SmartTuning

1 Introduction

The deployment of Cloud-Native applications (CNA), relies on placing a collection of small, independent, loosely coupled and, ideally, self-managed services into a cluster. This architectural style brings some advantages to the applications. Due to the high decoupling of application's components, it is possible to isolate failures to avoid the crash of the whole application. Moreover, this architectural style allows a decoupled evolution of the application by updating their components individually. However, the high decoupling in this architectural style makes difficult the application management, since there are a large number of distributed pieces of software that the engineers should handle during the application execution.

To mitigate some of challenges on the management of CNA, their services are mostly bundled up with several runtime abstraction layers to facilitate their execution or runtime management. Some examples of these layers are application servers (e.g., OpenLiberty), runtime environment (e.g., Java Virtual Machine), hardware isolation (e.g., containers and virtual machines), and cluster orchestrator (e.g., Kubernetes).

Runtime abstractions layers are general purpose with many parameters (knobs), that the application engineer adjusts to better satisfy the use cases of the application. Many of these knobs are related to data structures or infrastructure mechanisms, so their adjustment affects the application performance. An important issue regarding these knobs is the colossal number of possibilities that an engineer has to consider to improve the application execution.

Every layer added to an application increases exponentially the number of possible configurations that the engineer has available to experiment. Making matters worse, an optimal configuration for an application is mutable, it should changes according to the environment, such as fluctuations of number of users requests, saturation of the cluster, version of libraries being used, and so on. For tuning an application the engineer has to evaluate many different configurations to find one that extracts the most of the application performance for a given situation.

To tackle this challenge, we are proposing SmartTuning, a mechanism to automatically identify and apply optimal configurations onto CNA regarding the environment where the application is deployed. The rational of this mechanism is find out and continually updates an optimal configuration which extracts the maximum performance of the application and may reduce the application resource consumption. A new optimum configuration should be computed whenever the application performance drops based on changes of the application workload along the time – incoming requests, resources consumption, and application performance.

2 SmartTuning challenges

In this section we exemplify the use of SmartTuning with a real application and discuss the main challenges on applying its approach in real world. AcmeAir[1] is an implementation of a fictitious airline developed in Java. The application was built with the ability to scale to billions of web API calls per day. AcmeAir is bundled up with several runtime layers each one dealing with a specific aspect of application execution, Figure 1. The inner most layer is the application server OpenLiberty which transparently handle connections to database, components integration, and monitoring of the application. Next, as a Java application, both AcmeAir and OpenLiberty runs on the JVM, which abstracts the execution of code, and the management of memory and threads. All these components are encapsulated into a Docker Container, which abstracts the file system layer and several libraries of the OS. Finally, the container is

[1] `https://github.com/blueperf/acmeair-monolithic-java`

encapsulated into a Kubernetes Pod, which abstracts aspects regarding the cluster management such as scaling out and roll out new version.

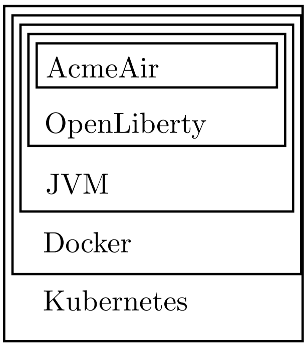

Figure 1: AcmeAir bundle.

Each of these layers exposes its own configuration interface with many options that engineers uses to set up the execution of the application. The combination of configurations among interfaces affects the behavior of the application. For example, the number of threads set in the OpenLiberty's thread pool is directly affected with the max heap size set in JVM, the number of cores and memory available set in containers, and the affinity set in a pod, so that Kubernetes deploys the application into nodes with more or less co-located pods. Therefore, a misconfiguration in one of these layers can potentially drops the performance of the application. We can notice that for the simple aspect of thread pool size, the number of possibilities may easily explode to hundreds of valid configurations, making unfeasible for engineers try all, while looking for one that maximizes the applications performance.

