A Self-Adaptive System in Information Sharing Platform

Team 12

Guanjie Wang
University of Waterloo
Waterloo, Ontario, Canada
t73wang@uwaterloo.ca

Lucian Zhao
University of Waterloo
Waterloo, Ontario, Canada
l32zhao@uwaterloo.ca

***Abstract*—Modern web applications must navigate unpredictable environments, making it impractical to list all configurations during the designing phase. Thus, dynamic run-time planning is crucial for self-optimization. Self-adaptive software, which adjusts behaviour in response to environmental changes, is essential in this context. These systems must decide when and how to adapt, balancing between performance and cost. Given the initial Food Lover, a non-adaptive web application allowing users to share recipes, we developed a self-adaptive Food Lover (SAFD), rooted in adaptive computing, dynamically adjusts to diverse operational environments, ensuring high performance and responsiveness bolstered by the integration of run-time models [1] and three-layer architecture [2] adaptation. Incorporating a multi-layer architecture, which views the system as serious components interconnected by connectors, enhances system adaptability and improves the system's flexibility and efficiency. This architecture allows changes in both components and interconnections, crucial for dynamic software architectures. Furthermore, the run-time model provides another way to handle system unexpected changes by horizontally scaling the system. Last but not least, apart from the vanilla MAPE-K loop [3], several different analyzers, such as statistical approach and time-series forecasting based on machine learning are implemented and introduced into the proposed system, and we tried to switch them depending on different conditions.**

***Index Terms*—Self-Adaptive, MAPE-K, Statistic Analysis, Rule-based Analysis, Machine Learning**

I. Introduction

A. Problem Description

A high-quality information server needs to simultaneously achieve three key attributes: high performance, high concurrency, and high availability to ensure a stable user experience during different times, seasons, and traffic fluctuations. However, due to variations in the number of users, different time points, and network conditions, the server's load may also differ. Specifically, the main challenges currently facing the original food lover application include two aspects:

1) Load Expansion Issue: When facing a sudden surge in user traffic, the server may exceed its current maximum load capacity, potentially causing delays in responding to user requests. This can have a negative impact on the server's ability to provide services to users.

2) Extended Wait Times: As the volume of user traffic increases, the waiting time for each user to access the required information also significantly increases. This can adversely affect user satisfaction and the overall user experience.

B. Problem Solution

To address the previously mentioned issues, we have developed an adaptive system which transferred our old system to the new three-layer architecture and integrated the new run-time model. The application architecture is shown in figure 1.

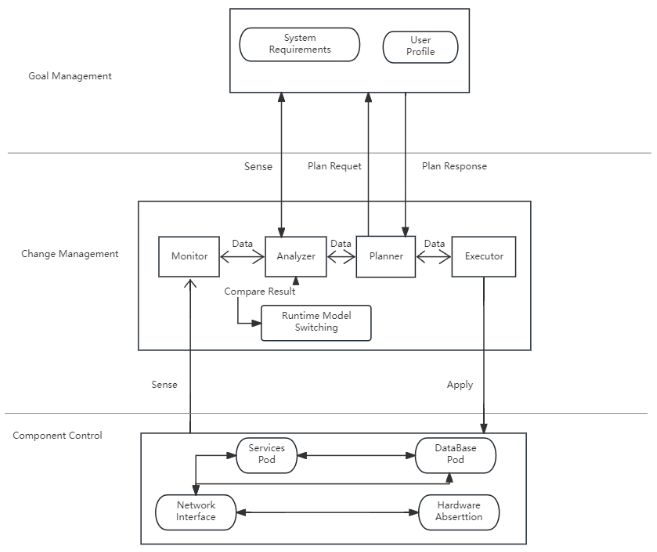

Fig. 1: System Architecture Diagram

Time-series prediction is also necessary to anticipate impending load spikes as the analyzer proactively adapts to our system. Statistical methods like Exponentially Weighted Moving Average (EWMA) and Auto-regression [4], simple but effective techniques for identifying trends of data, are commonly used for workload forecasting in computing systems. Time-series forecasting techniques based on machine learning have been gradually applied for server optimization tasks including resource provisioning, load balancing, and auto-scaling [5]. Long Short-Term Memory (LSTM) networks in particular have proven effective at workload prediction in cloud computing environment [6].

