GameArena Leaderboard Documentation

Overview

The GameArena Leaderboard System is a scalable web application designed to record player scores, compute rankings, and display leaderboard results with low latency.
The system supports three core operations: submitting a score, retrieving the top-ranked players, and checking an individual player's rank. These requirements match the assignment objective of building optimized leaderboard APIs capable of handling millions of records while maintaining consistency and performance.

The implementation focuses on production-level concerns such as database efficiency, caching strategy, concurrency handling, real-time updates, and monitoring.

Architecture Summary

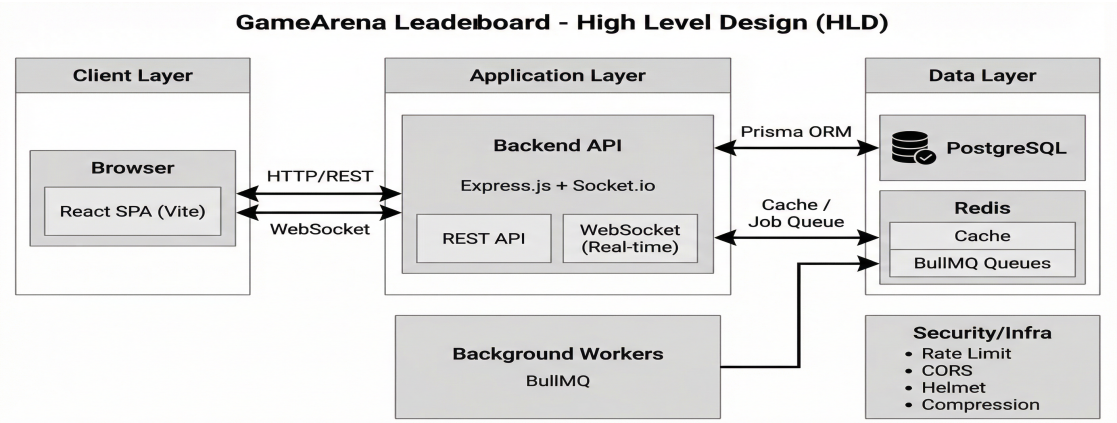
The system follows a layered architecture consisting of a client layer, application layer, and data layer.

The **client layer** is a React single-page application built with Vite. It communicates with the backend through REST APIs for deterministic operations and WebSocket connections for real-time leaderboard updates.

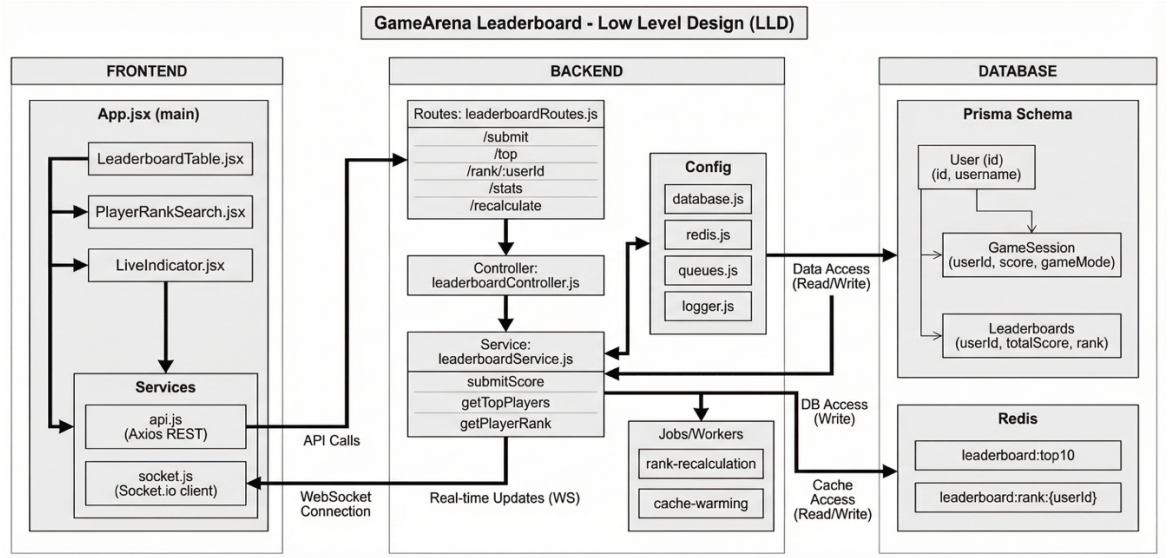
The **application layer** is implemented using Node.js and Express, with Socket.io enabling bidirectional communication. This layer validates requests, executes business logic, manages transactions, interacts with Redis cache, and triggers background jobs for expensive computations such as rank recalculation.

