

# 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 the 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, based on changes of the application workloads along the time (incoming requests, resources consumption and application performance), find out and continually updates, an optimal configuration which extracts the maximum performance of the application and for some scenarios it also might reduce the application resource consumption.

**AS** ▶*think about small example at end of section 1 that has example complex enough to show value of prediction and tuning: load balancer and database*◀

## 2 Assumptions

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 find 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 as 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.
- Two equal workflows, i.e., both have the 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 slight differences among them. Besides, we assume that these differences can be parametrized so that SmartTuning can group these patterns into different types.
- 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.
- 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.

### 3 Design Overview

**AS** ▶ *add a concise representation of the flow in the doc* ◀

**AS** ▶ *begin(example)* ◀

- Stage 1a - observe workload
  - watch incoming URLs
  - build an internal representation that (histogram)
  - watch trends over time
- Stage 2b - trend analysis of workload
  - use model techniques to categorize the workload (e.g., appear there are 3 different workloads at different times)

AS ▶end(example)◀

AS ▶describes each phase: classification, tuning, prediction◀

The SmartTuning operation unfolds into two main steps: to identify and group workloads of the application, and to find out an optimal configuration to a given workload. Figure 1 depicts these steps.

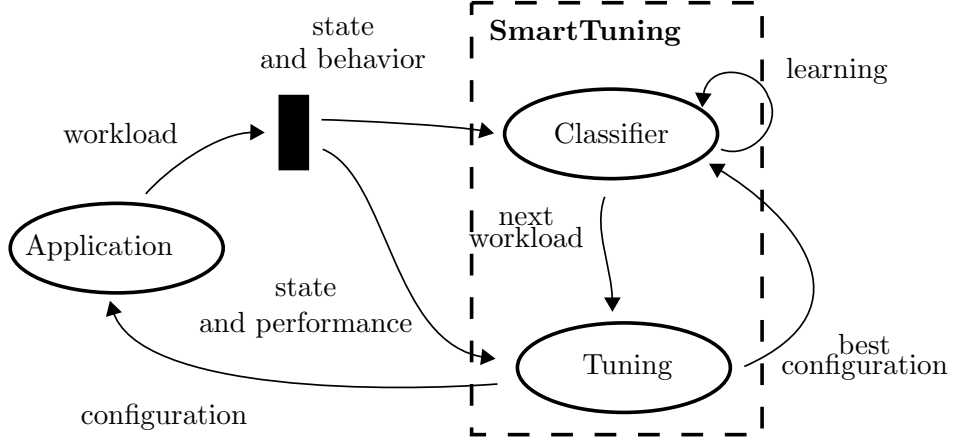


Figure 1: SmartTuning Overview.

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 interval  $t$  to an workload  $w_t$ . A configuration is a set of knobs which the engineer can set in the application's runtime layers. An workload,  $w_t = (b_t \in B, s_t \in S, p_t \in P)$ , models respectively the behavior, the state, and the performance of the application along a time interval  $t$ .

$B$  are all possible behaviors of application and  $b_t$  models the application's behavior through the distribution of urls hits against the application endpoints in the interval  $t$ , 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  $t$ . Therefore, SmartTuning aims to improve (max) the application  $X$ : 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  $t$ . Equation 1

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

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 learns when a workload type comes up from the application, associating the workload type to time time-interval while the application is running, Equation 3. Finally, Classifier can forecast which workload type will come up from the application in a given time interval  $t$ , Equation 4.

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

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

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

After SmartTuning has learned about the workload's application, 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.

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

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 resources consumption, Figure 2. To do so, Tuning has to 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  $t$  for new application's state and performance. The number of iterations to find an optimal configuration depends on the application and which optimization technique SmartTuning uses.

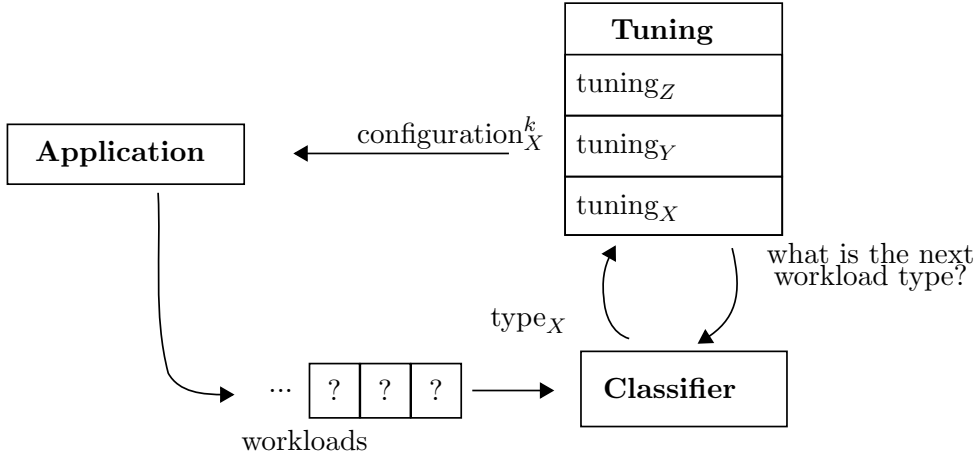


Figure 2: K-th tuning iteration for workload of type X.

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 2 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  $X$ .