C. Environment

For the SAFD project, we meticulously engineered a dockerization process, encapsulating all essential services within Docker containers to streamline deployment and scalability. We strategically leveraged the robust IBM Cloud as our deployment platform, taking advantage of its reliable and secure infrastructure. Alongside deployment, we harnessed the powerful IBM Cloud monitoring tools, including Sysdig, to conduct vigilant surveillance over our web application's performance metrics and health indicators, ensuring its operational integrity.

Furthermore, we orchestrated real-time adaptations using the OpenShift container orchestration platform, allowing us to swiftly respond to dynamic changes and demands within the operational environment. To validate and stress-test our system's resilience and scalability, we devised a bespoke Python script seamlessly integrated with the JMeter performance testing tool. This script systematically introduced a spectrum of simulated workloads, rigorously challenging the system's error-handling proficiency and adaptability to fluctuating traffic conditions.

This multi-faceted approach not only fortified our system's robustness but also enhanced our capacity for proactive adaptation and maintenance, ensuring uninterrupted service and optimal user experience even under variable and unpredictable workloads.

The used technologies are shown below in table I.

TABLE I: Technology Used

Technology	**Description**
Docker	Service running environment
Python	Data analysis
Sysdig	Data Connection
Flask	BackEnd Services
HTML, CSS, JS	FrontEnd UI
MongoDB and MySQL	Data Storage
OpenShift	Docker Management

D. Components of the system

This project is architecturally segmented into three principal components refer to figure 1, each serving a pivotal role in the system's overall functionality: the application module, the data persistence module, and the dynamic adaptation module. The application module itself bifurcates into a front-end and a back-end submodule. The front end is meticulously crafted using the foundational web technologies—HTML for structure, CSS for presentation, and JavaScript for interactivity—culminating in a user interface that is both intuitive and responsive. This interface serves as the gateway through which users interact with the system, designed with a focus on user experience and accessibility.

On the flip side, the back-end is the system's workhorse, responsible for executing the business logic. It is the backbone that processes user requests, marshalling data between the front end and the database. This segment is powered by the Python Flask framework, a choice that brings to the table simplicity, flexibility, and scalability.

For data management, the system utilizes a dual-database setup: MySQL and MongoDB. MySQL, a relational database management system, is used to store user's confidentiality. In contrast, MongoDB, a NoSQL database, responds to handling unstructured data like PDF files and images and facilitating high-speed transactions necessary for this application.

An in-depth overview of each API endpoint is provided and demonstrated in the table II. These endpoints encapsulate the communication protocols that enable client-server interactions, facilitating operations such as user authentication, resource uploads, and data retrieval, all while maintaining a seamless and secure data flow.

TABLE II: API Endpoints and Descriptions

Page Title **Description**	**Endpoint**	**Method**
Welcome Page Renders the main welcome page.	/ or /welcome	GET, POST
Login Handles user login.	/login	POST
Register Handles new user registration.	/register	POST
Upload Allows users to upload files.	/upload	POST
Description Renders a page with a description.	/description/{pdf_id}/{tr}	GET, POST
Display PDF Displays a PDF file.	/display_pdf/{file_id}/{tr}	GET
Display Image Displays an image.	/display_image/{file_id}/{tr}	GET
Breakfast Recipes Displays breakfast recipes preview.	/breakfast	GET
Lunch Recipes Displays lunch recipes preview.	/lunch	GET
Dinner Recipes Displays dinner recipes preview.	/dinner	GET
Snack Recipes Displays snack recipes preview.	/snack	GET

II. Methods

A. Step-by-step Design

1) MAPE-K loop: After developing a simple MAPE-K loop, we upgraded it by modifying the Analyzer more sophisticatedly which can predict future workloads of the system based on previous data captured in the monitoring stage. We implemented different types of predictors, from statistical models to deep learning methods. Then the results have to be predicted by the appropriate model, relying on a simple model-switching approach. In the planning phase, we have implemented a rule-based Planner to decide whether to increase, decrease, or maintain the service's pod count for the upcoming period, depending on the trend identified by the Analyzer. Then, the executor leverages the output of the Planner to execute the scaling operation for a specific service. This is achieved by comparing the current number of running pods for the service with the count of pods provided by Planner and subsequently calculating the expected pod count which will then be used to execute the required scaling operation in conjunction with a provided command.

