Title TBD

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

Due to the advancement in distributed systems and the increasing industrial demands, software systems contain multiple components with complex interactions, e.g databases and their replication, caching components, proxies and load balancers, application instances and their complex configuration parameters. The engineers in a project must thinker with many configuration parameters that change the behavior and/or structure of the system, this can cause many problems that affect the quality of the service. In other words, dealing with high dimensionality is both cognitively demanding and risky for the project.

In this work we show the design and analysis of a pragmatic machine learning based tool that aims to assist the engineering of systems that can: 1) Monitor themselves, 2) Forecast workloads and performance metrics and 3) Change themselves in run-time by self-configuring and adapting for a specific scenario. After the integration of this tool with a system, it should be able to answer the question: given that we have many configuration parameters, how can we change them in order to optimize a certain metric for a given predicted workload?

We show that it can decrease the risk of changing systems' configurations in run-time and decrease the engineering effort that otherwise would be spent manually optimizing parameters, usually following a trial-and-error approach.

1 INTRODUCTION

The industrial adoption of microservices has led to increasingly complex configuration schemes that are commonly fine-tuned by engineers manually. Ganek and Corbi (2003) discussed the need for autonomic computing to handle the complexity of managing software systems. They noted that managing complex systems has become too costly, prone to error, and labor-intensive, because people under such pressure make mistakes, increasing the potential of system outages with a concurrent impact on business [**?**]. This has driven many researchers to study self-adaptive systems over the years [**? ? ? ? ? ?**]; however, the software industry still lacks practical tools to provide self-adaptation mechanisms to their systems. Thus, most of the configuring and tuning of the systems are performed manually, often in run-time, which is known to be a very time consuming and risky practice [**? ? ?**].

In this work we present Finch, an accessible tool that allows engineers to integrate machine learning into systems, it delegates the configuring or tuning of a system to a learned model, rather requiring engineers to perform these operations manually or through manually tuned heuristics.

One of our main goals is to provide the aforementioned support with minimal effort required from the engineers. Finch uses ideas from 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.

Our approach consists of providing an API to collect relevant systems' metrics and configurations that represent the state of the system in relation to time. We map Service Level Agreements (SLAs) to a subset of these metrics, feed them into a machine learning component that is concurrently re-learning the model while analyzing and predicting the workload and the optimal configurations. As a result it provides adaptation plans that can be both 1) automatically executed, allowing the system to have self-adaptive capabilities, and 2) interpretable, allowing engineers to know the impact of a change in the configuration space before it is deployed.

The tool not only provides reactive adaptations—adaptations that happen only when a constraint is violated—but also provides ahead-of-time adaptation plans that can be executed before a violation happens. This is achieved by predicting the context of the observed system.

In summary, our main contributions are:

- Providing a methodology to assist 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 the tool described in this work.
- Showing how to make the minimal necessary changes to the system, and how to model SLAs/SLOs and map them to the optimization objectives. These tasks being the development cost incurred by the engineers.
- Presenting a case study that shows how a software system's response time, throughput, and usage were improved by *A%*, *B%*, and *C%* respectively after the integration of Finch.

The rest of this paper is structured as the following: in Section 2 we discuss some past research in the space of self-adaptive systems and provide fundamental background. In Section 3 we outline our approach, explaining the blend of ideas from different fields. In Section 4 we describe internal details and design decisions of our implementation. In Section 5 we present our case study followed by a discussion and future directions in Section 6. Finally, we conclude our findings with Section 7.

2 RELATED WORK AND FOUNDATIONS

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 [**? ? ? ? ? ?**]. Angelopoulos et al discussed

the intersection between software engineering and control theory [**?**]. 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. These works were shown to be invaluable to the development of Finch.

2.2 Machine learning in control theory

The applications of machine learning in control theoretical models have been discussed in [**?**], where the main idea is to take advantage of high performance of machine learning methods while using control theory to guarantee safety. Reinforcement learning [**?**] 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.

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 the years of 2007 and 2011, many techniques for forecasting workload and performance metrics using time series data have been realized [**? ? ? ? ?**]. With these forecasts, they provided methodologies for virtual machine allocation in data centers. 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 [**?**]. 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.

2.4 Workload modelling

Another important piece in Finch is how we simulate the workload intensity to initially train the adaptation model. To have an accurate workload, we need to model it as close as possible to a real-world workload. Herbst et al. presented the Descartes Load Intensity Model [**?**], 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 workloads and simulate these workloads to train the adaptation model.

