a combined scientific and technical contribution:

- **Partner matching in low-density markets** — a three-tier candidate pipeline that degrades gracefully to guarantee a non-empty feed, addressing the cold-start problem that defeats naive matching in geographically clustered communities.
- **Recommendation engine design** — a single nullable-parameter SQL query that powers all matching tiers, instrumented from day one with a like-feature log that prepares a path from rule-based ranking to a learned model without re-instrumentation.
- **Reusable architecture** — a modular monolith whose domain boundaries are real (each module owns its schema slice and generated queries), giving microservice-grade separation with a fraction of the operational cost and a defined extraction path.
- **Cross-platform delivery** — a single SvelteKit codebase shipped to both web and native iOS via Capacitor, eliminating an entire class of duplication for a lean team.
- **Observability-first implementation** — metrics, traces, logs, error reporting, and product analytics treated as first-class concerns and cross-linked from the outset, which is uncommon at this stage of a product.

1.5 Methodology

The project combined a structured discovery phase with iterative agile delivery. Discovery established the problem and requirements through secondary analysis of existing platforms and market data and primary research within the Ukrainian dance community. Implementation then proceeded over three month-long sprints, each opening with goal-setting and closing with a review and retrospective. This cadence is documented in detail, with results and retrospectives, in Chapter 4.

1.6 Structure of this Thesis

- **Chapter 2 — Domain Research and Analysis** establishes the problem, describes how dancers search today, surveys existing solutions, sizes the market through a TAM/SAM/SOM funnel, derives the requirements, and presents the economic justification.
- **Chapter 3 — System Design and Architecture** details the architectural decisions and ADRs, the technology-stack evaluation, the recommendation engine, the data layer, integrations, deployment, security, and observability with SLIs/SLOs.
- **Chapter 4 — User Journey, Implementation, and Engineering** documents the end-to-end user journey and data flow, the development methodology, the three sprints with retrospectives, the critical code implementations, and the CI pipeline.
- **Chapter 5 — Verification and Validation** demonstrates correctness through the implemented smoke-testing strategy and CI, then frames market usefulness through measurable product metrics and a forward experimentation plan.
- **Chapter 6 — Conclusion** reflects on the academic, technical, and market contributions and the future roadmap.

1.7 Chapter Summary

This chapter set out the objectives of the MatchUp capstone and located the work within its market, technical, and academic context, with market relevance leading because the project's defining constraint is the economics of a real, underserved community. It defined the object of research as the partner-search processes of the DanceSport ecosystem and the subject as the methods and architectures that make efficient matching possible, and it consolidated the project's scientific and technical contribution around low-density matching, recommendation design, reusable architecture, cross-platform delivery, and observability-first engineering. The methodology — research-first discovery followed by three agile sprints — frames everything that follows. The next chapter grounds these claims in evidence, establishing the problem's scale and the requirements that drive the system design.

Chapter 2. Domain Research and Analysis

This chapter establishes why MatchUp is needed and what it must do. It defines the partner-search problem and how dancers solve it today, evaluates existing solutions and the gap between them, sizes the opportunity through a market funnel, derives the requirements that drive the architecture in Chapter 3, and presents the economic case for the product.

2.1 Research Questions and Methodology

The investigation was guided by a focused set of questions: What is the scale of the DanceSport community in Ukraine and globally? How do dancers currently search for a partner, and where does that process fail? What does the search cost in time, money, and motivation? Which segments experience the most friction? And what features would carry the highest value in a digital matching solution? The method combined secondary research — analysis of existing platforms, federation data, and the competition calendar — with primary research: seven semi-structured interviews across stakeholder groups, an online survey of the Ukrainian dance community, and field observation at a national competition in Kyiv.

2.2 How Dancers Search for Partners Today

Before evaluating products, it is essential to understand the behaviour they would replace. Partner search today is entirely manual and spread across channels never designed for it. Dancers and their parents post on Instagram and scroll hashtags; they join Facebook groups and Telegram communities and watch for announcements; they rely on trainer referrals and club recommendations; they network at competitions; and they fall back on personal contacts. None of these channels offers structured filtering on the criteria that determine compatibility, none verifies the information shared before a trial meeting, and none centralises the people actively searching. The result is a slow, repetitive, emotionally costly process in which much of the effort is spent screening irrelevant candidates and discovering mismatches too late.

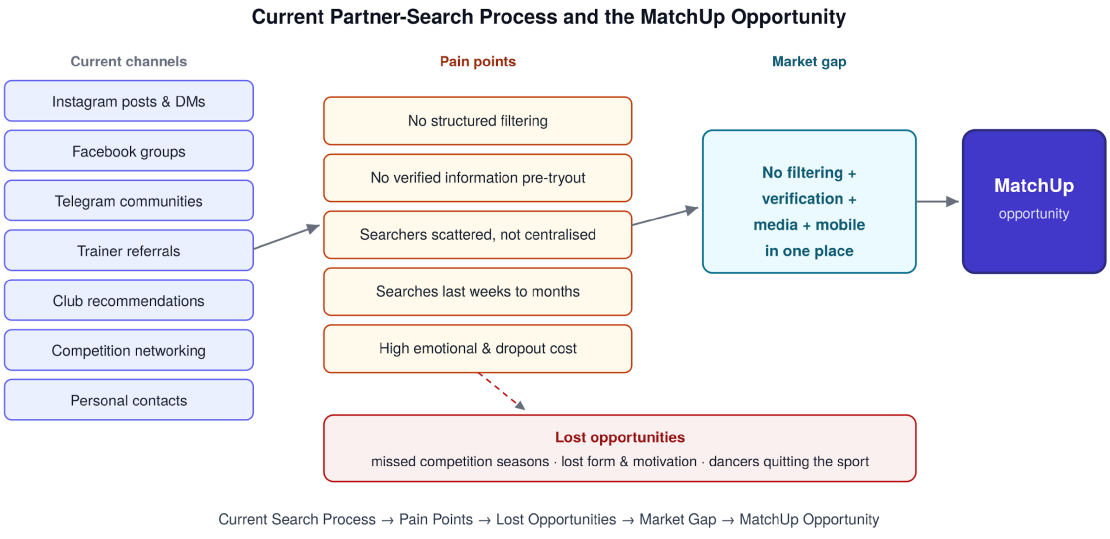

Figure 2.1. How dancers search today: fragmented channels lead to recurring pain points and lost opportunities, exposing the market gap MatchUp is built to close.

2.3 The Partner-Search Problem

Primary research found the problem to be frequent, structurally inevitable, and severe. Dancers change partners repeatedly over a career — driven by growth-related height mismatches, programme divergence, injury, motivational gaps, and, in the Ukrainian context, war-related relocation — and each search routinely lasts from several weeks to more than three months. The information available before a trial is frequently inaccurate, producing wasted try-outs and significant emotional cost. The survey confirmed the severity quantitatively: the large majority of respondents had searched for a partner, rated the process as difficult, and a substantial share had considered leaving the sport entirely as a result. Crucially, although female dancers face higher friction owing to a structural scarcity of male partners, the data refuted the assumption that male dancers face no difficulty — both sides experience the problem. Two further insights shaped the product: parents of junior dancers act as primary decision-makers and need simple, trustworthy tools, which motivates a first-class parent account type; and the same stakeholders independently raised adjacent unmet needs — a costume marketplace, an events calendar, and remote viewing for families — positioning MatchUp as the foundation for a broader community platform.

2.4 Market Context and Scale

The DanceSport community is large, active, and globally connected. Ukraine alone has several hundred registered clubs and an estimated tens of thousands of active dancers, with a dense year-round competition calendar and institutional depth extending to dedicated higher-education programmes. Internationally, the World DanceSport Federation unites ninety-nine national member federations across five continents. The market is sized below as a funnel from the total ecosystem, through globally active dancers, to the reachable Ukrainian-language core at launch; the assumptions and calculations are stated in the figure.

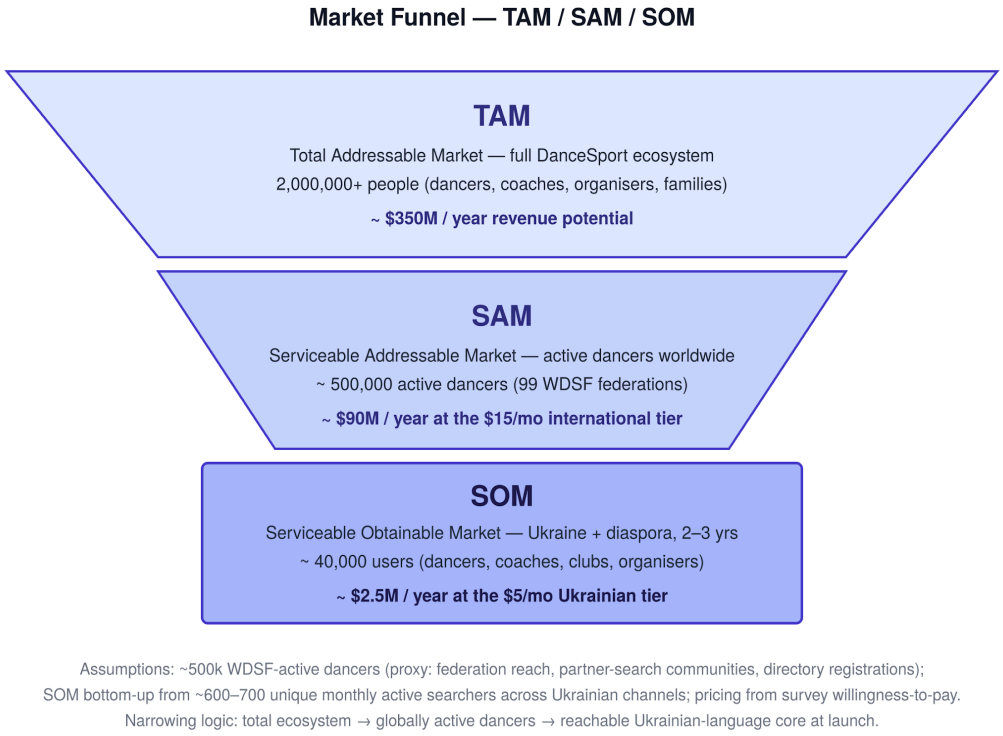

Figure 2.2. Market funnel. TAM is the full DanceSport ecosystem; SAM the active dancers worldwide; SOM the Ukrainian + diaspora core reachable at launch. Assumptions and calculations are annotated.

2.5 Competitive Analysis and Gap

Existing solutions divide into three categories: social platforms (Instagram, Facebook, Telegram), which offer activity and media but no structured filtering, verification, or organisation; legacy web directories, which offer formal listings but suffer outdated design, low activity, and minimal filtering; and offline channels (trainer networks, word of mouth), which remain manual and opaque. A recent multi-sport entrant shows the market recognises the pain but dilutes supply across many sports rather than concentrating it within dance. Mapping these shortcomings against the user needs surfaced in research makes the gap precise: dancers need filtering on the real compatibility criteria (no channel provides it), verified pre-tryout information (none provides it), and a single place where active searchers gather (none provides it). No existing solution combines high activity, two-sided filtering, verification, media-rich profiles, and a mobile-first experience — the gap MatchUp fills.