The **data layer** combines PostgreSQL and Redis. PostgreSQL provides reliable relational storage for users, game sessions, and aggregated leaderboard data. Redis is used for in-memory caching and as the message broker for BullMQ background workers. This separation ensures both strong consistency and fast read performance.

High Level Design



Low Level Design



Data Processing Flow

When a score is submitted, the backend executes a database transaction that inserts the game session, recomputes the player's total score, and updates the leaderboard entry.
Using a transaction guarantees atomicity and prevents race conditions during concurrent submissions.

Read-heavy endpoints such as leaderboard retrieval and rank lookup are served primarily from Redis cache. If cached data is unavailable, the system queries PostgreSQL and stores the result in Redis with a short expiration time. This cache-aside strategy significantly reduces database load and response latency.

Performance Optimization

Initial testing showed high latency due to full table scans, lack of indexing, and absence of caching.
Performance improvements were achieved through several coordinated optimizations:

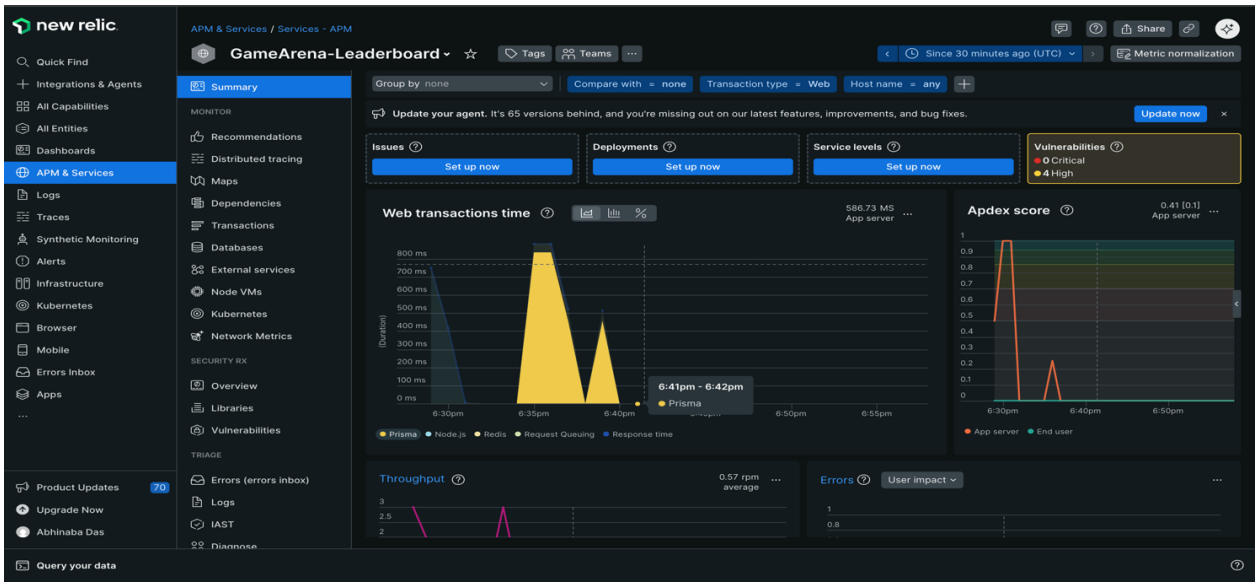
- Strategic database indexing to eliminate sequential scans
- Use of SQL window functions for efficient ranking queries
- Redis caching to serve frequent reads in a few milliseconds
- Background job processing to avoid blocking API responses
- HTTP compression and rate limiting for network efficiency and stability

