A Self-Adaptive System Artifact in Social Media Platform

1st Weiqiang Yu
Team 13
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
w8yu@uwaterloo.ca

2nd Xuecheng Zhou
Team 13
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
x38zhou@uwaterloo.ca

***Abstract*—A Self-Adaptive System (SAS) [1] is a dynamic, feedback-driven closed-loop system designed to achieve adaptivity. The current state-of-the-art self-adaptive systems find application in various domains [2], including web services [3], IoT [4], and Infrastructure as a Service (IaaS) [5]. With the proliferation of social media platforms, there is an escalating demand for visual content, resulting in a surge of server requests. Consequently, the implementation of self-adaptation techniques has become essential to enhance both user experience and system stability, as indicated in recent studies. In response to this growing need, we have developed a unique system known as SAINS, which is dedicated to accelerating the display speed of social networking site pages containing images. This innovative system is rooted in the principles of self-adaptation. To illustrate its efficacy, we present an example demonstrating the utilization of SAINS as an experimental web service tailored to test and showcase novel self-adaptation solutions.**

***Index Terms*—Self-adaptive system, Social Networking Service**

I. Introduction

A superior social media server must not only deliver exceptional availability, scalability, and reliability for applications but also confront challenges arising from abrupt traffic spikes or workload fluctuations induced by heightened demand, seasonal variations, or unforeseen events. This necessitates a nuanced consideration of the intricacies and risks associated with tasks like uploading and viewing images on a social media platform. An eminent challenge in this context is the heightened demand for bandwidth and processing power when numerous users concurrently access the website. This surge can culminate in prolonged loading times for images, detrimentally impacting user satisfaction, experience, and the overall performance and ranking of the website.
Furthermore, the server's susceptibility to overload or crashes due to constrained resources amplifies the stakes, rendering the website inaccessible and jeopardizing user experience. This predicament not only poses risks of data loss, revenue decline, and reputation damage but also opens avenues for potential legal complications and security breaches. A pertinent example is Instagram, a platform that grapples with challenges such as lag and load failures during sudden traffic spikes. Despite the implementation of a distributed system featuring multiple servers strategically positioned globally, this setup does not inherently guarantee the punctual and reliable delivery of images to users.
The efficacy of the distributed system is contingent on an interplay of various factors, including network latency, server capacity, access restriction algorithms, and fault tolerance mechanisms. Thus, Instagram, like any other social media giant, must engage in continuous vigilance, monitoring, and optimization of its distributed system to contend with unpredictable workloads and furnish users with a seamless and gratifying experience. This perpetual endeavor involves a meticulous assessment of factors such as network latency, server capacity, access restrictions, and fault tolerance mechanisms to ensure the efficient and dependable delivery of images. By proactively addressing these challenges and fine-tuning its distributed infrastructure, Instagram can not only fortify its standing in the competitive social media landscape but also bolster user trust and satisfaction. In the dynamic realm of social media, the pursuit of excellence necessitates an unwavering commitment to adaptability, optimization, and the perpetual refinement of systems to meet the evolving demands of a global user base.

II. Methodology

A. Run-time Model

Our proposed solution leverages the runtime model WATINS, a specialized social networking software meticulously designed to augment social interactions and foster connections within the academic community of the University of Waterloo. The comprehensive architecture of the WATINS system is delineated in Figure 1, featuring five discrete micro-services, each tailored to fulfill a specific core function:

- **Main Service (Front-end):** Serving as the front-end, this micro-service facilitates user access to all platform content. Users can seamlessly peruse posts from their followers. The front-end is meticulously crafted using the React framework for functionality, Tailwind CSS for formatting and styling, and Typescript for heightened security.
- **Authentication Service:** Responsible for user authentication, this service ensures that only authenticated users

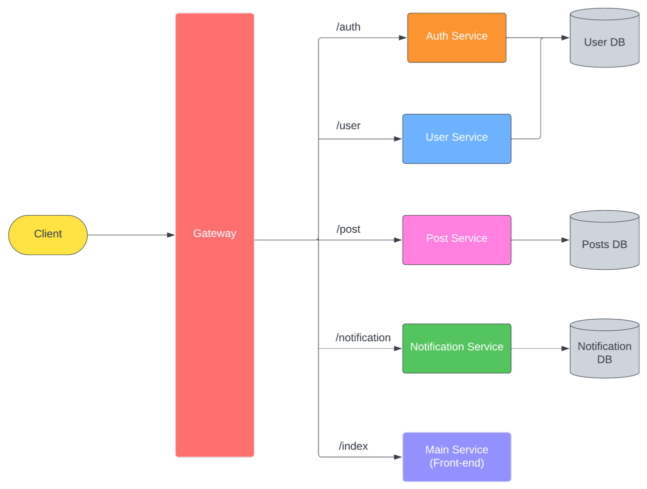

Fig. 1. WATINS Run-time Model

can access the website. Registration on the platform is a prerequisite for accessing various provided services.