Solution	Structured filters	Activity	Verified	Mobile-first
Instagram / Facebook / Telegram	None	High	No	No
Legacy web directories	Minimal	Low–Medium	No	No
International paid directory	Yes	Medium	No	No (web)
Multi-sport partner app	Yes	Low (new)	Partial	Yes (diluted)

Solution	Structured filters	Activity	Verified	Mobile-first
MatchUp	Two-sided, full	Target	Yes	Yes

Table 2.1. Competitor shortcomings mapped against the identified user needs.

2.6 Functional and Non-Functional Requirements

The research translates into the following requirements, which drive every architectural decision in Chapter 3 and are verified in Chapter 5.

FR	Requirement
FR1	Identity and profiles — email/password registration and login, Google sign-in, OTP email verification, password reset; four role-based account types (dancer, parent, trainer, club); two-sided profile and preference data; photo uploads.
FR2	Discovery and matching — a recommendation feed, swipe like/dislike, mutual-match detection, and a candidate pipeline that never returns an empty feed.
FR3	Communication — a polymorphic 1:1 chat (direct dancer-to-dancer threads created on a mutual match, plus user-to-club threads), with read receipts, keyword-based safety filtering, and per-message reporting.
FR4	Supply-side discovery — clubs (importable from a Google Maps link), trainers, and an interactive map with the same filtering logic as the feed.
FR5	Trust and safety — block, report, and admin ban (enforced at the authentication layer); per-message reporting with admin review and soft-delete; optional NSFW scanning of avatars; verified, server-side OAuth token validation.
FR6	Monetisation — freemium subscription gating, entitlement management, and supply-side visibility upsell on the roadmap.

Table 2.2. Functional requirements.

NFR	Requirement
NFR1	Read performance — the read-dominated feed-assembly path is the optimisation target.
NFR2	Reliability — a guaranteed non-empty feed and swappable, fail-open third-party integrations.
NFR3	Scalability — clean domain boundaries, per-city replication by configuration, and vertical headroom.
NFR4	Maintainability — type-safety from database to API contract and an explicit, readable composition root.
NFR5	Security — nonce-based session invalidation, rate limiting, CORS, moderation primitives, and image moderation.
NFR6	Observability — metrics, traces, logs, error reporting, and product analytics as first-class concerns.
NFR7	Localisation — Ukrainian by default with English available, and localised error surfacing.

Table 2.3. Non-functional requirements.

2.7 Economic Analysis

2.7.1 Subscription Model

MatchUp follows a freemium model that minimises onboarding friction and monetises once users see value. Browsing and basic filtering are free to build the profile density on which matching depends; the advanced two-sided filters and direct messaging that actually accelerate a search sit behind a subscription. The Ukrainian launch price is five US dollars per month, chosen because it is aligned with survey willingness-to-pay responses, lower than the alternatives, presents a low entry barrier, and therefore encourages market adoption; subscriptions are wired through RevenueCat.

2.7.2 Infrastructure Costs

Item	Frequency	Cost
Hosting (Hetzner / DigitalOcean)	monthly	~$20
Domain	yearly	~$15 (~$1.25/mo)
Cloudflare tunnel	monthly	Free
Junior engineer support	monthly	~$800
Total operating cost	monthly	~$821

Table 2.4. Monthly operating cost at the early stage.

2.7.3 Break-Even Analysis

With a monthly operating cost of approximately $821 and a subscription price of $5 per month, the break-even point is 821 / 5 = 164.2, rounded to 165 paying subscribers. Because fixed costs are low and subscription revenue scales linearly with subscribers at this stage, profitability follows quickly once the break-even threshold is crossed, as the sensitivity scenarios below show.

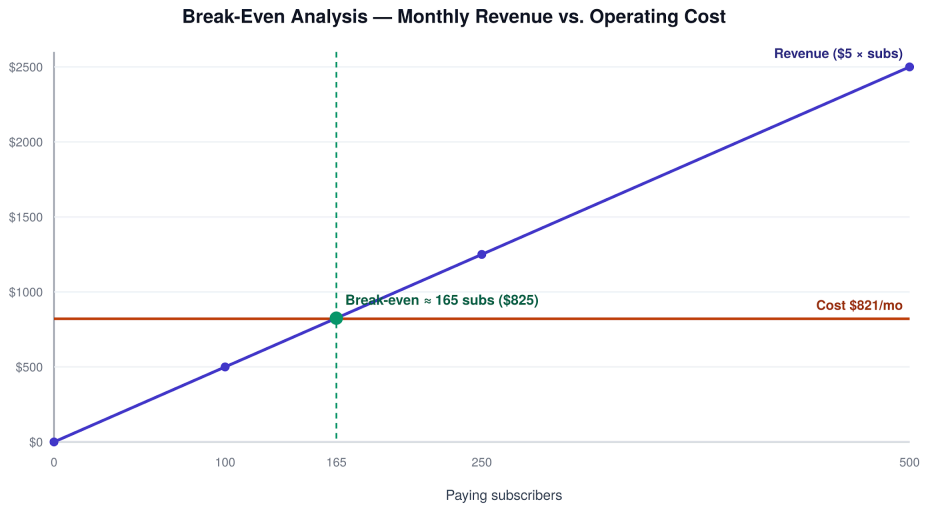

Figure 2.3. Break-even analysis. Monthly subscription revenue ($5 per subscriber) crosses the ~$821 operating-cost line at approximately 165 paying subscribers.

Subscribers	Revenue / mo	Cost / mo	Result
100	$500	$821	-$321 (below break-even)
165	$825	$821	≈ $4 (break-even)
250	$1,250	$821	+$429 profit
500	$2,500	$821	+$1,679 profit

Table 2.5. Sensitivity analysis: monthly revenue versus cost across subscriber scenarios.

2.7.4 Addressable Market in Revenue Terms

Tier	Audience (approx.)	Annual gross revenue potential
SOM	40,000 (Ukraine + diaspora)	~$2.5M / year
SAM	500,000 (active dancers globally)	~$90M / year
TAM	2,000,000+ (incl. coaches, organisers, families)	~$350M / year

Table 2.6. Addressable market, computed bottom-up at the two pricing tiers.

2.8 Chapter Summary

This chapter set out to establish whether a real problem exists and what a solution must do. It found, through primary and secondary research, that partner search is a severe and recurring problem solved today only through fragmented, manual channels that provide neither filtering, verification, nor a central marketplace of searchers. It sized the opportunity through a TAM/SAM/SOM funnel and showed it is viable at a $5 Ukrainian price point, with break-even at roughly 165 subscribers. The competitor analysis mapped each existing shortcoming to a concrete user need, and those needs were distilled into six functional and seven non-functional requirements. The engineering implication is direct: the system must optimise for read-heavy matching, guarantee a non-empty feed in a low-density market, and remain cheap to operate and replicate — the precise constraints the architecture in the next chapter is designed to satisfy.

Chapter 3. System Design and Architecture

The MatchUp architecture is a direct answer to the constraints established in Chapter 2. Matching is a network-density problem rather than a national-coverage one: value appears the moment a single city reaches enough active dancers that the next person to open the feed finds a real candidate, which places liquidity and a never-empty feed above raw scale. The product also grows along a fixed social chain — dancers pull in parents, parents and dancers anchor trainers, trainers anchor clubs, and clubs host the events module that drives retention and supply-side revenue — so the system must be cheap to replicate per city and structurally prepared for an entity model far larger than dancer-to-dancer swiping. The design therefore prioritises maintainability, read performance, graceful degradation, and a low operational footprint, while preserving a clean extraction path to a larger distributed system.

3.1 Architecture Overview and Requirements Alignment

Each major structural choice traces to a requirement it satisfies. The recommendation module realises two-sided discovery (FR2); the users module owns identity and role-based accounts (FR1); the chat module realises match-triggered communication (FR3); the clubs and map modules model supply-side discovery (FR4); and the moderation and subscriptions modules realise trust/safety and monetisation (FR5, FR6). Non-functionally, the schema-first type-safe pipeline serves maintainability (NFR4); swappable fail-open integrations and the never-empty feed serve reliability (NFR2); the modular monolith with configuration-driven city replication serves scalability (NFR3); targeted indexing serves read performance (NFR1); and a first-class telemetry stack serves observability (NFR6).

3.2 System Context

The platform is the central hub through which dancers, parents, trainers, and clubs interact, mediating every external dependency behind its own API so that no client contacts a third party directly.

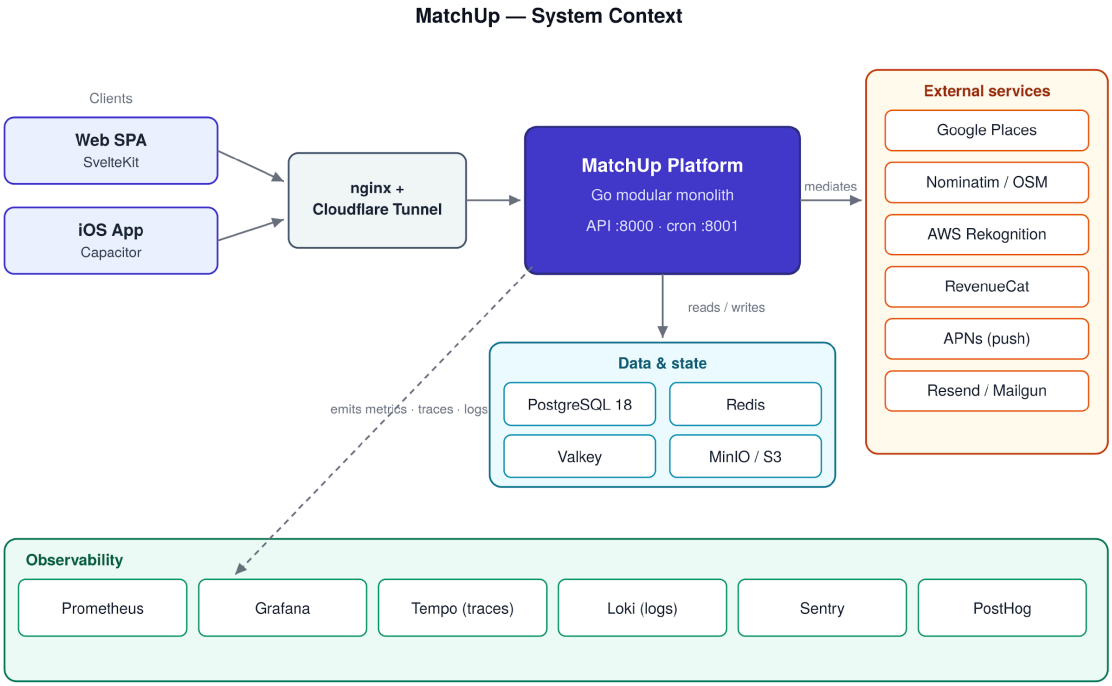

Figure 3.1. C4 system context — MatchUp and its external integrations. Clients reach the API through nginx and a Cloudflare tunnel; the API mediates all external services.

3.3 Major Architecture Decisions (ADRs)

3.3.1 Modular Monolith

Decision: a modular monolith — self-contained domain modules compiling into one deployable binary — over a layered monolith or microservices. **Justification:** each module owns its schema slice and generated data-access code exactly as a service would, giving strict boundaries and a clear extraction path without the operational tax of running many services on day one. For a pre-scale product served by a single vertically scaled instance, the boundaries that make a future split cheap are present from the outset while the cost of the split is deferred. **Trade-offs:** microservices were rejected as premature distributed-systems complexity for a solo developer; a layered monolith was rejected for blurring boundaries and making later extraction expensive.

