POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

AGORA: AN ADAPTIVE FRAMEWORK SUPPORTING RUNTIME APPLICATION AUTOTUNING

M.Sc Thesis of:
Bernardo Menicagli

Supervisor:
Prof. Gianluca Palermo

Co-supervisor:
Dott. Davide Gadioli

Academic Year 2019/2020

Abstract

APPLICATION autotuning is a promising path investigated in literature to improve computation efficiency. In this context, the end-users define high-level requirements and an autonomous manager is able to identify and seize optimization opportunities by leveraging trade-offs between extra-functional properties of interest, such as execution time, power consumption or quality of results. The relationship between an application configuration and the extra-functional properties might depend on the underlying architecture, on the system workload and on features of the current input. For these reasons, autotuning frameworks rely on the so called application-knowledge to drive the adaptation strategies. The application-knowledge is typically produced off-line because it highly depends on an expensive phase called Design Space Exploration whose fruition requires significant effort in order to reduce its overhead.

This master thesis aims at providing an adaptive framework which supports application autotuning at runtime in order to get the application-knowledge during the production phase and in a distributed fashion. Experimental results show how the proposed approach is able to learn the application knowledge maintaining the quality thresholds while exploring a small fraction of the design space.

Contents

CHAPTER 1

Introduction

The end of Dennard scaling [1] forced the beginning of a new era where the optimization focus towards the search of efficiency in a wide range of scenarios drastically increased, from embedded systems integration to High-Performance-Computing (HPC). This marked the birth of the *autotuning* concept that has been identified as a promising research field. Autotuning refers to the automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation. Autotuning has the potential to dramatically improve the performance portability of applications [2]. In this direction many frameworks begin appearing on the scene with goals that go from the optimization of a specific task to the optimization of certain configurations, often referred to as *software-knobs*, which can be identified both at system-level (e.g. the core frequency) and at application-level (e.g. software parameters). It can happen that the software-knobs configurations may be changed at runtime, for which they're referred to as *dynamic-knobs* [3]. Moreover, there is a large class of applications that implicitly expose software-knobs to find accuracy-throughput tradeoffs. This approach goes with the name of *approximate computing* [4] which can significantly increase the application throughput by decreasing the result accuracy. Following this philosophy, several techniques were born such as *loop perforation* [5] or *task skipping* [6] to name a few well known.

As a consequence of this, application requirements in terms of *extra functional properties* (EFPs), such as execution time, power consumption or quality of results, are increasing in complexity. Moreover, these EFPs might depend on the actual inputs of the application, on the available resources and on the underlying configuration. A subset of software-knobs relates to parameters that aim at tailoring the application for the underlying architecture, such as work-group size, MPI runtime parameters or

compiler options and typically the autotuning frameworks that address these parameters perform a *Design Space Exploration* (DSE) at design-time to find the most suitable configuration to be used during production. A second subset of software-knobs relates to application-specific parameters, such as the confidence level, loop perforation factors or algorithm parameters. Typically, it is easier to change these software-knobs during the production phase and their effects on EFPs are strongly coupled with the features of the current input. For these reasons the autotuning frameworks often model this relationship as an *application-knowledge* and leverage it to identify and seize the optimization opportunities [7].

1.1 Thesis Motivations

The main challenge of the general approaches described previously is the exponential growth of the Design Space (DS) when considering several, and usually unbounded, software knobs. Given the complexity of the this tuning process, the DSE phase is usually done off-line prior to the application execution, leaving only the configuration selection at run-time. The main issue behind this is that the code could be ported to a new architecture, updated, or the application could even require the elaboration of new input data. This could potentially nullify the whole DSE and eventually lead to a forced restart of the whole off-line tuning process. On the other hand, porting this phase online at production time requires a significant effort to minimize the tuning time and the overhead. Despite being a known problem in literature, this has not been throughly investigated and remains an open question.

1.2 Thesis Contributions

This thesis' work aims at advancing the state-of-the-art proposing a framework which exploits a model-driven approach to learn the application-knowledge *online*, at the beginning of the production phase. The methodology is based on an iterative approach to obtain the application knowledge using as few samples as possible. It has been designed to work in a distributed context where different entities can collaborate to the knowledge collection. The framework mainly targets the context of HPC, where an application is composed of more than one process and it usually executed for a long period. However, it might be applied also in a wider range of scenarios. There are multiple benefits in learning the application-knowledge at runtime:

- The actual execution environment of the application is being used and is stored for dynamic autotuning;
- The application behavior learning phase uses the actual features of the input set;
- Being able to leverage the parallelism of the platform, it is possible to reduce the *time-to-knowledge*.

The starting point of this thesis is *mARGOt* [8], a dynamic autotuning framework that has been implemented as an adaptation layer which gives to the target application mechanisms to adapt in a reactive and proactive fashion, based on the application-knowledge provided. The learning module is designed to enhance *mARGOt*. It is

important to highlight that during this work no new modeling techniques were created nor comparison between existing ones were made. The main focus was to leverage existing techniques in order to reduce the time required to compute the application-knowledge that *mARGOt* uses.

The main contributions of this thesis can be summarized as follows:

1. The methodology implementation has been named *Agora*. *Agora* is a C++ auto-tuning framework to learn the relationship between EFPs, software-knobs configurations and input features during production time. The incorporated iterative exploration strategy is employed in order to reduce as much as possible the required number of samples. *Agora* is able to address an arbitrary number of EFPs, leaving the end-user the ability to integrate it without the need of porting it to another language or using a different compiler for the use. A skeleton of the main module was already available but not fully integrated with the rest of the framework nor was scalable inside a distributed context. The effort of this thesis made it an asset for the exploitation of *mARGOt*. The framework source code is publicly released [9].
2. A modular, flexible and fully extendible plugin system that takes care of each step of the iterative procedure independently. The end-user can extend the plugins with its own implementation in a language agnostic fashion.
3. Thanks to a scalable infrastructure, it can exploit the parallelism to strongly increase the application knowledge generation speed.
4. An experimental evaluation of *Agora* has been performed on known benchmark applications to demonstrate the introduced benefits.

1.3 Thesis Outline

This thesis has been divided into the following chapters. Chapter 2 describes the background and introduce the reader to the main concepts covered in this thesis. It also provides an overview of the current state-of-the-art, highlighting the contributions of the proposed framework. Chapter 3 describes the proposed approach in details, explaining the methodology in depths and the design efforts. Chapter 4 outlines *Agora* from the implementation point of view, describing the build system used and how the library and the plugin system work in details, specifying the technologies used and the most critical aspects faced during development. A brief example is presented to demonstrate the integration steps needed to leverage *Agora*. Chapter 5 describes the experimental validations that have been made to address both benefits and limitations of the proposed framework. Starting from the description of the target architecture where the experiments have been running, it follows a summary of some known benchmarks which have been considered as target applications. After describing the experiments that addressed the framework scalability and overhead, the chapter ends with a final evaluation on the methodology quality. Finally, Chapter 6 concludes this thesis by summarising the main contributions and leaving some recommendations and thoughts on future works.

CHAPTER 2

Background and Previous Works

This chapter provides an introduction to the research field related to this thesis' work and gives a definition of the main concepts and technologies that will be addressed during *Agora* framework description and comparison with the current state-of-the-art. In the following sections, the reader is introduced to some key concepts which hopes to facilitate the understanding behind the methodology that has been adopted. We start by presenting the autonomic computing world along with some general definitions and terminology of the field. Then, we describe some recent works and comparison in relation to this thesis' scope. Moreover, we take a background overview on the experimental design approach that aims at reducing the Design Space by obtaining suitable configurations to explore. We introduce the concept of process modeling along with the reasons why it is important in conjunction with the experimental design output. After that, we outline the clustering technique in relation to the input features sensitivity problem. Then, we briefly describe a lightweight network message protocol which has been leveraged inside *Agora*'s distributed context. Finally, we introduce *mARGOt*, a dynamic autotuning framework that has been the starting point of *Agora*'s core development.

2.1 Autonomic Computing

In mid-October 2001, IBM released a manifesto [10] that the main obstacle to further progress in the IT industry is a looming software complexity crisis. The manifesto pointed out that the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. As

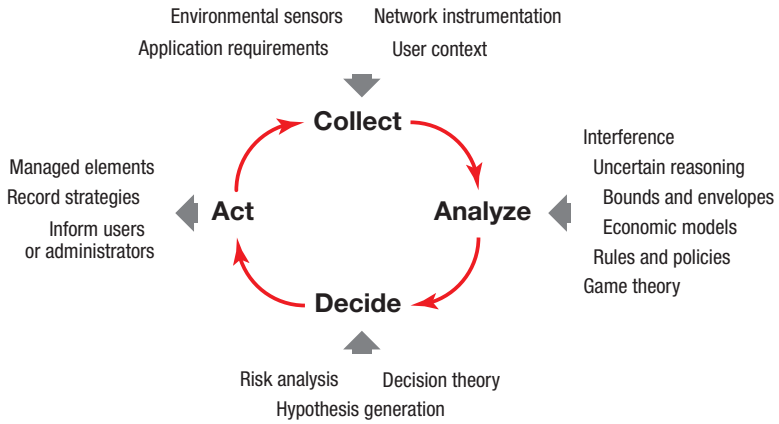


Figure 2.1: *Technologies applied to the four stages of the autonomic control loop. Although inspired by control theory, this structure encompasses symbolic and other techniques within a common framework as well as aspects of both computing and communications.*

systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime.

In the autonomic computing [7], an application is perceived as an autonomic element capable of self-management. Among the *self-** properties required by this self management, autotuning frameworks aim to provide *self-optimization*. In this specific context, the end-user should specify high-level requirements and the application should adapt accordingly. Autonomic systems will continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost. Just as muscles become stronger through exercise, and the brain modifies its circuitry during learning, autonomic systems will monitor, experiment with, and tune their own parameters and will learn to make appropriate choices about keeping functions or outsourcing them. They will proactively seek to upgrade their function by finding, verifying, and applying the latest updates.

By this premise it comes natural that developers considering the evolution and management of systems in terms of self-* properties must take a different perspective, for example, by including programmatic monitoring and management interfaces. Such a perspective, while common in telecommunications in the form of managed components, is unusual in software architectures still based largely on configuration files read only at start-up time. As Figure 2.1 shows, providing monitoring and control suggests the application of control theory expressing a control action derived from a system's observed behavior against a model of intended or expected behavior. Researchers have successfully applied such techniques to, for example, power management [11], to achieve clear closed-form representations. However, it is less clear whether the techniques can be applied more broadly in areas where the control domain changes dynamically [12].

2.1.1 Definitions and Terminology

Autotuners work at the application level, leveraging the assigned resources to reach end-user requirements and therefore, orthogonal decisions are taken. Before further discussing the current state-of-the-art and the related works, it is important to clearly

define the key concepts behind the methodology presented in this thesis.

With the term *application*, we may refer to any software that is possible to execute on the target architecture. However, we consider only applications that perform an elaboration and that do not require human interaction, such as a video encoder, a navigation system or scientific applications. Moreover, end-users or system administrators may have preferences or requirements on the application *extra-functional properties* (EFPs), such as execution time, energy consumption or quality of the results. For example, the user of a video encoder application would like to convert a video streaming with the highest quality, provided a throughput of at least 25fps; or the administrators of a navigation system would like to minimize the energy consumption while respecting a Service Level Agreement on the response time and quality of the results. We refer to the set of EFPs relevant for the end-user or system administrator as *metrics*, defining the application performance as a vector of values.

A large class of applications exposes tunable parameters that alter the application performance, named *software-knobs*. We may have application-specific software-knobs and application-agnostic software-knobs such as tile size or the number of Monte Carlo simulations. The main idea is that a change in the software-knobs configuration leads to a change in the application performance/output as well. A portion of all the possible software-knobs configurations is called *Design Space*. More details about this concept are discussed in Section 2.3. The main goal of an application autotuner is to automatically tune the software-knobs according to end-users or system administrator preferences or requirements. The major challenge is that the relation between a software-knobs configuration and the application performance is unknown and usually depends also on the underlying architecture and on the current input. For this reason, it is possible to use the characteristics of the current input, such as its size or autocorrelation, to better describe the relationship with the application performance. This set of values is named *input features*. Finally, the representation used by application autotuner to describe the relation between software-knobs configurations, input features and the application performance is named *application-knowledge*.

2.2 Previous Works

In the following section we will take a quick overview of the most recent works that can be found in literature and which are related to this thesis' work. After outlining them, a comparison with *Agora* is made, drawing attention to the main differences and contributions with respect to the current state-of-the-art.

2.2.1 Static vs Dynamic Autotuning

It is important to draw a line that separates two significant categories of autotuners in order to better grasp the nature of autotuning in its whole. As stated, in the autonomic computing world frameworks are born according to their vision on how to provide self-optimization properties. Often this is reflected with the selection of the most suitable configuration of software-knobs to leverage the assigned resources. Among these approaches, we can have *static autotuners* and *dynamic autotuners* that respectively select the suitable configuration **before** and **during** the production phase.

Static autotuners target software-knobs that tailor the application for the underlying

architecture, such as tiling size, loop unrolling factor, compiler options and algorithm selection. This tailoring process implies that static autotuners have to consider a fair amount of knobs with a large, possibly unbounded, domain of possible values. They are typically designed to find the best configuration that maximizes/minimizes an utility function in a reasonable amount of time and once they settle with it they are not willing to change it anymore.

In contrast, dynamic autotuners can continuously tune the knobs configuration at runtime leveraging informations about the actual execution context, which is bundled to the application-knowledge. They usually exploit this to predict the behaviour of a configuration and to drive the decision process.

2.2.2 State of the Art

Among the static autotuners, there are frameworks that aim at applying code or binary transformations to introduce the possibility of exploiting accuracy-throughput trade-offs. In this category we can find for instance Quick-Step [13], Paraprox [14] and PowerGAUGE [15]. The main focus of these frameworks is on how to expose trade-offs by introducing software-knobs. QuickStep and Paraprox target the applications parallel regions while performing the code transformation and without preserving the code semantics. PowerGAUGE works at assembly level and exploits a genetic algorithm to expose and optimize the accuracy-throughput trade-off. The parameter tuning phase is done at design-time by relying on the representative input set. We also have frameworks whose goal is exploring a very large Design Space in order to find the best configuration according to the application requirements before the production phase. ATune-IL [16], OpenTuner [17] and ATF [18] are some representatives of this category. ATune-IL provides a mechanism to prune and reduce the configuration space according to the code structure and to the dependencies among software knobs. This approach only aims at minimizing the execution time. OpenTuner uses a multi-armed bandit framework to find the best search algorithm for the target application. It is also possible to define the EFPs as a constrained multi-objective optimization problem. Finally ATF is a language agnostic autotuning framework that improves the OpenTuner strategies by considering also domain-constraints of the parameters. Keeping the focus on static autotuning, in the context of HPC many frameworks exist aiming at optimizing specific domains. ATLAS [19] for matrix multiplication routing, FTTW [20] for FFTs operations, OSKI [21] for sparse matrix kernels, SPIRAL [22] for digital signal processing, CLTune [23] and GLINDA [24] for OpenCL applications, Patus [25] and Sepya [26] for stencil computations.

On the dynamic autotuning category, there are frameworks that target streaming applications and that typically learn the application-knowledge at design-time, to be leveraged during production. The Green framework [27] , the Sage framework [28] and PowerDial [3] are some examples. They focus on how to provide reaction mechanisms to a streaming application and therefore they use representative inputs to derive the application-knowledge. Green framework, after an integration step, performs at first a Design Space Exploration to generate QoS Data and after that an external program in MATLAB performs curve fitting and interpolation with re-calibrating steps if necessary. Sage targets specific CUDA kernels. It takes as input the original kernel and a metric that represents the elaboration quality. In the first step, it analyses the code trying to

find some transformation opportunities. After that, it uses a greedy approach to select the most suitable kernel version and tune its parameters to minimize the execution time while monitoring quality oaddressver time in order to set a more accurate configuration in case of quality drops. The PowerDial framework takes as input the source code of the application, the command-line options, a representative input set and an output abstraction in order to measure the accuracy. It performs a Design Space Exploration to sort the software-knobs configuration according to a speed-up with respect to a baseline throughput. Then it generates a binary with a manager based on control theory that selects the speedup required to reach the target throughput. Another group of frameworks that adapt an application also in a proactive way exploits the application input features to learn the knowledge during design-time. Petabricks [29] and Capri [30] are some renowned examples. The idea behind is to derive the knowledge assuming the possibility to select which input to consider in the representative set used during learning process. Petabricks is a language that exposes algorithm choices. It analyses the code and generates a configuration file that selects the fastest algorithm and configuration according to the input size. This framework has been subject to major improvements during time to leverage accuracy-throughput trade-offs, check quality levels at runtime or taking into consideration input features besides the input size during design time. Capri framework is another work which uses a set of representative inputs at design time to model a cost metric (e.g. execution time, energy, etc.). Its controller aims at finding the software-knobs configuration at runtime that minimizes the cost function given an error bound and a probability that measures how the selected bound is satisfied according to the representative inputs. Finally, an interesting work that adapt an application in a proactive way without learning the application-knowledge at design time is IRA [31]. This framework defines the concept of *canary input* as the smallest sub-sampling of the actual input, which has the same property of the original input. Using the canary for a runtime parameter exploration for each data to be processed, it then selects the fastest configuration of knobs resulting within a given bound on the minimum accuracy.

2.2.3 Comparison with the state-of-the-art

For what concerns the *static* autotuning world, the possibility of having an unbounded exponential growth of the Design Space forces the vast majority of frameworks to search for the best configuration that maximizes/minimizes an utility function in a reasonable amount of time. Even if a small fraction of those is able to perform such exploration at runtime, once they settle with the chosen configuration they're not willing to change it anymore regardless potential changes on the underlying architecture or in the input space. For instance ATLAS, FTTW, OSKI, SPIRAL, CLTune, GLINDA, QuickStep, Paraprox and PowerGAUGE are all frameworks that take tuning decisions at design-time and falls under this category. Even ATune-IL, OpenTuner and the ATF framework, given that the tuning phase is done at design-time, usually target software-knobs loosely coupled with the inputs. As highlighted, the output of the tuning process is a single software-knob configuration, not the application-knowledge required to adapt dynamically during the production phase.