2) Adaptation Goal: More importantly, we defined our adaptation goal as maximizing overall throughput while minimizing costs. Its quality requirements can be described as follows:

- R1: During light workloads, ensure that the number of pods does not exceed 10 and minimize costs.
- R2: Maximize throughput consistently across all periods.

These requirements emphasize the need to maintain efficient resource utilization during light workloads by keeping the pod count within a specified limit and optimizing cost. Additionally, the goal is to achieve maximum throughput at all times, indicating a focus on enhancing system performance.

3) Analyzer: The role of the analyzer function is to evaluate the latest information and determine whether the system meets the adaptation goals. If the system does not meet the goals, the analyzer function analyzes potential configurations for adaptation.

In this project, we emphasize improving the analyzing module of MAPE-K loop. Firstly, the analyzer function operates on a predefined time window to initiate its workflow. It begins by evaluating the current conditions against the adaptation goals to determine if adaptation is necessary. Various mechanisms are employed to carry out this assessment. One straightforward approach involves checking whether the system presently violates any of the individual adaptation goals. If such a violation is detected, it triggers the initiation of a system adaptation process. A more sophisticated mechanism involves determining the utility of the system by combining weighted values of relevant quality properties. This approach assigns weights to different quality properties based on their importance. In our system, the predicted workload factor, the count of current pods, and the difference between the current pod count and the expected one are selected as the primary quality properties. To determine their overall utility properly, we give weights 0.3, 0.4, and 0.3 to them respectively after a lot of experiments. This means that the analyzer function will consider the importance of these two properties equally when assessing the system's utility and deciding whether adaptation is needed. Therefore, the utility function could be computed as follows:

$$\mathrm{U}_c = w_{curPod\_cnt} \cdot p_{curPod\_cnt} + w_{estWorkload\_fctr} \cdot p_{estWorkload\_fctr} + w_{Pod\_diff} \cdot p_{Pod\_diff}.$$

The following table III shows the functions of factor value for utility function calculation.

TABLE III: Functions for Calculating Pod Count, Pod Difference, and Workload Ratio

Function	Conditions	Return Value
cal_pod_count(x)	$x \geq 9 : 0.5$ $6 \leq x < 9 : 0.6$ $3 \leq x < 6 : 0.7$ $1 < x < 3 : 0.8$ otherwise : 1.0	Returns the pod count scaling factor.
cal_pod_diff(x)	$x \geq 5 : 0$ $3 \leq x < 5 : 0.1$ $2 \leq x < 3 : 0.2$ $1 \leq x < 2 : 0.5$ otherwise : 1.0	Returns the pod difference scaling factor.
cal_workloadRatio(x)	$x < 0.2 : 1.0$ $0.2 \leq x < 0.5 : 0.9$ $0.5 \leq x < 0.7 : 0.8$ $0.7 \leq x < 0.95 : 0.7$ $0.95 \leq x < 1.1 : 0.5$ otherwise : 0	Returns the workload ratio scaling factor.

As part of our implementation of the assessment of quality properties, the analyzer leverages predictive models to estimate future incoming requests across multiple steps using LSTM by default. To be specific, if the accuracy of the LSTM model used during the training stage falls below a predefined threshold or if the training time becomes excessive, the analyzer will employ alternative but faster methods to achieve medium-quality predictions within the required time frame. The decision to initiate adaptation is then based on comparing the future utility with a threshold value. The reason why we only predict incoming requests is simple because we will modify the count of pods which can be easily estimated by dividing the average number of requests by the predefined workload limit of a single pod. By focusing on predicting incoming requests, we can indirectly estimate the required count of pods for handling the workload. This simplifies the prediction process and allows for easy estimation of pod count based on the workload requirements of your system.

4) Planner: By analyzing the future throughput and cost, the utility value of the system can be computed. This utility value serves as an indicator for the managed system to determine when scaling is required. When the utility value reaches a certain threshold, it signifies that scaling is necessary. However, the decision to scale up or down is determined by comparing the estimated pod count with the current pod count. If the estimated pod count exceeds the current count, scaling up is initiated. Conversely, if the estimated pod count is lower than the current count, scaling down is triggered. This approach ensures that the system dynamically adjusts its resources based on the anticipated workload demands and maintains the desired balance between performance and cost efficiency.

