Enabling Configuration Self-Adaptation Using Machine Learning

by

Rodrigo Araujo

B.Sc. Computer Science, Universidade Salvador, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August 2018

© Rodrigo Araujo, 2018
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Enabling Configuration Self-Adaptation Using Machine Learning**

submitted by Rodrigo Araujo in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

**Examinining Committee:**

Reid Holmes, Computer Science  
*Supervisor*

Mike Gelbart, Computer Science  
*Supervisory Committee Member*
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, I 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.

I 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.
Lay Summary

Software systems are increasingly more complex. Manually configuring these software systems is an error prone tasks that software engineers have to perform frequently. Modern distributed software systems usually need to be re-configured frequently to adapt to different contexts. In this thesis I present Finch, a tool that enables software systems to configure itself by learning the system’s optimal configurations, allowing the it to adapt to different contexts in order to keep the quality of service and decrease the need to manually configure these systems.
Preface

The work presented in this thesis was conducted in the Software Practices Lab at the University of British Columbia, Vancouver, British Columbia.

Reid Holmes was the supervisory author on this project and was involved throughout the lifecycle of this project.
# Table of Contents

Abstract ......................................................... iii  
Lay Summary ................................................. iv  
Preface ......................................................... v  
Table of Contents ........................................... vi  
List of Tables ................................................. ix  
List of Figures ............................................... x  
Glossary ....................................................... xii  
Acknowledgments ............................................. xiii  

1 Introduction ................................................. 1  

2 Background and Related Work ............................... 4  
   2.1 Control theory in software engineering ................. 4  
   2.2 Machine learning in control theory ..................... 5  
   2.3 Time series analysis ..................................... 5  
   2.4 Workload modelling .................................... 6  
   2.5 Self-adaptive systems .................................. 6  
   2.6 Machine learning-enhanced software systems ........... 9
3 Design and Usage .................................................. 10
  3.1 Machine learning based MAPE-K feedback loop .......... 10
    3.1.1 Monitor ................................................. 11
    3.1.2 Analyzer ............................................... 11
    3.1.3 Planner ............................................... 11
    3.1.4 Executor ............................................... 12
  3.2 Finch as a self-adaptation enabler ............................. 12
  3.3 System’s heuristics and configuration as a learning problem . 14
  3.4 Learnable patterns in system’s context ......................... 14
    3.4.1 Observability and System Instrumentation ............... 15
    3.4.2 SLA, SLO, and SLI .................................. 16
    3.4.3 Measuring performance metrics ......................... 17
    3.4.4 Defining adaptive configurations ....................... 18
    3.4.5 Resulting dataset ................................... 20
  3.5 Running Finch ............................................... 21

4 Architecture and Implementation ................................. 24
  4.1 Architecture overview ...................................... 24
  4.2 The MAPE-K feedback loop .................................. 25
  4.3 Machine learning architecture ............................... 28
    4.3.1 Training pipeline .................................... 29
    4.3.2 Predicting the optimal configuration .................. 30
  4.4 Passive and active training mode ............................ 32
    4.4.1 Passive training mode ................................ 34
    4.4.2 Active training mode ................................ 34
  4.5 Controlling adaptations ..................................... 34
    4.5.1 Detecting improvement after carrying out an adaptation 34
  4.6 Finch configuration ......................................... 35
    4.6.1 SLI adjustment effect ................................ 37

5 Evaluation ........................................................ 38
  5.1 Experiment .................................................... 38
    5.1.1 Initial training phase with workload simulation ....... 39
List of Tables

Table 5.1  Data from using Finch with artificially generated configuration parameters ........................................ 41
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>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.</td>
<td>2</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Classical feedback loop model that inspired the MAPE-K feedback loop</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Types of adaptations in response to different stimuli, from João Sousa [31]</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Classical view of the autonomic computing manager created by IBM</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>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.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Finch’s state machine to ensure progress in the target system</td>
<td>28</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Slope that shows improvement in the target system</td>
<td>36</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>CPU profile showing the performance overhead incurred by Prometheus</td>
<td>42</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>CPU profile showing the performance overhead incurred Finch’s MAPE Loop</td>
<td>43</td>
</tr>
</tbody>
</table>
Figure 5.3  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%.
Glossary

This glossary uses the handy \texttt{acroynym} package to automatically maintain the glossary. It uses the package’s \texttt{printonlyused} option to include only those acronyms explicitly referenced in the \LaTeX source.

API Application Programming Interface

MAPE-K Monitor, Analyze, Predict, and Execute over a Knowledge base

ML Machine Learning

SLA Service Level Agreement

SLI Service Level Indicator

SLO Service Level Objective
Acknowledgments

I would like to thank my supervisor Reid Holmes, for giving me a great opportunity to pursue my intellectual curiosity and for supporting me along the way. This work would not have been possible without his guidance.