On the contrary, *dynamic* autotuners can continuously tune the knobs configuration at runtime leveraging informations about the execution context. Even if the tuning process can happen at runtime, the overall process remains fundamentally static with a predefined

workflow. A great part of them typically learns the application-knowledge at design-time to be leveraged during the production phase. Green, Sage and PowerDial frameworks are examples in this category. Capri and Petabricks are somewhat different since they try to adapt an application also in a proactive way by exploiting input features. However, they still learn the application-knowledge at design-time. Moreover, the methodology used to derive the application-knowledge assumes the possibility to select which input to consider in the representative set used during the learning process while *Agora* has been designed to use directly the production input and therefore it is not possible to apply the same approaches. Furthermore, these frameworks are able to express a trade-off between a quality metric and an additional EFP only, while the proposed methodology is able to consider an arbitrary number of EFPs. It is worth to mention that *Agora* is partly inspired by the Petabricks work especially concerning the approach towards input sensitivity but tries to offer some advantages by being able to extrapolate the application-knowledge at runtime and in a distributed manner. Moreover, Petabricks requires the target application to be ported on their own language to expose algorithmic choices. Finally, another interesting work that inspired the *Agora* development is the IRA framework but the main drawback of this methodology is that the presented sub-sampling technique applies to matrix-like inputs only, therefore limiting the applicability of the framework.

In conclusion, *Agora* proposes to be an enhancement of an existing dynamic autotuner, giving the end-user the possibility to perform a Design Space Exploration, and thus the knowledge collection, at runtime with respect to the application execution. The comparison with the current state-of-the-art reveals the impossibility to make this utilization and even the few frameworks that enable it, still remains essentially static within a predefined workflow. The amount of configuration points that has to be chosen in order to generate the application-knowledge is fixed while *Agora* gives the possibility to automate this choice. Finally, the most important concept that has to be stressed is that *Agora*'s methodology, which will be outlined in details in Chapter 3, is not offering a new revolutionary process modeling approach but whole framework itself gives the possibility to do something different. The application can be adaptive as its learning process, regardless of the model complexity.

2.3 Design of Experiments

Within the theory of optimization, an experiment is a series of tests in which the input variables are changed according to a given rule in order to identify the reasons for the changes in the output response. According to Montgomery [32]:

> Experiments are performed in almost any field of enquiry and are used to study the performance of processes and systems. [...] The process is a combination of machines, methods,people and other resources that transforms some input into an output that has one or more observable responses. Some of the process variables are controllable, whereas other variables are uncontrollable, although they may be controllable for the purpose of a test. The objectives of the experiment include: determining which variables are most influential on the response, determining where to set the influential controllable variables so that the response is almost always near the desired optimal value, so that the variability in the response is small, so that the effect of uncontrollable variables are minimized.

Thus, the purpose of experiments is essentially optimization. The Design of Experiments (DoE) is the name given to the techniques used for guiding the choice of the experiments to be performed in an efficient way. Usually, data subject to experimental

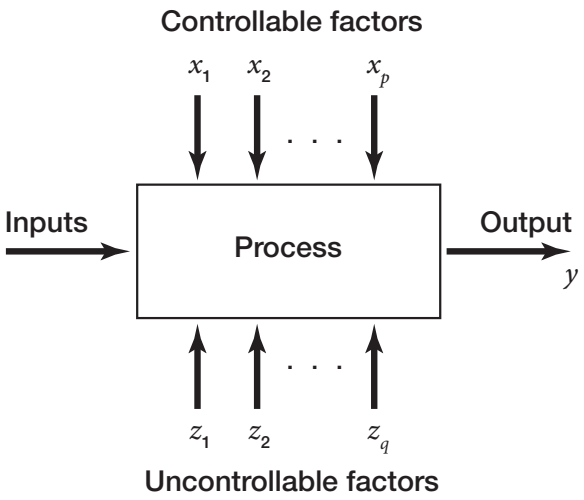


Figure 2.2: *General model of a process in a system.*

error (noise) are involved, and the results can be significantly affected by noise. Thus, it is better to analyze the data with appropriate statistical methods. The basic principles of statistical methods in experimental design are *replication* and *randomization*. Replication is the repetition of the experiment in order to obtain a more precise result (sample *mean value*) and to estimate the experimental error (sample *standard deviation*). Randomization refers to the random order in which the runs of the experiment are to be performed. In this way, the conditions in one run neither depend on the conditions of the previous run nor predict the conditions in the subsequent runs.

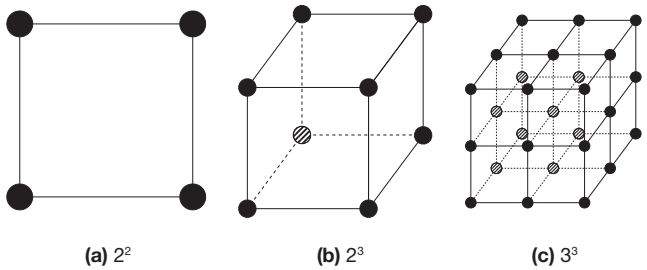
2.3.1 Terminology

In order to perform a DoE it is necessary to define the problem and choose the variables, which are called *factors* or parameters by the experimental designer. A *design space*, or region of interest, must be defined, that is, a range of variability must be set for each variable. The number of values the variables can assume in DoE is restricted and generally small. Therefore, we can deal either with qualitative discrete variables, or quantitative discrete variables. Usually the DoE technique and the number of levels should be selected according to the number of experiments which can be afforded. By *levels* we identify the number of different values a variable can assume according to its discretization. In experimental design, the objective function and the set of experiments to be performed are called *response variable* and *sample space* respectively.

The experiments are used to study the performance of applications/processes and systems. These entities can be represented by the model shown in Figure 2.2. We can usually visualize the process as a combination of operations, machines, methods, people, and other resources that transforms some input (often a material) into an output that has one or more observable response variables. Some of the process variables and material properties $x_1, x_2, ..., x_p$ are **controllable** (e.g. software-knobs), whereas other variables $z_1, z_2, ..., z_q$ are **uncontrollable** (e.g. input features). Retrieving the output produced by each experiment is what we have referred to as Design Space Exploration.

Experiment number	Factor level			Response variable	Two- and three-factors interactions			
	X_1	X_2	X_3		$X_1 \cdot X_2$	$X_1 \cdot X_3$	$X_2 \cdot X_3$	$X_1 \cdot X_2 \cdot X_3$
1	-1 *(l)*	-1 *(l)*	-1 *(l)*	$y_{l,l,l}$	+1	+1	+1	-1
2	-1 *(l)*	-1 *(l)*	+1 *(h)*	$y_{l,l,h}$	+1	-1	-1	+1
3	-1 *(l)*	+1 *(h)*	-1 *(l)*	$y_{l,h,l}$	-1	+1	-1	+1
4	-1 *(l)*	+1 *(h)*	+1 *(h)*	$y_{l,h,h}$	-1	-1	+1	-1
5	+1 *(h)*	-1 *(l)*	-1 *(l)*	$y_{h,l,l}$	-1	-1	+1	+1
6	+1 *(h)*	-1 *(l)*	+1 *(h)*	$y_{h,l,h}$	-1	+1	-1	-1
7	+1 *(h)*	+1 *(h)*	-1 *(l)*	$y_{h,h,l}$	+1	-1	-1	-1
8	+1 *(h)*	+1 *(h)*	+1 *(h)*	$y_{h,h,h}$	+1	+1	+1	+1

Figure 2.3: *Example of 2^3 full factorial experimental design.*



(a) 2^2 **(b)** 2^3 **(c)** 3^3

Figure 2.4: *Examples of L^k full factorial experimental designs with $\{L = 2, k = 2\}$ (a), $\{L = 2, k = 3\}$ (b) and $\{L = 3, k = 3\}$ (c).*

2.3.2 Techniques

To end this section, three of the most relevant design categories related to this thesis' work are presented and briefly discussed to introduce the reader to the topic, showing the main techniques which are used in practice. The designs are divided into *Factorial Designs*, *Response-Surface Designs* and *Randomized Designs*. The following material is referring to [33].

Full Factorial (Factorial)

Full factorial is probably the most common and intuitive strategy of experimental design. In the most simple form, the two-levels full factorial, there are k factors and $L = 2$ levels per factor. The samples are given by every possible combination of the factors values. Therefore, the sample size is $N = 2^k$. Starting from any sample within the full factorial scheme, the samples in which the factors are changed one at a time are still part of the sample space. This property allows for the effect of each factor over the response variable not to be confounded with the other factors. Let us consider a full factorial design with three factors and two levels per factor (Figure 2.3). The full factorial is an *orthogonal* experimental design method. The term orthogonal derives from the fact that the scalar product of the columns of any two-factors is zero.

The idea of the 2^k full factorial experimental designs can be easily extended to the general case where there are more than two factors and each of them have a different number of levels. The sample size of the adjustable full factorial design with k factors $X_1, ..., X_k$, having $L_1, ..., L_k$ levels, is $N = \prod_{i=1}^{k} L_i$. In Figure 2.4 we can see some visual representations of L^k full factorial designs.

Block \ Parameter			
	±	±	0
	±	0	±
	0	±	±
	0	0	0

(a) *3 parameters: 3 out of 3 blocks with 2^2 full-factorial, plus the central point, 13 samples overall, 10 coefficients needed for fitting a 2^{nd} order interpolating polynomial.*

Block \ Parameter				
	±	±	0	0
	±	0	±	0
	±	0	0	±
	0	±	±	0
	0	±	0	±
	0	0	±	±
	0	0	0	0

(b) *4 parameters: 6 out of 6 blocks with 2^2 full factorial, plus the central point, 25 samples overall, 15 coefficients needed for fitting a 2^{nd} order interpolating polynomial.*

Figure 2.5: *Box-Behnken tables for $k = 3$ (a) and $k = 4$ (b).*

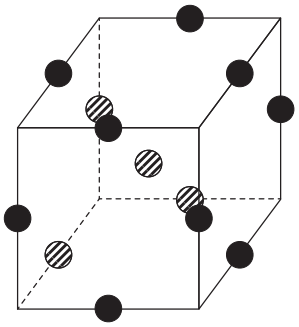


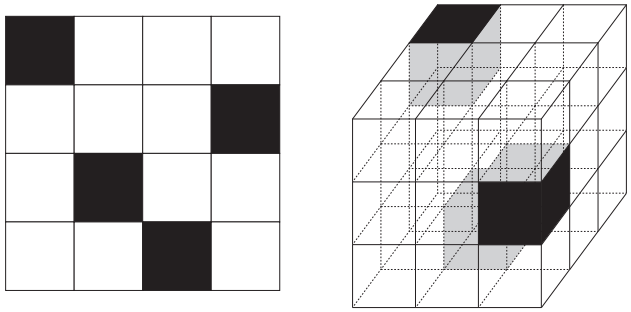
Figure 2.6: *Example of Box-Behnken experimental design for $k = 3$.*

The advantage of full factorial designs is that they make a very efficient use of the data and do not confound the effects of the parameters, so that it is possible to evaluate the main and the interaction effects clearly. On the other hand, the sample size **grows exponentially** with the number of parameters and the number of levels.

Box-Behnken (Response-Surface)

This technique is not exploited by the proposed framework but we think that it is still worth to mention as future developments may consider its integration. *Box-Behnken* are incomplete three-levels factorial designs. They are built combining two-levels factorial designs with incomplete block designs in a particular manner. Box-Behnken designs were introduced in order to limit the sample size as the number of parameters grows. In Box-Behnken designs, a block of samples corresponding to a two-levels factorial design is repeated over different sets of parameters. The parameters which are not included in the factorial design remain at their mean level throughout the block. The type (full or fractional), the size of the factorial, and the number of blocks which are evaluated, depend on the number of parameters and it is chosen so that the design meets, exactly or approximately, the criterion of *rotatability*. An experimental design is said to be rotatable if the variance of the predicted response at any point is a function of the distance from the central point alone.

Since there is not a general rule for defining the samples of the Box-Behnken designs, tables are given by the authors for the range from three to seven, from nine to twelve and for sixteen parameters. Figure 2.5 shows two examples. Each line stands for a factorial



(a) $k = 2, N = 4$ **(b)** $k = 3, N = 3$

Figure 2.7: *Example of latin hypercube designs.*

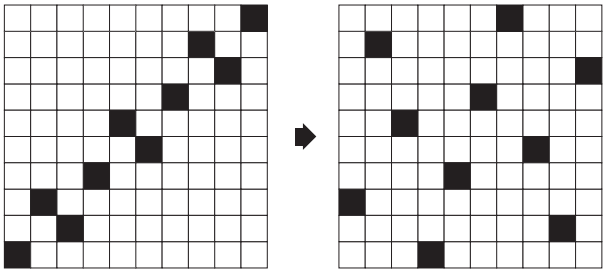


Figure 2.8: *Example of correlation reduction in a latin hypercube DoE with* $k = 2$*,* $N = 10$*.*

design block, the symbol ± individuates the parameters on which the factorial design is made, "0" stands for the variables which are blocked at the mean level.

Let us consider the Box-Behnken design with three parameters (Figure 2.5a), in this case a 2^2 full factorial is repeated three times:

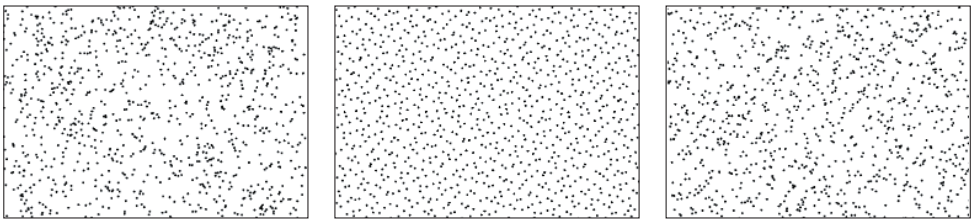
- on the first and the second parameters keeping the third parameter at the mean level (samples: *llm,lhm,hlm,hhm*),
- on the first and the third parameters keeping the second parameter at the mean level (samples: *lml,lmh,hml,hmh*),
- on the second and the third parameters keeping the first parameter at the mean level (samples: *mll,mlh,mhl,mhh*),

then the central point (*mmm*) is added. Graphically, the samples are at the mid-points of the edges of the design space and in the centre (Figure 2.6).

Latin Hypercube (Randomized)

In *latin hypercube* DoE the design space is subdivided into an orthogonal grid with N elements of the same length per parameter. Within the multi-dimensional grid, N sub-volumes are individuated so that along each row and column of the grid only one sub-volume is chosen. In Figure 2.7, by painting the chosen sub volumes black gives, in two dimensions, the typical crosswords-like graphical representation of latin hypercube designs. Inside each sub-volume a sample is randomly chosen.

It is important to choose the sub-volumes in order to have no spurious correlations between the dimensions or, which is almost equivalent, in order to spread the samples



(a) Random **(b)** Sobol **(c)** Latin hypercube

Figure 2.9: *A comparison between different space filling DoE techniques for* $k = 2$*,* $N = 1000$*.*

all over the design space. For instance, a set of samples along the design space diagonal would satisfy the requirements of a latin hypercube DoE, although it would show a strong correlation between the dimensions and would leave most of the design space unexplored. There are many techniques which are used to reduce the correlations in latin hypercube designs which although won't be discussed further as they're not in the scope of this thesis.

Figure 2.8 shows the effect of the correlation reduction procedure for a case with two parameters and ten samples. The correlation reduction was obtained using one of the above-mentioned methods. Finally, Figure 2.9 shows a comparison between random, Sobol (another randomized design technique which has not been discussed) and latin hypercube space filling DoE techniques on a case with two parameters and a thousand samples. It is clear that the random method is not able to completely avoid samples clustering. Using latin hypercubes the samples are more uniformly spread in the design space. The Sobol sequence gives the most uniformly distributed samples.

2.4 Process Modeling

We have seen what is a Design of Experiments and some of the main techniques that revolve around this world. But why do experiments exist and what's their purpose in our particular scenario? In its simplest form, an experiment aims at predicting the outcome of a general process by introducing a change of the preconditions, which is represented by one or more independent variables (*predictor variables*), which till now we've referred to as software-knobs. The change in one or more independent variables is generally hypothesized to result in a change in one or more dependent variables (*response variables*), also referred previously as metrics. As stated, experimental design involves not only the selection of suitable dependent variables, but planning the delivery of the experiment under statistically optimal conditions given the constraints of available resources.

With this premise, *process modeling* [34] aims at giving a concise description of the total variation in one quantity, y, by partitioning it into:

1. A deterministic component given by a mathematical function of one or more other quantities, $x_1, x_2, ..., x_n$.

2. A random component that follows a particular probability distribution.

For example, the total variation of the measured pressure of a fixed amount of a gas in a tank can be described by partitioning the variability into its deterministic part, which is

a function of the temperature of the gas, plus some left-over random error. Charles' Law states that the pressure of a gas is proportional to its temperature under the conditions described here, and in this case most of the variation will be deterministic. However, due to measurement error in the pressure gauge, the relationship will not be purely deterministic. The random errors cannot be characterized individually, but will follow some probability distribution that will describe the relative frequencies of occurrence of different-sized errors.

There are three main parts to every process model. These are:

1. The response variable, usually denoted by y.
2. The mathematical function that describes it, usually denoted as $f(\vec{x}; \vec{\beta})$.
3. The random errors, usually denoted by ϵ.

The general form of the model is

$$y = f(\vec{x}; \vec{\beta}) + \epsilon$$

The response variable y is a quantity that varies in a way that we hope to be able to summarize and exploit via the modeling process. Generally it is known that the variation of the response variable is systematically related to the values of one or more other variables before the modeling process is begun, although testing the existence and nature of this dependence is part of the modeling process itself.

The mathematical function consists of two parts. These parts are the predictor variables $x_1, x_2, ..., x_n$ and the parameters $\beta_0, \beta_1, ..., \beta_n$. The predictor variables are observed along with the response variable. They are the quantities described as the inputs to the mathematical function $f(\vec{x}; \vec{\beta})$. The collection of all of the predictor variables is denoted by $\vec{x}$ for short. The parameters are the quantities that will be estimated during the modeling process. Their true values are unknown and unknowable, except in simulation experiments. As for the predictor variables, the collection of all of the parameters is denoted by $\vec{\beta}$ for short. The parameters and predictor variables are combined in different forms to give the function used to describe the deterministic variation in the response variable.

Process models are used for four main purposes: *Estimation*, *Prediction*, *Calibration* and *Optimization*. In our particular scenario, this thesis revolves around the prediction purpose. The goal of prediction is to determine either

1. the value of a new observation of the response variable, or
2. the values of a specified proportion of all future observations of the response variable

for a particular combination of the values of the predictor variables. Predictions can be made for any combination of predictor variable values, including values for which no data have been measured or observed.

The basic steps used for model-building are the same across all modeling methods. The details vary somewhat from method to method but in general it is always provided a framework in which the results from almost any method can be interpreted and understood. The basic steps of the model-building process are:

1. Model selection.
2. Model fitting.
3. Model validation.

In the model selection step, usually process knowledge and assumption about data are used to determine the form of the model to be fit to the data. Then, using the selected model and possibly information about the data, an appropriate model-fitting method is used to estimate the unknown parameters $\vec{\beta}$ in the model. When the parameter estimates have been made, the model is then carefully assessed during validation phase to see if the underlying assumptions of the analysis appear plausible. If the assumptions seem valid, the model can be used to answer the scientific or engineering questions that prompted the modeling effort. If the model validation identifies problems with the current model, however, then the modeling process must (or at least should) be repeated.

