Multimedia Tools and Applications (2024) 83:26369–26386
https://doi.org/10.1007/s11042-023-16587-0

ARIMA-PID: container auto scaling based on predictive analysis and control theory

Nisarg S Joshi[1] · Raghav Raghuwanshi[1] · Yash M Agarwal[1] · B Annappa[1] · DN Sachin[1]

Received: 3 August 2022 / Revised: 29 May 2023 / Accepted: 21 August 2023 /
Published online: 30 August 2023

Abstract
Containerization has become a widely popular virtualization mechanism alongside Virtual Machines (VMs) to deploy applications and services in the cloud. Containers form the backbone of the modern architectures around microservices and provide a lightweight virtualization mechanism for IoT and Edge systems. Elasticity is one of the key requirements of modern applications with various constraints ranging from Service Level Agreements (SLA) to optimization of resource utilization, cost management, etc. Auto Scaling is a technique used to attain elasticity by scaling the number of containers or resources. This work introduces a novel mechanism for auto-scaling containers in cloud environments, addressing the key elasticity requirement in modern applications. The proposed mechanism combines predictive analysis using the Auto-Regressive Integrated Moving Average (ARIMA) model and control theory utilizing the Proportional-Integral-Derivative (PID) controller. The major contributions of this work include the development of the ARIMA-PID algorithm for forecasting resource utilization and maintaining desired levels, comparing ARIMA-PID with existing threshold mechanisms, and demonstrating its superior performance in terms of CPU utilization and average response times. Experimental results showcase improvements of approximately 10% in CPU utilization and 30%

Keywords Cloud computing · Virtual machines · Auto-scaling · Containerization

✉ DN Sachin
sachindn.207cs004@nitk.edu.in

Nisarg S Joshi
njnisarg@gmail.com

Raghav Raghuwanshi
raghavraghuwanshi0101@gmail.com

Yash M Agarwal
yashagarwal786@gmail.com

B Annappa
annappa@ieee.org

[1] Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal, Mangaluru, India

1 Introduction

Cloud Computing has been a paradigm shift in how computing resources can be accessed and used. Cloud has enabled small businesses, the research community, and independent developers to try out their work by renting cheap computing resources. One of the key benefits of the cloud has been the capability to provision resources on demand. Cloud providers allow you to provide servers, storage, and network on demand and even scale them elastically [1]. VMs have been the go-to mechanism for deploying applications and services in the cloud. Provisioning a VM and scaling VM instances has become easy nowadays. Another mechanism of virtualization gaining a lot of momentum is containers. Containers are a lightweight virtualization mechanism running inside VMs to provide isolation and packaging of applications within a single VM. Containers can be created and destroyed with lesser overhead and time than VMs. Hence containers are being widely used in deploying applications in the cloud [2].

With the growth of containers, the requirement of elasticity has been more critical than ever. Applications in today's world need to be highly elastic to meet the traffic and user requirements. This requires mechanisms to automatically scale containers as and when the demand arises to maintain the user experience in terms of response times and also to keep SLAs intact. Over the past few years, new work has been done in container auto-scaling, intending to scale containers according to the traffic, demand, and resource requirements [3–5]. Auto-Scaling of VMs has been widely studied and hence is very useful in deriving techniques for container auto-scaling. Still, containers pose a different set of challenges as compared to VMs.

Docker and Kubernetes are some of the leading container technologies [6]. Docker is a containerization tool that is built on top of Linux virtualization abstractions like namespaces and control groups that allow isolation of resources like network, storage, processes, CPU, and so on within a single instance of a VM, allowing potentially running multiple isolated applications. Kubernetes is an orchestration tool used to manage multiple containers in a VM or across VMs in the cloud. Kubernetes provides services like load balancing of requests across containers, spinning up and monitoring the health of containers, scaling mechanisms, etc. These tools are very popular in the current context of cloud and microservices architectures. Every major cloud provider has a managed service for Kubernetes. For example, AWS has Elastic Kubernetes Service (EKS), Azure has Azure Kubernetes Service (AKS), GCP has Google Kubernetes Engine (GKE), and so on. Hence there has been wide adoption of these tools [7]. The presented work utilizes Kubernetes as an underlying orchestration mechanism and implements the auto-scaler on top of Kubernetes architecture.

Artificial Intelligence (AI) plays a crucial role in enhancing container environments in cloud systems through auto-scaling, optimization, dynamic workload management, and intelligent resource provisioning [8]. By leveraging machine learning and predictive analytics, AI algorithms enable intelligent auto-scaling of containerized applications by analyzing historical data and workload patterns to dynamically adjust the number of container instances based on predicted resource requirements. Moreover, AI optimization algorithms optimize resource allocation by exploring different configurations to ensure efficient utilization of container resources, considering factors like cost, performance, and SLA requirements. Additionally, AI-driven dynamic workload management continuously monitors workload characteristics and resource utilization to optimize resource allocation in real-time, ensuring efficient utilization of containers and preventing underutilization or overutilization scenarios. Lastly, AI-based intelligent resource provisioning utilizes historical data, performance

metrics, and workload patterns to predict resource requirements and provision resources proactively, ensuring the availability of resources when needed and enabling smooth operation of containerized applications. Overall, AI empowers container environments in cloud systems by enabling intelligent auto-scaling, optimization, dynamic workload management, and intelligent resource provisioning, leading to improved performance, cost optimization, and efficient management of cloud resources [9].

These studies [10–12] aim to achieve high-level intelligence, accuracy, robustness, and low power consumption, surpassing the capabilities of existing artificial intelligence methods. By integrating these research findings into auto-scaling mechanisms, several benefits can be realized. The use of [10] enhances the adaptability and accuracy of auto-scaling algorithms, enabling precise predictions and efficient scaling decisions in dynamic cloud environments. The adoption of [11] improves the robustness and accuracy of auto-scaling algorithms, enabling effective scaling decisions in scenarios with limited data and evolving workloads. Furthermore, the SAM neuron model [12] with working memory capabilities enhances the understanding of long-term workload patterns and facilitates proactive scaling decisions based on memory-based learning. Neuromorphic computing, with its emphasis on low power consumption and high-speed response, offers valuable contributions to auto-scaling in cloud environments. The scalable digital neuromorphic architecture enables the implementation of sophisticated neural network models, enhancing intelligence and accuracy [13]. The context-dependent learning framework allows for adaptive decision-making based on spike routing, optimizing resource provisioning [14]. The CerebelluMorphic model improves learning and adaptation, leading to dynamic adjustments based on workload characteristics [15]. Incorporating these concepts enhances resource allocation, adaptability, and overall system performance in auto-scaling for cloud applications.