- **Post Service:** This micro-service facilitates user interactions with individual posts, allowing users to add comments and likes to each post. Additionally, users can effortlessly upload new photos or videos to WATINS, thereby enhancing their creative expression within the platform.
- **User Service:** Through our user services, individuals possess the ability to update their profiles, manage their media content, and explore other users' profiles. Users can also personalize their experience by changing avatars and choosing to follow or unfollow new connections.
- **Notification Service:** The notification service enables users to receive alerts for activities such as new followers, likes, or saved posts.

The back-end services were implemented using the Spring Framework, with MongoDB serving as the database for storing metadata, including user, post, and notification information. While storing images/videos in the database is a conceivable approach, it was observed that storing data in BSON is inefficient and unsustainable as a long-term solution for our platform. To address issues related to inconsistency and scalability, we opted for IBM Cloud Object Storage to house our media, such as images and videos. This strategic decision ensures that even with a scaled post service, media retrieval remains expeditious.

All these micro-services collectively coalesce into a robust system, providing a seamless and secure user experience on our platform.

B. SAINS

While WATINS offers an array of services and features, its existing infrastructure lacks the requisite adaptability to effectively manage surges in incoming requests, potentially resulting in service unavailability during periods of high demand. Such limitations have the potential to significantly compromise the overall user experience. In response to these identified challenges, this study introduces a self-adaptive system artifact specifically engineered to optimize the performance, availability, scalability, and cost-effectiveness of social networking services. SAINS (Self-Adaptive Instagram) is an experimental framework meticulously designed to augment the adaptability and reliability of web services. The primary objectives include enhancing overall system efficiency and ensuring an elevated standard of user experience. To gain a visual understanding of SAINS's architectural framework, readers are encouraged to consult Figure 2. Through the strategic integration of self-adaptive mechanisms, SAINS endeavors to transcend the limitations posed by static infrastructures, providing a dynamic and responsive foundation for social networking services.

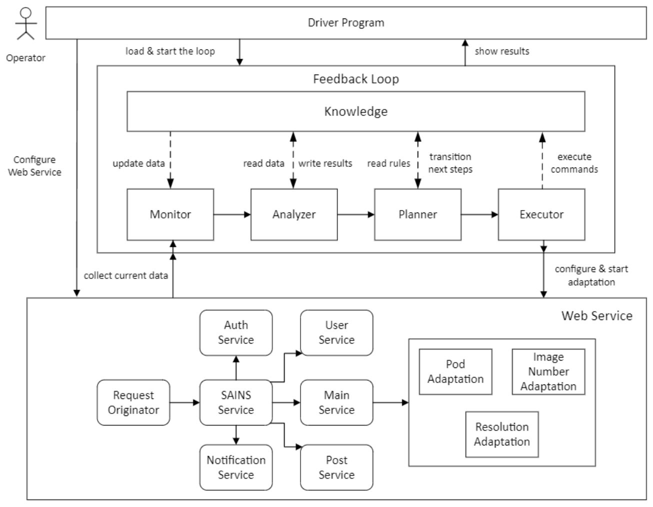

Fig. 2. Architecture of SAINS

Prior to incorporating adaptive features into our system, a meticulous exploration of uncertainties and quality requirements was conducted.

From our vantage point, the primary source of uncertainties emanates from the dynamic nature of web traffic. The variability in the volume of requests and active users per hour constitutes a fundamental parameter introducing significant uncertainty into the server environment. Escalations in these metrics can precipitate a cascade of challenges, encompassing issues such as congestion, latency, errors, or potential system failures. The existing server capacity may prove insufficient to adeptly handle the intensified demands, leading to suboptimal performance and a reduction in overall system availability.