In this class of problems which we will refer to as *regression problems*, inside the model validation phase there are usually two scoring factors which are taken into consideration. One is the *coefficient of determination* R^2 which measures the fraction of the total variability in the response that is accounted for by the model. Since unfortunately a high R^2 value does not always guarantee that the model fits the data well, another scoring metric which is commonly taken into consideration is the *residual*. The residuals from a fitted model are the differences between the responses observed at each combination values of the explanatory variables and the corresponding prediction of the response computed using the modeling function. Mathematically, the definition of the residual for the i^{th} observation in the data set is written

$$e_i = y_i - f(\vec{x_i}; \vec{\beta}) = y_i - f_i$$

with y_i denoting the i^{th} response in the data set and $\vec{x_i}$ represents the list of explanatory variables, each set at the corresponding values found in the i^{th} observation in the data set. The residuals are used in combination to produce a general scoring system in regression problems. For instance the R^2 metric is defined as

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

where $SS_{tot} = \sum_i (y_i - \bar{y})^2$ is the total sum of squares (proportional to the variance of the data), $SS_{res} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$ is the sum of squares of the residuals and $\bar{y} = \frac{1}{n}\sum_i^n y_i$ is the mean of the observed data. The *mean absolute error* is another common scoring metric and it is defined as

$$MAE = \frac{1}{n}\sum_i^n |y_i - f_i| = \frac{1}{n}\sum_i^n |e_i|.$$

A final error metric worth to mention is the *mean absolute percentage error*. The idea of this metric is to be sensitive to relative errors. For example, it does not change by a global scaling of the target variable. It is defined as

$$MAPE = \frac{1}{n}\sum_i^n \frac{|y_i - f_i|}{max(\epsilon, |y_i|)} = \frac{1}{n}\sum_i^n \frac{|e_i|}{max(\epsilon, y_i)}$$

where ϵ is an arbitrary small yet strictly positive number to avoid undefined results when y is zero.

2.5 Clustering

As stated, the relation between a software-knobs configuration and the application performance is unknown and usually depends also on the underlying architecture and on the current input. We've referred to the relation between these characteristics, such as size or autocorrelation, as input features.

A daunting challenge faced by program performance autotuning is input sensitivity, where the best autotuned configuration may vary with different input sets. For a large class of problems, the best optimization to use depends on the input data being processed. For example, sorting an almost-sorted list that contains many repeated values can be done most efficiently with a different algorithm than one optimized for sorting random data. A common solution is to search for good optimizations on *every* training input, based on which, it builds a process model that predicts the best optimization to use according to the features of the new input. These choices are often sensitive to input features that are domain-specific and require deep, possibly expensive, analysis to extract. It can happen that following this philosophy hundreds or thousands of training inputs are often prepared such that the complex input feature space could get well covered. The large number of inputs, combined with the long training time per input, means searching for good configurations for every training input may take months or years for a program, which makes the traditional exhaustive approach impractical. An intuitive solution is through *clustering*. It first splits training inputs into groups with the members in the same group featuring similar values of predefined input features, and then finds the configuration that works the best for the centroid of each group. In a production run, when a new input comes, the program extracts its feature values, based on which, it identifies the cluster that the input resembles the most, and then runs with that cluster's configuration. [29]

Clustering deals with finding a structure in a collection of unlabeled data. A cluster is therefore a collection of objects which are "similar" between them and are "dissimilar" to the objects belonging to other clusters. In this thesis we introduce two categories which are relevant to the subject: *partitioning* and *density-based*. Partitioning methods relocate instances by moving them from one cluster to another, starting from an initial partitioning. Such methods typically require that the number of clusters will be pre-set by the user. To achieve global optimality in partitioned-based clustering, an exhaustive enumeration process of all possible partitions is required. Because this is not feasible, certain greedy heuristics are used in the form of iterative optimization. Density-based methods assume that the points that belong to each cluster are drawn from a specific probability distribution. The overall distribution of the data is assumed to be a mixture of several distributions. The aim of these methods is to identify the clusters and their distribution parameters. [35]

2.5.1 Algorithms

For each category, one of the most relevant clustering techniques is presented and briefly discussed to introduce the reader to the topic. The following material refers to [36].

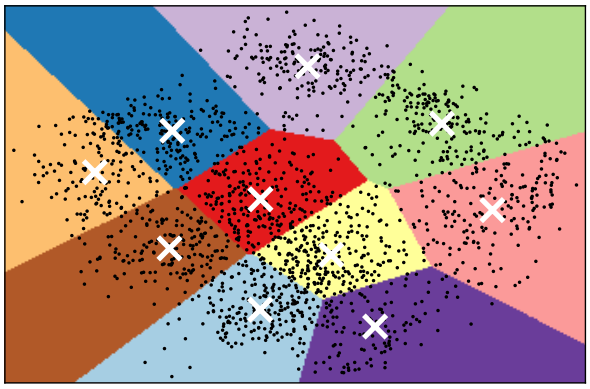


Figure 2.10: *K-means example on PCA (Principal Component Analysis) reduced data. Centroids are marked with a white cross.*

K-means (Partitioning Method)

The simplest and most commonly used algorithm, employing a squared error criterion is the *K-means* algorithm. This algorithm partitions the data into K clusters $(C_1, C_2, ..., C_K)$, represented by their centers or means. The center of each cluster is calculated as the mean of all the instances belonging to that cluster. Algorithm 1 presents a simplified pseudo-code of the K-means algorithm. The algorithm starts with an initial set of cluster centers, chosen at random or according to some heuristic procedure. In each iteration, each instance is assigned to its nearest cluster center according to the Euclidean distance between the two. Then the cluster centers are re-calculated.

Data: S (instance set), K (number of clusters)
Result: $(C_1, C_2, ..., C_K)$
Initialize K cluster centers ;
while *termination condition is not satisfied* **do**
    Assign instances to the closest cluster center ;
    Update cluster centers based on the assignment ;
end

Algorithm 1: How the K-means method finds cluster representatives.

The center of each cluster is calculated as the mean of all the instances belonging to that cluster:

$$\mu_k = \frac{1}{N_k} \sum_{q=1}^{N_k} x_q$$

where N_k is the number of instances belonging to cluster k and μ_k is the mean of the cluster k.

A number of convergence conditions are possible. For example, the search may stop when the partitioning error is not reduced by the relocation of the centers. This indicates that the present partition is locally optimal. Other stopping criteria can be used also such as exceeding a pre-defined number of iterations. The K-means algorithm may be viewed as a gradient-decent procedure, which begins with an initial set of K cluster-centers and iteratively updates it so as to decrease the error function. A visualization example is shown in Figure 2.10.

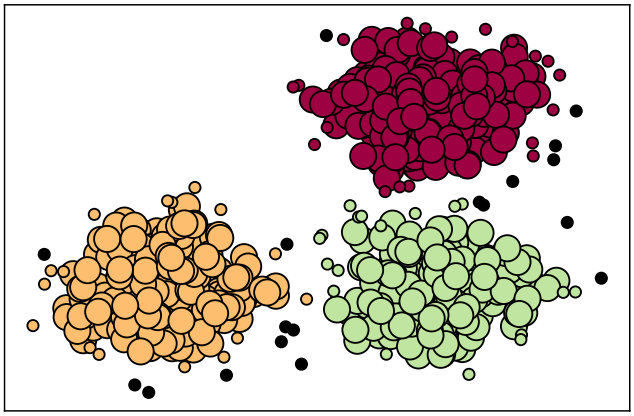


Figure 2.11: *DBSCAN example. Estimated number of clusters is 3. The outliers are in black.*

DBSCAN (Density-based Method)

Like other methods of this kind, *DBSCAN* is designed for discovering clusters of arbitrary shape which are not necessarily convex, namely:

$$x_i, x_j \in C_k$$

This does not necessarily imply that:

$$\alpha \cdot x_i + (1 - \alpha) \cdot x_i \in C_k$$

The idea is to continue growing the given cluster as long as the density (number of objects or data points) in the neighborhood exceeds some thresh-old. Namely, the neighborhood of a given radius has to contain at least a mini-mum number of objects. The DBSCAN algorithm (density-based spatial clustering of applications with noise) discovers clusters of arbitrary shapes and is efficient for large spatial databases. The algorithm searches for clusters by searching the neighborhood of each object in the database and checks if it contains more than the minimum number of objects. The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to K-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core samples*, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples).

A visualization example is shown in Figure 2.11. The color indicates cluster membership, with large circles indicating core samples found by the algorithm. Smaller circles are non-core samples that are still part of a cluster. Moreover, the outliers are indicated by black points.

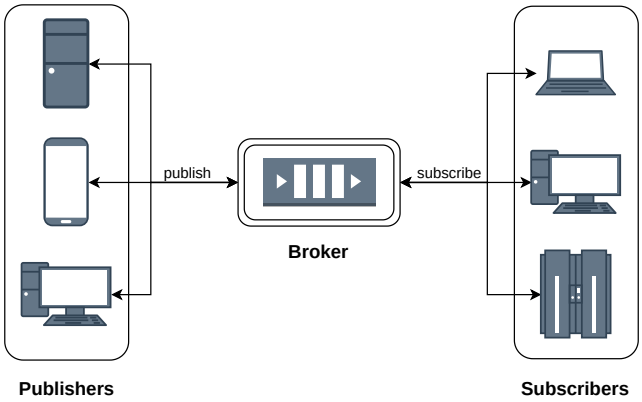


Figure 2.12: *The architecture of MQTT.*

2.6 Distributed Communication

As we stated during introduction, this thesis's project has been designed to be applied in a wide range of scenarios but manly targets the context of HPC. Inside a distributed scenario like this, MQTT has been chosen as the network message protocol in order to enable the exchange of informations in a client-server fashion, where *Agora* is deployed as a server machine and the target applications are running as client instances. More on this will be covered while discussing about the implementation details in Chapter 4.

MQTT has been manly utilized as part of many IoT (*Internet of Things*) gadgets and instant message delivery systems because it was intended to work on low-power machines as a light-weight protocol [37]. It still remains a valuable message protocol also in the context of HPC and embedded systems thanks to the low power consumption property. It is a standardized publish/subscribe *push* protocol that was released by IBM in 1999. MQTT was planned to send a data accurately under the long network delay and low-bandwidth network condition.

2.6.1 Architecture

Some of the MQTT key concepts are listed in the following section in order to explain how the architecture works. A full in-depth summary can be found on the survey [38] they are taken from.

MQTT uses the client/server publish-subscribe model. Every device that is connected to a server, using TCP known as (broker) message in MQTT is a discrete chunk of data and it is ambiguous for the broker. Therefore, MQTT is a message oriented protocol. The address that the message published to it is called *topic*. The device may subscribe to more than one topics, and it receives all messages that are published to these topics. The *broker* is a central device and its main responsibilities are processing the communication between MQTT clients and distributing the messages between them based on their interested topics. Upon receiving the message, the broker must search and find all the devices that own a subscription to this topic.

MQTT architecture contains three components (Figure 2.12). Those are a publisher, a broker, and a subscriber. The device that is interested in a specific topics registers on it as a subscriber to be informed when the publishers publishing his topics by the

broker. The publisher transfers the information to the subscribers via the broker (i.e. the interested entities).

2.6.2 Quality of Service

There are three levels of Quality of Service (*QoS*) in order to maintain the reliability of messages in the MQTT. Level 0 is called one delivery (**at most**) and the messages are delivered based on the effort of the network. Level 1 is one delivery (**at least**) and the messages are being sent at least once and the duplicate may exist in messages. The last level is Level 2, which is called one delivering (**exactly**). An additional protocol is required in this level to guarantee that the message is delivered only once (i.e. Highest level of QoS). Table 2.1 provides a summary of QoS levels and their meanings.

QoS Level	Meaning
Level 0	A message is delivered at most once and no acknowledgement of receiving is required.
Level 1	Every message is delivered at least once and a confirmation of receiving a message is required.
Level 2	A four-way handshake mechanism is used exactly once for the delivery of a message.

Table 2.1: *Quality of Service levels of MQTT.*

2.7 mARGOt Framework

mARGOt is a dynamic autotuning framework that was developed at *Politecnico di Milano* and which *Agora* has been based on. In this section a general overview of *mARGOt* and its functionalities is given to get a grasp of what's going on under the hood [8], [39].

mARGOt is an adaptation layer that provides to the target application mechanism to adapt in a reactive and proactive fashion, based on the application-knowledge provided. From an implementation point of view, *mARGOt* is a C++ library that is linked to the target application and works at the function level. It employs separation of concerns between functional and extra-functional requirements. End-user might define or change requirements at runtime, according to application phases. Moreover, by using feedback information from runtime monitors, it is possible to react to changes in the execution environment, providing to the application the most suitable software-knobs configuration. Furthermore, it leverages input features to identify and seize optimisation opportunities according to the current input.

Figure 2.13 shows an overview of *mARGOt* and how it interacts with an application. To simplify the description of the autotuning methodology, we consider an application that is composed of a single phase. However, *mARGOt* is designed to manage different phases, or blocks of code, independently. Each phase is composed of a single kernel g that elaborates an input i to generate the desired output o. Moreover, the kernel algorithm is assumed to expose software-knobs that alter its EFPs, such as the number of Monte Carlo simulations or the parallelism level. Let $\bar{x} = [x_1, ..., x_n]$ the vector of software-knobs, then we might define a kernel as $o = g(\bar{x}, i)$. Within this abstraction, the end-user requirements are defined as follows. The metrics of interest (i.e. EFPs) are seen as the vector $\bar{m} = [m_1, ..., m_n]$. Supposing that the application developers are

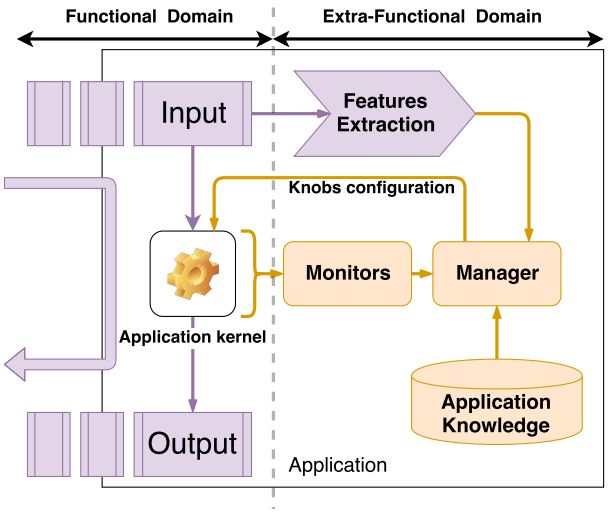


Figure 2.13: *Global architecture of the mARGOt framework. Purple elements represent application code, while orange elements represent mARGOt high-level components. The black box represents the executable boundary.*

capable of extracting features of the current inputs, such properties are represented by the vector $\bar{f} = [f_1, ..., f_n]$. The end-user can define the application requirements as in Equation 2.1:

$$\begin{aligned} \max(\min) \quad & r(\bar{x}; \bar{m} \mid \bar{f}) \\ \text{s.t.} \quad & C_1 : \omega_1(\bar{x}; \bar{m} \mid \bar{f}) \quad \propto \quad k_1 \quad with\ \alpha_1\ confidence \\ & C_2 : \omega_2(\bar{x}; \bar{m} \mid \bar{f}) \quad \propto \quad k_2 \\ & \dots \\ & C_n : \omega_n(\bar{x}; \bar{m} \mid \bar{f}) \quad \propto \quad k_n \end{aligned} \tag{2.1}$$

where r denotes the objective function (named *rank* in *mARGOt* context), defined as a composition of any of the variables defined in $\bar{x}$ or $\bar{m}$, using their mean values. Let C be the set of constraints, where each C_i is a constraint expressed as the function ω_i, defined over the software-knobs or the EFPs, that must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$ with a threshold value k_i and with a confidence α_i (if ω_i targets a statistical variable). Since it is agnostic about the distribution of the target parameter, the confidence is expressed as the number of times to consider its standard deviation. If the application is input-dependent, the value of the rank function r and the constraint functions ω_i also depend on the features of the input $\bar{f}$.

In this formulation, the main goal of *mARGOt* is to solve the optimization problem: finding the configuration $\hat{\bar{x}}$ that satisfies all the constraints C and maximizes (minimizes) the objective function r, given the current input i. The application must have a configuration to use even if it is not feasible to satisfy all the constraints. For this reason, *mARGOt* might relax constraints until a feasible solution is found, starting by relaxing the lowest priority constraint. Therefore, the end-user must sort the set of constraints by their priority.

2.7.1 Components

In this subsection *mARGOt* three main components and briefly described to give the reader a general understanding of the internal units behaviour.

Application Knowledge

As we've seen, for a generic application, the relation between software-knobs, EFPs of interest and input features is complex and unknown a priori. Therefore, we need a model of the application extra-functional behavior to solve the optimization problem stated in Equation 2.1. We've referred to this as the application-knowledge. *mARGOt* uses a list of *Operating Points* (OPs) as application-knowledge, where each Operating Point θ states the target software-knob configuration and the achieved EFPs with the given input features. For instance it can be defined as

$$\theta = \{x_1, \ldots, x_n, f_1, \ldots, f_n, m_1, \ldots, m_n\}$$

This list incorporates informations about software-knobs configurations, metrics of interest and (potentially) input features. The OPs list is considered a required input. Therefore, *mARGOt* is agnostic on the methodology used to obtain the application-knowledge. Even if the latter is considered an input, it is of paramount importance to *mARGOt* for solving the optimisation problem.

The application-knowledge is codified inside a configuration file that is parsed by *mARGOt* before beginning the tuning phase. More details on this will be discussed further in Chapter 4.

Monitors

This module provides to *mARGOt* the ability to observe the actual behaviour of either the application or the execution environment. The application-knowledge defines the expected behaviour of the application. However, it might change according to the evolution of the system. For example, a power capper might reduce the frequency of the processor due to thermal reasons. In this case, we would expect that the application notices a degradation in its performance and it reacts, by using a different configuration to compensate. This adaptation is possible only if we have feedback information.

From the implementation point of view, *mARGOt* provides a suite of predefined monitors with broad applicability both at high- and low-level. Some examples of monitors implemented in *mARGOt* are listed in Table 2.2 with a brief description.

Moreover, the monitors are implemented using a modular approach. In this way, application developers might implement a custom monitor for observing an application-specific metric easily.

Application Manager

This component is the core of the *mARGOt* dynamic autotuner, which provides the self-optimization capability using a lightweight framework. From the methodology point of view, this component is in charge of solving the optimisation problem stated in Eq. 2.1: to find the software-knobs configuration $\hat{\overline{x}}$, while reacting to changes in the execution environment and adapting proactively according to input features.

