A Three-Layer Self-Adaptive Web Application for the Toronto Fitness Club System

Ethan Li, Caules Ge
University of Waterloo, Waterloo, ON, Canada

I. Abstract

This paper introduces a three-layer self-adaptive web application for the Toronto Fitness Club system. When facing the traffic loads that are bursty and unpredictable, the system uses a MAPE-K feedback loop to manage the resources dynamically in client, application, and server layers. We use OpenShift for horizontal scaling, Redis for caching, and React for the adaptive UI to optimize performance and availability. The experiment result shows that the adaptive system can reduce the latency and error rates by about 50% in high-load scenarios. Also, it minimizes the resource consumption during the off-peak periods compared to the non-adaptive baseline.

II. Introduction

A. Problem Context

The Toronto Fitness Club application serves members, trainers, and administrators. It supports sign-up, class booking, membership management, payments, and notifications. Users often open the app at the same time before work, during lunch, and in the evening when new classes or promotions are announced. These peaks create high and changing load in a cloud environment [1].

The system provides several key features such as studio viewing and course-related functions, which are closely tied to the user experience [1]. When any of these components slow down or fail, members face long waiting times or errors during booking. Because user traffic is bursty and unpredictable, it becomes difficult to size the system properly. If we over-provision resources, cloud costs increase. If we under-provision, latency and error rates go up, and users gradually lose trust in the platform.

Key quality goals include performance (keeping the p95 latency within the target), availability (minimizing downtime and quickly recovering by scaling up when high traffic makes the system temporarily unavailable), scalability (being elastic for both predictable and unpredictable peaks), user experience (providing fast and clear feedback), and cost awareness (avoiding wasted resources during off-peak periods [2]). We also need to preserve important business rules, such as class capacity limits and cancellation windows, even when the services scale dynamically.

A multi-layer MAPE-K approach is needed because different problems appear at different layers. The UI layer needs fast reflexes, such as simplifying the interface on low-speed or low-performance devices, to keep pages responsive [2]. The application layer needs rule-based analysis to detect rising latency or error rates and to trigger actions such as toggling a degraded mode to reduce load and save resources. The server layer requires slower but steadier decisions, such as horizontal scaling by adding or removing pods based on the score of the utility function. Each loop in the server layer shares knowledge through metrics, logs, and the previous scaling window so that actions stay coordinated and avoid unnecessary oscillation.

Success means that the adaptive system performs better than a non-adaptive baseline: lower latency during peak periods, fewer errors under high traffic, lower resource consumption, and reduced operator intervention.

B. Objectives

- **Performance and responsiveness:** Keep p95 latency for booking and payment APIs under target during predictable (class releases) and unpredictable peaks. Avoid long waits for UI [2].

- **Availability:** Reduce error rate as much as possible [2].
- **Elasticity:** Scale up at the high traffic time without exhausting system resources, and also scale back down after peaks to avoid overspending [2]. Align scaling with real demand instead of manual pre-provisioning.
- **Operational clarity:** Maintain observability via terminal logs that record adaptation triggers and executed actions, providing sufficient information during incidents.

III. Background and Related Work

A. Overview of Self-Adaptive Systems

Self-adaptive systems (SAS) are designed to adjust the system's behavior or structure at runtime in response to changes in the environment or the system itself. The primary motivation for this concept is to manage uncertainty and maintain quality of service (QoS) goals, such as performance, availability, and resource efficiency, without requiring manual human intervention. In modern web applications, traffic fluctuations are often volatile. Therefore, there is a need for systems that can automatically detect and handle performance degradation or reduce resource waste.

