IRON TRACKER

Design Document v1.0

Machine-Aware, Set-Centric Gym Tracking

February 2026

Status: Draft

Confidential

Table of Contents

1. Executive Summary

Iron Tracker is a mobile-first web application for gym-goers who train on multiple machines and want precision tracking of their lifts. Unlike every existing fitness app on the market, Iron Tracker treats the specific machine (not just the exercise name) as a first-class entity. A user performing Chest Press on a Hammer Strength plate-loaded unit, a Life Fitness selectorized stack, and a Cybex VR3 can track all three independently, with per-machine saved settings, separate weight histories, and distinct progress charts.

The app follows a set-centric design philosophy: the atomic unit of data is a single set, not a session or workout. Users can log a quick set of pull-ups between meetings without the ceremony of starting a workout. Sets logged within a configurable time window are automatically grouped into implicit sessions for history browsing.

This design document covers system architecture, data modeling, technology choices, analytics strategy, and AI integration for Iron Tracker v1.

2. Analysis of Shelved v0 (gym-tracker)

The v0 prototype (github.com/mmm00007/gym-tracker, 135 commits, 2 contributors) established the deployment topology and several core features. This analysis identifies what to carry forward, what to redesign, and what to drop.

2.1 Architecture Carried Forward

The v0 correctly established the three-tier deployment model: React SPA on Netlify, FastAPI on Render, and Supabase for auth plus PostgreSQL. The frontend performs direct CRUD via the Supabase JS client (protected by RLS), while the FastAPI backend serves as a lightweight proxy for the Anthropic Claude API, keeping the API key server-side. This split remains optimal and should be preserved in v1.

2.2 Features to Evolve

AI Machine Identification: The v0 used Claude to identify machines from photos and extract exercise names, target muscles, and form tips. This is a strong differentiator. In v1, this should feed directly into the machine variant creation flow rather than existing as a standalone feature.

Machine Library: The v0 stored machines per-user with editable fields. In v1, this evolves into the parent exercise + equipment variant hierarchy described in Section 4.

Set Logging with Sliders: The v0 used large sliders and quick-adjust controls for sweaty-hands usability. Competitive analysis shows that direct numeric input with stepper buttons is faster than sliders. The v1 should replace sliders with a numpad bottom-sheet plus inline stepper buttons.

Auto Rest Timer: The v0 implemented an auto-starting rest timer with saved rest times. This should persist, with per-exercise-type defaults added.

Soreness Tracking: The v0 prompted users 1-3 days post-workout for muscle soreness (0-4 scale) and fed this into AI analysis. This is valuable training data for the recovery model and should continue.

2.3 Architectural Issues to Resolve

Feature Flags as Env Vars: The v0 used VITE environment variables (SET_CENTRIC_LOGGING, LIBRARY_SCREEN_ENABLED, ANALYSIS_ON_DEMAND_ONLY) as feature flags, requiring redeployment for changes. The v1 should use a runtime feature flag system (Supabase edge config or a simple flags table).

Session-Set Duality: The v0 maintained both a sessions table and a sets table, with sessions marked as non-authoritative in Phase 1. The v1 should resolve this by making sets the single source of truth, with sessions as a derived/computed grouping.

Schema Coupling: The v0 schema had recommendation_scopes and analysis_reports tables tightly coupled to Claude-specific output formats. The v1 should decouple AI output schemas from core data models to allow swapping AI providers.

No Offline Support: The v0 had no offline capability. Given gym WiFi conditions, v1 must implement optimistic updates and offline queue from day one via TanStack Query persistence.

3. System Architecture

3.1 Deployment Topology

The system uses a three-tier architecture optimized for cost, developer velocity, and gym-environment constraints (poor connectivity, sweaty hands, brief interactions).