3.3.2 Cross-Platform Frontend Strategy

Decision: a single SvelteKit codebase wrapped by Capacitor for native iOS. **Alternatives:** native Swift, React Native, Flutter. **Evaluation criteria:** one team's maintenance burden, web-and-mobile reach from one effort, and fit with a swipe-centric UI. **Justification:** native Swift would force a second, web-less codebase; React Native and Flutter would still split web and mobile and add a runtime. A SvelteKit SPA delivered to the browser and, through a WebView, to the App Store yields one UI codebase, one design system, and one set of flows. **Trade-off:** the

WebView constrains the deepest native capabilities, accepted because the product's interactions are well within WebView reach.

3.3.3 Database Selection

Decision: PostgreSQL 18. **Alternatives:** MySQL, MongoDB. **Evaluation criteria:** relational integrity for a graph of users, preferences, and matches; rich indexing for a read-heavy matching query; and operational simplicity. **Justification:** MongoDB's document model fits poorly with inherently relational two-sided matching and risks denormalisation bugs; MySQL lacks PostgreSQL's GIN and partial-index strength on array fields central to the recommender. PostgreSQL self-hosted gives the indexing the workload needs with full operational control. **Trade-off:** self-hosting carries an operational burden, mitigated by the one-shot migration container and containerised deployment.

3.3.4 Analytics Platform Selection

Decision: PostHog. **Alternatives:** Mixpanel, Firebase Analytics, Amplitude. **Evaluation criteria:** product-funnel analytics, self-hostability alongside the existing Docker stack, and cost at low volume. **Justification:** Firebase Analytics would pull the project into a separate cloud ecosystem; Mixpanel and Amplitude are capable but commercial SaaS with volume-based pricing. PostHog covers funnels, retention, and event capture, runs within the same Dockerised infrastructure as the rest of observability, and keeps product data under the project's control. **Trade-off:** self-hosted analytics adds a service to operate, accepted because it consolidates analytics with the monitoring stack.

3.4 Build vs. Buy Decisions

Several capabilities were explicitly evaluated as build-versus-buy. In every case the deciding factor was whether the capability was core differentiation (build) or undifferentiated heavy lifting better bought and kept swappable (buy).

Capability	Build vs. buy decision	Justification
Subscriptions (RevenueCat)	Buy	App Store receipt validation and entitlement logic are error-prone and non-differentiating; a webhook maps products to internal plans.
Email (Resend / Mailgun)	Buy	Deliverability is a solved, undifferentiated problem; a mock provider covers local dev, and the provider is pluggable.
Image moderation (AWS Rekognition)	Buy	Building NSFW detection is infeasible solo; integration is optional and fail-open.
Geocoding / club import (Google Places)	Buy	Map data is undifferentiated; a daily spend cap bounds cost, with Nominatim as the open fallback.

Capability	Build vs. buy decision	Justification
Error/crash reporting (Sentry)	Buy	Mature, cross-platform, and far cheaper than rebuilding crash aggregation.
Edge exposure (Cloudflare Tunnel)	Buy	Secure public exposure without managing inbound networking or certificates at this stage.
Recommendation engine	Build	Core differentiation — the two-sided, never-empty matching logic is the product and cannot be outsourced.
Auth / sessions	Build	Nonce-based session invalidation is simple, keeps the trust boundary in-house, and avoids an identity-provider dependency.
Google sign-in	Buy (verify) / Build (link)	Google issues and signs the ID token; the backend only verifies it server-side and links it via the user_identities table — no credential handling.

Table 3.1. Build-versus-buy decisions across the platform.

3.5 Technology Stack Selection

Each layer was evaluated as Technology → Alternatives → Criteria → Selected → Trade-offs against four cross-cutting criteria: minimise technical debt and maintenance (NFR4), avoid bottlenecks (NFR1), enable vertical scaling (NFR3), and keep the operational footprint low.

Layer	Alternatives	Evaluation criteria	Selected	Trade-offs
Backend	Node.js, Python, Java	Perf, deploy simplicity, concurrency	Go 1.25 + Gin	Smaller library ecosystem
Database	MySQL, MongoDB, Supabase	Relational integrity, indexing	PostgreSQL 18	Self-host operational burden
Data access	ORM (GORM)	Query transparency, no N+1	pgx + sqlc	More upfront SQL
Frontend	React, Vue	Lean client, swipe UI fit	SvelteKit 2 / Svelte 5	Smaller talent pool
Mobile	Native Swift, RN, Flutter	Web+iOS reuse	Capacitor	WebView capability limits
Storage	Local disk, GCS	S3-swap, env-driven	MinIO / S3	Eventual consistency
Analytics	Mixpanel, Firebase, Amplitude	Self-host, funnels, cost	PostHog	Extra service to operate

Layer	Alternatives	Evaluation criteria	Selected	Trade-offs
Monitoring	Datadog, New Relic	Self-host, cost, OTel-native	Prometheus/Grafana/Tempo/Loki	Self-managed stack
Messaging	Managed chat SaaS	Control, extractability	In-house chat module	Built, not bought

Table 3.2. Technology stack: alternatives, criteria, selection, and trade-offs.

The unifying principle is one codebase, one language per tier, and type-safety carried from the database to the API contract. Go compiles to a single static binary that deploys and scales vertically with ease; PostgreSQL through pgx and sqlc generates type-safe Go from hand-written SQL, eliminating ORM opacity and N+1 access; and the SvelteKit-plus-Capacitor client ships once to two platforms.

3.6 Modular Monolith — Domain Ownership

The backend is organised as fourteen self-contained modules — users, profiles, recommendation, feed, matches, chat, clubs, trainers, maps, moderation, subscriptions, notifications, files, and a shared core, supported by otp, email, and ratelimit. Each owns its slice of the schema and its generated queries, wired explicitly in a single composition root. A custom generator stitches the per-module schema files, in dependency order, into one combined PostgreSQL schema, which is what allows a profile to reference a club across a module boundary while each module's schema stays self-contained. The boundaries are therefore real, not nominal — they are the seam along which a module can later be cut into its own service.

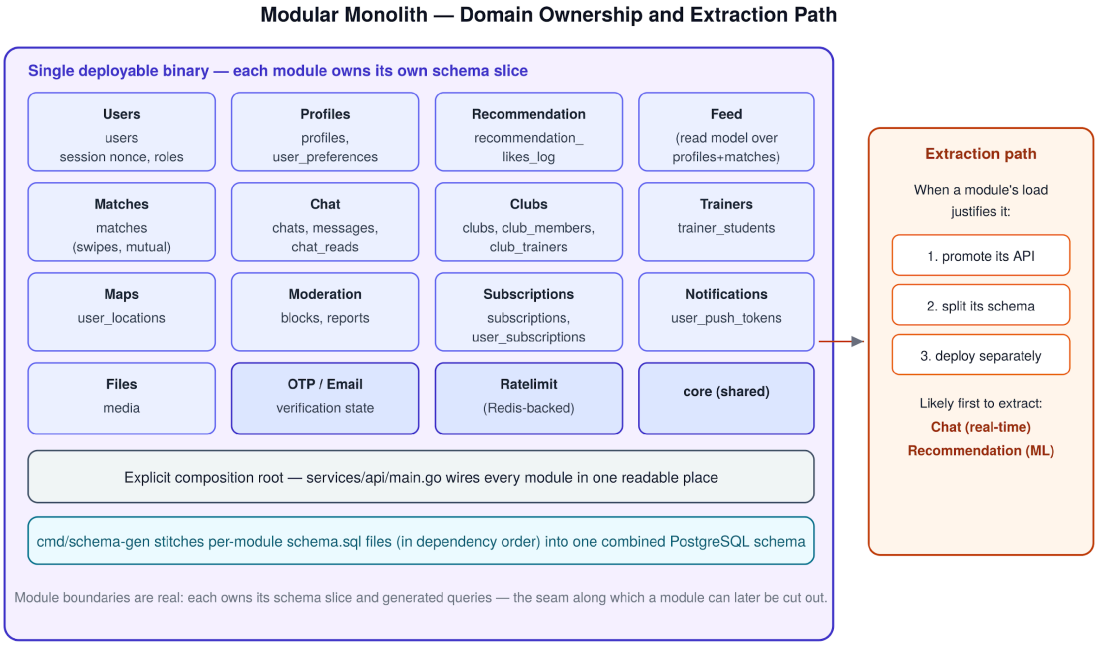

Figure 3.2. Module ownership and extraction path. Each module owns its schema slice and queries; the composition root wires them; the most likely first extractions are the real-time chat and the ML recommendation services.

3.7 API Contract and Data Flow

The API is a REST/JSON interface with a single uniform response envelope: every response takes the shape { data, error*code }. A successful response carries its payload in data with an empty error*code; a failure carries a stable, machine-readable error_code that the frontend maps to a localised message. Standardising on a single, minimal envelope is what lets the client handle and localise every error generically, without special-casing endpoints — serving localisation (NFR7) and maintainability (NFR4).

```
// Uniform response envelope returned by every endpoint.
type Response struct {
    Data      any    `json:"data"`
    ErrorCode string `json:"error_code"`
}
```

The data flow for a typical request is a fixed, observable chain: client → nginx → Gin middleware (panic recovery → Sentry → request ID → OpenTelemetry tracing → Prometheus metrics → structured logging → CORS) → per-route auth and rate-limit guards → controller → service → generated query → PostgreSQL, with third-party calls mediated server-side and telemetry emitted asynchronously.

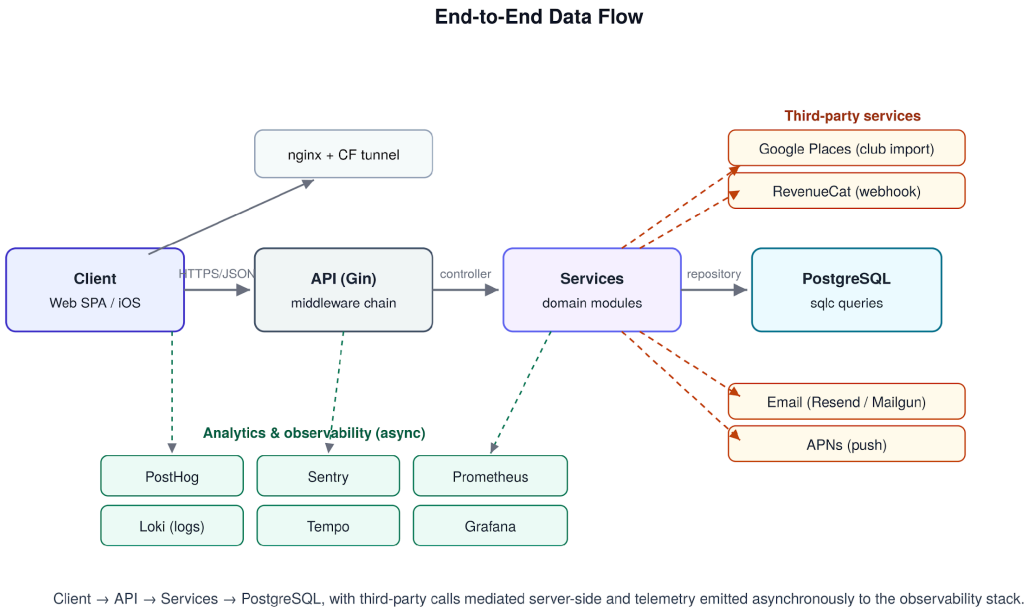

Figure 3.3. End-to-end data flow: Client → API → Services → PostgreSQL, with third-party services mediated server-side and analytics/observability emitted asynchronously.

