Lightweight Self-Adaptive Configuration Using Machine Learning

Rodrigo Araújo
Department of Computer Science
University of British Columbia
Vancouver, Canada
rodarauj@cs.ubc.ca

Reid Holmes
Department of Computer Science
University of British Columbia
Vancouver, Canada
rtholmes@cs.ubc.ca

ABSTRACT

Modern distributed systems are comprised of many components that often have complex configuration parameters to allow them to be tuned to differing runtime requirements. Engineers must manually adjust many of these parameters to achieve their desired runtime behaviours. Unfortunately, static configurations are often insufficient, but ad hoc configuration modifications can unexpectedly degrade overall system quality.

In this work, we describe Finch, a tool for injecting a machine learning-based MAPE-K feedback loop into existing REST-based systems to automate configuration tuning. Finch configures and optimizes systems according to service-level agreements under uncertain workloads and usage patterns. Rather than changing the core infrastructure of a target system to fit the feedback loop, Finch asks the user to perform a small set of actions: adding limited instrumentation to the code and configuration parameters and defining service-level objectives and agreements. With these changes, Finch learns how to dynamically configure the system at runtime to self-adapt to dynamic workloads.

We provide a proof-of-concept evaluation to demonstrate how Finch can provide an automated self-adaptive system that replaces the trial-and-error engineering effort that otherwise would be spent manually optimizing a system's wide array of configuration parameters.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **System administration**.

KEYWORDS

Self adaptation, Software configuration, Machine learning

ACM Reference Format:
Rodrigo Araújo and Reid Holmes. 2021. Lightweight Self-Adaptive Configuration Using Machine Learning. In *Proceedings of CASCON Conference (CASCON'21).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

The industrial adoption of microservices (e.g., databases and their replication infrastructure, caching components, proxies, and load balancers) has led to increasingly complex configuration schemes that are manually fine-tuned by engineers. Ganek and Corbi discussed the need for autonomic computing to handle the complexity of managing software systems [14]. They noted that managing complex systems has become too costly, error-prone, and labour-intensive because pressured engineers make mistakes, increasing the potential of system outages that can impact business operations. This has driven many researchers to study self-adaptive systems (e.g., [11, 13, 17, 24, 25, 29]); however, the software industry still lacks practical tools to provide self-adaptive system configurations. Thus, most system configuration and tuning is performed manually, often at runtime, which is known to be a very time consuming and risky practice [10, 14, 31].

In this work we present Finch, a tool that enables engineers to integrate self-adaptation mechanisms into their systems. Finch delegates the configuration and tuning of a system to a learned model, rather than requiring engineers to perform these operations manually or through manually tuned heuristics. A high-level view of how Finch integrates with a target system is given in Figure 1.

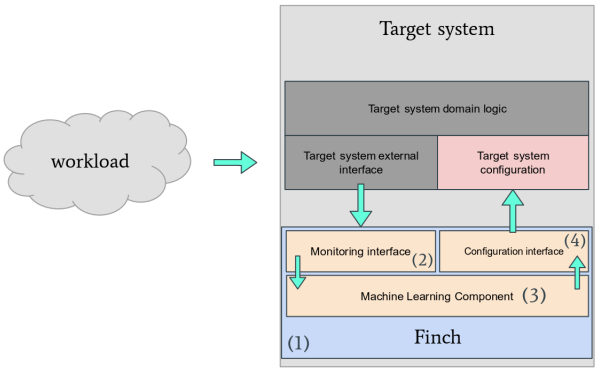

Figure 1: Overview of Finch target system integration. Finch: (1) is injected into the target system; (2) monitors and analyzes the target system's context; (3) learns how to configure the target systems; and (4) executes configuration adaptation plans to update the target system's configuration.

Building self-adaptive systems is a major engineering challenge [5]. Finch aims to enable lightweight self-adaptation by giving the user the ability to inject the main components of a self-adaptive mechanism into an existing target system in a loosely-coupled fashion. One of Finch's main goals is to provide lightweight self-adaptive

CASCON'21, November 22–26 2021, Toronto, Canada

configuration support with minimal engineer effort. Finch automatically assesses the system's workload and predicts the impact of configuration changes that could potentially improve the system's runtime behaviour, and automatically applies these changes.

Our approach consists of providing mechanisms for injecting a control loop into an existing target system, and an API for collecting relevant system metrics and configurations as the system executes. Developers map Service Level Agreements (SLAs) to a subset of the metrics collected for the system. These are evaluated by the machine learning component that is concurrently relearning the model while analyzing current event data and predicts optimal configurations for the system for its given workload. Finch provides configuration adaptation plans that can be both *automatically executed*, allowing the system to have self-adaptive capabilities, and *interpretable*, allowing engineers to understand the impact of a change in the configuration space before it is deployed.

The main contributions of this paper are:

- An initial approach for augmenting existing systems with a self-adaptive feedback loop for tracking and improving system behaviours under load.
- A group of experiments to evaluate Finch's performance when integrated into a web service, demonstrating how Finch can learn how to improve system performance with automatically learned configurations.

Section 2 discusses past research in the space of self-adaptive systems. Section 3 outlines the design and usage of Finch. Section 4 presents an initial evaluation of the approach, followed by a discussion on limitations and future directions in Section 5. Section 6 concludes.

2 RELATED WORK AND FOUNDATIONS

Our approach draws ideas from many different, although overlapping, fields. Here we discuss where these ideas come from and how they relate to Finch.

Control theory in software engineering. The ideas in control theory have been widely adopted in the software engineering research community, with special attention to the Monitor-Analyze-Plan-Execute over a shared Knowledge, known as MAPE-K feedback loop, which proved to be a powerful tool to build self-adaptive systems [1, 4, 8, 10, 19, 29]. Angelopoulos et. al. discussed the intersection between software engineering and control theory [12]. They showed how control-theoretical software systems are implemented and their design process, as well as the differences of the word "adaptation" in both fields. All These works were shown to be invaluable to the development of Finch, because the injection of a MAPE-K loop into the target system is the core component of Finch.

