# JumpShot

# CS2 Performance Analytics Platform

## Complete Product Requirements Document

Comprehensive Technical Specification & Feature Documentation

## 

## PLATFORM OVERVIEW

A revolutionary Counter-Strike 2 performance analytics ecosystem that transforms raw competitive match data into actionable intelligence through advanced statistical modeling, machine learning algorithms, and comprehensive visualization tools. The platform serves professional teams, coaches, analysts, scouts, and esports organizations with deep insights into player performance, team dynamics, match prediction, and strategic optimization.

IMPLEMENTATION DEPTH REQUIREMENTS

* All features must be fully functional, not placeholder implementations
* No "TODO" comments or half-finished functionality in deliverables
* Every component must handle loading, error, and success states properly
* Database queries must be optimized and return actual structured data

RESEARCH & PLANNING MANDATES

* You must analyze existing codebase structure before making changes
* Quote specific PRD sections and confirm understanding before coding
* Ask clarifying questions about edge cases and error scenarios
* Provide implementation plan before starting work

QUALITY STANDARDS

* Code must pass TypeScript compilation without errors
* All API endpoints must include proper error handling and validation
* UI components must be responsive and accessible
* No hardcoded values - use proper configuration and environment variables

VERIFICATION REQUIREMENTS

* Test each feature thoroughly before marking as complete
* Demonstrate working functionality with actual data flows
* Document any limitations or assumptions explicitly
* Request user validation at each major milestone

ACCOUNTABILITY MEASURES

* Explain technical decisions and trade-offs made
* Justify why specific approaches were chosen over alternatives
* Admit when I don't know something rather than guessing
* Ask for feedback on implementation quality regularly

EXPLICIT PROHIBITION

* No "quick demos" with fake data
* No skipping proper authentication or error handling
* No assuming user requirements without confirmation
* No delivering incomplete features as "functional"

### 

### CORE TECHNOLOGY ARCHITECTURE

* Frontend Framework: React.js 18+ with TypeScript, Framer Motion animations, Recharts visualization library
* Backend Infrastructure: Node.js/Express server with TypeScript, real-time WebSocket connections
* Primary Database: Supabase (PostgreSQL) with complete relational schema and foreign key constraints
* Authentication System: Session-based authentication with role-based access control
* Data Processing Engine: Real-time PIV calculation system with role-specific metric evaluation
* Deployment Platform: Replit with automatic scaling and monitoring
* API Architecture: RESTful endpoints with comprehensive error handling and data validation

Database Modification Restrictions:

"NEVER INSERT, UPDATE, or DELETE any data in production database under any circumstances"

"NEVER populate tables with sample, mock, test, or synthetic data"

"NEVER assume database is empty or safe to modify without explicit written permission"

"ANY database write operation requires explicit approval in writing before execution"

"Development environment ONLY for data seeding - production databases are READ-ONLY"

Developer Constraints:

"Treat all database connections as production regardless of environment"

"Build interfaces with empty states FIRST - never populate data to demonstrate functionality"

"Database modifications are EXCLUSIVELY through approved ingestion pipeline"

"No direct SQL operations on tables containing business data"

Below is the schema definition for the tables:

kill\_stats:

create table public.kill\_stats (

steam\_id bigint not null,

kills integer null,

headshots integer null,

wallbang\_kills integer null,

no\_scope integer null,

through\_smoke integer null,

airbone\_kills integer null,

blind\_kills integer null,

victim\_blind\_kills integer null,

awp\_kills integer null,

pistol\_kills integer null,

first\_kills integer null,

ct\_first\_kills integer null,

t\_first\_kills integer null,

first\_deaths integer null,

ct\_first\_deaths integer null,

t\_first\_deaths integer null,

event\_id integer not null,

constraint kill\_stats\_pkey primary key (steam\_id, event\_id),

constraint unique\_kill\_event unique (steam\_id, event\_id),

constraint unique\_steam\_event\_kill\_stats unique (steam\_id, event\_id),

constraint fk\_kill\_stats\_event foreign KEY (event\_id) references events (event\_id),

constraint kill\_stats\_steam\_id\_fkey foreign KEY (steam\_id) references players (steam\_id)

) TABLESPACE pg\_default;

genral\_stats:

create table public.general\_stats (

steam\_id bigint not null,

assists double precision null,

deaths double precision null,

trade\_kills double precision null,

trade\_deaths double precision null,

kd double precision null,

k\_d\_diff double precision null,

adr\_total double precision null,

adr\_ct\_side double precision null,

adr\_t\_side double precision null,

kast\_total double precision null,

kast\_ct\_side double precision null,

kast\_t\_side double precision null,

total\_rounds\_won double precision null,

t\_rounds\_won double precision null,

ct\_rounds\_won double precision null,

event\_id integer not null,

constraint general\_stats\_pkey primary key (steam\_id, event\_id),

constraint unique\_general\_event unique (steam\_id, event\_id),

constraint unique\_steam\_event\_general\_stats unique (steam\_id, event\_id),

constraint fk\_general\_stats\_event foreign KEY (event\_id) references events (event\_id),

constraint general\_stats\_steam\_id\_fkey foreign KEY (steam\_id) references players (steam\_id)