An application like AcmeAir has several aspects which engineers want to tune. Moreover, some of these configurations have interdependences in a same level, for example, the number of max connections which should remain opened in OpenLiberty depends on the size of its thread pool. Or, the garbage collection policy set depends on the heap size available in the JVM, and so on. It increases even more the complexity to find out and try all valid configurations of an application.

Another consideration when configuring application is the impact of the environment on itself. An application running with a same configuration and a same rate of incoming requests may have different performance along the time, thanks to fluctuations of other applications in the cluster. For instance, in a given time AcmeAir is deployed with very few other co-located applications. However, in another time AcmeAir remains with the same incoming rate and number of replicas, but the number of replicas of other applications drastically increase. Consequently, AcmeAir starts to suffer CPU contention by other replicas co-located in the cluster node dropping its performance.

Finally, modern CNA like AcmeAir are frequently evolving, and every next evolution may require a different configuration, which adds a new level of complexity to find optimal configurations to the application. At the end of the day, it becomes impossible for application's engineers try all possible configurations for each new scenario manually. To handle it, the engineers attempts a same configuration which suits the application in all possible scenarios. However, they have no guarantee if this configuration will be good enough for the whole application lifetime, and a careless set up eventually leads the application to a bad performance.

We propose SmartTuning to automatically support the search of good configurations for applications. SmartTuning identifies configurations based on analysis of the environment around the application and the application itself, tuning the application to reach the best performance possible. The concept of automatically tuning application's configuration is well known and successfully applied in context of machine learning [], in this domain named *hyper parameters optimization*. However, apply the same strategies of machine learning in the context of CNA might be a naive solution.

As we mentioned, CNA are dynamic and sensible applications which suffer with changes on itself (architecture or configuration) and on the environment where it is deployed, so that its behavior changes many times along its lifespan. Moreover, a CNA must satisfy many functional and non-functional requirements, which makes itself a “multi-purpose” application, with due proportions. Lastly, the lifespan of a CNA is nondeterministic, it runs in definitely since in most of cases it is providing a service for people or other applications.

On the other hand, machine learning applications are mostly defined to satisfy a single functional requirement – training a model to do specific task – with very few non-functional requirements – it should be fast and their outcome must have the minimum error. Besides, machine learning applications does not evolve frequency like CNAs, neither suffer so much impacts from the environment, since these application are used to run in dedicated hardware. Therefore, once it is figured out an optimal configuration for a machine learning application, it remains the same during its whole lifespan, which differently of CNA it is deterministic.

Therefore, in order to make possible apply the idea of automatically find out configurations to application (or auto-tuning), we are assuming that changes on the environment and on the application are periodical, so that SmartTuning can identify patterns and associate them to configurations. Every new pattern that SmartTuning observes from application or environment, which drops the application performance, trigger SmartTuning to figure out a best new configuration for this pattern. Hence, SmartTuning can handle this pattern properly in the next time it comes up.

Although the benefits brought by using SmartTuning, it is no free lunch. Even SmartTuning doing the hard work searching for an optimal configuration automatically, it may be unfeasible to get an optimal configuration in a reasonable short time. SmartTuning should be used in collaboration with the application engineers, so that they delimit the search space boundaries for SmartTuning looking for optimal configurations in a limited scope. Furthermore, we list in Section 3 other choices we made to make possible apply auto-tuning in CNAs.

3 Design choices

AS ▶*missing something describing gradual changes, fast change and memory leaks*◀

We have made some assumptions to design SmartTuning. Following a non-definitive list of them.

- An application starts with a default configuration initially computed on its test phase. We assume that the optimal configuration found during the tests are good enough to give the application a better initial configuration than a set by engineers manually.

- An application is an periodic function. We assume that if an application is observed long enough, some patterns on its behaviour and resources usage appears, so that SmartTuning can learns and use these patterns to improve how optimal configurations are computed and applied on applications.

- The application is a black-box function. Despite our assumption of the application be a periodic function (and then differentiable), we cannot found its max and min analytically. It is impossible to know *a priori* the frequency of the application's patterns neither the rules that define the slopes of their performance and resource usage along time. Therefore, we assume that the best way to find an optimal configuration is using a sort of sequential optimization method.

