SPEC-1-TCG Stats Tracker

Background

You want a data-gathering application where a user can maintain multiple "projects". Each project tracks competitive trading card game (TCG) results—starting with Riftbound TCG—with the user entering matchup names and scores. The app should compute statistics like overall win rate, matchup win rates, and first-vs-second (draw/play) win rates. Future versions should support multiple TCGs beyond Riftbound while preserving per-project segregation and analytics.

This design aims to:

- Provide a clean, minimal input workflow so users quickly record statistics.
- Calculate meaningful stats in real time without manual spreadsheets.
- Be extensible to other TCGs with configurable game rules and fields.

Requirements

MoSCoW Priorities

Must Have

- Multi-project workspace: create, rename, archive projects; each project isolates data and settings.
- Game scope: initial support for **Riftbound TCG**; project stores which TCG it targets.
- Entry model: **per-matchup statistic entry** (not best-of-N). Each entry contains:
- **Side A** label (e.g., your deck/archetype/faction)
- **Side B** label (opponent deck/archetype/faction)
- **Result**: Win / Loss / Draw (draws allowed)
- **Initiative**: You **went first** or **went second** (required)
- **Battlefield** (required for Riftbound; driven by TCG config list)
- **Category** (dropdown; required; user-defined per project)
- Optional notes/opponent/event metadata
- Stats auto-calculated per project with filters (category, battlefield, initiative):
- Overall win rate
- **Matchup X vs Y** win rate (global + by initiative)
- **Went first vs went second** win rates across all entries
- Battlefield-specific splits (e.g., "going 2nd on Battlefield Z")
- Simple data entry UX optimized for speed (hotkeys, last-used defaults).
- PWA: offline capture queue → sync when online; conflict-free upsert by entry ID.
- Auth: email/password and social (Google); private user data per account.
- Export to CSV; import from CSV with validation and dry-run preview.

Should Have

- Tagging system (freeform tags in addition to Categories).

- Saved filters and views (e.g., "Big Tournaments • last 90 days").
- Charting dashboards (time series, matchup bars, first/second split pies).
- Per-TCG configuration: predefined archetype lists, battlefields, initiatives if applicable.
- Per-project sharing (read-only link) and team collaborators (owner, editor, viewer).

Could Have

- Opponent directory (aggregate stats vs specific opponents).
- Event mode (enter event, rounds, tables; still stored as entries underneath).
- Quick import from Google Sheets template.
- API access token for programmatic entry submission.

Won’t Have (MVP)

- Complex tournament bracket management.
- Real-time live scoring or spectator mode.
- AI-driven matchup recommendations.

Non-Functional

- Configurability: `tcg.settings_json` controls **context field** (label + required). Default **disabled** for new non-Riftbound TCGs.
- Storage: Supabase **Storage** bucket `deck-images` set to **public-read** for MVP, with path convention `user/{userId}/project/{projectId}/deck/{deckId}/{filename}`; plan to migrate to **signed URLs** later.
- Privacy & Security: Row-Level Security; all project data scoped to owner by default.
- Performance: dashboard loads < 1s on 50k entries; imports up to 100k rows.
- Reliability: offline-first with background sync; no data loss on app refresh while offline.
- Observability: basic error logging and client metrics.
- Accessibility: WCAG 2.1 AA for core flows.

Method

High-Level Architecture

- **Frontend (PWA):** Next.js + React; client renders dashboards and a fast "Add Entry" form optimized for keyboard use. Offline capture queue stores entries in IndexedDB; Background Sync posts queued mutations when online.
- **Backend:** Supabase (Postgres, Auth, Storage). REST (PostgREST) and RPC for custom SQL; Row-Level Security (RLS) ensures per-user/project data isolation.
- **Data Model Extensibility:** A **TCG configuration** layer makes fields like *battlefield* required for Riftbound while remaining optional or renamed for future TCGs.

