

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 overall application. Moreover, this architectural style allows a decoupled evolution of the application by updating each component separately. However, the high decoupling makes it difficult to manage the application, since there are many distributed pieces of software that the engineers should **AS** ► *handle – vague. what is the problem, particularly one that you are focused on solving* ◄ during the application execution.

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

Runtime abstractions layers are general purpose and have many parameters (knobs), that engineers adjust to better satisfy the application. Many of these knobs are related to data structures or infrastructure mechanisms, so their adjustment affects performance. An problem with these knobs is the number of possibilities that an engineer has to consider.

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

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

TODO ► *give a roadmap about what SmartTuning is* ◄

TODO ► *how the paper is structured* ◄

2 SmartTuning challenges

TODO ► *this section needs beter structure. Challenges and go through them one by one; bold each challenge you discuss* ◄

TODO ► *add devops perspective to this section* ◄

TODO ► *also add to the problem that ops teams now have to manage a variety of software components for a variety of applications. they don't have time to focus on perfecting each one* ◄

In this section we explain SmartTuning using a real application example and discuss the main challenges of applying our approach in the real world. AcmeAir¹ 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

¹<https://github.com/blueperf/acmeair-monolithic-java>

execution (Figure 1). The inner most layer is the application server OpenLiberty which transparently handles connections to the database, components integration, and monitors of the application. Next, as a Java application, both AcmeAir and OpenLiberty runs in 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 encapsulated into a Kubernetes Pod, which abstracts aspects regarding the cluster management such as scaling out and roll out new version.

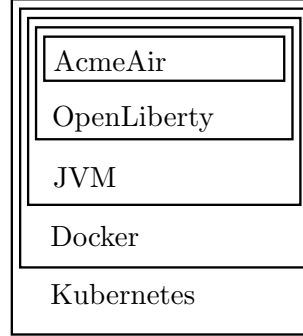


Figure 1: AcmeAir bundle.

Each of these layers exposes its own configuration interface with many options that engineers use 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 OpenLiberty’s thread pool is directly affected with the max heap size set in JVM, the number of cores and memory available to the container, 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 drop the performance of the application. Notice that for the simple knobs controlling thread pool size, the number of possibilities may easily explode to hundreds of valid configurations, making it unfeasible for engineers to try all, while looking for one that maximizes the application’s performance.

An application like AcmeAir has several aspects which engineers want to tune. Moreover, some of these configurations have interdependencies, for example, the number of max connections which should remain open 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 an application is the impact of the environment. An application running with the same configuration and the same workload may have different performance due to 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 each step in their evolution may require a different configuration, which adds a new level of complexity in finding an optimal configuration. At the end of the day, it becomes impossible for application engineers try all possible configurations for each new scenario manually. Engineers to try a configuration which suits the application in all possible scenarios. However, they have no guarantee if this configuration will be good enough for the application at all times, and a careless set up eventually leads the application to a bad performance.

We propose SmartTuning to automatically support the search of good configurations. SmartTuning identifies configurations based on analysis of the environment around the application and the application itself, tuning the application to reach the best possible performance. The concept of automatically tuning

an application’s configuration is well known and has been successfully applied in the context of machine learning [], in Machine Learning domain named *hyper parameters optimization*. However, applying in the same strategy of machine learning in the context of CNAs is a naive solution.

As we mentioned, CNAs change behavior many times over time. Moreover, a CNA must satisfy many functional and non-functional requirements, which makes it a “multi-purpose” application. Lastly, the lifespan of a CNA is nondeterministic, it runs indefinitely since in most of the cases it provides a service.

TODO ▶*start by discussing the application of ML to your problem context – On the other hand*◀, machine learning applications are mostly defined to satisfy a single functional requirement – training a model to do the specific task – with very few non-functional requirements – fast training and high model accuracy. Besides, machine learning does not frequently evolve like CNAs, nor suffer as many impacts from the environment. Therefore, a machine learning configuration remains the same during all the time since it is deterministic, differently of CNAs.

TODO ▶*assumption → next reaction?*◀ Therefore, to apply the idea of automatically finding a configuration to an application (or auto-tuning), we assume that changes in the environment and of the application are periodic, and these changes are observed through application’s workloads, so that SmartTuning can identify patterns and associate them to configurations. Every new pattern that SmartTuning observes from an application or the environment, which drops application performance, trigger SmartTuning to figure out the best new configuration for this pattern. Hence, SmartTuning can handle this pattern properly in the next time it comes up.