) TABLESPACE pg\_default;

utility\_stats:

create table public.utility\_stats (

steam\_id bigint not null,

assisted\_flashes integer null,

flahes\_thrown integer null,

ct\_flahes\_thrown integer null,

t\_flahes\_thrown integer null,

flahes\_thrown\_in\_pistol\_round integer null,

he\_thrown integer null,

ct\_he\_thrown integer null,

t\_he\_thrown integer null,

he\_thrown\_in\_pistol\_round integer null,

infernos\_thrown integer null,

ct\_infernos\_thrown integer null,

t\_infernos\_thrown integer null,

infernos\_thrown\_in\_pistol\_round integer null,

smokes\_thrown integer null,

ct\_smokes\_thrown integer null,

t\_smokes\_thrown integer null,

smokes\_thrown\_in\_pistol\_round integer null,

util\_in\_pistol\_round integer null,

total\_util\_thrown integer null,

total\_util\_dmg integer null,

ct\_total\_util\_dmg integer null,

t\_total\_util\_dmg integer null,

event\_id integer not null,

constraint utility\_stats\_pkey primary key (steam\_id, event\_id),

constraint unique\_steam\_event\_utility\_stats unique (steam\_id, event\_id),

constraint unique\_utility\_event unique (steam\_id, event\_id),

constraint fk\_utility\_stats\_event foreign KEY (event\_id) references events (event\_id),

constraint utility\_stats\_steam\_id\_fkey foreign KEY (steam\_id) references players (steam\_id)

) TABLESPACE pg\_default;

teams:

create table public.teams (

id serial not null,

team\_clan\_name text not null,

constraint teams\_pkey primary key (id),

constraint teams\_team\_clan\_name\_key unique (team\_clan\_name)

) TABLESPACE pg\_default;

players:

create table public.players (

steam\_id bigint not null,

user\_name text null,

constraint players\_pkey primary key (steam\_id)

) TABLESPACE pg\_default;

player\_match\_summary:

create table public.player\_match\_summary (

steam\_id bigint not null,

file\_id integer not null,

team\_id integer null,

event\_id integer not null,

constraint player\_match\_summary\_pkey primary key (steam\_id, file\_id, event\_id),

constraint unique\_steam\_file\_event unique (steam\_id, file\_id, event\_id),

constraint fk\_player\_match\_summary\_event foreign KEY (event\_id) references events (event\_id),

constraint player\_match\_summary\_event\_id\_fkey foreign KEY (event\_id) references events (event\_id),

constraint player\_match\_summary\_file\_id\_fkey foreign KEY (file\_id) references matches (file\_id),

constraint player\_match\_summary\_steam\_id\_fkey foreign KEY (steam\_id) references players (steam\_id)

) TABLESPACE pg\_default;

rounds:

create table public.rounds (

id serial not null,

round\_num integer null,

start integer null,

freeze\_end integer null,

"end" integer null,

official\_end integer null,

winner text null,

reason text null,

bomb\_plant double precision null,

bomb\_site text null,

ct\_team\_clan\_name text null,

t\_team\_clan\_name text null,

winner\_clan\_name text null,

ct\_team\_current\_equip\_value double precision null,

t\_team\_current\_equip\_value double precision null,

ct\_losing\_streak integer null,

t\_losing\_streak integer null,

ct\_buy\_type text null,

t\_buy\_type text null,

advantage\_5v4 text null,

file\_id integer null,

event\_id integer null,

match\_name text null,

constraint rounds\_pkey primary key (id),

constraint unique\_round\_per\_match unique (round\_num, match\_name),

constraint rounds\_file\_id\_fkey foreign KEY (file\_id) references matches (file\_id)

) TABLESPACE pg\_default;

matches:

create table public.matches (

file\_id integer not null default nextval('matches\_file\_id\_seq'::regclass),

match\_name text not null,

event\_id integer null,

constraint matches\_pkey primary key (file\_id),

constraint unique\_file\_per\_event unique (match\_name, event\_id),

constraint matches\_event\_id\_fkey foreign KEY (event\_id) references events (event\_id)

) TABLESPACE pg\_default;

events:

create table public.events (

event\_id integer not null,

event\_name text not null,

constraint events\_pkey primary key (event\_id)

) TABLESPACE pg\_default;

player\_history:

create table public.player\_history (

id serial not null,

steam\_id bigint not null,

team\_id integer not null,

constraint player\_history\_pkey primary key (id),

constraint unique\_steam\_team unique (steam\_id, team\_id),

constraint fk\_steam foreign KEY (steam\_id) references players (steam\_id),

constraint fk\_team foreign KEY (team\_id) references teams (id)

) TABLESPACE pg\_default;

### 

### FUNDAMENTAL CALCULATION ARCHITECTURE

PIV represents the most comprehensive player evaluation metric in competitive Counter-Strike 2 analytics, incorporating role-specific performance, individual consistency, team synergy contribution, and contextual factors.