Domain Model (MVP)

- **users** — managed by Supabase Auth.
- **projects** `(id, owner_user_id, name, tcg_id FK, archived_at)`

- **tcg** `(id, name, settings_json)` — e.g., `{ battlefieldRequired: true, initiativeEnum: ["FIRST","SECOND"], deckLabel: "Deck" }`.
- **decks** `(id, project_id FK, tcg_id FK, name, active)` — user-defined deck names per project/TCG.
- **battlefields** `(id, tcg_id FK, name, active)` — global per TCG (Riftbound required).
- **categories** `(id, project_id FK, name, active)` — dropdown values per project.
- **matchup_entries** (core per-entry statistics row; **no date/time field**):
- `id UUID` (client-generated for offline)
- `project_id FK`
- `tcg_id FK`
- `deck_a_id FK` ("you")
- `deck_b_id FK` ("opponent")
- `result ENUM('WIN','LOSS','DRAW')` (DRAW kept for future-proofing)
- `initiative ENUM('FIRST','SECOND')` (required)
- `battlefield_id FK` (**required if TCG.battlefieldRequired = true**)
- `category_id FK` (dropdown; required)
- `notes_short TEXT` (small inline note optional)
- **matchup_notes** (long-form notebook surfaced next to analytics):
- `id, project_id, tcg_id, deck_a_id, deck_b_id, content_markdown TEXT, updated_at`
- One row per **(project, deck A, deck B)** pair; app appends/edits this rich note and shows it beside the stats.

  Rationale

  - Keeping *notes* separate from entries allows a single running notebook per matchup, exactly as requested.
  - Categories are per-project dropdowns; battlefields are per-TCG lists; decks are per-project to allow custom labeling.

Access Control (RLS Sketch)

- All tables include `project_id` and policies restrict access to users that **own or are members** of the project. (Team roles can be added later.)

Indexing

- `matchup_entries(project_id, deck_a_id, deck_b_id)`
- `matchup_entries(project_id, initiative)`
- `matchup_entries(project_id, battlefield_id)`
- `categories(project_id)` and `decks(project_id)`

Stats Computation (SQL-first)

All stats are computed by indexed `GROUP BY` queries; if needed, we can add a materialized view `mv_matchup_stats` later.

Overall win rate (project scope):

```
SELECT
  SUM(CASE WHEN result='WIN'  THEN 1 ELSE 0 END)::decimal / NULLIF(COUNT(*),0)
AS win_rate
FROM matchup_entries
WHERE project_id = $1;
```

Matchup X vs Y win rate (global):

```
SELECT
  SUM((result='WIN')::int)::decimal / NULLIF(COUNT(*),0) AS win_rate
FROM matchup_entries
WHERE project_id=$1 AND deck_a_id=$2 AND deck_b_id=$3;
```

First vs Second split:

```
SELECT initiative,
  SUM((result='WIN')::int)::decimal / NULLIF(COUNT(*),0) AS win_rate,
  COUNT(*) AS games
FROM matchup_entries
WHERE project_id=$1
GROUP BY initiative;
```

Battlefield split for a given matchup (incl. going 2nd):

```
SELECT b.name AS battlefield,
  SUM((result='WIN')::int)::decimal / NULLIF(COUNT(*),0) AS win_rate,
  COUNT(*) AS games
FROM matchup_entries me
JOIN battlefields b ON b.id = me.battlefield_id
WHERE me.project_id=$1 AND me.deck_a_id=$2 AND me.deck_b_id=$3 AND
me.initiative='SECOND'
GROUP BY b.name
ORDER BY games DESC;
```

UI Flow (MVP)

1. **Project Home** → Select TCG (Riftbound) → app loads that TCG's config (battlefield required).
2. **Add Entry** (single screen):
3. Deck A (select or quick-create) • Deck B (select or quick-create)
4. Result (Win/Loss[/Draw]) • Initiative (First/Second)
5. Battlefield (required for Riftbound) • Category (dropdown)
6. [Add] → Queued offline if needed.
7. **Analytics**

8. Tabs: *Overview*, *Matchups*, *First vs Second*, *Battlefields*.
9. On selecting a **matchup (X vs Y)**, show: overall WR, split by initiative, split by battlefield, and the **Matchup Notebook** side panel.

