Enabling Configuration Self-Adaptation Using Machine Learning

Anonymous Author(s)

ABSTRACT

Due to advancements in distributed systems and the increasing industrial demands placed on these systems, distributed systems are comprised of multiple complex components (e.g databases and their replication infrastructure, caching components, proxies, and load balancers) each of which have their own complex configuration parameters that enable them to be tuned for given runtime requirements. Software Engineers must manually tinker with many of these configuration parameters that change the behaviour and/or structure of the system in order to achieve their system requirements. In many cases, static configuration settings might not meet certain demands in a given context and ad hoc modifications of these configuration parameters can trigger unexpected behaviours, which can have negative effects on the quality of the overall system.

In this work, we show the design and analysis of Finch; a tool that injects a machine learning based MAPE-K feedback loop to existing systems to automate how these configuration parameters are set. Finch configures and optimizes the system to meet service-level agreements in uncertain workloads and usage patterns. Rather than changing the core infrastructure of a system to fit the feedback loop, Finch asks the user to perform a small set of actions: instrumenting the code and configuration parameters, defining service-level objectives and agreements, and enabling programmatic changes to these configurations. As a result, Finch learns how to dynamically configure the system at runtime to self-adapt to its dynamic workloads.

We show how Finch can replace the trial-and-error engineering effort that otherwise would be spent manually optimizing a system's wide array of configuration parameters with an automated self-adaptive system.

1 INTRODUCTION

The industrial adoption of microservices 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 [16]. They noted that managing complex systems has become too costly, prone to error, and labour-intensive because pressured engineers make mistakes, increasing the potential of system outages with a concurrent impact on business. This has driven many researchers to study self-adaptive systems (e.g., [2, 13, 15, 20, 26, 28]); 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 [1, 12, 16].

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.

Building self-adaptive systems is a major engineering challenge [7]. Finch's proposal is to enable 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.

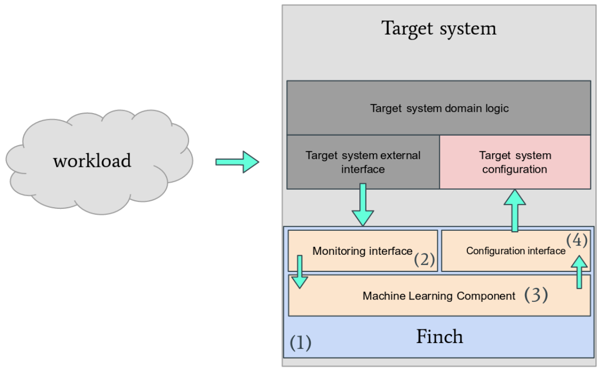
Target system
Target system domain logic
workload
Target system external interface
Target system configuration
Monitoring interface (2)
Configuration interface (4)
Machine Learning Component (3)
(1)
Finch

Figure 1: How finch integrates into a target system; (1) Finch is injected into the target system, (2) it Monitors and analyzes the target system's context, (3)Finch learns how to configure the target systems, (4) Interface executes configuration adaptation plans, adapting the target system.

One of Finch's main goals is to provide self-adaptive configuration support with minimal engineer effort. Finch uses ideas from self-adaptive systems, system observability, machine learning, and control theory to automatically asses the system's environment, predict the impact of changes that could potentially improve the system, and make these changes automatically.

My approach consists of providing mechanisms for injecting a control loop into an existing target system through an API for collecting relevant system metrics and configurations as the system executes. The user maps Service Level Agreements (SLAs) to a subset of these metrics, feed them into a machine learning component that is concurrently relearning the model while analyzing current event data which then predicts optimal configurations for the system for its given context. As a result, Finch provides 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:

- A methodology for assisting the development and evolution of self-adaptive systems, regardless of the presence of self-adaptability in the system's foundations. Such methodology is encapsulated in Finch.

- Demonstrating how minimal changes to the system can support this approach, and how Service-Level Agreements can be modeled and mapped to optimization objectives.
- A group of experiments to evaluate Finch's performance when integrated into a web service, demonstrating how Finch can learn how to configure it and improve its performance while incurring 8.5% of performance overhead.

Chapter 2 discusses past research in the space of self-adaptive systems and provides fundamental background for our approach. Chapter 3 outlines the design and usage of Finch, explaining the blend of ideas from different fields that lead to its principle design decisions. Chapter 4 describes Finch and its implementation. Chapter 5 presents the evaluation performed on Finch, followed by a discussion on limitations and future directions in Section 6.