The modern-day applications are highly interactive and dynamic in serving content to the users and engaging them. Such applications often face a varying amount of workload and requests. The usage and consumption of applications have become so widespread that the developers are constantly tuning their applications for minimum downtime and high availability while maintaining performance. Elasticity is the key requirement of such applications. This current work devises a new algorithm that combines two very popular techniques Auto-regressive Integrated Moving Average ARIMA [16] and Proportional Integral Derivative (PID) [17] comes up with an algorithm that can outperform the baseline rule-based models and perform better than individual algorithms. The different metrics used to compare the algorithm with the baselines are user-facing metrics that matter the most, like the response times, the SLA violation rate, and request drop rate. ARIMA, a predictive algorithm, forecasts the resource utilization values to take a scaling decision. PID controller, a feedback-based reactive algorithm, tries to maintain the system at a particular point in resource usage. Combining these algorithms gives the desired tradeoff on the scaling decision trying to neither overshoot nor undershoot the desired number of containers. Also, the combination of algorithms helps combat the tradeoffs of individual algorithms. Major contributions of proposed work include a comprehensive evaluation of different auto-scaling techniques, an in-depth analysis of ARIMA-PID performance, and practical insights into containerization and cloud computing. Experiments show that proposed approach outperforms existing methods in terms of scalability, efficiency, and cost-effectiveness. This work has important implications for various domains such as IoT, Edge systems, and Big Data applications.

The rest of the paper is organized into the following sections. In Section 2 we explore the state of the artworks and literature and review them in brief. In Section 3 we discuss in detail the proposed mechanism and solution. In Section 4 we explain in detail the architectural setup followed by us for this work. In Section 5 we discuss the method we adopted for carrying out

the experiments. In Section 6 we discuss the results of the work. Finally we conclude with Section 7.

2 Related work

There have been quite a few works focusing on container auto-scaling with various techniques for scaling, ranging from Machine Learning (ML) to Mathematical optimization to control theory and Reinforcement learning. Also, these works have different scaling directions like vertical scaling, which increases the number of resources on a single container to scale it up. Horizontal scaling increases or decreases the number of containers and hybrid scaling instances. These works also consider different metrics for evaluation like response time, SLA violation rate, resource utilization, etc. In [18] authors use control theory principles to scale and control the number of containers in response to the real-time traffic and demand. Here the authors employ a horizontal scaling mechanism for the containers. They consider the current number of containers and the average response time of the requests, and based on their algorithm desired number of containers is given as output. This work employs the PID controller approach for scaling containers. The Ziegler-Nichols method [19] holds historical significance as one of the earliest and most widely used methods for tuning PID controllers. This method provides a systematic approach to determining the optimal proportional, integral, and derivative gains of a PID controller by performing step tests and analyzing the system response. Due to its simplicity and practicality, the Ziegler-Nichols method has found extensive application in various industries and has been extensively documented in literature. However, recent advancements in the field of PID controller tuning have brought forth new methods that offer improved performance and robustness compared to traditional methods like Ziegler-Nichols. These advancements primarily include model-based methods and machine learning-based approaches [20, 21].

This work is inspired from this approach and combine it with ARIMA for our scaling purposes. In [22] we see the use of threshold and rule-based methods for horizontal scaling of the containers. The threshold is laid on request count, CPU, and memory usage statistics. In the domain of mathematical optimization, we have [23] which presents an exciting task on scaling of containers as well as VMs that house the containers. The device an optimization function based on integer linear programming problem and solve that every regular interval of time to find the optimal number of containers that need to be running and hence scale them horizontally up or down. [24] presents a policy-based horizontal scaling technique. The work focus on how the policy is set and considers a variety of parameters for decision making. [25] uses a reinforcement learning approach to scale container count. They employ a horizontal scaling technique, model the auto-scaling problem as a Markov decision process and apply Q Learning to find the optimal number of containers over time. In [26] the authors of this work present a new approach to thinking about scaling. They have tried to augment static threshold-based mechanisms with reinforcement learning techniques to predict correct threshold dynamically over time. Thus they use Q Learning to find the threshold values to scale up or down, and then when the condition is met, the containers(and/or VMs) are horizontally scaled up or down in number. [27] employs a combination of reactive and proactive models of auto-scaling and devises a hybrid algorithm to scale up and down the number of container instances in a hybrid cloud environment. [28] presents a predictive analysis approach using time series analysis of the resource utilization using the ARIMA model to predict the load and scale up or down horizontally. [29] Compares two different approaches to auto-scaling.

One uses neural networks to determine whether to increase or decrease the number of existing containers or replicas. Another algorithm used by the authors is Q Learning to predict the action to be taken to scale up or down. Both approaches horizontally scale the resources. ElasticDocker [30] has an interesting take on the elasticity of containers by considering vertical elasticity along with live migration of containers if needed in case the resource demand exceeds the total possible allocation. They follow a static rule and threshold-based approach for vertical scaling. [31] takes into consideration the vertical scaling of both the containers and the underlying VMs.They also adopt a rule and threshold-based approach to scaling the containers and the VM. In [32], the authors define the auto-scaling problem as a multi-objective(fourfold) optimization problem where they consider the vertical and horizontal elasticity of both VMs and containers. Hence this work follows a hybrid approach. [33] also carries out hybrid scaling of the containers and demonstrates 3 approaches for the same using reinforcement learning. The Three different strategies vary from model-free to model-based reinforcement approaches and exploits varying degrees of system knowledge.