2.5 Self-adaptive systems

In 2015, Cornel Barna et al proposed Hogna, a platform to deploy self-adaptive applications in cloud environment [**?**]. It provides a framework that abstracts the deployment details, for example: spinning-off and managing instances on Amazon EC2 or OpenStack, the user then can 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 the rigorous formal modelling to a machine learning model that will be fed with the identified goals (SLOs), instrumented data, and identified knobs 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.

All previously cited works approach the adaptation problem with a reactive strategy: 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 adapation, 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. Similar to one of our goals, 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, 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. [**?**]

This worked 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 experiment this idea in a HTTP rest API's 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

Before diving into the implementation details, it is interesting to understand each conceptual piece in our approach. From a high-level view, our approach's workflow to turn a target system into a self-adaptive system (machine-learning enhanced system?) consists of:

- Instrumenting the target system
- Detecting relevant configuration knobs
- Defining Service Level Agreements for a subset of the instrumented data
- Allowing configuration knobs to be changed programmatically

These 4 steps (see figure 1) will create the necessary environment for the target system and Finch to implement a MAPE-K loop. In the following subsections we discuss the rationale behind these steps.

3.1 System's heuristics and configuration as a learning problem

A learning problem can be simply put as a set of observations comprised of input data and output data and some unknown relationship between the two, and the goal of a learning system is to learn a generalized mapping between input and output data such 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 we can learn a model that can find patterns between the system's observations and the system's knobs—configuration knobs or the values used in heuristics that control the system's behaviors—enabling the system to predict the best set of knobs for a given specific observed scenario.

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

In this 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 and memory.

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

3.2 Learnable patterns in systems context

3.2.1 System instrumentation. The first step in our approach is to observe the system's behavior and context, this is an important technique to understand how a running system behaves; by having data that tells us how a system behaves under the combination of different workloads and different configuration knobs, we can see that interesting patterns emerge from this data.

Software instrumentation refers to pieces of code inserted in some parts of the system's codebase to record its context, for instance: the values of function parameters, latencies, and time to execute a certain block of code. The purpose of these pieces of code is to help measuring performance, debugging, finding bottlenecks, and similar tasks.

Luckily, the software industry has been enforcing system instrumentation by providing many solutions for software instrumentation and monitoring such as Dtrace [**?**], Prometheus [**?**], Nagios [**?**], and Datadog [**?**].

Instrumentation is heavily used in industry also to detect Service Level Objectives (SLOs) violation and to perform resource management—two tasks that are crucial to Finch.

Under Finch's layers all monitoring is performed using Prometheus. Prometheus is a pull-based monitoring tool and time-series database. A normal concern is: what is the overhead incurred by a pull-based monitoring tool? The answer to this question is straightforward: the overhead is negligible. Unlike monitoring tools like Nagios, which frequently executes check scripts, it 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 metrics of that target over HTTP and has no other execution overhead that would be pull-related.

Another reason why Prometheus overhead is low is that Prometheus is not an event-based system, it works by regularly collecting aggregated time series data that represents the current state of a given set of metrics, not the underlying events that led to the generation of those metrics. Thus, it does not send a message to Prometheus server for each request as it is handled, it simply count up those requests in memory—causing no monitoring overhead or traffic—then Prometheus pulls this data every few configurable seconds, returning the current counter value and its timestamp.

In this first step, what we ask from the users is to choose parts of their systems to instrument, more specifically, instrument parts that are closely related to performance and resource usage. What has showed to work for us—and for many practitioners in industry–is to instrument: CPU usage, CPU idleness, memory usage, I/O writes per second, I/O reads per second, workload—it can be, for instance, requests per second—and latencies of desired services, for instance: the latency of each web service endpoint.