- SmartTuning observes the patterns that come up from different aspects of the application, such resources consumption and its inner behavior. We consider that all applications are properly instrumented, and all metrics necessary are exposed by the application at tuning phase.

- We are assuming that the inner behavior of the application can be modeled as a histogram of application's urls reached at runtime. Besides, we consider that the inner behavior is responsible to make changes on resources usages and performance of the application, i.e., different shapes of the histograms result on a different resource consumption and application performance. AS ▶*this doesn't sound quite right – we're assuming that the workload can be modeled that way, but not that the application itself can be*◀

- Two equals workflow, i.e., both have same shape and intensity of for every buckets, may result in different resources consumption and performance of the application. So, we are assuming that histograms cannot suppress the need of resource consumption metrics.

- Patterns that arise along the time from the application and can be grouped given a threshold. We suspect that patterns come up from the application frequently, but it may exist a slight differences among them. Besides, we assume that theses differences can be parametrized so that SmartTuning can group these patterns into different types. AS ▶*this one wasn't clear – need to give example of what you mean*◀

- We assume that changes on the patterns that come up from the application means the application has new non-functional requirements. So, for each new workload type a new configuration should be set to the application to get its best on performance and resources consumption. AS ▶*this one wasn't as clear – need to give example of what you mean*◀

- We expect the application remains that same, i.e., without new deployments, long enough for SmartTuning computes a optimal configuration and the application still take advantage of the new configuration for a significant amount of time.

- The application can be modeled by observing a reduced scope. We assume that the incoming traffic of the application is the only input necessary to provide changes on the application performance. Despite other variations on the environment may reflect on application performance, we are considering the environment stable. Hence, we can model the application behavior by measuring the only part incoming traffic, so that we can generalize the patterns of the whole application by observing the patterns that arise from a single replica, for instance. AS ▶*memory leaks or I/O contention from other things sharing disks, or other things sharing services the app is dependent on may have impact*◀

- AS ▶*engineers should set the length of the workload time interval – in future work this interval may be updated/set automatically*◀

4 SmartTuning Overview

To compute configurations for applications, SmartTuning relies on three main phases: workload classification, application tuning and tuning-workload prediction. Next we overview the SmartTuning's design, listing step-by-step how to maintain an application with good configurations, and highlighting the main phases of its workflow.

4.1 SmartTuning's algorithm

- Stage 1a - observe workload
  - watch incoming URLs
  - build a internal representation that (histogram)
  - watch resource consumption
  - defines workload in terms of incoming URLs and resource consumption
  - watch trends over time

- Stage 1b - trend analysis of workload
  - group the workloads in types
  - use model techniques to categorize the workload (e.g., appear there are 3 different workloads at different times)
  - use forecasting techniques to learn when a workload category arises

- Stage 2 - compute configuration
  - use hyper-parameter optimization techniques to pick a configuration candidate for tuning the application
  - continually evaluates the performance of the application given a new configuration and a workload and drops the configuration if the performance reached is worst then a previously given a same context (i.e., workload type)
  - contextualize the search of a optimal configuration with a workload type
  - associates configurations and workloads types
- Stage 3 - workload forecasting
  - forecast next workload and configuration of the application

4.2 Design Overview

TODO ►*learning takes some time, how long we need to wait before to start forecasting, we are going to forecast to avoid computing another a configuration for an already known workload type — describe this in this subsection or in next*◄

The SmartTuning operation unfolds into three main steps: to identify and group workloads of the application, learning an optimal configuration for a given workload, and foresee next workload and configuration. Figure 2 depicts these steps.