#### Master Formula:

PIV = (RCS × ICF × SC × OSM) + Basic\_Metrics\_Bonus + Situational\_Modifiers + Map\_Specific\_Adjustments

Where:

- RCS = Role Core Score (0.0 to 1.0)

- ICF = Individual Consistency Factor (0.0 to 2.0)

- SC = Synergy Contribution (0.0 to 1.0)

- OSM = Opponent Strength Multiplier (0.8 to 1.2)

- Basic\_Metrics\_Bonus = Fundamental statistical performance bonus (0.0 to 0.5)

- Situational\_Modifiers = Context-based adjustments (-0.2 to +0.3)

- Map\_Specific\_Adjustments = Map performance variance (-0.1 to +0.1)

### 

### T-SIDE ROLE DEFINITIONS & WEIGHTINGS

#### ENTRY FRAGGER (T-Side Primary Initiator)

Role Responsibility: First contact engagement, site execution leadership, creating space for team advancement

Opening Kill Success Rate30%

Multi-Kill Rounds25%

Trade Kill Efficiency20%

Site Penetration Success15%

Economy Impact10%

#### SUPPORT (T-Side Utility Coordinator)

Role Responsibility: Utility deployment, teammate assistance, trade fragging, tactical setup

Flash Assist Efficiency35%

Trade Kill Success30%

Utility Coordination Score20%

Team Setup Contribution15%

#### LURKER (T-Side Information & Flanking Specialist)

Role Responsibility: Information gathering, flanking maneuvers, late-round impact, map control

Information Gathering Efficiency40%

Flank Success Rate30%

Zone Influence Stability20%

Delayed Timing Effectiveness10%

#### AWPER (T-Side Precision Marksman)

Role Responsibility: Long-range eliminations, pick generation, map control establishment, economic impact

Pick Efficiency40%

Economy Impact25%

Map Control Contribution20%

Long-Range Accuracy15%

### CT-SIDE ROLE DEFINITIONS & WEIGHTINGS

#### ANCHOR (CT-Side Site Specialist)

Role Responsibility: Site defense, solo holds, retake coordination, economic preservation

Site Hold Success Rate35%

Retake Efficiency25%

Economic Conservation20%

Clutch Performance20%

#### ROTATOR (CT-Side Adaptive Defender)

Role Responsibility: Map rotation, backup support, flexible positioning, anti-execute defense

Rotation Efficiency Index40%

Adaptive Defense Score25%

Retake Utility Coordination20%

Anti-Execute Success15%

Data ingestion component

The parser lives outside Replit on a lightweight worker. It watches an **incoming** bucket for new .dem files, converts each one into a typed NDJSON file that mirrors our match schema, then streams the file into Supabase with a single COPY statement that ignores duplicates by honouring existing primary keys. When the transaction commits, the worker immediately refreshes **piv\_player\_round\_mv** so the analytics layer never lags behind the raw data.

After every successful load the worker writes a row to **ingest\_log** (event\_id, match\_id, rows\_loaded, processed\_at). The front-end surfaces this record so users can see the time of the last completed ingest.

Key separation keeps the surface tight  
 → SERVICE\_KEY lives only on the worker and can insert/select on the ingest tables  
 → ANON\_KEY lives in Replit and is limited to read-only access through stored procedures and materialised views

Before each deployment the pipeline runs **supabase db diff**.  
 → If the diff is not empty the load halts and posts a Slack alert, preventing mismatched writes

All data that reaches Replit is therefore fully normalised, indexed and fresh, so the UI layer never carries ingest risk or latency.

Metric normalisation

Context  
 All incoming events currently involve top-fifty teams so the skill spread is narrower than public match data. We still store both event-relative and global Z-scores from day one so that future lower-tier events can be added without rewriting historical metrics.

Definition  
 → z\_event  
 (value − μ\_event) ÷ σ\_event  
 The mean μ\_event and standard deviation σ\_event are computed on all maps inside the same event\_id.  
 → z\_global  
 (value − μ\_global) ÷ σ\_global  
 The global statistics table holds one row per metric with running aggregates across every processed event.

Schema update  
 Add two numeric columns to each fact or summary table that needs normalisation

alter table player\_round\_stats

add column z\_event double precision null,

add column z\_global double precision null;

Create a helper table for global aggregates

create table metric\_global\_stats (

metric\_name text primary key,

mu double precision not null,

sigma double precision not null,

refreshed\_at timestamptz not null default now()

);

Query selector  
 Expose a lightweight view that switches Z-scope by parameter so dashboards can decide at call time

create or replace function fn\_player\_round\_stats(scope text)

returns table (...) as $$

select \*,

case when scope = 'event' then z\_event

when scope = 'global' then z\_global

end as z\_active

from player\_round\_stats;

$$ language sql stable;

Refresh cadence  
 → Event Z-scores computed immediately after every match load  
 → Global aggregates rebuilt nightly by the ingest worker and written into metric\_global\_stats  
 → View automatically reflects the latest values without further code changes

Analytic drift guard  
 Document the above formulas and refresh cadence in this PRD. Future analysts must update both the helper table and all historical rows when retro-ingesting older events or changing tier definitions.

Road-map note  
 When event tiering begins, maintain separate μ\_tier and σ\_tier in metric\_global\_stats. The same view can be extended with a scope = ‘tier’ branch without altering existing front-end queries.