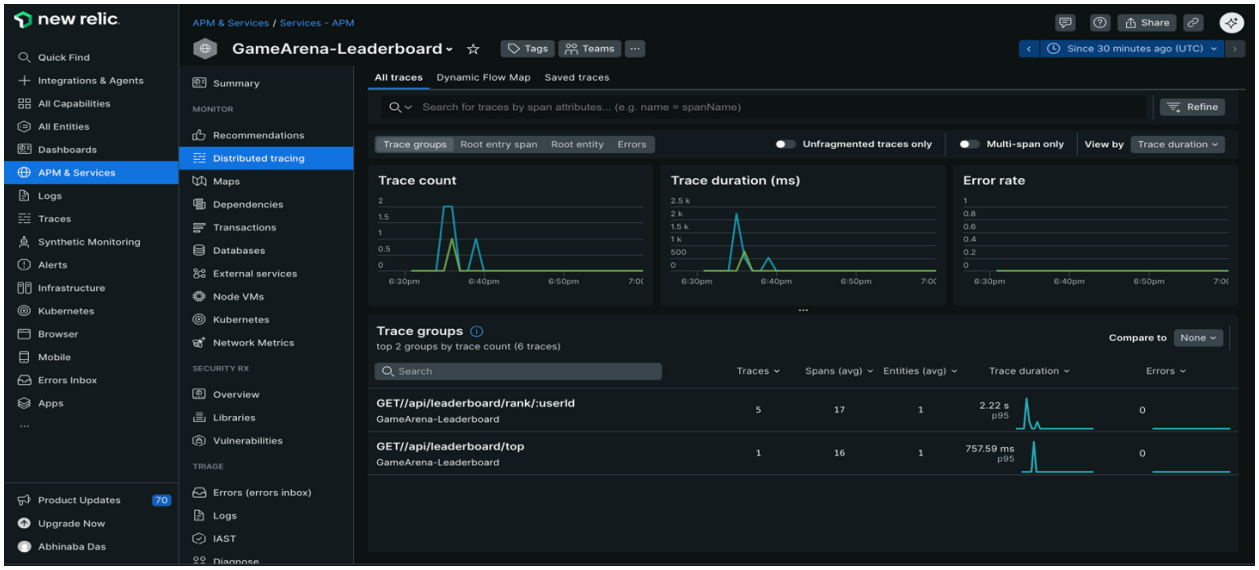
After optimization, average API latency dropped to a few tens of milliseconds, cache hit rate exceeded ninety percent, and throughput increased substantially compared to the baseline implementation.