Therefore of using SmartTuning, it may be unfeasible to find an optimal configuration in a reasonable time. SmartTuning should be used in collaboration with the application engineers, so that they delimit the search space boundaries for SmartTuning. Furthermore, we list in Section 3 other choices we made to make it possible to 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. The following is a non-definitive list:

- 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.
- The behavior of an application is described as a 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 learn and use these patterns to improve how optimal configurations are computed and applied on applications.
- The behavior of an application is a black-box function. Despite our assumption of the application be a periodic function (and then differentiable), we cannot find its max and min values 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 the 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 assume that the application can be modeled as a histogram of application’s urls reached at runtime. Besides, we consider that this behavior is responsible for make changes in resources usage and performance of the application, i.e., different shapes of the histogram 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 equal **TODO** ► *workflows – to define workflows somewhere before this sentence* ◀, i.e., both have same shape and intensity of for every bucket, 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 these 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 in 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 the same, i.e., without new deployments, long enough for SmartTuning computes an 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 an 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 the pattern of 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 the worst then a previously given a same context (i.e., workload type) **AS** ▶*this is a key loop that needs to be broken into more steps*◀
 - contextualize the search of an 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.

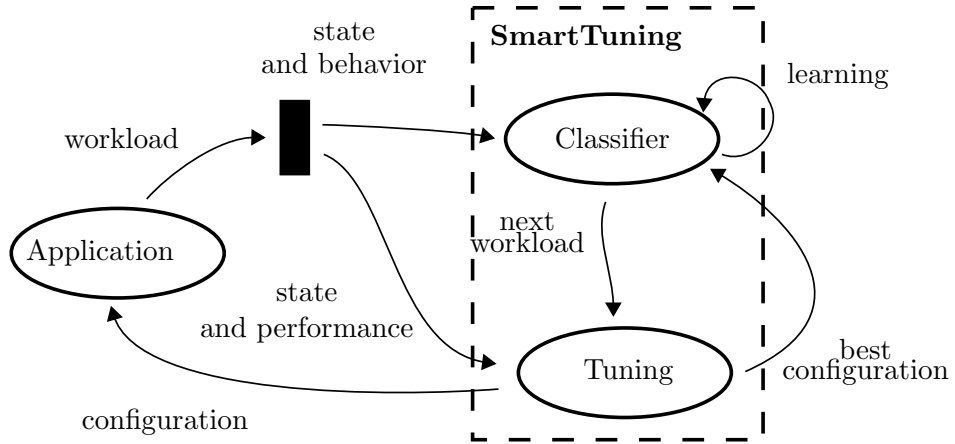


Figure 2: SmartTuning Overview.

We model an application as a black-box function $X : (c_k, t) \rightarrow w_t$, 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. A 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 affect 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

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

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.

$$K : w_t \rightarrow type_i \quad (2)$$

$$T : (t, type_i) \cup L, L = \{\forall t \exists type_i \mid t \rightarrow type_i\} \quad (3)$$

$$F : (t, L) \rightarrow type_i \quad (4)$$

After SmartTuning has learned about the workload's application, **AS** *►not clear how SmartTuning is learning about the workload'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.

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

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 resource 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.

