Journal of Network and Systems Management (2023) 31:35
https://doi.org/10.1007/s10922-023-09726-3

Minimizing Resource Allocation for Cloud-Native Microservices

Roland Erdei[1] · Laszlo Toka[1,2,3]

Received: 23 June 2022 / Revised: 27 June 2022 / Accepted: 24 January 2023 /
Published online: 9 February 2023
© The Author(s) 2023

Abstract

With the continuous progress of cloud computing, many microservices and complex multi-component applications arise for which resource planning is a great challenge. For example, when it comes to data-intensive cloud-native applications, the tenant might be eager to provision cloud resources in an economical manner while ensuring that the application performance meets the requirements in terms of data throughput. However, due to the complexity of the interplay between the building blocks, adequately setting resource limits of the components separately for various data rates is nearly impossible. In this paper, we propose a comprehensive approach that consists of measuring the resource footprint and data throughput performance of such a microservices-based application, analyzing the measurement results by data mining techniques, and finally formulating an optimization problem that aims to minimize the allocated resources given the performance constraints. We illustrate the benefits of the proposed approach on Cortex, an extension to Prometheus for storing monitored metrics data. The data-intensive nature of this illustrative example stems from real-time monitoring of metrics exposed by a multitude of applications running in a data center and the continuous analysis performed on the collected data that can be fetched from Cortex. We present Cortex's performance vs resource footprint trade-off, and then we build regression models to predict the microservices' resource consumption and draw a mathematical programming formulation to optimize the most important configuration parameters. Our most important finding is the linear relationship between resource consumption and application performance, which allows for applying linear regression and linear programming models. After the optimization, we compare our results to Cortex's recommendation, leading to a CPU reservation reduced by 50–80%.

Keywords Cloud-native · Microservices · Performance · Resource footprint · Optimization

✉ Laszlo Toka
toka.laszlo@vik.bme.hu

Extended author information available on the last page of the article

1 Introduction

Cloud computing is widely used in the digital industry as a technology that enables cheap and easy deployment not only for online web services but also for big data processing, machine learning, and storing data for the long term. The cloud is backed by physical data centers worldwide that host virtual machines offered to customers and users. The cloud concept empowers users to replace their hardware with cloud server instances and creates a new economic model where the customer pays only for the usage and not for the entire hardware itself. However, decision-making in cloud environments can be complex due to the diversity in pricing models and service offerings. There are no rules of thumb as each customer could have a specific set of constraints and requirements of their cloud application when it comes to selecting the perfect cloud environment [1].

Microservices have recently gained striking popularity due to being highly maintainable and easy to develop. Microservices-based applications allow the deployment of each microservice (or component) to be in physically separated virtual machines if needed. Microservices, being separately manageable, have a massive advantage in scaling, which is one of the most critical features of the cloud context. Instead of launching multiple instances of a whole application, there is a possibility to only scale in or scale out a specific microservice component [2]. Hence creating microservices with the help of container technologies can result in robust and easy deployment with a small footprint. However, the goal is not only to create a small footprint application, but also to match the Quality of Service (QoS) of the original monolith performance-wise. When it comes to data-intensive applications, data throughput is the most crucial aspect. The components of the application communicate with each other via APIs. Wrongly configuring a microservice component where the data throughput is consequently low can result in a bottleneck in terms of the whole application's performance. Scaling the bottleneck components until the QoS is met will mitigate the issue. The problem we tackle is how to correctly configure the resource provisioning of each and every microservice component so that the overall application performs as required.

To create a resource-efficient microservice, the most critical settings to tune are the CPU and memory limits that control the resource usage of a microservice instance. These limits can prevent components from using more resources than needed, and with the help of these limits, the app provider or operator makes sure in a private cloud that concurrent applications use the cluster as efficiently as possible, whereas, in a public cloud, limits help to pre-estimate costs. Setting the limits for several components at once raises a complex optimization problem when it comes to microservices. Application components have their tasks while working closely with other components, and if one of the components lags, the whole microservices-based application will suffer. On the other hand, resource provisioning must consider the shared resources: over-provisioning one component can cause starvation at other components. Furthermore, each component may have several configuration parameters which need to be tested in terms of resource usage to find the optimal limit values making the specific component work as efficiently as possible.

In this paper, we tackle this exact challenge: we propose an optimization framework that estimates resource usage vs performance of microservices-based, data-intensive applications with the help of machine learning, then minimizes the allocated resources given the QoS constraints. In our prior work [3] we tackled the optimization of CPU provisioning. The current manuscript has been extended with our approach for memory optimization, which is of paramount importance when it comes to cloud cost estimation.

The rest of the paper is organized as follows. Section 2 presents the related work, and in Sect. 3 we propose the methodology to solve the problem at hand. In Sect. 4 we introduce an illustrative example for microservice-based applications, Cortex, then we train regression models on the performance and resource consumption measurements data and create a linear programming formula for resource optimization, completed with numerical analysis. Section 5 concludes our findings.