2 RELATED WORK AND FOUNDATIONS

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

2.1 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 [3, 6, 10, 12, 21, 28]. Angelopoulos et al discussed the intersection between software engineering and control theory [14]. 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.

2.2 Machine learning in control theory

The applications of machine learning in control theoretical models have been discussed in [17], where the main idea is to take advantage of high performance of machine learning methods while using control theory to guarantee safety and controllability. Reinforcement learning [29] has similar goals to those in control theory, but with different approaches. Finch uses ideas from machine learning, reinforcement learning, and control theory to enable self-adaptability. Thus, instead of hard-coding configuration heuristics in a system, the control theory aspect of Finch (the MAPE-K loop) uses machine learning techniques to learn patterns in the target system and make fast predictions in order to create adaptation plans.

2.3 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.

Between 2007 and 2011, many techniques for forecasting workload and performance metrics using time series data have been realized [5, 9, 18, 19, 24]. 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.

Herbst et al. contributed with a survey of state-of-the-art forecasting approaches based on time series analysis and a decision-based technique to select the best forecasting method based on user-specified objectives [20]. They provided a useful technique to reliably forecast workload and performance metrics based on time series analysis, which is an important component in our tool. To enable self-adaptation in a system is to first understand the patterns in its context over time. In Finch we use these ideas and techniques to forecast workloads that, with the current configuration, could lead to SLA violations. Thus, enabling Finch to create an adaptation plan for this forecast and schedule the adaptation carry out.

2.4 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 [22], 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.

2.5 Self-adaptive systems

Cornel Barna et al proposed Hogna, a platform for deploying self-adaptive applications in cloud environments [4]. 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.

Finch can be used either for managing adaptive deployment schemes or optimizing finer-grained knobs of a system, for instance, optimizing configuration knobs of a Postgres instance used by a system in order to improve performance and prevent SLO violations—which is our case study in this paper. In addition to that, a minor difference between these two tools is how much is asked from the user; Hogna asks for a configuration file that describes the topology to be deployed, monitors to use, and more related settings, custom java classes to handle specific behaviors, and PXL file with the model description and a configuration file for Kalman filters, whereas Finch requires fewer actions from the user, while enabling self-adaptability and giving flexibility to the user: it can be used both for higher level tasks—deployment—and lower level tasks—self-tuning and self-configuring of smaller pieces of software.

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 [2]. 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.

2.6 Machine learning-enhanced software systems

In a recent work entitled The Case for Learned Index Structures, Kraska et. al. have demonstrated that machine learned models have the potential to provide significant benefits over state-of-the-art database indexes [23]. 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 DESIGN AND USAGE

To integrate Finch into the target system, the user has to do the following:

- Instrument the target system
- Identify relevant configuration parameters for Finch
- Define Service Level Agreements related to a subset of the instrumented data
- Allow configuration parameters to be changed programmatically

These 4 steps will create the necessary environment to enable self-adaptation in the target system, by enabling Finch to learn how to configure it. In the following sections we discuss in more details these steps and the design principles behind them.

3.1 Finch as a self-adaptation enabler

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 [7]. 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 [7]. 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 where the *injection* part of Finch enters. Rather than hard wiring the self-adaptation mechanisms inside the target system, Finch keeps it loosely-coupled. Upon integration into the target system, Finch acts as a co-pilot and starts collecting data related to the system's context, environment, and states, storing this data for future reference and model training. After a certain time, with learned models ready to make predictions, Finch starts analyzing event data. Guided by the internal feedback loop, it then carries out execution plans that aim to optimize the target system. The adaptation leads to more event data to be stored and analyzed, and the cycle repeats.

3.2 System's heuristics and configuration as a learning problem

We can define *learning problem* 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 generalized 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 scenario. This prediction can be used to either adapt to different scenarios that require different configurations or to prevent poor configurations.

To reiterate, machine learning is about learning to predict a certain behavior, based on what was experienced in the past. Thus, an important step when modelling a problem as a learning problem is the choice of observations used to train the system.

In this work's context, observations could be anything that relates to the system's behavior, performance, inputs, and outputs. For instance: Throughput, requests per second, latencies, and machine's resources usage (CPU, memory, IO) are some examples for the aforementioned context.

In Finch's case, it is important that Finch is provided with the necessary means to collect the best possible set of observations from the system and its environment. In order to model the system's heuristics and configuration as learning problem, Finch assumes that the system is properly instrumented.