Layer	Technology	Responsibility	Cost
Frontend	React + Vite + TanStack	All UI, auth, direct Supabase CRUD, offline queue	Netlify free tier
Backend	FastAPI + Pydantic	AI proxy, analytics pre-computation, cron jobs, webhook handlers	Render $7/mo (avoid cold starts)
Database	Supabase (PostgreSQL 15)	Data storage, auth, RLS policies, edge functions for simple triggers	Free tier (500 MB)
AI	Anthropic Claude Sonnet	Machine identification, coaching, session insights	$0.01-0.05/interaction
Cache	Redis (Upstash)	Analytics cache, rate	Free tier (10K

Layer	Technology	Responsibility	Cost
		limiting, session state	cmds/day)

3.2 Data Flow Patterns

Pattern A — Direct CRUD (80% of operations): Frontend → Supabase JS Client → PostgreSQL (RLS-enforced). Used for: logging sets, reading history, managing machine library. No backend involvement. Latency: <100ms.

Pattern B — AI-Proxied (15% of operations): Frontend → FastAPI → Claude API → FastAPI → Frontend. Used for: machine photo identification, session insights, coaching queries. The backend validates auth (Supabase JWT), rate-limits, and injects user context.

Pattern C — Background Compute (5% of operations): Cron / Webhook → FastAPI → Supabase RPC → PostgreSQL. Used for: daily analytics rollup, weekly progress reports, PR detection batch, recovery score recalculation. Triggered by Render cron or Supabase webhook on set insertion.

3.3 Frontend Architecture

Library	Version	Role
React	18.x	UI framework
TanStack Router	1.x	File-based routing, type-safe params, code-splitting
TanStack Query	5.x	Server state, caching, optimistic updates, offline persistence
MUI (Material UI)	6.x	Component library with custom MD3 theming
@material/material-color-utilities	latest	MD3 dynamic color generation from seed color
@supabase/supabase-js	2.x	Auth and direct DB access
Recharts	2.x	Primary charting (line, bar, scatter)
Nivo	0.87.x	Advanced visualizations (heatmap, radar, calendar)
Zustand	5.x	Local UI state (active workout, timer, preferences)
Workbox	7.x	Service worker for offline-first PWA capabilities

3.4 Backend Architecture

The FastAPI backend follows a domain-driven module structure with strict separation of concerns.

Module	Responsibility
auth/	JWT verification via Supabase JWKS, get_current_user dependency
ai/	Claude API proxy, prompt templates, response parsing, token budget enforcement
analytics/	Pre-computation endpoints, materialized view refresh triggers, PR detection
cron/	Scheduled tasks: daily rollup, weekly report generation, recovery score update
core/	Pydantic settings, Supabase client initialization, Redis connection pool

Configuration uses pydantic-settings with @lru_cache for singleton pattern. All Pydantic models use v2 ConfigDict(from_attributes=True) for seamless ORM/PostgREST response mapping.

3.5 Authentication and Security

Authentication is handled entirely by Supabase Auth on the frontend. The backend verifies JWTs using Supabase JWKS (asymmetric signing), extracting the user ID from the sub claim. Every user-facing database table has RLS policies enforcing auth.uid() = user_id. The FastAPI backend uses a service role key only for analytics cron jobs that aggregate across users (anonymized).

4. Data Model

The data model centers on the parent exercise + equipment variant hierarchy, which is the core differentiator of Iron Tracker.

4.1 Core Entity Hierarchy

Exercise (parent) → Equipment Variant (child) → Set (atomic record). An Exercise represents a movement pattern (e.g., Chest Press). An Equipment Variant represents a specific machine, dumbbell configuration, or barbell setup for that exercise. A Set is a single instance of reps at a weight on a specific variant.

4.2 Table Definitions

exercises: The canonical exercise library. Populated from a seed set of ~400 common exercises. Users can add custom exercises.