I would like to thank Mike Gelbart, not only for being the second reader of this work, but also for his genuine curiosity and desire to understand and help me to make this work better.

Also, I must thank my family and friends, for believing in me and providing me a strong foundation.

And last but not least, I must thank my partner, Gulipek Candan, for all the support I could’ve asked for.
Chapter 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., [13, 15, 20, 27, 28, 30]); 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 I 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 [6]. 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.

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
Figure 1.1: 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.

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 thesis of this research is that the configuration of a system can be delegated to a self-adaptive machine-learning based system to adapt to different workload patterns. I achieved this by exploring a design that enables self-adaptability without
incurring intrusive architectural changes in a target system, requiring the user to only carry out tasks that are common in the software industry: Defining service-level agreements and interfaces to the configuration, and enabling observability in the system by using minimal instrumentation.

The main contributions of this thesis 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.
Chapter 2

Background and Related Work

Finch draws ideas from many different, although overlapping, fields. Here I give a brief overview of these ideas and how they relate to Finch.

2.1 Control theory in software engineering

Control theory is an interdisciplinary branch of engineering and mathematics that studies how dynamic systems behave and how it changes with respect to modified control inputs through feedback mechanisms, one of the main goals of control theory is to devise techniques and models to make systems achieve certain goals by controlling the system’s input.

The ideas in control theory have been widely adopted in the software engineering research community, with special attention to the Monitor, Analyze, Predict, and Execute over a Knowledge base (MAPE-K) feedback loop, which proved to be a powerful tool to build self-adaptive systems [3, 7, 10, 12, 22, 30]. 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

Machine learning is a set of principles, algorithms, and techniques, strongly rooted in statistics, that provides systems the ability to automatically learn, improve, and perform tasks without being explicitly programmed, all based on the data that it is fed to.

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 [32] 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 theoretical 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.

Like one of the main goals of machine learning—enabling a system to perform a task without being explicitly programmed—Finch learns how to find the optimal or sub-optimal set of configuration parameters without being explicitly programmed.

2.3 Time series analysis

Time series data refers to a series of data points organized in a temporal order. 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, this is highly used in Finch.

Between 2007 and 2011, many techniques for forecasting workload and performance metrics using time series data have been realized [5, 9, 18, 19, 25]. With these forecasts, they 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 the end goal of this work.

Finch’s main abstraction for data storing and dataset creation is time-series
data. All patterns observed by Finch in the target system are with respect to the time of the observation.

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 accurate workloads, it is needed to model them as closely as possible to real-world workloads. Herbst et al. presented the Descartes Load Intensity Model [23], 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

Self-adaptation describes the ability of a system to change some aspects of itself in runtime rather than in design time. Generally, in a self-adaptive system, we have the autonomic manager and the managed element. The autonomic manager is always monitoring and analyzing the managed element, and adapting it in response to changes in its context in order to achieve a higher goal, such as better performance, fault-tolerance, or efficiency.

The figure 2.3 shows the MAPE-K (Model, Analyze, Plan, Execute, over a Knowledge base) model used to engineer self-adaptive systems. At its core, it has the feedback loop, which interacts with the target system or, as it is originally called, the managed element. This interaction happens through its sensors and effectors. This model is strongly related to the classical view of feedback loops in Control Theory (figure 2.1) where a controller is always monitoring a process in order to change some of its aspects in response to changes.

In the MAPE-K model, the monitor, analyzer, planner, and executor work together, using a knowledge base, toward a common goal: the adaptation of some aspect of the target system.

In 2013, João Sousa properly mapped the types of possible adaptations in response to different stimuli [31], as shown in the figure 2.2. Finch focuses on runtime structure adaptation in response to: QoS goals, system metrics, parameters,
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 injects a MAPE-K loop, abstracting the formal modeling 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 op-
timizing 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 SLA violations. 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 behaviours, 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.

Andrew Pavlo et. al. presented Peloton, a database system designed for autonomous operation [27]. 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 my 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, I am embedding a subset of these ideas in a tool that aim to be integrated in any arbitrarily chosen software system.

### 2.6 Machine learning-enhanced software systems

In a 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 [24]. This research showed that by replacing manually tuned heuristics with learned models enabled it to outperform cache-optimized B-Trees by up to 70%.

I draw much 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 Application Programming Interface (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.
Chapter 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 I discuss in more details these steps and the design principles behind them.

3.1 Machine learning based MAPE-K feedback loop

In Section 2 I discussed about self-adaptive systems, feedback loops, such as MAPE-K, and how they relate to Finch. However, the actual implementation of Finch uses a slight variation of the aforementioned feedback loop, in which the MAPE-K loop is based on a machine learning model.