3.3 Learnable patterns in systems context

Finch's ultimate strategy is to enable self-configuration in the system it is integrated to. Finch achieves this through learning exhibited patterns in the target system. In order to accomplish this, Finch needs the user to properly instrument the target system. Thus, a solid foundation in system instrumentation and observability comes a long way with Finch.

3.3.1 Observability and System Instrumentation. In order to observe the system thoroughly yet efficiently, Finch makes extensive use of modern observability and software instrumentation techniques. These techniques refer to code inserted in parts of the system's codebase to record its context. Function parameters, latencies, and time to execute a certain block of code are some values in a codebase that we can instrument. The purpose of collecting information from these pieces is, for instance, to help measure performance, assist debugging tasks, and find bottlenecks. In return, Finch greatly benefited from the recorded values throughout the system.

Any user who wants to integrate Finch into their system needs to carry out such instrumentation of their target system. Luckily, the software industry has been enforcing system instrumentation by providing many solutions, such as Dtrace [8], Prometheus [27], Nagios [25], and Datadog [11], so this requirement should not come across as an extra necessity, but a system requirement regardless of Finch's presence.

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.

Under Finch's layers, all monitoring is done using Prometheus. 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 API for it. As of now, this monitoring API consists of three methods; one for workload monitoring, another for latency monitoring, which takes the endpoint, the HTTP method, and the duration of the request, and a last one for configuration parameter monitoring, which after loading the file that contains the adaptive configuration parameters, it will put these values in memory and read from it.

Listing 1: Example of a normal adaptive configuration definition

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

Listing 2: Example of a custom adaptive configuration definition. Finch will call the procedure changeConfParam1 when it needs to change this specific configuration

```
[
  "custom_param_1": {
    "value": 1000,
    "valueType": "discrete",
    "values": [1000, 1],
    "isCustom": true,
    "adaptationMethod": "changeConfParam1"
  }
]
```

3.3.2 Defining adaptive configurations. Now that Finch collected data points that describe the performance and the behavior of the target system, the state of the configuration parameters at a given time should also be captured. Finch asks the user to create a file containing adaptive configurations. An adaptive configuration parameter can be a property, parameter, or variable whose value controls a certain aspect of a system or an algorithm. Finch tries to find the optimal value for this parameter in order to better configure the target system. The configurations can have a variety of values and depend highly on the kind of system the user is working with.

There are two types of configuration parameters that can be defined by the user; normal configuration parameter and custom configuration parameter. Both are declared in an adaptive configuration JSON file.

The normal configuration parameter holds its value both in memory and in the adaptive configuration file. When Finch predicts the optimal configuration, it will change the parameter value both in-memory and in the adaptive configuration file. It assumes that the target system is reading from one of those sources, and thus, the adaptation is easily carried. The normal configuration parameter is defined by passing the name of the configuration parameter, the current value, and the possible values or range values. 1 is an example of a normal configuration parameter definition.

However, there are scenarios where a simple in-memory or changing the value in a file is not enough to change the configuration of an aspect of a system. For instance, to change some of Postgres configuration parameters, one must change the value on its configuration file and then restart the Postgres instance. 2 is an example of a custom configuration parameter definition.

Finch's custom configuration parameters work in a way that assist this configuration changing process. When defining a custom configuration parameter, the user should also provide a procedure to be ran when an adaptation plan is carried that involves this configuration parameter. Upon adaptation, Finch calls this procedure, passing the appropriate parameters. Using the Postgres example, the user can provide a procedure that changes the Postgres configuration file using arguments being passed to it and restart the instance. Finch predicts the optimal configuration parameter, then calls this procedure passing the predicted value.

3.3.3 Resulting dataset. To construct the 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 the machine learning models could be trained on, which would later make predictions based on these features. These particular features were chosen in order to answer the following question: *Given the workload —e.g requests per seconds— 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 should capture the context of a system with respect to workload, instrumented metrics, and the values in configuration parameters.

3.4 Running Finch

Because there are many ways of using Finch as a library in a target system, there is no single right way to use. Here it is illustrated how it was used in a scenario where the target system is a HTTP REST service that uses Gorilla mux for its URL router and dispatcher, Viper for configuration management, Interpose for HTTP middleware, and Postgres as its main database.

We start by instantiating and initializing Finch where the target system does the same tasks in the codebase as shown in listing 3.