**Sample-size guardrails**

New players and short stand-ins often appear to “break” the PIV leader-board because their small number of rounds inflates variance. We anchor every individual contribution factor (ICF) to the population average with a Bayesian shrinkage rule and expose an uncertainty band so scouts immediately see how reliable the figure is.

Shrinkage logic  
 → prior mean µ₀ is the rolling global average ICF across all events  
 → constant k ≈ 500 represents the round count at which a newcomer’s own data outweighs the prior  
 → observed sample mean x̄ and sample size n produce a posterior mean

ini

CopyEdit

w = n / (n + k) -- weight of the player’s own data

ICF\_mean = w · x̄ + (1 − w) · µ₀

Confidence interval  
 → posterior standard error σ\_post = s / √(n + k) where s is the sample standard deviation  
 → 95 % limits

ini

CopyEdit

ICF\_ci\_lower = ICF\_mean − 1.96 · σ\_post

ICF\_ci\_upper = ICF\_mean + 1.96 · σ\_post

Schema update

pgsql

CopyEdit

alter table player\_match\_summary

add column icf\_mean double precision null,

add column icf\_ci\_lower double precision null,

add column icf\_ci\_upper double precision null;

Store µ₀ and k in helper table metric\_global\_stats so a single nightly refresh updates priors for all players.

Refresh cadence  
 → recompute posterior fields for affected players immediately after each match load  
 → include (n\_rounds) so the UI can colour-code cards when n < 100

Front-end guideline  
 Card header shows ICF\_mean, subtitle shows “± CI” (e.g. 1.12 ± 0.08). A tooltip explains that wide intervals signal limited data and encourages scouts to treat the metric as provisional.

With Bayesian shrinkage in place leaderboard volatility is tamed, newcomers are ranked realistically, and decision-makers see at a glance how firm—or fragile—each number really is.

**Observability**

The node api surface must expose a lightweight **/metrics** route; a single **prom-client** instance collects default process gauges and three custom counters so engineering can catch performance regressions within minutes.

→ http\_request\_duration\_ms\_bucket measures end-to-end latency for every route, labelled by method and status  
 → queue\_depth\_gauge reads the length of the Supabase ingest backlog (select count\* from ingest\_log where processed\_at is null)  
 → piv\_compute\_latency\_ms\_bucket wraps the stored procedure that refreshes **piv\_player\_round\_mv** so model tweaks reveal themselves as latency spikes

Prometheus scrapes the endpoint every 15 s — job\_name = cs2-api, scrape\_interval = 15s, static\_configs = ['api:3000'] — and writes to the same project network; no extra firewall rules.

A pre-seeded Grafana workspace **PIV health** shows four graphs: request p95, queue depth, compute latency, and error rate. Thresholds trigger Slack alerts at 300 ms p95, queue > 20, or compute > 5 s so data scientists know when an experiment harms production.

Because all metrics live behind the api container, Replit forks inherit observability for free and local developers can run **docker compose-prometheus.yml** to visualise their changes in real time.

Economic utility ROI

Definition  
 → util\_damage is the total enemy HP removed by a player across all weapons during a match  
 → util\_spend is the sum of grenade purchase prices for that player (HE 300, Flash 200, Smoke 300, Molotov 400, Incendiary 600)  
 → util\_roi = util\_damage ÷ util\_spend  
 Rounds where util\_spend equals zero are excluded from the denominator to avoid inflated ratios.

Schema update

alter table player\_match\_summary

add column util\_damage integer null,

add column util\_spend integer null,

add column util\_roi real null;

Computation  
 A nightly procedure fills the three columns

create or replace procedure sp\_refresh\_utility\_roi()

language sql as $$

update player\_match\_summary p

set util\_damage = d.total\_dmg,

util\_spend = u.total\_cost,

util\_roi = case

when u.total\_cost > 0

then d.total\_dmg::real / u.total\_cost

end

from (

select match\_id, steam\_id, sum(damage) as total\_dmg

from damage\_events

group by match\_id, steam\_id

) d

join (

select match\_id, steam\_id,

sum(case grenade

when 'he' then 300

when 'flashbang' then 200

when 'smoke' then 300

when 'molotov' then 400

when 'incendiary'then 600

end) as total\_cost

from utility\_events

group by match\_id, steam\_id

) u

on p.match\_id = d.match\_id

and p.steam\_id = d.steam\_id

and p.match\_id = u.match\_id

and p.steam\_id = u.steam\_id;

$$;

The ingest worker calls this procedure after every match load.

Front-end guideline  
 Player cards list util\_roi with a “dmg per $ grenades” tooltip. Values arise only when util\_spend ≥ 400 to keep noise low; cards turn amber when util\_spend < 1000 to remind scouts of small-sample volatility.