The main difference between the original and the machine learning based MAPE-K loop stems from the Monitor (M) and the Knowledge (K) components. In the classical MAPE-K loop, the Knowledge of the system is the data collected by the Monitor component. However, in Finch’s implementation of this loop, the Monitor component uses the monitored data to train the machine learning models. As a result, the model built by the Monitoring component is shared across other
components. In other words; *the learned models are the K in Finch’s MAPE-K implementation.*

The planner component (P) uses these models to create an adaptation plan by predicting the optimal configuration for the target system. Just like the classical MAPE-K, Finch’s feedback loop then executes the adaptation plan through the interface it shares with the system, and continues the loop. Figure 3.1 illustrates Finch MAPE feedback loop design.

Following sub-sections explain each component of Finch’s MAPE-K feedback loop in more detail.

### 3.1.1 Monitor

Finch includes an API for monitoring the target system’s context. Currently it only supports semantics for HTTP endpoints, but extending it to support different monitoring semantics is fairly simple.

During the monitoring, Finch periodically extracts, parses and builds a dataset containing the context of the system (more details on how it is done can be found in the Architecture Section 4). It then uses machine learning to train models that are able to predict user-defined aspects of the target system, such as the optimal or the sub-optimal configuration and how it will affect the quality of the service.

This dataset grows incrementally over time, and the models are retrained every time the dataset is incremented. As a result, Finch can capture emerging patterns in the target system and improve the accuracy of the models.

### 3.1.2 Analyzer

The Analyzer component frequently checks the current state of Finch and the target system. Depending on the improvements or violations, this component then triggers new adaptation plans.

### 3.1.3 Planner

The Planner, when triggered by the analyzer component, makes use of the current knowledge base (trained models) and predicts the optimal configuration for that specific context that the system is in.
When the adaptation plan is created, the Planner then calls the Executor.

### 3.1.4 Executor

The Executor, as the name suggests, executes the adaptation plans created by the Planner. The plans can be as simple as changing in-memory or in-file configuration parameters, or more elaborate, such as managing Docker containers. The latter requires the user to define custom adaptation methods.

When another iteration of the cycle begins, the Monitor will gather data related to the most recent adaptation. This collection will be used when training new models. The analyzer, in the meantime, will watch for improvements or deterioration, and will trigger a new adaptation plan depending on its observations. The Planner, in return will use the collected data to create new adaptation plans, which will be executed by the Executor.

### 3.2 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 [6]. Given this definition, I 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 [6]. 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
Figure 3.1: 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.

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.
3.3 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 modeling 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.

3.4 Learnable patterns in system’s 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.

### 3.4.1 Observability and System Instrumentation

The first step in my approach was to observe the system’s behavior and context, which gave me insight on the system’s characteristics. By collecting data on different combination of workloads and configuration parameters, I have observed interesting patterns emerge from the data.

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 [29], Nagios [26], 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. A normal concern is the overhead incurred by a pull-based monitoring tool. However, the response to this concern is straightforward: The overhead is negligible. 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.
Another reason why Prometheus has low overhead is, it is not an event-based system. Prometheus regularly collects data on the aggregated time series, which represents the current state of the given metrics, and not on the underlying events that led to the generation of those metrics. Moreover, the system using Prometheus does not send updates to Prometheus server for each handled request. The system simply counts up the requests in memory, causing no monitoring overhead or traffic. Then Prometheus pulls this data every few seconds (which can be configured), returning the current counter value and its timestamp.

### 3.4.2 SLA, SLO, and SLI

The recent emergence of the DevOps culture has put key concepts and techniques from Observability and Systems Monitoring on a spotlight. Finch is built upon some of these concepts, such as Service-Level Agreements, Service-Level Objectives, and Service-Level Indicators.

These 3 terms are used interchangeably sometimes. So to avoid confusion, below are the proper definitions for each:

**Service Level Indicator (SLI)**

A quantitative measure on the target system’s current service quality levels. An example that is heavily used here is latency of the system’s endpoints. It could also be the throughput, the error rate, or the availability.

**Service Level Objective (SLO)**

A target value or a goal to be achieved by a given SLI.

For example, suppose our service has an endpoint $A$, the latency being measured in this endpoint is our SLI. Our objective is for it to be under 200 milliseconds. Thus, our SLO is $SLI \leq SLO$, where $SLO = 200ms$. In this case, the SLO serves as an upper bound on our observed SLI.

**Service Level Agreement (SLA)**

A rather formal contract between the service provider and its users. This contract encapsulates the SLOs and SLIs, and the consequences of its violation.
To build on top of the previous SLI and SLO values; if our SLO is to serve endpoint A in at most 200 ms, then the SLA can dictate that 95% of all requests should be served in accordance with the SLO.