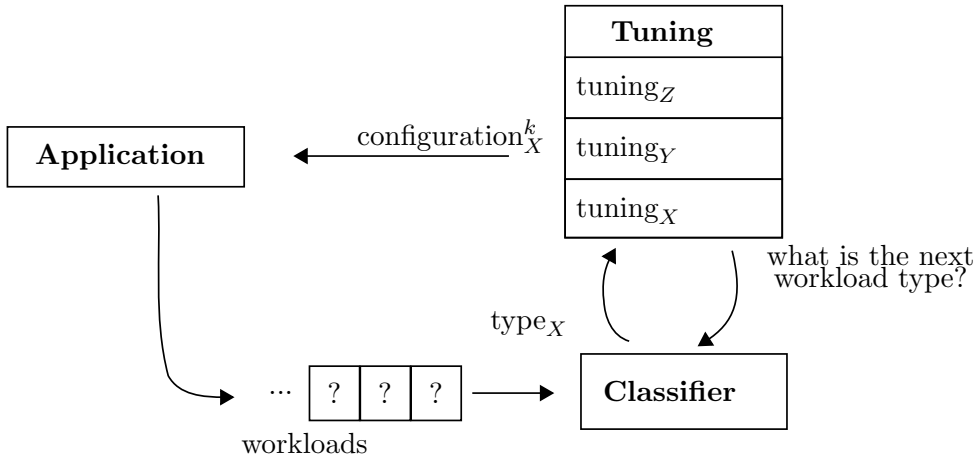


Figure 3: K-th tuning iteration for workload of type X.

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 key 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 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 **AS** ▶ *this seems like something the system needs to figure out. per section 2 we need to think of ops teams as unable to spend time studying each app*◀. 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 **AS** ▶ *my guess would be we'd want smaller d's (eg. 15-30 min)... but that's probably something for our experiments to figure out*◀. Figure 4 depicts the overview of workload classification.

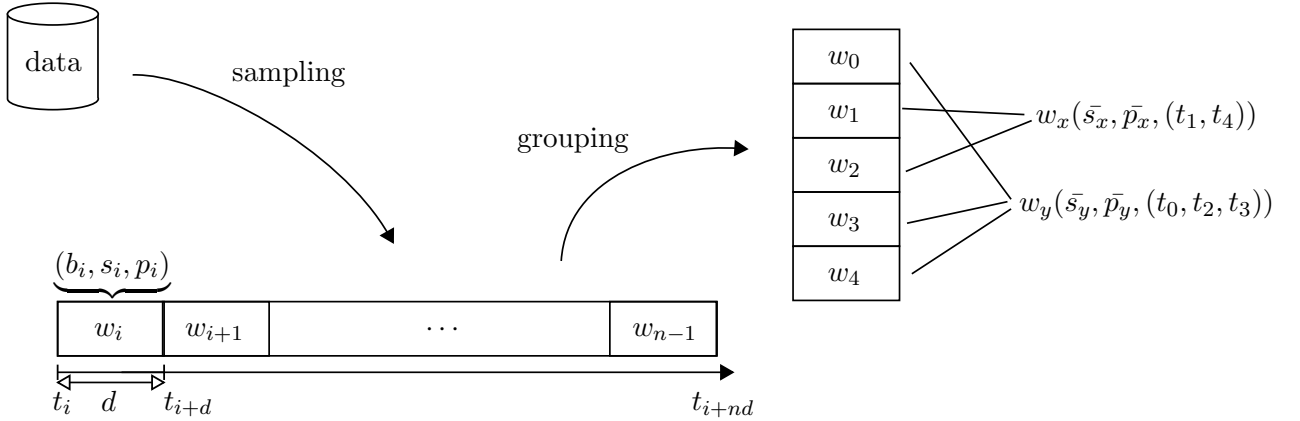


Figure 4: Workload classification.

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

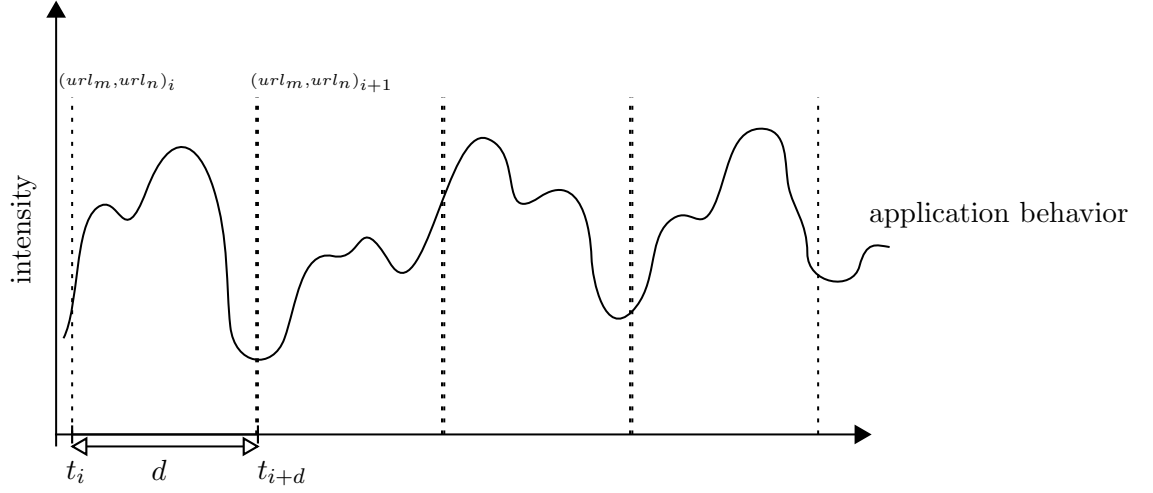


Figure 5: Histogram behavior.