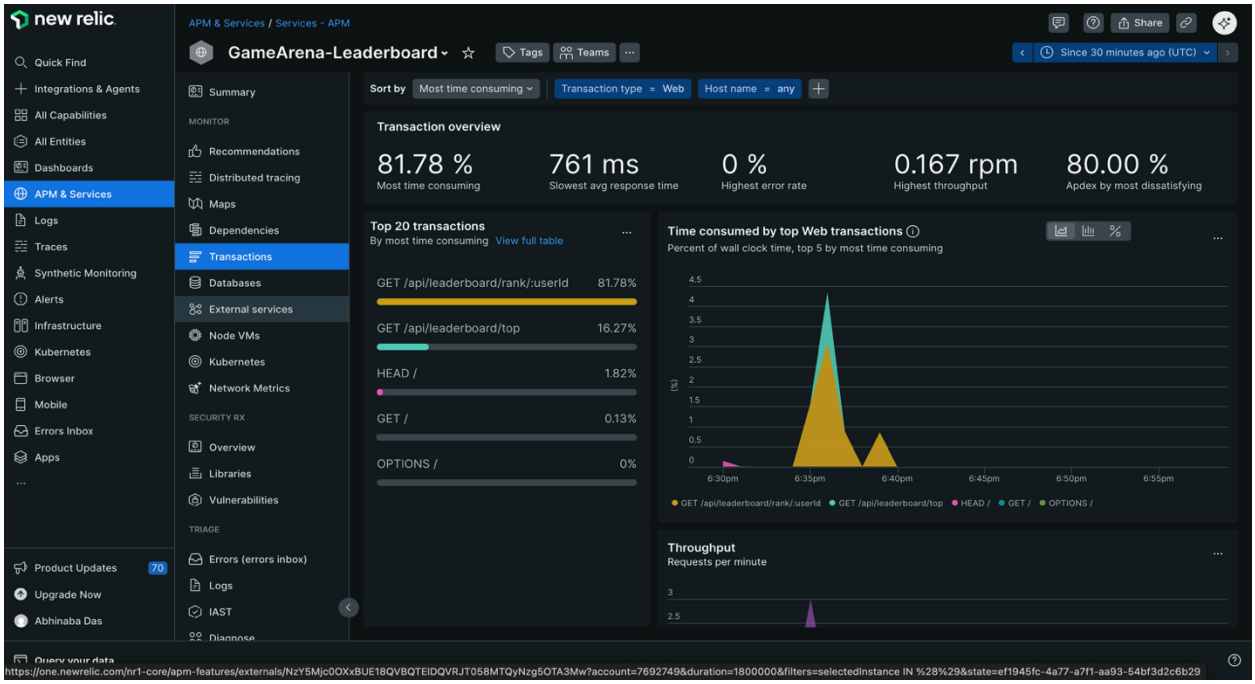
Monitoring and Observability

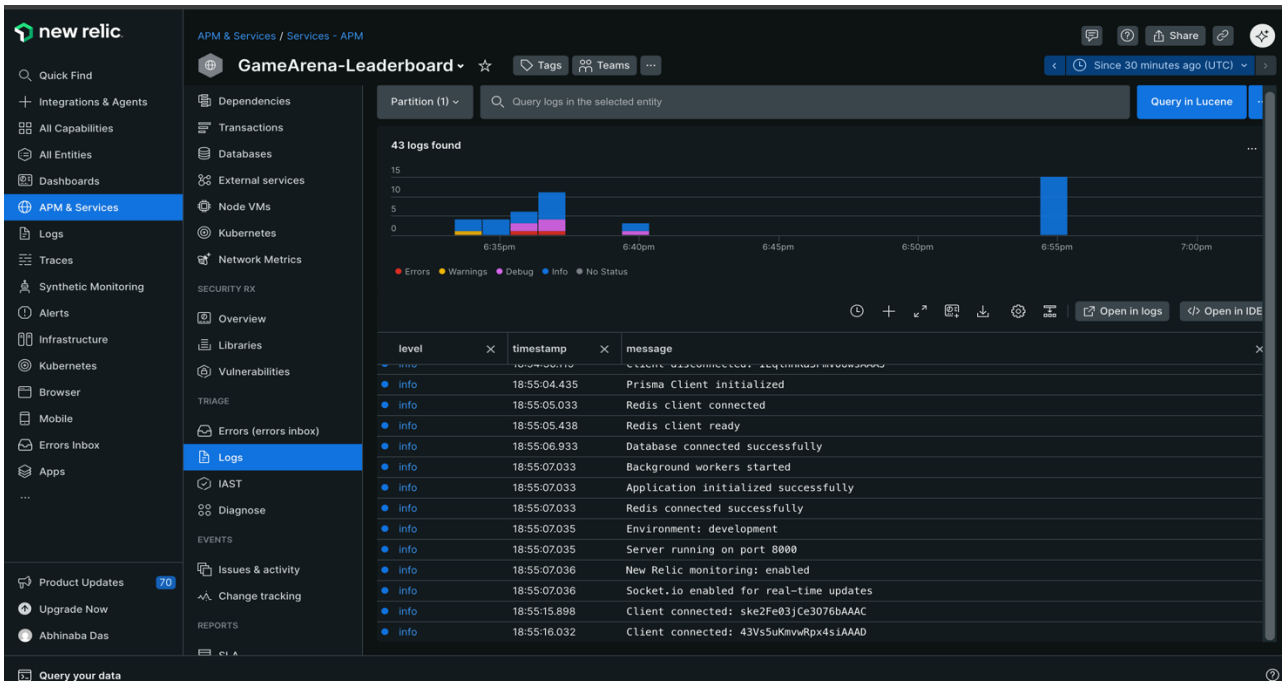
New Relic APM was integrated to track real-time performance metrics such as response latency, database query duration, error rate, and throughput.
Monitoring confirmed that optimized endpoints consistently remained within acceptable latency ranges and that database resource usage decreased significantly due to caching.