B. Definition of MAPE-K Loop

The most common reference model for Self-Adaptive Systems' structure is the MAPE-K loop. It stands for Monitor, Analyze, Plan, Execute, and Knowledge.[3] These four functions work together in a loop to control the managed system:

- **Monitor:** This part collects data from the system and the environment. In our implementation, this is like reading CPU usage or response time from the sensors.
- **Analyze:** The analysis phase looks at the data to see if there is a problem. It compares the current situation with the goals or rules stored in the Knowledge base to decide if a change is needed.
- **Plan:** If the analysis finds a problem, the planning phase decides what to do. It creates a set of actions (a workflow) to fix the issue, like deciding to add more servers.
- **Execute:** This phase actually carries out the plan. It uses effectors (actuators) to change the managed system.
- **Knowledge:** This is the shared data that all other phases use. It includes things like policies, logs, and metrics history.

C. The Self-CHOP Properties

A key concept in self-adaptive systems is Autonomic Computing, a vision introduced by IBM. The core idea is to build systems that operate much like the human autonomic nervous system, capable of managing their own behavior without constant human oversight.[4] This framework is defined by four primary objectives, known as the "Self-CHOP" properties:

- **Self-Configuration:** The system can automatically configure its components to adapt to a changing environment. For example, when a new server is added to the cluster, the system should automatically recognize it and start sending traffic to it without an admin manually editing config files.
- **Self-Healing:** The system aims to discover, diagnose, and repair disruptions on its own. If a software module crashes or a hardware node fails, a self-healing system detects the error and restarts the service to maximize availability.
- **Self-Optimization:** The system continuously monitors its operation and tunes its resources to improve performance or efficiency. In our project, this is the most relevant property, as our system tries to minimize response time while also minimizing the number of active pods to save resources.
- **Self-Protection:** The system can anticipate, detect, and defend against malicious attacks or cascading failures. While our project focuses mostly on performance, the "Degraded Mode" also acts as a form of protection, preventing the database from being overwhelmed by too many requests.

D. Architectural Patterns

To build a system that can adapt efficiently, the underlying software architecture is very important. We considered several patterns to support the monitoring and execution phases.

1) Observer and Monitor Patterns: In a self-adaptive system, the "Monitor" needs to know when the state of the system changes. The Observer pattern is often used conceptually here, where the monitoring component subscribes to updates or continuously checks the status of the managed system. In our project, we use the metrics from Sysdig as the sensor to observe the situation of the managed system. This pattern decouples the monitoring logic from the business logic, allowing the application to run normally while the monitor watches from the outside.

2) Decoupled vs. Monolithic Architectures: Traditional applications often use a monolithic architecture, where all functions (UI, business logic, data access) are bundled into a single deployable unit. While this is simple to develop initially, it has a big drawback for adaptation: if one specific module is under high load, the entire application stack must be scaled, which wastes memory and CPU on components that do not need more resources.

In contrast, a decoupled service architecture separates the system into independent components, such as the frontend, backend, and database. These components run in separate containers, and each of them can be deployed and scaled independently based on its specific resource demands. This fits our self-adaptive goals perfectly because we can specifically scale up the backend service when calculation demand is high, without needing to replicate the frontend or other unrelated components.

3) Cloud-Native Considerations: Cloud-native development is about building applications specifically to run on modern cloud platforms. A key technology here is containerization (e.g., Docker), which packages code and dependencies together so it runs the same everywhere.

For self-adaptive systems, containers are much better than traditional Virtual Machines (VMs) because they are lightweight and start up much faster. This "fast startup" is crucial for the "Execute" phase of the MAPE-K loop. When the system detects high traffic, it can launch new pods in seconds to handle the load, whereas a VM might take minutes to boot. Our project uses OpenShift to manage these containers dynamically.

IV. System Overview

A. The Managed System Architecture

The managed system in this project is the Toronto Fitness Club full-stack web application, which follows a classic multi-tier architecture. It consists of a React single-page application (SPA) frontend, a Django-based REST backend, a relational database for persistent data, and a Redis instance used for caching, performance monitoring, and supporting the self-adaptive logic.

B. The Managing System Architecture

The managing system implements the MAPE-K feedback loop for self-adaptation around the deployed fitness application. It is primarily realized through a set of shell and Python scripts that generate load automatically, monitoring, analysis, planning, and execution on the underlying OpenShift deployments, while using Redis and files as the knowledge base.