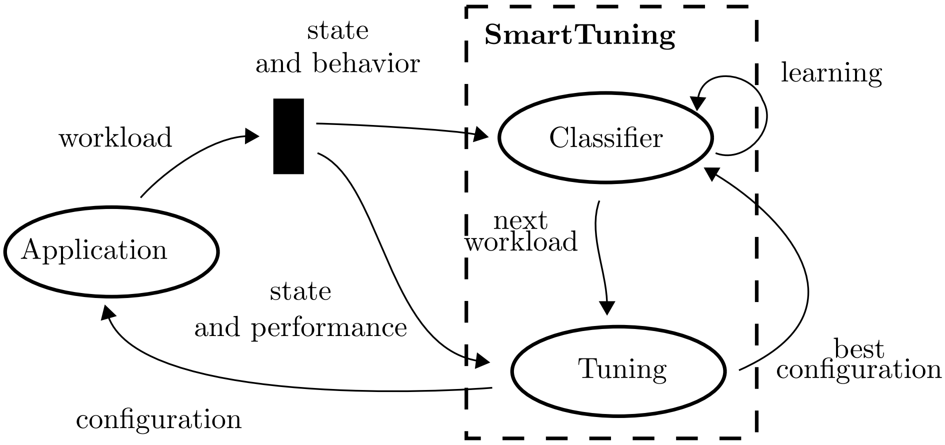

Figure 2: SmartTuning Overview.

We model an application as a black-box function $X : (c_k, t) \to w_t$, where X maps a configuration c_k in a time t to a workload w_t. A configuration is a set of knobs which the engineer can set in the application's runtime layers. An workload, $w_t = (b_t \in B, s_t \in S, p_t \in P, d)$, models respectively the behavior, the state, and the performance of the application beginning in t and with a duration d set by application engineer. The configuration c_k affects the behavior and performance of the application, and cannot affects affects the behavior of the application which reflects the incoming URLs to the application.

B are all possible behaviors of application and b_t models the application's behavior through the distribution of incoming URLs against the application endpoints in the interval $(t, t+d)$, i.e., urls histogram. In turn, S and P are all possible states and performance of an application, and s_t and p_t models respectively the resource consumption, e.g., CPU and memory, and application performance of the application, e.g., throughput and latency in the same interval $(t, t+d)$. Therefore, SmartTuning aims to improve (max) the application X, i.e., to find an optimal configuration c_k^* which maximizes the performance p_t and might minimize the resources consumption s_t both subject to a behavior b_t in a time interval $(t, t+d)$. Equation 1

$$X(c^*, t) = \max X = \begin{cases} \min(S) \\ \max(P) \end{cases} \quad , \text{subject to}\, b_t \tag{1}$$

Initially, the module named Classifier observes the workloads of the application under analysis. It groups the workloads based on similarities of their behavior or states, normalizing them into types, Equation 2. As next step, the Classifier uses techniques of time-series analysis to learn when a workload type comes up from the application. It associates the workload type to a time step while the application is running, Equation 3. Hence, Classifier can forecast which workload type will come up from the application in a given time t, Equation 4.

$$K : w_t \rightarrow \mathit{type}_i \tag{2}$$

$$T : (t,\, \mathit{type}_i) \cup L, L = \{\forall t\, \exists \mathrm{type}_i \,|\, t \rightarrow \mathrm{type}_i\} \tag{3}$$

$$F : (t,\, L) \rightarrow \mathit{type}_i \tag{4}$$

After SmartTuning has learned about the workload's application, AS ▶*not clear how SmartTuning is learning about the workloads's application at this point – so far in the flow it's just looking at incoming workload*◀ it is time of the module named Tuning figures out the best configuration for the application. The Tuning module uses the Classifier to forecast when and which will be the next workload of the application so it can compute a new configuration. For each workload type, Tuning computes several configurations trying to find out the one, c^*, which best improves the application. Then, SmartTuning updates the learning model L, appending the optimal configuration of each type, Equation 5.

$$T' : (\mathrm{type}_i, c^*) \cup L', L' = \{\forall\, \mathrm{type}\, \exists\, c^* \mid \arg\max_c X(c, t) = c^*\} \tag{5}$$

When Tuning applies a configuration to the application, SmartTuning observes differences on application workload's performance p or resource consumption s. Because the application is a black-box function, and thus unknown whether it is differentiable, SmartTuning has to sequentially tries different configurations c_k looking for one which leads the application to its best – high performance or low resources consumption, Figure 3. To do so, Tuning try different configurations for a same type of workload and maintain internally a history of the configurations already tested against the application. Tuning applies a configuration c_k to the application and wait d for new application's state and performance. The new configuration feedbacks the Classifier, and updates its learning model if the performance of the application is improved. The number of iterations to find an optimal configuration depends on the application and which optimization technique SmartTuning uses.