Such observability is essential for identifying bottlenecks, validating optimizations, and ensuring production reliability.





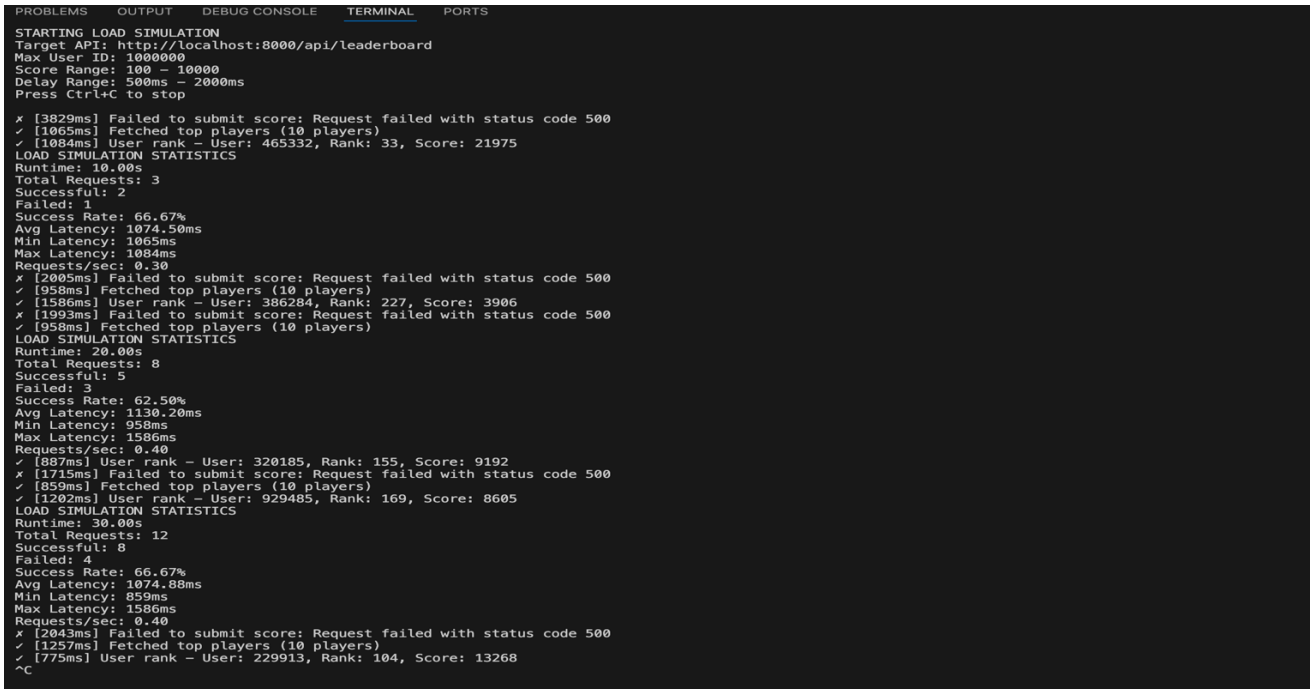




Load Simulation Findings

A load simulation script was used to emulate continuous user activity.
Leaderboard retrieval and rank lookup remained stable, indicating effective indexing and caching.
However, intermittent server errors occurred during score submission, suggesting possible issues with transaction handling, Redis connectivity, or invalid user identifiers.

These findings highlight the importance of validating infrastructure dependencies and concurrency logic in write-heavy scenarios.



Deployment

The frontend is deployed on Vercel and accessible at:

https://game-arena-leaderboard.vercel.app/

The backend is designed for deployment on Render connected to managed PostgreSQL, Redis, and monitoring services.
Because the API layer is stateless, the system can scale horizontally by adding additional server instances behind a load balancer.

Conclusion

The GameArena Leaderboard System demonstrates a production-oriented approach to building scalable backend services.
Through indexing, caching, transactional consistency, asynchronous processing, and monitoring, the system achieves low latency, high throughput, and reliable real-time updates while operating on large datasets.

This project reflects practical understanding of modern backend architecture and performance engineering suitable for real-world software systems.