At the top level, the `run.sh` script is used for local development: it activates the backend virtual environment, starts the Django backend, and then launches the React development server. For experiments related to self-adaptation, the main entry point is `mapek.sh`, which implements the core MAPE-K control loop for performance evaluation under different load levels.

C. The Frontend Architecture

The frontend is implemented as a React single-page application (SPA). It uses React Router for client-side routing and Material-UI components for layout and styling. The application entry point defines the main navigation structure and routes for core features [1]:

- Account management: login, registration, profile viewing and editing, and logout.
- Studio browsing: interactive studio map, list of studios, studio details, and studio discovery based on location.
- Class management: studio class schedules, users' enrolled class schedules, and class history.
- Subscription and billing: adding and editing subscriptions, and viewing payment history.

All API calls are centralized via a configurable base URL, which decouples the frontend from a specific backend host. For example, studio and class data are fetched via standard REST endpoints for listing studios (using geolocation or default coordinates) and managing classes, subscriptions, and payments.

From an architectural perspective, the frontend acts as a pure presentation and interaction layer. It maintains local UI state (e.g., selected studio, search query, pagination) and delegates all business logic and persistence to the backend via REST calls [1].

D. The Backend Architecture

The backend is implemented in Python using Django and Django REST–style views. It exposes a RESTful API that supports the major use cases of the fitness platform [1]:

- User and authentication management (registration, login, profile).
- Studio management, including querying studios by location and listing available studios.
- Class schedule management, including studio-specific schedules, user enrollment, dropping classes, and retrieving personal history.
- Subscription management, including adding, updating, and querying user subscriptions.
- Payment management, including recording payments and providing historical views to the user.

Django models represent domain entities such as studios, classes, users, subscriptions, and payments. A relational database stores these entities. The backend also exposes special endpoints for the self-adaptive scripts.

The backend is containerized using Docker, running Django under the Gunicorn server. In the OpenShift deployment, the backend is exposed as a service and route, and its URL is injected into the frontend configuration during the build process.

E. Database

The database layer provides persistent storage for all business data. Django's ORM abstracts the underlying relational database, enabling the system to store [1]:

- User accounts, authentication data, and profiles.
- Studio information, such as name, location, and facilities.
- Class offerings, schedules (time, instructor, capacity), and enrollments.
- Subscription plans and each user's active subscription state.
- Payment transactions and histories.

The database runs as a separate service from the Django application. The backend connects to it using environment-specific configuration. This separation keeps the managed system modular and allows the managing system to scale or reconfigure application replicas without altering the persistent store.

F. Redis

Redis is deployed as a service in the cluster and is used for both caching and as a shared knowledge store for the adaptation logic. The self-adaptive scripts communicate with Redis using a connection URL provided by the environment configuration. This enables the system to store and retrieve runtime metrics and decisions that inform the adaptation loop. In addition, Redis can be used for caching frequently accessed data or session information to reduce backend response times under load.

G. Deployment Architecture

In OpenShift, the managed system is deployed as two primary applications: `tfc-frontend` (React + nginx) and `tfc-backend` (Django + gunicorn).

The frontend listens on port 8080 and serves the compiled SPA, while the backend listens on port 8000 and provides the REST API. OpenShift routes expose both services externally. The user's browser interacts only with the frontend route, and the frontend communicates with the backend via its route. Redis is exposed internally within the cluster and is not directly visible to end users.

Overall, the managed system architecture is a layered web application where the React SPA offers the user interface, Django provides business logic and REST APIs, the database ensures persistence, and Redis supports caching and adaptation-related state. The frontend and backend are scalable, while Redis has 1 fixed pod.