### 3.4.3 Measuring performance metrics

A common mistake made in industry and in the research community is measuring performance —especially latency or response time— using averages. Averages hide outliers and are usually very skewed. To better illustrate this problem, let’s consider the following scenario: Our target system receives 100 requests per minute, 80 of which take 200ms to serve, which is relatively fast. On the other hand, the remaining 20 requests take 10000ms (10 seconds). If we assess the performance of the system using the average, we’ll evaluate the latency as 2.1 seconds, which is an acceptable value. However, this value hides the fact that 20% of our requests are taking 10 seconds, which is an unacceptable latency value.

Rather than taking the average of performance metrics, Finch takes three different approaches to measure performance metrics, which are more reliable than taking average:

- Measuring percentiles, such as 99th and 90th, in order to capture outliers. This way we can understand upper bounds and uncover more silent failures of the performance metrics.
- Using SLAs defined by the user to assess the performance metrics. For example, an agreement could require 95% of the POST requests to endpoint A to be served under 200ms. This is a good strategy because rather than taking the average of the POST requests, we focus on how well we served a big portion of the requests.

APDEX is an industry standard that gives a score of satisfaction based on the latency or response time of requests. It is calculated by: 

\[
APDEX_T = \frac{S + Tol}{R},
\]

where \(T\) is a selected threshold, \(S\) is the number of satisfied requests, or requests that take less than \(T\) to be served, and \(R\) is the total number of requests. \(Tol\) is the tolerated requests, or requests that take between \(T\) and \(T \times 4\) to be served, and \(R\) is the total number of requests. A request is considered frustrated if it takes more than \(T \times 4\) to
be served.

To compare APDEX score to taking the average, 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%. This $APDEX_{2s}$ score indicates the threshold we set, the situation is not bad, we just had an anomalous case.

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

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 (which I will talk about in the next sections), it will put these values in memory and read from it.

### 3.4.4 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.
Listing 3.1: Example of a normal adaptive configuration definition

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

Listing 3.2: 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"
  }
}
```

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. 3.1 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. 3.2 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.

### 3.4.5 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. The reason why I choose to use these particular features is 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:

\[
\begin{align*}
t_1 \ W_1 \ M_1, \ M_2, \ldots \ M_i, & \ \ k_1, \ k_2, \ldots \ k_l, \ SLI_1, \ SLI_2, \ldots \ SLI_{\delta_i}, \\
t_2 \ W_2 \ M_1, \ M_2, \ldots \ M_i, & \ \ k_1, \ k_2, \ldots \ k_l, \ SLI_1, \ SLI_2, \ldots \ SLI_{\delta_i}, \\
\vdots & \ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \ \vdots \\
t_n \ W_n \ M_1, \ M_2, \ldots \ M_i, & \ \ k_1, \ k_2, \ldots \ k_l, \ SLI_1, \ SLI_2, \ldots \ SLI_{\delta_i},
\end{align*}
\]

Where:

1. \( n \) is the number of collected samples
2. \( t_\phi \) is the timestamp of the \( \phi^{th} \) example
3. \( M_{j_\phi} \) is the \( j^{th} \) instrumented metric in timestamp \( t_\phi \). \( j \) ranges from 1 to \( i \), the
last instrumented metric.

4. $k_{cφ}$ 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. $SLI_{oφ}$ 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.

### 3.5 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 an 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.

I start by instantiating and initializing Finch where the target system does the same tasks in the codebase as shown in listing 3.3. 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 3.6, and another one which defines the adaptive configuration parameters, as shown in the previous section. An SLA description example can be found in 3.7.

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 is done by creating a logging middleware that uses Finch’s logging API, as shown in listing 3.4 and adding it to the middleware used by the target system, as shown in listing 3.5. 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
Listing 3.3: 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
    }
    cookieStoreSecret := config.Get("cookie_secret").(string)
    app := &Application{
        config: config,
        dsn: dsn,
        db: db,
        db.SetMaxIdleConns(10),
        sessionStore: sessions.NewCookieStore([]byte(cookieStoreSecret))
    }
    return app, err
}
```

Listing 3.4: Defining logging middleware for Finch’s logging

```go
func Log(Finch *finchgo.Finch) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            // Inject monitoring loop
            Finch.MonitorWorkload(r.Method, r.basePath)
            Finch.MonitorLatency(r)
            Finch.MonitorConfigurationParameters()
        })
    }
}
```

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.
Listing 3.5: Injecting Finch’s logging middleware into target system’s HTTP middleware

```go
func (app *Application) MiddlewareStruct() (*interpose.Middleware, error) {
    middle := interpose.New()
    middle.Use(middlewares.SetDB(app.db))
    middle.Use(middlewares.SetSessionStore(app.sessionStore))
    middle.Use(middlewares.Log(Finch))
    middle.UseHandler(app.Mux())
    return middle, nil
}
```

Listing 3.6: finch_sla.json describe each SLA in the target system

```json
[
  {
    "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>"
  }
]
```

Listing 3.7: Example of an SLA definition in finch_sla.json

```json
[
  {
    "sla": "95% of the POST requests to Users should be under 150ms",
    "endpoint": "/users",
    "method": "POST",
    "metric": "latency",
    "threshold": 150,
    "agreement": 95
  }
]
```
Chapter 4

Architecture and Implementation

This chapter discusses in more details how the main components of Finch work. Finch was built using the programming language Go, for its excellence with distributed and concurrent programming and its ease to construct reliable systems. Because Finch needs to provide reliability, speed, and a fast way to experiment with machine learning using an arbitrary dataset, using Python for machine learning and wrapping its service and providing a solid infrastructure using Go was the best approach for this job.