Time series analysis. Time series data has been used to analyze and predict patterns in data with respect to time, with applications on understanding how to efficiently allocate computational resources, which is a key strategy in our work to provide ahead-of-time adaptation.

Many techniques for forecasting workload and performance metrics using time series data have been developed (e.g., [3, 7, 15, 16, 22]). With these forecasts, these authors provided methodologies for virtual machine allocation in data centres. These works did not focus on tools for applying machine learning to software systems nor on tools to enable self-adaptability in arbitrary software systems—which is our end goal.

Workload modelling. Another important aspect of Finch is being able to simulate workload intensity for initial training of the adaptation model. To have an accurate workloads, we need to model them as closely as possible to real-world workloads. Herbst et. al. presented the Descartes Load Intensity Model [20], a powerful tool for describing load intensity variations over time, that can also be used for an accurate benchmarking based on realistic workload and performance analysis. Finch uses some of these ideas to model and simulate workloads for training the adaptation model.

Self-adaptive systems. Cornel Barna et. al. proposed Hogna, a platform for deploying self-adaptive applications in cloud environments [2]. Hogna provides a framework that abstracts deployment details, for example: spinning-off and managing instances on Amazon EC2 or OpenStack, enabling the user to focus on the adaptation mechanism. A key difference between Hogna and Finch is that Finch is not a deployment framework, but rather a library that assists the implementation of a MAPE-K closed loop by abstracting formal modelling to a machine learning model that can be matched with the specified SLAs, instrumented data, and identified configuration parameters of the system.

Most previous work approaches the adaptation problem with reactive strategies: when a violation occurs—the service gets slower, errors are thrown—an adaptation is triggered and executed, stabilizing the system. Finch is capable of providing the same style adaptation, while providing ahead-of-time adaptation: we use time-series analysis to create an adaptation plan for a future time, executing it right before a violation occurs, mitigating the risk of a potential SLO violation.

Andrew Pavlo et. al. presented Peloton, a database system designed for autonomous operation [24]. Similar to Finch, one of their main goals was to decrease the need for manually-performed operations, though they focused solely on applying their ideas and techniques to their DBMS implementation. They achieved this by classifying the workload trends, collecting monitoring data, and forecasting resource utilization, then training a model based on this data to predict the best optimization plan. These ideas are important to our work, the key difference is that instead of directly embedding these ideas in a specific system—in this case a DBMS—and requiring the autonomous components to be tightly coupled to the system being configured, we are embedding a subset of these ideas in a tool that can be integrated in any arbitrarily chosen software system.

More recently, self-adaptive configuration has been explored to support auto-scaling of containerized environments (e.g., [18, 27, 28]). Some approaches have also explored using Q-Learning to help learn SLA parameters to avoid the drawbacks of manually-defined SLA thresholds [18]. These techniques have also been used to enable serverless computing approaches to flexibly scale based on user demand and other resource constraints [30].

Machine learning-enhanced software systems. In a recent work entitled The Case for Learned Index Structures, Kraska et. al. have

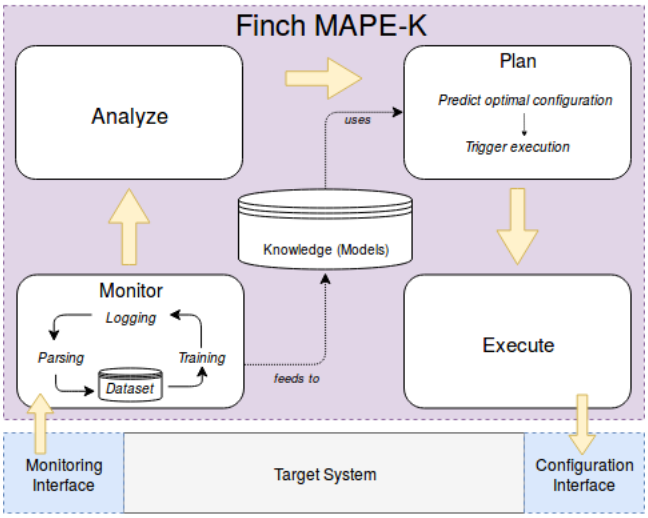

Figure 2: Finch's high-level architecture. The target system needs to provide information to the monitoring and configuration interfaces, but are otherwise unchanged. The Finch ML-based MAPE-K feedback loop monitors and plans adaptation plans that are automatically deployed to the system at runtime according to workload demands.

demonstrated that machine learned models have the potential to provide significant benefits over state-of-the-art database indexes [21]. This research showed that by replacing manually tuned heuristics with learned models enabled it to outperform cache-optimized B-Trees by up to 70%.

We draw much of our inspiration from this work; Finch's central idea is to allow systems that relies heavily on manual configurations and heuristics to be enhanced with learned models. This could be applied to many different domains. In this work we apply this idea to a REST-based API backend.

The idea of machine learning-enhanced software systems is to move from using the same algorithms, heuristics, data structures, or configurations in multiple different contexts, to personalized configurations; different configurations that perform better for different scenarios. This relates well to the No Free Lunch theorem:

> If an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems.

This is the main idea behind Finch: the integration of learned models to generate adaptation plans according to the different scenarios.

3 APPROACH

Our approach has been designed to integrate into existing systems with as few manual steps as possible. In this section we discuss these integration steps and then detail how Finch uses these steps to enable the target system to automatically adapt its configurations according to runtime workloads.

3.1 Target system integration

The high-level architecture of Finch is shown in Figure 2. The bottom tier of this figure represents the target system. To enable automatic self-adaptation, the target system developer must add some limited instrumentation so Finch can monitor the system at runtime. They must also provide a description of the key configuration parameters that Finch can modify to adapt the target system.