V. Architecture

A. Component Diagram

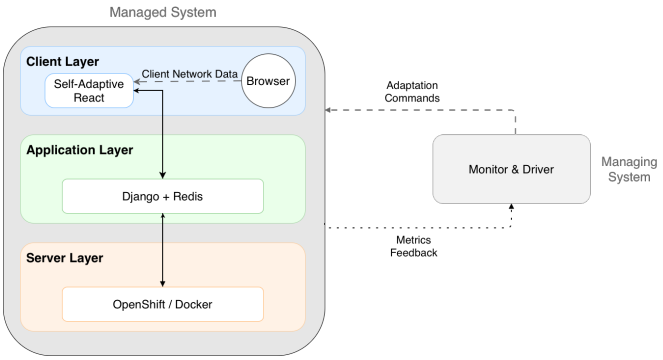

Fig. 1. System-level block diagram of the self-adaptive architecture.

The system is self-adaptive on 3 layers: client, server, and application, as shown in Figure 1. The server and application adaptations are correlated and share one MAPE-K loop, while the client-layer adaptation works independently.

VI. Self-Adaptation Approach

A. MAPE-K Realization

1) Monitor: The monitor `Monitor.py` uses Sysdig to collect metrics, including CPU usage, error rate, and p95 latency, with a sampling interval of 10 seconds from front-end and back-end deployments while the JMeter traffic simulation is running, making adaptation happen in real time.

The client layer adaptation happens on the Studio List page, which contains an interactive studios map and a sorted list of studios with their photos. A `useAdaptiveMode` hook is implemented with `React`, which returns a flag indicating whether to enable the lightweight mode for future rendering (e.g., page navigation). It monitors performance and network conditions by calculating the average FPS on each round, retrieving the device's network type (e.g., 4G, 3G), downlink throughput, round-trip time, and estimating the user's download speed in Kbps.

This hook updates the UI mode type every 5 seconds because of the component re-render caused by state update from its embedded `useFPS` and `useNetworkInfo` hooks.

The `useFPS` hook measures the average FPS by leveraging the browser's `window.requestAnimationFrame` API to execute a callback before each repaint. On every frame, the hook records the current timestamp in a queue that functions as a sliding window over the last five seconds. Timestamps older than five seconds are removed from the front of the queue, and the average FPS is computed by dividing the number of stored frame timestamps by the window duration.

Network type information is obtained through the browser's Network Information API (`navigator.connection`), which exposes metrics such as network type `effectiveType`, browser-estimated download bandwidth `downlink`, and round-trip time `rtt`. These values allow the system to classify the user's connection quality in real time.

To measure download speed, the system performs an active probe every 5 seconds by downloading the site's favicon with a randomized query parameter to disable caching. The probe records both the file size and the time required to retrieve it. Because the favicon is served by the frontend hosting layer (e.g., a CDN) rather than the application backend, the resulting throughput reflects the user's actual network download bandwidth and is unaffected by backend load, server-side processing delays, or database latency.

2) Analyze and Plan:

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

For the utility function defined in Equation (1), weights (e.g., 0.35 for ErrorRate) represent the relative importance of response time, throughput, CPU usage, error rate, and P95 latency.

For server layer adaptation, if any deployment `cpu.cores.used` reaches 0.4 (80% of usage, compared to the limit of 0.5 core), P95 latency

exceeds $2000\,\mathrm{ms}$, or utility score is less than 0.6 during runtime, the `scaleHandler` will be called and a new pod will be added to the service to balance the workload if there are any available pods left. Otherwise, if the pod's `cpu.cores.used` is less than 0.1, which is 20% compared to the limit of 0.5 cores, the `scaleHandler` will try to decrease the number of pods of the service by one if it has more than one pod. Since the resource limit is 0.5 core per pod and each group has a maximum of 8 cores, the maximum number of pods is set to 16.

To prevent oscillations and maintain service stability, the adaptation mechanism applies asymmetric cooldown intervals: a shorter cooldown ($30\,\mathrm{s}$) after scaling up to allow rapid recovery from overload, and a longer cooldown ($90\,\mathrm{s}$) after scaling down to verify that reduced load conditions persist before releasing resources. The distribution of pods is first-come, first-served, until no pods remain available.