Name	Description
Time Monitor	This monitor reads the time elapsed between a start point and a stopping point.
Throughput Monitor	This monitor computes the throughput as the amount of elaborated data over the observed time interval.
Memory Monitor	This monitor observes the resident set size of the virtual memory that the process is using.
System CPU Usage Monitor	This monitor computes the average utilisation of the processors at the system-level.
Process CPU Usage Monitor	This monitor is similar to the System CPU Usage Monitor, but it computes the average utilisation of the processor by the application, defined as the time the application spent executing on the processors over the elapsed time.

Table 2.2: *List of some monitors implemented in mARGOt.*

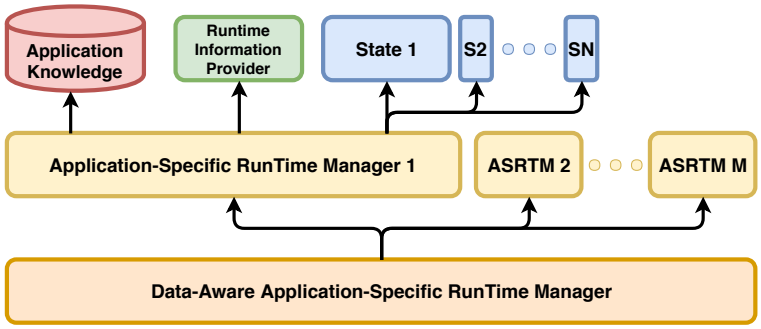


Figure 2.14: *Overview of the Application Manager implemented in mARGOt, based on a hierarchical approach.*

From the implementation point of view, the application manager has a hierarchical structure, as shown in Figure 2.14 where each sub-component solves a specific problem. The *Data-Aware Application-Specific Run-Time Manager* (DA AS-RTM) provides a unified interface to application developers to set or change the application requirements, to set or change the application-knowledge and to retrieve the most suitable configuration $\hat{\overline{x}}$. Internally, the DA AS-RTM clusters the application knowledge according to input features $\overline{f}$, creating an *Application-Specific Run-Time Manager* (AS-RTM) for each cluster of Operating Points with the same input features. Therefore, the application knowledge implicitly defines the clusters of Operating Points. Given the input features of the current input, the DA AS-RTM selects the cluster with features closer to the ones of the current input. Once the cluster for the current input is selected, the corresponding *Application-Specific Run-Time Manager* (AS-RTM) solves the optimisation problem relying on the following components.

The *State* element is in charge of solving the optimisation problem by using a differential approach. The initial optimisation problem does not have any constraints (i.e. $C = \emptyset$), and the objective function minimises the value of the first software-knob. From this initial state, the application might dynamically add constraints, define a different objective function or change the application-knowledge. The solver can find the new optimal configuration efficiently, evaluating only the involved ones, by building an internal representation of the optimisation problem.

The *Runtime Information Provider* correlates an EFP of the application knowledge with an application monitor. In particular, it compares the observed behaviour with the expected one, and it computes a coefficient error defined as $e_{m_i} = \frac{expected_i}{observed_i}$, where e_{m_i} is the error coefficient for the i-th EFP. To avoid the "zero trap", the numerator and denominator are incremented by 1 when $observed_i$ is equal to zero.

2.8 Summary

In this chapter we have outlined the most recent works that can be found in literature, which this thesis' work is inspired to, describing the main differences and contributions. After that, the reader has been introduced to a background of the main modules that constitutes *Agora* and *mARGOt*. The next two chapters aim at describing their usage and implementation.

CHAPTER 3

Methodology

This chapter describes the methodology behind *Agora*. First we introduce the architectural view in the context of the whole adaptive framework, highlighting the main components and design choices. Then, we focus on the internal details of the adopted approach leveraged to build the application-knowledge at runtime during the production phase.

3.1 Framework Architecture

The approach proposed in *Agora* is based on the *mARGOt* autotuning framework, which has been outlined in Section 2.7 of the previous chapter. *mARGOt* aims at enhancing a generic target application with an adaptation layer to provide mechanisms to adapt in a proactive and reactive way to potential environment changes.

As we've seen, for a generic application, the relationship between software-knobs, EFPs of interest and input features is complex and unknown a priori. In order to solve its optimization problem, *mARGOt* needs a model of the application extra-functional behaviour: the *application-knowledge*. *mARGOt* interprets the application-knowledge as a discrete set of *Operating Points* (OPs). A generic OP relates a software-knobs configuration $\overline{K}$ with the expected metric values $\overline{M}$ (EFPs) and according to a set of input features $\overline{F}$ (if available).

$$Operating\_Point =< \overline{K}, [\overline{F},]\overline{M} >$$

Most frameworks try to produce the application-knowledge leveraging a Design Space Exploration (DSE). The main challenge of these general approaches is the exponential growth of the Design Space and given the complexity of this process, the DSE is usually done off-line prior to the application execution. Porting this phase online requires

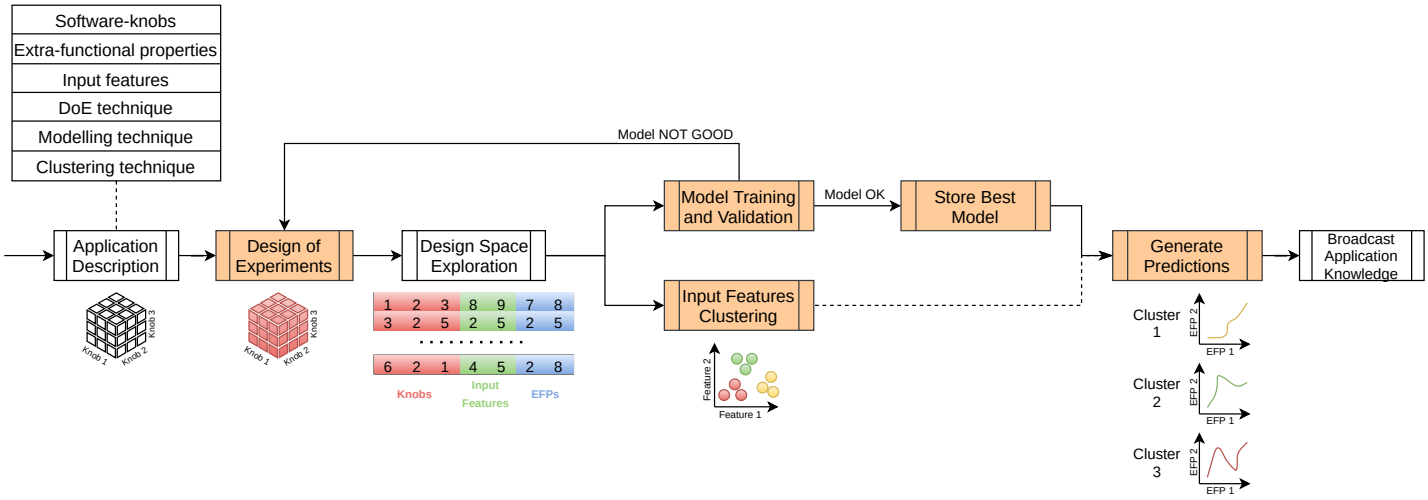


Figure 3.1: *A simplified overview of the proposed methodology. The orange elements represent the learning module components while the white ones represent external components (of the Agora and mARGOt framework).*

significant effort to minimize the tuning time and the overhead. The main goal of *Agora* is to distribute the DSE among all the instances of an unknown application at runtime with respect to their execution. This is done in order to obtain the application-knowledge of the target application.

Agora tries to generate the application-knowledge during the production phase without requiring any design-time profiling. Leveraging the underlying *mARGOt* infrastructure, the key idea is to explore the Design Space that has been considered by dynamically updating the OPs list of the target application client, assigning every time a new software-knobs configuration. Once the application-knowledge is obtained, it will be distributed to the application client starting *mARGOt*'s autotuning process.

The methodology behind *Agora*'s approach is an **iterative learning process** and Figure 3.1 gives the reader a simplified overview. The goal of this process is to learn as accurate as possible the relationship between software-knobs configurations, EFPs and input features. To achieve this *Agora* exploits an external learning module in which each component performs a different task inside the learning process. The main challenges that had to be faced can be summarized as follows:

- Since we're stealing time to the application execution, there's the need to reduce as much as possible the time required in order to generate the application-knowledge.
- As previously highlighted, *mARGOt* can be exploited to force an application to use a certain configuration during execution but on the contrary, the input set may change at every production run. The input features are parameters describing the input characteristics and we have no ability in controlling them, as opposed to the software-knobs configurations.

To tackle the first challenge, the idea is to sample the Design Space of an unknown application (which represents the set of all the possible software-knobs configuration that the application can accept) using techniques from the Design of Experiments (DoE) world, outlined previously in Section 2.3. Then, by distributing to the available clients

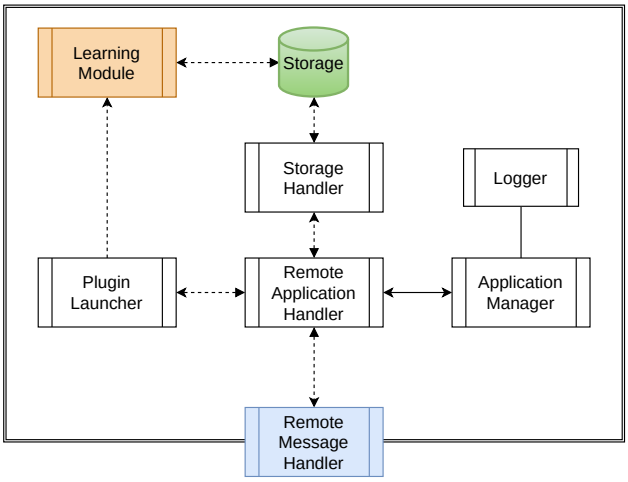


Figure 3.2: *The Agora framework overview.*

the configurations, we collect the corresponding output values which composes the so called *response surface*. In order to complete this surface we need to predict the missing points leveraging process modeling techniques to build a model for each of the EFP of interest. The solve the second issue, a solution is to cluster observed input features to find suitable representatives.

Finally, if the best model we've found for each EFP is deemed valid, we use it to predict the whole response surface and generate the application-knowledge as a list of OPs to subsequently broadcast to the application clients. A different application-knowledge is generated for each input cluster (if any). In case no valid model was created, we iterate this process until we find a suitable one for each EFP, generating additional software-knob configurations to be evaluated.

3.1.1 Components

Figure 3.2 shows the overview of the learning framework to guide the reader while introducing the main internal components. *Agora* is designed to work in a distributed context so we can look at it as a dedicated central server deployed inside a closed network, while each application instance is being run locally on the client side. The main actors are the *Learning Module*, the *Plugin Launcher*, the *Storage Handler* and the *Remote Message Handler*, which represents the communication point with the outside. Each of these components interacts with the central entities of *Agora*: the *Remote Application Handler* and the *Application Manager*. Finally, the *Logger* component is connected to the Application Manager.

The *Learning Module* is the core of the proposed approach. Its purpose is to model and predict the relationship between the EFPs, the software-knobs configurations and the input features clusters. It performs four main tasks:

1. It leverages some of the Design of Experiments techniques to sample efficiently the Design Space to be explored.

2. It uses state-of-the-art modelling techniques to fit the explored configurations. This is followed by a validation stage to test whether the quality of the obtained models

is acceptable or not.

3. It analyses input features (if any) in order to find representative clusters to be exploited inside the application-knowledge.

4. It interpolates out-of-sample predictions using the best models found.

This module has been designed as a plugin system in order to be both modular and extensible. Each plugin represents one of the tasks listed above. The implementation is pretty straightforward and the end-user who deems it necessary can extend/substitute one using its preferred language and approach by following few input/output constraints. *Agora* provides four default plugins that enforce the learning tasks. An extensive description of this learning process is provided in Section 3.2.

Agora is able to interact with each plugin thought the *Plugin Launcher* component. By managing their execution, the framework can orchestrate the whole learning process. More details on this will be discussed in the next chapter. Data informations (e.g. the application description, a list of configurations to explore, etc.) are stored inside an external storage as CSV (*Comma-Separated Values*) files. This format has been chosen for its simplicity and ease of use. During the framework design, the Apache Cassandra database [40] was also considered as a potential storage system for its great features in terms of scalability and high availability without compromising performance. At the end, we opted for something with less requirements and memory consumption although leaving a future integration still possible thanks to the flexibility of the Storage Handler. The *Storage Handler* component indeed offers an abstraction of the main functionalities in order to interact with the storage in use. This happens regardless of the chosen implementation (e.g. CSV files, database tables or even a combination of the two).

Concerning the communications to the outside, as anticipated, *Agora* exploits the MQTT protocol which key concepts have been described in Section 2.6. The *Remote Message Handler* empowers *Agora* to send/receive messages to/from external application instances in an asynchronous way. Every message encapsulate data infos that are assigned to a specific topic, depending on the learning phase they are related to.

The whole learning process is managed and coordinated by the *Remote Application Handler*. It has been designed as a thread pool for scalability reasons, managing multiple (and potentially diverse) applications. Using the Remote Message Handler, it communicates with the local application instances, sending new configurations to explore and collecting their corresponding output (which from now on will be referred to as *observations*). It coordinates the learning phase through the Plugin Launcher and interacts with the storage via the Storage Handler in order to retrieve new configurations to explore or store application description data. The glue of the whole framework is the *Application Manager*. It is a resource manager which manages each internal instance used by *Agora* and registers new applications. Finally a *Logger* component is attached to the Application Manager. The Application Manager will dispatch a separate Logger instance to every major component inside the framework in order to register their activity. The next chapter will explain in details how it works.

Since we're in a distributed context, Figure 3.3 gives a wider overview of a typical interaction between the central dedicated server running *Agora* and an heterogeneous pool of client machines, running multiple application instances. The communication is happening through the exchange of MQTT messages along a shared communication

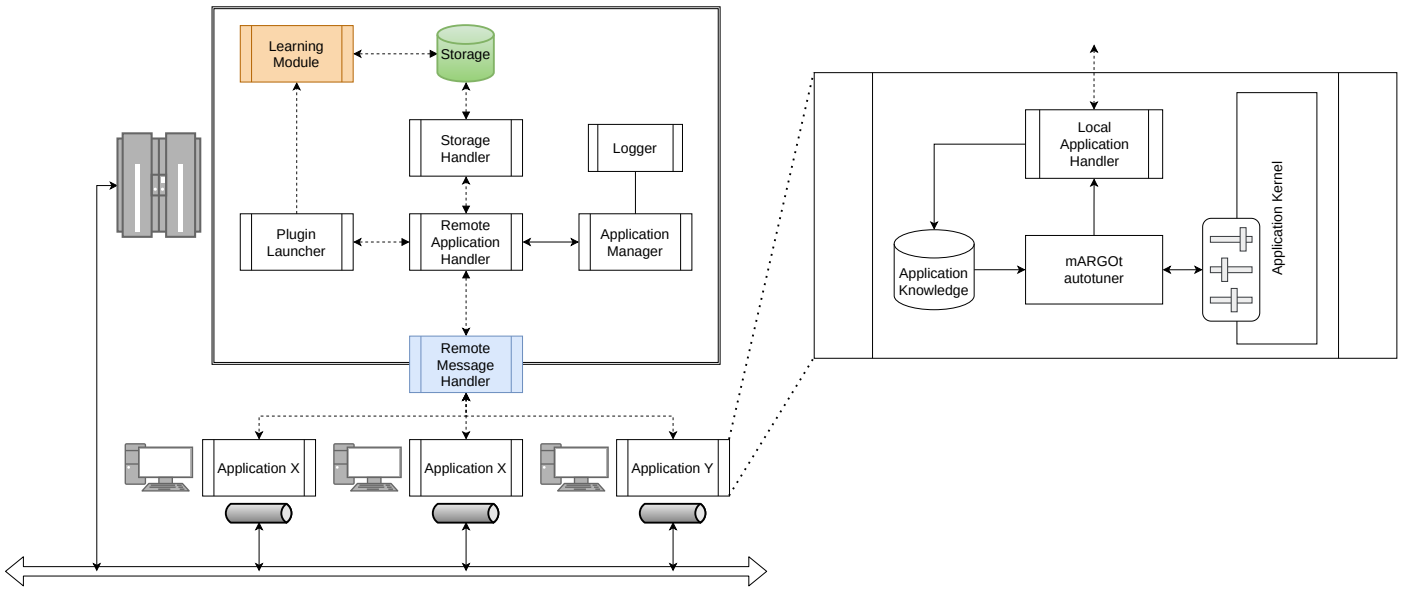


Figure 3.3: *A global overview of Agora interacting with external clients. The MQTT protocol performs node communication between application instances.*

channel set (by default) on port 1883. While the *Remote Application Handler* is running on the server side, a *Local Application Handler* is simultaneously running on the client side. This component is an asynchronous utility thread that sends out telemetry informations and manipulates the client application-knowledge. In particular, it will force the *mARGOt* autotuner to select the software-knobs configuration to be evaluated by the application kernel and it will set the final application-knowledge produced at the end of the remote learning process once received.

3.2 Iterative Learning Approach

This section describes in detail the online learning phase that generates the application-knowledge during production time. We present and formalizes the main components of this approach. Since the learning phase exploits a plugin system, the default plugins offered in *Agora* are characterized in relation with the task implemented, their constraints and the leveraged techniques. The main components are the followings: the *DoE plugin*, the *Modelling plugin*, the *Clustering plugin* and finally the *Predicting plugin*. As anticipated, the plugin creation is pretty straightforward and the end-user has total freedom over its implementation. Nevertheless, there are some constraints that needs to be taken into account:

1. Every plugin needs to parse a configuration file that is automatically set by *Agora* before each launch. This configuration file is a list of environmental variables that contains data like the application description or the storage address.

2. Data is stored as CSV files so the loading process inside each plugin should be almost agnostic with respect to the language used for the implementation. Despite this, in the future there might be added different storage implementations (e.g. a

new database system) which require additional features/wrappers to handle them. This remains up to the end-user.

3. The number and shape of the inputs and outputs is fixed, so it is mandatory for each plugin to handle this requirement. On the following subsections every plugin's description includes details about the matter.

3.2.1 Design of Experiments

In order to reduce as much as possible the Design Space Exploration, it is important to sample the Design Space accordingly and in an efficient way. The proposed DoE plugin offers two well-known techniques: a *full-factorial* design and a *latin hypercube* design. The methods have been described in Sub-Section 2.3.2.

Full-Factorial

This kind of design is pretty simple in its form and lets the user create a full grid containing all possible combinations starting from the software-knobs' domain space. For each knob (or factor) one can specify its *level* (L_i), which describes the number of discrete values that the knob can obtain. In its simplest form the design is realized by inputing a list of levels:

$$fullfact([L_1, L_2, ..., L_n])$$

where n is the total number of software-knobs for the application.