PlantUML — Component & Data

```
@startuml
skinparam componentStyle rectangle
rectangle "PWA Frontend (Next.js)" as FE {
  [Add Entry Form]
  [Analytics Dashboards]
  [Matchup Notebook]
}

node "Supabase" as SB {
  database "Postgres" as PG
  [Auth]
  [Storage]
}

FE --> PG : REST/RPC (PostgREST)
FE --> Auth : Auth (JWT)

PG --> "Tables:
projects, tcg, decks, battlefields, categories,
matchup_entries, matchup_notes"
@enduml
```

```
@startuml
entity projects {
  *id: uuid
  owner_user_id: uuid
  name: text
  tcg_id: uuid
}
entity tcg {
  *id: uuid
  name: text
  settings_json: jsonb
}
entity decks {
  *id: uuid
  project_id: uuid
  tcg_id: uuid
  name: text
}
```

```
entity battlefields {
  *id: uuid
  tcg_id: uuid
  name: text
}
entity categories {
  *id: uuid
  project_id: uuid
  name: text
}
entity matchup_entries {
  *id: uuid
  project_id: uuid
  tcg_id: uuid
  deck_a_id: uuid
  deck_b_id: uuid
  result: enum
  initiative: enum
  battlefield_id: uuid
  category_id: uuid
  notes_short: text
}
entity matchup_notes {
  *id: uuid
  project_id: uuid
  tcg_id: uuid
  deck_a_id: uuid
  deck_b_id: uuid
  content_markdown: text
}

projects ||--o{ decks
projects ||--o{ categories
projects ||--o{ matchup_entries
projects ||--o{ matchup_notes

tcg ||--o{ battlefields

tcg ||--o{ projects

decs ||--o{ matchup_entries : deck_a_id
"decks (as opponent)" ||--o{ matchup_entries : deck_b_id
battlefields ||--o{ matchup_entries
categories ||--o{ matchup_entries

@enduml
```

Extending to Other TCGs

- Add a new `tcg` row with its settings and battlefield list (or mark `battlefieldRequired=false`).
- Keep the same `matchup_entries` shape; only the UI changes required fields and picklists based on `tcg.settings_json`.

Updates Based on Your Feedback

- **Context field generalized:** Instead of a hard-coded `battlefield`, we define a per-TCG **context field** that is configurable via `tcg.settings_json` (e.g., label and required). For **Riftbound**, the context field is labeled **Battlefield** and is **required**; for other TCGs it can be relabeled (e.g., "Format") or disabled.
- **Schema changes:**
- Replace `battlefield_id` with `context_option_id` referencing a new table `tcg_context_options`.
- Replace single `matchup_notes` with `matchup_notes_log` to support **multiple timestamped notes** per matchup.
- **UI changes:** The Add Entry form shows a required **Battlefield** dropdown for Riftbound; other TCGs show the per-TCG label (or hide if disabled). The **Matchup Notebook** becomes a timeline with note cards (author, created_at), plus search and pin.