3 Proposed mechanism

The proposed work introduces a combined reactive-proactive approach for horizontal container auto-scaling in cloud environments. In horizontal auto-scaling, the number of container instances is dynamically adjusted based on changes in external load to optimize resource utilization. The approach leverages two algorithms: the proactive ARIMA model and the reactive PID controller. The ARIMA model is a proactive algorithm that predicts future resource utilization by analyzing historical data. It anticipates the resource requirements and suggests the desired number of containers accordingly. By forecasting resource utilization ahead of time, the ARIMA model allows for proactive decision-making and avoids sudden spikes or drops in container instances. On the other hand, the reactive PID controller serves as a closed-loop feedback system that reacts to changes in the current resource utilization. It continuously monitors the system and adjusts the number of containers to maintain a specific metric, such as resource utilization, at a fixed value. The PID controller excels at maintaining system stability and gradually converging towards the desired number of containers. To determine the final number of required containers, the outputs of both algorithms are combined using a weighted average. This approach combines the advantages of proactive prediction from the ARIMA model and the stability maintenance of the PID controller. By striking a balance between proactive anticipation and reactive adaptation, the combined approach enables effective and efficient auto-scaling of container instances in response to varying workloads.

In the upcoming sub sections, each algorithm and the final model will be described in greater detail, providing a clearer understanding of how they contribute to the overall auto-scaling mechanism in cloud environments.

3.1 ARIMA model

ARIMA process is a class of dynamic stochastic processes proposed by Box and Jenkins in the 1970s. In theory, ARIMA models are the most general models for forecasting time series data. The ARIMA models require the data distribution to be stationary, and the resource utilization data fits the requirements of the ARIMA model. The ARIMA model forecasts short-term resource demand based on resource usage statistics.

The AR part of the ARIMA stands for Auto Regression. It is a linear regression model which predicts the variable of interest in our case CPU utilization, using a linear combination of past values of the variable. AR model has a parameter called order denoted by p. In the first-order AR model *AR(1)*, At any given moment, the projected variable t is only connected to time periods that are one period apart, i.e. *t-1*. Data from two or three periods apart would be linked in a second or third-order AR process. In the form of an equation, the *AR(p)* model may be extended.

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + ... + \phi_p y_{t-p} \tag{1}$$

Where y_s are the past values and ϕ are the regression parameters. AR model predicts unbiased and favorable results if the past values are not correlated and are independent of each other. However, the resource utilization time series data may have a significant correlation, show trends, and might not be stationary.A stationary time series is one whose characteristics are constant regardless of the time period observed, and whose mean and variance remain constant across time. Ensuring that the time series data is stationary brings us to the second part of the ARIMA model, which is I. The integrator transforms time-series data into a stationary one by computing the differences between the consecutive observations. Differencing removes variance from a time series, eliminates trend seasonality, and consequently stabilizes the mean of the time series. Sometimes a single differencing may not transform a time series into a stationary one, and then the second differencing is computed using the values of the first differencing. This step is repeated until a stationary series is obtained. The number of times the time series data is differenced is called the degree of differencing denoted by d. Differencing is computed mathematically as follows

$$y_t\prime = y_t - y_{t-1} \tag{2}$$

$$y_t\prime\prime = y_t - 2y_{t-1} + y_{t-2} \tag{3}$$

where $y_t\prime$ and $y_t\prime\prime$ are first and second-order differencing respectively. The MA part of the ARIMA model, which stands for Moving Average, works similar to the Auto Regression. Rather than using the past values for prediction, it uses past forecast errors in a regression-like model to predict the variable. Like AR, MA model has a parameter called the order of moving average denoted by q, which refers to the number of past forecast error values that should be considered for prediction. Moving Average model of order q is generalized as *MA(q)*:

$$X_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + ... + \theta_q \epsilon_{t-q} \tag{4}$$

Where μ is the mean, θ are the regression parameters and ϵ are the error term of the series.

The parameters in the *ARIMA(p, q, d)* model need to be fine-tuned to fit the time series data to estimate the future resource demand optimistically. The ARIMA model needs to be trained on the resource utilization data to calculate the parameters. It can be done manually by applying the mathematical models as discussed in [28] which plots the Autocorrelation function *ACF* and Partial Autocorrelation function *PACF* between the time series data points to estimate the value of parameters. Otherwise, the modern implementation of ARIMA performs auto-estimation of the parameters. Based on the resource utilization data, it anticipates the parameters beforehand and dynamically trains the model on those parameters. Every time the resource utilization data changes, it re-estimates the parameters and fits the ARIMA model based on new data points' structure, trends, and behavior. Instead of performing prediction on old ones, it predicts on the new model. Hence the accuracy of the prediction model increases.

As described above ARIMA model can be explained using the above-given equations. To apply ARIMA model to the container auto-scaling problem, we can outline it in the form of algorithm given:

Algorithm 1 Auto Scaling using ARIMA.

```
set values of p , q , d
arima = new ARIMA(p, q, d);
while every interval T do
    y = getCurrentCPUUtilization();
    predUtil = arima.arimaPredict();
    desiredContainers = getDesiredContainers(predUtil);
    currentContainers = getNumContainers();
    arima.arimaUpdate(y);
    if desiredContainers ≥ currentContainers then
        scaleUp(desiredContainers - currentContainers);
    else
        scaleDown(currentContainers-desiredContainers);
    end if
end while
```

We can see in the above algorithm we first set the ARIMA parameters of p,q, and d for the model, which are the key parameters. These parameters are obtained by training the model on different data and then using it on a live system. For every time interval T, we run the ARIMA model, trying to predict the utilization for the next interval using *arima.arimaPredict()* function. Then we obtain desired container count, and based on its value, we either scale up or down the number of container instances.