2 Related Work

Performance modeling of microservices-based applications allows us to determine the capacity distribution among each microservice. This enables planning for applications and the detection of the bottleneck in microservices. [4] proposes to apply statistical models, e.g., Theil-Sen estimator or Support Vector Regression, for this purpose. After analyzing the data acquired by their approach on the example applications, it was identified that the microservices follow a typical performance versus workload relationship pattern, which suggests the performance degrading with the increase of workload up until a certain point when all the virtual resources are used. In all tests, the CPU utilization increased linearly with the number of requests sent to the microservices. The approach was tested on several test applications, including a compute-intensive application, a database accessing an application, and a web accessing application. Similarly to [4] we also want to create a model to detect and avoid bottlenecks; however, we want to provision the resources individually for each microservice using as minimal resources as possible.

Zhang et al. [5] presented Sinan, a machine learning- driven resource manager for microservice-based applications. Sinan presents the challenges of managing complex microservices and leverages a set of scalable and validated models to reduce resource usage while meeting the end-to-end QoS. Sinan trains two models with the traces: a CNN (Convolutional Neural Network) model for short-term performance prediction and a Boosted Trees model to evaluate long-term performance evolution. Combining the two models allows Sinan to be effective in both near-future and distant future resource management. Similarly to [5], our goal is to meet the QoS requirements with an analytical model which can predict the expected average resource usage.

The detection of a bottleneck component is important if we want to increase the performance of a complex application deployed in the cloud. [6] presents an analytical model that can detect bottlenecks and predict the performance of a multi-tier application. The suggested approach consists of two resource provisioning steps: a predictive one for long-term scales and a reactive one for short timescales.

Long-term prediction is useful when the load can be predicted, e.g., a typical daily pattern, but only reactive provisioning can handle unexpected high workloads. Using both and combining them can create an effective provisioning scheme. With this approach in a scenario where the workload of a three-tier application has been doubled, the technique showcased in [6] was able to double the application capacity within five minutes while maintaining the QoS targets. Unlike [6], our focus is to predict and create resource limits for the average resource usage of microservices of any cloud-native application.

Our model shows similarities to CostHat [7], an approach to model the deployment costs, including compute and IO costs. CostHat is a graph-based model of the deployment costs and can be used for applications implemented on top of AWS Lambda. Just like our results, [7] demonstrates that between the used features, there are linear dependencies. Similarly to [7] we focus on reducing costs if the model is used in a public cloud environment. However, our model calculates CPU and memory usage as output, which is extremely useful for private clouds.

Leitner et al. [7] presented a cost model to calculate and optimize the deployment cost of microservices. Their model takes into account several characteristics, e.g., the connection between services, the processing capacity and expected load. Results show that the proposed algorithm can calculate the cost in real-time, however the cost model does not consider the adaptation of the application: it is designed only for static application analysis. Ma et al. in [8] introduced a task scheduling cost optimization framework for cloud IoT applications, where they consider cost and deadline parameters. Their algorithm achieves state-of-the-art performance in simulations, on the other hand the main focus of the paper is on scheduling tasks to a given infrastructure, and does not consider scaling of the application. Authors of [9] proposed a reinforcement learning (RL)-based solution for horizontal and vertical resource provisioning. They use a cost function which considers the cost of scaling, running the application and violating SLA, and the agents choose their actions to minimize this cost. With enough training data and time, RL agents are able to find the optimal resource provisioning action in a given state. However, the state on which the agents are trained contains only the current resource and system information, therefore the agent can hardly understand the long-term effect of its actions. Authors of [10–12] developed cost models and used them for cost-aware resource provisioning in edge-cloud environment [10] and for optimizing network functions in a telco environment [11, 12]. The model in [10] also considers the load prediction for the next time interval, however the applicability of the models for long-term traffic or resource allocation are not discussed. Indeed, workload prediction is being widely explored to solve issues such as resource under-utilization, load balancing and power consumption, using time series analysis regression and neural networks based models. The time series analysis based models are unable to capture the dynamics in the workload behavior whereas neural network based models offer better accuracy on the cost of high training time. [13] presents a workload prediction model based on extreme learning machines whose learning time is very low and forecasts the workload more accurately. The performance of the model is evaluated over two real world cloud server workloads i.e. CPU and Memory demand traces of Google cluster and compared with predictive models based on state-of-art techniques. It is observed that the

proposed model outperforms the state-of-art techniques by reducing the mean prediction error up to 100% and 99% on CPU and memory request traces respectively. Comparing with the aforementioned works, our method achieves both short-term and long-term resource optimization considering the cost of running and scaling the cloud native application according to the load.

3 Methodology and Model

Our goal is to minimize the CPU limit provisioned for each component in a microservice-based application for a static load. To this end, we formulate mathematical programming problems with user-defined constraints. Then the result of the optimization yields the expected CPU usage based on which the resource limits of each microservice can be appropriately set. In order to formulate the optimization model, first, we need to find the most important features of the operation of the microservices, as well as their coefficients. Our approach is to build a regression model from a measurement dataset that incorporates the inference between configuration settings, application performance and resource footprint. Figure 1 shows the steps of our proposed approach.