In particular, the fluctuating workloads can manifest in discernible disparities in CPU and memory utilization, thereby further accentuating the uncertainties associated with dynamic traffic patterns. To effectively address these uncertainties arising from the variable nature of web traffic, it becomes imperative for the server to embody an intrinsic adaptability. This adaptability empowers the server to autonomously recalibrate its capacity and dynamically allocate resources during periods of heightened traffic load, ensuring optimal performance and resilient responsiveness.

Table I succinctly delineates the specific quality requirements that have been identified for our system.

TABLE I
QUALITY REQUIREMENT

Quality Requirement
CPU usage percentage per 3-minute period should not exceed 50%
Memory usage percentage per 3-minute period should not exceed 50%
Response latency per 3-minute period should not exceed 7 seconds

Our quality requirements are meticulously formulated to ensure equitable resource utilization, concurrently optimizing the user experience by mitigating response latency within its capacity. In line with these objectives, the adaptation goals are succinctly delineated in table II: The central objective of SAINS revolves around optimizing user experience and diminishing response times for user-initiated requests. This overarching adaptation goal is expounded in Tables III, where a comprehensive depiction of the associated strategies is presented. The intricate alignment of these strategies with the identified adaptation goals underscores the system's commitment to achieving heightened responsiveness and user satisfaction. By meticulously outlining these goals and strategies, SAINS aims to establish a framework that not only acknowledges but actively addresses the critical aspects of user interaction, contributing to a more efficient and gratifying user experience.

TABLE II
ADAPTATION GOALS AND COMPOSED CONTEXTS

Adaptation Goals
Optimize response latency
Keep CPU Usage under 50% and above 25%
Keep Memory Usage under 50% and above 25%
Keep Pod number between 1 and 5
Keep Image Loading Quantity between 9 and 18
Keep Image Resolution between low and high quality

TABLE III
ADAPTATION GOALS AND ASSOCIATED TASKS

Adaptation Goal	Strategies
Optimize response latency	Increase the pods; reduce image resolution; reduce image numbers
Keep CPU/Memory Usage under 50% and above 25%	Increase/decrease the pods; increase/reduce image resolution; increase/reduce image numbers

C. MAPE-K Loop

The principle strategy in our case is "MAPE components share a common set of run-time models" [6], as illustrated in figure 3.

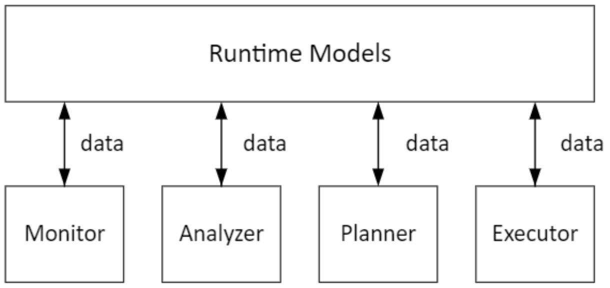

Fig. 3. Principle Strategy for SAINS

The detailed explanation of the MAPE-K [7] loop is as follows:

1) Monitor: In the course of our study, Sysdig was implemented within the IBM Cloud infrastructure to facilitate real-time data monitoring. The integration of Sysdig into IBM Cloud Monitoring proved instrumental in furnishing a comprehensive analysis of performance, availability, and security parameters.

Within the scope of our project, we meticulously defined a set of metrics, each characterized by distinct aggregation methodologies, including but not limited to averages and summations. The retrieval and storage of data for each metric were executed through the invocation of dedicated functions, specifically designed to communicate with a remote server and persistently archive the acquired information in JSON format. Notably, the pivotal metrics of "cpu_quota_used_percent", "memory_limit_used_percent," and "net_http_request_time" assumed a central role in subsequent analytical steps.

To ensure the systematic and periodic acquisition of metrics from the cloud environment, an iterative mechanism was incorporated into the main program. Following each data collection cycle, the program systematically enters a 4-minute dormancy period before initiating the subsequent iteration. This procedural intricacy serves to facilitate real-time monitoring and enables timely adjustments in response to emerging exigencies.