3.2 PID controller

PID is a closed-loop feedback system that monitors the value of a particular input variable (to be controlled) in the system. It takes action to converge the input variable to the desired setpoint value. PID controls the input variable and maintains it at a steady value over time. The action taken is in form of a control variable *u(t)* whose value is calculated as a function of the error term *e(t)* which is simply the difference between the setpoint *r(t)* and the measured value of input variable *y(t)*. In order to calculate *u(t)* PID controller uses 3 terms namely - Proportional, Integral and Derivative - which represent different components of the error term *e(t)*. PID controller can be summarized with the following equations:

$$e(t) = r(t) - y(t) \tag{5}$$

$$u(t) = K_p * e(t) + K_i * \int_0^t e(t^{'})dt^{'} + K_d * de(t)/dt \tag{6}$$

We can see that *e(t)* is the error component in equation 5, which informs the controller how far the input variable is from the target setpoint *r(t)*. The control equation 6 then explains how the control variable *u(t)* is computed and operated upon the system to minimise error over time and converge the input variable *y(t)* to *r(t)*. The current value of the error *e(t)* is proportional to term P in this model. Considering the gain factor K_p, if the error is substantial and positive, the control output will be equally significant and positive. Because the balanced response requires a mistake to arise, using proportional control alone will result in an error between the setpoint and the actual process value. There is no remedial action if there is no fault. To

obtain the I term, the term I account for the error's initial values and integrates them through time. Consider the case when, after proportional control, there is still a residual error. The integral term in this example seeks to eliminate the residual error by adding a control effect based on the error's cumulative historical value. Once the error is corrected, the integral term will cease expanding. As the error becomes smaller, the proportionate impact will decrease, but the growing integral effect will compensate. Based on the present rate of change, Term D is the strongest predictor of the future trend of the SPPV error.Because it tries to reduce the effect of a mistake by exerting a control influence depending on the rate of error change, it's also known as "anticipatory control". The higher the controlling or damping impact, the faster the change.

A general PID controller can be represented in Fig. 1.

3.2.1 Applying PID to container auto scaling

To apply the Principles of PID controller to the problem of container auto-scaling, we need to model the input and control variables according to our problem statement.

For our work, we chose resource utilization in terms of CPU utilization as our system's input variable, which will be monitored. The desired setpoint $r(t)$ was chosen different values while experimenting. Still, we ended up using $r(t) = 75\%$ for our final results and analysis, but it can be easily tweaked to be any desired value. Hence we aimed at maintaining the CPU utilization of the system at 75% . To maintain this setpoint, we need to pick a control variable that we will change over time to achieve the desired CPU utilization. The obvious choice in our case is the scaling up or down as the action to be performed and the control variable $u(t)$ as the number of containers desired at any given point of time.

One of the most challenging part of working with PID controllers is to arrive at the values of the coefficients K_p, k_d and K_i for the control equation 6. Different approaches, such as the manual method (guess and check) and the Ziegler - Nichols method, are frequently used in literature and industry [34]. However, recent advancements in the field of PID controller tuning have brought forth new methods that offer improved performance and robustness compared to traditional methods like Ziegler-Nichols. These advancements primarily include model-based methods and machine learning-based approaches. These approaches are introduced more sophisticated and effective alternatives. These new methods leverage mathematical models or data-driven techniques to achieve superior control performance and adaptability, making them valuable tools in modern control engineering applications [35]. The coefficients of each phrase are modified manually using trial and error in the manual technique. This necessitates an understanding of the implications that each parameter has on the controller

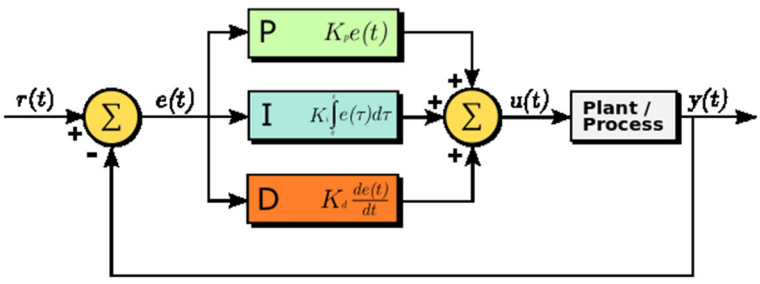

Fig. 1 PID Controller

output. To begin, set the coefficients Ki and Kd to zero, then raise the term Kp until the cycle output begins to oscillate. To lower the steady error, the term K, i should be gradually increased.The term K_d is increased at this point to reduce the oscillations at the cycle output. In this work, we use this manual method to adjust the coefficients. The values that we arrived in our experiments were, $K_p = 0.009$, $K_i = 0.0018$ and $K_d = 0.01125$.

As described above, PID controller can be explained using the above-given equation 6. To apply PID controller to the container auto-scaling problem, we can outline it in the form of the algorithm given.

Algorithm 2 Auto Scaling using PID

```
1: K_p = 0.009;
2: K_i = 0.0018;
3: K_d = 0.01125;
4: pid = new PID(K_p, K_i, K_d);
5: pid.setTarget(setPoint);
6: while every interval T do
7:     y = getCurrentCPUUtilization();
8:     desiredContainers = pid.compute(y);
9:     currentContainers = getNumContainers();
10:    if desiredContainers ≥ currentContainers then
11:        scaleUp(desiredContainers - currentContainers);
12:    else
13:        scaleDown(currentContainers-desiredContainers);
14:    end if
15: end while
```

The algorithm for PID auto-scaling outlined above is simple enough to grasp. We first initialize the controller coefficients to $K_p = 0.009$, $K_i = 0.0018$ and $K_d = 0.01125$. These values are arrived at after the manual guess and check method for PID tuning.

Then we initialized an object called *PID* which is an algorithmic implementation of the given PID equation 6. We also set the desired setpoint using the *pid.setTarget* method. This implementation can be easily found in popular libraries of any language. Using this *pid* object, we take the observed CPU Utilization value and compute the desired container count from the *pid.compute* function every certain interval of time T. Then, based on the value of desired containers, we either scale up or down the number of containers.

3.3 Combined algorithm

This current work devises a new algorithm that combines two very popular techniques - ARIMA and PID - and comes up with an algorithm that can outperform the baseline rule-based models and perform better than individual algorithms. The different metrics used to compare the algorithm with the baselines are user-facing metrics that matter the most, like the response times, the SLA violation rate, and request drop rate. ARIMA, a predictive algorithm, forecasts the resource utilization values to take a scaling decision. PID controller, a feedback-based reactive algorithm, tries to maintain the system at a particular point in resource usage. Combining these algorithms gives the desired tradeoff on the scaling decision trying to neither overshoot nor undershoot the desired number of containers. Also, the combination of algorithms helps combat the tradeoffs of individual algorithms.