Listing 3: Initializing the target system and Finch

```go
func New(config *viper.Viper) (*Application, error) {
  Finch := finchgo.NewFinch()

  Finch.InitMonitoring()

  dsn := config.Get("dsn").(string)

  db, err := sqlx.Connect("postgres", dsn)
  if err != nil {
    return nil, err
  }

  // proceeds with the normal flow of the app

  return app, err
}
```

The highlighted lines are the lines that were added to the target system. In the target system's entry point, the original $funcNew(config)$ will be normally called, but now it will also configure and initialize Finch.

Upon initialization, Finch will try to find two JSON files, defined by the user, in the target system's root folder: one that describes each SLA for the target system, as shown in listing 4, and another one which defines the adaptive configuration parameters, as shown in the previous section.

The next step is to intercept the logging mechanism and use Finch's logging API to log the necessary data to train the dataset, analyze the context of the target system, and make predictions. This can be done by creating a logging middleware that uses Finch's logging API, and then adding it to the existing middleware used by the target system. With that done, all requests to this service will be analyzed by Finch so that it can perform its tasks.

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.

4 ARCHITECTURE AND IMPLEMENTATION

This chapter discuss in more details how the main components of Finch work. These main components were written using the programming language Go, and the machine learning components were written in Python.

4.1 Architecture overview

Finch's runtime spawns two main lightweight threads, which in Go is called Goroutine. These two Goroutines are two observer threads. The first one is responsible for periodically building the dataset. It will, periodically, extract all collected metrics from Prometheus through its HTTP API, parse this data, and save the dataset. Then, it calls the machine learning component to train the models using this

Listing 4: finch_sla.json describe each SLA in the target system

```
[
  {
    "sla": "<SLA description>",
    "endpoint": "<endpoint name>",
    "method": "PUT | POST | GET | DELETE",
    "metric": "latency | throughput",
    "threshold": "<Threshold number that defines what
        ,↪ 's an acceptable latency or throughput>",
    "agreement": "<Percentage number>"
  }
]
```

dataset, all models, scaler, and encoders—necessary components to make predictions—are persisted on disk.

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. The former has a significant impact on Finch's performance and is discussed in more details a later subsections.

4.2 The MAPE-K feedback loop

These two Goroutines running in loop compose the main abstraction of Finch: the MAPE-K feedback loop.

They communicate internally using Go's channels, which can be thought as pipes that connect concurrent Goroutines. The philosophy behind Go's channel is: share by communicating, not by sharing. Instead of sharing memory between threads, the sharing happens by sending messages between channels. Instead of handling concurrency by using mutex locks, it's favored by Go's community to use channels and messaging. The main loop spawns two other goroutines and control them with two different channels.

The goroutine that periodically builds the dataset starts an inner loop that extract all metrics from Prometheus and builds the necessary dataset for training every *DatasetBuilderFrequency* minutes as shown below, where this *DatasetBuilderFrequency* is configurable.

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.

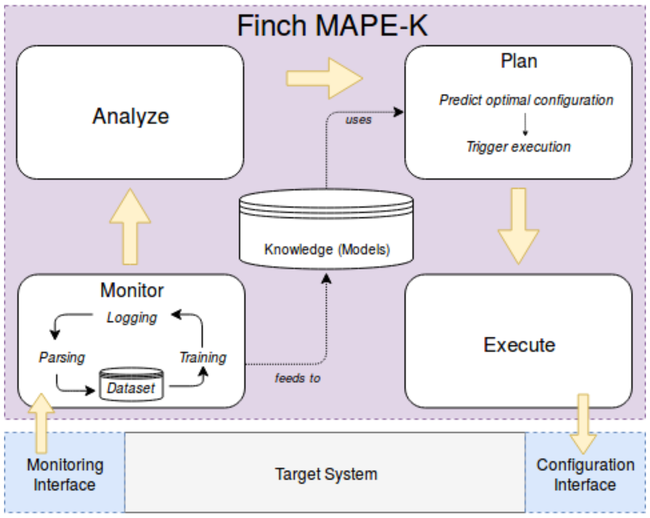

Figure 2: Finch's ML-based MAPE-K feedback loop design. The knowledge base is composed of learned models. The monitor and plan components use machine learning techniques in order to build the dataset and create adaptation plans.

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 3 illustrates this state machine.