The measurement dataset contains the monitored microservices' resource usage. Assuming a data-intensive application at hand, there are typically two paths that must be monitored: the writing path (data ingestion) and the reading path (data retrieval). The cloud-native application is typically constituted by various

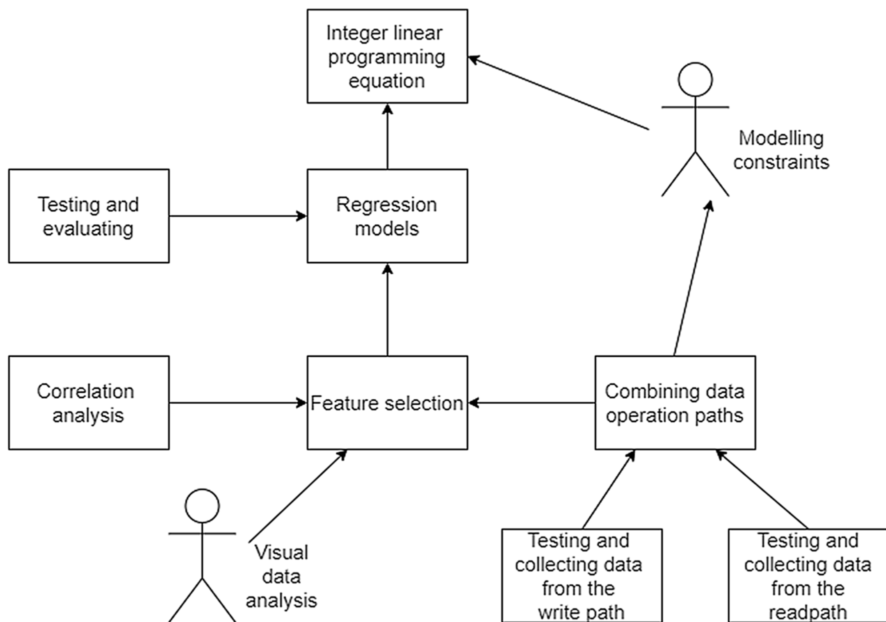

Fig. 1 Methodology diagram for the thought process

microservices which are closely coupled together. These microservices (or components) can be customized with configuration parameters and settings. While measuring the effects of these settings and scaling in and out the components, a data collector pipeline stores the microservices' resource usage and performance indicators. The important features that are later used in the data mining step can be, for example, the replica number of each component, the number of incoming service requests, or the number of served queries per second. After measuring several scenarios with various settings and analyzing the key differences between each setup, we propose to train a regression model.

We create a separate regression model for every main component of the application. Feature selection is important for keeping the model simple. Most of the potential features show linear dependencies with the resource usage, as pointed out in [4] and [7], therefore in this paper, we make a case for linear relationships so that we can later formulate the optimization with linear programming. By keeping only the highly correlating features, the linear programming solvers can find the solution quickly even if some variables are integers, turning the problem into non-deterministic polynomial-time (NP) complexity [14]. After training the regression model on the measurement dataset, given the coefficients of the regression, we can continue with linear programming. Note that in case the features exhibit a non-linear effect on the resource usage or on the application performance, then polynomial regression might be used, but then the respective optimization problem will not be a linear programming instance.

The decision variables are the important features from the regression model. Using these variables, one must create the algebraic expression which describes the target function; in our case, we strive to minimize the CPU resource usage. As the regression model is trained for precisely that dependent variable, the target function is produced by the regression formula: features as variables, their coefficients and the intercept. Based on the measurement data, one can create constraints on the features to be taken into account in the optimization problem. Typical constraints include but are not limited to high availability requirements on microservice instances, maximal memory consumption, application performance in terms of data throughput or rate of requests served. In general, it is beneficial to minimize the search space as much as possible, resulting in faster optimization. Non-negativity constraints can be applied for most of the features.

With the optimization model formulated, any solver can optimize the problem with the given user inputs and suggest the best possible setup for all the features. In the next section, we illustrate how this approach can save resources for the operation of a cloud-native microservices-based application.

4 Illustrative Example of Cortex

Cortex [15] is a horizontally scalable, multi-tenant, long term storage for Prometheus [16] written in Go. It is a Cloud Native Computing Foundation (CNCF) project [17]. Cortex can receive metrics from multiple Prometheus servers and serve aggregated queries across all data in a single place which is useful when

the cloud application operator wants to get an overview of all the metrics and data collected by these Prometheus servers. Our long-term storage choice was MinIO [18]. Figure 2 illustrates our setup with Cortex components.

Cortex consists of multiple horizontally scalable microservices from which we introduce the most important ones in terms of performance and resource footprint in this section. Figure 2 contains most of Cortex's components, including the external applications that vary on the use case. Writing path is the microservice's data ingestion process where the collected and gathered data gets stored in the long-term storage following pre-processing. On the other hand, the reading path is the process when the data gets retrieved from long-term storage. Some components are vital components for both paths. When it comes to the writing path, the key components are the Distributor and Ingester. The Ingesters have a similarly important role regarding the reading path. In addition to the Ingester and Distributor, we introduce the Querier, Query frontend (optional in deployment), Ingester, and the Store-gateway.

The Distributor microservice is responsible for handling the incoming samples of Prometheus. It is the first step in the writing path. When the Distributor receives samples from the Prometheus servers, each piece is checked and validated for correctness and to make sure that it is within the preconfigured tenant limits and falling back to preconfigured ones in case limits have not been overridden for the specific tenant. The valid samples are then split into batches and sent to multiple Ingesters in parallel by the Distributor.

The Ingester microservice is responsible for writing incoming time series to a long-term storage backend (in our case, into MinIO) on the writing path and returning samples stored in-memory for queries on the reading path.