Algorithm 3 Auto Scaling using ARIMA-PID.

```
set values of p , q , d
arima = new ARIMA(p, q, d);
K_p = 0.009;
K_i = 0.0018;
K_d = 0.01125;
pid = new PID(K_p, K_i, K_d);
pid.setTarget(setPoint);
initialize w1, w2;
while every interval T do
    y = getCurrentCPUUtilization();
    predUtil = arima.arimaPredict();
    desiredContainers1 = getDesiredContainers(predUtil);
    desiredContainers2 = pid.compute(y);
    desiredContainers = Floor(w1*desiredContainers1 + w2*desiredContainers2)
    currentContainers = getNumContainers();
    arima.arimaUpdate(y);
    if desiredContainers ≥ currentContainers then
        scaleUp(desiredContainers - currentContainers);
    else
        scaleDown(currentContainers-desiredContainers);
    end if
end while
```

From the algorithm, we can see that we take a weighted average of the outputs of both the algorithms previously mentioned in the ARIMA PID model. These weights w1 w2 are set by trying out and experimenting with various loads over time. The best performance can be obtained when we set w2 (for PID) to be 0.2 and w1 (for ARIMA) to be 0.8. We observe that PID's slow converges can hinder the predictions of ARIMA, but at the same time, a small weight to PID ensures stability against short-lived bursty predictions by ARIMA. Hence it works better than individual algorithms.

4 Architecture

This work proposes a Horizontal auto-scaling algorithm. This model is a combined reactive, proactive approach to solve the auto-scaling problem in the servers effectively. More specifically, this work is done for the container auto scaling inside the VMs of the servers. At first, the model tries to learn the pattern of the incoming load via ARIMA and then carries out auto-scaling according to the load prediction. The reactive part of the algorithm tries to auto-scale so that the metric under consideration is kept fixed around a particular value. The proposed mechanism gathers information from various sources to build historical models for predicting future resource utilization. The prediction model uses the ARIMA model to forecast resource utilization based on the collected data. The control system utilizes the PID controller as a closed-loop feedback system to maintain resource utilization at the desired level. It adjusts the number of container instances based on inputs from the prediction model. Lastly, container orchestration manages the deployment and scaling of container instances using inputs from the control system. By combining these components, the mechanism enables efficient and dynamic auto-scaling of containers in response to workload changes.

We use minikube here, which allows us to run the Kubernetes cluster locally. Minikube will enable us to run a single-node Kubernetes cluster. Kubernetes is an open-source container orchestration tool used for managing, deploying, and scaling computer applications.

The overall mechanism is visually represented in Fig. 2, illustrating how the predictive analysis and control theory are combined to enable horizontal auto-scaling of containers in cloud environments. This model consists of four major components. The first component is the Resource Map, which stores the representation and state of the Kubernetes cluster. It allows the autoscaler to interact with the cluster and execute scaling. This component has multiple classes and objects like Deployment, Pod, Service to represent the Kubernetes objects and maintain their state. It also provides functionalities and facilities like pollPodStatus to continuously monitor the health of the pods and getCPURequest to get the capacity of the pods for auto scaler calculations.

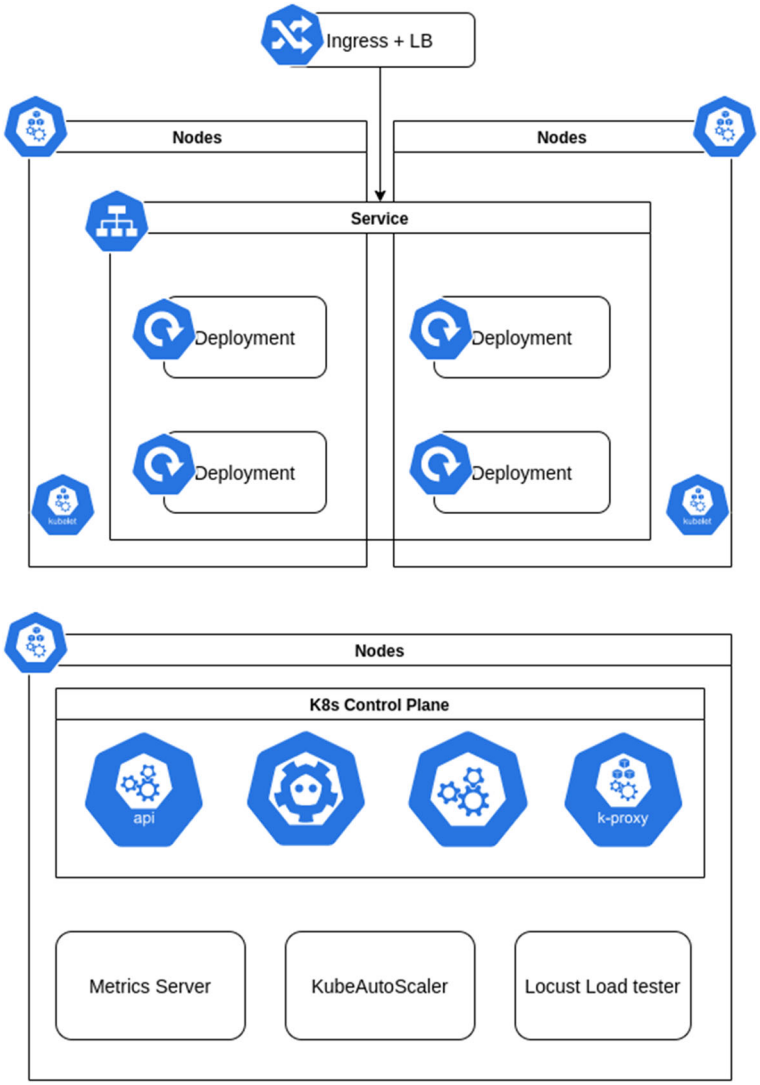

Fig. 2 Model Architecture

The second component is the monitor that talks to the metric server deployed inside the Kubernetes to gather metrics required to calculate the number of pods required, such as CPU utilization and memory utilization.

The third component is the Auto Scaler which is the core component of the algorithm, and the auto-scaling decision is taken here. This component talks to Resource Map and Monitor to gather information about the pods, then the Auto Scaler runs the algorithms and does the scaling up or down action by calling the resource map again.