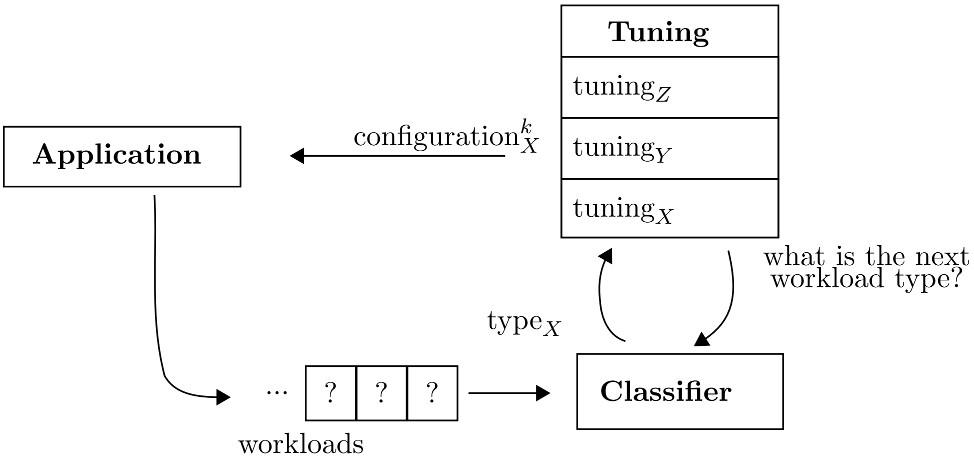

Figure 3: K-th tuning iteration for workload of type X.

After n, $n \geq k$ iterations with a same type of workloads, SmartTuning learns which is the best configuration for that type. The optimization is driven by the types of workloads, whenever a new type comes up to Tuning, it has to internally changes its context to does not mess up with the computations made previously for the other types. To guarantee that each type has an optimal configuration, Tuning maintain an inner state for each type of workload previously analyzed. Figure 3 depicts its behavior. When Classifier forecasts a new workload type from the application, Tuning computes a new configuration and evaluates if it improves the application X.

4.3 SmartTuning phases

TODO ►*write a quick summary here*◄

4.3.1 Workload classification

The main objective of SmartTuning is to associate an application's optimal configuration to a workload. To do so, first SmartTuning has to characterize the workloads. Along the time, several different workloads come up from the application. To reduce the scope and the number of configurations to be computed, SmartTuning characterizes the workloads into groups.

SmartTuning characterizes a workload in a multidimensional histogram-ish structure constituted with the workload's behavior (b), state (s), and performance (p) of the application in a given time interval. The application engineer should set in SmartTuning the length d of the time interval which bounds every workload. Hence, periodically, SmartTuning samples all suitable data from the application within the interval (t_i, t_{i+d}) to characterize the workloads. For instance, SmartTuning samples data regarding AcmeAir from Prometheus periodically, in intervals of $d = 6$ hours to build up the workloads. Figure 4 depicts the overview of workload classification.