5) Executor: The executor will perform the optimization that the planner identified to the managed system, such as horizontally increasing the pod number.

B. Final SAS Solution

For the overall solution, we have developed a three-layer architecture with run-time model switching in it, the overall architecture is shown in figure 1.

1) Goal Management Layer: The goal management layer mainly contains the user and system requirements, the detail requirements shown below table IV.

TABLE IV: System Requirements

Metrics	**User Requirement**
Cost	Computing resources are minimized as much as possible
Total Error Rate	Must Less Than 5%
Throughput	Maximize throughput consistently across all periods

2) Change Management Layer: The change management layer mainly includes the MAPE-K loop, the detail of the MAPE-K loop is discussed in part A. We need to pay attention to the analyzer, for which we implemented a real-time model-switching algorithm in our MAPE-K loop. The run-time model switching analyzer is shown below in figure **??**.

3) Components Management Layer: The function and components in the component layer have been discussed in the introduction subsection D.

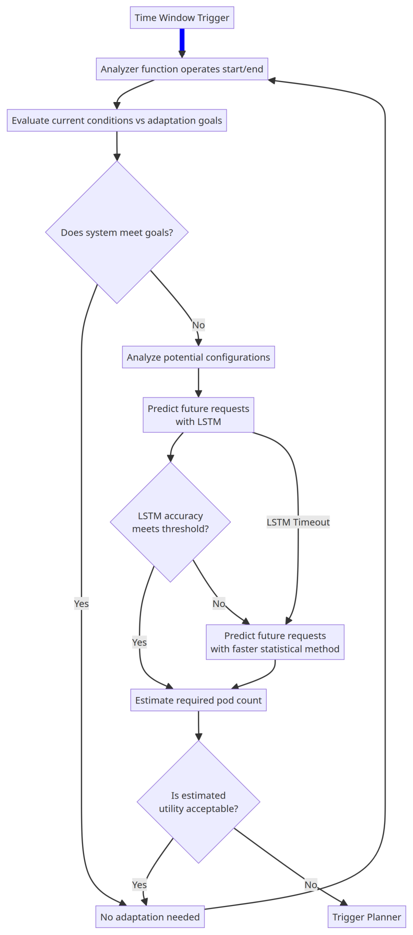
Time Window Trigger
Analyzer function operates start/end
Evaluate current conditions vs adaptation goals
Does system meet goals?
No
Analyze potential configurations
Predict future requests with LSTM
LSTM accuracy meets threshold?
LSTM Timeout
Yes
Yes
No
Predict future requests with faster statistical method
Estimate required pod count
Is estimated utility acceptable?
Yes
No
No adaptation needed
Trigger Planner

Fig. 2: Analyzer Model-Switching

III. Obtained Results

A. Process

1) Implementation: The main() function initializes the key variables and runs the main control loop. It starts by defining the list of services to monitor, as well as a self_adapt flag to enable/disable adaptation. Then it sets up the timestamp variables to track intervals for data collection and adaptation.

The core loop iterates through the services, executing the four MAPE-K stages for each one. First, it calls monitoring() to query the latest metrics from Sysdig. The returned data is pre-processed and passed to analyzing() along with the current pod count. analyzing() focuses on the app service. It extracts the relevant metrics and calls get_utility_func() to calculate a utility score. This function implements the logic to compute utility based on weighted factors like pod count and request rate. It returns both the utility array and estimated pod count. Next, planning() determines the adaptation action - scale up, down, or no change - by comparing the estimated pods to the current count. The command and scale amount are returned. Finally, executing() carries out the scaling operation via OpenShift CLI. It checks the command value and executes oc scale as needed to adjust the pod count. The new count is returned to update the value for the next loop iteration.

In between loops, the code sleeps to create fixed intervals. The timestamps are updated, metrics are collected, and the process repeats continuously.