3.8 Component Deep Dive: The Recommendation Engine

The recommendation engine is the technical heart of the system, and its central property — that the feed is never empty — is a direct response to the low-density market. The engine implements a three-tier candidate pipeline that degrades gracefully so there is always

something to evaluate: Tier 3 returns genuine two-sided matches (the candidate passes the user's filters and the user passes theirs); Tier 2 returns everyone satisfying the user's preferences; and Tier 1 is a proximity safety net of gender and distance only. The recommender walks the tiers in quality order, deduplicating by user until a page is full. Distance is a haversine expression in SQL anchored on the primary club, falling back to a city centroid; already-swiped and blocked users are excluded in the query.

Tier	Candidate definition	Ordering
3 — Mutual	Passes the user's filters and the user passes theirs (two-sided)	Distance
2 — Filter	Satisfies all of the user's preferences	Distance
1 — Proximity	Gender + distance only; fills the deck	Distance

Table 3.3. The three-tier recommendation pipeline.

A design economy underpins the pipeline: the filters are nullable query parameters, so a single parameterised query powers all three tiers — they differ only in which arguments they bind. The engine is also instrumented for its future: every like writes a feature snapshot to a like-log, so the training data for a learned ranking model is already accumulating and the system can move from rule-based SQL to a learned model without re-instrumentation.

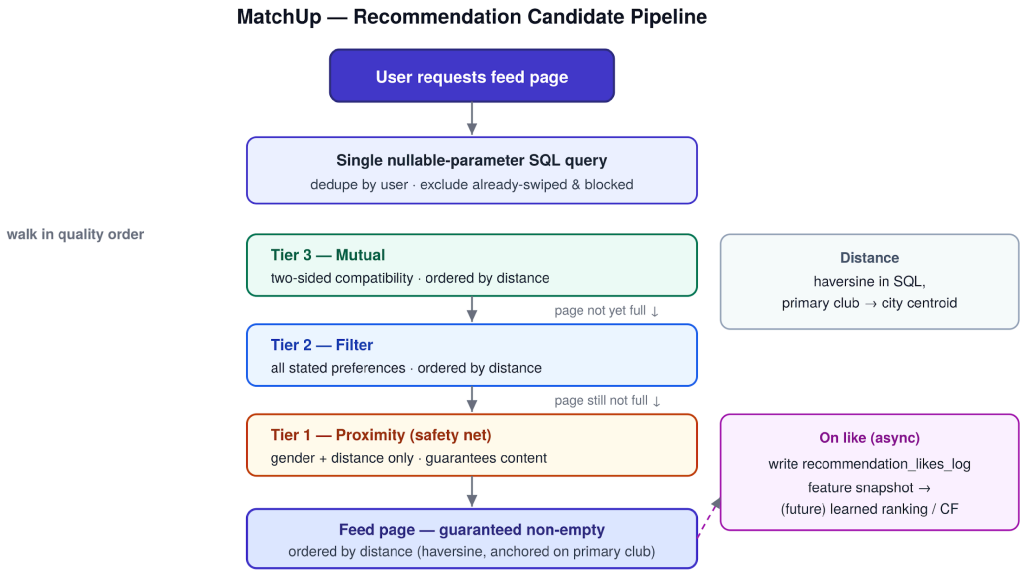

Figure 3.4. Recommendation candidate pipeline: one nullable-parameter query invoked in quality order until a full page is assembled; likes logged asynchronously for future learned ranking.

3.9 Data Layer and Schema Design

The data model is schema-first: each module declares its own schema, stitched in dependency order into one combined schema that sqlc reads. The core entities are users (identity, role, and

the session nonce) with user*identities decoupling external OAuth providers from the users table; profiles and user*preferences (the two-sided matching data) plus the recommendation*likes*log; matches; a polymorphic chats table with messages, chat*reads, and message*reports; the clubs cluster (clubs, club*members, club*trainers, trainer*students); user*locations; moderation tables (blocks, reports); media; subscriptions; and push tokens. Because a swipe app is read-dominated, the profiles table is deliberately over-indexed for matching: GIN indexes on the dance-style and category arrays, btree indexes on every filterable scalar, and partial indexes for hot predicates such as visible profiles only and the trainer/dancer split (NFR1). Migrations follow a two-path strategy — a fresh database receives the full combined schema, an existing one only the pending incremental migrations — applied by a one-shot container the API waits on before booting.

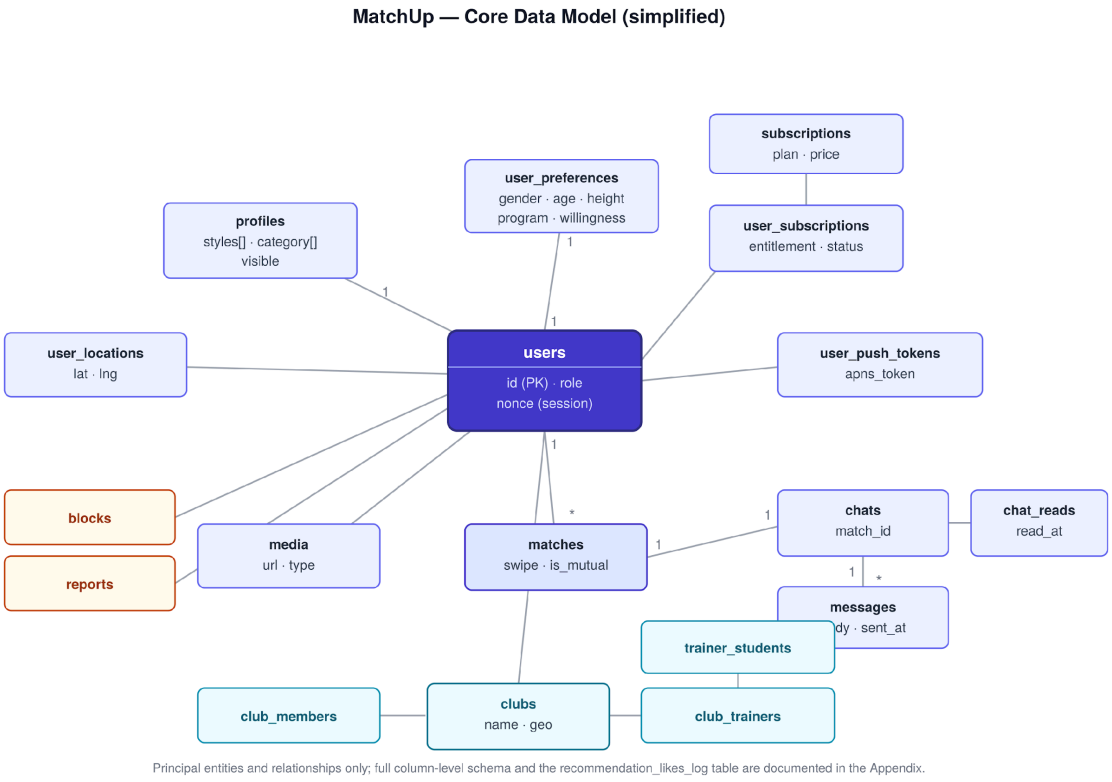

Figure 3.5. Core data model. The users, profiles, and preferences entities form the matching core; matches, chat, the clubs cluster, locations, moderation, media, subscriptions, and push tokens extend it.

3.10 Integration Architecture

Every external dependency follows one pattern: integrate the real provider, keep it swappable, and make it degrade safely, so no third party can take the whole product down (NFR2).

Concern	Provider	Degradation / swappability
Sign-in	Google Identity (OAuth)	Optional alongside email/password; ID-token verified server-side
Club import	Google Places	Daily spend cap; import optional

Concern	Provider	Degradation / swappability
Geocoding	Nominatim / OSM	Open provider; replaceable
Avatar moderation	AWS Rekognition	Fail-open; toggled by env var
Subscriptions	RevenueCat	Webhook-driven entitlements
Push	APNs (apns2)	Per-device token; non-blocking
Email	Resend / Mailgun	Pluggable; mock in dev
Object storage	MinIO / S3	Identical client, env endpoint

Table 3.4. Integration fallback and swappability matrix.

3.11 Deployment and Infrastructure

Current state. The application is live and internet-accessible. The entire stack runs through Docker Compose — PostgreSQL, Valkey, MinIO, the migration job, the Go API, and an nginx container serving the static frontend — on a private bridge network, exposed via a Cloudflare tunnel. The observability components run within the same Dockerised infrastructure. **Planned state.** The next milestone is a clean separation into dev, staging, and production environments and a continuous-deployment pipeline that builds and rolls out the Docker images automatically; the prerequisites — multi-stage Dockerfiles producing tiny static binaries, a migration container the API waits on, externalised configuration, and S3-swappable storage — already exist, so promotion touches infrastructure, not application code.

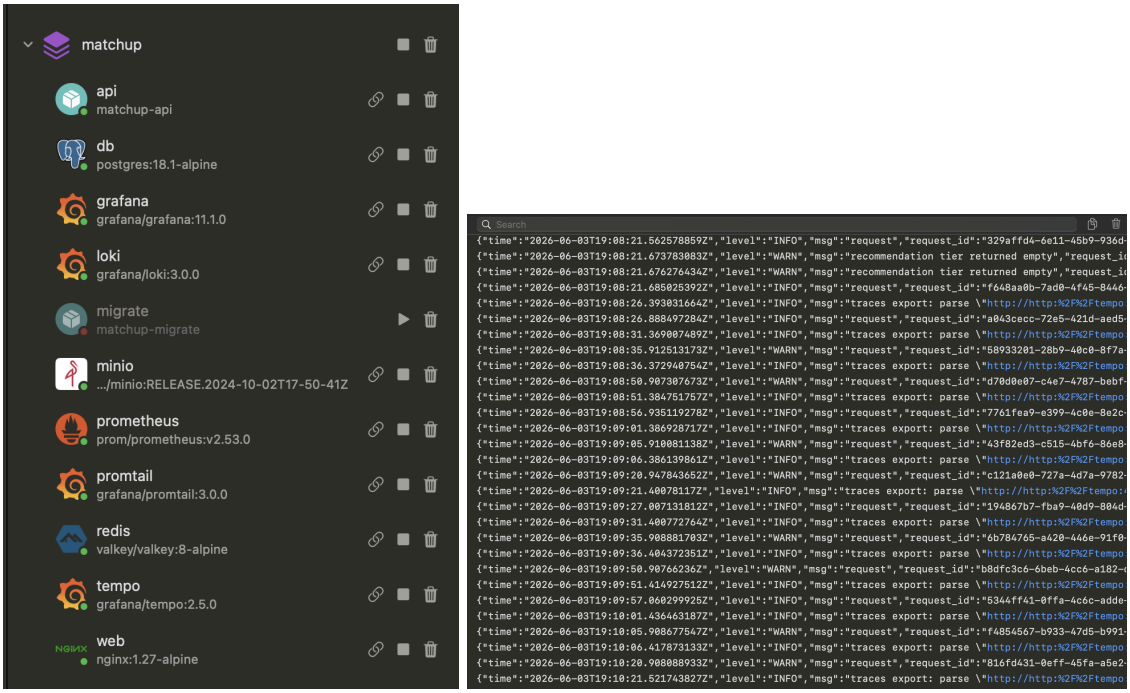

Figure 3.6. Docker Compose architecture and production deployment.