2) Analyzer: Subsequent to the retrieval of data, the analyzer engages in the computation of pertinent metrics, including average CPU utilization, memory consumption, response time, and request volume, specifically for identified microservices. The metrics thus computed are systematically evaluated against predetermined quality benchmarks. In the event of any deviation from the stipulated quality thresholds, the analyzer instigates an adaptation process. For each distinct scenario, the analyzer formulates a spectrum of adaptation alternatives. Illustratively, when confronted with a response time surpassing 7 seconds, three adaptation plans are proposed:

1) Plan 1 involves augmenting the pod count by 1
2) Plan 2 entails diminishing the quantity of images displayed on the webpage
3) Plan 3 recommends a reduction in the resolution of exhibited images to ameliorate response time and mitigate

server load.

Prior to the implementation of these alternatives, a meticulous feasibility assessment is conducted, mindful of the constraints imposed by limited available resources, as encapsulated in the inequality formulation (e.g., number of pods ≤ 5).

Post feasibility validation, the analyzer proceeds to evaluate forthcoming CPU, memory, and response time trajectories through the utilization of pre-computed data combinations. Leveraging Python's sklearn library, a rudimentary linear regression model is employed for data prediction. A series of experiments, contingent on varied services, workloads, pod allocations, image loading quantities, and image resolutions, is initially conducted. Initial data pre-processing steps involve the application of one-hot encoding to represent discrete values as binary vectors. Subsequent to data accrual, a simple linear regression model is trained, with cross-validation protocols implemented to evaluate model performance. Once the model is established, real-time data can be fed into it to predict average CPU, memory, and response time for each configuration proffered by the analyzer.

Following the acquisition of data from the predictive model, a subsequent step involves the assessment of utility for each configuration. In the context of our project, the definition of the utility function is tailored to align with our adaptation objectives. The parameterized form of the utility function is articulated as follows:

$$\text{utility}_{\text{CPU}} = \begin{cases} 1 & \text{if CPU Usage} \leq 25\%, \\ 0.5 & \text{if } 25\% < \text{CPU Usage} \leq 50\%, \\ 0 & \text{if CPU Usage} > 50\%. \end{cases}$$

$$\text{utility}_{\text{Memory}} = \begin{cases} 1 & \text{if Memory Usage } \leq 25\%, \\ 0.5 & \text{if } 25\% < \text{Memory Usage} \leq 50\%, \\ 0 & \text{if Memory Usage } > 50\%. \end{cases}$$

$$\text{utility}_{\text{Latency}} = \begin{cases} 1 & \text{if Response Latency Times} \leq 3.5, \\ 0.5 & \text{if } 3.5 < \text{Response Latency Times} \leq 7, \\ 0 & \text{if Response Latency Times} > 7. \end{cases}$$

$$\text{utility}_{\text{Resource Cost}} = \begin{cases} 1 & \text{if we change the image quantity,} \\ 0.5 & \text{if we change the image resolution} \\ 0.25 & \text{if we change the pod number.} \end{cases},$$

$$\text{utility} = 0.2 \times \text{utility}_{\text{CPU}} + 0.2 \times \text{utility}_{\text{Memory}} + 0.45 \times \text{utility}_{\text{Latency}} + 0.15 \times \text{utility}_{\text{Resource Cost}}$$

Evidently, the preeminence of the response latency weight within our system is manifest, as it encapsulates the primary focal point of our ongoing optimization endeavors. Concurrently, due consideration is accorded to the weights assigned to CPU, memory, and resource cost. This judicious distribution of weights is intended to ensure the holistic optimality of the devised plan, striking a balance between enhancing user experience and mitigating the costs associated with our services.

Ultimately, the computed utility, derived from the predicted data and configuration parameters, is encapsulated within a unified object, which is subsequently transmitted to the planner for further procedural iterations.

3) Planner: The implementation of the planner is characterized by its direct and efficient nature, predominantly relying on a rule-based methodology. The planner's algorithm meticulously generates adaptation plans by evaluating a predetermined set of adaptation options. A cardinal tenet underlying this approach posits the selection of the plan exhibiting the highest utility among the myriad possibilities. Subsequently, the identified optimal plan is transmitted to the executor for execution. This streamlined process underscores the efficacy of the rule-based paradigm employed, aligning with principles of efficiency and systematic decision-making within the framework of adaptive systems.