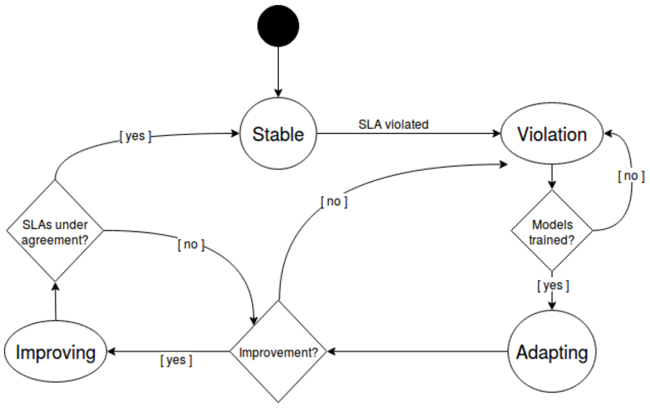

Figure 3: Finch's state machine to ensure progress in the target system

4.3 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 one is training the models, which includes basic standardization, normalization, grid search, and cross validation. The Second one is predicting the SLI, given the configuration parameters. After running these pipelines, the last step is finding the optimal set of configuration parameters.

4.3.1 Training pipeline. As mentioned before, Finch trains a model for each SLA indicator. To elaborate further on it; 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.

4.3.2 Learning how to learn: creating adaptive machine learning models. Since the dataset is very personalized with respect to the target system, there's no one model to rule them all, for example: we cannnot simply use logistic regression or a neural network with static hyperparameters. A personalized dataset means that it can have an arbitrary dimension (number of features) and size, it can have only continuous values or discrete values, or both. Finch cannot know this beforehand. Thus, to work with uncertain datasets, when training the models we must perform 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, 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 models-hyperparameters combination perform better with highly dimensional dataset, some perform better with well-balanced dataset, some are more resistant to outliers, sometimes the outliers is what you are trying to find.

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 it is something that cannot be avoided when choosing a model that will not overfit or underfit on dynamically generated datasets.

However, the grid search is only performed in two scenarios: first 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. The second scenario where grid search is performed is when the model's prediction performance starts to degrade, meaning that the dataset changed in some aspects, thus, needing to re-learn better a model and hyperparameters for the new dataset.

In this pipeline, it is considered models such as linear regression, ridge regression, lasso, support vector machines, and decision trees. 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 the majority of times: 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.

4.3.3 Predicting the optimal configuration. The user, when defining the adaptive configuration parameters, also defines the value range or the possible values, in case of discrete configuration parameters. 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, we 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 showed not to be very effective, and we devised an additional algorithm on top of this straightforward call to prediction method. This algorithm came as answer to cope with the following problem: in some cases, a configuration parameter does not overlap with respect to its effects on different SLAs, thus, 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 thinking about the other SLAs.

To overcome this problem and find the configuration that satisfies all SLAs or the majority of the SLAs, the algorithm created establishes some sort of consensus between the SLAs through a voting mechanism. To start it, it 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 big matrices (show this in experiment).

Then, for each SLA's predictions, it filters the configurations that satisfy the SLA plus a tolerance rate. Now we have, for each SLA, a set of configuration that is both diverse and satisfiable. In the last step, for each configuration parameter, in case of a discrete parameter, we pick the one with the highest occurrence, and in case of a continuous value, we compute the mean of the predicted values for this parameter.

In the end it outputs the set of configuration parameters that, based on past experience, is most likely to satisfy all SLAs or the majority of SLAs.

4.4 Passive and active training mode

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 sits passively collecting data and training models, but not making predictions and adaptation plans. Thus, during this period, it is necessary to collect a diverse dataset, Finch needs to know how to target systems respond to different configurations under different workloads. There are two ways to achieve that: passive and active training modes. These two options mode can be configured in Finch's configuration file. The passive mode just collects data and train models while the system is running, not intervening with the target system's natural flow. The active training mode, as a way to speed up the learning process, Finch will actively and frequently mutate the configuration parameters in the target system in order to fasten the process of gathering a more diverse dataset.

5 EVALUATION

To guide and evaluate our work, four research questions are used:

- **RQ1:** Can Finch learn the optimal or sub-optimal configuration parameters in a target system?
- **RQ2:** How much performance overhead is incurred by Finch?
- **RQ3:** How much training data is needed to make accurate plans?

In the next subsection we discuss the setup for the experiment of the tool and techniques previously discussed.