3.2.2 Performance percentages and percentiles over average. A common mistake both in industry and in the research community is measuring performance—specially latency or response time—using averages. Averages hide outliers and it is usually very skewed. To better illustrate this problem, consider having 100 requests per minute to your target system, if 80 of these requests take 200ms, which is relatively fast, and 20 of these requests take 10000ms, or 10 seconds, your average is 2.1 seconds, which is an acceptable

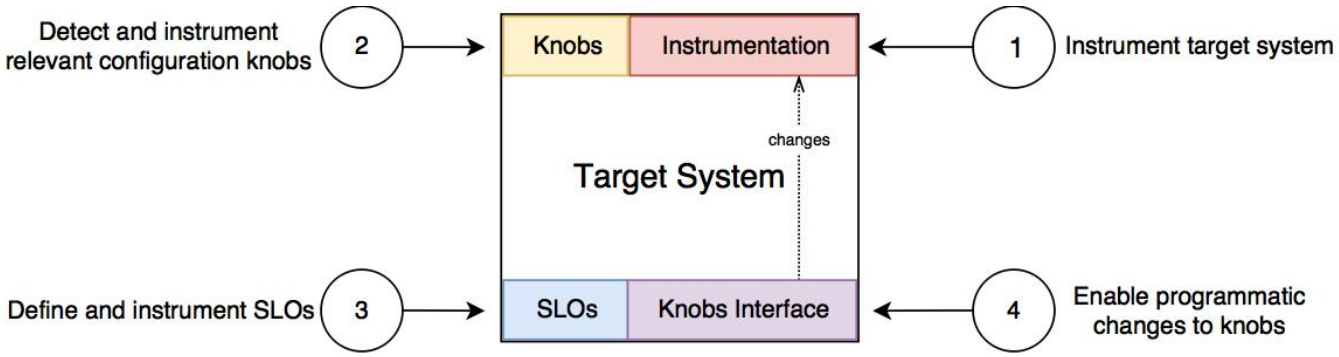

Figure 1: Finch's approach from a user perspective. The four necessary steps to change the target system into a self-adaptive system

latency. However, it hides the fact that 20% of these requests are taking an unacceptable amount of time to be served.

Rather than averaging performance metrics, Finch takes three more reliable approaches to measure performance metrics:

- Percentiles, such as 99th and 90th percentiles , in order to capture and understand outliers. This way we can understand upper bounds and uncover more silent bugs.
- Agreements on a percentage of requests being served under an agreed threshold. For example, an agreement could be: *95% of the POST requests to endpoint A will be served under 200ms*. This is a good strategy because rather than focusing on averaging—which is not reliable—we focus on serving well a big portion of the requests. This synergizes well with histograms and buckets.
- APDEX [**?**] as an extension on agreements.

APDEX is an industry standard that gives a score of satisfaction based on the latency or response time of requests. It is calculated by: $APDEX_T = \frac{S+\frac{T}{2}}{R}$. Where T is a selected threshold, S is the number of satisfied requests, or requests that take less than T to be served, T is the tolerated requests, or requests that take between T and $T * 4$ to be served, and R is the total number of requests. A request is considered frustrated if it takes more than $T * 4$ to be served.

To compare APDEX to averaging latencies, think of two scenarios:

- 60% of the requests take $200ms$, 20% of the requests take $10ms$, and 20% of the requests take 10 seconds. The average of latencies in this case is $2.1s$, which gives you a false sense of confidence. The $APDEX_{2s}$ is 80%, which is considered low.
- 1 request takes 5 minutes, 10 requests take 200ms. Here we have a case of anomalous latency. The average is 27 seconds, which is very high, whereas $APDEX_{2s}$ is 91%, telling us that, given our selected threshold, the situation is not bad, rather, it was just an anomalous case.

Finch uses APDEX agreements and 99th and 90th percentiles as features and targets when training the predictions models.

3.2.3 Configuration knobs instrumentation. Now that we have data points that describe the performance and behavior of our system, we have to capture the current state of the configuration knobs; it can range from a variety of configurations, and it is highly dependent on the kind of system you are dealing with. Here we define a configuration knob as a property, parameter, or variable that holds a value that controls a certain aspect of a system or algorithm.

3.2.4 Defining Service Level Agreements. Having collected the instrumented data—data that represents the snapshot of both the configuration knobs and the performance at a given time—we have to point which of these instrumented data points are under service level agreements. Here we call these features Service Level Indicators (SLIs), they are a subset of the set of instrumented metrics that must satisfy a desired objetive, the SLA.

For instance, given that we instrumented the latency of the endpoint A of our web service, it is agreed that 95% of the requests to this endpoint should have latency below $200ms$. This is properly captured by Finch's API.

3.2.5 Resulting dataset. We have collected four sets of features: metrics related to the system's context—performance and behavior metrics—configuration knobs, the workload, and service level indicators that must satisfy their objectives. The goal for the collection of this data is to build a dataset that will be used later to train models that will be used to make prediction based on these three sets of features, and the models will be trained to answer the following question: *given a workload—e.g requests per seconds— and the performance metrics of the system serving these requests, what is the optimal set of configuration knobs that will prevent service level objective violations?*

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 time captured by a timestamp.