3.12 Security

3.12.1 Authentication and Sessions

Authentication supports email/password and Google sign-in. Email/password sessions are JWTs (HS256) whose validity is bound to a per-user nonce on the users record; rotating the nonce invalidates every outstanding token, implementing log-out-everywhere and credential-compromise response without server-side session storage. Tokens travel either as a Bearer header or an auth_token cookie, the latter used by the web app with credentialed requests. OTP email verification (five-digit codes held in Valkey with a fifteen-minute TTL and an attempt cap) and password reset complete the identity flows, with attempt limits enforced by the rate-limit module.

Google sign-in uses an ID-token verification flow on both web (Google Identity Services) and Capacitor native: the client obtains an ID token, the backend verifies it server-side with audience pinning, and a user*identities row links the external subject to an internal account — creating a passwordless, pre-verified user on first sign-in. The user*identities table decouples providers from the users table, so one user can hold several linked identities and a new provider needs only a new row and handler, which is also the seam reserved for the WebAuthn and Okta work on the roadmap.

Banned accounts are enforced at the protocol level: immediately after token validation the JWT middleware rejects any user whose role is banned with a 403 response, before any handler runs, so a ban takes effect across every authenticated route at once. A Redis-backed rate limiter guards every abuse-prone endpoint, CORS is enforced in the middleware chain, sqlc's parameterised queries close off SQL injection by construction, and the moderation module provides block, report, and ban primitives; avatar uploads are screened by the optional Rekognition integration (NFR5).

Endpoint	Window	Limit	Key
Login	1 min	3	email
Register	1 hour	5	client IP
Swipe	1 min	200	user ID
Message	1 min	60	user ID
Upload	24 hours	20	user ID

Table 3.5. Per-endpoint rate limits on abuse-prone routes.

3.12.2 Passwordless Future Strategy

Google sign-in is already in place, layered over the email-first, OTP-verified flow through the provider-agnostic user_identities table. The roadmap extends this seam toward fully passwordless onboarding to further reduce password-related risk and improve conversion: WebAuthn / passkeys and device biometrics (Sign in with Apple, Face ID and Touch ID, and Android biometrics), with Okta as an OIDC option for B2B and club-operator single sign-on. Because sessions are nonce-bound rather than password-bound and identities are already decoupled into their own table, adding each provider requires only a new identity row and

handler rather than changes to session management; the principal external constraint is the maturity of the Capacitor WebAuthn plugin on the iOS WebView.

3.12.3 OWASP Top 10 Security Assessment

OWASP Top 10 risk	Current status	Mitigation
A01 Broken Access Control	PASS	/metrics protected by METRICS_TOKEN bearer middleware; role-based auth enforced on all admin routes; JWT validated in AuthMiddleware; banned-account enforcement added; media paths non-enumerable (UUID); X-Robots-Tag: noindex on media assets
A02 Cryptographic Failures	PASS	DB_SSL_MODE defaults to 'require'; production startup guard rejects disable; Redis TLS optional via REDIS_TLS_ENABLED; bcrypt for passwords (cost 12); JWT HS256 with secret guard; HTTPS enforced via nginx HSTS header
A03 Injection	PASS	All DB queries use sqlc parameterised statements; UUID path params validated with StringToUUID; HTML emails rendered with html/template (auto-escape); no raw string concatenation in SQL paths
A04 Insecure Design	PASS	forgot_password_token_expires_at column added (1-hour expiry); token cleared on use; password-reset token no longer valid indefinitely; admin_audit_log records all moderation actions
A05 Security Misconfiguration	PASS	nginx security headers: X-Frame-Options DENY, X-Content-Type-Options nosniff, Referrer-Policy, Permissions-Policy, HSTS, CSP; gin.SetMode(release) in production; placeholder guard rejects known weak secrets at boot; Docker base image pinned to golang:1.25.8-alpine
A06 Vulnerable & Outdated Components	PASS	govulncheck added to CI Go job; pnpm audit --audit-level=high added to CI frontend job; deprecated @codetrix-studio/capacitor-google-auth (Capacitor 6 only, archived) replaced with @capgo/capacitor-social-login v8.x (Capacitor 8 compatible)
A07 Identification & Authentication Failures	PASS	Redis-backed login lockout after 10 failed attempts (15-minute TTL); lockout cleared on successful login; per-minute sliding-window rate limiter remains on /auth/login; password-reset token expiry enforced (A04); SameSite=Lax on auth cookie

OWASP Top 10 risk	Current status	Mitigation
A08 Software & Data Integrity Failures	PASS	SameSite=Lax set on auth_token cookie in all three auth issuance paths (password, OTP, Google); RevenueCat webhook verified with shared secret; no serialized untrusted objects; HSTS and CSP reduce XSS-driven CSRF surface
A09 Security Logging & Monitoring Failures	PASS	Structured slog WARN logs on every auth_failure (unknown email, wrong password); admin_audit_log DB table records all ban/hide-message actions; admin_audit structured log emitted for hide_message; Grafana alert rules for auth-failure spike and 4xx spike added to config/observability/
A10 Server-Side Request Forgery	PASS	resolveURL/LookupURL in gmaps package now validates every URL (initial + each redirect hop) against an explicit allowlist: google.com, maps.app.goo.gl, goo.gl, maps.googleapis.com, places.googleapis.com; any other host — including internal addresses — returns an error

Table 3.6. OWASP Top 10 assessment structure (to be completed).

3.13 Observability, SLIs and SLOs

Observability is a first-class concern (NFR6). Prometheus scrapes the API and cron processes with custom counters for swipes, matches, and per-tier recommendation hits alongside connection-pool gauges; Grafana provides a provisioned API-overview dashboard; OpenTelemetry exports traces to Tempo; structured slog logs ship to Loki via Promtail; Sentry captures errors and crashes on backend and frontend including native iOS; and PostHog tracks the product funnel. The three pillars are cross-linked, so a single request can be followed across logs, metrics, and traces.

Beyond tooling, the system defines explicit service-level indicators (SLIs) and objectives (SLOs) as engineering targets. Where live measurements are not yet captured, the objective is marked as a projected target for early production operation; where dashboards exist, the SLI references the corresponding Grafana panel.

SLI	SLO (target)	Status
API availability	≥ 99.5%	Projected target for early production operation
Feed generation latency (P95)	< 500 ms	Projected target; Grafana latency panel provisioned
Match creation success rate	> 99%	Projected target for early production operation
Crash-free sessions	> 99%	Projected target; tracked via Sentry

SLI	SLO (target)	Status
Error rate	< 1%	Projected target; Grafana 5xx panel provisioned
Chat delivery success rate	> 99%	Projected target for early production operation

Table 3.7. Service-level indicators and objectives. Each SLI maps to a monitoring dashboard.

SCREENSHOT FROM SERVER-SIDE METRICS WILL BE AVAILABLE DURING FINAL SUBMISSION

Figure 3.7. Grafana API-overview dashboard (request rate, 5xx, P50/P95/P99 latency, pool health)

3.14 Chapter Summary

This chapter set out to translate the requirements of Chapter 2 into a justified design. It recorded the major architecture decisions — a modular monolith, a shared SvelteKit/Capacitor client, PostgreSQL, and PostHog — each as an explicit decision-alternatives-criteria-trade-off record, and added a build-versus-buy analysis that kept the recommendation engine and authentication in-house while buying undifferentiated capabilities behind swappable interfaces. It detailed the two-field API envelope and the end-to-end data flow, the three-tier recommendation engine and its read-optimised schema, the security model and its passwordless roadmap, and a first-class observability stack now anchored by explicit SLIs and SLOs. The engineering implication is that the system is not only built but measurable and operable: the next chapter shows how it was implemented, sprint by sprint, and how the user journey maps onto these internals.

Chapter 4. User Journey, Implementation, and Engineering

This chapter connects the product experience to the system internals and documents how the architecture of Chapter 3 was realised in working software. It opens with the end-to-end user journey and data flow, then covers the development methodology, the three sprints with their retrospectives, the critical code implementations, and the continuous-integration pipeline.

4.1 End-to-End User Journey

The complete experience runs from first launch to subscription. A user opens the application and registers or logs in, then verifies their email through a one-time-password sent by the email provider. Onboarding follows in four steps: personal profile information; partner-search preferences; current dance-club selection (choosing an existing club or creating a new one, the latter optionally imported from a Google Maps link); and profile-photo upload — after which the profile is complete. The user then enters feed discovery, where they browse recommendations, swipe profiles, receive matches, and open chats. In parallel, the map offers the same filtering logic to discover dancers and clubs geographically and to search for or create new clubs. Settings provide profile management throughout, and the subscription flow offers the premium upgrade at the point where advanced filters and messaging add value.

Each step pairs a user action with backend, database, and analytics activity. The table below traces the journey through the stack, making explicit the relationship between what the user does and what the system does in response.

Step	Backend action	Database interaction	Analytics event
Register / login	users module validates credentials or verifies a Google ID token; issues a nonce-bound JWT	insert/select on users, user_identities	auth_started, auth_succeeded
Email verification	otp module issues and checks a one-time code	OTP state in Valkey	verification_completed
Onboarding — profile	profiles module persists profile fields	insert on profiles	profile_step_completed
Onboarding — preferences	profiles module persists two-sided preferences	insert on user_preferences	preferences_set
Onboarding — club	clubs module links or creates a club (Google Places optional)	insert/select on clubs, club_members	club_selected / club_created
Photo upload	files module stores media; optional Rekognition scan	insert on media; object to MinIO/S3	photo_uploaded
Feed	recommendation module runs the tiered query	indexed read on profiles, matches	feed_viewed

Step	Backend action	Database interaction	Analytics event
Swipe	matches module records the swipe; detects mutual	insert on matches	swipe, match_created
Chat	chat module auto-creates a conversation on match	insert on chats, messages, chat_reads	chat_opened, message_sent
Subscription	subscriptions module updates entitlement via RevenueCat webhook	update on user_subscriptions	subscription_started

Table 4.1. User journey traced through user action, backend, database, and analytics.