4) Executor: The *Executor* assumes the pivotal role of interfacing with an external URL, undertaking the execution of adaptation plans designed to alter the configurations and parameters of the SAINS server. Its purview encompasses several essential methods, elucidated as follows:

1) **Image Loading Quantity Configuration (Method 1):** Initiates a modification by transmitting a POST request to the REST API embedded in the back-end server.
2) **Image Resolution Adjustment (Method 2):** Commences alterations by dispatching a POST request to the back-end server's REST API.
3) **Pod Adaptation Plans Execution (Method 3):** Enacts adaptation strategies for pods through the execution of bash scripts, effecting changes in CPU, memory, replica, and service configurations.

Methods 1 and 2 utilize the HTTP POST method to dispatch configuration parameters, such as image loading quantity and resolution, to the SAINS server. The server, in response, processes these requests, updating the corresponding image loading quantity and quality settings. The realization of automated configuration modifications involves the manipulation of associated YAML files. To streamline this process, a dedicated shell script has been devised.

This approach underscores a systematic and scalable methodology for orchestrating dynamic adjustments within the SAINS server's infrastructure.

D. Technology

In the process of taking our application to the online domain, we opted for IBM Cloud as our deployment platform, a strategic choice driven by its robust features and reliability. Prior to the actual deployment phase, we meticulously crafted several Docker files to encapsulate our application's components, ensuring seamless portability and consistency across various environments. The Docker compose command proved instrumental in orchestrating these containers, allowing us to rigorously test the functionalities and interactions among different elements.

Once satisfied with the local testing, we proceeded to formalize our deployment strategy by creating corresponding

YAML files for deployments, services, and routes. These configurations were carefully tailored to IBM Cloud's infrastructure, ensuring optimal performance and resource utilization. To realistically gauge our application's response to varying levels of user activity, we employed JMeter, a versatile testing tool. Through JMeter, we simulated an influx of requests, allowing us to assess the system's scalability, responsiveness, and overall adaptation to real-world usage scenarios. This meticulous testing phase aimed not only to verify the application's immediate functionality but also to anticipate and address potential challenges that might arise under the dynamic conditions of a live online environment.

E. Three-layer Architecture

SAINS incorporates a methodology grounded in architectural principles [8] to facilitate self-adaptation. The three-layer architecture [9] [10] in self-adaptive systems comprises Goal Management, Change Management, and Component Control. Goal Management establishes high-level objectives, serving as the foundation for system behavior. Change Management assesses the system state against goals, determining adaptations needed for alignment. Component Control executes selected adaptations at the component level, ensuring real-time adjustments without compromising stability. This hierarchical structure enables dynamic responses to changing conditions, maintaining the system's alignment with its intended objectives and requirements, and is integral to the self-adaptation capabilities of complex systems in various domains.
As shown in figure 3, the SAINS architecture comprises three pivotal layers that collectively ensure adaptive behavior and optimal performance:

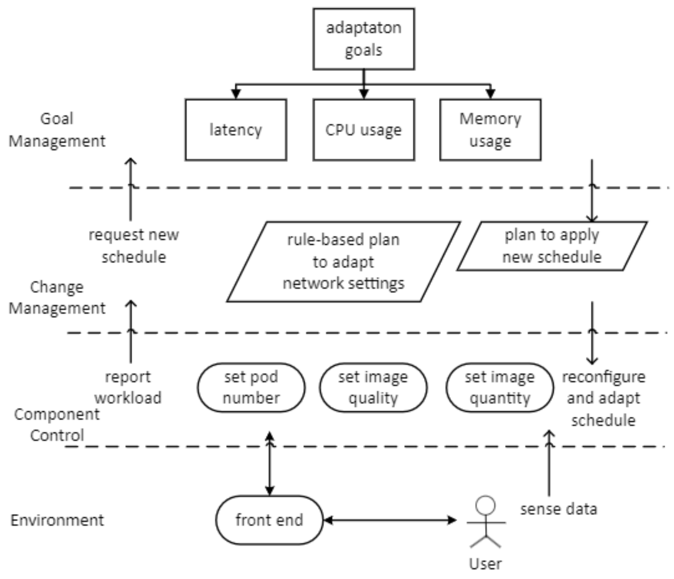

Fig. 4. SAINS Micro-service Architecture

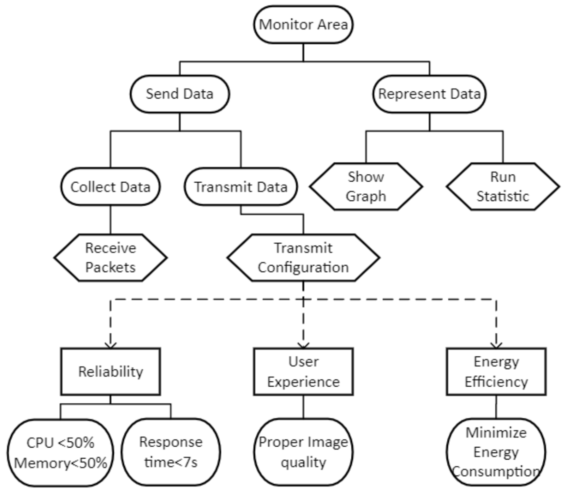

Fig. 5. Meta-Requirements for Self-Adaptation of SAINS

- **Goal Management Layer:** The Goal Management layer in SAINS encompasses a set of objectives aimed at optimizing system performance. These objectives include minimizing response latency and ensuring equitable CPU and memory utilization. The system strives to maintain resources within an appropriate range, mitigating both idle and overused resources.
- **Change Management Layer:** The Change Management layer integrates rule-based plans, enabling the system to autonomously adapt to varying network settings. These plans serve as guidelines for dynamic adjustments, ensuring the system's resilience and responsiveness to changing environmental conditions.
- **Component Control Layer:** The Component Control layer of SAINS possesses the capability to detect environmental changes and trigger the adaptation process when necessary. This layer acts as the operational executor, implementing changes dictated by the higher layers to maintain system integrity and functionality.

In the event of a quality requirement failure detected by the Component Control layer, it promptly reports to the Change Management layer. Subsequently, the Change Management layer initiates a request to the Goal Management layer for a set of feasible plans. The Goal Management layer meticulously analyzes the situation, generating multiple suitable plans with associated utilities. The Change Management layer evaluates these plans, selecting the one with the highest utility, which is then executed on the Component Control layer, ensuring the continuous alignment of the system with dynamic operational conditions.

F. Requirements-Drive Adaptation

Various strategies exist for integrating Requirements-Driven Adaptation into self-adaptive systems. For instance, Relax [11] introduces a novel requirements language specifically tailored for self-adaptive systems. In our project, we have chosen a distinctive methodology, wherein we conceptualize Modeling Requirements for Adaptive Behavior as Meta-Requirements [12]. This unique methodology serves to address the challenge of managing conflicting requirements, such as Reliability, User Experience, and Energy Efficiency, within our self-adaptive system. The intricate interplay of these requirements is visually represented in Figure 5. Through this strategic framework, we establish a robust foundation to govern the adaptive behavior of the system, thus enhancing its overall performance and coherence.

Reliability, defined by parameters such as CPU usage (<50%), memory usage (<50%), and response time (<7s), is crucial. However, it must be balanced with User Experience, which calls for maintaining proper image quality and quantity, and Energy Efficiency, which necessitates minimizing energy consumption and maximizing resource utilization.

The meta-requirements serve as high-level rules, stipulating how the system should adapt under specific circumstances. For instance, when CPU usage exceeds its threshold, the system might offload tasks or degrade non-critical functionalities to enhance Reliability. Similarly, if image quality drops below a satisfactory level, the system could adapt by increasing image resolution or loading speed, thereby enhancing User Experience. However, such adaptations are always cognizant of the Energy Efficiency requirement.

These meta-requirements offer an integrated view of system performance. While trying to meet one requirement, the system ensures it does not disproportionately compromise the others. This holistic perspective allows our self-adaptive system to effectively balance conflicting requirements and deliver optimal performance under varying conditions. This approach forms the essence of our design strategy, enabling the system to adapt swiftly and efficiently, always aligning with its intended goals.

III. Obtained Results

We employed JMeter as a load testing tool for SAINS service in our project. We designed HTTP Requests to target the login, user, and post services of our web page, introducing differing workloads to simulate a variety of usage scenarios. JMeter's thread count was programmed to change every four minutes, escalating from a baseline of 0 threads to 3, then to 10, and finally peaking at 50 threads, before gradually decreasing in the same succession.

Figure 6 and 7 depict this process of escalating load. Figure 6 presents the dynamics of CPU load without the application of any adaptive system. Notably, at 16:26, 16:30, and 16:35, the server experienced sudden surges in accesses, mirrored by steep increases in CPU usage. Ultimately, the CPU load reached a critical level of around 50%, illustrating a precarious situation—any further increase in accesses would risk overloading the server, as it would struggle to manage the corresponding volume of requests.