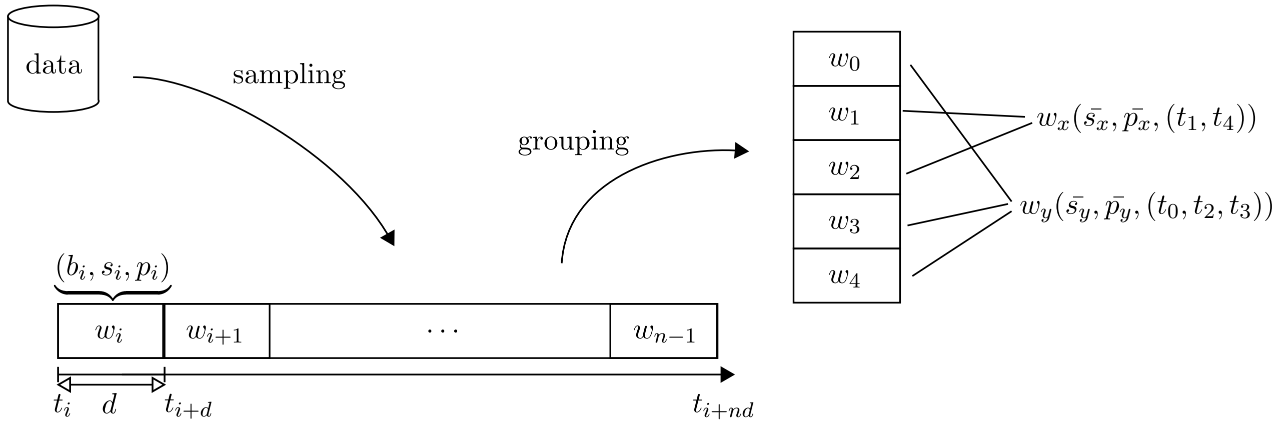

Figure 4: Workload classification.

The behavior of the workload is given by the incoming URLs to the application. The distribution and frequency of these URLs along the time interval model the shape and intensity of the workload behavior histogram respectively, Figure 5. The histogram is enhanced with state and performance dimensions. The dimension state add an information about the resource consumption of the application in the time interval. The resource consumption, usually CPU, memory, and I/O components, is measured and the average, standard deviation, and number of samples of each metric are kept in the histogram structure. Moreover, the average and standard deviation, and number of samples of different performance metrics of the application, such as throughput and latency, are kept in the histogram as well.

Every workload that SmartTuning samples are grouped into types according to their characteristics, reducing the number of configurations to be computed afterwards. The workloads are compared and grouped according their similarities. Finally, SmartTuning stores each workload type with a list of times when its comes up for forecasting purposes.

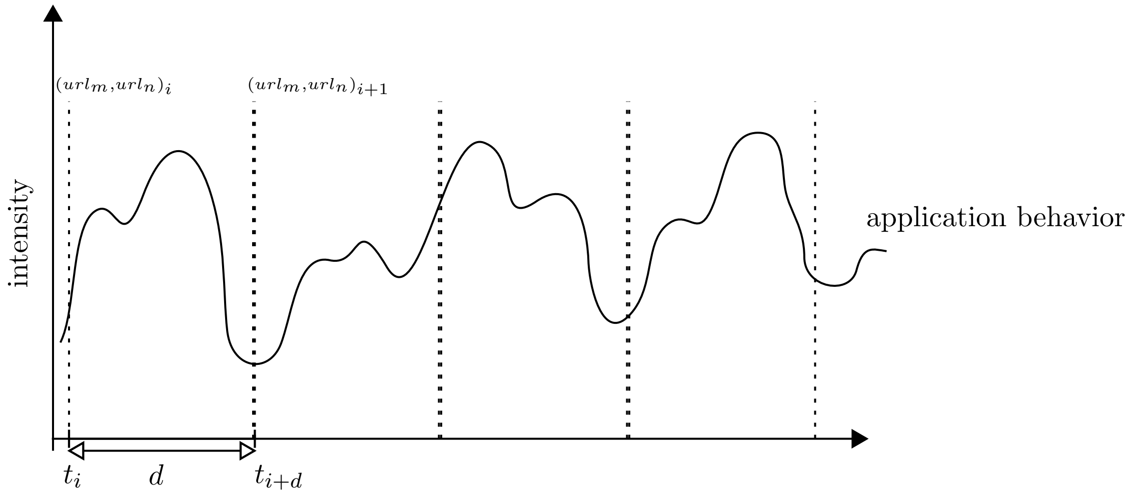

Figure 5: Histogram behavior.