The fourth and final component of the model is the Load Profile used for load testing the application. The Load Profile does load testing by passing a variable load, and it collects information such as to request per second, response time, etc.

Overall, the proposed model is a combined reactive, proactive approach to solve the auto-scaling problem in the servers effectively. The architecture is designed to learn the pattern of the incoming load via ARIMA and then carry out auto-scaling according to the load prediction. The reactive part of the algorithm tries to auto-scale so that the metric under consideration is kept fixed around a particular value.

5 Experimental evaluation

The algorithm was evaluated by conducting load testing on a test application designed for image processing tasks such as image scaling and rotation. This application was chosen as it represents computationally intensive tasks that generate load similar to real-life scenarios.

To perform the load testing, a load generator was employed. The load generator periodically sends requests to the test application, simulating different levels of load. The load profile depicted in Fig. 3 demonstrates the number of requests per second. The load generator was designed to simulate heavy load scenarios as well as scenarios with fluctuating loads. This allowed for testing the algorithm's scalability and adaptability in varying load conditions, assessing how effectively it can handle scaling up or down.

The experimental system configuration utilized a single instance of a virtual machine (VM) running on minikube, as the focus of the algorithm is on auto-scaling containers. The hardware

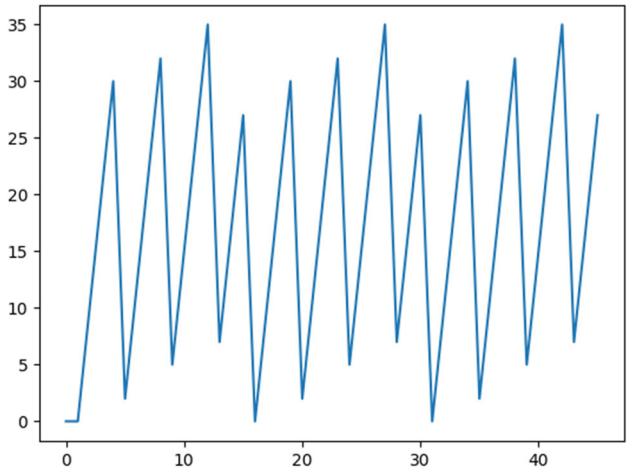

Fig. 3 Load Generator

setup consisted of an Intel i5 6th generation processor. The number of pods, representing individual containers, ranged from 1 to 5, with a restriction to not exceed five due to CPU limitations. Each pod was allocated 300m of CPU, denoted in Kubernetes terminology as "thousandth of a core," which corresponds to 30 percent of a core. This setup allowed for a maximum of five pods, each with a CPU capacity of 300m.

During the load testing process, requests were generated by the load generator, and the load was monitored by the metric server. The collected information was then fed into the mathematical model within the auto scaler component, which made scaling decisions based on the current number of pods. The resource map component implemented the scaling decisions by either adding new pods or removing existing ones.

Metrics collected during the experimentation included CPU utilization, requests per second, number of containers, and memory utilization. These metrics provided insights into the performance and behavior of the algorithm under varying load conditions.

Overall, the load testing was conducted on an image processing test application using a load generator. The experimental setup involved a single VM instance with a specified number of pods, CPU allocation, and resource restrictions. Metrics were collected and analyzed to evaluate the algorithm's auto-scaling performance, with a focus on CPU utilization as the primary metric for scaling decisions.

6 Results and analysis

In this section, we show the cumulative results and analysis. We present three different graphs showing a comparative study of 2 algorithms: 1.) ARIMA-PID and 2.) Horizontal Pod AutoScaler (HPA).

Horizontal Pod AutoScaler is the default auto-scaling algorithm used in Kubernetes for horizontal auto-scaling. It is a basic rule and threshold-based scaling algorithm. It takes in input as an upper threshold and lower threshold of CPU Utilization and scales the number of containers proportionally when the usage hits the threshold. HPA has been widely accepted as a baseline in many research works and is currently an industry standard used by many products and businesses. Hence, comparing our combined ARIMA-PID algorithm performs better would show the better overall performance of our proposed model.

The metrics we have considered to compare finally are:

1. Response Time
2. CPU Utilization
3. Number of Pods/Containers

The results that we obtained after testing the algorithms are shown below:

In the Fig. 4 we can see that the *actual* curve is the measured CPU utilization of the system across all containers. The curve depicting the predicted utilization shows the forecasted CPU prediction by the ARIMA model. The curve representing the threshold-based algorithm shows the allocated CPU based on the number of containers allocated at the given time.

Thus we can see that the threshold-based algorithm clearly over-allocates the number of containers and hence can waste a lot of resources. At the same time, the predicted values are much closer to the actual utilization that is obtained. Therefore the ARIMA model outperforms HPA in this respect.

In the Fig. 5 we can see that the number of containers maintained by the (ARIMA+PID) as compared to HPA is much more stable of overtime since PID does not allow a sudden change in value even if predictions changes overtime. This allows the system to remain stable and

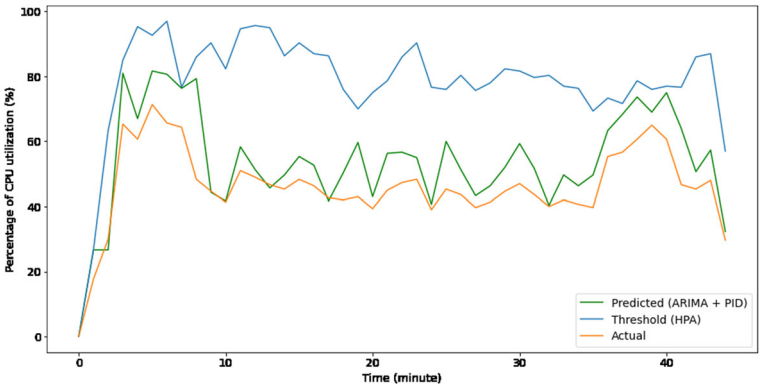

Fig. 4 The variation of CPU utilization over time

gradually lower the number of containers. This can ensure a high degree of stability against the burst of traffic. At the same time, the HPA based auto-scaler shows a bursty and sudden change in scaling activity which can cause a lot of system overhead.