Latin-Hypercube

The Latin-Hypercube design tries to maximize the minimum distance between design points subdividing the space into an orthogonal grid with N elements of the same length per parameter. This randomized method follows a *space-filling* approach where each configuration found is spaced out evenly over the region of interest. It is realized with by following function:

$$lhs(n, samples[, criterion])$$

where n is the total number of software-knobs for the application, $samples$ designates the number of sample points to generate and $criterion$ specifies how to perform the sampling. If not specified, the algorithm simply randomizes the points within the intervals. In addiction one can select:

- *center*: centering the points within the sampling intervals;
- *maximin*: maximize the minimum distance between points, but place them in a randomized location within their interval;
- *centermaximin*: same as *maximin*, but centered within the intervals.
- *correlation*: minimize the maximum correlation coefficient between the points.

One issue behind this method is that the output design scales all the variable ranges from zero to one, sampling a *continuous* Design Space. The application description provided to *Agora* defines a *discrete* domain for each software-knob, therefore the selected samples needs to be transformed afterwards. To achieve this, the DoE plugin performs an *affine*

for $kb_i \in KNOBS$ **do**
  $max = max_{kb_i \in KNOBS}(kb_i)$;
  $min = min_{kb_i \in KNOBS}(kb_i)$;
  $round(min + DoE[:, i] * (max - min))$;
end

Algorithm 2: How the affine transformation with rounding discretizes the lhs design and maps the $[0, 1]$ values to the closest corresponding knob value.

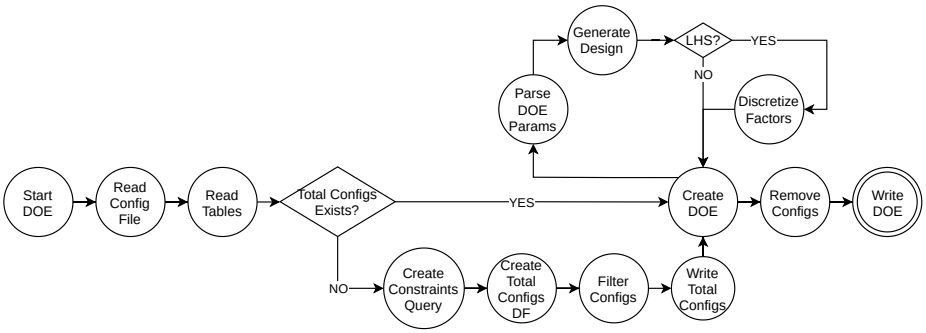


Figure 3.4: *A flowchart diagram representing the DoE plugin process.*

transformation with rounding which, after selecting the minimum and maximum value from each software-knob's domain, rounds the obtained design mapping the samples to the closest available value. Algorithm 2 shows the pseudo code of this process. Another significant problem that had to be tackled is the avoidance of repeated experiments. Since the algorithm is not able to consider old design points when producing a new one, the production of a set of experiments entirely unseen cannot be guaranteed. For this reason, on the first call the plugin generates a number of configurations always assuming the worst case scenario (i.e. reaching the maximum number of iterations) and removing duplicates before returning the output design. Subsequent calls extracts a new subset of configurations from the initial generated design. This hack let us avoid the production of identical configurations but remains of course limiting. As outlined during this thesis' conclusions 6.2, alternatives may be taken into consideration in the future.

Apart from which algorithm has been selected, on the first launch the plugin computes a full-factorial design to obtain the list of all the possible configurations to be used during the prediction phase in order to create the final application-knowledge. There is also another reason behind this. The end-user might want to set restrictions on the software-knobs domain a priori. More complicated Design Spaces could indeed require some additional constraints to express linear or even nonlinear relationships between software-knobs. To solve this, the plugin firstly applies those restrictions to the full-factorial design computed beforehand, and secondly removes all the configurations that are not inside the restricted space.

The end-user may also specify how many times each selected configuration needs to be explored. This is a very important factor because on one hand it increases the robustness of the modelling process in case of non-deterministic applications, while on the other hand, given that the input features are not controllable, might learn the knowledge from different feature sets over multiple runs of the same configuration.

Figure 3.4 shows the whole DoE generation process in a flowchart diagram.

Input & Output

For what concerns the input, the plugin starts by reading a configuration file containing the addresses of the required data-tables. At the end of the process, it is required to write the generated design and the total configurations table in output. Table 3.1 summarizes those tables along with a brief description.

Name	Type	Description
Properties Table	Input	Contains some global parameter like the number of configurations to generate or how many observations to collect for each configuration.
Knobs Table	Input	Contains the list of available software-knobs and their domain space values.
Parameters Table	Input	Contains the DoE parameters like the design algorithm to use, the criterion to be adopted or the list of constraints to be applied.
Total Configurations Table	Output	Contains all possible configurations that the target application can set on its kernel.
DoE Table	Output	Contains the list of configurations to be explored.

Table 3.1: *List of input/output tables for the DoE plugin.*

3.2.2 Modelling

This plugin can be considered in a sense the core component of the learning module. All the methods inside this plugin are expected to be part of the *Supervised* learning family. Supervised machine learning algorithms are designed to learn by example. The name "supervised" originates from the idea that training this type of algorithm is like having a teacher supervise the whole process. When training a supervised learning algorithm, the training data will consist of inputs paired with the correct outputs. During training, the algorithm will search for patterns in the data that correlate with the desired outputs. In this scenario we consider a subcategory called **regression**. Regression is a predictive statistical process where the model attempts to find the important relationship between dependent and independent variables. It out case it is used to learn the relationship between EFPs, software-knobs and input features (if any). The learning process models each EFP *independently* and therefore creates a separate model for each metric. In mathematical notation, $\hat{y}$ represents the expected value of the target EFP, while X represents the matrix of predictors (i.e. software-knobs and input features) which is extracted from the observations collected during the Design Space Exploration. In particular, each EFP is modeled as

$$\hat{y} = m(X)$$

where function m is represented by a modelling technique.

The techniques considered inside this component are intended for regression in which the target value is expected to be a linear or nonlinear combination of the predictors. For this reason, this plugin exploits a set of different models in order to fit a much wider range of data. This offers the possibility to select among these models the most accurate one every time, with much more flexibility than testing just one single model. The modelling algorithms inside the default plugin are the followings. An *Ordinary Least*

Squares, a *Ridge* regression and an *Elastic-Net*, which are methods where the target value is expected to be a linear combination of the features. In mathematical notation it means that, if $\hat{y}$ is the predicted value, then it is defined as $\hat{y}(w, x) = w_0 + w_1x_1 + ... + w_px_p$. Moreover, outside the linear models family, we can find a *Support Vector Machines* method, a *Neighbors-based* regression and finally a *Decision Trees* regression algorithm.

Ordinary Least Squares

This method fits a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w ||Xw - y||_2^2$$

The coefficient estimates for Ordinary Least Squares rely on the independence of the features. When features are correlated and the columns of the design matrix have an approximate linear dependence, the design matrix X becomes close to singular and as a result, the least-squares estimate becomes highly sensitive to random errors in the observed target, producing a large variance.

In order to mitigate some of the known issues of this method, we consider a variant of it in which a preprocessing step takes place before fitting the model. We apply a *power transformation*. Power transforms are a family of parametric, monotonic transformations that aim to map data from any distribution to as close to a Gaussian distribution as possible in order to stabilize variance and minimize skewness. In particular, we apply the *Box-Cox* transformation which is defined in Equation 3.1. The transformation is parameterized by λ, which is determined through maximum likelihood estimation.

$$x_i^{(\lambda)} = \begin{cases} \dfrac{x_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x_i) & \text{if } \lambda = 0, \end{cases} \tag{3.1}$$

Ridge regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients. The ridge coefficients minimize a penalized residual sum of squares:

$$\min_w ||Xw - y||_2^2 + \alpha||w||_2^2$$

α is called the *complexity parameter* and it's a positive value that controls the regularization strength. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Varying $alpha$ we change the amount of *shrinkage*: the larger the value of α, the greater the amount of shrinkage and thus the coefficients become more robust to collinearity.

Elastic-Net

ElasticNet is a linear regression model trained with both ℓ_1 and ℓ_2-norm regularization of the coefficients. This combination allows to learn a sparse model where few of the

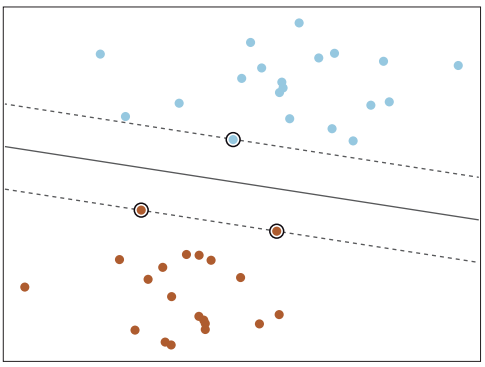


Figure 3.5: *Decision function for a linearly separable problem, with three samples on the margin boundaries, called "support vectors".*

weights are non-zero, while still maintaining the regularization properties of Ridge. Elastic-net is useful when there are multiple features which are correlated with one another.

The objective function to minimize is in this case:

$$\min_{w} \frac{1}{2n_{\text{samples}}}||Xw - y||_2^2 + \alpha\rho||w||_1 + \frac{\alpha(1-\rho)}{2}||w||_2^2$$

Support Vector Machines regression

A support vector machine (SVM) constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for regression or even other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called *functional margin*), since in general the larger the margin the lower the generalization error of the model. Figure 3.5 shows the decision function for a linearly separable problem, with three samples on the margin boundaries, called "support vectors". In general, when the problem isn't linearly separable, the support vectors are the samples *within* the margin boundaries [41].

In particular, concerning the support vector regression (SVR), the problem can be defined as follows. Given training vectors $x_i \in \mathbb{R}^p, i = 1, ..., n$ and a vector $y \in \mathbb{R}^n$, ε-SVR solves the primal problem defined in Equation 3.2.

$$\begin{aligned} \min_{w,b,\zeta,\zeta^*} \frac{1}{2}w^T w + C\sum_{i=1}^{n}(\zeta_i + \zeta_i^*) \\ \text{subject to } y_i - w^T\phi(x_i) - b \leq \varepsilon + \zeta_i, \\ w^T\phi(x_i) + b - y_i \leq \varepsilon + \zeta_i^*, \\ \zeta_i, \zeta_i^* \geq 0, i = 1, ..., n \end{aligned} \tag{3.2}$$

Here, we are penalizing samples whose prediction is at least ε away from their true target. These samples penalize the objective by ζ_i or ζ_i^*, depending on whether their predictions lie above or below the ε tube.

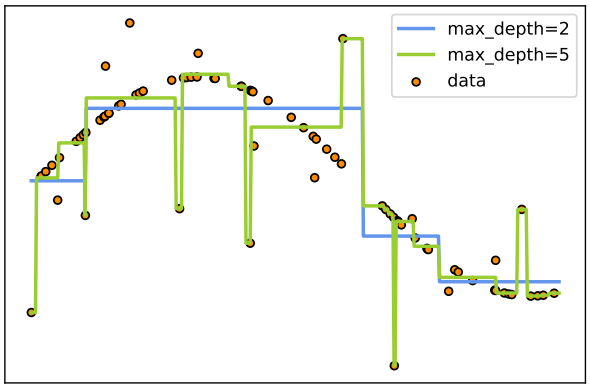


Figure 3.6: *Decision trees learning from data to approximate a sine curve. If the maximum depth of the tree is set too high, the decision trees learn too fine details of the training data and learn from the noise.*

Nearest Neighbors regression

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. In our case, we use a k-nearest neighbor algorithm with a user-defined constant set by default to 5.

In the Neighbors-based regression, the label assigned to a query point is computed based on the mean of the labels of its nearest neighbors. The basic nearest neighbors regression uses *uniform* weights: that is, each point in the local neighborhood contributes uniformly to the classification of a query point. In our case we choose a *distance* weight, assigning weights proportional to the inverse of the distance from the query point.

This modelling technique exploits different algorithms to compute the nearest neighbors, which we list without discussing further details: *Brute Force*, *Ball Tree* and *K-D Tree* algorithm. The user has no control over this choice as an internal mechanisms will attempt to decide the most appropriate algorithm based on the values passed to it.

Decision Trees regression

Decision Trees (DTs) are a non-parametric supervised learning method used for regression and more machine-learning problems. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

For instance, in Figure 3.6 we can see how decision trees learn from data to approximate a sine curve with a set of *if-then-else* decision rules. The deeper the tree, the more complex the decision rules and the fitter the model. The tree depth is controlled by the max_depth parameter.

From a mathematical point of view, given training vectors $x_i \in R^n, i = 1, ..., l$ and a label vector $y \in R^l$, a decision tree recursively partitions the feature space such that the samples with the same labels or similar target values are grouped together. Let the data

at node m be represented by Q_m with N_m samples. For each candidate split $\theta = (j, t_m)$ consisting of a feature j and a threshold t_m, it partitions the data into $Q_m^{left}(\theta)$ and $Q_m^{right}(\theta)$ subsets defined as:

$$Q_m^{left}(\theta) = \{(x, y)|x_j <= t_m\}$$
$$Q_m^{right}(\theta) = Q_m \setminus Q_m^{left}(\theta)$$

The quality of a candidate split of node m is then computed using an impurity function or loss function. In case of regression problems, the criteria used to minimize as for determining locations for future splits is the Mean Squared Error (MSE).

Model Training

Since the goal is to broadcast to the application clients a reliable application knowledge, it is crucial to evaluate the model quality and accuracy. Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called *overfitting*. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a *test set*. This split creates two partitions containing respectively 75% and 25% of the original data. This test set of unknown data will be used at the end of the process to evaluate the model and compute its accuracy. However, by partitioning the available data, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, test) sets. Given that the *Agora* approach is done online, especially at early stages we might have a small set of observations for training and validation. This plugin applies a common solution to this problem called *cross-validation* (CV for short). Using its basic approach, called *K-fold CV*, the training set is split into k smaller sets named *folds*. The following procedure is followed for each of the k "folds":

1. Each of the models is trained using $k - 1$ of the folds as training data;
2. The resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

Figure 3.7 provides a graphic visualization of the method.

The performance measure reported by K-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems where the number of samples might be very small.

The user-end can specify the number of cross-validation folds k to perform which by default is set to 5. Moreover, if n is the number of explored software-knob configurations and if the ratio $\frac{n-k}{n}$ is less than a 0.75 threshold, a *Leave One Out* (LOO) cross-validation is used instead where each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for n samples, we have different training sets and n different tests set.

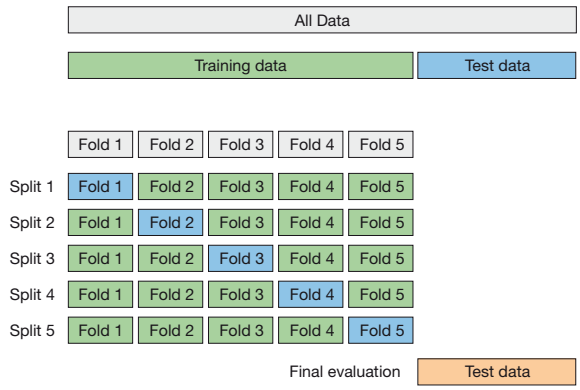


Figure 3.7: *K-fold cross-validation example where the original dataset is partitioned into a training and test set (75% - 25%). Then the training set is splitted into 5 folds. Finally the test set is used as final evaluation.*

Model Validation

To quantify the prediction quality of a model, by default two common regression metrics are considered. The *coefficient of determination* R^2 and the *mean absolute percentage error* ($MAPE$). The reason behind this choice is that in this way the end-user has the ability to set a threshold using a *percentage* value which can adapt to all kind of scenarios and doesn't require knowing the application behaviour a priori. For the same reason, we also provide a variant of the *mean absolute error* MAE which normalizes the value by the observed values range of the target metric: $|max(EFP_i) - min(EFP_i)|$. These metrics have been described in Section 2.4 so they won't be discussed again.

Once every model is evaluated, is deemed eligible if it has an R^2 score and a $MAPE$ respectively *higher* and *lower* than a threshold. This threshold is set by default to 0.8 for R^2 and to 0.1 for the $MAPE$ but the end-user has the ability to decide a custom value for each at will. Finally, the plugin chooses as final model the best accurate one among the models that have verified the quality thresholds. Alternatively, if the framework reached the maximum number of iterations the best model found is returned bypassing the threshold checks. See on Section 3.2 for more informations on this last event.

Name	Formula
Coefficient of Determination	$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y_i})^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}$
Mean Absolute Percentage Error	$MAPE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}\frac{y_i - \hat{y_i}}{max(\epsilon, \lvert y_i \rvert)}$
Mean Absolute Error	$MAE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}\lvert y_i - \hat{y_i} \rvert$
Mean Squared Error	$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y_i})^2$
Mean Squared Logarithmic Error	$MSLE(y, \hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}(log_e(1 + y_i) - log_e(1 + \hat{y_i}))^2$
Median Absolute Error	$MedAE(y, \hat{y}) = median(\lvert y_1 - \hat{y_1} \rvert, ..., \lvert y_n - \hat{y_n} \rvert)$

Table 3.2: *List of available metrics for model validation.*

Apart from the default metrics, Table 3.2 summarizes all available metrics with their corresponding definition. Figure 3.8 shows an overview of the whole modelling process.

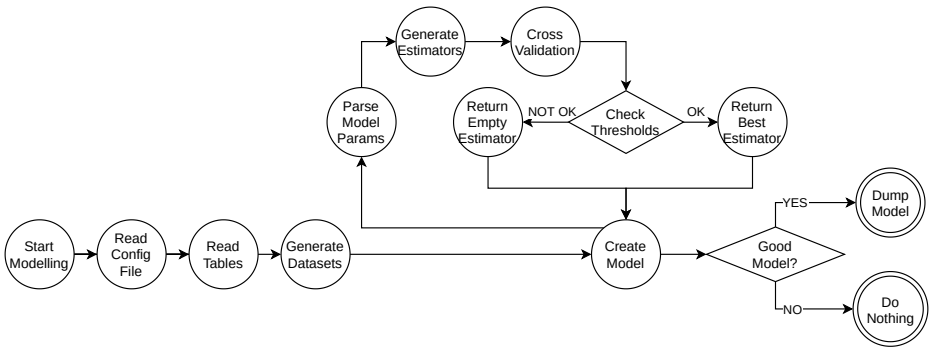


Figure 3.8: *A flowchart diagram representing the modelling plugin process.*

Input & Output

The plugin reads a configuration file containing the required data-tables. Table 3.3 summarizes those tables along with a brief description. This plugin has quite some different requirements on the output with respect to the others. *Agora* tries to persist the models into storage for future use. This gives two major advantages: first of all in case we're recovering from a previous crash the models don't need to be retrained and can easily loaded up; second of all this allows us to compute the predictions only once at the end of the learning process.