grouped according their similarities. Finally, SmartTuning stores each workload type with a list of times when its comes up for forecasting purposes.

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?* ◀ **AS** ▶ *again this might be something we pick defaults for and only in rare situation would humans get involved in setting this* ◀

The behavior of two workloads are compared using Hellinger distance [?], a 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 their components (a_0, a_1, \dots, 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 $\theta\hat{s} = \tan^{-1}\hat{s}$, 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 1000 MB 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 (~ 1.47 rad and ~ 0.09 rad respectively).

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

$$s' = \sqrt{\text{magnitude}^2 + \text{direction}^2} \quad (6)$$

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.

$$\text{distance} = \sqrt{b^2 + s'^2} \quad (7)$$

After the Classifier groups the workloads, as depicted in Figure 4, it also associates 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:

(type_k, average of performance, list of time intervals)

This structure is the core of the next phases of SmartTuning. Tuning phase uses this structure to calculate the improvement on the application when it applies 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

SmartTuning continually tunes the application for different workloads. Tuning component asks Classifier, at every time step t_i of duration d , what is the next workload w_p for the application. Next, Tuning picks a configuration c_k attempting to improve the application for this given workload. Figure 6 depicts how Tuning works.

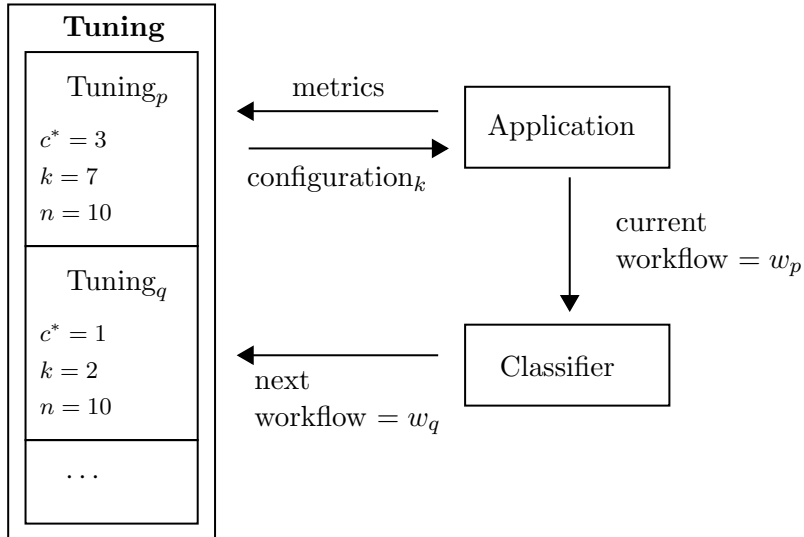


Figure 6: Tuning internals.

Tuning identifies the best configuration c_p^* , i.e., which $c_{k,p}$ leads the application to its best, after n iterations over a same workload w_p . For each configuration $c_{k,p}$, Tuning component measures the performance and resource usage of the application and if $c_{k,p}$ improves the application regarding $c_{k-1,p}$, $c_{k,p}$ becomes the new best configuration c_p^* . So, after k iterations, Tuning resets its inner state and resumes its search

for a new optimal configuration. New best-configurations are constantly updated because the requirements of the application and the environment where it is deployed changes along the time, making that an old best-configuration get worse after some time.

Both Tuning and Classifier components use the same time interval to evaluate a configuration and to delimitate a workload respectively. For each time interval starting in t_i of length d , Tuning asks Classifier what is the next workload that will come up to the application in t_{i+d} . Based on this information, Tuning picks a configuration for this next workload and sets it to the application, measuring for the next d time units the application's performance and resource consumption.