The Querier microservice handles queries using the PromQL query language. The Queriers fetch samples both from the Ingesters and long-term storage

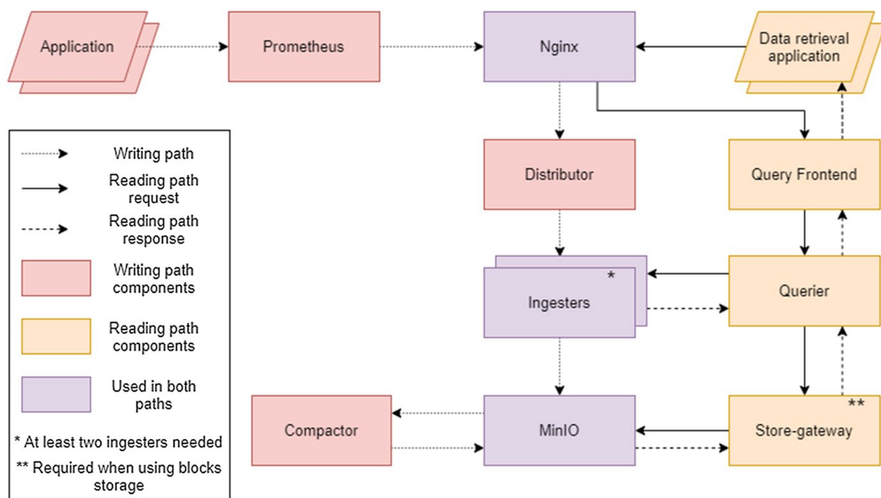

Fig. 2 Cortex's architecture including the external applications [15]

solution: the Ingesters hold the in-memory series which have not yet been written to the long-term storage.

The Query frontend is an optional microservice implementing Querier's API endpoints and used to accelerate the reading path. If the Query frontend is deployed, incoming query requests should be directed to a Query frontend rather than straight to the Queriers. The Querier microservice will still be required within the cluster for executing the queries. The Frontend does not query the necessary data. The Query frontend internally performs query adjustments and holds queries in an internal queue similarly to a database. The Store-gateway is the Cortex microservice responsible for query series from blocks; each block is composed of chunk files containing the timestamp-value pairs for multiple series and an index, which indexes the metric names and the labels.

4.1 Cortex Measurements

We deployed Cortex in a Kubernetes [19] cluster and monitored the CPU usage of each component. We ran data writing measurement experiments for various configuration scenarios for a couple of hours each in order to measure the whole pipeline's CPU usage. Some components are only active at the beginning or at the very end of the pipeline. Also, we tested query performance multiple times and created a data retrieval application to perform different queries to avoid serving data from the cache. Based on our experiments, we selected the most important parameters of the Cortex microservices in order to find the features which accurately infer total resource usage. Our target function is the overall CPU usage, including every component of Cortex and Prometheus.

During our data ingestion experiments, we wrote several blocks into MinIO to make sure we tested the whole data write pipeline numerous times for all tested settings. We generated 60,000–300,000 time series to be ingested and deployed 1–5 replicas of each microservice on the write path of Cortex. We found the number of time series continuously written into Cortex and the number of Nginx [20], Distributor and Ingester components to be the most important features affecting CPU consumption. These components contribute majorly to the write path performance of Cortex; scaling out other components does not change the CPU usage or the ingestion capacity. Figure 3 shows the relationship between the CPU usage and the number of Ingesters used, while the coloring indicates the number of time series ingested by Cortex. The histogram of measurements, distinguished again by colors for various numbers of time series ingested by Cortex, against the average CPU usage, can be seen in Fig. 4.

For testing the query performance, we fetched 60,000–300,000 time series with various time aggregation levels (between 10 s and 1 h) from Cortex. With every test, we queried the last 4 h of data from Cortex while continuously writing data to ensure we always have data both in the long term storage and in the Ingesters' memory. The requests were sent to Nginx ingress; it forwarded those to the Query frontend, which could optionally transform a query into smaller queryset batches and send it to the Querier that performed the data retrieval and aggregation. From

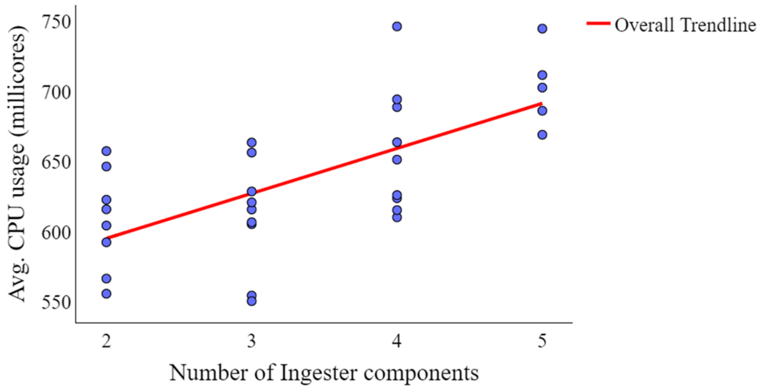

Fig. 3 Write path CPU usage vs. number of Ingesters used with the various numbers of time series stored by Cortex