5.1 Experiment

We needed a production level web service exposed over a REST API. It is very hard to find these as open source, and the ones that we found were usually a simple proof of concept for a tool. They mostly had very few endpoints and a simple business logic, which is not realistic enough to test Finch. As a result, we developed a web service with this goal in mind. This system captured the most common points of complexity in web services, which are:

- A backend component that holds all the core logic of the application and containerized with Docker.
- Multiple HTTP endpoints served over REST API. In our scenario, these endpoints are subject to a set of Service Level Agreements measured using APDEX. For instance, *endpoint_A_POST* is an HTTP POST endpoint for the service A, and $X\%$ of all requests to it should not take more than Yms to respond.
- Another Docker container holding a Postgres database

At last, after developing this target system, small modifications on it were made in order to integrate Finch into it, such as basic monitoring and providing SLAs file, adaptive configuration files, and adaptation methods.

5.1.1 Initial training phase with workload simulation. We found that, to make accurate and useful predictions, Finch needed a dataset of reasonable size. However, my web service was not a system in production, and the only way to collect the mentioned dataset was through workload simulation. We accomplished this by mimicking realistic user cases for the system. For instance, a user can browse shopping items, add and remove items arbitrarily to a shopping cart, and finalize the 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.

At the end, Finch ran alongside the target system for a while, collecting data and learning the system's patterns.

5.1.2 Experiment 1: configuration-controlled throttling. This first experiment intended to answer the following question: *Can Finch infer the optimal configuration parameters without being explicitly programmed?*

To validate and answer this initial question, we needed not to focus on implementation details of the actual configuration changing, for example, gracefully handling Docker's containers and Postgresql restarts.

To achieve this, we created a script that randomly (and temporarily) generates throttling points in the target system, these blocking points block the flow in the code for either B_i milliseconds or $(\frac{1}{B_i}) * 10000$ milliseconds for each throttling point $B \in 1 \ldots i$. That means that a throttling will block the flow 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 flow for 1000ms or 1s. In a inversely proportional throttle point, it will block the flow for 10ms. 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 throttling point.

These B_i values are now our artificial configuration parameters that affect the performance of the target system. Thus, if Finch can perform its workflow—monitor, analyze, predict, and execute the adaptation plan—and correctly create an adaptation plan and configure the parameters in such a way that the performance will be either optimal or sub-optimal, again, *without being explicitly programmed for this task*, then we validate the architecture of Finch is achieving its goals and that predicting real configuration parameters is a matter of dataset quality, and thus, time to learn correctly.

We ran 3 different sets of random artificially generated configuration parameters and studied Finch's performance on them. We focused on 3 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?
- How accurate is the model?

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 $\sum_{i=1}^{n}(y_{true} - \overline{y_{true}})^2$.

5.1.3 Experiment 1 results. All tests were ran on a personal Dell laptop running Ubuntu 14.04 with 4 Intel Core i7-5500U CPU @ 2.40GHz and 16 GB of memory ram. The machine learning code makes uses of parallelism on all 4 cores when training the models and predicting.

Each training cycle equals 1 hour, the target system had 5 configuration parameters. From these experiments it is possible to see that after the first cycle Finch learns to find 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.

Training cycle can become a bottleneck if we keep growing the dataset for many days, however this is an easy bottleneck to overcome, as the user can configure Finch to extract the dataset less frequently after it gets a stable model accuracy. 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 is surprisingly fast, performing the prediction around 100 milliseconds and 200 milliseconds, even with the algorithm to find the best optimal configuration by performing multiple exhaustive predictions.

For all three runs of the experiment, after the second training cycle, it took between 5 to 10 minutes for the target system stabilizes all its SLA violations.

5.1.4 *Experiment 2: performance overhead evaluation.*

Given the results found in experiment 1, the training algorithm is observed to become a bottleneck as the dataset gets bigger. To investigate this bottleneck further and to closely observe Finch's resouce usage, we collected a bigger dataset, with a total of 28147 rows and for 10 hours. The strategy for the configuration parameters used were the same as in the first experiment; random throttling points in the target system's code.

The first training cycle, the one that performs an expensive grid search, took 29 minutes to find the optimal models and their hyperparameters for 5 SLI models. The subsequent training already knew the best model(so it just fitted the model onto the data), and took 4 minutes to train and 268 milliseconds to predict.