In the Fig. 6 we can see that (ARIMA + PID) model outperforms the HPA based model on average response times. HPA being reactive allows a lot of fluctuation of response times and is more than our proposed model when the load varies drastically over time. Thanks to PID, since the number of containers is stable and scaling activity is controlled, the response time also gradually converges to lower values.

To provide a more statistical comparison between HPA and ARIMA-PID, we can see that in the case of average CPU utilization, HPA got an average utilization across all time to be 83.62%. In contrast, ARIMA-PID achieved a much lower utilization of 73.40%, an improvement of 10.22% over the baseline.

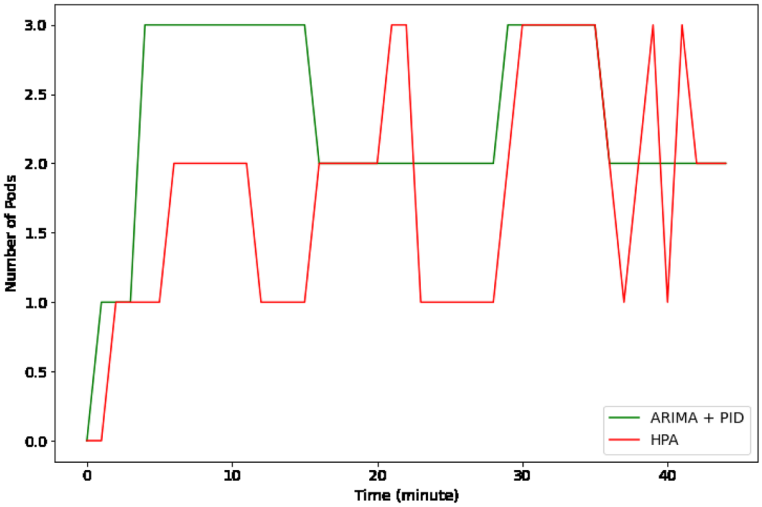

Fig. 5 The variation of number of pods over time

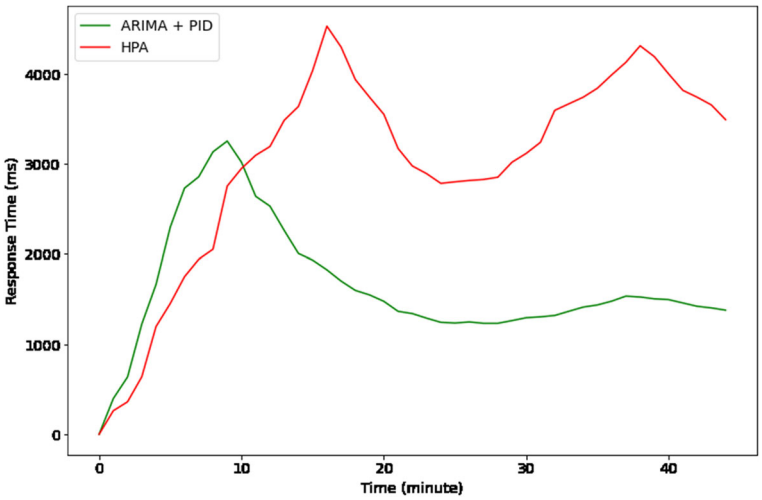

Fig. 6 The variation of response times

In the case of response times, also we see a drastic improvement in the case of ARIMA-PID. The average response time overall times for ARIMA-PID is 1934.1 ms, and HPA is around 2796.2 ms. This shows an improvement of 30.83%. The summary of these results have been shown in Table 1

The above results show that the proposed ARIMA-PID algorithm outperforms the HPA algorithm in terms of resource utilization and response time. The ARIMA-PID algorithm achieves a higher resource utilization rate while maintaining a lower response time compared to the HPA algorithm. This is because the ARIMA-PID algorithm uses predictive analysis to anticipate future resource requirements and control theory to maintain resource utilization levels at the desired value. In terms of cost-effectiveness, both algorithms perform similarly. However, the ARIMA-PID algorithm has an advantage over HPA in scenarios where there are sudden spikes in workload or unpredictable changes in external load. This is because the ARIMA-PID algorithm can quickly adjust the number of container instances based on forecasted requirements, while HPA may take some time to react to changes in external load.

Overall, the interpretation of results suggests that the proposed mechanism for auto-scaling containers based on predictive analysis and control theory can improve resource utilization and response time compared to traditional auto-scaling techniques such as HPA. However, further research is needed to evaluate the performance of this mechanism under different scenarios and workloads.

Table 1 Comparison Summary

Model/Metric	ARIMA-PID	Threshold	Improvement
Response Time	1934.1 (ms)	2796.2 (ms)	30.83%
Response Time	1934.1 (ms)	2796.2 (ms)	30.83%
CPU Utilization	73.40 %	83.62%	10.22%

7 Conclusion

Our proposed solution can perform better on several key metrics: user-facing like response times and system facing like CPU utilization and the number of containers. (ARIMA + PID) leverage the strengths of each other and overshadow the weakness. With its accurate predictions, ARIMA can give an exact requirement of containers. Still, it can be misleading in a drastically varying external load since it can lead to more scaling activity than needed. PID adds stability to the system since it allows slow changes, which can be useful in bursty traffic scenarios as considered in our case. From the results in Table 1 we can see that ARIMA-PID can give an improvement of about 30% in response times of applications and a reduction of up to 10 % in the CPU utilization of the system. Thus combined, both the algorithms prove to perform well against the baseline mentioned. However, there are some limitations to our work. First, our experiments were conducted on a limited set of applications and may not generalize to other scenarios. Second, our approach assumes that the workload is predictable and stationary, which may not hold true in dynamic environments. Third, our implementation relies on certain assumptions about the underlying infrastructure and may not be applicable to all cloud providers or architectures.