4.1 Architecture overview

Prometheus is used to store the observed metrics and their respective timestamps. Prometheus is a time series database that also offers a very complete monitoring and alerting API. Thus, instead of implementing a time-series store from scratch, I decided to use Prometheus underneath Finch, incurring an extra performance overhead. I discuss this performance overhead in chapter 5.

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 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 current SLO 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 at a later subsection.

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. This idea comes from Hoare’s Communicating Sequential Processes [21]. This turned out to be helpful when implementing something highly concurrent as the MAPE loop in Finch.

The main loop spawns two other goroutines and control them with two different channels, as shown in the code below.
func (f *Finch) StartMAPELoop() {
    ticker := time.NewTicker(MonitorFrequency * time.Second)
    quitWatcher := make(chan struct{})
    go f.MonitorAndAnalyzeContext(ticker, quitWatcher)

    datasetTicker := time.NewTicker(DatasetBuilderFrequency * time.Minute)
    datasetQuit := make(chan struct{})
    go f.buildDatasetPeriodically(datasetTicker, datasetQuit)
}

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.

func buildDatasetPeriodically(tickerBuilder time.Ticker, quitBuilder chan) {
    for {
        select {
            case <-tickerBuilder.C:
                f.DatasetBuilder(true, <dataset_range>)
            case <-quitBuilder:
                ticker.Stop()
                return
        }
    }
}

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.
```go
func (f *Finch) MonitorAndAnalyzeContext(ticker time.Ticker, quitWatcher chan) {
    SLAMetricsHistory := make(map[SLA][]float64)
    adaptationWasCarried := false
    isImproving := false
    for {
        select {
            case <-ticker.C:
                currentSLAMetrics := f.getSLAMetrics()
                SLAMetricsHistory = f.appendSLAMetricsHistory(SLAMetricsHistory, currentSLAMetrics)
                if f.checkForViolation(currentSLAMetrics) {
                    if adaptationWasCarried {
                        isImproving = f.checkForImprovement(SLAMetricsHistory)
                        if !isImproving {
                            adaptationWasCarried = false
                        }
                    }
                    if !adaptationWasCarried {
                        predictedOptimalKnobs := f.predictOptimalKnobs()
                        f.carryAdaptationPlan(predictedOptimalKnobs)
                        adaptationWasCarried = true
                    }
                }
            case <-quitWatcher:
                ticker.Stop()
                return
        }
    }
```

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 4.1 illustrates this state machine.
The interoperability between Golang and Python is done by giving Finch’s main components control over the machine learning component (written in Python). Because the machine learning component exposes a simple API, the main components use this API by executing bash commands.

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 Machine Learning (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\textsuperscript{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.

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 cannot 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.

Given all these options and uncertainties with respect to the collected dataset, grid search is a way to find the most well suited model and hyperparameters for the dataset being collected by the target system.
Grid search exhaustively considers all hyperparameter combinations and many different models, train one model per combination, and select the most best performing one. This adds a considerable 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 (section 5), which used a few variations of dataset structure, I have found that one machine learning model/technique worked best the majority of times: gradient boosting with decision trees.

Gradient boosting is not exactly a machine learning model, but rather a class of techniques called Ensemble, where instead of using a single model, we train many slightly different models with different portions of the dataset, and combine them in a variety of ways to achieve a higher performance. Unlike common and more simple ensemble techniques such as bagging, the trained models are not working independently, instead, they work sequentially in such a way that the following predictors learn from the mistakes of the previous predictors.

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

### 4.3.2 Predicting the optimal configuration

The user, when defining the adaptive configuration parameters, i.e: telling Finch which configuration parameters it should learn to configure, 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
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, and averageScore
2 models ← [];
3 scores ← [];
4 foreach SLI S ∈ Dataset do
5   y ← Dataset[S];
6   X ← Dataset \ Dataset[S];
   /* gridSearch will check if an initial grid search has already been performed, it not, it will exhaustively search for the optimal model and its hyperparameters for the collected dataset, if it already happened, it will use previously found best hyperparameters and train the model on the new data */
7   regressor ← gridSearch(X,y);
8   score ← Cross_validation(regressor,X,y);
9   scores.add(score);
10  models[S] ← regressor.fit(X,y);
11  averageScore ← computeAverage(scores);
12 end
13 return models, averageScore;

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 I created 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 af-
fect 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. Algorithm 2 illustrates this process.