Name	Type	Description
Properties Table	Input	Contains some global parameter like the maximum number of iterations to perform or the metric name.
Knobs Table	Input	Contains the list of available software-knobs and their domain space values.
Features Table	Input	Contains a description of the input features specified by the application.
Observations Table	Input	Contains the list of the configurations explored and the corresponding output that has been produced by the target application (input features and EFPs).
Parameters Table	Input	Contains the modelling parameters like the model to use or a metric threshold value limit.

Table 3.3: *List of input tables for the modelling plugin.*

3.2.3 Clustering

The main goal of this plugin is to find representative clusters based on the input features to be exploited in the application-knowledge. The clustering plugin is a *parallel* process with respect to the modelling phase since they can operate independently. Once a suitable model have been selected for each EFP, the generated clusters are combined and used to generate the application-knowledge. This is an *optional* process, in a sense that the application may not have any features specified. Therefore, the plugin will be launched only if there are input features enabled for the application. The proposed clustering plugin offers two well-known techniques: the *K-means* algorithm and the *DBSCAN* algorithm. A detailed description of each method have been provided in Sub-Section 2.5.1.

K-means

The K-means algorithm divides a set X of N input features into K disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster *centroids* and they are not, in general, points from X, although they live in the same space. The algorithm clusters data by trying to separate the features in n groups of equal variance, minimizing a criterion known as the *inertia* or within-cluster sum-of-squares defined as:

$$\sum_{i=0}^{n} \min_{\mu_j \in C} ||x_i - \mu_j||^2$$

It requires the number of clusters to be specified and scales well to large number of features. The following function finds a set of clusters and returns a centroid for each of them:

$$kmeans(n, max\_iter, init)$$

where n is the number of clusters to form as well as the number of centroids to generate (default: 5), max_iter is the maximum number of iterations of the K-means algorithm for a single run (default: 300) and $init$ is the method of initialization to use. The following methods are available:

- *random*: choose n input features (rows) at random from data for the initial centroids.
- *k-means++* (default): selects initial cluster centers for K-means clustering in a smart way to speed up convergence. Given enough time, K-means will always converge, however this may be to a local minimum. This is highly dependent on the initialization of the centroids. The *k-means++* method initializes the centroids to be (generally) distant from each other, leading to probably better results than random initialization.

Inertia can be recognized as a measure of how internally coherent clusters are. One of the drawbacks is that inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated. Finally one thing to highlight is that the *mARGOt* autotuner implicitly expects the application-knowledge to be composed by clusters of Operating Points and selects the cluster with the features closer to the ones of the current input by using an Euclidean distance between the two vectors or a normalized one in case one vector is numerically different with respect to the other. Since K-means also focuses on minimizing a squared euclidean distance in order to generate the final cluster centroids, this is a well suited method for the clustering process and it is chosen by default.

DBSCAN

The DBSCAN algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to K-means which assumes that clusters are convex shaped. The central component to the DBSCAN is the concept of *core samples*, which are samples that are in areas of high density. A cluster is therefore a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). More formally, a core

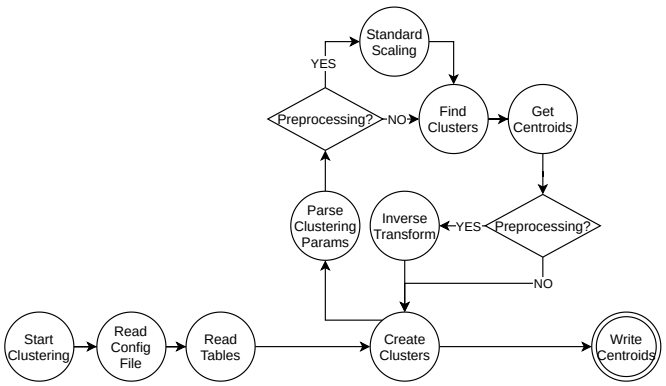


Figure 3.9: *A flowchart diagram representing the clustering plugin process.*

sample is defined as a sample in the dataset such that there exist a minimum number of other samples within a distance threshold, which are defined as *neighbors* of the core sample. This tells that the core sample is in a dense area of the vector space. The algorithm is being called with the following:

$$DBSCAN(eps, min\_samples, algorithm)$$

where eps is the maximum distance between two samples for one to be considered as in the neighborhood of the other (default: 0.5), $min\_samples$ is the number of samples (or total weight) in a neighborhood for a point to be considered as a core point (default: 5) and $algorithm$ specifies to be used to compute point-wise distances and find nearest neighbors (default: automatic selection). While the parameter $min\_samples$ primarily controls how tolerant the algorithm is towards noise (on noisy and large data sets it may be desirable to increase this parameter), the parameter eps is crucial to choose appropriately for the data set and distance function. It controls the local neighborhood of the points. When chosen too small, most data will not be clustered at all. When chosen too large, it causes close clusters to be merged into one cluster, and eventually the entire data set to be returned as a single cluster. This is why it must be chosen properly and leaving the default value may not work in some cases. As a final note, since DBSCAN doesn't provide a list of centroids as K-means does, these are found by computing the mean value for each area (cluster). For these reasons, this why in a general scenario the end-user is always recommended to select a K-means approach.

Regardless of the method used, by default the plugin carries out a scaling on the input features before clusterizing them. This preprocessing step is always advisable in order to to avoid biases imposed by the different value scales indifferent dimensions. The plugin standardize features by removing the mean and scaling to unit variance. The standard score of a feature f is calculated as:

$$z = \frac{f - u}{s}$$

where u is the mean of the input features processed and s is their standard deviation.

Figure 3.9 shows the whole clustering process in a flowchart diagram.

Input & Output

The plugin reads a configuration file containing the required data-tables. At the end of the process it writes the list of the centroid representing each cluster found. Table 3.4 summarizes those tables along with a brief description.

Name	Type	Description
Features Table	Input	Contains a description of the input features specified by the application.
Observations Table	Input	Contains the list of the configurations explored and the corresponding output that has been produced by the target application (input features and EFPs).
Parameters Table	Input	Contains the clustering parameters like the algorithm to use or if the preprocessing step has to be performed.
Centroids Table	Output	Contains a list of centroids found for each cluster.

Table 3.4: *List of input/output tables for the clustering plugin.*

3.2.4 Predicting

This last plugin is meant to be called once a model has been deemed acceptable for each EFP and, in case of input features, there are clusters available. The task performed is quite simple: it loads up all the final models that were stored and the total configurations table that was created during the DoE initial phase; at this point, the configurations table is transformed into a matrix-like dataset and used as input to the prediction method of each model. In case of feature clusters, the matrix is merged with the centroids table with a cross-product in order to have distinct predictions for each cluster representative. The generated output is the application-knowledge that will be converted into a compatible format before being distributed to the application clients. Algorithm 3 shows the pseudo code of the plugin while Figure 3.10 displays the whole process in a flowchart diagram.

$X = to\_matrix(observations\_table)$;
if $FEATURES \notin \emptyset$ **then**
  $F = to\_matrix(centroids\_table)$;
  $X = X \times F$;
end
$predictions = \emptyset$;
for $m_i \in METRICS$ **do**
  $model = load\_model(m_i)$;
  $predictions = predictions \cup model.predict(X)$;
end
return $predictions$;

Algorithm 3: How the prediction process generates the final application-knowledge.

Input & Output

As usual, the plugin firstly reads a configuration file containing the required data-tables. At the end of the process it writes a table containing all the predictions. Table 3.5 summarizes those tables along with a brief description. Please note that even if not specified, the prediction plugin is also provided with the directory path of the selected

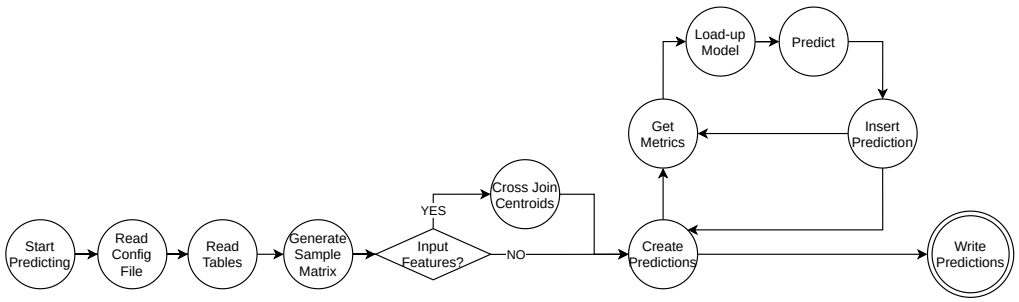


Figure 3.10: *A flowchart diagram representing the predicting plugin process.*

plugins. This is agnostic with respect to the storage handler implementation since the models will be always stored locally inside the *Agora* server machine.

Name	Type	Description
Knobs Table	Input	Contains the list of available software-knobs and their domain space values.
Features Table	Input	Contains a description of the input features specified by the application.
Metrics Table	Input	Contains a description of the EFPs specified by the application.
Observations Table	Input	Contains the list of the configurations explored and the corresponding output that has been produced by the target application (input features and EFPs).
Centroids Table [opt]	Input	Contains a list of centroids found for each cluster.
Predictions Table	Output	Contains the predictions made for each EFP.

Table 3.5: *List of input/output tables for the clustering plugin.*

3.2.5 Iterating the learning process

After illustrating each of the *learning module* components, we can put the pieces together to describe the learning process as a whole as previously outlined in Section 3.1. The key idea is that given the description of an unknown application, first of all the DoE plugin is used to sample efficiently the Design Space. After the experimental configurations have been extracted, they are sent to the available clients in order to collect the corresponding observations. Once a certain number of observations are retrieved, the modelling phase and the clustering phase are started **in parallel**. Then, if the generated models and cluster centroids are eligible for the final phase, the prediction plugin is launched to generate the list of Operating Points to broadcast to the application clients. Otherwise, the DoE plugin is started once again aiming at finding new software-knob configurations to explore in order to restart the Design Space Exploration and hence improving the models quality.

The operations described above are considered to be one iteration cycle. As the experimental results will prove, increasing the number of iterations provide better results on the final solution. The *global parameters* managing the whole cycle are three:

1. *max_number_of_iterations* (default: 100): the maximum number of iterations to perform before stopping the learning process and generating the final predictions using the best models found until that moment.

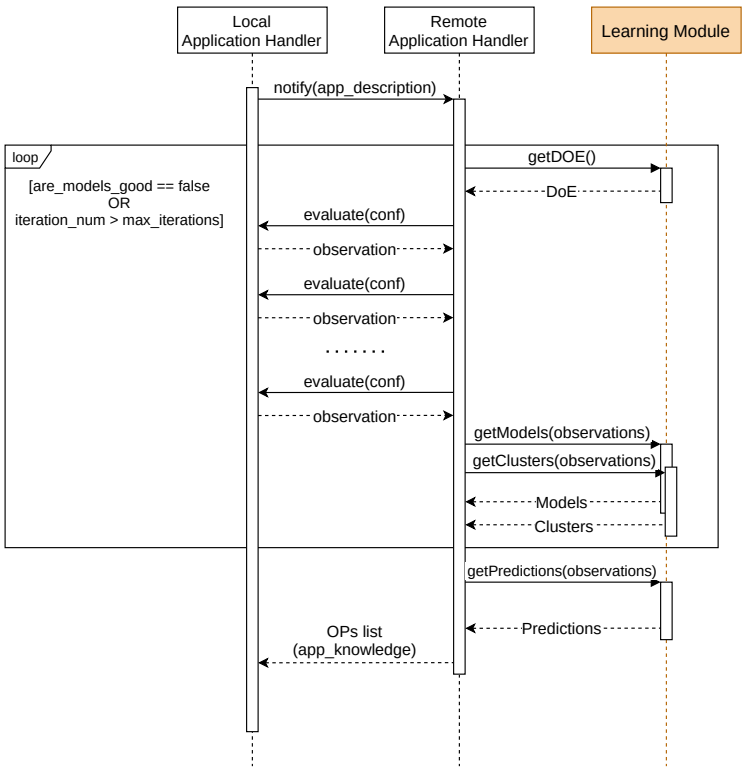


Figure 3.11: *A sequence diagram representing a typical interaction between the Remote Application Handler, the Local Application Handler and the Learning Module.*

2. *number_of_configurations_per_iteration* (default: 30): how many configurations to explore (e.g. that has to be sent to the application clients) on each iteration.

3. *number_of_observations_per_configuration* (default: 2): how many times the same configuration has to be explored.

Even if the parameters have a default value, the end-user has total freedom on them and is invited to explicitly set them at will.

To end this section, a typical workflow of the framework is reported below during the interaction with an unknown application that has just manifested itself. It is visualized as a sequence diagram in Figure 3.11.

1. The client's Local Application Handler notifies its presence to the server encapsulating the application description inside the message sent.

2. The Remote Application Handler adds the new client and stores the received informations such as the number of software-knobs and their domain, the DoE technique to use, etc.

3. Once the data have been collected, the Plugin Launcher is used to call the corresponding plugin inside the Learning Module to generate a set of configurations to explore.

4. The Remote Application Handler dispatches to the client the configurations that needs to be evaluated in a round robin fashion.

5. Once the Local Application Handler receives a configuration, it forces the autotuner to set it on the application kernel and waits for the output to be processed. Afterwards it sends the result back to the server awaiting further instructions.
6. Once the server received enough observations, it starts the modelling procedure while simultaneously generating a set of clusters in case there are input features available.
7. If the quality of the derived model is above the acceptance criteria or we reached the maximum number of iterations, the Remote Application Handler starts the final prediction phase waiting for the application-knowledge to be broadcasted. Otherwise, it restarts from Step 3 waiting for more observations to increase the quality of the results.

3.3 Summary

In this chapter we've described the methodology behind *Agora*. The framework architecture is based on the *mARGOt* autotuning framework and its main goal is to distribute the Design Space Exploration in order to obtain the application-knowledge of a target application at runtime, with respect to its execution. To achieve this, *Agora* relies on an iterative learning approach which leverages external modules that respectively:

- Apply DoE techniques in order to efficiently sample the Design Space and produce a list of configurations to explore.
- Exploit known machine learning models to describes the relationship between EFPs of interest, software-knobs and input features.
- Cluster the input features space in order to create representative centroids to be used during the application-knowledge production.

After discussing these components, we've shown a typical workflow to give the reader a detailed overview of the interactions between the main actors.

On the following chapter we will describe the framework's internal components in details and provide a deeper look into the technologies used and the most critical aspects from the implementation point of view.

CHAPTER 4

Implementation

This chapter outlines *Agora* from the implementation point of view giving a deeper look into the technologies and the most critical aspects faced during development. At first, we give an overview on how the project has been structured, describing the technologies used by each macro-component. Then, the reader is introduced to the *Agora* library, its main entities and how every module interacts with each other. Following up we describe the structure of the plugin system and the third-party libraries exploited. Then, we briefly outlines the *Agora* server binary and how its deployment works. Finally, we show a full integration example in order to outline the effort required from the end-users and application developers to integrate *Agora* in their application.

4.1 Build System

This project was born as an enhancement of an already existing framework: *mARGOt*. *Agora* can be seen as a separate module attached to it in the bigger picture. It is made out of three main components:

1. The *Agora* **binary**: a console executable that is ideally deployed on a dedicated server and which coordinates the learning process by reading and dispatching MQTT messages.

2. The *Agora* **library**: exploited by the binary, exposes the main functionalities used by the framework.

3. The **plugin system**: a separate module inside *Agora* that contains the plugins leveraged during the learning process. Chapter 3 described the methodology behind.

The binary and the library have been developed in *C++*, a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language. The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. In particular C++17 has been chosen as standard. Concerning the learning module, as stated each plugin can be integrated inside the framework in a language-agnostic fashion. Having said that, the default plugins shipped with *Agora* have been written in *Python*. Python is an interpreted, high-level and general-purpose programming language which is widely used in the machine learning field thanks to its high readability and large support from third-party libraries that allows easy and powerful implementations.

The compilation, linking and building process is managed by *CMake*. CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the chosen compiler environment. It is also very useful to automatically manage and check software dependencies during configuration time.

The main repository of the project is online and public [9]. It is organised as follow:

heel	*mARGOt* heel source files (lib + exe)
margot	*mARGOt* autotuner source files (lib)
agora	*Agora* source files (lib)
agora / server	*Agora* binary source files (exe)
agora / plugins	*Agora* plugins source files
doc	The user manuals

4.2 Agora Library

This section describes the library components in details and how they interact with each other. In the previous chapter, Section 3.1 gave a high level overview over the main actors inside the framework: the Remote Application Handler, the Application Manager, the Storage Handler, the Remote Message Handler, the Plugin Launcher and the Logger. In Figure 4.1 it is provided a more in depth view of them as a UML (*Unified Modeling Language*) class diagram showing their main interactions. This diagram shown is for explanatory purposes only, hence every class doesn't show any internal methods or members since the goal is to offer a complete view of the internal structure without going in too much details. The online repository can be inspected for further specifics.