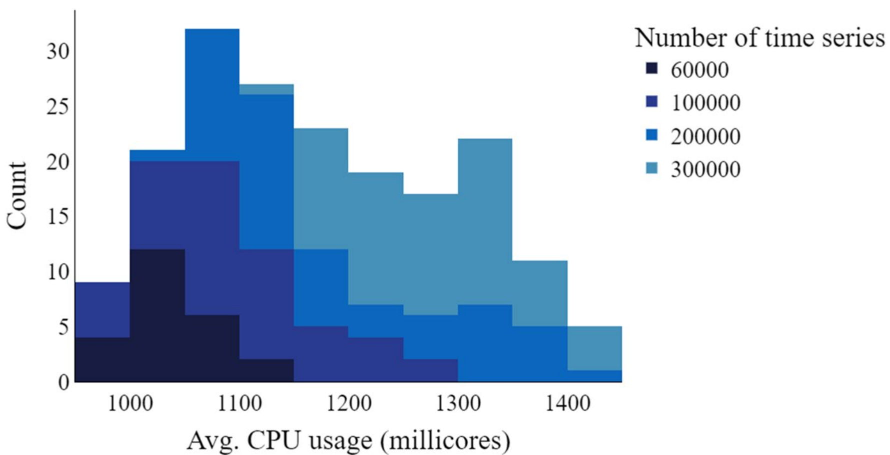

Fig. 4 Histogram of measurements for the average CPU used. The number of time series ingested by Cortex is illustrated by coloring

the read (query) path, we found the number of time series queried in Cortex and the aggregation (or step as in Prometheus' nomenclature) to be the important features. Figure 5 presents the measured CPU usage of the microservices and the level of data aggregation in seconds.

The correlation matrix shown in Fig. 6 shows the most dominant features of both paths. The write path model includes the number of ingested time series and the number of required components of Nginx, Distributors and Ingesters. The read path model uses the number of time series and the aggregation used while querying the data stored in MinIO. The conclusion from the measurements is that the resource footprint of the data ingestion pipeline of Cortex is mainly affected by the number of time series stored and later queried. Using several Ingester instances increases the capacity of the write path but also induces higher resource consumption: this

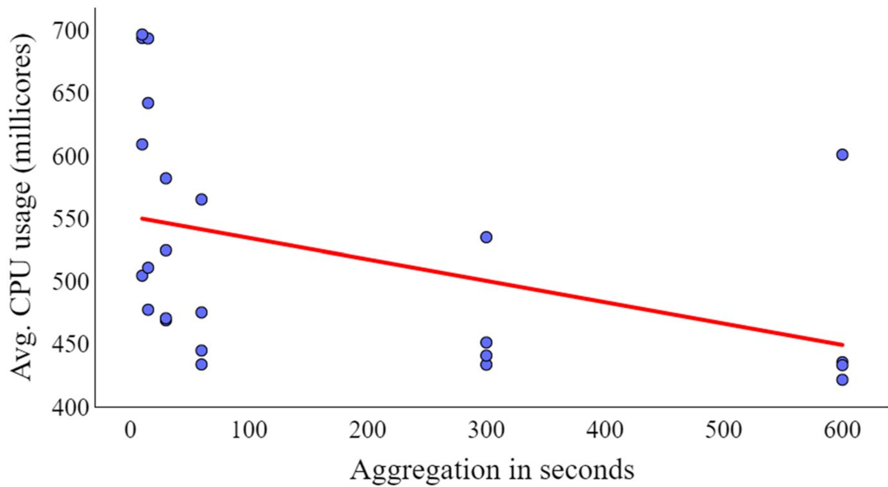

Fig. 5 Read path CPU usage vs. level of data aggregation in seconds

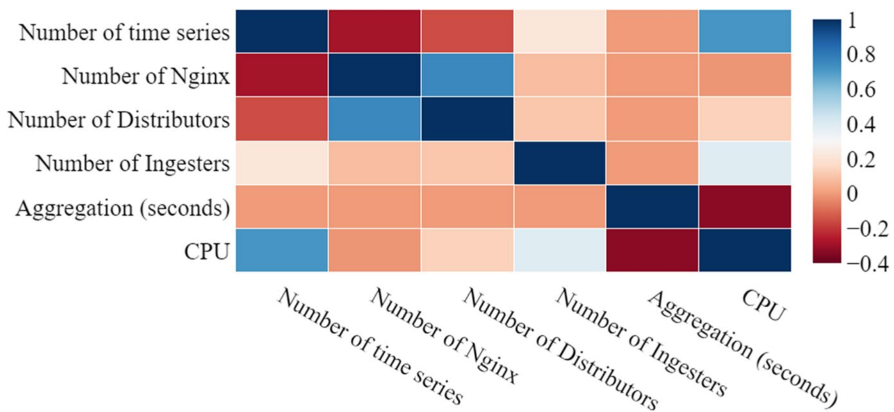

Fig. 6 The two paths' correlation matrix

component stores the metrics and time series before those get written into the long term storage, e.g., MinIO. On the read path, higher data aggregation leads to less CPU usage.