The following matrix summarizes how the dataset is organized:

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

Where:

(1) n is the number of collected samples
(2) t_ϕ is the timestamp of the $\phi - th$ 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 knob in timestamp t_ϕ. c ranges from 1 to t, 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 knobs.

3.3 Machine learning architecture

Given the previously defined dataset, Finch trains many different models, each one targeting a different task. We have 2 ML pipelines: training the models, which also involves cross validating each model in isolation, and validating the prediction workflow: the workflow that triggers the interaction between the many models.

3.3.1 Training the models. There are two main classes of learned models:

(1) SLI models: these models predict, given the system's context, knobs values included, the SLI value. For instance, given the context, it outputs the latency of a certain endpoint. We will train a model for each collected SLI.
(2) Knob models: these models predict, given the system's context, other knobs *not* included, one SLI s included, and others SLIs that are not s excluded, the optimal knob value. For instance, it outputs the optimal knob k to achieve a certain latency x. We will train a model for each combination of SLI and knob collected.

The training of the different models consists of slicing the original dataset to fit the models' needs. For instance, when we want to predict the latency of endpoint A, we must say that latency A is the target, or y, of the model, and the corpus, or X, is the rest of the dataset *minus* the latency A column. Then we train the model passing this specific X and y.

The algorithms 1 and 2 show with more details the process of training the models.

Note that before training the final models, we cross validate the model with multiple—currently 100—splits of the dataset and performing the cross validation with 80/20 ratio: train with 80% of the dataset, test with 20% of the dataset, 100 times. This way we prevent cases of overfitting the model to the data.

3.3.2 Validating the prediction workflow. The cross validation in algorithms 1 and 2 can only validate the accuracy of the each model in isolation. This is not enough to tell if the prediction workflow will work properly. For instance, one of our goals is to predict the best set of configuration knobs for a set of violated SLOs, so that after executing the adaptation plan, their respective SLIs will improve, preventing the SLO to be violated.

To achieve this level of validation, we perform bidirectional predictions: we grab a sample from the dataset that contains many SLO violations, predict the optimal knobs that could prevent the violations, then, using the predicted knobs, we predict the SLIs and see if it indeed improved in comparison to the original SLIs from the bad sample. Algorithm 3 shows with more details how this procedure works.

Algorithm 1: Trains a model for each SLI and return a list of models

```
1  TrainSLIModels (Dataset)
     inputs : A dataset that contains the system's context
     output : n SLI models, where n is the number of collected
              SLIs
2    models ← [] ;
3    foreach SLI S ∈ Dataset do
4      y ← Dataset[S];
5      X ← Dataset \ Dataset[S];
6      regressor ← learning_algorithm(...);
7      score ← Cross_validation(regressor, X, y);
8      if (score ≥ 70%) then
9        models[S] ← regressor.fit(X, y);
10     end
11   end
12   return models;
```

Algorithm 2: Trains a model for each Knob x SLI combination and return a 2D array containing the models

```
1  TrainKnobModels (Dataset)
     inputs : A dataset that contains the system's context
     output : i * j SLI ⇒ Knob models, where i is the number of
              collected SLIs and j is the number of collected
              knobs
2    models ← [] ;
3    foreach SLI S ∈ Dataset do
4      foreach Knob K ∈ Dataset do
5        y ← Dataset[K];
6        X ← Dataset \ Dataset[Knobs] ∧ Dataset[SLIs ≠ S];
7        regressor ← learning_algorithm(...);
8        score ← Cross_validation(regressor, X, y);
9        if (score ≥ 70%) then
10         models[S, K] ← regressor.fit(X, y);
11       end
12     end
13   end
14   return models;
```

3.4 Control-theoretic approach to self-adaptation

Here we talk first about the architecture with regards to MAPE-k and how we generate adaptation plans. Then we talk about creating adaptation plans for future workload, by predicting the workload using time-series data.