Column	Type	Notes
id	uuid PK	Default gen_random_uuid()
name	text NOT NULL	Display name: Chest Press, Squat, etc.
category	text	push, pull, legs, core, cardio
primary_muscles	text[]	Array of muscle group identifiers
secondary_muscles	text[]	Array of secondary muscles
movement_type	text	compound | isolation
is_custom	boolean	User-created exercises flagged separately
created_by	uuid FK	NULL for seed data, user_id for custom
created_at	timestamptz	Default now()

equipment_variants: Machine-specific instances of an exercise, owned by the user.

Column	Type	Notes
id	uuid PK	Default gen_random_uuid()
user_id	uuid FK	Owner (RLS enforced)
exercise_id	uuid FK	Parent exercise
name	text NOT NULL	User-facing label: Hammer Strength Plate-Loaded
equipment_type	text	machine_selectorized | machine_plate | cable | barbell | dumbbell | bodyweight | smith_machine | other
manufacturer	text	Brand name (nullable)
weight_increment	decimal	Minimum weight step (e.g., 2.5 for plates, 5 for stack)
weight_unit	text	kg | lb
seat_settings	jsonb	Saved positions: {seat_height: 3, back_pad: 2, ...}
notes	text	Free-form user notes (grip width, cable attachment, etc.)
photo_url	text	Optional machine photo (Supabase Storage)
is_default	boolean	Auto-selected when logging this exercise

Column	Type	Notes
last_used_at	timestamptz	For MRU sorting
created_at	timestamptz	Default now()

sets: The atomic unit of training data. Every logged set creates one row.

Column	Type	Notes
id	uuid PK	Default gen_random_uuid()
user_id	uuid FK	Owner (RLS enforced)
exercise_id	uuid FK	Parent exercise
variant_id	uuid FK	Equipment variant (nullable for quick-logs)
weight	decimal	Weight used
weight_unit	text	kg | lb
reps	integer	Repetitions completed
rpe	decimal	Rate of Perceived Exertion (6.0-10.0, nullable)
rir	integer	Reps In Reserve (0-5, nullable)
set_type	text	working | warmup | dropset | failure | amrap
rest_seconds	integer	Rest before this set (auto-captured)
duration_seconds	integer	Set duration for time-based exercises
tempo	text	Tempo notation e.g., 3-1-2-0 (nullable)
notes	text	Per-set notes
logged_at	timestamptz	Precise timestamp of logging
created_at	timestamptz	Default now()
session_group	date	Derived: DATE(logged_at) for daily grouping

sessions_view (materialized): Computed grouping of sets into logical sessions using a 90-minute inactivity gap.

personal_records: Tracks PRs across multiple dimensions (estimated 1RM, rep maxes at each weight, volume records). Updated incrementally on set insertion via a PostgreSQL trigger or Supabase edge function.

soreness_reports: Carried forward from v0. Muscle-specific soreness ratings (0-4) prompted 1-3 days post-workout.

analytics_cache: Pre-computed analytics stored as JSONB with a type discriminator (weekly_volume, exercise_1rm_trend, muscle_distribution, etc.) and a computed_at timestamp for cache invalidation.

4.3 Key Indexes

Performance-critical indexes include: sets(user_id, exercise_id, logged_at DESC) for exercise history, sets(user_id, logged_at DESC) for session timeline, equipment_variants(user_id, exercise_id, last_used_at DESC) for MRU variant selection, and personal_records(user_id, exercise_id, variant_id, record_type) for PR lookups. All RLS policy columns must be indexed.

5. Analytics Architecture

The analytics system uses a hybrid computation model: server-side for historical aggregations, client-side for active workout interactivity, with incremental updates to avoid recomputation.