4.2 Linear Regression Models

After the measurements had been performed, we created different regression models for the CPU usage of each component and trained them. We used linear regression models because the chosen features showed linear dependencies with the target-dependent variable, i.e., CPU consumption. The models are trained on data from the two paths: write path and read (query) path for each microservice, as a

Table 1 Regression coefficients

Features	Coefficients
Number of time series	9.28
Number of distributors	32.54
Number of ingesters	24.21
Number of Nginx	1.91
Query aggregation	− 0.182
Intercept	129.3

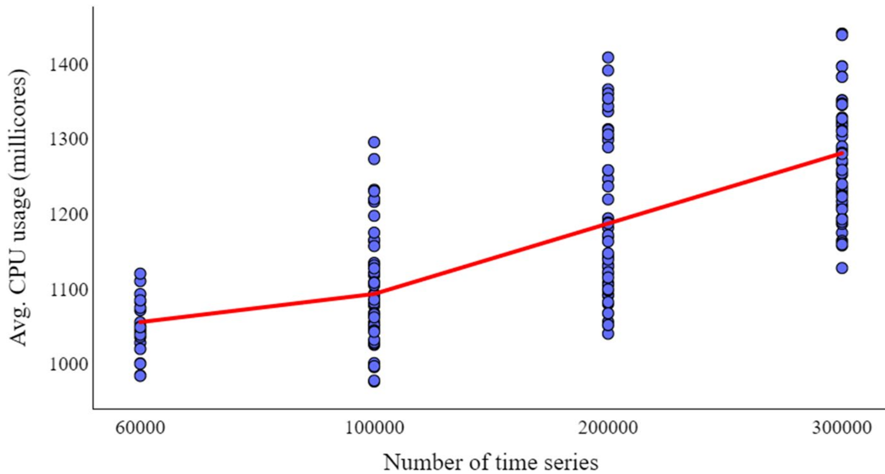

Fig. 7 The two paths combined CPU usage by the number of time series used by Cortex

component might be used continuously by both paths, so creating an aggregate model which takes features and information about the different paths at the same time is important.

Table 1 shows the coefficients of the features used in the model. The performance of the regression model is best reflected by R2—the coefficient of determination—which was around 0.75. The Mean Absolute Error (MAE) is around 58.2 [millicores]. For bringing the coefficients to the same order of magnitude, we divided the number of time series by 10,000. With this data transforming step, the coefficient (9.28) is for every 10,000 time series used. Figure 7 shows all the data points used for this model: the overall CPU consumption against the stored and queried number of time series.

4.3 Integer Linear Programming for CPU Optimization

With a complex regression model including both the read and write paths in our hand, we created an integer linear programming (ILP) optimization for the

application with specific constraints. Finally, we configured the microservices based on the results of the optimization.

We minimize CPU usage for the user inputs, i.e., the number of time series used in the system, the available memory for the microservices, and the data aggregation given in seconds for the query part. The output is the average CPU utilization expected by components, the number of Ingesters, Nginx, and Distributors suggested to use, and whether the given memory is enough for the microservices.

We use the following notation:

- T = number of time series used [10,000]
- I = number of ingester instances
- D = number of distributor instances
- N = number of Nginx instances
- Q = level of data aggregation [s]
- M = memory resource available [MB]

The ILP minimization target is:

$$9.28T + 32.54D + 24.21I + 1.91N - 0.182Q, \tag{1}$$

subject to

$$I > 1 \tag{2}$$

$$D > 1 \tag{3}$$

$$N > 1 \tag{4}$$

$$T > 0 \tag{5}$$

$$Q > 0 \tag{6}$$

$$1500 + 128N + 512D + 1500I + 1000T/10 < M \tag{7}$$

$$I - T/10 > 0. \tag{8}$$

The objective function (1) stands to minimize the CPU resource given by the user's input and the regression coefficients. We use (2), (3), (4) constraints for high availability of the components: multiple replicas are helpful for load balancing on th e write path but are also important for providing reliability, e.g., a Distributor failure may result in a temporary halt on the write path. It can potentially lead to data loss if not handled correctly. As for (5) and (6), we assume that T, Q are provided by the user, currently set to be non-negative. We also assume that the memory limit M is also provided by the user. From the measurements, we managed to formalize the main Cortex components' memory consumption, and we formalized it in Constraint (7). For the components which are not incorporated in the model, the total memory

need is approximately 1500 MB, not significantly varied by the load on the application. As for the three components used in the model, an Ingester needs approximately 1500 MB memory when used by both paths parallel, an Nginx only needs 128 MB memory, while the Distributor needs approximately 512 MB. Furthermore, from the tests, we can confirm that not only the CPU usage is being influenced by the number of time series but also the memory usage: for every 100,000 metrics, there is a need for another 1000 MB by the microservices in total. Constraint (8) stands to ensure high availability: from the measurements, we found that launching an additional Ingester replica for every 100,000 time series provides a safe operation with the load balance applied.

When the optimization yields the minimal CPU usage results with the optimal I, D, N replica numbers, one must translate the results to resource limits for every microservice instance. In the Cortex case, Table 2 indicates the fraction of the total CPU usage among the respective microservices.

4.4 Integer Linear Programming for Memory Optimization

We also build on the previously proposed regression model in this section but this time we use the memory constraint as the target function to create a minimization problem for the memory to be allocated. As we examine the same microservices, most constraints can be applied without any modification.