Conversely, Figure 7 demonstrates the CPU load dynamics after the deployment of our self-adaptive system. At 16:51, a

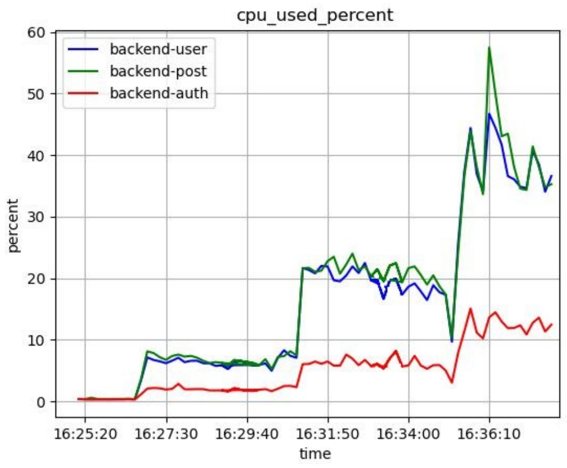

Fig. 6. CPU Usage Without Self-Adaptation

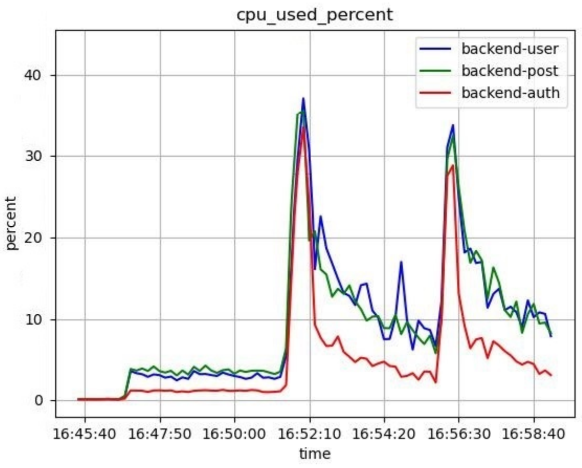

Fig. 7. CPU Usage With Self-Adaptation

sudden surge in traffic was quickly responded to by the system, which increased the number of operating pods. However, due to the time required for the pod count to update, the CPU usage only gradually returned to a normal level after 16:52.

A similar situation occurred at 16:55, when a sudden load spike prompted the system to reduce both the number of images and their resolution. Following these adjustments, by 16:56, the server was successfully modified and the CPU usage had returned to its normal range. This comparison clearly demonstrates the effectiveness of our self-adaptive system in managing sudden increases in server load.

Figure 8 and 9 present a comparison of server access time before and after the deployment of the self-adaptive system. Figure 8 reveals a gradual increase in access time when the system is not deployed, as server load intensifies. Initially, access times were in the hundreds of milliseconds, but these rose to approximately 6 seconds, and even surpassed 10 second in peak load conditions. Such lengthy access times are highly