Defining configuration parameters. In order to automatically adapt a target system's configuration, Finch needs to know what the valid configuration space is. This is a required, but incremental, process: developers do not need to specify all possible configuration parameters. Parameters can be either discrete or continuous. Discrete parameters take specific value; the space of which must be specified. Continuous parameters are only applicable for numeric values; these parameters require minimum and maximum values. Both kinds of parameters are provided with an initial starting value. Additionally, there are two kinds of configuration parameters, *normal* parameters can be changed and will automatically be reflected in the runtime execution of the system, and *complex* parameters that require an external script to be called (e.g., to restart a service) once the value has been changed.

Normal configuration parameters are described by defining the name of the configuration parameter, the current value, and the possible values or range values. Figure 3 shows an example of a normal configuration parameter definition. In this case, `parameter_1` can take any value between 1 and 1,000, and the user has specified that 750 should be used as the initial value. Figure 4 shows an example of a complex parameter for the Postgres database system. In this case, the developer has defined an initial value of 128 for the `pg_shared_buffers` parameter and given four other discrete values Finch can use while evaluating alternative configurations. If Finch deploys a configuration that modifies this parameter value, it calls the user-created *configurePG* script which will restart the Postgres server to deploy the change.

```
[
  "parameter_1": {
    "value": 750,
    "valueType": "range",
    "values": [1, 1000],
    "isCustom": false
  }
]
```

Figure 3: A normal adaptive configuration definition. The initial value for the parameter is 750 but Finch is free to assign any value between 1 and 1000.

Some configuration parameters require updating files on disk and restarting services before they can take effect. The `isCustom` property identifies these cases by allowing Finch to invoke the function defined in `adaptationMethod` to update the configuration and restart the service; these methods are typically only a few lines of code (e.g., update a value in a text-based configuration file, restart the service).

```
[
  "pg_shared_buffers": {
    "value": 128,
    "valueType": "discrete",
    "values": [16, 128, 4000, 16000],
    "isCustom": true,
    "adaptationMethod": "configurePG"
  }
]
```

Figure 4: A complex adaptive configuration definition. The variable can be set to one of four values, with its initial value being 128. After Finch applies a configuration that changes this parameter, the specified `adaptationMethod` script will be called to reflect the value in the running system.

```
[
  {
    "sla": "Retrieving the listings should not take > 250ms",
    "endpoint": "/v2/listings",
    "method": "GET",
    "metric": "latency",
    "threshold": "250",
    "agreement": "95"
  }
]
```

Figure 5: Example of a latency-based SLA a user could use to characterize how long a request should take. In this case, 95% of the requests to the `listings` endpoint should take fewer than 250ms.

```
[
  {
    "sla": "Report service should scale to 100 requests per second",
    "endpoint": "/v2/reports",
    "method": "POST",
    "metric": "throughput",
    "threshold": "100",
    "agreement": "90"
  }
]
```

Figure 6: Example of a throughput-based SLA a user could define for their service. Here, the `reports` endpoint should be able to scale to 100 requests per second 90% of the time.

Specifying Service Level Agreements (SLAs). The current implementation of Finch only supports SLAs that evaluate the performance of REST endpoints. As such, supporting the monitoring interface requires only a single programmatic call needs to be made from the target system Finch which then automatically adds instrumentation to all existing REST endpoints in the target system to track latency and throughput for each endpoint. These performance metrics can then be compared evaluated along with the current workload relative to the SLAs to determine the quality of the underlying configuration.

As with the configuration parameter space, these need not be exhaustive: developers can start with a single SLA. They can also add more SLAs, or modify existing SLAs over time as their operational needs and goals evolve. SLAs are defined with percentile thresholds to be more sensitive to important violations of expected performance attributes. This is especially important for latency and throughput because average values can hide outliers and can be easily skewed. To better illustrate this problem, consider the following scenario: A target system receives 100 requests per minute, 80 of which take 200ms to serve (which is relatively quick). The remaining 20 requests take 10,000ms (10 seconds). Assessing the performance of the system using the average, suggests the latency is 2.1 seconds, which is an acceptable value; however, this obscures the fact that 20% of the requests take 10 seconds, which is an unacceptable latency value.

Exemplar SLAs are given in Figure 5 and Figure 6. Finch currently only supports `latencey` and `throughput` metrics for SLAs. The threshold and agreement values are used to determine whether the current system performance is in violation of the SLA. Defining SLAs at the HTTP endpoint level of REST-based systems matches well with how designers specify the quality attributes of these endpoints. Endpoints like these are also an accessible level of abstraction for engineers as they define important runtime expectations for their systems.

Observing runtime behaviour. To observe the system as thoroughly and efficiently as possible, Finch makes extensive use of modern observability and software instrumentation techniques. These techniques inject code into parts of the target system to record its context. Instrumentation can evaluate function parameters, latencies, and time to execute certain code blocks. The purpose of collecting information can be used to help measure performance, assist debugging tasks, and find system bottlenecks.

A user needs to instrument their system to use Finch. Luckily, the software industry has been enforcing system instrumentation by providing many solutions, such as Dtrace [6], Prometheus [26], Nagios [23], and Datadog [9], so this requirement is often not out-of-step with common system infrastructure. Instrumentation is also heavily used in industry to detect Service-Level Agreement violations and to perform resource management—two tasks that are essential for Finch to fulfill its purpose.

Finch instruments the target system using the Prometheus platform [26]. Prometheus contains many language-specific client libraries to enable it to instrument systems implemented in over a dozen languages. Prometheus is a pull-based monitoring tool and a time-series database. Unlike monitoring tools like Nagios, which frequently executes check scripts, Prometheus only collects time series data from a set of instrumented targets over the network. For each target, the Prometheus server simply fetches the current state of all the metrics over HTTP and has no other execution overhead that would be pull-related.

To monitor the target system, Finch provides a small instrumentation API. This API consists of three methods. The first is for workload monitoring. The second is for latency monitoring, which takes the endpoint, the HTTP method, and the duration of the request. The last method is for configuration parameter monitoring.