We again use the following notation:

- T = number of time series used [10,000]
- I = number of ingester instances
- D = number of distributor instances
- N = number of Nginx instances
- Q = level of data aggregation [s]
- C = CPU resource available [millicore]

The constraints are nearly the same; our user input will be the number of time series used, the level of aggregation, and this time the CPU resource available to use.

$$128N + 512D + 1500I + 1000T/10 \tag{9}$$

Table 2 The components' CPU consumption shares

Component	% of total CPU usage
Ingester	32
Distributor	26
Querier	14
Query frontend	13
Store gateway	9
Nginx	6

subject to

$$I > 1 \tag{10}$$

$$D > 1 \tag{11}$$

$$N > 1 \tag{12}$$

$$T > 0 \tag{13}$$

$$Q > 0 \tag{14}$$

$$9.28T + 32.53D + 24.21I + 1.91N - 0.182Q < C \tag{15}$$

$$T/100 - I > 0. \tag{16}$$

Our objective function is (1), where we want to minimize the memory resource given by the user's input. We still use constraints (2), (3), (4) for high availability between the components. (5) and (6) are the constraints for non-negativity, constraint (7) in this case the CPU constraint which was previously used as a target function. Constraint (8) is for the high availability of the Ingesters while increasing the number of time series.

4.5 Illustrative User Input Examples and Optimization Results

In order to demonstrate the applicability of the suggested optimization, we created a few user input examples and calculated the respective results. We used PuLP [21] for the ILP optimization. We used the COIN-OR Branch and Cut Solver (CBC), which is an open-source mixed-integer linear programming solver written in C++. Other available solvers include GNU Linear Programming Kit (GLPK), LP Solve, Coin-or linear programming (Clp). Given the size of the problem space and the carefully crafted constraints, the ILP examples were solved in seconds.

Although Cortex's documentation recommends capacity planning [22], those resource figures are not optimal when we strive to reduce and keep the resource usage as low as possible. Along with the illustrative examples, in Table 3 we also show the recommended CPU provisions based on Cortex's documentation and the relative saving which we managed to achieve with the regression and optimization models. In the examples, this saving ranges from 54 to 82%. If we translate these numbers to cost, i.e., we deploy these microservices in a public cloud and pay for the reserved resources, then we can save significant amounts by applying the proposed method.

With the first example, the inputs are 300,000 time series with 12 GB (gigabyte) memory, and the aggregation should be 10 s. This test shows that the expected memory usage will be around our available amount. The model suggests using 3

Table 3 Illustrative CPU optimization examples with Cortex's recommendation

Time series used (1000)	Memory available (GB)	Aggregation (s)	Ingester replica number	Predicted CPU usage (millicores)	Cortex planned CPU (millicores) [22]	Relative saving
300	12	10	3	1300	6000	78
400	16	60	4	1405	8000	82
120	9	15	2	1105	2400	54
250	8	120	Infeasible		5000	–
250	12	120	3	1218	5000	76

Ingesters and 2–2 Distributors and Nginx. Our expected average CPU usage will be around 1300 millicores, and the memory usage should be around 11.8 GB.

With the second example, the inputs are 400,000 time series with 16 GB (gigabyte) memory available, and our aggregation should be 60 s. The model suggests using 4 Ingesters and 2–2 Distributors and Nginx. Our expected average CPU consumption will be around 1405 millicores, and the memory usage should be around 14.8 GB. This test shows that the expected memory usage will be enough, and the microservice will still have some available amount of memory if needed.

With the third example, the inputs are 120,000 time series with 9 GB (gigabyte) memory available, and the aggregation should be 15 s. The model suggests using 2 Ingesters and 2–2 Distributors and Nginx. The expected average CPU usage will be around 1105 millicores, and the memory usage should be around 8 GB. This test shows that the expected memory usage will be around the available amount, just like the first example.

With the fourth example, the inputs are 250,000 time series with 8 GB (gigabyte) memory available, and the aggregation should be 120 s. The model suggests using 2 Ingesters and 2–2 Distributors and Nginx. The expected average CPU consumption will be around 1205 millicores, and the memory usage should be around 11.3 GB. Here the model shows that using 8 GB memory will not be enough, so we should solve the problem by using more memory or reducing the number of time series used.

Table 3 shows the illustrative examples to demonstrate the model's capability for resource savings. The inputs are the number of time series used, the memory available for use, and the aggregation level of the data. Our outputs are the number of Ingesters, Nginx, Distributors, and the predicted CPU usage. We omitted the replica numbers for Nginx and Distributors in the table as these numbers are constantly 2–2. The reason for this is because we only have a high availability constraint for these microservices so that the model will use at least 2, but because there is no real impact when scaling these components, the replica numbers remain 2.

Using the other ILP model introduced in Sect. 4.4, we can also calculate the precise memory usage we expect. The results we get are lower than the input values we applied in Table 3, and in most cases lower than the official recommendation for Cortex. For the first test, the model had 12 GB of memory; the expected average

Table 4 Illustrative memory optimization examples with Cortex's recommendation

Time series used (1000)	CPU available (millicore)	Aggregation (s)	Ingester replica number	Predicted memory usage (MB)	Cortex planned memory (MB) [22]	Relative saving
300	1300	10	3	11,800	13,500	13
400	1450	60	4	14,800	24,000	38
120	1450	15	2	7800	3600	− 117
250	1250	120	3	10,300	11,250	8

usage will be around 11.8 GB. The second example has 16 GB available; the results should be around after the calculations 14.8 GB. For the third example, the model can use up to 10 GB of memory; the output was 7.8 GB of memory usage. Furthermore, for the last example, our initial 8 GB was not enough, as we can see the optimization is infeasible. The problem was that our memory reservation input was too low. After giving 12 GB of spare memory, the model returned with an expected usage of 10.3 GB.