5.1.5 *How much performance overhead is incurred by Finch and Prometheus.*

Golang's pprof was used to perform a thorough profiling of both CPU-time and heap 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 window was 90 milliseconds out of 1530 milliseconds (5.88%). In this 2-minute window, Finch's MAPE-K loop, its main component, took 40 milliseconds out of 1530 milliseconds (2.61%), as shown in figure **??**. Thus, while running only its monitoring/analyzing loop, Finch incurred roughly 8.5% CPU overhead. This answers the second research question (**RQ2**)

5.1.6 *Experiment 3: finding the optimal configuration for Postgres.*

It is known that, because of the fact that Postgres contains a very big set of configuration parameters, it is a challenging task to adapt Postgres to different scenarios. For instance, for a certain type of query, properly configuring Postgres' *work_memory* variable can drastically improve its performance. There are many cases like this one, however, it would not be productive to discuss each one of them.

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 configuration parameters were:

- Shared buffers
- Effective cache size
- Work memory
- Write ahead log (WAL) buffers
- Checkpoint completion target
- Maintenance work memory
- Checkpoint segmnets
- Default statistics target
- Random page cost

Listing 5: Example of one of the adaptive configurations used in experiment 3

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

The SLAs were the same as in the previous experiments. However, the adaptive configuration file is different, as it contains all previously cited Postgres parameters and in each one of them, it defines the configuration parameter as a custom one, pointing it to the appropriate file that contains a procedure that Finch should run when adapting this configuration parameter, as shown in listing 5.

The adaptive method in this experiment, called *configurePG*, takes the predicted optimal configuration and perform the following actions:

- Loads the current Postgres configuration file.
- Changes the predicted parameters to their respective predicted values and persists it to disk.
- Using Docker remote API, it creates a new Postgres container, which loads the new configuration file, this Postgres instance points to the same volume as the current running Postgres instance.
- Tells the service what is the new Postgres instance's ip:port.
- Kills the previous Postgres Docker container

This way, all changes to Postgres will be effective because of the full restart of the instance.

5.1.7 *Results.*

In this experiment, the target system started with the default configuration for Postgres. After a while, given the heavy workload, some of the SLAs were violated. However, that happened before Finch learned its models, so nothing could be done before the training. After the training cycle, which took 2 hours, Finch triggered an adaptation, since the target system was in a state of SLA violation. After predicting the best optimal configuration for that scenario and carrying out the adaptation, the 99th percentile latency was reduced by 39.85%, the comparison of before and after the adaptation can be seen in figure 4. Thus, answering the research question 1 (**RQ1**).

5.1.8 *How much data is needed to make accurate predictions?*

The research question 3 (**RQ3**) touches a question commonly asked in the machine learning community: how much data is needed to make accurate predictions?

The answer to this question is: it depends on the properties of the dataset, such as size (how many rows), dimension (how many features), and overall quality of the dataset. For both experiments 1 and 3, it was needed at least 1000 rows in the dataset to reach a good cross validation accuracy. However, this could change if we had many more configuration parameters.

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

Run 1	Configuration precision	Models average accuracy	Dataset size (# of rows)	Training time	Prediction time
Initial	40%	N.A	N.A	N.A	N.A
Cycle #1	60%	71%	326	46 seconds	200 milliseconds
Cycle #2	100%	80%	1082	1 min 20 seconds	117 milliseconds
Run 2	Configuration precision	Models average accuracy	Dataset size (# of rows)	Training time	Prediction time
Initial	40%	N.A	N.A	N.A	N.A
Cycle #1	60%	68%	374	51 seconds	165 milliseconds
Cycle #2	100%	80%	1165	1 min 12 seconds	128 milliseconds
Run 3	Configuration precision	Models average accuracy	Dataset size (# of rows)	Training time	Prediction time
Initial	20%	N.A	N.A	N.A	N.A
Cycle #1	80%	87%	334	43 seconds	125 milliseconds
Cycle #2	100%	94%	1125	1 min 7 seconds	119 milliseconds

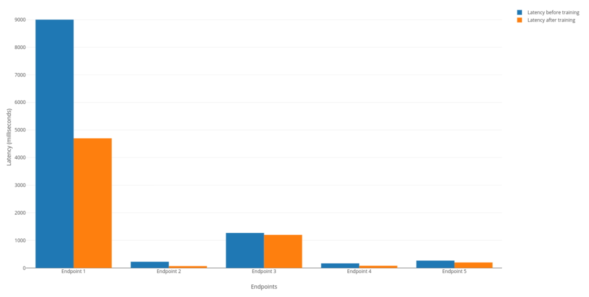

Figure 4: Experiment 3: 99th percentile latency of all endpoints before and after adaptation. Each item in the X axis is an endpoint affected by a configuration parameter. Y axis is the latency. The average latency reduction was 39.85%**.**

This brings up an important takeaway point from this work: the empirical knowledge collected from running Finch in one target system will not properly transfer to running Finch in a different target system with different contexts and components. Even thought that is the case, Finch was designed to handle this uncertainty, as its training pipeline performs an extensive grid search to find the best model for a given observed dataset.