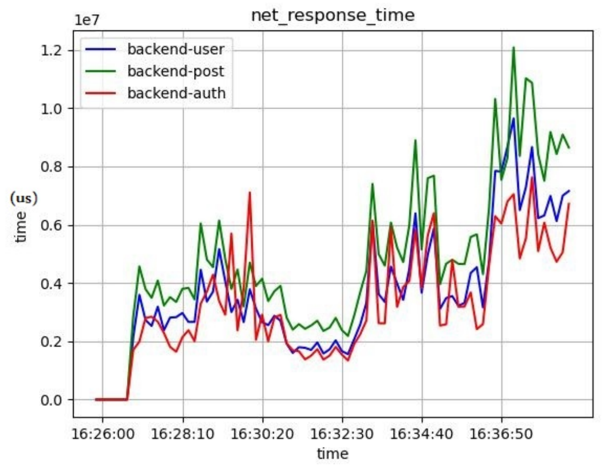

Fig. 8. Response Time Without Self-Adaptation

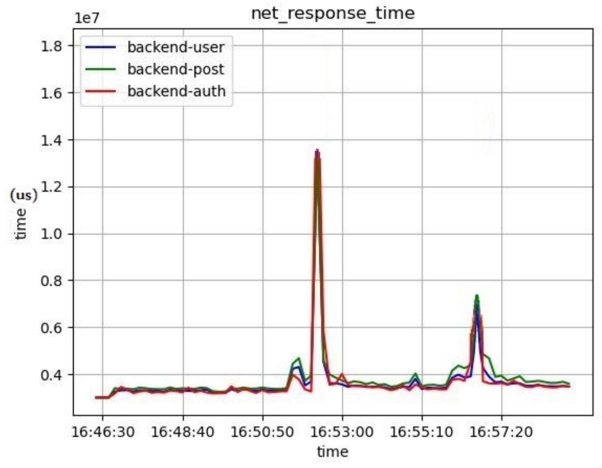

Fig. 9. Response Tim With Self-Adaptation

detrimental to user experience.

In contrast, Figure 9 demonstrates the effectiveness of the self-adaptive system. Upon a sudden increase in load at 16:52, the server's response time also spiked. However, within just a minute, the server was able to adjust according to the quality requirements, successfully maintaining the response latency within the 7-second threshold. A similar scenario occurred at 16:56 when a high volume of accesses re-emerged. Despite this spike in demand, the server was able to swiftly respond and adjust, ensuring stable and low response times for user, post, and auth microservices. This reaffirms the robustness and agility of our self-adaptive system in optimizing server response times while handling substantial load increases or decreases.

IV. Conclusion

This paper introduces a novel self-adaptive system, denoted as SAINS, designed as an artifact within the context of a social media platform. The inherent nature of our solution lies in its capacity to dynamically adjust its behavior during run-time, responding to fluctuations in network traffic. While SAINS presents a noteworthy advancement, it is essential to acknowledge the potential areas for refinement and enhancement within the system.

One notable aspect that warrants attention is the absence of asynchronous functions in our current back-end services, a deficiency that may impede service responsiveness. Incorporating asynchronous functionality could ameliorate this drawback, ensuring a more streamlined and responsive system overall. Additionally, the present reliance on MongoDB for metadata storage and IBM Cloud Object Storage for media storage could be optimized further. The integration of Redis as a memory database, specifically for caching media, holds the promise of substantially enhancing response time, particularly when users access the same media repeatedly.

The potential for improvement and extension of SAINS is not confined solely to technical dimensions but extends to the broader realm of adaptability in service provision. For instance, introducing a request throttling adaptation strategy offers a viable means to curtail the frequency with which a single user accesses the website, thereby addressing security concerns associated with excessive and potentially malicious traffic. Additionally, in the realm of notification service, the adaptive strategies of the application could be further extended.

The adoption of self-adaptive strategies inherently promises heightened flexibility and usability in comparison to conventional methodologies. By exploring and implementing these adaptive strategies, SAINS stands poised to evolve into a more robust and versatile system, offering a nuanced response to the evolving landscape of social media platforms.

Acknowledgment

We would like to extend our deepest gratitude to professor Ladan for her invaluable instruction and guidance throughout this project. Her knowledge and insights have greatly enriched our work and understanding.

We also want to express our sincere thanks to our teaching assistants, Mingyang and Ryan. Their assistance and dedication have been instrumental in the successful execution of this project.

Furthermore, we would like to acknowledge the technical support we have received from IBM which has significantly helped us in developing and testing our self-adaptive system.

References

[1] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), may 2009.

[2] Terence Wong, Markus Wagner, and Christoph Treude. Self-adaptive systems: A systematic literature review across categories and domains, 2022.

[3] K. S. May Chan and Judith Bishop. The design of a self-healing composition cycle for web services. *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 20–27, 2009.

[4] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansée, Danny Weyns, and Danny Hughes. Deltaiot: A self-adaptive internet of things exemplar. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '17, page 76–82. IEEE Press, 2017.

[5] Support for disconnected operation via architectural self-reconfiguration. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, page 114–121, USA, 2004. IEEE Computer Society.

[6] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, oct 2009.

[7] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[8] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*, 14(3):54–62, 1999.

[9] Erann Gat. *Three-Layer Architectures*, page 195–210. MIT Press, Cambridge, MA, USA, 1998.

[10] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, 2007.

[11] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, 2009.

[12] Esma Maatougui, Chafia Bouanaka, and Nadia Zeghib. Towards a meta-model for quality-aware self-adaptive systems design. In *Mod-Comp@MoDELS*, 2016.