We summarize the memory optimization results in Table 4. The Cortex planned memory column contains memory allocation figures based on the recommendation "Each million series in an ingester takes 15 GB of RAM". The rightmost column of Table 4 shows slight relative savings compared to this baseline for all illustrative cases but one: when the number of ingested time series is relatively low, this rule of thumb at [22] underestimates the necessary amount of memory Cortex consumes.

5 Conclusion

In this paper we argued that resource provisioning is essential and, in fact, possible to perform scientifically when using cloud-native microservices. We proposed an approach that we demonstrated on a data ingestion and storage cloud-native application, Cortex. After measuring the performance vs resource footprint trade-offs with several configuration settings and creating a regression model for resource usage, we showed the most critical features and their quantitative effects on CPU utilization, which proved to be linear. Then we introduced an integer linear programming formulation that can be solved to minimize the expected CPU usage for given user inputs. With some illustrative examples, we showed that a prudent resource reservation might halve the costs paid for cloud resources while meeting the QoS constraints set for the application.

Author Contributions RE and LT wrote the main manuscript text and RE prepared all figures. LT reviewed the manuscript.

Funding Open access funding provided by Budapest University of Technology and Economics. This work was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects (i) no. 135074 under the FK_20

funding scheme and (ii) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme. L. Toka was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Supported by the ÚNKP-22-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

Declarations

Conflict of interest Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Samreen, F., Elkhatib, Y., Rowe, M., Daleel, G.S.B.: Simplifying cloud instance selection using machine learning. In: IEEE/IFIP Network Operations and Management Symposium (NOMS 2016) (2016)
2. Richardson, C.: What are microservices?, https://microservices.io Accessed 30 May (2022)
3. Erdei, R., Toka, L.: Optimal resource provisioning for data-intensive microservices. In: IEEE/IFIP International Workshop on Analytics for Network and Service Management (AnNet'22) (2022)
4. Jindal, A., Podolskiy, V., Gerndt, M.: Performance modeling for Cloud microservice applications. In: ACM/SPEC International Conference on Performance Engineering (ICPE'19) (2019)
5. Zhang, Y., Hua, W., Zhou, Z., Suh, E., Sinan, D.C.: Data-driven resource management for interactive multi-tier microservices. In: Workshop on ML for Computer Architecture and Systems (MLArchSys) (2020)
6. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P.: Dynamic provisioning of multi-tier internet applications. In: International Conference on Autonomic Computing (ICAC'05) (2005)
7. Leitner, P., Cito, J., Stöckli, E.: Modelling and managing deployment costs of microservice-based cloud applications. In: IEEE/ACM International Conference on Utility and Cloud Computing (UCC) (2016)
8. Ma, X., Gao, H., Xu, H., et al.: An IoT-based task scheduling optimization scheme considering the deadline and cost-aware scientific workflow for cloud computing. J. Wirel. Comun. Netw. **2019**, 249 (2019)
9. Rossi, F., Nardelli, M., Cardellini, V.: Horizontal and vertical scaling of container-based applications using reinforcement learning. In: IEEE International Conference on Cloud Computing (CLOUD'19) (2019)
10. Li, C., Liu, J., Lu, B., Luo, Y.: Cost-aware automatic scaling and workload-aware replica management for edge-cloud environment. J. Netw. Comput. Appl. **180**, 103017 (2021)
11. Eramo, V., Lavacca, F.G.: Proposal and investigation of a reconfiguration cost aware policy for resource allocation in multi-provider NFV infrastructures interconnected by elastic optical networks. J. Lightw. Technol. **37**(16), 4098–4114 (2019)
12. Wang, C.-H., Llorca, J., Tulino, A.M., Javidi, T.: Dynamic cloud network control under reconfiguration delay and cost. IEEE/ACM Trans. Netw. **27**(2), 491–504 (2019)
13. Kumar, J., Singh, A.K.: Decomposition based Cloud resource demand prediction using extreme learning machines. J. Netw. Syst. Manag. **28**(4), 1775–1793 (2020)

14. Sudan, M.: The P vs. NP problem. https://madhu.seas.harvard.edu/papers/2010/pnp.pdf (2010)
15. Cortex https://cortexmetrics.io. Accessed 30 May (2022)
16. Prometheus. https://prometheus.io. Accessed 30 May (2022)
17. Cloud Native Computing Foundation. https://www.cncf.io. Accessed 30 May (2022)
18. MinIO. https://min.io. Accessed 30 May (2022)
19. Kubernetes. https://kubernetes.io. Accessed 30 May (2022)
20. Nginx. https://www.nginx.com. Accessed 30 May (2022)
21. PuLP. https://pypi.org/project/PuLP. Accessed 30 May (2022)
22. Capacity Planning. https://cortexmetrics.io/docs/guides/capacity-planning. Accessed 30 Oct (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Roland Erdei[1] · Laszlo Toka[1,2,3]

Roland Erdei
erdeiroland@edu.bme.hu

1 Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, 2, Magyar tudosok krt., Budapest 1117, Hungary

2 MTA-BME Network Softwarization Research Group, Budapest, Hungary

3 ELKH-BME Cloud Applications Research Group, Budapest, Hungary