The monitoring() function authenticates with the Sysdig API using the provided credentials. It constructs the data query with the required metrics, time range, filter, etc. The API response is parsed to extract the metrics into a normalized format with timestamps. This structured data is returned.

analyzing() focuses on the app metrics. Mostly, it passes the predicted request count from predictor_LSTM() to get_utility_func(), if the accuracy is above a predefined threshold and the time of training and prediction is appropriate. This LSTM model calculates the utility score based on weighted factors like pod count and requests. The utility thresholds determine if scaling is needed.

get_utility_func() implements the core logic to compute utility. The workload factor is calculated as requests/max_per_pod. This is fed into utility functions based on thresholds. The pod count utility is also calculated via another function. These weighted values are combined into an overall utility score for that period.

The other functions help with timestamp formatting, plotting metrics, and generating utility plots. They provide logging and visualization capabilities.

Last but not least, the final feedback loop of our implemented self-adaptive system is completed by encapsulating four fundamental methods (monitoring, analyzing, planning, and executing) properly and connecting them. Additionally, the system is designed to run continuously, even 7/24, using a timer. This timer helps us determine the current start timestamp and end time when receiving feedback, allowing for the self-optimization of related services.

In summary, the core MAPE-K loop orchestrates the adaptation process. Monitoring gathers the latest data. Analyzer evaluates the metrics against goals and models to determine if/how to adapt. Planning provides the commands and scale amounts. Execution carries out the scaling operations. The analyzer and utility functions implement the logic to quantify system goals and drive intelligent adaptation decisions. Together they enable continuous self-optimization.

2) Testing: In order to better evaluate SAFD's preference and scalability, we developed a test script using Jmeter to test each API shown at tableII. To simulate the user behaviour and test the timing-related workload, we wrote a driver script using Python to randomly change the Jmeter script's thread number to adjust the system's load.

The detail of the implementation of the Python driver program is that the Jmeter.py includes imports for modules like subprocess, time, datetime, signal, sys, and random, those libraries involve process management, time-based operations, and random value generation. Then script defines a function run_jmeter_test that accepts test_plan and thread_count as parameters, indicating its role in executing JMeter tests with varying thread counts. The script constructs a command to run JMeter tests and employs the sub-process module to execute this command. This setup focuses on automating the execution of JMeter tests, and mainly for the performance testing under different loads (as random libraries produce the varying thread counts).

B. Result

Figure 3 is the utility function without applying the adaptive method in the food lover project, with 5 static pods.

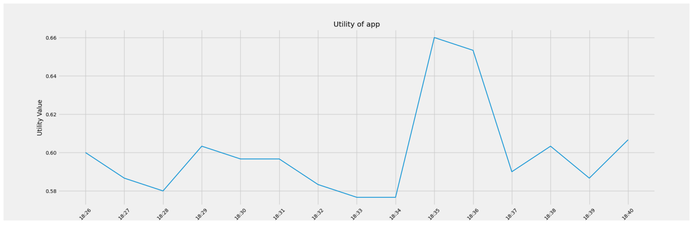

Fig. 3: Utility Function without Enable Adaptation

Figure 4 is the utility function for applying the adaptive method in the food lover project, with 5 pods initially and self-adaptation.

We can see the average value of utility after turning on self-adaptation can reach a little bit higher than Non-adaptive one.

The figure 5 shows the real-time prediction for the incoming request and dynamically changes the service pod count in order to meet the adaptation goals and meet user requirements.