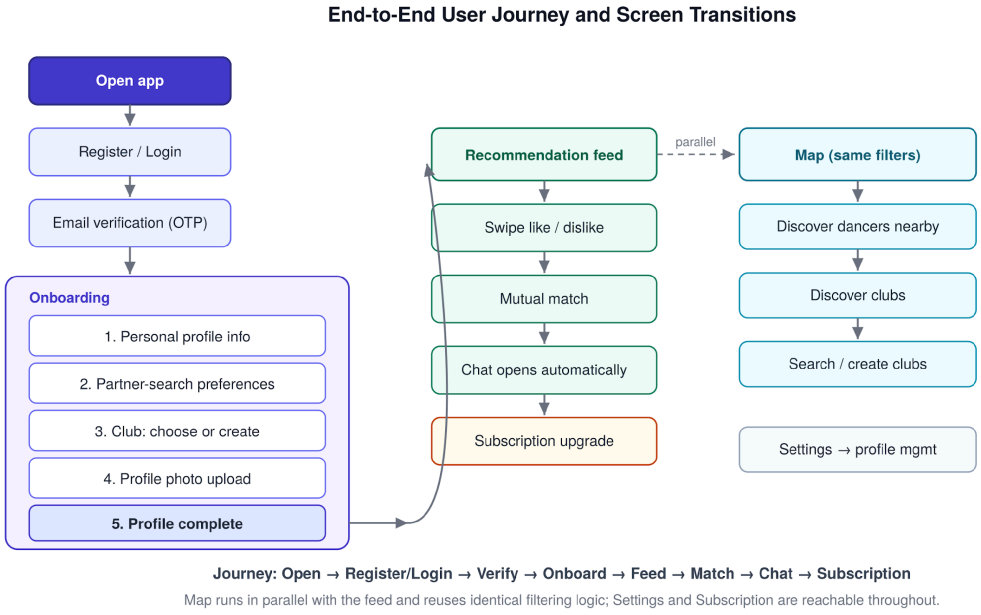

Figure 4.1. User journey and screen transitions: Open → Register/Login → Verify → Onboard → Feed → Match → Chat → Subscription, with the map running in parallel on the same filtering logic.

4.2 Key UI Screens

The interface was designed in Figma and validated through usability testing before and during implementation. The principal screens are reserved below for the corresponding UI artefacts.

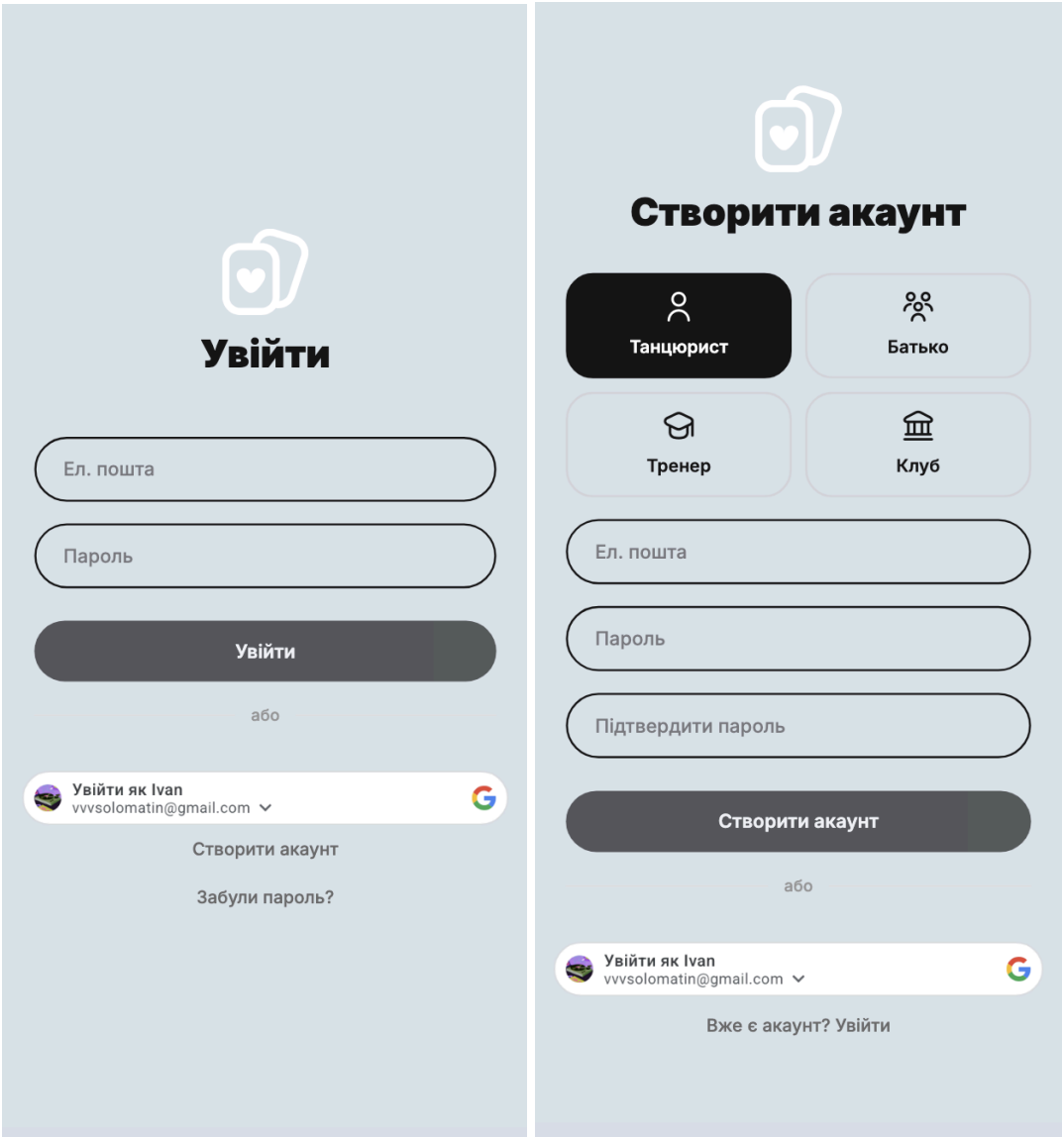

Figure 4.2. Login and registration — email/password and Google sign-in.

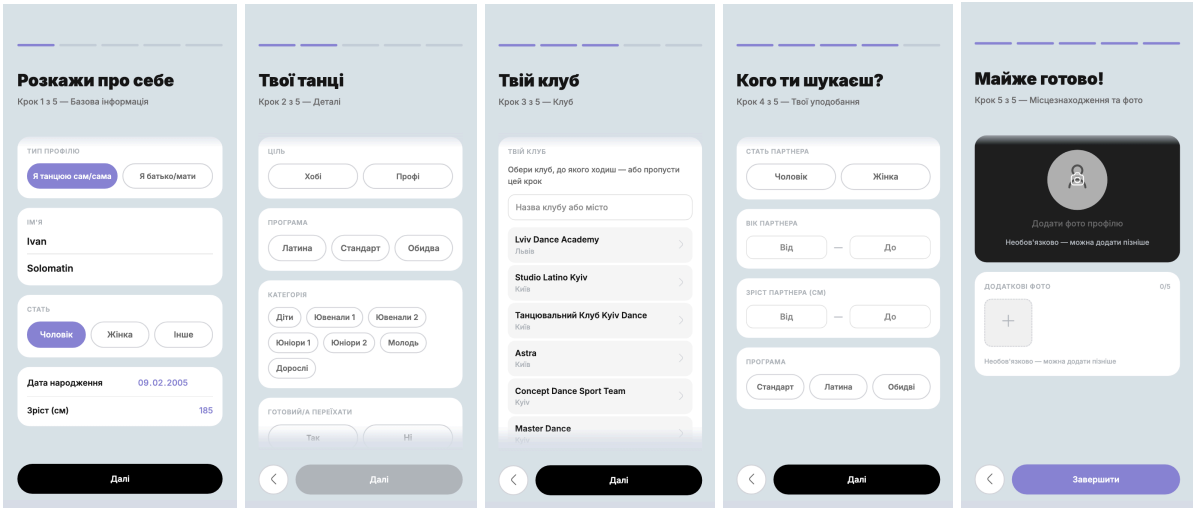

Figure 4.3. Onboarding flow — profile, preferences, club selection.

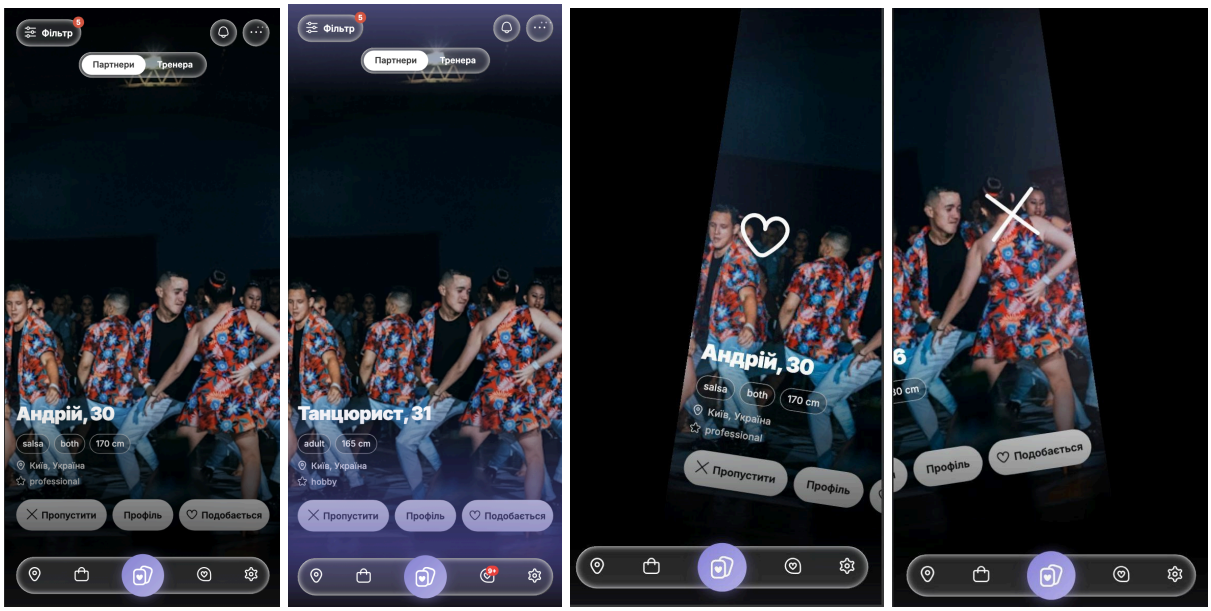

Figure 4.4. Recommendation feed and swipe interaction.

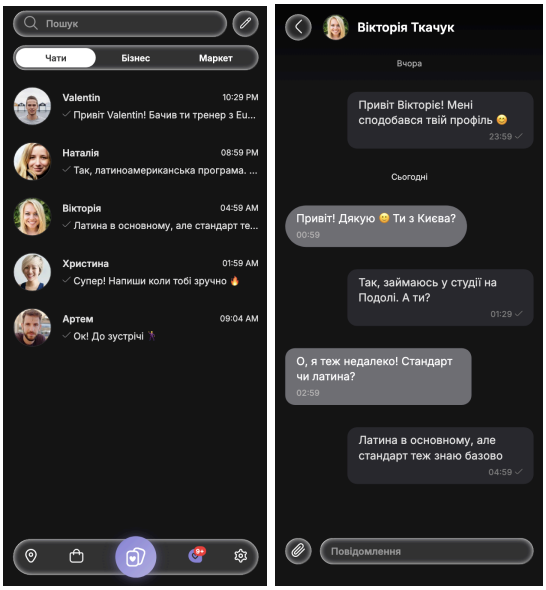

Figure 4.5. Match and in-app chat / messaging flow.

