A Self-Adaptive Canada Fitness Club Web Application Using React and Django

Caules Ge, Ethan Li

Abstract

The Canada Fitness Club Application, built with **React** and **Django**, demonstrates a self-adaptive web system for a chain of fitness clubs across Canada [1]. The application supports two user roles: administrators and members. Administrators can perform CRUD operations for studios and their affiliated amenities, while members can sign up, log in, view available studios, and schedule courses.

The system is deployed on **OpenShift** using **Docker** and is designed to be self-adaptive at three layers: server, application, and client. Runtime metrics are continuously monitored, analyzed, and used to trigger adaptive actions that maintain performance and efficiency under varying workloads.

I. Problem Description

The traffic fluctuation for fitness club services is highly volatile. High concurrency occurs when course reservations open, and promotional campaigns at the start of each season bring in large volumes of new user registrations. If the system is not self-adaptive, two negative outcomes are likely: either more resources are pre-allocated, leading to resource waste during off-peak hours, or insufficient resources during peak periods cause system slowdowns and increased error rates, resulting in a worse user experience that directly leads to revenue loss and customer churn for the fitness club. Therefore, utilizing self-adaptation to scale up the system during high traffic and maintain appropriate resources during normal times is economically essential.

As a web-based system, the Canada Fitness Club Application faces challenges under fluctuating workloads. During high user traffic, response times and error rates increase due to limited compute resources and database contention. Conversely, during low traffic periods, maintaining multiple active replicas results in inefficient resource usage. Meanwhile, clients with poor network conditions may experience long loading times due to heavy content rendering.

II. Proposed Solution and Architecture

The system adopts a self-adaptive architecture integrating an external driver program with multiple feedback loops. The design follows the **MAPE-K** (Monitor, Analyze, Plan, Execute, Knowledge) model across the three levels of adaptivity: server, application, and client.

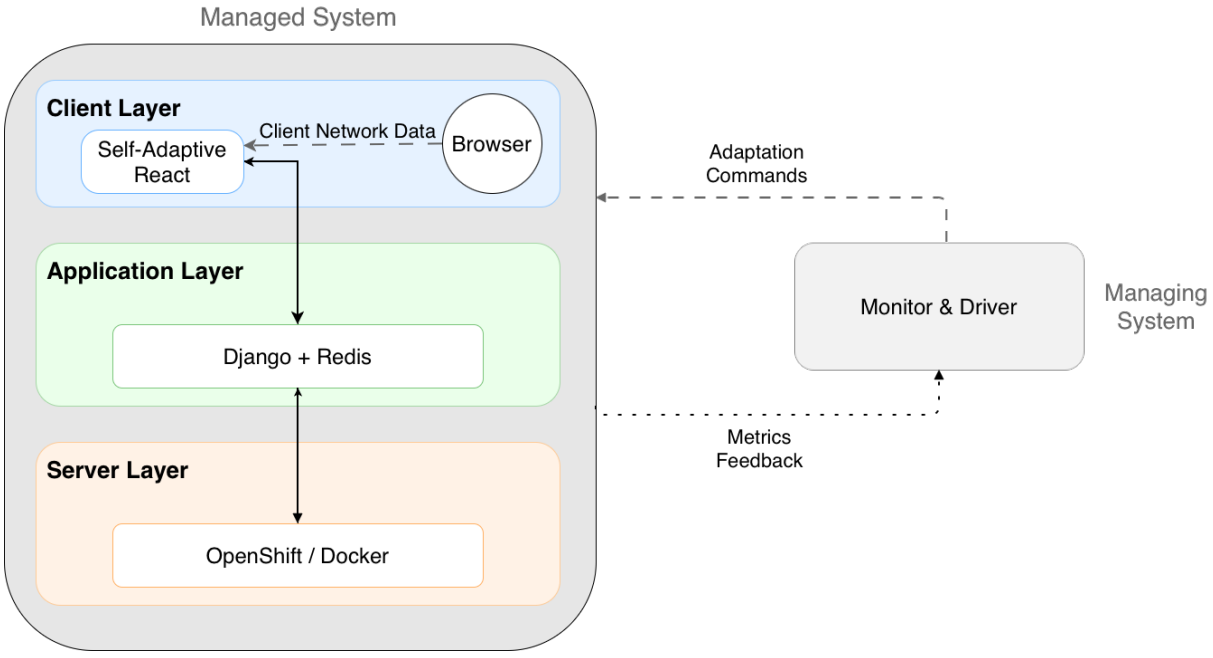

Fig. 1. System-level block diagram of the self-adaptive architecture.