3.2 Enabling self-adaptation with Finch

According to the self-adaptive systems community, a centralized and top-down self-adaptive system operates with the guidance of a central controller. This controller assesses its own behavior with respect to its current surroundings, and adapts itself if the monitoring and analysis warrants it [5]. Given this definition, we built Finch to follow a centralized and top-down approach.

The main design goal of Finch is to allow its users to inject a MAPE-K feedback loop into their system through its API. To carry out an effective reasoning on the target system's context uncertainty, we need visible feedback loops that are first class citizens in the system, as discussed by Y. Brun et. al. [5]. In industry, the self-adaptation mechanism is *hard-wired* into the managed system most of the time. That is, they change the managed element's structure to fit the feedback loop into the target system.

Of course, this requires a noticeable engineering effort; usually systems are not initially designed with self-adaptability in mind. This is why Finch relies on automatically-injected instrumentation; rather than hard wiring the self-adaptation mechanisms inside the target system, Finch acts as a co-pilot that automatically collects data related to the system's context, environment, and states, storing this data for future reference and model training. Guided by an internal feedback loop, Finch carries out execution plans that aim to ensure the target system does not violate its SLAs. The adaptation leads to more event data to be stored and analyzed, and the cycle repeats.

Configuration adaptation as a learning problem. A *learning problem* can be defined as a set of observations comprised of input and output data, and some unknown relationship between the two. The goal of a learning system is to learn a mapping between input and output data, so that predictions can be made for new instances drawn from the domain where the output variable is unknown.

The main hypothesis behind Finch is that if we can model the configuration scheme or the heuristics of a system as a learning problem, then Finch can learn models that capture patterns between the system's context and the system's configuration parameters, enabling the system to predict the optimal set of configuration parameters for a specific observed workload. This prediction can be used to either adapt to different workloads that require different configurations or to prevent poor configurations.

To construct its dataset, Finch collects four classes of features on the target system; performance and behavior metrics, configuration parameters, the workload, and service level indicators. The goal of choosing these features was to gather a dataset that a machine learning model could be trained on, which would later make predictions based on these features. These particular features were chosen so Finch would be able to dynamically find an optimal set of configuration parameters for a given workload.

3.2.1 Resulting dataset. To make its configuration adaptation plans, Finch collects four classes of features on the target system: performance and behavior metrics, configuration parameters, the workload, and service level indicators (SLI). The goal of these features is to gather a dataset that the machine learning models can be trained on, which would later make predictions based on these features. These particular features were chosen in order to answer the following question: *For a given the workload (e.g, requests per second) and the performance metrics of the system, what is the optimal set of configuration knobs (parameters) that will prevent SLA violations, in this case, in the requests served?*

The dataset is constructed in such a way that each row describes the context of the system —workload, metrics, SLIs, and configuration knobs— at a given timestamp.

The following matrix summarizes how the dataset is organized:

$$\begin{vmatrix} t_1 & W_1 & M_{1_1} & M_{2_1} & \dots & M_{i_1} & k_{1_1} & k_{2_1} & \dots & k_{l_1} & SLI_{1_1} & SLI_{2_1} & \dots & SLI_{\delta_1} \\ t_2 & W_2 & M_{1_2} & M_{2_2} & \dots & M_{i_2} & k_{1_2} & k_{2_2} & \dots & k_{l_2} & SLI_{1_2} & SLI_{2_2} & \dots & SLI_{\delta_2} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ t_n & W_n & M_{1_n} & M_{2_n} & \dots & M_{i_n} & k_{1_n} & k_{2_n} & \dots & k_{l_n} & SLI_{1_n} & SLI_{2_n} & \dots & SLI_{\delta_n} \end{vmatrix}$$

Where:

(1) n is the number of collected samples
(2) t_ϕ is the timestamp of the ϕ^{th} example
(3) M_{j_ϕ} is the j^{th} instrumented metric in timestamp t_ϕ. j ranges from 1 to i, the last instrumented metric.
(4) k_{c_ϕ} is the value in the c^{th} configuration parameter in timestamp t_ϕ. c ranges from 1 to l, the last collected configuration knob.
(5) SLI_{o_ϕ} is the o^{th} service level indicator in the timestamp t_ϕ— which is one of the instrumented metrics that was set to be an SLI. o ranges from 1 to δ, the last collected SLI.

This dataset captures the context of a system with respect to workload, instrumented metrics, and the values in configuration parameters.

3.3 Running Finch

Upon launch, Finch reads the SLA and configuration parameter descriptions and will add the required instrumentation to the target system. Initially, Finch will work as passive co-pilot; it collects data, analyzes it, and frequently builds a dataset with this data. After a while, it starts training models on this dataset, and if the accuracy is acceptable, whenever there's an SLA violation, it will trigger configuration adaptation in order to try to improve the target system performance.

Finch's main components were written using the Go programming language and the machine learning components were developed in Python. Finch's runtime spawns two main lightweight threads, which in Go are called Goroutine(s). These two Goroutines are two observer threads. The first is responsible for periodically building the dataset. This thread will periodically extract all collected metrics from Prometheus through its HTTP API, parse this data, and save the dataset. It then calls the machine learning component to train the models using this dataset. The second Goroutine is responsible for monitoring the current state of the system by querying Prometheus every few seconds, and checking the the current SLI values. Upon violation of an SLA, it calls the machine learning component, uses the most recently trained models to predict the most optimal configuration, then calls each respective adaptation method responsible for changing its configuration in the target system. Both Goroutines are controlled by two variables: one that

controls how often the dataset is constructed, and another one controls how frequently the current context is observed.

3.3.1 The MAPE-K feedback loop. The MAPE-K feedback loop periodically builds the dataset starts an inner loop that extracts all metrics from Prometheus and builds the necessary dataset for training at a configurable interval.

Monitoring and analyzing the current context and state of the target system requires more non-trivial work, such as periodically extracting, from Prometheus, a single row of metrics of that given timestamp, analyzing, extracting, and saving the current state of all SLAs defined by the user for the target system against the current context, checking if there is any SLA violation, triggering the adaptation procedure, waiting for the adaptation to fully propagate, and checking for improvements in order to prevent unnecessary new adaptations. A depiction of the target system's runtime behaviour triggering a Finch configuration change and responding to the change is shown in Figure 8.