The Application Adaptation is triggered only when no pod is left, but the system still requires scale up because it sacrifices user experience: during degraded mode, user-specific methods like sorting by distance is disabled for cache. As a user-oriented system, it should prioritize UX over the cost of resources.

For client layer adaptation, `useAdaptiveMode` hook detects user has a poor network type, including slow-2g, 2g, and 3G, browser-estimated download bandwidth is lower than 1.5, the round-trip time is greater than 300ms, or the kbps download speed is lower than 40. If any of the above conditions are met, the hook marks the performance as unhealthy and increments the `unhealthyCounter`. If the performance is healthy, then the counter will be reset to 0, and the hooks indicate the page to use standard mode. On the other hand, to prevent the impact of temporary network fluctuation, the `useAdaptiveMode` hook only indicates the page to enable lightweight mode when the `unhealthyCounter` is greater than 3.

3) Execute: On the server layer, the `scaleHandler` uses the OpenShift command to scale the service horizontally based on the adaptation plan.

The application layer adaptation is triggered when all pods are consumed. The `scaleHandler` sets a flag in Redis to notify the backend to use the degraded mode (see Figure 2), which disables expensive user-specific features, such as distance-based sorting on the Studio list page, allowing Redis cache. For other static pages that have been cached, it increases the cache expiry time by 10 minutes. Meanwhile, the degraded mode also reduces the pagination size from 9 to 3 and extends the debounce time from 300ms to 1000ms for the search methods on the Studio List page for fewer DB queries.

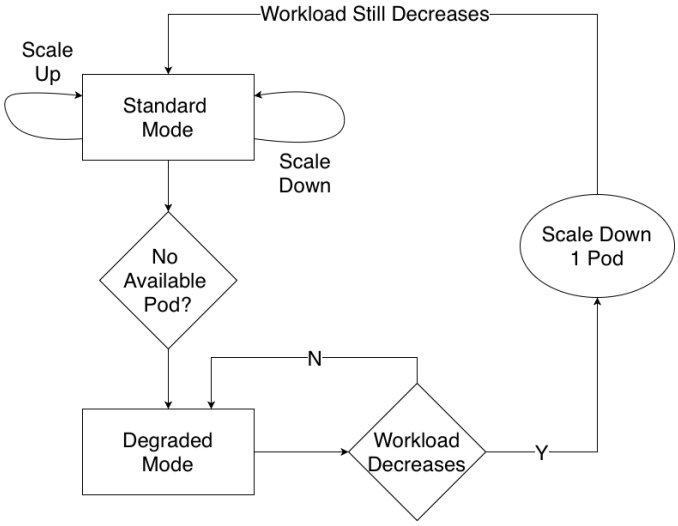

Fig. 2. Workflow between standard and degraded mode

If the workload decreases during the degraded mode, the system will not switch back to the standard mode directly, but decrease the pod by 1, and then decide whether to change to the standard mode in the next round. The reason is that switching back to the standard mode when all pods are running may overload all pods simultaneously and cause instant, significant performance fluctuation. By retaining one active pod first and using it as a buffer, the system maintains stability during the threshold.

For the client-side adaptation, the UI initially starts in a lightweight mode, which excludes the interactive map showing studio locations and displays lower-resolution images for each studio.

The mode state is updated by the `useAdaptiveMode` hook every 5 seconds. Once the client's network is healthy, the state updates and the Studio List page re-renders instantly, enabling the interactive map and using standard-quality images.

If the system later transitions from the standard mode into the lightweight mode, the studio map will not be removed; instead, it is degraded to reduce resource usage by limiting the number of rendered markers and disabling panning and zooming. This approach preserves the value of resources that have already been loaded while preventing additional tile downloads and unnecessary rendering work, yet still provides basic spatial context to the user.

VII. Results

A. Server and Application Level Evaluation

The following graphs compare the results of running 3 levels of workload tests for 300 s with and without adaptation in latency, CPU usage, and error rate. The workload details are listed in Table I.