5.1 Server-Side (Python/FastAPI)

Metric	Method	Trigger
Estimated 1RM trend	Epley formula on historical sets, smoothed	On set insert (per exercise)
Weekly volume per muscle group	SUM(weight * reps) grouped by ISO week	Daily cron, incremented on set insert
PR detection	Compare new set against personal_records table	On set insert (trigger)
Muscle recovery score	Time-decay model from last volume + soreness	On set insert + soreness report
Training frequency	COUNT(DISTINCT session_group) per period	Daily cron
Strength score	Normalized 1RM across key compounds vs population	Weekly cron

5.2 Client-Side (JavaScript/React)

Active workout volume (running total during session), chart filtering/zooming/date-range on pre-loaded data, set-by-set progression within a session, RPE/RIR calculations during logging. The threshold is approximately 1,000 data points: below this, compute client-side; above, use pre-computed server data.

5.3 Caching Strategy

Three layers: TanStack Query on the client (staleTime: 5 min for analytics, 0 for active workout), Redis on the server (time-bucketed partial objects via fastapi-cache2), and PostgreSQL materialized views for expensive cross-user aggregations refreshed on workout completion.

5.4 Charting Library Choices

Primary: Recharts. 24.8K GitHub stars, declarative React components, responsive by default. Used for 1RM trend lines, weekly volume stacked bars, and set scatter plots.

Secondary: Nivo. Used for muscle distribution donut charts, body heatmaps, frequency calendar heatmaps (GitHub-style), and radar charts for training balance.

6. AI Integration Strategy

The AI strategy follows a rule-based-first, LLM-coaching-later philosophy, informed by how the most successful fitness AI apps (Fitbod, Dr. Muscle, Alpha Progression) actually work: sophisticated deterministic algorithms with optional natural language explanations.

6.1 Phase 1: Deterministic Engine (Weeks 1-8)

Weight Progression: Track estimated 1RM via Epley formula (e1RM = weight x (1 + reps/30)). Calculate target weight from RPE charts. Apply progressive increases: 2-5 lbs upper body, 5-10 lbs lower body per session. Auto-regulate: if actual RPE exceeds target by 1+, reduce next session target.

Volume Optimization: Evidence-based defaults by goal. Hypertrophy: 10-20 sets/muscle/week. Strength: 5-12 sets/muscle/week. Track weekly volume trends and flag over/under-training against the target range.

Fatigue Management: Trigger deload recommendations after 4+ consecutive weeks of increasing volume, declining performance over 2 weeks, or average RPE exceeding 8.5. Deload prescription: reduce volume 40-60%, intensity 10-15%.

PR Detection: Real-time across multiple dimensions: new estimated 1RM, new rep max at specific weights (1RM, 3RM, 5RM, 8RM, 10RM), best set volume (weight x reps), most reps at any weight.

6.2 Phase 2: Machine Photo ID (Weeks 4-8, Carried from v0)

User photographs a gym machine. The image is sent to Claude via the FastAPI proxy. Claude returns: exercise name(s), equipment type, manufacturer guess, target muscles, form tips. The response feeds directly into the equipment variant creation flow, pre-filling fields and letting the user confirm/edit. Token budget: ~500 input (image) + 200 output tokens per identification, ~$0.004 per call.

6.3 Phase 3: LLM Coaching Layer (Weeks 9-16)

A natural language coaching interface powered by Claude Sonnet. The LLM does not make programming decisions; it explains and presents recommendations generated by the deterministic engine. Context injection: last 4 weeks of training data, current recovery scores, active PRs, user goals and preferences. Estimated cost at 1,000 daily active users with 2 queries/day: ~$45/month using Claude Sonnet.

Use Cases: Weekly progress report generation in natural language, answering questions about training data, explaining why a deload is recommended, suggesting exercise substitutions with rationale, form tip reminders based on exercise history.

6.4 Cold Start

New users complete a brief profile (experience level, primary goal, training frequency, available equipment types). Population-based 1RM estimates provide initial priors. Within the first 3-5 workouts, the system calibrates by observing actual performance across key compound movements.

7. Offline-First and Performance

Gym environments have notoriously unreliable WiFi. Iron Tracker treats offline as a first-class state, not an error condition.