Internally, Finch implements a state machine to keep track of its operations in order to ensure that the target system is progressing and to control adaptations. Figure 7 illustrates this state machine.

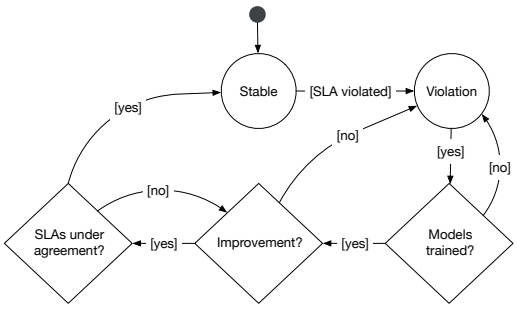

Figure 7: Finch's state machine that is followed at runtime to monitor the target system.

3.3.2 Machine learning architecture. Given the previously defined dataset, Finch trains many different models, one for each SLA's indicator. In the end we want to predict the SLI, given the set of configuration parameters and system metrics—including workload.

Finch has 2 ML pipelines. The first ML pipeline trains the models, which includes basic standardization, normalization, grid search, and cross validation. The second ML pipeline predicts the SLI, given the configuration parameters. After running these pipelines, the last step is finding the optimal configuration.

Training pipeline. As mentioned before, Finch trains a model for each SLA indicator. If the user has two SLAs with respect to the latency of endpoint A and B, then the two collected SLIs are the 99^{th} percentiles of these endpoints' latency. Thus, Finch will train two models, one for each SLI.

Training different models requires slicing the original dataset to fit the models' needs. For example, when we want to predict the latency of endpoint A, latency A is the target, or y, of the model, and the corpus, or X, is the rest of the dataset *minus* the other SLIs collected. This way, the system metrics, workload and configuration parameters are isolated for the model training.

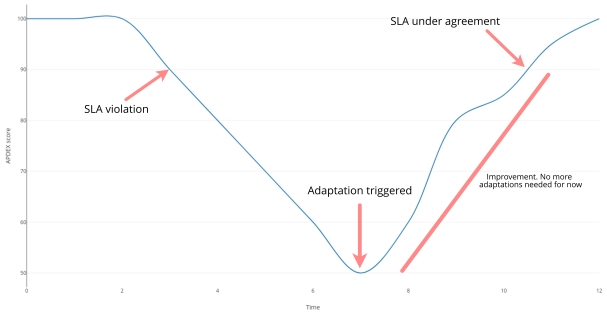

Figure 8: An exemplar of how a SLA violation could be detected, a threshold for adaptation reached triggering a new configuration, and how the SLA could come back into compliance once the new configuration is deployed to the target system.

3.3.3 Creating adaptive machine learning models. Since the dataset is specific to the target system, there is no one global configuration model that would be applicable to all systems. For example: we cannot simply use logistic regression or a neural network with static hyperparameters. A system-specific dataset means that it can have an arbitrary dimension (number of features) and size, it can have continuous values, discrete values, or both. Finch cannot know this beforehand. Thus, to work with uncertain datasets, when training the models Finch must perform a grid search.

Grid search is a technique to search for the best hyperparameters and models. Hyperparameters are parameters that are not directly learned by the models. They are parameters that configure certain aspects of a given machine learning models, for instance: how deep a decision tree should be, how many decision trees (i.e estimators) a random forest should have, or how many layers a neural network should have. Each machine learning model performs better when choosing the right model and the right hyperparameters for a given dataset. Some combinations of models-hyperparameters perform better with highly dimensional data, while some perform better with well-balanced datasets, some are more resistant to outliers, while sometimes the outliers is what you are trying to identify.

In Finch's training pipeline, grid search exhaustively considers all hyperparameter combinations and many different models, trains one model per combination, and selects the best performing one. This adds a considerate time and space complexity in the training pipeline, but cannot be avoided when choosing a model that will not overfit or underfit on dynamically generated datasets.

To ameliorate this, grid search is only performed in two scenarios. First, in the initial training cycle, where Finch first handles the extracted dataset, after the first training cycle, it will know the best hyperparameters for the learned models and use them to re-train the model with the new data. Second, when the model's prediction performance starts to degrade, indicating that the dataset has changed in some important aspect, triggering a need to re-learn improved models and hyperparameters from the new dataset.

In this pipeline, models such as linear regression, ridge regression, lasso, support vector machines, and decision trees are all

considered. However, in the evaluations performed in this work, which used a few variations of dataset structure, we have found that one machine learning model/technique worked best for the majority of datasets: gradient boosting with decision trees.

To validate the grid search and avoid overfitting, Finch performs cross-validation with 5 splits, and 30%/70% test/train split ratio.

3.3.4 Predicting the optimal configuration. When defining the adaptive configuration parameters, the user also defines the value range or the possible values. For instance, a certain configuration parameter A could take values ranging from 1 to 100, and another parameter B could take the following array of discrete values: 1, 5, 10, 50. After the models have been trained, Finch could simply predict the optimal configuration by passing the desired SLIs as our X, and in return get the optimal configuration as the y coming from the prediction method.

However, that approach turned out not to be very effective, and we devised an additional algorithm on top of this straightforward call to prediction method. This algorithm was devised to cope with the following problem: in some cases, a configuration parameter did not overlap with respect to its effects on different SLAs. In these cases, a model for a specific SLA predicts the right configuration parameter, but only for that given parameter which affects it directly, and makes inaccurate predictions for the other configuration parameters, since it does not affect it directly. This prediction affects other SLAs negatively. Think of an SLA being selfish and only caring about the configuration parameter that affects it, and not considering the other defined SLAs.

