PROJECT SHOWCASE / ENGINEERING & PRODUCT PORTFOLIO

MetricForge

Technical Case Studies

Prepared for professional review | November 4, 2026 | English edition

Evidence note: Based on prior project discussions, described architecture, selected development/debugging history and portfolio records available at preparation time. This is a portfolio case study, not a fresh source-code audit, external security review, market study or verified commercial due diligence. Planned capabilities and unverified outcomes are labeled accordingly.

MetricForge | Technical Case Study

Social analytics SaaS | Stage: Working application with YouTube integration; broader platform scope evolving

Technical summary

MetricForge is a multi-user social analytics platform intended to centralize account data, media performance, and analytical insights across social channels. Its confirmed development history includes authentication, a protected analytics dashboard, GraphQL domain models, MongoDB persistence, and YouTube account authorization and synchronization.

Problem and engineering context

Creators and small organizations must move between platform dashboards to correlate audience, content, engagement, and performance. Context becomes fragmented, comparison is expensive, and historical snapshots are difficult to standardize.

Technology stack

- Next.js 16 + React and TypeScript for application UI and server endpoints.
- GraphQL Yoga and Apollo Client for typed query/mutation workflows.
- MongoDB Atlas / Mongoose for users, connected accounts, content records, metric snapshots and analytics-related entities.
- YouTube OAuth and YouTube API synchronization; credentials-based access with password hashing and cookie/session mechanisms discussed.
- Docker deployment and Caddy-based reverse proxy on a VPS; production configurations require secret and callback management.

System architecture and component boundaries

- Web interface: authenticated dashboards and analytical views backed by GraphQL queries.

- Application boundary: authentication and integration routes, GraphQL resolvers, synchronization jobs and domain services.
- Persistence: collections for users, social connections/accounts, content items, historical metrics, comments and insight-related entities.
- Integration pipeline: obtain permitted channel access, fetch source records, normalize payloads, persist snapshots, then aggregate KPIs.
- Deployment: containerized application with reverse proxy and remote MongoDB Atlas; network allowlists, OAuth redirects and environment secrets are operational dependencies.

Data model and domain contracts

- User and authorization context: isolate connected social accounts per user and enforce ownership in each resolver.
- SocialConnection/SocialAccount: external provider identifiers, token metadata, owner and synchronization state.
- ContentItem and metric snapshots: timestamped platform metrics, content metadata and aggregation inputs.
- Comments/insights: text-derived analytical fields where actually available; distinguish computed analysis from native platform information.
- Analytics: engagement rate and an application-level metricForgeScore discussed in development; formula, sampling and validation must be documented before external performance claims.

Primary application flows

- User registration and login; access protected pages and a personalized dashboard.
- Connect a YouTube channel via OAuth; persist authorization safely and handle expired/denied permissions.
- Run synchronization for channel/content/comments/metrics, then display selected dashboard KPIs.
- Read and filter data through the GraphQL domain; protect every operation with authorization and appropriate input validation.

Engineering challenges and tradeoffs

- OAuth callback correctness across local and production environments; redirect URI consistency.
- MongoDB Atlas network access and secret management during VPS deployment.
- Potential duplicated or stale metric snapshots when sync jobs retry; idempotency and upsert keys are important.
- Provider quotas, pagination, rate limits and partial synchronization require observability and backoff.
- Cross-provider normalization can hide incomparable definitions; metrics must retain provenance and provider-specific meaning.

Available evidence and limits

- Previously discussed production domain and Docker deployment, build routes and YouTube-specific sync flows.

- Prior code/context describes GraphQL entities, account ownership, KPIs and metric snapshots.
- YouTube is the directly supported integration in available context; Facebook and Instagram were modeled but are not verified as fully synchronized.
- No audited performance benchmarks, paid customer figures, or independent security assessment supplied.

Recommended technical validation / next steps

- Add integration-level tests for OAuth expiration, permission revocation, pagination and sync failures.
- Instrument job metrics, lag, rate-limit counters and data freshness labels.
- Formalize authorization tests for tenant isolation and secrets-at-rest policy.
- Validate cross-provider metric definitions before claiming comparable dashboards.
- Only present FREE/PRO/BUSINESS as commercial tiers if payment, entitlements and operations are actually implemented.

Recruiter interview prompts

- Explain an end-to-end YouTube synchronization and where idempotency is enforced.
- How do you authorize GraphQL resolvers to prevent user-account data leakage?
- How would you handle provider quotas and stale metrics?

Disclosure: This report intentionally distinguishes observed discussions from proposed engineering improvements. Tests, benchmarks, revenue, user adoption and security compliance are not asserted without supporting evidence.