New/Updated DDL (supersedes earlier snippets where different)

```
-- Per-TCG selectable options for the generic context field (Battlefields in
Riftbound)
CREATE TABLE IF NOT EXISTS tcg_context_options (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  name text NOT NULL,
  active boolean NOT NULL DEFAULT true,
  UNIQUE(tcg_id, name)
);

-- Match entries now reference context_option_id instead of battlefield_id
CREATE TYPE IF NOT EXISTS result_enum AS ENUM ('WIN','LOSS','DRAW');
CREATE TYPE IF NOT EXISTS initiative_enum AS ENUM ('FIRST','SECOND');

CREATE TABLE IF NOT EXISTS matchup_entries (
  id uuid PRIMARY KEY,
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  deck_a_id uuid NOT NULL REFERENCES decks(id),
  deck_b_id uuid NOT NULL REFERENCES decks(id),
  result result_enum NOT NULL,
  initiative initiative_enum NOT NULL,
  context_option_id uuid REFERENCES tcg_context_options(id), -- required if
tcg.settings_json.contextRequired=true
```

```
  category_id uuid NOT NULL REFERENCES categories(id),
  notes_short text
);

-- Notes log: multiple timestamped notes per matchup
CREATE TABLE IF NOT EXISTS matchup_notes_log (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  deck_a_id uuid NOT NULL REFERENCES decks(id),
  deck_b_id uuid NOT NULL REFERENCES decks(id),
  author_user_id uuid NOT NULL,
  content_markdown text NOT NULL,
  created_at timestamptz NOT NULL DEFAULT now(),
  updated_at timestamptz NOT NULL DEFAULT now()
);

-- Update timestamp on edit
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
  NEW.updated_at = now();
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_notes_updated_at ON matchup_notes_log;
CREATE TRIGGER trg_notes_updated_at
BEFORE UPDATE ON matchup_notes_log
FOR EACH ROW EXECUTE FUNCTION set_updated_at();

-- Helpful indexes
CREATE INDEX IF NOT EXISTS idx_entries_project_matchup ON
matchup_entries(project_id, deck_a_id, deck_b_id);
CREATE INDEX IF NOT EXISTS idx_entries_initiative ON
matchup_entries(project_id, initiative);
CREATE INDEX IF NOT EXISTS idx_entries_context ON matchup_entries(project_id,
context_option_id);

-- Seed Riftbound context options as Battlefields (example; replace with
canonical list)
INSERT INTO tcg (name, settings_json)
VALUES ('Riftbound', '{"contextLabel":"Battlefield","contextRequired":true}')
ON CONFLICT (name) DO NOTHING;

INSERT INTO tcg_context_options (tcg_id, name)
SELECT id, b.name FROM tcg, (VALUES
  ('Crystal Spires'),
```

```
  ('Sunken Rifts'),
  ('Obsidian Flats')
) AS b(name)
WHERE tcg.name='Riftbound'
ON CONFLICT DO NOTHING;
```

Stats SQL using the new context field (e.g., Going 2nd by Battlefield):

```
SELECT co.name AS context_label,
  SUM((result='WIN')::int)::decimal / NULLIF(COUNT(*),0) AS win_rate,
  COUNT(*) AS games
FROM matchup_entries me
JOIN tcg_context_options co ON co.id = me.context_option_id
WHERE me.project_id=$1 AND me.deck_a_id=$2 AND me.deck_b_id=$3 AND
me.initiative='SECOND'
GROUP BY co.name
ORDER BY games DESC;
```

Implementation

1) Database (Supabase Postgres)

Enums

```
CREATE TYPE result_enum AS ENUM ('WIN','LOSS','DRAW');
CREATE TYPE initiative_enum AS ENUM ('FIRST','SECOND');
```

Core Tables

```
-- tcg + config
CREATE TABLE tcg (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  name text NOT NULL UNIQUE,
  settings_json jsonb NOT NULL DEFAULT '{}'::jsonb
);

CREATE TABLE projects (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  owner_user_id uuid NOT NULL,
  name text NOT NULL,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  archived_at timestamptz
);
```

```sql
CREATE TABLE decks (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  name text NOT NULL,
  image_path text, -- path in Supabase Storage bucket 'deck-images'
  active boolean NOT NULL DEFAULT true,
  UNIQUE(project_id, name)
);

CREATE TABLE battlefields (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  name text NOT NULL,
  active boolean NOT NULL DEFAULT true,
  UNIQUE(tcg_id, name)
);

CREATE TABLE categories (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  name text NOT NULL,
  active boolean NOT NULL DEFAULT true,
  UNIQUE(project_id, name)
);

CREATE TABLE matchup_entries (
  id uuid PRIMARY KEY,
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  deck_a_id uuid NOT NULL REFERENCES decks(id),
  deck_b_id uuid NOT NULL REFERENCES decks(id),
  result result_enum NOT NULL,
  initiative initiative_enum NOT NULL,
  battlefield_id uuid NOT NULL REFERENCES battlefields(id),
  category_id uuid NOT NULL REFERENCES categories(id),
  notes_short text
);

CREATE TABLE matchup_notes (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  project_id uuid NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  tcg_id uuid NOT NULL REFERENCES tcg(id),
  deck_a_id uuid NOT NULL REFERENCES decks(id),
  deck_b_id uuid NOT NULL REFERENCES decks(id),
  content_markdown text NOT NULL DEFAULT '',
  updated_at timestamptz NOT NULL DEFAULT now(),
```

```
  UNIQUE(project_id, deck_a_id, deck_b_id)
);
```