To overcome this problem and find configurations that satisfies as many SLAs as possible, the algorithm establishes consensus between the SLAs through a voting mechanism. To start, the algorithm creates a 2D array with the Cartesian product of all possible parameter combinations, then, for each SLA, it predicts its respective SLI value for each of these combinations. The time to predict all these combinations is negligible, since predictions usually take a short amount of time, even with large matrices.

Next, for each SLA's predictions, Finch filters the configurations that satisfy the SLA plus a tolerance factor. For each SLA this results in a set of configuration parameters that is both diverse and satisfies the SLA constraints. In a final step, the configurations are combined. For each discrete configuration parameter the value that most frequently occurs in the surviving configurations is selected; for continuous parameters, the mean parameter from the configurations is used. Ultimately, this results in a set of configuration parameters which Finch believes will satisfy as many SLAs as possible given the current system load.

3.3.5 Passive and active training modes. Finch is always re-training its models with current data. However, the initial training cycles require grid search to be performed, and during these first few cycles, Finch passively collects data and trains models, but does not make predictions or adaptation plans. During this period, it is necessary to collect a diverse dataset so that Finch can see how how the target system responds to different configurations under different workloads. There are two ways to achieve this: passive and active training modes. These two options mode can be configured in Finch's configuration file. The passive mode just collects data and trains models while the system is running, not intervening with the target system's natural execution. The active training mode can speed up the learning process by allowing Finch to actively and frequently mutate the configuration parameters in the target system in order to more quickly gather a diverse dataset.

4 EVALUATION

We have conducted an initial proof-of-concept evaluation of Finch to provide initial evidence for three main research questions:

- **RQ1:** Can Finch learn configurations that do not violate target system's SLAs?
- **RQ2:** What performance overhead is incurred by Finch?
- **RQ3:** How much training data is needed to make accurate configuration adaptations?

For this prototype evaluation, we opted to use an in-house REST-based inventory management and shopping system; this system captured the most common points of complexity in web services, which are:

- A Docker-based backend component comprising the core logic of the application.
- Multiple HTTP endpoints served over a RESTful API. In our scenario, these endpoints are subject to a set of Service Level Agreements.
- A Postgres database running with a Docker container.

After building this system we modified it to support Finch (as described in Section 3.1). To enable Finch to make meaningful predictions, we created a workload simulator that simulated realistic use cases for the system. The simulator generated random workloads consisting of users browsing shopping items, adding and removing items arbitrarily to a shopping cart, and finalizing their shopping session by checking out. The simulation we created ran these user cases multiple times in parallel in order to stress the system in a realistic way.

4.1 Experiment 1: Configuration throttling

To examine whether Finch infers viable configurations that do not violate the target system's SLAs (**RQ1**), we created a configuration parameter that randomly (and temporarily) slowed down the target system by blocking execution for either B_i milliseconds or $(\frac{1}{B_i}) * 10000$ milliseconds for each throttling point $B \in 1 \ldots i$. This means that a delay will block the execution either proportionally or inversely proportionally to the value of a configuration parameter. For instance, if a configuration value is 1000, in a proportional throttle point, it will block the execution for 1000 ms. In a inversely proportional throttle point, it will block the execution for 10 ms. Thus, if this configuration can take a number between 1 and 1000, it could be one extreme or the other, depending on the type of blocking point.

These B_i values were used as artificial configuration parameters to influence the performance of the target system. We then ran Finch to see if it could monitor, analyze, predict, and execute the adaptation plans that would correctly modify the configuration parameters in such a way that the performance will resolve any SLA violations that arise from the system delays. These adaptations must be automatic: that is, Finch must modify the target system's configuration to minimize the incurred delays without any explicit programming.

To evaluate this, we ran 3 different sets of random artificially generated configuration parameters and studied Finch's performance on them. We focused on 2 questions during this experiment:

- Can Finch predict the optimal or the sub-optimal configuration?
- If yes, how long does it take to converge to the optimal or sub-optimal configuration?

For the model accuracy, it was used the coefficient of determination R^2 of the prediction, where $R^2 = 1 - \frac{u}{v}$, where u is the residual sum of the squares $\sum_{i=1}^{n}(y_{true} - y_{pred})^2$ and v is the regression sum of squares $\sum_{i=1}^{n}(y_{true} - \overline{y_{true}})^2$.

Experiment 1 Results: All tests were ran on a Dell laptop running Ubuntu 14.04 with 4 Intel Core i7-5500U CPU @ 2.40GHz and 16 GB of memory. The target system had five configuration parameters and each training cycle took one hour. From these three experiments, Finch successfully adapted the configuration after a single training cycle and learned the optimal set of configuration parameters, achieving 100% on its predictions and stabilizing after the second cycle. Table 1 shows the results of these 3 experiments.

However, when the system runs for multiple days, training cycles can become a bottleneck. This is an addressable concern though, as the developer can configure Finch to extract the dataset less frequently after the model stabilizes. For future work, training the models could be easily distributed to machines that are not running the target system, in order to prevent resource saturation and affect the service quality. Predicting the optimal configuration parameters was relatively quick: performing the prediction took between 100 and 200 milliseconds. All SLA violations were automatically resolved in less than 10 minutes.

4.2 Experiment 2: Performance overhead

Given the results found in Experiment 1, the training algorithm is observed to become a bottleneck as the dataset grows. To investigate this bottleneck further and to closely observe Finch's resource usage (**RQ2**), we collected a larger dataset, with a total of 28,147 rows over 10 hours of execution. The strategy for the configuration parameters used was the same as in the first experiment.

The first training cycle, the one that performs an expensive grid search, took 29 minutes to find the optimal models and their hyperparameters for five SLI models. The subsequent training executions already knew the best model (enabling Finch to just fit the model to the data), and took 4 minutes to train and around 250 milliseconds to make predictions.

Experiment 2 results: Golang's `pprof` was used to perform a thorough profiling of both CPU-time and heap usage of the target system when using Finch. While acting as a passive co-pilot (no training and no adaptations created/carried out) and monitoring alongside Prometheus, Finch's performance overhead over a 2-minute profiling windows averaged 5.9%. In the same 2-minute windows, Finch's MAPE-K loop, its main component, averaged 2.6% overhead. While running only its monitoring/analyzing loop, Finch incurred roughly 8.5% CPU overhead.