TanStack Query Offline Persistence: The entire query cache persists to IndexedDB via PersistQueryClientProvider. Set networkMode: offlineFirst on all mutations. Mutations queue when offline and replay automatically when connectivity returns.

Optimistic Updates: Every set log uses TanStack Query optimistic updates: onMutate immediately adds the set to the local cache, onError rolls back, onSettled invalidates to sync with server. Users see their set logged instantly regardless of network.

Service Worker (Workbox): Pre-caches the app shell, exercise library, and user machine library. Runtime caching for API responses with stale-while-revalidate strategy. Push notifications for rest timer expiration on lock screen.

Performance Budget: First Contentful Paint < 1.5s on 4G. Time to Interactive < 3s. Lighthouse score > 90. Bundle size < 200KB gzipped for initial route.

8. Open Questions: Recommendations

8.1 Server-Side vs Client-Side Analytics

Recommendation: hybrid approach as detailed in Section 5. Pre-compute historical aggregations server-side (1RM trends, weekly volumes, recovery scores, PRs). Compute active-session metrics client-side (running volume, set-to-set progression). The crossover threshold is approximately 1,000 data points. A typical user generates 3,600-5,200 sets per year, meaning single-exercise histories stay client-computable for ~1 year, while cross-exercise and multi-year analytics should always be pre-computed. Use incremental computation on every set insert rather than batch recomputation.

8.2 AI Integration Approach

Recommendation: the three-phase approach in Section 6. Start with a deterministic rule-based engine (this is what Fitbod and Alpha Progression actually use under their AI marketing). Add machine photo ID as a differentiating onboarding feature. Layer LLM coaching on top as a premium feature for natural language interaction with training data. Avoid building custom ML models until 10,000+ users provide sufficient collaborative filtering data. The deterministic engine handles 90%+ of recommendation value at zero marginal cost per user.

9. Risk Register

Risk	Impact	Likelihood	Mitigation
Machine selection adds logging friction	High	Medium	Progressive disclosure: hide picker with 1 variant, pre-select MRU, one-tap chip switching
Render cold starts degrade AI response time	Medium	High	$7/mo always-on instance; frontend shows skeleton/loading state
Supabase free tier 500MB exhausted	High	Low	Sets table grows ~50KB/user/month; 10K users = ~6GB/year. Plan upgrade at ~8K users
Offline sync conflicts	Medium	Medium	Last-write-wins with logged_at timestamp; conflict UI for rare edge cases
Claude API cost overrun	Medium	Low	Token budget caps per interaction; rate limiting

Risk	Impact	Likelihood	Mitigation
			per user; caching repeated queries

10. Technology Stack Summary

Category	Choice	Rationale
Frontend Framework	React 18 + Vite	Ecosystem maturity, team familiarity, fast HMR
Routing	TanStack Router	Type-safe, file-based, integrated query preloading
Server State	TanStack Query v5	Offline persistence, optimistic updates, cache management
UI Components	MUI v6 + MD3 Theme	50+ components, massive ecosystem, MD3 approximation via custom theme
Local State	Zustand	Lightweight, minimal boilerplate for timer/active workout state
Backend	FastAPI + Pydantic v2	Async-native, auto-docs, type safety, pydantic-settings for config
Database	Supabase (PostgreSQL 15)	Auth + RLS + PostgREST + Realtime + Storage in one platform
DB Client (Backend)	supabase-py (async)	Native async support, PostgREST + RPC, matches frontend client API
AI	Anthropic Claude Sonnet	Vision capability for machine ID, strong reasoning for coaching
Charts	Recharts + Nivo	Recharts for standard charts, Nivo for heatmaps/radar
Cache	Redis (Upstash)	Server-side analytics cache, rate limiting
Hosting	Netlify + Render + Supabase	Free/low-cost, CI/CD built-in, matches v0 topology

End of Design Document.