Upset Security & Data Integrity Framework

What Upset Does

Upset is a sports analytics platform that processes data from third-party APIs and delivers actionable insights to users. The entire value proposition depends on data accuracy, which means security architecture has to balance three competing priorities: preventing unauthorized access, maintaining 98-99% data accuracy, and protecting user information. If the data isn't accurate, the platform is useless and if users can't trust it, they won't pay for it.

Authentication Design

I chose OAuth 2.0 through Google for the same reasons I avoid building custom auth anywhere: it eliminates password management vulnerabilities and delegates security to a provider with dedicated teams focused on the problem. This approach provides simple onboarding while reducing the attack surface by not storing credentials. Session management uses secure HTTP-only cookies with reasonable expiration windows that balance security against constant re-authentication.

Key controls I implemented:

- Token refresh mechanism prevents session hijacking from long-lived tokens
- Subscription tier enforcement at API layer blocks privilege escalation
- Role-based access controls separate standard users from admin functions

The subscription enforcement happens server-side on every API request because client-side checks are suggestions, not security.

API Rate Management

The data pipeline depends on third-party sports APIs that actively block excessive requests, which creates an interesting constraint. I needed reliable data ingestion while respecting provider limits and avoiding service interruption. Rate limiting had to work at multiple layers:

- Request throttling prevents hitting provider rate limits
- Staggered ingestion schedules distribute load across time windows
- Exponential backoff with automatic retry handles temporary failures gracefully
- Primary and backup data sources ensure continuity if one provider denies access

Monitoring alerts trigger before reaching rate limits so I can adjust ingestion patterns before service gets interrupted. The incremental update strategy minimizes unnecessary requests by only fetching changed data, which reduces both costs and the risk of getting blocked.

This defense-in-depth approach means no single point of failure brings down data collection. If one API goes down, I fall back to alternate providers automatically.

Data Accuracy Controls

Data accuracy is the core deliverable, so the validation pipeline needs to catch errors before they corrupt the database. I implemented validation at multiple layers because relying on a single check means one bug compromises everything:

- Automated checks flag impossible values like negative points or statistics over 100%
- Database constraints prevent corruption at storage layer through NOT NULL, foreign keys, and unique indexes
- Source verification confirms data comes from legitimate providers
- Outlier detection catches statistical anomalies that pass individual field validation

User Data Protection

I designed the data architecture around privacy-by-design principles to drive specific technical decisions:

- Personal identifiable information (PII) stays in Google's ecosystem and never touches the app's infrastructure
- Subscription status gets stored separately from user preferences and activity
- Payment processing happens through the payment provider with no card data entering platform systems
- User-generated content is encrypted at rest with database-level access controls

Data minimization - The platform only collects what's necessary for service delivery.

Purpose limitation - Data is used only for stated purposes and is never sold to third parties.

User control - Account deletion permanently removes all associated data within 30 days through automated cleanup procedures.

Threat Model & Risk Response

The threat model considers both technical attacks and business risks.

Data poisoning attacks could inject false statistics and corrupt data stores, but source verification plus outlier detection catches those attempts.

Credential stuffing gets neutralized through OAuth delegation since there are no passwords to stuff.

Subscription bypass attempts get prevented through server-side tier validation on every API request.

Automated scraping gets deterred through rate limiting and API authentication requirements. The risk response framework includes:

- Incident response procedures for data quality issues with rollback and correction capabilities
- Real-time monitoring dashboards tracking ingestion health, API failures, and anomalies

- Automated alerting on suspicious patterns with escalation procedures
- User notification workflows for data corrections that affect analytics

Core Design Principles

A few architectural decisions shaped everything else:

- OAuth delegation eliminated credential management vulnerabilities
- Multi-layer validation catches different error types at appropriate stages
- Defense-in-depth prevents single points of failure in data ingestion
- Privacy-by-design drives technical architecture, not just compliance checkboxes
- Server-side enforcement prevents client-side bypass attempts
- Proactive monitoring enables issue detection before user impact