4.3 Experiment 3: Postgres configuration

Due to Postgres's large set of configuration parameters, it can be a challenging task to adapt Postgres to different workloads. For instance, for a certain type of query, properly configuring Postgres' *work_memory* variable can drastically improve its performance, whereas for other queries this parameter may be less important. In this example, we use Finch in the same target system from Experiment 1 and 2, but now instead of random throttling points, Finch tries to learn how to better configure the Postgres database behind the target system. Postgres 9.4 was used, and the monitored several additional configuration parameters:

- `Shared buffers`
- `Effective cache size`
- `Work memory`
- `Write ahead log buffers`
- `Checkpoint completion target`
- `Maintenance work memory`
- `Checkpoint segments`
- `Default statistics target`
- `Random page cost`

The SLAs were the same as in the previous experiments. However, the adaptive configuration file contains the previous configuration points and as well as configurations for the variables listed above. The workload simulator then ran on the target system.

Experiment 3 Results: The target system started with the default configuration for Postgres. Under heavy workloads, some of the SLAs were violated. After a two-hour training cycle, Finch triggered an adaptation to address these SLA violations. After predicting the best optimal configuration for the workload and carrying out the adaptation, the 99th percentile latency was reduced by 39.85%. This demonstrates that Finch can automatically identify configurations that improve SLA compliance. The comparison of pre- and post-adaptation latency can be seen in Figure 9.

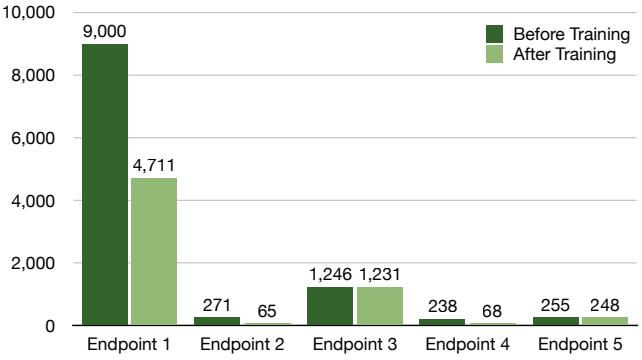

Figure 9: Experiment 3: 99th percentile latency of all endpoints before and after adaptation. Each item in the X axis is an endpoint affected by a configuration parameter. Y axis is the latency. The average latency reduction was 39.85%**.**

4.4 How much data is required?

A fundamental question for all ML-based approaches is how much data is needed to make accurate predictions (**RQ3**)? For both Experiments 1 and 3, Finch needed at least 1,000 rows in the dataset to reach a good cross validation accuracy. However, this could grow if we had larger configuration parameter spaces. At the same time, due

Table 1: Data from using Finch with artificially generated configuration parameters.

	Configuration precision	Average model accuracy	Dataset size (# of rows)	Training time	Prediction time
Run 1					
Initial	40%	N/A	N/A	N/A	N/A
Cycle #1	60%	71%	326	46 seconds	200 milliseconds
Cycle #2	100%	80%	1,082	1 min 20 seconds	117 milliseconds
Run 2					
Initial	40%	N/A	N/A	N/A	N/A
Cycle #1	60%	68%	374	51 seconds	165 milliseconds
Cycle #2	100%	80%	1,165	1 min 12 seconds	128 milliseconds
Run 3					
Initial	20%	N/A	N/A	N/A	N/A
Cycle #1	80%	87%	334	43 seconds	125 milliseconds
Cycle #2	100%	94%	1,125	1 min 7 seconds	119 milliseconds

to the incremental nature of Finch, this relatively limited number of samples shows that it is possible to make effective predictions even with limited training data. That said, the models trained in Finch will likely not generalize between target applications, so these data points must be collected on a per-system basis.

5 DISCUSSION

Finch represents a prototypical tool that has several limitations and while promising leaves much space for future improvement.

Threats to validity. Our initial prototypical evaluation sought to determine whether Finch could monitor, plan, and adapt a running system. The primary threat to the external validity of these results was that we only evaluated on a single system. In terms of construct validity, the system was also exercised under synthetic loads, which could be addressed by applying Finch to a deployed system (or a mirror of a deployed system). Our evaluation also focused on a small subset of Postgres configuration options. Additionally, the evaluation did not explicitly examine the time-series impact of the data that was collected. Evaluating the approach's ability to adapt configurations for real systems under real load remains future work, although our initial analysis does at least seem to point to the prototype heading in the right direction.

In terms of internal validity, gaining user experience defining SLAs and configuration parameters would be helpful to ensure that the Finch SLA and parameter schemas are expressive enough for users to effectively define their configuration and monitoring needs. Additionally, comparing to other self-adaptive approaches from research and industry (both in terms of developer overhead and runtime performance) would help to better situate Finch among related approaches.

Future work. The domain of the tool is currently limited to REST-based systems. This is not a limitation of the underlying data models or approach, but rather a reflection of our selection of Prometheus which specifically instruments REST endpoints. If we were to expand to another type of tracing or logging framework (such as Dtrace [6]) or by allowing developers to manually generate instrumentation events we could expand the breadth of the kinds of system events Finch could observe.

Increasing the range of systems we could observe would also cause us to increase the breadth of SLAs our system could support. Right now our SLA definitions only support evaluating throughput and latency which both naturally map to REST-based systems. Expanding to a richer set of measures would also increase the power of the tool (for example CPU, memory, IO, error rates, or perhaps even the cost function associated with cloud-based resources).

In some training cycles, grid search is automatically used in order to improve the quality of the accuracy. This special and costly training happens during the first few cycles, and when the accuracy starts dropping, usually because of change in the usage patterns. Because this is a computationally-expensive operation, this can negatively affect the target system by using too much compute power during the training operation. This performance issue could be reduced by distributing this training pipeline to other machines.