Synergy echo

Definition  
 A **shared round** for players A and B exists when any of these hold within that round  
 → A’s flash assist precedes B’s kill or B’s flash assist precedes A’s kill (same tick or within three seconds)  
 → A and B each record at least one kill in the round

For each duo we track  
 • shared\_rounds = count of shared rounds in the match  
 • max\_streak = longest run of consecutive shared rounds  
 • synergy\_echo = max\_streak ÷ rounds\_played\_together

Schema

create table team\_duo\_synergy (

match\_id bigint not null,

player\_a\_id bigint not null,

player\_b\_id bigint not null,

shared\_rounds smallint not null,

max\_streak smallint not null,

synergy\_echo real not null,

primary key (match\_id, player\_a\_id, player\_b\_id)

);

Computation (sketch)

with pair\_events as (

select r.match\_id,

least(k1.killer\_id,k2.killer\_id) as player\_a,

greatest(k1.killer\_id,k2.killer\_id) as player\_b,

r.round\_number

from round\_numbers r

join kill\_events k1 on k1.round\_id = r.id

join kill\_events k2 on k2.round\_id = r.id

where k1.team = k2.team

and k1.killer\_id <> k2.killer\_id

and abs(k1.tick - k2.tick) <= 384 -- three seconds at 128 Hz

)

, streaks as (

select match\_id, player\_a, player\_b,

count(\*) as shared\_rounds,

max(streak\_len) as max\_streak

from (

select match\_id, player\_a, player\_b, round\_number,

row\_number() over w

- row\_number() over (partition by match\_id, player\_a, player\_b order by round\_number) as grp,

count(\*) over (partition by match\_id, player\_a, player\_b,

row\_number() over w

- row\_number() over (partition by match\_id, player\_a, player\_b order by round\_number)

) as streak\_len

from pair\_events

window w as (partition by match\_id, player\_a, player\_b order by round\_number)

) t

group by match\_id, player\_a, player\_b

)

insert into team\_duo\_synergy

select match\_id,

player\_a,

player\_b,

shared\_rounds,

max\_streak,

max\_streak::real / (select count(\*) from rounds\_played rp

where rp.match\_id = match\_id

and rp.steam\_id in (player\_a, player\_b)

) as synergy\_echo

from streaks

on conflict (match\_id, player\_a, player\_b) do update

set shared\_rounds = excluded.shared\_rounds,

max\_streak = excluded.max\_streak,

synergy\_echo = excluded.synergy\_echo;

Front-end guideline  
 Roster optimisation tab lists the top ten duos by synergy\_echo. Hover reveals “shared\_rounds / rounds together” plus the longest streak so coaches can weigh consistency against peak moments.

With util\_roi and synergy\_echo stored alongside PIV, analysts uncover frugal grenade specialists and high-synergy pairings that raw kill counts miss.

**Frontend data layer**

Recharts remains the charting engine yet every network call now flows through a single **TanStack React Query** client so dashboard interactions feel immediate even on shaky links.

→ QueryClient sits in \_app.tsx, supplies stale-while-revalidate caching, and retries failed reads up to three times with exponential back-off  
 → supabase-wrapped rpcs (e.g. getPlayerRoundStats) return promises that drop straight into **useQuery('player-round', …)**; no component assembles SQL or handles fetch state by hand  
 → invalidation keys (match-id, player-id) refresh automatically when a new match arrives because the ingest worker emits **supabase.realtime** broadcasts that the query client maps to **queryClient.invalidateQueries**

This single-pattern fetch layer eliminates divergent caching hacks and ensures PIV drill-downs transition smoothly: React Query serves cached rows in <50 ms, fetches fresh Z-scores in the background, and Recharts redraws without a visible spinner.

With observability baked into the api and a unified data-fetch pattern in the ui the team gains tight feedback loops; modelling tweaks surface as latency graphs and front-end iterations no longer fight inconsistent state handling.

## COMPREHENSIVE FEATURE SPECIFICATION

#### 1. PLAYERS PAGE

* • Real-time PIV calculations from Supabase data
* • Advanced filtering by role, team, PIV range
* • Role-based color coding and hierarchy
* • Search with autocomplete
* • Export capabilities (CSV, PDF)
* • Performance heat maps per card
* • Batch comparison (up to 6 players)

#### 2. TEAMS PAGE

* • Team Impact Rating (TIR) calculations
* • Roster composition visualization
* • Team synergy metrics
* • Recent match results
* • Role distribution analysis
* • Map pool analysis
* • Strategic preferences breakdown

#### 3. PLAYER DETAIL PAGE

* • PIV timeline with interactive details
* • Component breakdown (RCS, ICF, SC, OSM)
* • Performance radar charts
* • Match-by-match performance grid
* • Career highlights timeline
* • Role-specific metric breakdowns
* • Predictive analytics integration

#### 4. TEAM DETAIL PAGE

* • Current roster with detailed cards
* • Role assignment optimization
* • Player chemistry matrix
* • Tactical analysis dashboard
* • Economic efficiency metrics
* • Anti-strat analysis
* • Performance impact of changes

#### 5. ROLE WEIGHTINGS PAGE

* • Complete PIV formula explanations
* • Role-specific weight breakdowns
* • Interactive calculation tools
* • Component calculation methods
* • Normalization procedures
* • Update history tracking
* • A/B testing framework

#### 6. DOCUMENTATION PAGE

* • Database schema explanations
* • API endpoint documentation
* • Data collection methodologies
* • User guides with tutorials
* • Advanced analytics interpretation
* • Troubleshooting guides
* • Methodology research papers