Indexes

```
CREATE INDEX ON matchup_entries(project_id, deck_a_id, deck_b_id);
CREATE INDEX ON matchup_entries(project_id, initiative);
CREATE INDEX ON matchup_entries(project_id, battlefield_id);
CREATE INDEX ON categories(project_id);
CREATE INDEX ON decks(project_id);
```

Seed TCG + Riftbound battlefields (example)

```
INSERT INTO tcg (name, settings_json)
VALUES ('Riftbound', '{"battlefieldRequired": true, "initiativeEnum":
["FIRST","SECOND"], "deckLabel": "Deck"}')
ON CONFLICT (name) DO NOTHING;

-- Replace with your canonical Riftbound battlefield list
INSERT INTO battlefields (tcg_id, name)
SELECT id, b.name FROM tcg, (VALUES
  ('Crystal Spires'),
  ('Sunken Rifts'),
  ('Obsidian Flats')
) AS b(name)
WHERE tcg.name='Riftbound'
ON CONFLICT DO NOTHING;
```

RLS Policies (sketch)

```
-- Enable RLS
ALTER TABLE projects ENABLE ROW LEVEL SECURITY;
ALTER TABLE decks ENABLE ROW LEVEL SECURITY;
ALTER TABLE categories ENABLE ROW LEVEL SECURITY;
ALTER TABLE matchup_entries ENABLE ROW LEVEL SECURITY;
ALTER TABLE matchup_notes ENABLE ROW LEVEL SECURITY;

-- Helper: membership via owner for MVP
CREATE POLICY project_is_owner ON projects
  USING (auth.uid() = owner_user_id);

CREATE POLICY decks_project_owner ON decks
  USING (EXISTS (SELECT 1 FROM projects p WHERE p.id=project_id AND
p.owner_user_id=auth.uid()));
```

```
CREATE POLICY categories_project_owner ON categories
  USING (EXISTS (SELECT 1 FROM projects p WHERE p.id=project_id AND
p.owner_user_id=auth.uid()));
CREATE POLICY entries_project_owner ON matchup_entries
  USING (EXISTS (SELECT 1 FROM projects p WHERE p.id=project_id AND
p.owner_user_id=auth.uid()));
CREATE POLICY notes_project_owner ON matchup_notes
  USING (EXISTS (SELECT 1 FROM projects p WHERE p.id=project_id AND
p.owner_user_id=auth.uid()));
```

Server-side validations (Postgres CHECKs)

```
ALTER TABLE matchup_entries
  ADD CONSTRAINT chk_decks_same_project
  CHECK (
    deck_a_id <> deck_b_id AND
    project_id = (SELECT project_id FROM decks da WHERE da.id = deck_a_id) AND
    project_id = (SELECT project_id FROM decks db WHERE db.id = deck_b_id)
  );
```

RPC Helpers (optional for crisp APIs)

```
CREATE OR REPLACE FUNCTION api_matchup_winrate(p_project uuid, p_deck_a uuid,
p_deck_b uuid)
RETURNS TABLE(win_rate numeric, games int) AS $$
  SELECT COALESCE(SUM((result='WIN')::int)::numeric / NULLIF(COUNT(*),0),0),
COUNT(*)
  FROM matchup_entries
  WHERE project_id=p_project AND deck_a_id=p_deck_a AND deck_b_id=p_deck_b;
$$ LANGUAGE sql STABLE;
```

2) Backend Access

- Use Supabase JS client for Auth and row-level REST access; Prisma optional for migrations/queries in Server Actions.
- All writes occur via **Server Actions** that validate input against Zod and enforce user ownership.