Additionally, before deploying Finch in practice, a more comprehensive interface would be required. In particular, the current prototype automatically deploys new configurations either by modifying the in-memory configuration or changing the configuration file and restarting the system appropriately. While this automatic process is works, it is understandable that an operator may want to examine and approve plans before they are deployed, or to explicitly specify when those configuration updates may take place.

6 CONCLUSIONS

In this paper we introduce Finch, a proof-of-concept tool for enabling self-adaptation in target systems without requiring complex architectural changes. Currently, the tooling for building self-adaptive systems is scarce and complex; our proposed approach provides initial evidence that a lightweight facility for enabling self-adapting configuration of non-autonomous REST-based systems is possible.

We show that Finch learns how to configure a target system after it ran alongside the system for a short training period. Finch observes system behaviours under workloads and evaluates whether

the system violates any of its Service Level Agreements. If any violations are encountered, Finch executes adaptations that change the target system's configuration; in our prototype study, these changes have been shown to successfully optimizes the system's performance. The success of the adaptation stems from the machine learning-based MAPE-K feedback loop that is injected into the target system. In summary, we feel that Finch is to provide a lightweight mechanism for enabling configuration self-adaptation in existing REST-based systems, incurs a performance overhead no higher than 8.5%, and is able to successfully recommend configurations that can resolve SLA violations. We hope that this lightweight approach can be applied in additional configuration-rich settings where varying runtime conditions would benefit from dynamic configurations and that Finch can provide initial evidence that such a system could have practical utility.

REFERENCES

[1] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2015), 13–23. https://doi.org/10.1109/seams.2015.10

[2] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 2015. Hogna: A patform for self-adaptive applications in cloud environments. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2015), 83–87. https://doi.org/10.1109/seams.2015.26

[3] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. 2007. Dynamic Placement of Virtual Machines for Managing SLA Violations. *International Symposium on Integrated Network Management (INM)* (7 2007), 119–128. https://doi.org/10.1109/inm.2007.374776

[4] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. 2013. *A Design Space for Self-Adaptive Systems.* 33–50. https://doi.org/10.1007/978-3-642-35813-5_2

[5] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems.* 48–70.

[6] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems.. In *USENIX Annual Technical Conference, General Track.* 15–28.

[7] Eddy Caron, Frédéric Desprez, and Adrian Muresan. 2011. Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. *Journal of Grid Computing* 9, 1 (November 2011), 49–64. https://doi.org/10.1007/s10723-010-9178-4

[8] Autonomic Computing. 2006. An architectural blueprint for autonomic computing. *IBM White Paper* 31 (2006).

[9] Datadog. 2018. Datadog. (2018). https://www.datadoghq.com/

[10] Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.). 2013. *Software Engineering for Self-Adaptive Systems II.* Lecture Notes in Computer Science, Vol. 7475. Berlin, Heidelberg. DOI: 10.1007/978-3-642-35813-5.

[11] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao. 2014. Architecting Self-Aware Software Systems. In *International Conference on Software Architecture (ICSA).* 91–94. https://doi.org/10.1109/WICSA.2014.18

[12] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2015. Software Engineering Meets Control Theory. *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (2015), 71–82. https://doi.org/10.1109/seams.2015.12

[13] Archana Sulochana Ganapathi. 2009. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning.* Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[14] Alan G. Ganek and Thomas A. Corbi. 2003. The dawning of the autonomic computing era. *IBM systems Journal* 42, 1 (2003), 5–18.

[15] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. *Workload Analysis and Demand Prediction of Enterprise Data Center Applications.* 171–180 pages. https://doi.org/10.1109/iiswc.2007.4362193

[16] Markus Hedwig, Simon Malkowski, and Dirk Neumann. 2010. Towards Autonomic Cost-Aware Allocation of Cloud Resources. In *International Conference on Information Systems (ICIS).*

[17] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. 2014. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience* 26, 12 (Aug. 2014), 2053–2078. https://doi.org/10.1002/cpe.3224

[18] Shay Horovitz and Yair Arian. 2018. Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning. In *International Conference on Future Internet of Things and Cloud (FiCloud).* 85–92. https://doi.org/10.1109/FiCloud.2018.00020

[19] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[20] Jóakim Von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. 2017. Modeling and Extracting Load Intensity Profiles. *Transactions on Autonomous and Adaptive Systems (TAAS)* 11, 4 (2017), 23. https://doi.org/10.1145/3019596

[21] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2017. The Case for Learned Index Structures. *arXiv:1712.01208 [cs]* (Dec. 2017). arXiv: 1712.01208.

[22] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. 2010. Efficient resource provisioning in compute clouds via VM multiplexing. In *International conference on Autonomic computing (ICAC).* 11–20. https://doi.org/10.1145/1809049.1809052

[23] Nagios. 2018. Nagios. (2018). https://www.nagios.org/

[24] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. *Self-Driving Database Management Systems.*

[25] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* GA, 333–348.

[26] Prometheus. 2018. (2018). https://prometheus.io/

[27] Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti, and Matteo Nardelli. 2020. Geo-distributed efficient deployment of containers with Kubernetes. *Computer Communications* 159 (2020), 161–174. https://doi.org/10.1016/j.comcom.2020.04.061

[28] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019. Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning. In *International Conference on Cloud Computing (CLOUD).* 329–338. https://doi.org/10.1109/CLOUD.2019.00061

[29] M. Salehie and L. Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS).

[30] Lucia Schuler, Somaya Jamil, and Niklas Kühl. 2021. AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments. In *International Symposium on Cluster, Cloud and Internet Computing (CCGrid).* 804–811. https://doi.org/10.1109/CCGrid51090.2021.00098

[31] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. 2004. Using Probabilistic Reasoning to Automate Software Tuning. *Proceedings of the joint international conference on Measurement and modeling of computer systems (SIGMETRICS)* 32, 1 (June 2004), 404–405. https://doi.org/10.1145/1012888.1005739