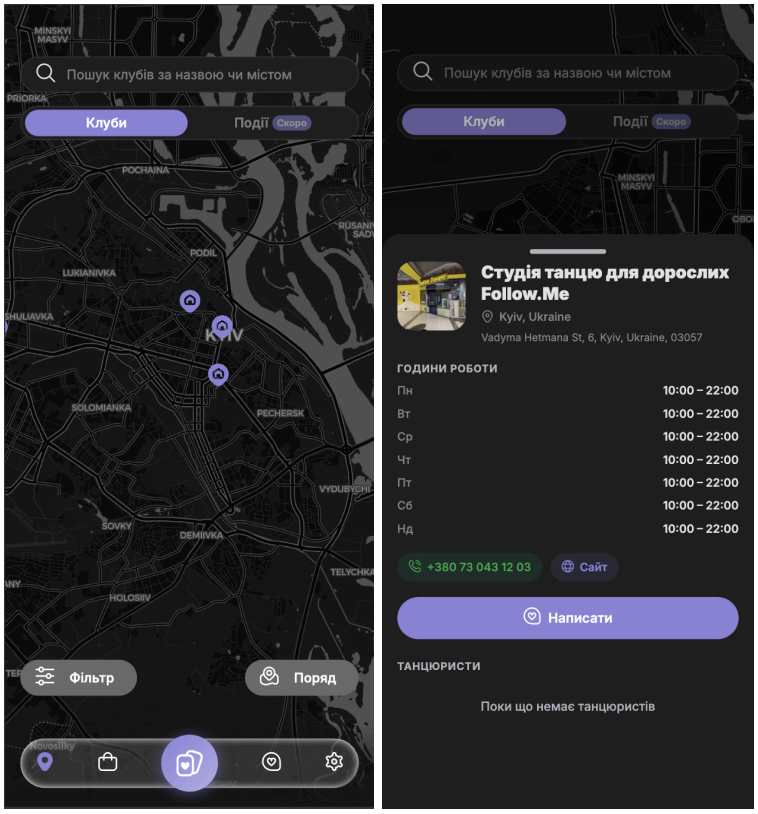

Figure 4.6. Map view — dancers and clubs with feed-equivalent filters.

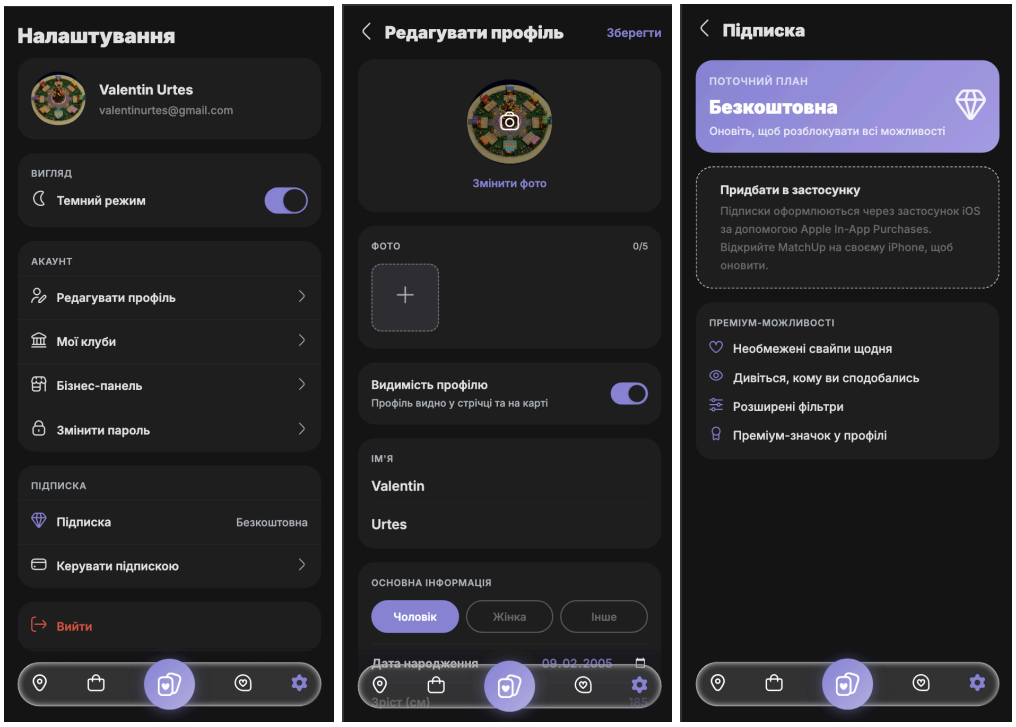

Figure 4.7. Profile management and subscription upgrade.

4.3 Development Methodology

The project was delivered by a single developer following an agile methodology structured around three month-long sprints, each opening with goal-setting and closing with a review and retrospective. Because the work was solo, the discipline a team gets from division of labour came instead from the architecture: the modular-monolith boundaries acted as the organising principle, letting whole domains be added in sequence without destabilising what already worked.

4.4 Sprint 1 — Foundations and Identity

Results. The modular-monolith skeleton, the schema-first sqlc pipeline, and the full authentication stack — registration, JWT sessions with nonce-based log-out-everywhere, OTP email verification, and password reset.

What went right. Committing to schema-first development with sqlc this early paid off across the project — type-safety from database to API and no accumulated ORM debt.

What went wrong. The authentication edge cases — session invalidation and OTP attempt limits — were underestimated and took longer than planned.

Improvement carried forward. Lock down the core data contracts before building features on top of them.

4.5 Sprint 2 — The Core Loop: Profiles, Feed, and Matching

Results. Profiles and two-sided preferences, the three-tier recommendation engine, the swipe feed, mutual-match detection, automatic chat creation on a match, and the like-log instrumentation for future learned ranking.

What went right. The three-tier never-empty-feed design solved the cold-start and low-density problem cleanly, and the nullable-filter SQL technique kept all three tiers behind a single maintainable query.

What went wrong. The first matching queries were slow until the right indexes were added — performance was treated as an afterthought rather than designed in.

Improvement carried forward. Design indexes alongside the query, and instrument from day one — which is why the like-log exists now.

4.6 Sprint 3 — Ecosystem, Safety, Monetisation, and Mobile

Results. Clubs with Google Maps import, trainers, the map, the polymorphic chat (direct and club threads) with keyword filtering and per-message moderation, the full block/report/ban suite with protocol-level banned-user enforcement, Google sign-in via server-side ID-token verification, file uploads with optional NSFW scanning, RevenueCat subscriptions, the complete observability stack, and the Capacitor iOS application with native push.

What went right. Because the module boundaries were clean, adding whole new domains — clubs, subscriptions, push — was fast and low-risk; the monolith structure proved itself under breadth.

What went wrong. Breadth outran depth: continuous deployment and a formal automated-test suite were deferred to ship surface area, and a couple of pieces — the cron container and the marketplace screen — are coded but not yet fully wired.

Improvement carried forward. The next sprint is explicitly a hardening sprint — a formal test suite, a CI test step, proper environments and continuous deployment, and finishing the half-wired pieces — before adding the events module that opens retention and promotion revenue.

4.7 Critical Code Implementations

4.7.1 The Tiered Recommendation Query

All three tiers are served by one parameterised query whose filters are nullable: a NULL argument disables filtering on that dimension. The service binds the full preference set for Tier 3, relaxes the mutual condition for Tier 2, and reduces to gender and distance for Tier 1, walking the tiers until a page is filled and excluding already-swiped and blocked candidates in the query itself.

```
-- One nullable-filter query powers all three tiers.
-- A NULL parameter means "do not filter on this dimension".
SELECT p.user_id, haversine(p.lat, p.lng, $anchor_lat, $anchor_lng) AS dist
FROM profiles p
WHERE p.visible
```

```
  AND ($gender  IS NULL OR p.gender  = $gender)
  AND ($age_min IS NULL OR p.age     >= $age_min)
  AND ($age_max IS NULL OR p.age     <= $age_max)
  AND ($program IS NULL OR p.program = $program)
  AND p.user_id NOT IN (SELECT swiped_id  FROM matches WHERE actor_id   = $me)
  AND p.user_id NOT IN (SELECT blocked_id FROM blocks  WHERE blocker_id = $me)
ORDER BY dist ASC LIMIT $page;
```

4.7.2 Messenger Architecture

The in-app messenger required the most architectural deliberation of any component; several approaches were evaluated before the current design was chosen for scalability, reliability, and future expansion. A single polymorphic chats table serves two thread kinds — direct dancer-to-dancer conversations and user-to-club threads — distinguished by whether a club_id is set, which avoids a second schema for what is structurally the same one-to-one messaging primitive. A direct conversation is created automatically the moment a match becomes mutual; messages, read state, and per-message reports are modelled alongside; and outgoing messages pass through a keyword safety filter that blocks phone numbers, social handles, and slurs before send, which matters in a product used by minors and their parents. Message-level moderation soft-deletes (hides) a reported message while preserving a content snapshot for the audit trail, and hidden messages are filtered from every client-facing query. Chat delivery is currently HTTP polling on a three-second interval; migrating to WebSockets or server-sent events for lower latency is on the roadmap. Because chat lives in its own module with its own schema slice, it can later be extracted to a dedicated real-time service without disturbing the rest of the system.

4.7.3 Identity and Session Management

Sessions are JWTs whose validity is tied to a per-user nonce. Ordinary verification checks signature and expiry; binding additionally to the nonce means rotating it invalidates every prior token, implementing log-out-everywhere without server-side session storage. Google sign-in is layered on through an ID-token verification flow — the client obtains a token from Google, the backend verifies it server-side and links it via the user_identities table — so a Google user is created passwordless and pre-verified on first sign-in and receives the same session cookie as an email/password user. OTP verification and password reset round out the flows, with attempt limits enforced through the rate-limit module, and a banned role is rejected at the middleware before any handler runs.

4.7.4 Resilient Swipe and Match-to-Chat Flow

The swipe path is built to fail gracefully. On the client, a card is removed optimistically the instant the user swipes, but a network error or a 429 rate-limit response re-inserts the card and surfaces a distinct toast rather than silently dropping the action. On the server, a swipe records the like or pass in a transaction and, on a mutual like, creates the direct chat as a best-effort step: if chat creation fails the match rows are still committed and the mutual-match result is still returned, with the conversation created lazily on the next inbox open. A no-profile guard returns an explicit signal when the caller has not completed a recommendation profile, which the client

renders as a “complete your profile” call to action instead of a blank deck — a small but important difference between an empty feed that looks broken and one that guides the user forward.

4.7.5 Safe-by-Default Integrations

The integration code embodies the swappability and fail-open principle: Rekognition moderation is optional and fail-open and can be disabled by environment variable; Google Places import is bounded by a daily spend cap; RevenueCat entitlements update through a secured webhook; and every provider is wrapped behind an interface so development and production differ only by configuration, with a mock email provider preventing accidental sends from dev.

4.8 Continuous Integration

A GitHub Actions pipeline runs on every push and pull request to the master branch, in two parallel jobs. The backend job runs go vet and go build across all packages, so nothing that fails to compile or fails vet can merge. The frontend job installs with a frozen lockfile, runs pnpm check (Svelte and TypeScript type-checking), and performs a full production build. The pipeline therefore guarantees that the entire monorepo compiles, type-checks, and builds on every change; building and pushing Docker images and deploying are the manual steps that the planned hardening sprint will automate.

Figure 4.8. GitHub Actions pipeline — backend and frontend jobs on every push/PR.