The Planner determines the number of pods to scale up or

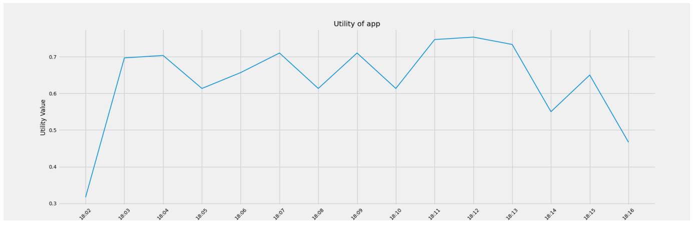

Fig. 4: Utility Function with Enable Adaptation

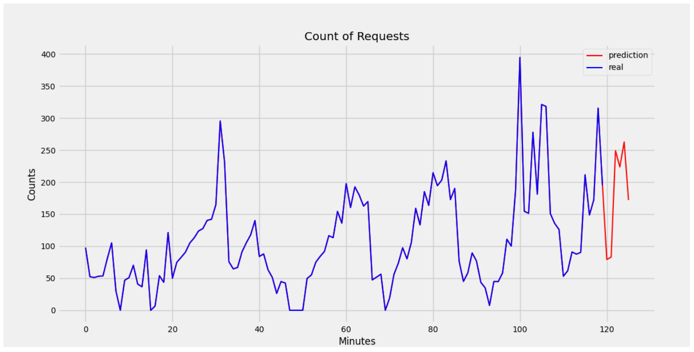

Fig. 5: Real-Time Prediction for Incoming Request

down based on the utility value calculated by the Analyzer using the predicted request counts mentioned earlier. This capability allows our system to detect fluctuations in workloads in real-time and take proactive measures to enhance performance or reduce costs, as needed.

IV. Conclusion

The transformation of the 'Food Lover' web application into the self-adaptive SAFD system, with its innovative three-layer architecture and dynamic run-time model switching, marks a significant leap in addressing critical issues of load expansion and extended wait times during peak usage. At the heart of this transformation is the implementation of the MAPE-K loop, enabling real-time system adaptation through continuous monitoring, analysis, planning, and execution based on changing environmental conditions and performance metrics. This approach, bolstered by a range of predictive models from statistical to deep learning methods, significantly enhanced the system's analytical capabilities, leading to improved accuracy in predictions and more efficient resource management. Our testing confirmed that the SAFD system notably outperforms its non-adaptive predecessor, especially in terms of responsiveness and efficiency, under varying workloads.

V. Future Work

The integration of Model-Switching into the self-adaptive Food Lover (SAFD) system has opened up numerous avenues for future development and enhancements. Building on the system's demonstrated efficacy in handling fluctuating workloads, the focus of future work will primarily be on advancing predictive analytics and optimization strategies. Key areas of interest include:

A. Enhanced Predictive Analytics with Deep Learning

Building on the foundation of Model-Switching, future efforts should focus on integrating more advanced deep-learning algorithms. This would enhance the system's ability to accurately predict workload fluctuations and user behaviour. Employing deep neural networks (DNNs) for more precise predictions could further optimize the system's responsiveness and efficiency under varying conditions.

B. Dynamic Model Management

Exploring dynamic model management strategies is crucial. This involves developing mechanisms to switch between different ML models based on real-time workload analysis. The system could use simpler models during high-load periods and more complex, accurate models when the workload is lighter, thereby maintaining a balance between accuracy and performance.

C. Effective Accuracy Optimization

A critical focus should be on optimizing the effective accuracy of the system. This means ensuring that the accuracy of ML model responses not only remains high but also aligns with the users' expectations within given deadlines. Future work could explore adaptive accuracy thresholds that dynamically adjust based on the system's current state and user requirements.

VI. Acknowledgment

We appreciate IBM providing us with computing resources like IBM Cloud, Openshift, etc. We also appreciate Prof. Ladan Tahvildari and TAs Ryan Liu and Mingyang Xu for guiding us.

References

[1] D. Ardagna and L. Zhang, *Run-time models for self-managing systems and applications*, ser. Autonomic systems. Basel: Birkhauser, 2010.

[2] M. Calvo and M. Beltrán, "A model for risk-based adaptive security controls," *Computers security*, vol. 115, pp. 102 612–, 2022.

[3] D. G. D. L. Iglesia and D. Weyns, "Mape-k formal templates to rigorously design behaviors for self-adaptive systems," *ACM transactions on autonomous and adaptive systems*, vol. 10, no. 3, pp. 1–31, 2015.

[4] M. M. Xie, T. N. T. N. Goh, and V. V. Kuralmani, *Statistical models and control charts for high-quality processes*, 1st ed. Boston, Massachusetts: Kluwer Academic Publishers, 2002.

[5] F. Ullah, M. Bilal, and S.-K. Yoon, "Intelligent time-series forecasting framework for non-linear dynamic workload and resource prediction in cloud," *Computer networks (Amsterdam, Netherlands : 1999)*, vol. 225, pp. 109 653–, 2023.

[6] Y. Zhu, W. Zhang, Y. Chen, and H. Gao, "A novel approach to workload prediction using attention-based lstm encoder-decoder network in cloud environment," *EURASIP journal on wireless communications and networking*, vol. 2019, no. 1, pp. 1–18, 2019.