TABLE I
WORKLOAD LEVELS FOR PERFORMANCE TESTS

Level	Workload
High level	500 threads
Medium level	100 threads
Low level	10 threads

After adaptation, the maximum average response time decreased from 15 s to approximately 6 s under high-level workload (500 threads); from 15 s to approximately 9 s under high-level workload (500 threads), as illustrated in Figure 3. After several tests with various workloads, we found that the system is better improved under the high-level workload than the medium level because the 500 threads test case exhausts all resources and triggers the application adaptation, enabling the cache functionality of the system. Since the system retrieves data from Redis, using a key-value pair table instead of the SQL database, the average response time drops significantly.

After adaptation, the average CPU usage drops after reaching 0.4 cores, compared to the continuous high usage under the baseline version, as shown in Figure 4. Since the maximum limit per pod is 0.5 core, the results align with our adaptation policy: scaling up the application when CPU usage is above 80%, and scaling down when CPU usage is below 20%.

The most noticeable improvement in the error rate happens under the high level of workload test.

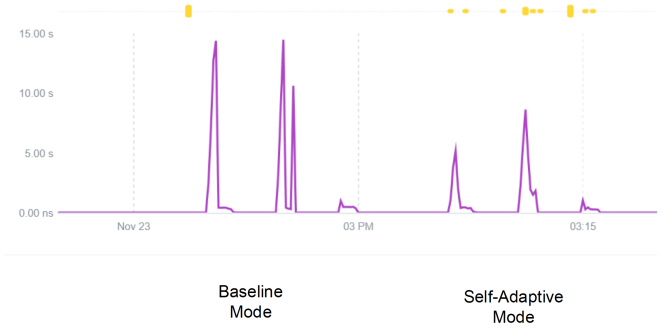

Fig. 3. Comparison of average response time under three workload levels, from high to low

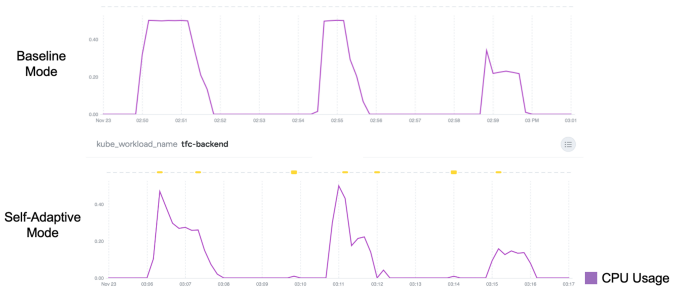

Fig. 4. Comparison of average CPU usage under three workload levels, from high to low

After adaptation, the maximum error rate of the system drops from 46.7/s to 26.1/s, as demonstrated in Figure 5.

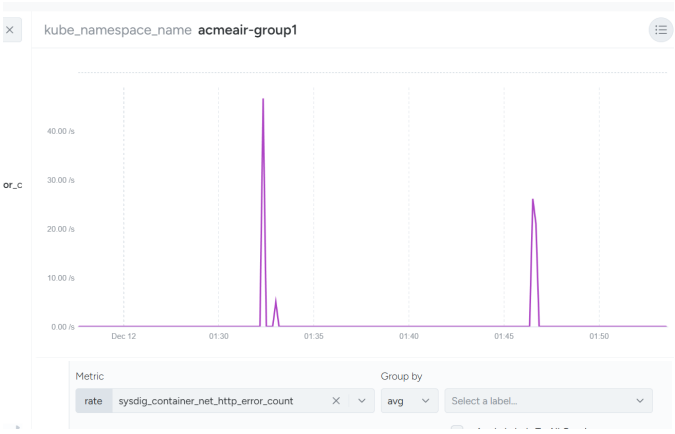

Fig. 5. Comparison of the average error rate under high-level workload, from baseline to the adaptive version

B. Client Level Evaluation

To test the client-level adaptation under poor network conditions, we compared the UI results by

setting the Network in the web browser's developer tools to 3G, as shown in Figure 6.