During the design phase of the framework, we decided to apply a *factory method* pattern to the main entities. In class-based programming, the factory method pattern is a *creational* pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. Object creation is done by calling a factory method — either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes — rather than by calling directly a constructor. This philosophy has been applied to the main actors inside the framework (i.e. the Remote Message Handler, the Storage Handler, the Plugin Launcher and the Logger). A new instance is created by packing together

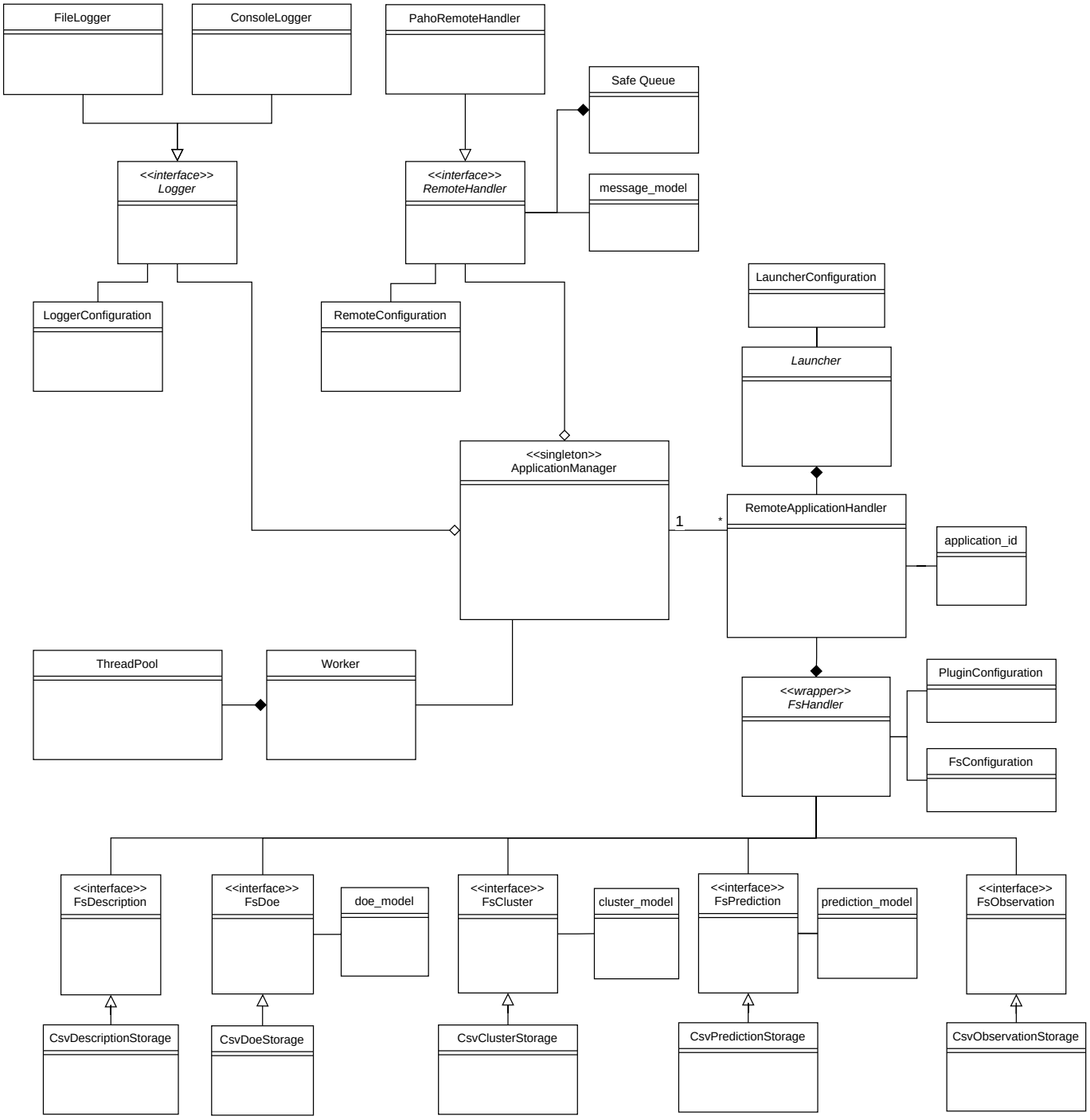


Figure 4.1: *UML class diagram showing Agora internal entities and their interactions.*

a specific *Configuration* object (e.g. LoggerConfiguration, RemoteConfiguration, etc.) which describes the implementation features and the required parameters to construct the object. This decision was made in order to acquire two major benefits:

1. Ease the Application Manager effort in managing internal instances for each application. Creating an object often requires complex processes not appropriate to

include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. This approach handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

2. Now every subclass of these entities can represent a new implementation to carry out the task, leaving future developments and expansions a minimal integration effort. New implementation details can be added to the configuration object and just by overloading the factory method in the new subclass the integration process is completed. This approach indeed relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.

The factory method is defined as $get\_instance(configuration)$. By passing the configuration parameter the factory returns a C++ *std::unique_ptr* pointing to the new object created. Without going into too much detail, a *std::unique_ptr* is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope. A unique_ptr explicitly prevents copying of its contained pointer, so it is possible to have only one owner at the same time. A unique_ptr suits C++ factory patterns well because once the pointer is returned, it is the caller's duty to decide on its ownership by maintaining the unique_ptr properties or by converting it to a *std::shared_ptr*, which instead allows multiple owners. A shared_ptr uses reference counting ownership of its contained pointer in cooperation with all copies and once that counter reaches zero, the object gets destroyed.

This section concludes by discussing the relevant classes reported in Figure 4.1.

ThreadPool and Worker

As anticipated, *Agora* is based on a thread pool design. A thread pool is a software design pattern for achieving concurrency of execution in a system. It maintains multiple threads waiting for tasks to be allocated for concurrent execution by a supervisor. By maintaining a pool of threads, the system increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks. The number of available threads is tuned to the computing resources available. If on one hand this gives us great advantages, on the other hand it requires every exposed method to be thread-safe. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. Each thread is a C++ *std::thread*. The class thread represents a single thread of execution. Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument.

Each thread perform the same task specified inside the Worker class. The task is a continuous loop that waits for new incoming messages. Depending on the message type, different actions are performed. Because of the synchronization requirements between multiple threads, incoming messages are stored inside a message queue (SafeQueue) which guarantees the thread-safety properties. Figure 4.2 exemplifies the message pulling process happening inside *Agora*.

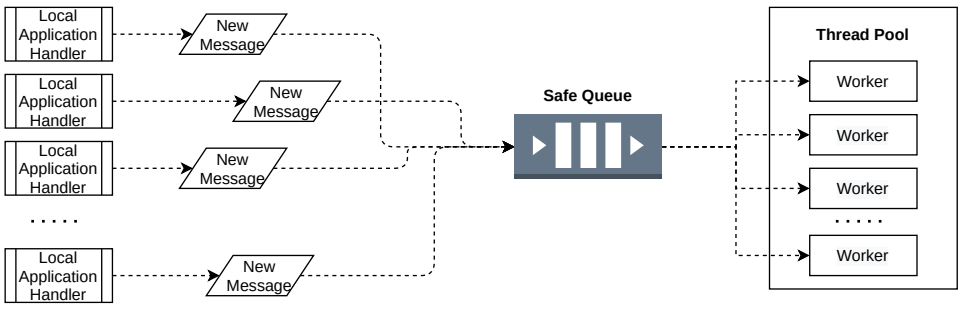


Figure 4.2: *How the thread pool handles incoming messages from the outside. Multiple Local Application Handlers send messages towards the Agora server which puts them inside a synchronized queue. Every free thread pulls out a new message in a safe way without rising race-conditions.*

FsHandler

This component represents the Storage Handler described in the previous chapter. The FsHandler is a wrapper that exposes an API with write/read methods for handling the framework storage. Calling one of those methods leverages the corresponding implementation based on the FsConfiguration specifications. A meaningful decision was to split data into different parts based on common concepts. For example, every application description data is managed by the FsDescription interface; DoE data tables (i.e. total configurations and experimental configurations) are handled by the FsDoe interface; and so on and so fort. This approach was made in order to modularize the storage handling process. For instance, if we had three different storage implementations available, we could decide to use the first one to manage application descriptions and DoE data, the second for clustering informations and the third one for the remaining data. Moreover, this means that new implementations are not obliged to specialize every API method exposed by the FsHandler but can focus only on a specific subset, based on the related interface.

The current storage handler uses CSV files to store informations. File paths and directories are managed by the C++17 *Filesystem* library. The Filesystem library provides facilities for performing operations on file systems and their components, such as paths, regular files, and directories.

Besides this, the FsHandler is used to store the plugin configuration files containing the list of required environmental variables. A PluginConfiguration object contains those informations as a list of <*key,value*> pair, where the key is the parameter name. Since the informations concern data tables location, depending on the storage implementation the FsHandler will output the corresponding address (e.g. a file path for CSVs or the table name for databases). Details about this part are outlined in Section 4.3.

RemoteHandler

This class represents the Remote Message Handler and exposes an API that manages the communication channel based on the MQTT protocol. The implementation exploits the *Eclipse Paho MQTT C Client* library [42] inside the PahoRemoteHandler class. The Eclipse Paho project provides reliable open-source implementations of open and standard messaging protocols aimed at new, existing, and emerging applications for Machine-to-Machine (M2M) and Internet of Things (IoT). Paho contains MQTT publish/subscribe

client implementations along with corresponding server support.

The client library supports two modes of operation. These are referred to as *synchronous* and *asynchronous* modes and *Agora* exploits the latter. In asynchronous mode, the client application runs on several threads. Messages are sent and subscribed to by calling functions in the client library to publish and subscribe, which also happens in asynchronous mode. Processing of handshaking and maintaining the network connection is performed in the background. Notifications of status and message reception are provided to the client application using callbacks. This API is not thread safe however: it is not possible to call it from multiple threads without synchronization and therefore, when a new message is received, the corresponding callback function puts the message inside the Safe Queue, which on the contrary is thread safe by design.

The message abstraction is defined inside the *message_model* data structure as a pair of *std::string* <topic,payload>. Before pushing a new message into the queue, *Agora* performs a *filtering* step in order to whitelist the topic and the payload from unaccepted characters.

RemoteApplicationHandler

This component manage a generic application learning process, parsing its informations, collecting observations and launching plugins. Since a RemoteApplicationHandler is created as a *shared_pointer*, multiple threads might have mutual access on it. For this reason, the *std::mutex* class is leveraged in order to synchronize them and avoid race conditions. The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

Each application is identified uniquely by an *application_id*. The ID is composed by the **application name**, the **application version** and the **block name**. This means that the same application could be composed by more than one block. In this scenario each block is handled independently. The concept of a *block* is better explained later on in Section 4.5.

The logic that dictates the workflow exploits an *Internal Status* implemented as a bitmask. Table 4.1 gives an overview on the states. Depending on which state the handler finds itself involved, different actions are performed. A transition from a state to another is performed if the action ends successfully.

The RemoteApplicationHandler is in charge of generating and broadcasting the application-knowledge once the final predictions are available. *mARGOt* interprets the application-knowledge as a list of Operating Points (OPs) in JSON (*JavaScript Object Notation*) format. Each OP has up to three fields:

1. The feature section (if any).
2. The software-knobs section.
3. The metric (EFP) section.

Each section of the OP is a key-value pair, where the key is the name of the field and the value is a number, which represents the average value. Figure 4.3 shows an example of the application-knowledge syntax, that defines an OP list for a block named "foo". In this case *mARGOt* reads it by identifying two feature clusters, one at $(f_1 = 30, f_2 = 12)$ and the other at $(f_1 = 10, f_2 = 2)$.

State	Description
Clueless	The handler has just been created and is waiting for new incoming messages.
Recovering	*Agora* has been restarted and is trying to recover data from storage (if any).
WithInformation	*Agora* has retrieved the application informations.
BuildingDoe	*Agora* is waiting the doe plugin for new configurations to explore.
WithDoe	*Agora* has retrieved new DoE configurations.
BuildingCluster	*Agora* is waiting the clustering plugin.
WithCluster	*Agora* has retrieved new cluster centroids.
BuildingModels	*Agora* is waiting the modelling plugin.
WithModels	*Agora* has stored all the models required for the prediction phase.
BuildingPredictions	The models have been deemed eligible and *Agora* is waiting for the final predictions in order to broadcast the application-knowledge.
WithPredictions	*Agora* has retrieved the final predictions.
Exploring	*Agora* is performing the design space exploration and it's waiting for new observations.
Undefined	Some undefined behaviour happened during execution and *Agora* has to abort and restart.

Table 4.1: *List of internal states that the RemoteApplicationHandler can take.*

```
{
  "foo":
  [
    {
      "features": { "feature1": 30, "feature2": 12 },
      "knobs": { "knob1": 1, "knob2": 3 },
      "metrics": { "exec_time": 20000, "error": 10 }
    },
    {
      "features": { "feature1": 30, "feature2": 12 },
      "knobs": { "knob1": 3, "knob2": 8 },
      "metrics": { "exec_time": 15000, "error": 2 }
    },
    {
      "features": { "feature1": 10, "feature2": 2 },
      "knobs": { "knob1": 2, "knob2": 1 },
      "metrics": { "exec_time": 10000, "error": 4 }
    }
  ]
}
```

Figure 4.3: *Example of an application-knowledge in JSON format for a block named "foo".*

In order to create JSON data to inject into the message payloads, the RemoteApplicationHandler exploits the *boost::property_tree* library. Boost [43] is a powerful set of C++ libraries that provides support for tasks and structures.

ApplicationManager

The ApplicationManager is used to manage *Agora*'s internal instances. It is designed following the *Singleton* pattern. The singleton pattern is a software design pattern that restricts the instantiation of a class to one *single* instance. This is useful when exactly one object is needed to coordinate actions across the system.

Being a singleton, from an implementation point of view is seen as an instance with a global scope. It is in charge of storing the list of active RemoteApplicationHandlers and

an instance of the RemoteHandler and Logger. These last two entities are dispatched as shared_pointers to new application handlers once a new application connects to *Agora*.

Logger

This component is used by *Agora* to register events happening during execution. Table 4.2 summarizes the available logging levels. Specifying the minimum logging level tells *Agora* which type of events to register and which to ignore instead.

Level	Description
Info	Logs the general flow of the application.
Warning	Logs an abnormal or unexpected event in the application flow that however do not cause the application execution to stop.
Pedantic	Logs the most detailed messages. These messages may contain sensitive application data.
Debug	Logs used for interactive investigation during development. These logs should primarily contain information useful for debugging and have no long-term value.

Table 4.2: *List of available logging levels.*

There are two Logger specializations: the *FileLogger* which saves the events to file and the *ConsoleLogger* that prints the events to standard output (i.e. terminal/console). The first type requires synchronization facilities to achieve thread safety while the ConsoleLogger exploits the C++ *std::cout* global object which controls output to a stream buffer of implementation-defined type, associated with the standard C output stream. It is safe to concurrently access std::cout from multiple threads for both formatted and unformatted output.

Launcher

This class is used by the Remote Application Handler to execute a generic plugin. It exposes the following main methods:

- *initialize_workspace*: copies the plugin source directory into a user-specified directory. This operation is performed in order to create a sandboxed environment where the plugin can run isolated. The main reasons behind this are two: on one hand this allows a developer to modify the plugin source code without worrying about altering *Agora*'s workflow, on the other hand this keeps the working directory clean with respect to potential temporary files.
- *launch*: forks the current thread to execute the main plugin script file *asynchronously* and returns the process ID.
- *wait*: by specifying the process ID as input, this static method can be leveraged by the caller to wait the specified process to end its computation. It adds a *synchronous* feature to the plugin calls.

4.3 Plugin System

This section provides the implementation details for each of the default plugins shipped along *Agora*. A generic plugin is called by the RemoteApplicationHandler by exploiting

```
#!/bin/bash
ENVIRONMENTAL_FILE=$1
source $ENVIRONMENTAL_FILE

################################################################################
# THIS IS THE PLUGIN ENTRY POINT
#-------------------------------------------------------------------------------
#
# The environmental file provides to this script the following variables:
#   - APPLICATION_NAME -> the name of the application
#   - BLOCK_NAME -> the name of the block of code managed
#   - VERSION -> the version number of the application
#   ................. more variables .................
#   - WORKING_DIRECTORY -> the plugin working directory.
#   - CONFIG_FILE_PATH -> the plugin environmental configuration file path.
#
# Is up to the plugin writer to use this script to call the tools that perform
# the plugin task.
################################################################################

# exit if fail
set -e

python3 $WORKING_DIRECTORY/main.py $CONFIG_FILE_PATH
```

Figure 4.4: *Example of a generic plugin entry point script.*

a *bash script* as entry point. The script sources a configuration file containing a list of environmental variables that needs to be loaded by the plugin main script before being launched. In Figure 4.4 an example is provided showing the syntax. The RemoteApplicationHandler checks the return value of the script to make sure that everything went fine. Once the script completes, it is assumed that the plugin has finished.

Every plugin exploits the *DotEnv* package [44] to read key-value pairs from *.env* files. As stated during the introduction to this thesis' work, the goal of the project was developing a framework leveraging existing well-known technologies, not creating new ones. Therefore each plugin methods are taken from the following sources.

DoE Plugin. DoE methods implementations are taken from the PyDoE package [45] which is designed to construct appropriate experimental designs.

Modelling Plugin. The models are taken from the Scikit-Learn package [46] which is an open-source project designed to offer simple and efficient tools for predictive data analysis. The model persistence is achieved through the *JobLib* open-source package [47].

Clustering Plugin. As for the modelling techniques, the clustering algorithms are based on the Scikit-Learn package.

Some other worth-to-mention packages are *Pandas* [48] and *NumPy* [49]. Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, which is leveraged as a read/write mechanism for CSV tables. NumPy is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Finally, since software-knobs can also be a *string* type, the DoE and Modelling

plugins needed a mechanism to map each string to a numeric value which their methods could interpret correctly. The solution was to exploit the *LabelEncoder* utility class from Scikit-Learn that helps normalize labels such that they contain only values between 0 and $n\_classes - 1$. For instance, if we had a software-knob $alghorithm$ defined as $[\text{"}number\_1\text{"}, \text{"}number\_2\text{"}, \text{"}number\_3\text{"}]$, the LabelEncoder would map this list to $[0, 1, 2]$ before continuing the execution flow.

4.4 Agora Binary

This final section briefly describes the *Agora* executable. Once the binary is created by the CMake script, ideally it should be deployed within the server machine. The executable is named `agora` and expose several flags and options to configure its behaviour. Using the *–help* command option provides the end-user with the full list of available options. In particular, it is possible to configure parameters for the MQTT connection or parameters to setup the storage configuration. The server requires only three parameters to successfully start its execution:

- *workspace-directory*: where the plugins store logs and temporary files.
- *plugin-directory*: the directory containing the plugins implementations.
- *models-directory*: the location where the modelling plugin will store the models once deemed eligible.

The program options are parsed using the *boost::program_options* library by Boost. This library allows programs to obtain program options, that is (name, value) pairs from the user, via conventional methods such as command line and configuration file. It has been chosen in place of a straightforward hand-written code because it's easier syntax, error reporting is better and options can be read from anywhere. So if future developments wanted configuration files or maybe even environment variables, these could be added without any significant effort.

4.5 Integrating in a target application

This section describes the effort required from end-users and application developers to integrate *Agora* in their application. In this scenario the *end-users* are seen as the final utilizers of the application and therefore they are in charge of defining the application requirements (e.g. which are the EFPs of interest and how the objective function is defined) and identifying input features (if any). Application *developers* instead are the ones that write the application source code; therefore they are in charge of identifying software-knobs and extracting features from the input (if any). To ease the integration process in the target application, *Agora* is packaged with a utility tool called *mARGOt Heel* that starting from a JSON description of the extra-functional concerns, generates a high-level-interface tailored for the target application. The main configuration file describes the adaptation layer by stating:

1. the monitors of interest for the application;
2. the geometry of the problem (i.e. the EFPs of interest, the software-knobs and features of the input);

3. the application requirements (i.e. the optimization problem and the *Agora* required informations to perform the online design space exploration).

Starting from this high-level description, *mARGOt Heel* generates a library with the required glue code that aims at hiding, as much as possible, the implementation details.