While Tuning is searching the best configuration, at least one another instance of the application must run in parallel: one for the default or best configuration found, and another that is used by Tuning to search for a new best configuration. Figure 7 depicts the time line of application tuning.

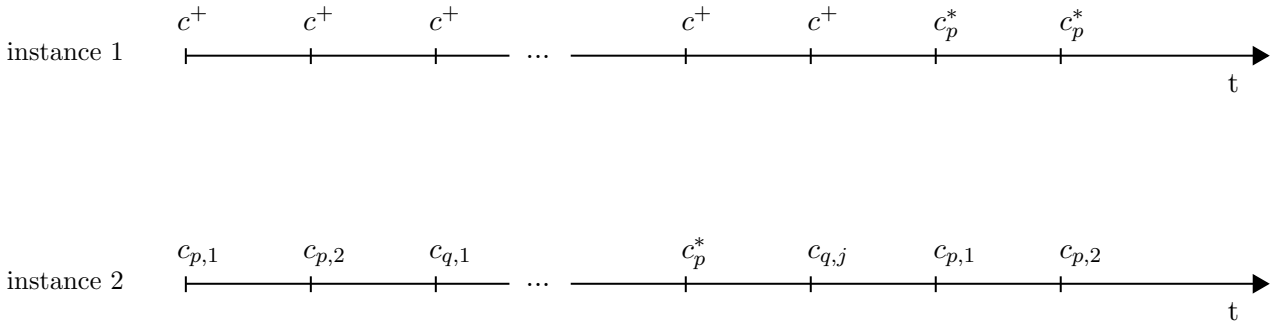


Figure 7: Configurations being computed along the time using two instances of an application. Instance 1 runs with default c^+ and best configurations c_p^* while instance 2 runs with different configurations. For each workload that comes up Tuning searches for new best configurations on instance 2.

Initially, one instance of application starts with a default and general configuration c^+ previously defined by application engineers. Then, whenever a new workload w_p comes up, Tuning apply a configuration $c_{p,i}$, and along the time, Tuning tracks which $c_{p,i}$ is the best for the configuration after k iterations.

Different workloads arise interleaved along the time. If the next workload is the same type of the current, Tuning picks another configuration and measures the application again, tracking which configuration has improved the application. Otherwise, Tuning switches its context to track what is the best configuration for this new workload. Every k iterations, the best configuration c^* for a given workload is updated, and whenever this workload type arises again, the application is set to use c^* .

Thus, the process of searching a new configuration has small footprint in the overall application, since one instance runs with the best configuration while Tuning tries other possibilities to improve the application. **AS** ▶ *We assume that a small portion of the workload can individualize it from another* ◀. Hence, when Tuning identifies that a configuration degrades the performance of the application after n time units, $d - n \leq n$, it can set new configuration on the application, avoiding instances running long times with unacceptable performance.

Finally, all process of tuning deeply relies on picking a new configuration for the application being observed. Tuning uses Bayesian Optimization [?], BO, to find out which is the best configuration for the application. Bayesian Optimization is a sequential design strategy for global optimization of black-box functions that does not require derivatives. These characteristics make BO an ideal to find optimal configurations for an application.

TODO ▶ *explain succinctly how BO works and its advantages over Random Search* ◀

AS ▶ *best suitable for continuous domains of less than 20 dimensions* ◀

The application engineer should set a search space where Tuning will look for a new configuration. For each time step t_i , Tuning uses the BO framework for picking a configuration from the search space. In the first iteration, BO picks values uniformly from each dimension of the search space. Next, at each iteration, BO updates the posterior probability² distribution on the function being optimized (application) using all available data. So, BO uses the past configuration found to be a maximizer of the acquisition function over next configuration, where the acquisition function is computed using the current posterior distribution. Finally, observes the performance and resource usage of the application for d time units, and repeat all again n times.

After n times, BO will find, at least, an quasi-optimal configuration which improves the application. If the application has more than 20 configurable parameters or most of their configurable parameters are in discrete domains, BO cannot use the benefits of Bayesian Statistics [?], and in this scenario, Tuning behaves like it is using a Random Search optimization [?].

4.3.3 Tuning-workload prediction

TODO ▶ *next week* ◀

References

²Posterior probability is the probability an event will happen after all evidence or background information has been taken into account.