4.9 Key Technical Challenges

Technology-stack selection was resolved by committing to end-to-end type-safety so whole categories of error are caught at compile time. **Cost optimisation** drove the choice of a single vertically scaling binary, an open geocoding provider with a budget-capped paid fallback, an open Redis fork, and storage identical in code between local and cloud backends. **Messenger architecture** required the most up-front deliberation, resolved by the module-isolated, match-triggered design described above. A cross-cutting challenge was the breadth-versus-depth tension of the final sprint, which traded deployment and test automation for feature surface — a deliberate, documented decision now scheduled for the hardening sprint.

4.10 Chapter Summary

This chapter set out to connect the user experience to the engineering and to document how the system was built. It traced the full journey from onboarding to subscription, pairing each user action with its backend, database, and analytics activity, and visualised both the journey and the

end-to-end data flow. It then recorded the three sprints with candid retrospectives — a disciplined foundation, a core loop whose design solved the cold-start problem, and a breadth-first third sprint that proved the monolith's modularity while deferring test and deployment automation. The critical implementations — the single nullable-filter query, the match-triggered messenger, nonce-bound sessions, and safe-by-default integrations — each make a Chapter 3 principle concrete, and the CI pipeline guarantees a building, type-checked monorepo. The engineering implication is a system that is feature-complete on its core loop and honestly bounded on its automation, which the next chapter verifies and validates.

Chapter 5. Verification and Validation

This chapter demonstrates two distinct things: that the system is correct (verification) and that it is useful to its market (validation). Verification draws on executed tests, the CI pipeline, and deployment checks; validation introduces a measurable product-metrics framework and a forward experimentation plan, shifting the emphasis from “the software works” to “users want the product.”

5.1 Verification

Verification establishes correctness against the requirements of Chapter 2 through three complementary mechanisms: an implemented smoke-testing strategy, the continuous-integration pipeline, and deployment verification on the live system.

5.1.1 Implemented Smoke-Testing Strategy

A suite of end-to-end smoke tests is implemented and executed against the live API and a real database, exercising the full set of core flows successfully. The seeded environment — an idempotent seeder generating sixty to eighty realistic dancer and trainer profiles with photos — provides lifelike density so that matching and feed behaviour are exercised under realistic conditions rather than against empty data. The smoke tests cover registration, login, onboarding, profile creation, the recommendation feed, matching, messaging, and subscription activation, confirming that each core flow operates end to end.

Flow	Verification outcome
Registration	Exercised end to end via smoke test — succeeds
Login	Exercised end to end — nonce-bound session issued
Onboarding	Profile, preferences, and club steps complete
Profile creation	Persisted with media upload
Recommendation feed	Returns a populated, deduplicated feed under seeded density
Matching	Swipe recorded; mutual match detected
Messaging	Conversation auto-created on match; messages delivered
Subscription activation	Entitlement updated via RevenueCat webhook

Table 5.1. Smoke-test coverage of core flows, executed against the live API and seeded database.

5.1.2 CI Pipeline and Deployment Verification

The CI pipeline (Section 4.8) guarantees on every change that the entire monorepo compiles, passes go vet, type-checks, and builds — a continuous correctness gate at the code level. Deployment verification confirms that the containerised stack starts in the correct order: the one-shot migration container brings the database to the current schema version before the API boots, and the API, frontend, and observability services come up healthy behind the Cloudflare tunnel on the live endpoints.

SCREENSHOT FROM SERVER-SIDE METRICS
WILL BE AVAILABLE DURING FINAL SUBMISSION

Figure 5.2. Deployment verification — running containers and live endpoints.

5.2 Validation

Validation asks a different question: not whether the software runs, but whether the product is useful to dancers. Because the platform is at the start of its production life, validation is framed as a measurable metrics framework with defined targets, instrumented through the analytics stack and ready to be read against real usage as the user base grows.

Validation metric	What it demonstrates
Retention rate	Users return across partner-search cycles — durable usefulness
Churn rate	Whether value sustains beyond a single search
Average session duration	Depth of engagement per visit
Average screen time	Sustained attention across the experience
Profile completion rate	Onboarding effectiveness and intent
Feed engagement rate	Whether recommendations prompt swiping
Match conversion rate	Whether the engine produces real matches
Subscription conversion rate	Willingness to pay for the value gate

Table 5.2. Product-validation metrics, instrumented via PostHog.

SCREENSHOT FROM SERVER-SIDE METRICS
WILL BE AVAILABLE DURING FINAL SUBMISSION

Figure 5.3. PostHog product-funnel analytics — swipes, matches, auth events, page views.

5.2.1 Future Experimentation Framework

Once usage volume permits, validation moves from observation to experiment. A structured A/B-testing framework will support onboarding experiments (reducing drop-off across the four onboarding steps), pricing experiments (testing the $5 anchor and premium tiers), and recommendation-ranking experiments (comparing rule-based ordering against the learned model that the like-log is already preparing). Each experiment ties directly to a metric in Table 5.2, closing the loop from instrumentation to decision.

5.3 Identified Limitations

The current system has clear, acknowledged limitations. A formal automated unit- and integration-test suite is not yet in place — correctness is verified through executed smoke tests rather than a test pyramid — and no load testing under high concurrency has been performed. Continuous deployment is not yet wired; deployment is manual from Docker Compose behind a Cloudflare tunnel, and the cron container (subscription-expiry jobs) is implemented but currently commented out of the default stack. Chat delivery is HTTP polling on a three-second interval rather than a push transport, with a WebSocket/SSE migration on the roadmap. The mobile build is iOS-only; no Android project exists yet, so the native Android Google-sign-in path is untested. A small number of pieces, notably the marketplace screen, are coded but not fully wired. Each is a deliberate consequence of prioritising a working, validated surface within the timeline, and each is scheduled in the roadmap of Chapter 6.

5.4 Chapter Summary

This chapter set out to show both correctness and usefulness. Verification demonstrated correctness through an implemented smoke-testing strategy that exercises every core flow against a seeded database, backed by the CI compile/type-check/build gate and deployment verification of the ordered container start-up. Validation reframed success around the user, defining a metrics framework — retention, conversion, completion, engagement — instrumented through PostHog, and a forward A/B-testing plan covering onboarding, pricing, and ranking. The limitations were named plainly: automated testing and continuous deployment remain the principal gaps. The engineering implication is a system proven to work on its core flows and instrumented to prove its value as usage grows — which sets up the concluding assessment of what the project contributed and where it goes next.

Chapter 6. Conclusion

This capstone began from a single, specific problem — DanceSport dancers struggle to find compatible partners, and no adequate tool exists to help them — and grew, through research, into a working product engineered for the particular economics of that problem. This concluding chapter assesses the contribution along three axes: academic, technical, and market.

6.1 Academic Contribution

The thesis produced documented, transferable knowledge. It characterised the partner-search process of an under-studied domain through primary and secondary research, and it articulated a reusable approach to **matching in low-density markets** — the three-tier, gracefully degrading candidate pipeline that guarantees a non-empty feed where naive matching fails. It also demonstrated, as an engineering method, how schema-first development with end-to-end type-safety and a modular-monolith structure can give a solo developer microservice-grade domain separation at a fraction of the operational cost, with a defined extraction path. These findings generalise beyond dance to any geographically clustered, two-sided matching problem.

6.2 Technical Contribution

The project delivered a live, internet-accessible system spanning fourteen domain modules: a two-sided recommendation engine built on a single nullable-parameter query and instrumented for a future learned model; a match-triggered in-app messenger; verified, role-based identity with nonce-bound sessions; club, trainer, and map discovery; a full trust-and-safety suite; RevenueCat-backed subscriptions; and a comprehensive, cross-linked observability stack with defined SLIs and SLOs. A single SvelteKit codebase ships to both web and native iOS through Capacitor. Measured against the five objectives of Section 1.1, the project met them substantially — the problem was validated and turned into requirements, the application and its recommendation engine were built and deployed, and the system is maintainable, type-safe, and observable — with automated testing and continuous deployment consciously sequenced into the next hardening phase rather than omitted by oversight.

6.3 Market Contribution

The project matters commercially and to its community. It validated, with evidence, that a large, active, globally connected, and long-underserved community faces a severe, recurring problem, and it showed the solution is financially viable — break-even at roughly 165 paying subscribers against modest operating costs, with an addressable market scaling to a substantial international opportunity. The validated assumptions — that dancers will adopt a structured tool, that both genders experience the problem, and that demand extends beyond Ukraine — de-risk the path from capstone to product. The entity model and roadmap (events, marketplace, community, live streams) position MatchUp to grow from a partner-search tool into the central digital platform for the DanceSport community.

6.4 Future Work

- **Hardening:** a formal automated test suite beginning with the recommendation tiers and safety filters, a CI test step, and continuous deployment across dedicated dev, staging, and production environments.
- **Completion:** finishing the coded-but-unwired pieces — the cron container and the marketplace screen — and populating the OWASP assessment and live SLO measurements.
- **Learned ranking:** moving the recommender from rule-based SQL to a learned model using the like-feature data already being collected.
- **Passwordless and SSO:** WebAuthn / passkeys and device biometrics on top of the in-place Google sign-in, plus Okta OIDC for club-operator single sign-on — all extending the existing provider-agnostic identity table.
- **Real-time and mobile reach:** migrating chat from three-second polling to WebSockets or server-sent events, enabling the commented-out cron container, and adding a native Android build.
- **Ecosystem and geographic expansion:** the events module as the retention and supply-side revenue flywheel, followed by marketplace, community feed, and live streams, and city-by-city then international growth enabled by configuration-driven replication.

6.5 Final Reflection

MatchUp demonstrates that a research-first approach to software engineering can yield both a validated business case and an architecture shaped by the specific realities of its market rather than by default convention. The decisions that define the system — a modular monolith over premature microservices, type-safe SQL over an ORM, one client codebase over two, a never-empty feed over a naive query, buy for the undifferentiated and build for the core, and an honest account of what is not yet done — were each made for a stated reason and against a stated trade-off. The project achieved its academic objective of producing transferable engineering knowledge and its practical objective of delivering a viable, deployed product to a community that is ready for exactly this kind of solution.

Bibliography

[1] The Go Programming Language. https://go.dev

[2] Gin Web Framework. https://gin-gonic.com

[3] PostgreSQL 18 Documentation. https://www.postgresql.org/docs/

[4] sqlc — Compile SQL to type-safe code. https://sqlc.dev

[5] SvelteKit. https://svelte.dev/docs/kit

[6] Capacitor. https://capacitorjs.com

[7] RevenueCat Documentation. https://www.revenuecat.com/docs

[8] Amazon Rekognition. https://aws.amazon.com/rekognition/

[9] OpenTelemetry. https://opentelemetry.io

[10] PostHog Documentation. https://posthog.com/docs

[11] OWASP Top 10. https://owasp.org/www-project-top-ten/

[12] S. Brown, The C4 model for visualising software architecture. https://c4model.com

Appendices

Per the KSE guidelines, secondary detail belongs in appendices to keep the main body within the page budget. Recommended inclusions:

- Appendix A — Full database schema and the like-log structure.
- Appendix B — The complete tiered recommendation query and representative module code.
- Appendix C — CI workflow file, Docker Compose, and Dockerfile definitions.
- Appendix D — Completed OWASP Top 10 assessment.
- Appendix E — Usability-testing protocol, findings, and resolutions.
- Appendix F — Survey instrument, interview guide, and research results.
- Appendix G — Full set of Figma UI screens.