A. Server-Level Adaptation

A Python driver monitors live system metrics via **Sysdig** and uses a set of predefined policies (Knowledge) to analyze the workload. The driver then plans the appropriate adaptation strategy and executes it. For example, the driver executes scaling replicas up or down based on response time and CPU/memory usage. When the replica count reaches a predefined maximum, the system switches to a degraded mode, serving simplified pages with limited functionality until the workload decreases.

B. Application-Level Adaptation

At the application layer, we use **Django-Prometheus** as a monitor, exposing key metrics such as per-view latency, database query time, cache efficiency, and error rate. These metrics allow the system to detect performance degradation. When sustained slowdowns are detected, the application:

- **Increases cache TTL** for the top most-accessed non-personalized endpoints.
- **Reduces pagination size**, decreasing query load per request.
- **Enters a degraded mode** where expensive features such as distance-based sorting, image-rich studio previews, or complex JOIN queries are temporarily disabled.

Redis is used as a distributed cache and feature flag storage, allowing adaptation decisions to be shared consistently across replicas. This enables the entire application to shift modes smoothly and simultaneously.

C. Client-Level Adaptation

To further improve usability under device or network variability, we implement a **local self-adaptive loop directly in the client**. The **React** front-end continuously monitors real user experience metrics using built-in Web APIs, including Frame rendering performance (FPS), Main thread blocking, and Network quality.

If the average FPS falls below 20 for more than 5 seconds, or the network connection is detected to be slow, the UI automatically **reduces visual and data complexity**, for example:

- Switching to lower-resolution images.
- Hiding dynamically generated maps or interactive UI modules.
- Reducing the frequency of UI updates or animation.

React state hooks are used to continuously monitor performance conditions and trigger adaptation actions. If performance recovers—such as when network conditions improve or rendering load decreases—the system automatically returns to the normal, full-detail mode. This prevents permanent degradation and preserves user experience whenever possible.

Unlike traditional adaptive systems, these adjustments occur **locally, per user**, without requiring server coordination. This ensures that users on low-powered devices or weak networks still receive a smooth and responsive experience, while high-performance clients maintain full functionality.

III. Tools Used

- **Docker/OpenShift**: Containerization and deployment.
- **Sysdig**: System-level metric collection (CPU/memory usage).
- **JMeter**: Workload generation for performance testing.
- **Django-Prometheus**: Exposes latency and cache metrics via the `/metrics` endpoint.
- **Redis**: Provides shared caching for consistent adaptive behavior.
- **React**: Collects and adapts based on browser performance metrics.

IV. Project Schedule

1) **Task 1: Deploy and Load Test** — Deploy the non-adaptive application on OpenShift and simulate workloads with JMeter.
2) **Task 2: Implement Server-Level Monitor** — Develop a Python monitor to collect and store Sysdig metrics.
3) **Task 3: Implement Server-Level Driver** — Extend the monitor for automatic scaling and degraded mode switching.
4) **Task 4: Implement Application-Level Monitor** — Integrate Django-Prometheus for latency and cache monitoring.
5) **Task 5: Implement Application-Level Driver** — Adjust cache TTL, pagination, and degraded mode via Redis.
6) **Task 6: Implement Client-Level Monitor** — Use React Web APIs to collect FPS, network, and rendering metrics.
7) **Task 7: Implement Client-Level Driver** — Link the front-end monitor to adaptive rendering logic for visual detail adjustment.

$$
\begin{aligned}
\mathrm{f} = \max\Bigg(0,\, & 0.20\left(1 - \min\left(\frac{\text{ResponseTime}}{1000}, 1\right)\right) \\
& + 0.15\left(\frac{\text{TPS}}{900}\right) \\
& + 0.10(1 - \text{CpuUsed}) \\
& + 0.35(1 - \text{ErrorRate}) \\
& + 0.20\left(1 - \min\left(\frac{P_{95}}{2000}, 1\right)\right)\Bigg)
\end{aligned}
$$

The weights (e.g., 0.35 for ErrorRate) represent the relative importance of response time, throughput, CPU usage, error rate, and P95 latency.

REFERENCES

[1] Caules Ge, *Toronto-Fitness-Club-Application (GitHub repository)*, 2025. CSC309: Introduction to Web Programming, University of Toronto. Available: https://github.com/fortii2/Toronto-Fitness-Club-Application

[2] Ladan Tahvildari, *ECE 750: Engineering Self-Adaptive Software Systems — Lecture 4 Part B*, Slide 17, Dept. of Electrical and Computer Engineering, University of Waterloo, 2025.