### 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.
Algorithm 2: Predicts the optimal or sub-optimal set of configuration parameters for the target system based on past experience

1. **PredictOptimalConfiguration** *(possibleParameterValues, SLOs, models)*
   
   `parameterCombinations ← ComputeCartesianProduct(possibleParameterValues) ; /* candidateConfiguration will hold the best configurations for each SLI */`

2. `candidateConfigurations ← {} /* SLI is the indicator we want to predict. SLO is the objective value agreed on the SLA. */`

3. `foreach SLI, SLO ∈ SLOs.items() do /* parameterCombinations is a 2D array with the cartesian products of all parameter combinations. predictions gets also a 2D array with the predicted SLI value for each parameter combination */`
   
   `predictions ← models[SLI].predict(parameterCombinations) ;`

4. `foreach prediction ∈ predictions do /* if (prediction ≤ SLO + toleranceRate) then */`
   
   `index ← predictions.indexOf(prediction) ;`

5. `candidateConfigurations[SLI].add(parameterCombinations(index)) ;`

6. `end`

7. `end`

8. `optimalConfiguration ← {} ;`

9. `foreach parameter ∈ possibleParameterValues do /* optimalParameterValue ← getHighestOccurrenceValue(candidateConfigurations, parameter) ; */`
   
   `optimalConfiguration[parameter] ← optimalParameterValue ;`

10. `end`

11. `return optimalConfiguration ;`
4.4.1 Passive training mode

In passive training mode, Finch will just collect data and train models while the system is running, not intervening with the target system’s natural flow. Thus, this could mean taking a longer time for Finch to start making adaptation plans, since it needs a diverse dataset.

4.4.2 Active training mode

In 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. Thus, it will know sooner how the target system respond to different configuration settings. Running it alongside a workload simulator also works well, which is what I did to evaluate an experiment, discussed in section 5. Since this a risky move to make in production, Finch should be ran alongside a testing/staging environment.

4.5 Controlling adaptations

In order to control the adaptations and prevent Finch to create and carry out adaptation plans right after a previous adaptation plan was carried out, it needs to know that a previous plan has been carried out and whether there is any improvement in the target system. To know whether there is any improvement is to understand and take in consideration the propagation effect of the execution of an adaptation plan, in other words: before trying, again, to predict a better configuration, it is better to wait for a previous adaptation to take full effect.

4.5.1 Detecting improvement after carrying out an adaptation

An algorithm was devised in Finch to detect if the target system is correctly adapting and improving, i.e: making progress. This algorithm came as an answer to the question: how do we know, given an array of historical SLO values for the extracted SLAs, whether the target system is improving or not?

A visual explanation to this problem is the detection of a positive slope in a graph after the adaptation was carried out as shown in the figure 4.2. Upon
detection of improvement, no adaptation procedure should be executed.

In order to detect progress, this algorithm performs a sum of the successive differences of the SLO values in the array containing the past $n$ historical—including the current—SLO values. For instance, using the data in the figure 4.2, we have an array containing the SLO values (in percentage):

$$SLO_{HistoricalValues} = (100 90 80 70 60 60 80 85 95 100)$$

We then perform successive subtractions: for each index in the array $SLO_{HistoricalValues}$, except for the last index, we get the result of the subtraction of:

$$SLO_{HistoricalValues}[index + 1] - SLO_{HistoricalValues}[index]$$

Thus, we get the array:

$$SuccessiveDifferences = (-10 -10 -10 -10 10 20 5 10 5)$$

After that, we sum the values in this $SuccessiveDifferences$ array, if it yields a number greater than zero, it means the target system is improving, and we return it as a boolean value to prevent firing overlapping adaptation plans. However, we keep checking for improvements, in cases where it had an initial improvement, but for instance, the workload pattern changed, causing the the sum of the $SuccessiveDifferences$ to be negative, we can return a boolean as false, causing the adaptation procedure to run again.

The number $n$ of past historical values, including the current one, shown to be working well for $n = 5$, meaning that we get the current SLO value, plus other 4 that were recently extracted.

### 4.6 Finch configuration

Although Finch can learn how to configure a target system, unfortunately it cannot configure itself. Correctly configuring Finch is key to achieve a good performance, as its configuration affects how the learning happens. The main Finch configura-
Figure 4.2: Slope that shows improvement in the target system

The parameters are:

- **datasetBuilderFrequency**: defines how often the dataset is extracted and built.
- **datasetTimeRange**: defines the range, in time, that the dataset will be built with. For instance, if it is set as 5 minutes, it will build the dataset starting from 5 minutes ago to now.
- **trainingMode**: boolean that enables Finch to mutate configuration for learning purposes.
- **configurationMutationFrequency**: defines how often Finch should try new configurations in order to learn how it affects the target system. This mutation stops after the first training cycle.