The SmartTuning's Classifier groups the workloads by their behaviors and states. It uses K-Nearest Neighbor (KNN) [?] with the number of groups k set by the application engineer. The core of KNN relies on the method to compute the similarity between the elements being classified. Our approach considers the behavior and state of workloads to calculate their proximity.

TODO ►*should K-nearest neighbors be static (the engineer set K) or dynamic – SmartTuning defines how many clusters based on a given threshold set by engineer?*◄

The behavior of two workloads are compared using Hellinger distance [?], an statistic method to quantify the similarity, between 0 (equals) and 1 (different), of two probabilities distributions. Since Hellinger distance works with probabilities distributions, the histograms should also be normalized between 0-1.

To avoid miscalculations and to preserve the semantics of all histograms, the classifier normalizes two behaviors by using MinMax [?], setting min and max as the smallest and largest values of both histograms under analysis, e.g., $h_1 = (3, 2, 5)$ and $h_2 = (2, 9, 5)$, min $= 2$ and max $= 9$. Therefore, $h'_1 = (0.1, 0.0, 0.3)$ and $h'_2 = (0.0, 0.7, 0.3)$. Behaviors with same shape and different intensities (height), means that the highest behavior consumes more resources, since workload height is consequence of the number of incoming URLs processed by application.

However, not always the resource consumption is related only to the quantity of incoming requests. In some cases, a low resource consumption is consequence of resources contention caused by another application which is co-located into a same node of the first. Or, the resource consumption changes because the application's replica under monitoring is running in a node with more or less resources available them previously. For a few URL incoming rate, Classifier cannot identify the variations on resources consumption by only analyzing the workload behavior.

Therefore, Classifier uses Z-Score [?] to compare the state of two workloads. Given two workloads, for each their components $(a_0, a_1, \cdots, a_n)$, e.g., CPU, memory, and I/O, in their states s_i, Classifier checks if all means of each state component are significantly equals. For values of Z-Score ≤ 1.960, we can assume they are significantly equals, and Z-Score ≥ 3 they are significantly different.

The behavior and state of a workload are orthogonal concepts. Then, after the Classifier calculates the similarity of both individually for two different workloads, it is necessary to bring these concepts to a same universe. Hence, Classifier can calculate the real similarity of two workloads. To do so, Classifier assumes that the state of a workload is a vector $\hat{s}$ and calculates its magnitude, $||\hat{s}||_2$ and direction $\theta\hat{s}$, both from origin $(0, 0)$.

The decomposition of $\hat{s}$ tells to Classifier the amount of each resource is being used (magnitude) and discern the semantics of each component in the vector (direction). For example, a resource consumption of 100 milicores and 1000MB has a different semantics of 1000 milicores and 100 MB, and consequently the application may need different configurations to handle it. In both cases, the magnitude are equal

(~ 1005), and their directions are different ($\sim 1.47\,\mathrm{rad}$ and $\sim 0.09\,\mathrm{rad}$ respectively).

Then, Classifier can transform the vector state $\hat{s}$ in in a scalar s' by calculating their distance on $\mathbb{R}^2$, Equation 6.

$$s' = \sqrt{\mathrm{magnitude}^2 + \mathrm{direction}^2} \tag{6}$$

Finally, with both behavior and state being scalars, Classifier can transform both orthogonal concepts in a single metric distance, Equation 7, used by KNN for grouping the workloads.

$$\mathrm{distance} = \sqrt{b^2 + s'^2} \tag{7}$$

After the Classifier groups the workloads, as depicted in Figure 4, it also associate to each group the time of their fundamental workloads occurred. Moreover, it computes the average of application performance and resource consumption and keeps a structure for each type as following:

$$(\mathrm{type}_k, \text{ average of performance, list of time intervals})$$

This structure is the core of the next phases of the SmartTuning. Tuning phase uses this structure to calculate the improvement on the application when it apply a configuration (it aims to minimize the resource consumption and maximize the application performance). The Tuning-workload prediction uses the structure to for modeling when a given type of workload arises in the application.

4.3.2 Application tuning

TODO ►*next week*◄

4.3.3 Tuning-workload prediction

TODO ►*next week*◄

References