Future research can explore several directions such as improving the accuracy of workload prediction models, incorporating machine learning techniques for adaptive auto-scaling, evaluating different control strategies for resource allocation, and investigating the impact of network latency on container performance. Our contributions include a comprehensive evaluation of different auto-scaling techniques, an in-depth analysis of ARIMA-PID performance, and practical insights into containerization and cloud computing. This work has important implications for various domains such as IoT, Edge systems, and Big Data applications.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Informed consent Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

References

1. Pal D, Chakraborty S, Nag A (2015) Cloud computing: A paradigm shift in it infrastructure. CSI Communications, January
2. da Silva VG, Kirikova M, Alksnis G (2018) Containers for virtualization: An overview. Appl Comput Syst 23(1):21–27
3. Sheganaku G, Schulte S, Waibel P, Weber I (2023) Cost-efficient auto-scaling of container-based elastic processes. Futur Gener Comput Syst 138:296–312
4. Rabiu S, Yong CH, Mohamad SMS (2022) A cloud-based container microservices: A review on load-balancing and auto-scaling issues. Int J Data Sci 3(2):80–92
5. Chouliaras S, Sotiriadis S (2022) Auto-scaling containerized cloud applications: A workload-driven approach. Simul Model Pract Theory 121:102654
6. Ganne A (2022) Cloud data security methods: Kubernetes vs docker swarm. Int Res J Mod Eng Technol 4(11):
7. Berton L (2023) Ansible for kubernetes cloud providers. In: Ansible for Kubernetes by Example, pp. 239–260
8. Jorge-Martinez D, Butt SA, Onyema EM, Chakraborty C, Shaheen Q, De-La-Hoz-Franco E, Ariza-Colpas P (2021) Artificial intelligence-based kubernetes container for scheduling nodes of energy composition. International Journal of System Assurance Engineering and Management, 1–9

9. Schuler L, Jamil S, Kühl N (2021) Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 804–811. IEEE
10. Yang S, Tan J, Chen B (2022) Robust spike-based continual meta-learning improved by restricted minimum error entropy criterion. Entropy 24(4):455
11. Yang S, Linares-Barranco B, Chen B (2022) Heterogeneous ensemble-based spike-driven few-shot online learning. Front Neurosci 16
12. Yang S, Gao T, Wang J, Deng B, Azghadi MR, Lei T, Linares-Barranco B (2022) Sam: a unified self-adaptive multicompartmental spiking neuron model for learning with working memory. Front Neurosci 16
13. Yang S, Deng B, Wang J, Li H, Lu M, Che Y, Wei X, Loparo KA (2019) Scalable digital neuromorphic architecture for large-scale biophysically meaningful neural network with multi-compartment neurons. IEEE Trans Neural Networks Learn Syst 31(1):148–162
14. Yang S, Wang J, Deng B, Azghadi MR, Linares-Barranco B (2021) Neuromorphic context-dependent learning framework with fault-tolerant spike routing. IEEE Trans Neural Networks Learn Syst 33(12):7126–7140
15. Yang S, Wang J, Zhang N, Deng B, Pang Y, Azghadi MR (2021) Cerebellumorphic: large-scale neuromorphic model and architecture for supervised motor learning. IEEE Trans Neural Networks Learn Syst 33(9):4398–4412
16. Imdoukh M, Ahmad I, Alfailakawi MG (2020) Machine learning-based auto-scaling for containerized applications. Neural Comput and Appl 32:9745–9760
17. Willis M (1999) Proportional-integral-derivative control. Dept. of Chemical and Process Engineering University of Newcastle
18. de Abranches MC, Solis P (2016) An algorithm based on response time and traffic demands to scale containers on a cloud computing system. In: 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), pp. 343–350. IEEE
19. Hang CC, Åström KJ, Ho WK (1991) Refinements of the ziegler–nichols tuning formula. In: IEE Proceedings D (Control Theory and Applications), vol. 138, pp. 111–118. IET
20. Wang X-S, Cheng Y-H, Wei S (2007) A proposal of adaptive pid controller based on reinforcement learning. J China Univ Min Technol 17(1):40–44
21. Li G-D, Masuda S, Yamaguchi D, Nagai M (2009) The optimal gnn-pid control system using particle swarm optimization algorithm. International Journal of Innovative Computing, Information and Control 5(10):3457–3469
22. Kukade PP, Kale G (2015) Auto-scaling of micro-services using containerization. Int J Sci Res (IJSR) 4(9):1960–1963
23. Nardelli M (2017) Elastic allocation of docker containers in cloud environments. In: ZEUS, pp. 59–66
24. Zhang F, Tang X, Li X, Khan SU, Li Z (2019) Quantifying cloud elasticity with container-based autoscaling. Futur Gener Comput Syst 98:672–681
25. Somma G, Ayimba C, Casari P, Romano SP, Mancuso V (2020) When less is more: Core-restricted container provisioning for serverless computing. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 1153–1159. IEEE
26. Horovitz S, Arian Y (2018) Efficient cloud auto-scaling with sla objective using q-learning. In: 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 85–92. IEEE
27. Li Y, Xia Y (2016) Auto-scaling web applications in hybrid cloud based on docker. In: 2016 5th International Conference on Computer Science and Network Technology (ICCSNT), pp. 75–79. IEEE
28. Meng Y, Rao R, Zhang X, Hong P (2016) Crupa: A container resource utilization prediction algorithm for auto-scaling based on time series analysis. In: 2016 International Conference on Progress in Informatics and Computing (PIC), pp. 468–472. IEEE
29. Sangpetch A, Sangpetch O, Juangmarisakul N, Warodom S (2017) Thoth: Automatic resource management with machine learning for container-based cloud platform. In: CLOSER, pp. 75–83
30. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P (2017) Autonomic vertical elasticity of docker containers with elasticdocker. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 472–479. IEEE
31. Al-Dhuraibi Y, Zalila F, Djarallah N, Merle P (2018) Coordinating vertical elasticity of both containers and virtual machines. In: CLOSER 2018-8th International Conference on Cloud Computing and Services Science
32. Hoenisch P, Weber I, Schulte S, Zhu L, Fekete A (2015) Four-fold auto-scaling on a contemporary deployment platform using docker containers. In: International Conference on Service-Oriented Computing, pp. 316–323. Springer

33. Rossi F, Nardelli M, Cardellini V (2019) Horizontal and vertical scaling of container-based applications using reinforcement learning. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 329–338. IEEE
34. Ziegler J, Nichols N (1993) Optimum settings for automatic controllers
35. Sun L, You F (2021) Machine learning and data-driven techniques for the control of smart power generation systems: An uncertainty handling perspective. Engineering 7(9):1239–1247