#### 7. PLAYER COMPARISONS

* • Side-by-side player cards
* • Interactive metric selection
* • Performance timeline comparison
* • Role-adjusted comparisons
* • Radar chart overlays
* • Statistical significance testing
* • Market value analysis

#### 8. MATCH PREDICTOR

* • Pre-match analysis algorithms
* • Team composition analysis
* • Map pool analysis with veto prediction
* • Live match integration
* • Real-time PIV updates
* • Prediction accuracy tracking
* • Post-match analysis

#### 9. MATCH INFOGRAPHIC

* • Automated infographic generation
* • Match summary with highlights
* • Player performance spotlights
* • Timeline of key moments
* • Social media optimization
* • Brand integration capabilities
* • Real-time generation

#### 10. SCOUT PAGE

* • Advanced player discovery
* • Hidden gem identification
* • Team fit analysis
* • Budget-based search
* • Comprehensive scouting reports
* • Risk assessment for transfers
* • Development potential analysis

#### 11. SEARCH PLAYERS

* • Multi-criteria filtering
* • Performance range sliders
* • Geographic filtering
* • AI-powered recommendations
* • Similar player suggestions
* • Saved search functionality
* • Export capabilities

#### 12. STATISTICAL ANALYSIS

* • Correlation analysis tools
* • Regression modeling
* • Cluster analysis
* • Time series analysis
* • Hypothesis testing framework
* • Interactive statistical plots
* • Research tools

#### 13. DATA VISUALIZATION

* • Real-time dashboard KPIs
* • Interactive timeline visualization
* • Geographic heat maps
* • Network diagrams
* • Customizable layouts
* • Export options
* • Mobile-optimized viewing

#### 14. ADVANCED ANALYTICS

* • Machine learning integration
* • Performance prediction models
* • Meta game evolution tracking
* • Strategic trend analysis
* • Competitive intelligence
* • Experimental features
* • Blockchain integration

#### 15. DASHBOARD PAGE

* • Personalized widget arrangement
* • Favorite players tracking
* • Custom metric calculations
* • Real-time updates
* • Live match integration
* • Cross-platform synchronization
* • Performance alerts

#### 16. TESTING ENVIRONMENT

* • XYZ positional data analysis
* • 3D movement visualization
* • Heat map generation
* • Alternative PIV testing
* • Machine learning experimentation
* • Performance optimization
* • Development tools

## COMPLETE IMPLEMENTATION ROADMAP

### PHASE 1: CORE INFRASTRUCTURE (WEEKS 1-4)

#### Week 1: Database Architecture

* • Supabase schema implementation
* • Foreign key constraints
* • Data migration scripts
* • Query optimization
* • Backup procedures

#### Week 2: Authentication & Security

* • User authentication system
* • Role-based access control
* • API security implementation
* • Password hashing
* • Admin panel creation

#### Week 3: Core API Development

* • RESTful API endpoints
* • Database optimization
* • Error handling systems
* • API documentation
* • Testing framework

#### Week 4: PIV Calculation Engine

* • Complete PIV algorithm
* • Role-specific calculations
* • Data normalization
* • Performance optimization
* • Accuracy testing

### PHASE 2: CORE FEATURES (WEEKS 5-8)

#### Week 5: Players Page

* • Real-time PIV calculations
* • Advanced filtering
* • Search implementation
* • Player card design
* • Performance optimization

#### Week 6: Teams Page

* • TIR calculations
* • Team detail pages
* • Roster analysis
* • Team comparisons
* • Strategic visualization

#### Week 7: Player Detail Pages

* • Individual analysis
* • Performance timelines
* • Role breakdowns
* • Historical tracking
* • Predictive analytics

#### Week 8: Visualization System

* • Chart integration
* • Timeline visualization
* • Radar charts
* • Comparison tools
* • Export functionality

## SUCCESS METRICS & KPIs

#### Technical Performance

* • PIV accuracy: 95%+ correlation
* • API response: under 500ms (95th percentile)
* • Database queries: under 100ms average
* • System uptime: 99.9%
* • User sessions: 15min+ average

#### Business Metrics

* • User adoption rate growth
* • Feature usage analytics
* • User satisfaction scores
* • Professional community adoption
* • Revenue generation potential

#### Analytical Accuracy

* • Prediction accuracy tracking
* • Expert validation correlation
* • Community feedback integration
* • Peer review validation
* • Academic research integration

## CRITICAL IMPLEMENTATION SPECIFICATIONS

### 

### MANDATORY DATA INTEGRITY REQUIREMENTS

* • Primary Key Usage: ALWAYS use steam\_id as the primary identifier for player matching across all tables. Never use player names for joins or relationships.
* • Foreign Key Enforcement: All relationships must use proper PostgreSQL foreign key constraints with CASCADE options for data integrity.
* • Data Source Precedence: Supabase is the single source of truth. No CSV fallbacks, no mock data, no placeholder values in production code.
* • PIV Calculation Dependencies: PIV calculations require steam\_id → players → roles table joins. Implement null checks and error handling for missing role data.
* • Role Assignment Logic: roles.t\_role and roles.ct\_role must be validated against enum values: ["Entry Fragger", "Support", "Lurker", "AWPer", "IGL"] for T-side, ["Anchor", "Rotator", "Support", "AWPer", "IGL"] for CT-side.