Many different and interesting effects have emerged from within Finch, thus, selecting the right configuration for it has shown to drastically affect its performance.

In the evaluation section we go in more details what has shown to work best.
4.6.1 SLI adjustment effect

The experiments ran have shown an important insight: time to collect the datminipageaset is the most crucial part of Finch workflow. One of the reasons is that, in a short period of time, only the tasks that happened more frequently will have more predominance in the dataset, and thus, only configuration closely related to these tasks will have accurate predictions. Allowing the dataset to grow bigger and more diverse showed to be a good way to prevent this sort of bias.

Another reason, and an even more important one, is what we called SLI adjustment effect. Currently, our service-level indicator for a single service is its 99th percentile of the requests latency. One important characteristics of the 99th percentile is: since it focuses on a very small sample of the data that went above the threshold and violated an SLA, it takes time for the newly adapted configuration, now actually showing signs of improvements, to take effect on the service-level indicator. If we prematurely take this dataset, it will indicate as if the new (and correct) configuration is not a good configuration, as the 99th percentile is still violating the agreement, however, the longer we wait for the SLI to adjust to the new configuration, the better the SLI will reflect it, balancing the dataset and enabling the machine learning component to make more accurate predictions.

Simple term: the sli points don’t reflect any current improvement now, it will only show improvements after a certain time.
Chapter 5

Evaluation

To guide and evaluate this work, I used three research questions:

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

In the next section I discuss the experiment setup and the techniques used to test Finch.

5.1 Experiment

I 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 I 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, I 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 my 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 \( Y \) ms 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 instrumentation. These modifications were discussed in more details on chapter 3. With Finch running alongside the target system, all that was needed was time for Finch to learn how to configure the target system.

5.1.1 Initial training phase with workload simulation

I found that, to make accurate and useful predictions, Finch needed a a reasonably sized dataset, about 1000 rows at least. However, my web service was not a system in production, and the only way to collect the mentioned dataset was through workload simulation. I accomplished this by mimicking realistic user actions 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 I created ran these user actions 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 do so?*

To validate and answer this initial question, I chose to keep my focus on configuration items that Finch can change directly in memory. Focusing on Postgresql configurations would have meant writing an extra method to gracefully handling the Postgresql and Docker container restarts, and passing this method to Finch as the adaptation method to change the Postgres configurations. For my first experiment, I wanted to focus on the prediction accuracy and not overcoming integration challenges.
To achieve this, I created a script that randomly (and temporarily) generated throttling points in the target system. These throttling points block the flow in the code for either \( B \) milliseconds or \((\frac{1}{B^i})*10000\) milliseconds for each throttling point \( B \in 1 \ldots i \). In other words, a throttling point 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 an inversely proportional throttle point, it will block the flow for 1ms. 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 the artificial configuration parameters that affect the performance of the target system. Thus, if Finch successfully predicts and executes adaptation plans that bring the performance back to an optimal/sub-optimal level, without being explicitly programmed to do so, then it can be concluded that Finch is achieving its goals. It could also be concluded that the right architecture has been built and predicting real configuration parameters is a matter of dataset quality, and the right duration to learn correctly.

For each run, the script created random throttling points and it expected to receive the right combination. In the meantime, Finch had no access to the script’s expectations; the script analyzed Finch’s prediction at the end of each run to validate the prediction accuracy, which set each run’s success rate.

I ran this experiment three times. For the model accuracy, I used the coefficient of determination \( R^2 \) of the prediction, where \( R^2 = 1 - \frac{u}{v} \), \( u \) is the residual sum of the squares \( \sum_{i=1}^{n} (y_{true} - y_{pred})^2 \) and \( v \) is the regression sum of squares \( \sum_{i=1}^{n} (y_{true} - \bar{y}_{true})^2 \).

**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 use of parallelism on all 4 cores when training the models and predicting. Each training cycle had 1 hour duration, and the target system had 5 configuration parameters.

The results show that, after the second training cycle, Finch learns to find the
optimal set of configuration parameters, achieving 100% accuracy on its predictions (5 out of 5 configuration parameters correct). The results after the 2\textsuperscript{nd} cycle are not included in the results table as the accuracy stabilized in the 2\textsuperscript{nd} cycle and kept at 100% on the subsequent training cycles. Table 5.1 shows the results of these 3 experiments.

One of the possible bottlenecks of this experiment is the training cycle. 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.

Predicting the optimal configuration parameters was surprisingly fast. Even with the algorithm to find the best optimal configuration by performing multiple exhaustive predictions discussed in 3, it took between 100 and 200 milliseconds to make predictions.