6 DISCUSSION AND FUTURE WORK

As of now, Finch is in its initial version and it is highly experimental. Thus, it has some limitations that should be addressed by future improvements.

The main additional overhead that comes with Finch is due to using Prometheus for storing observed data. To address the Prometheus performance overhead, a simple time-series database to store its logged events is enough for Finch. Such a database could be implemented from scratch as a simple log storage, or by using TimescaleDB, a simple implementation of time-series constructs on top of Postgres.

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 very 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 solved by distributing this training pipeline to other machines or simply delegating the training to an external service will solve the problem.

To address the lack of a control interface, a web UI that talks directly to Finch over a REST API should be implemented. Such UI should provide control over Finch 's workflow.

7 CONCLUSIONS

This work presented Finch; a tool for enabling self-adaptation in systems without requiring complex architectural changes. Currently, the tooling for building self-adaptive systems is scarce and complex; we propose Finch in support of building tools that enable configuration self-adaptation in non-autonomous systems.

We show that Finch learns how to properly configure a target system after it ran alongside the system for a short training period. Once Finch is imported into a system, it starts collecting data on the context of the system. When the workload pattern changes or the performance of the system degrades, Finch executes adaptations that change the system's configurations, which 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. As a result, besides enabling configuration self-adaptation in systems, Finch also addresses the complexity of configuring software systems that have a high degree of uncertainty in their environment. As the goal of Finch is to make integration to systems easier, it also provides a small and concise API, and incurs a performance overhead no higher than 8.5%.

REFERENCES

[1] [n. d.]. Using probabilistic reasoning to automate software tuning. ([n. d.]). http://ftp.deas.harvard.edu/techreports/tr-08-04.pdf
[2] 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 Tieying Zhang Andrew Pavlo, Gustavo Angulo. 2017. *Self-Driving Database Management Systems.*
[3] Paolo Arcaini, Elvinia Riccobene, and Patrizia Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2015), 13–23. https://doi.org/10.1109/seams.2015.10
[4] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 2015. Hogna: a Platform for Self-Adaptive Applications in Cloud Environments. *2015 IEEE/ACM*

10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (2015), 83–87. https://doi.org/10.1109/seams.2015.26

[5] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. 2007. *Dynamic Placement of Virtual Machines for Managing SLA Violations*. 2007 10th IFIP/IEEE International Symposium on Integrated Network Management. 119–128 pages. https://doi.org/10.1109/inm.2007.374776

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

[7] 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*. Springer, 48–70.

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

[9] 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 (11 2011), 49–64. https://doi.org/10.1007/s10723-010-9178-4

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

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

[12] 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. Springer Berlin Heidelberg, Berlin, Heidelberg. http://link.springer.com/10.1007/978-3-642-35813-5 DOI: 10.1007/978-3-642-35813-5.

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

[14] 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. *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2015), 71–82. https://doi.org/10.1109/seams.2015.12

[15] Archana Sulochana Ganapathi. 2009. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-181.html

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

[17] Jeremy Gillula. 2010. Fusing Machine Learning & Control Theory. (3 November 2010). http://chess.eecs.berkeley.edu/pubs/714.html Presented at weekly ActionWebs meeting.

[18] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. *Workload Analysis and Demand Prediction of Enterprise Data Center Applications*. 2007 IEEE 10th International Symposium on Workload Characterization. 171–180 pages. https://doi.org/10.1109/iiswc.2007.4362193

[19] Simon; Hedwig, Markus; Malkowski and Dirk Neumann. 2010. TOWARDS AUTONOMIC COST-AWARE ALLOCATION OF CLOUD RESOURCES. (2010). https://aisel.aisnet.org/icis2010_submissions/180

[20] 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

[21] 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

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

[23] 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). http://arxiv.org/abs/1712.01208 arXiv: 1712.01208.

[24] Xiaoqiao Meng, Canturk Isci, Jeffrey Kephart, Li Zhang, Eric Bouillet, and Dimitrios Pendarakis. 2010. *Efficient resource provisioning in compute clouds via VM multiplexing*. 11–20 pages. https://doi.org/10.1145/1809049.1809052

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

[26] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 333–348. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/porter

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

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

[29] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). MIT Press, Cambridge, MA, USA.