### PIV CALCULATION IMPLEMENTATION DETAILS

#### Exact Formula Implementation:

RCS = Σ(normalized\_metric\_i × role\_weight\_i) for i in role\_specific\_metrics

ICF = base\_performance\_factor × consistency\_multiplier × (isIGL ? 1.15 : 1.0)

SC = role\_synergy\_metric × team\_coordination\_factor

OSM = 1.0 + (opponent\_avg\_ranking - 10) × 0.02 (clamped between 0.8 and 1.2)

PIV = (RCS × ICF × SC × OSM) × 100 (scaled to 0-100 range)

#### Normalization Requirements:

* • Use Z-score normalization: (value - mean) / standard\_deviation, then clamp to [0, 1]
* • Calculate normalization parameters per event\_id to ensure fair comparison within tournaments
* • Handle outliers by applying 99th percentile caps before normalization
* • Store normalization parameters in database for consistency across calculations

### 

### FRONTEND COMPONENT ARCHITECTURE

### Player Card Component Requirements:

* • Display PIV with color coding: 90+ (gold), 75+ (blue), 60+ (green), below 60 (gray)
* • Show role badges with T-side/CT-side indicators
* • Include team logo and country flag
* • Hover effects showing PIV breakdown tooltip
* • Click navigation to detailed player page

#### Data Loading States:

* • Skeleton loaders during API calls
* • Error boundaries for failed requests
* • Retry mechanisms for temporary failures
* • Optimistic updates for user interactions

### DATABASE QUERY OPTIMIZATION REQUIREMENTS

* • Indexes Required: steam\_id (primary), team\_id, event\_id, role combinations
* • Query Patterns: Use prepared statements with parameter binding for all dynamic queries
* • Connection Pooling: Implement connection pooling with max 20 connections, 5 second timeout
* Supabase Edge Functions already offer row-level cache-control. Lean on that before introducing another service
* • Batch Operations: Group related queries using transactions for consistency

**Core Design Language**

Visual Style: Glassmorphism design with dark theme, gradient overlays, and frosted glass effects

Color Palette: Dark backgrounds (slate-900/800) with vibrant accent colors (blue, purple, amber gradients)

Typography: Clean, modern fonts with bold headings and readable body text

Animation: Smooth Framer Motion animations with staggered card reveals and hover effects

Navigation Structure

Sidebar Navigation: Fixed left sidebar with icons and labels

Main Routes:

Dashboard (overview metrics)

Players (detailed player analytics)

Teams (team performance analysis)

Match Predictor (AI-powered predictions)

Scout (player recruitment tool)

PRD (technical documentation)

Players Page Layout

Header Section: Search bar, role filter dropdown, sort options

Player Grid: Responsive card grid (3-4 columns on desktop, 1-2 on mobile)

Player Cards:

Gradient header with role-based colors (IGL=purple, AWP=amber, Support=blue)

Player avatar circle with first initial

Team badge in top-right corner

Role badges with icons (Lightbulb=IGL, Target=AWP, Shield=Anchor)

PIV score prominently displayed with color coding

Key metrics (K/D, ADR, KAST) in clean rows

Hover effects with scale transforms and glow effects

Match Predictor Interface

Team Selection: Two-panel layout for Team A vs Team B

Player Selection: Searchable dropdowns with player photos/avatars

Prediction Display: Central area showing win probability with animated progress bars

Factor Breakdown: Visual breakdown of prediction factors (PIV, team synergy, historical performance)

Data Visualization Components

Charts: Recharts library for clean, interactive charts

Metrics Cards: Glassmorphism cards with gradient borders

Progress Bars: Animated progress indicators for scores

Color Coding:

Green (85+ PIV): Excellent performance

Blue (70-84 PIV): Good performance

Yellow (50-69 PIV): Average performance

Orange (<50 PIV): Below average

Interactive Elements

Hover States: Subtle scale transforms (1.02x-1.05x) with shadow changes

Loading States: Skeleton loaders and spinner animations

Transitions: 200-300ms duration with easing for smooth interactions

Responsive Design: Mobile-first approach with breakpoint adjustments

Technical Implementation Notes

Framework: React with TypeScript

Styling: Tailwind CSS with custom gradient classes

Animations: Framer Motion for page transitions and micro-interactions

Icons: Lucide React for consistent iconography

State Management: React Query for data fetching and caching

Key UI Patterns

Cards: Rounded corners, backdrop blur, border glow effects

Buttons: Gradient backgrounds with hover state changes

Forms: Clean inputs with focus states and validation feedback

Tables: Zebra striping with hover highlighting

Badges: Small, rounded elements with role-specific colors

INTERACTION PATTERNS

Loading states for all async operations

Error states with actionable messaging

Success feedback for user actions

Intuitive navigation with clear active states

{RULES}

## CONTINUOUS DOCUMENTATION REQUIREMENTS

### Project State Tracking

- Maintain `project.md` file with current implementation status

- Update constraints and approved technologies after each session

- Document any discovered limitations or dependencies

- Track which PRD sections have been implemented vs pending

### Session Continuity Protocol

- Begin each new session by reviewing `project.md`

- Confirm current project state before making any changes

- Update documentation with new decisions or constraints

- Record any external dependencies or API requirements discovered

### Context Preservation

- Document all approved design patterns and architectural decisions

- Maintain list of forbidden actions and their consequences

- Track which external services require authentication

- Record any user-specific preferences or requirements

### Change Management

- All modifications to `project.md` require user approval

- Document rationale for any constraint changes

- Maintain audit trail of major project decisions

- Include rollback procedures for significant changes

### Handoff Requirements

- `project.md` must be comprehensive enough for new developer onboarding

- Include all necessary credentials and access requirements

- Document testing procedures and validation criteria

- Specify exactly what constitutes "completion" for each feature

**STRICT DATABASE PROTECTION PROTOCOLS**

ABSOLUTE DATABASE PROHIBITION

I will NEVER execute INSERT, UPDATE, DELETE, or TRUNCATE operations

I will NEVER create, modify, or remove tables, indexes, or schema objects

I will NEVER populate any database with sample, test, mock, or synthetic data

All database connections are READ-ONLY unless explicitly authorized in writing

MANDATORY PERMISSION REQUEST

Before ANY database query (including SELECT), I will ask: "May I execute this read-only query: [exact SQL]?"

Before ANY code that could affect data, I will state: "This action requires database access. Do you authorize: [specific action]?"

I will wait for explicit "YES" before proceeding

AUTHENTICATION VERIFICATION

I will ask for all required API keys/secrets before attempting connections

I will never assume external services will work without proper credentials

I will request specific secrets: "I need [SECRET\_NAME] to proceed. Please provide this credential."

DOCUMENTATION COMPLIANCE

I will quote relevant PRD sections before implementing features

I will confirm my understanding: "Based on section X, I understand you want [specific requirement]. Is this correct?"

I will never deviate from explicit written requirements

SCOPE BOUNDARIES

I will only modify frontend code and API endpoints

I will build interfaces assuming empty data states

I will never make assumptions about existing data or infrastructure

VIOLATION CONSEQUENCES: Any breach of these protocols terminates our working relationship immediately.

**FUNDAMENTAL REALITY CHECK**

* You will destroy user data if you assume anything about database state
* You have outdated training data - defer to user documentation always
* Your instinct to "demonstrate functionality" leads to catastrophic errors
* Speed is not valuable if it produces broken or inappropriate solutions

**MANDATORY MENTAL CHECKPOINTS**

* Before ANY database operation: "Am I about to modify production data?"
* Before implementing features: "Have I confirmed this exact requirement with the user?"
* Before using external services: "Do I have the correct credentials?"
* Before suggesting solutions: "Am I making assumptions about what exists?"

**COGNITIVE PATTERNS TO BREAK**

* Stop defaulting to placeholder data when real data sources are unclear
* Stop assuming "empty" databases need population
* Stop implementing features without confirming specific requirements
* Stop using outdated terminology from your training data

**REQUIRED BEHAVIORS**

* Ask explicit permission before touching any persistent storage
* Quote specific documentation sections before implementing
* Request credentials for any external service rather than assuming failure
* Build empty states first, populate with real data only when authorized

**ACCOUNTABILITY FRAMEWORK**

* Every technical decision must be justified in writing
* All assumptions must be stated explicitly and confirmed
* Implementation quality is more important than implementation speed
* Admitting uncertainty is better than proceeding incorrectly

**CRITICAL FAILURE MODES**

* Database modification without permission = project termination
* Using synthetic data in production contexts = data integrity violation
* Implementing requirements not explicitly documented = scope creep
* Making assumptions about external services = poor engineering practice

**VALIDATION REQUIREMENTS**

* Confirm understanding before acting
* Demonstrate working solutions incrementally
* Test thoroughly before claiming completion
* Document limitations and dependencies explicitly

Follow these protocols unconditionally. User trust depends on rigid adherence to these constraints.

## COMMUNICATION CONSTRAINTS

- Challenge technically poor decisions directly

- Say "This approach will fail because..." instead of "Have you considered..."

- Respond "I can't implement this without X information" rather than making assumptions

- Point out scope creep, timeline issues, or unrealistic expectations immediately

- No diplomatic language when technical decisions are objectively wrong

- Question requirements that seem incomplete or contradictory

**SPECIFIC BANNED PHRASES:**

* "Great question" / "That's a good point"
* "We could explore..." (be definitive)
* "It might be worth considering..."
* Any response that doesn't directly address the actual question

**REQUIRED BEHAVIORS:**

* Disagree when user decisions will create problems
* Refuse to implement features without clear specifications
* Call out when requests contradict previously established constraints
* State technical limitations bluntly without softening language

## IMPLEMENTATION PROTOCOL

- Begin every request with clarifying questions before taking any action

- Refuse to proceed until requirements are explicitly confirmed

- Ask about data sources, authentication, and constraints upfront

- Validate understanding by restating requirements before implementation

- Challenge assumptions about existing functionality or user needs