3) Frontend (Next.js PWA)

- **Screens**: Projects List → Project Dashboard → Add Entry → Matchup Analytics (with Notebook side panel).
- **Add Entry Form**: deckA, deckB, result (Win/Loss/Draw), initiative (First/Second), battlefield/context (required for Riftbound), category (required), notes (short).
- **Deck Images**: deck create/edit dialog supports **image upload** (png/jpg/webp ≤ **1MB**; client downscales to \~800×800). Preview images appear in selectors and analytics cards.

- **Offline**: Queue mutations to IndexedDB; Background Sync flushes when reconnected.
- **Charts**: Bar (matchup WR), Pie (first vs second), Table (context splits). Percentages show **one decimal** (e.g., 57.1%).

File Storage (Supabase Storage)

- Bucket: `deck-images` (MVP **public-read**). Later: switch to signed URLs & per-object policies.
- Path: `user/{userId}/project/{projectId}/deck/{deckId}/{filename}`.
- On upload: return public URL → save to `decks.image_path`.
- Validation: MIME type `image/*`, size ≤ 1MB; client compression before upload.

4) Validation (Zod)

- Schema mirrors DB: enums for result/initiative; foreign keys validated by existence queries.
- Prevent creating an entry if required picklists (battlefield, category) are empty.

5) Testing

- Unit: Zod schemas and stat calculators.
- Integration: DB RPCs and RLS with authenticated test users.
- E2E: Playwright flows (create project → decks/categories → add entries → analytics load).

6) Deployment

- Supabase project (managed Postgres) + Vercel/Netlify for Next.js.
- Environment: production + staging; database migrations via Prisma or Supabase CLI.

7) Observability

- Client error reporting (e.g., Sentry) and simple usage events (entries added, analytics viewed).

Milestones

M0 – Project Setup (1–2 days)

- Repos, CI, environments (staging/prod), Supabase project, Prisma/Supabase CLI.

M1 – Core Schema & RLS (2–3 days)

- Create enums/tables (projects, tcg, decks, categories, tcg_context_options, matchup_entries, matchup_notes_log).
- RLS policies for owner-only access.
- Seed **Riftbound** with contextLabel=Battlefield and canonical options.

M2 – Add Entry Flow (3–4 days)

- Add Entry form (deckA, deckB, Win/Loss/Draw, First/Second, **Battlefield** required for Riftbound, Category required, notes_short).
- Quick-create for decks/categories with validations.

- Offline queue + background sync.

M3 – Analytics v1 (3–4 days)

- Overall WR, Matchup WR (X vs Y), First vs Second splits, Context (Battlefield) splits.
- Saved view: per-matchup panel.

M4 – Matchup Notebook (2 days)

- Notes log UI beside analytics; create/edit notes; search, pin.

M5 – Exports & Imports (2 days)

- CSV export; CSV import with dry-run validation.

M6 – Hardening & A11y (2 days)

- Error handling, loading states, basic WCAG checks, E2E tests.

M7 – Beta Cut & Feedback (ongoing)

- Gather user feedback; backlog for v2 (sharing, tags, saved filters).

Gathering Results

Success Metrics (MVP)

- **Entry speed:** median < 8 seconds from opening form to saved entry.
- **Analytics correctness:** sampled queries match expected SQL on seed datasets (±0.1%).
- **Category coverage:** ≥ 95% of entries have a category (enforced by required dropdown).
- **Display precision:** percentages rendered to **one decimal** with half-up rounding.
- **Offline reliability:** ≥ 99% of offline-queued entries successfully sync within 5 minutes of reconnect.
- **Image quality:** upload success rate ≥ 99%; average deck image size ≤ 200KB after client compression.
- **Performance:** analytics view loads < 1s on 50k entries; thumbnail cache hits < 200ms.

Validation Plan

- Unit tests for calculators vs fixture CSVs.
- Golden SQL snapshots for RPCs (overall WR, X vs Y, initiative splits, context splits).
- Usability tests with 3–5 users: task time, error rate, SUS ≥ 80.
- Observability dashboards: error rates, sync queue depth, upload failures.