3.4.1 Ahead-of-time adaptation. As we showed before, the dataset contains the timestamp for each sample, meaning that we can take advantage of time-series analysis techniques to predict, given a certain time in the future, what will be the workload. Having the

Algorithm 3: Validates the prediction workflow by making bidirectional predictions. First we predict the best set of knobs given a bad scenario, then we predict the SLIs considering the predicted set of knobs

```
1  ValidatePredictionWorkflow (Dataset)
2    test ← Dataset[0] ;
3    X ← Dataset[1 :];
4    SLI_models = TrainSLIModels(X);
5    knob_models = TrainKnobModels(X);
6    knob_predictions ← [] ;
     /* In this first foreach, we predict the optimal knob values
        for a sample with SLO violations                          */
7    foreach SLI S ∈ Dataset do
8      SLI_test ← test \ test[Knobs] ∧ test[SLIs ≠ S];
9      foreach Knob K ∈ Dataset do
10       pred ← knob_models[S, K].predict(SLI_test)
           knob_predictions[S, K] ← pred ;
11     end
12   end
13   final_knobs ← most predicted set of knobs;
     /* Next we predict SLIs given the predicted set of knobs     */
14   inverse_test ← test ;
15   foreach Knob K ∈ Dataset do
16     inverse_test[K] ← final_knobs[K] ;
17   end
18   foreach SLI S ∈ Dataset do
19     knob_test ← inverse_test \ inverse_test[S] ;
20     Modify each remaining SLI value in knob_test to a value
         below its SLO ;
21     predicted_sli ← SLI_models[S].predict(knob_test) ;
22     Validate that predicted_sli is below its SLO ;
23   end
```

future workload in hands, we can use this data point to predict, like we did in the previous section, the best set of knobs based on the predicted SLIs.

The key trick is that this plan for the optimal set of knobs will not be immediately carried out—the time for this adaptation has not come yet. It will, however, be stored in a table that contains all scheduled plans, and an observer will be watching this table and carrying out the plans a few minutes before the prediction for a given workload.

Should it fail to predict the correct workload, it will, once again, predict the set of optimal knobs for the current workload and immediately carry out the plan.

This procedure will create adaptation plans for a short time window, executing adaptations before they are needed, thus, before a violation occurs.

3.4.2 Online training. Here we talk about the constant retraining of the models, since we will have data coming in all the data, so lots of corrections to be made.

3.5 Propagating adaptation plans

4 IMPLEMENTATION

5 EVALUATION

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

- **RQ1:** Can self-adaptation by learned models lead to more stable and faster software systems, reducing the need to manually configure and tune?
- **RQ2:** How much instrumentation, SLO mapping, and configuration mapping is required to integrate the tool in a system?
- **RQ3:** How much performance overhead is incurred by using this tool?
- **RQ4:** What metrics and features have more impact on the tool's performance?

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

5.1 Experiment

To evaluate the tool and the techniques here discussed, we used a toy web application as the target system of Finch. Although simple, this web application capture most commonly seen complexities, here is a brief description of it:

- It consists of the backend component that holds all the core logic of the application.
- The application's services are exposed through many HTTP endpoints that follow a REsT API approach. In our scenario, these endpoints are subject to a set of Service Level Objectives. For instance, *endpoint_A_POST* is a HTTP POST endpoint for the service *A*, and it should not take more than *Xms* to respond.
- This backend is containerized with Docker.
- Another Docker container holds a Postgres database used by the application.
- At last, another Docker container holds Finch itself, including the vendors' tools used by it—Prometheus, for instance.

5.1.1 The challenging task of configuring a Postgres database. It is known that, because of the fact that Postgres contains a very big set of configurations knobs, 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 discusss each one of them.

Postgres 9.4 was used, and the configuration knobs considered 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

In order to show that the Postgres default configuration is not the optimal one, we ran a workload simulation, which is similar to HTTP load testers, but instead of stressing a single endpoint with a very large number of requests per second, this workload simulation simulates different user workflows with varying intensities. For instance, 200 requests per second calling different sequences of endpoints: call endpoint *A*, wait for response, call *B*, wait response, and so on, until it completes a workflow that would be common for a user of this service.

We ran the simulation—which tested the web application, and not the Postgres directly—for a certain period of time using two different sets of Postgres configuration knobs. By running this simulation we ended up with the dataset discussed in Section 3. The results confirmed both our beliefs and what is experienced in industry: The latencies of the endpoints were drastically affected by the different combinations of configuration knobs; different types of workflow and different intensities required different set of configurations in order to keep latency low, a non-optimal configuration for a given scenario could cause the violation of an SLO.

6 DISCUSSION AND FUTURE WORK

7 CONCLUSIONS