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

Table 5.1: Data from using Finch with artificially generated configuration parameters

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

5.1.3 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 resource usage, I 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.

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/carry out) and monitoring alongside Prometheus, Finch’s performance overhead over a 2-minute profiling window was 90 milliseconds out of 1530 milliseconds (5.88%), as shown in figure 5.1. 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 5.2. Thus, while running only its monitoring/analyzing loop, Finch incurred roughly 8.5% CPU overhead. This answers the second research question (RQ2).

Figure 5.1: CPU profile showing the performance overhead incurred by Prometheus

5.1.4 Experiment 3: Finding the optimal configuration for Postgres

It is known that, because 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.

For this experiment, Postgres version 9.4 was used, and the monitored configuration parameters were as listed:

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

The SLAs were the same as in the previous experiments. However, the adaptive configuration file was different, as it contained the previously cited Postgres parameters. For each one of them, Finch defines the configuration parameter as a custom one, pointing it to the appropriate file that contains a procedure for Finch to run when adapting this configuration parameter, as shown in listing 5.1.

The adaptive method in this experiment, called `configurePG`, took the predicted optimal configuration and performed the following actions:

![CPU profile showing the performance overhead incurred Finch’s MAPE Loop](image)
• Load the current Postgres configuration file
• Change the predicted parameters to their respective predicted values and persist it to disk
• Using Docker remote API, create a new Postgres container, which loads the new configuration file. This Postgres instance points to the same volume as the current running Postgres instance.
• Tell the service what is the new Postgres instance’s ip:port.
• Kill the previous Postgres Docker container

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

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 5.3. Thus, answering the research question 1 (RQ1).

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. 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.

This brings up an important takeaway point from this work: the empirical knowledge collected from running Finch in one target system will not properly
Figure 5.3: 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%.

Listing 5.1: Example of one of the adaptive configurations used in experiment 3

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

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.
Chapter 6

Discussion

In this chapter I discuss the limitations, in both performance and usage, of the current version of Finch and the future work to address these limitations.

6.1 Limitations

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.

6.1.1 Prometheus performance overhead

The main additional overhead that comes with Finch is due to using Prometheus for storing observed data. This decision was made mostly because of the limited development time available, which rendered Prometheus as the most time feasible option. Although Prometheus has great performance metrics overall, it offers many features not needed by Finch, which created unnecessary performance overhead, as shown in section 5.

6.1.2 Ever growing dataset

By design, the dataset extraction and creation never stops in Finch. However, over time some parts of the data may become obsolete, not reflecting the current context and patterns of the target system: New configuration parameters could have been added, usage patterns might have changed, or more unexpected scenarios. To sus-
tain Finch’s success for the long haul, there needs to be a mechanism that regulates the dataset collection over time, ideally taking both size and the relevance to recently observed patterns into account. Such mechanism would ensure the accuracy of the models and lead to faster training cycles.

6.1.3 Local training pipeline

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.

6.1.4 Lack of control interface

All the information about the current state of Finch is currently shown through logs in a Docker container. It can be very time consuming to find the right information. Some Finch configurations could be changed through a GUI, which would not only make interaction with Finch easy, but also would show a more complete information about its processes.

6.2 Future work

In this section I discuss some future plans to improve Finch by addressing both to limitations and to additional features.

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.

To address the local training performance overhead, distributing this training pipeline to other machines or simply delegating the training to an external service will solve the problem. Currently the model training is performed by an external component, written in Python. Encapsulating this component in an external service
and calling this service from Finch is a simple and good solution.

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. To elaborate, this UI could allow the user to trigger a new training cycle, add/remove configuration parameters and trigger/cancel the creation of adaptation plans on the fly, all the while visualizing the impact of an adaptation plan on the SLAs and its SLIs even before executing it. All these features are realistic expectations about Finch’s growth, since the ML models predict the SLIs based on the configuration settings.

Another future feature for Finch is scheduled adaptations. Rather than adapting when an SLA is violated, Finch would be able to predict future workload (including usage pattern) based on historical patterns, before it happens. It can predict the optimal configuration beforehand, and schedule this adaptation for a certain period before the predicted future workload, preventing SLA violation altogether.
Chapter 7

Conclusions

This thesis presented Finch; a tool that was designed to enable self-adaptation in systems without requiring complex architectural changes. As of now, the tooling for building self-adaptive systems is very scarce and complex in the software industry. I propose Finch as a step in the direction to build tools that make it possible to enable configuration self-adaptation in non-autonomous systems.

I show that Finch learns how to properly configure a target system after it ran alongside the system for a while. 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%.

For future work, I discuss how the design of Finch can be improved in order to decrease its performance overhead, and make setting up Finch easier. The principle behind this work is that, rather than having hard-wired heuristics, systems should be able to adapt to different usage patterns by changing aspects of itself.
Bibliography