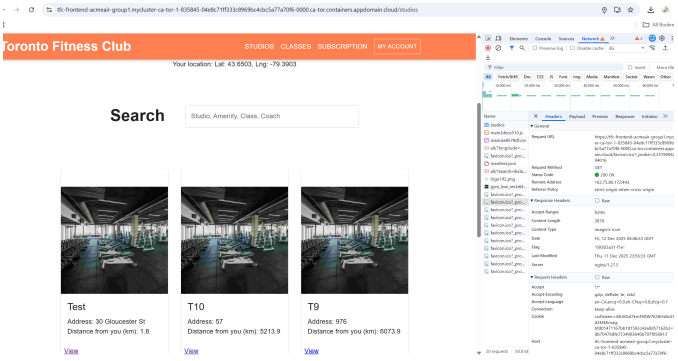

Fig. 6. The initial render of the Studio List page under poor network performance

Under the 3G network, the download speed monitored by our application is around 1.8 Kbps, which is much lower than the threshold of 40 Kbps required for standard rendering. Consequently, the interactive studio map is disabled during the initial render, and low-resolution images are used instead to reduce network and rendering overhead.

If network conditions degrade after the page has been rendered in standard mode, the system transitions to lightweight mode without removing the map entirely. Rather than unmounting the component, it replaces the fully interactive map with a simplified, static variant that shows only essential spatial context (e.g., a limited set of nearby markers) while disabling panning, zooming, and non-essential controls. This approach preserves user context and reduces future resource usage.

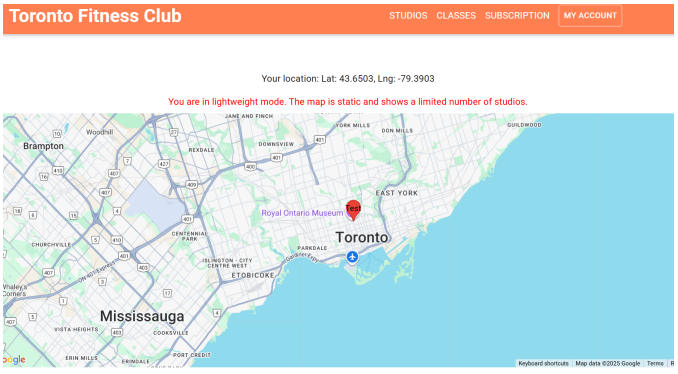

Fig. 7. Change from standard to lightweight mode

VIII. Conclusion

This project addresses the challenge of maintaining system performance and user experience under fluctuating workload and network conditions. To tackle this problem, we designed and implemented a three-layer self-adaptive architecture that integrates server-level, application-level, and client-side monitoring and adaptation mechanisms.

By evaluating the test result, the server and application level adaptation significantly improves system robustness under high workload scenarios by cutting the response time and error rate of the system nearly in half, while reducing resource consumption when the workload is low.

On the client side, the adaptive UI strategy further enhances resilience to poor network conditions. By simplifying the user interface while preserving and constraining already rendered resource-intensive components, the system avoids redundant loading costs and prevents future unnecessary network and rendering. This design maintains a balance between resource utilization, loading performance, and user experience, even when network quality degrades.

Through this project, we learned how to balance user experience and system performance when designing client-side adaptation strategies, and how to continuously refine coordination among the server and application layers by evaluating experimental results. By applying concepts and technologies learned in class in a real-world application, we developed a comprehensive understanding of self-adaptation and its practical implications.

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] Caules Ge and Ethan Li, *A Self-Adaptive Canada Fitness Club Web Application Using React and Django*. University of Waterloo, ECE750: Self-Adaptive Software Systems, October 2025.

[3] Ladan Tahvildari, *ECE 750: Engineering Self-Adaptive Software Systems*, Lecture 1 Slides, Dept. of Electrical and Computer Engineering, University of Waterloo, 2025.

[4] Ladan Tahvildari, *ECE 750: Engineering Self-Adaptive Software Systems*, Lecture 3 Slides, Dept. of Electrical and Computer Engineering, University of Waterloo, 2025.

[5] 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.