The idea is to model an application as composed of several independent kernels, named *blocks*. The following is a simple application model example that for simplicity has a single block named *foo* but it is trivial to generalize the example with more than one block.

```
main
{
    loop
    {
        output = foo(IN input, IN knobs)
    }
}
```

For each block *mARGOt Heel* generates the following functions meant to wrap the block of code:

- **init** : the global function that initializes the data structures.
- **update** : updates the application software-knobs with the configuration provided. This is used to force the kernel to explore a given configuration.
- **start_monitors** : starts the measurement of all the monitors that require a starting point (e.g. Time monitor).
- **stop_monitors** : ends the measurement of all the monitors that require a stopping point (e.g. Time monitor).
- **push_custom_monitor_values** : inserts a new value in a custom monitor (e.g. Quality monitor) and sends a new observation to the *Agora* server corresponding to the last software-knobs configuration set.
- **log** : prints runtime information on file and/or standard output.

These functions hide the initialization of the framework and its basic usage. For example the update function takes as output parameters the software-knobs of the application and as input parameters the features of the current input. In the event that we had already received the application-knowledge, it would use the features to select the most suitable cluster and then it would set software-knobs parameters according to the most suitable configuration found. Considering the previous example, the following is the logical integration that happens in the target application considering all the exposed functions and their order.

```
main
{
    init
    loop
    {
        update(IN input, OUT knobs)
        start_monitors
        output = foo(IN input, IN knobs)
        stop_monitors
        push_custom_minitor_values
        log
    }
}
```

To show the integration effort, in the following example we focus on a toy application with two software-knobs (*knob1* and *knob2*) and two input features (*feature1* and *feature2*). The application algorithm is rather simple: it is composed of a loop that continuously elaborates new inputs. In this example, we suppose that the end-user is concerned about execution time and the computation error. In particular, he/she would like to minimize the computation error, provided an upper bound on the execution time.

In the context of this example, Figure 4.5 shows the main JSON configuration file that states the extra-functional concerns. This file is composed of four sections: the monitor section (lines 7 - 17) the application geometry section (lines 18 - 41) the *Agora* section (lines 42 - 53) and the adaptation section (lines 54 - 65). For a detailed description of the JSON syntax and semantics, please refer to the *mARGOt Heel* user manual in the *Agora* repository [9].

The **monitor section** lists all the monitors of interest for the user. In this example, there is an execution time monitor (lines 9 - 12) and a custom monitor for observing the error (lines 13 - 16). All the monitors might expose to application developers a statistical property over the observations, such as the average value in this example.

The **application geometry** section lists the application software-knobs (lines 18 - 22), the input features (lines 24 - 28) and the metrics of interest (lines 29 - 41). For each software-knob it is required to specify its domain values (as a range or by expliciting them one by one). Input features don't have a domain but we must specify a type and how to compute the distance between feature vectors (line 23) and constraints on their selection. For instance, considering *feature2*, it is stated that a cluster is eligible to be selected only if its *feature2* value is lower or equal than the *feature2* value of the current input. No requirements are imposed for *feature1* instead. Finally for every metric of interest it is specified their type, the monitor which observes their behaviour and which plugin will be used to find a suitable model. In this case the *exec_time* metric will be predicted by "model_plugin_1" with an R^2 threshold of 0.8 (lines 32 - 34). On the other hand the *error* metric will be predicted by "model_plugin_2" using its default parameters.

The ***Agora*** section contains the MQTT informations needed to connect to the *Agora* server by specifying the broker url in the form *<protocol>://<url>:<port>*. One can specify the broker username and password to log in, the QoS level and the paths to the broker certificate, the client certificate and the client key leaving them empty if they're not required (lines 44 - 46). It is also required to specify the clustering plugin and the DoE plugin to use. In this case it is specified which clustering method to leverage and how many centroids to generate plus a constraint on *knob1* and *knob2*.

```
1  {
2    "name": "toy_app", "version": "1.0",
3    "blocks":
4    [
5      {
6        "name": "foo",
7        "monitors":
8        [
9          {
10           "name": "exec_time_monitor", "type": "time", "log": [ "average" ],
11           "constructor": [ "margot::TimeUnit::MILLISECONDS", 1 ]
12         },
13         {
14           "name": "error_monitor", "type": "float", "log": [ "average" ],
15           "stop": [ {"error":"float"} ]
16         }
17       ],
18       "knobs":
19       [
20         { "name": "knob1", "type": "int", "range": [ 1,32,1 ] },
21         { "name": "knob2", "type": "float", "values": [ 1,2,3,4,5 ] }
22       ],
23       "feature_distance":"euclidean",
24       "features":
25       [
26         { "name": "feature1", "type": "double", "comparison": "-" },
27         { "name": "feature2", "type": "double", "comparison": "le" }
28       ],
29       "metrics":
30       [
31         {
32           "name": "exec_time", "type": "float", "prediction_plugin": "model_plugin_1",
33           "prediction_parameters":
34           [ {"quality_threshold": "{'r2':0.8}"} ],
35           "observed_by": "exec_time_monitor"
36         },
37         {
38           "name": "error", "type": "float", "prediction_plugin": "model_plugin_2",
39           "observed_by": "error_monitor"
40         }
41       ],
42       "agora":
43       {
44         "borker_url": "127.0.0.1:1883",
45         "broker_username": "margot", "broker_password": "margot_psw", "broker_qos": 2,
46         "broker_ca": "", "client_cert": "", "client_key": "",
47         "clustering_plugin": "cluster_plugin",
48         "clustering_parameters":
49         [ {"algorithm": "kmeans"}, {"number_centroids": 5} ],
50         "doe_plugin": "doe_plugin",
51         "doe_parameters":
52         [ {"constraint": "knob1 + knob2 < 40"} ]
53       },
54       "extra-functional_requirements":
55       [
56         {
57           "name":"my_optimization_problem",
58           "minimize":
59           {
60             "linear_mean": [ {"error": 1} ]
61           },
62           "subject_to":
63           [ { "subject":"exec_time", "comparison": "le", "value": 1000.0 } ]
64         }
65       ]
66     }
67   ]
68 }
```

Figure 4.5: *The main JSON configuration file for the toy application, stating extra-functional concerns.*

Finally, the **adaptation section** states the application requirements of the end-user. In particular, it states the application goals (lines 62,63) and the constrained multi-optimization problem (lines 57 - 61). It is declared that the objective is to minimize the

```
1  #include <margot/margot.hpp>
2
3  int main()
4  {
5    margot::init();
6
7    int knob1 = 4;
8    int knob2 = 2;
9    float error = 0.0f;
10
11   while (work_to_do())
12   {
13     new_input = get_input();
14     const double feature1 = extract_feature1(new_input);
15     const double feature2 = extract_feature2(new_input);
16
17     // foo block
18     {
19       margot::foo::update(feature1, feature3, knob1, knob2);
20       margot::foo::start_monitors();
21
22       do_job(new_input, knob1, knob2);
23       error = compute_error(new_input);
24
25       margot::foo::stop_monitors();
26       margot::foo::push_custom_monitor_values(error);
27     }
28
29     margot::foo::log();
30   }
31 }
```

Figure 4.6: *Stripped C++ code of the target toy application, after the mARGOt Heel integration.*

error while keeping the execution time below 1000 milliseconds.

Starting from this configuration file, *mARGOt Heel* automatically generates the glue code accordingly, exposing to application developers a high-level interface tailored for the specific problem. Figure 4.6 shows the source code of the application after the integration. To highlight the required effort, the application algorithm is hided in three functions: *work_to_do* (line 11) tests whether input data are available, *get_input* (line 13) retrieves the last input to elaborate and *do_job* (line 22) performs the elaboration. The integration effort requires application developers to include the *mARGOt* header (line 1), to initialize the framework (line 5) and to wrap the block of code managed by *mARGOt* (lines 18 - 27). Note that even if we minimized the framework integration effort, it is still needed that application developers identify and write the code that extracts meaningful features from an input (lines 14,15); and a function that computes the elaboration error (line 23).

4.6 Summary

In this chapter we presented an overview on how *Agora* have been structured internally. We gave a description of three of the main components that constitutes the proposed framework at its core.

- The *Agora* binary: a console executable that is ideally deployed on a dedicated server and which coordinates the learning process by reading and dispatching MQTT messages;
- The *Agora* library: exploited by the binary, it exposes the main functionalities used to achieve the goal;
- The plugin system: a separate module inside *Agora* that contains the plugins leveraged during the learning process.

At the end, we've shown the effort required from end-users and application developers to integrate *Agora* in their application through a toy example.

In the next chapter, we will focus on the experimental evaluations that have been carried out to address the benefits and limitations of the proposed framework.

CHAPTER 5

Experimental Evaluation

This chapter aims at providing an experimental evaluation of the proposed framework, highlighting its benefits and limitations. At first, we introduce the platform in which *Agora* was deployed and the experiments took place. Next, we present an evaluation of the framework's scalability and overheads registered during a stress test benchmark. Then, we outlines the target applications that were integrated with *Agora*. They are taken from a benchmark suite composed of multithreaded programs. The suite focuses on emerging workloads and was designed to contain a diverse selection of applications that is representative of next-generation shared-memory programs for chip-multiprocessors. Finally, we show how the runtime learning approach might be beneficial for each the target applications.

5.1 Target Architecture

The target architecture in which *Agora* has been deployed is a cluster of virtual machines to experiment features or run long background jobs, powered by *PoliCloud*. It is composed by a login node and the Compute ARchitecture for Ligand Optimization (*carlo*) partition. Figure 5.1 gives an overview over the cluster architecture, highlighting the connections between different components. In particular, both the login node and the computation nodes, share the file system through NFS (*Network File System*). NFS is a distributed file system protocol allowing a user on a client computer to access files over a computer network much like local storage is accessed. In this way all the active nodes can access the same files as soon as they are available.

The login node and the *carlo* partition are based on *Ubuntu 20.04 (Focal Fossa)*. To provide different versions of compilers and libraries we use *Environmental Modules* [50]. This tool simplifies shell initialization and lets users easily modify their environment

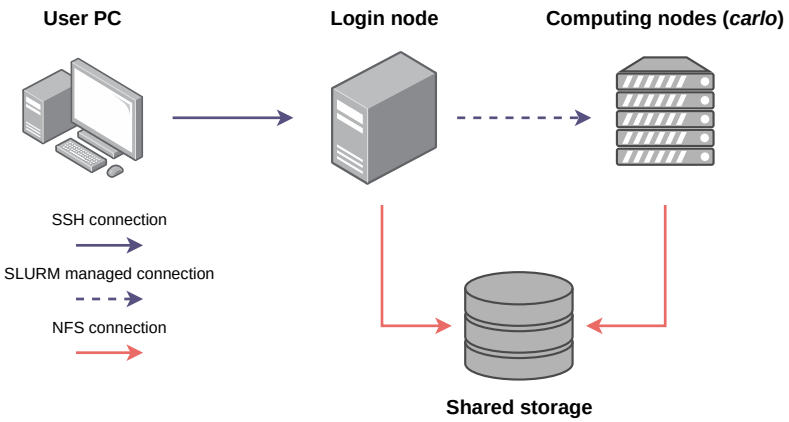


Figure 5.1: *Overview of the cluster architecture.*

during a session using *module-files*. Each module-file contains the information needed to configure the shell for an application. Once the Modules package is initialized, the environment can be modified on a per-module basis using the module command which interprets module-files. For instance, loading gcc, cmake and boost by specifying their version can be issued with the following command:

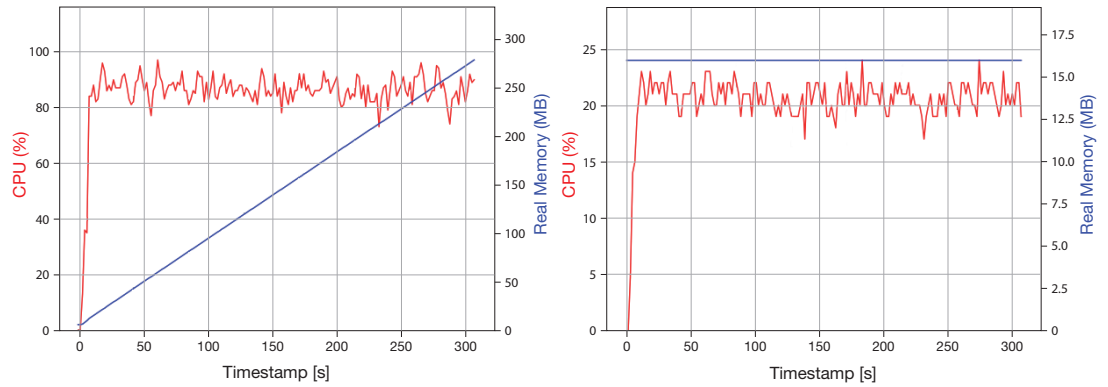
module load gcc/9.3.0 cmake/3.16.5 boost/1.72.0_gcc_9.3.0

The login node has a virtual processor @2.2Ghz with 32 GB of memory and 16 cores, with 1 thread per core. A user is allowed to submit jobs to a computing node or to directly use the login node as computing machine if not interested on distributed computation. In our scenario, the login node is exploited as the central server node in which the *Agora* binary gets executed.

The *carlo* partition includes four computation node. Each computation node has a virtual processor @2-2Ghz with 32 GB of memory and 8 cores, with 1 thread per core. A user is not allowed to login to any computation node, but should submit a job using the SLURM (Simple Linux Utility for Resource Management) manager [51], a free and open-source job scheduler for Linux and Unix-like kernels. During experiments, each computation node is used as a pool of clients running multiple instances of applications and contributing to the application-knowledge production in a distributed way.

5.2 Evaluating the Framework Scalability and Throughput

Agora's communication happens between the Remote Application Handler and the Local Application Handler which are simultaneously running respectively on the server side and on the client side. We tested the communication channel in order to get a numeric evaluation about the CPU and memory consumption, as well as the maximum throughput that the server can handle in terms of messages per second. One important thing to stress is the fact that the bandwidth on the network and the NFS storage play a large role in performance and different runs may behave differently with respect to others. That's why each experiment has been performed 10 times and the following values refer to the total average. The same experiment could have different outputs in a system with a different setup but since the goal is not a comparison between different



(a) *Agora performance.*

(b) *Mosquitto performance.*

Figure 5.2: *CPU and Real Memory usage of Agora (a) and Mosquitto (b) during five minutes of message production.*

platforms, this was not investigated during this thesis.

The experiment in this section relies on a benchmark application that stresses the MQTT channel sending $\approx 8000\ msg/sec$ towards the server. This was a reasonable value to saturate the channel and reach its limits. The application has been deployed inside the *carlo* cluster and ran simultaneously on each of the four computing nodes available, leading to an overall throughput of $\approx 32000\ msg/sec$ in output. The MQTT communication channel *QOS level* is set to 0 in order to simulate an HPC environment and it is managed by the *Eclipse Mosquitto* [52] message broker. Mosquitto is an open source message broker that implements the MQTT protocol versions 5.0, 3.1.1 and 3.1. It is lightweight and is suitable for uses on all devices, from low power single board computers to full servers.

The measurements start at the reception of the first observation. From this point onwards *Agora* enters the "hot phase", which represents a loop over the following operations:

1. Reception of an incoming observation.
2. Selection of a new eligible configuration to explore.
3. Dispatch of the configuration to a new active client.

The application has been designed with fake configuration points and fake observation values in order to simulate the behaviour of a generic application during the DSE phase. The message production lasted 5 minutes, which appeared to be long enough to reach a steady state.

During the tests, both the *Agora* server and the Mosquitto processes have been monitored, recording their **CPU** and **Real Memory** usage. Figure 5.2 shows the recorded values. After an initial setup phase, both processes quickly reach a stable point. Since the aim was to find *Agora* limits, the incoming message throughput is set high enough to cap *Agora* CPU percentage to 100% as shown in Figure 5.2a. Figure 5.2b shows instead that Mosquitto stabilizes around 20-23% during the benchmark. Data has been collected exploiting *PsRecord* [53], a small tool utility that uses the `psutil` Python library to record the CPU and memory activity of a process.

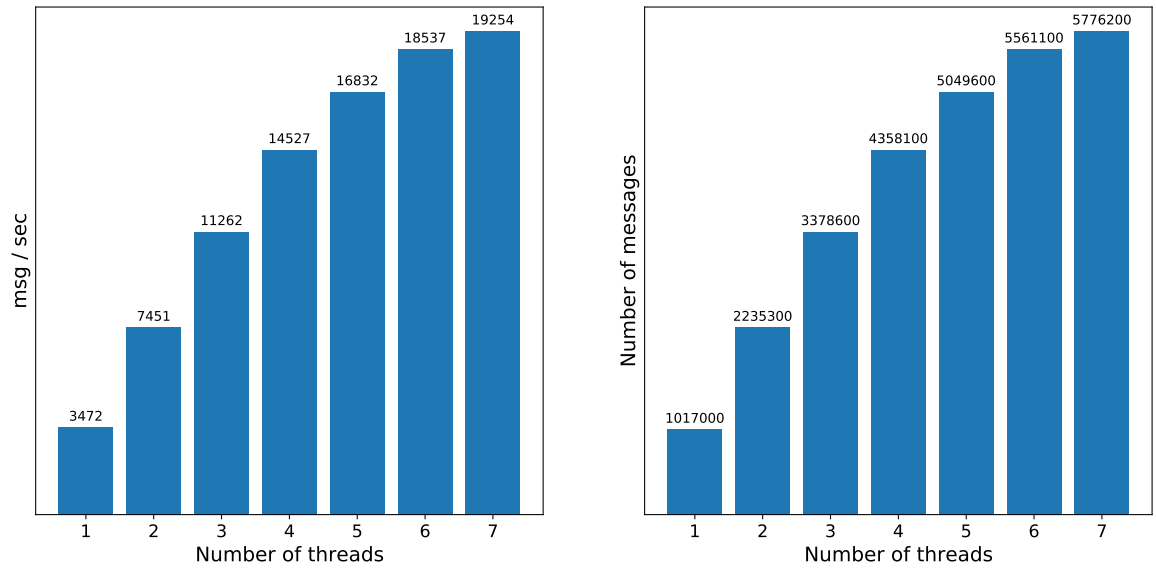


Figure 5.3: *Throughput and number of messages received by varying the number of workers inside the thread pool (5 minutes time frame).*

During the benchmark, we counted the number of messages M processed and the corresponding throughput of each recording was calculated as follows:

$$throughput_i = \frac{M}{(t_i - t_0)}$$

where t_i is the time of recording and t_0 is the starting time (in *seconds*).

In order to show the scalability benefits of *Agora*, Figure 5.3 reports how the throughput increases by varying the number of workers exploited by the server thread pool. Each column correspond to the maximum value registered after reaching a stable phase expressed as *number of messages per second*. The *Agora* disposal throughput is equal to the input throughput. Increasing the thread-pool size the throughput increases as well. The speedup is noticeable and starts decreasing from 4-5 threads which is probably due to a higher number of lock primitives calls. Figure 5.4 shows the trend of the process, highlighting how the stable phase is reached rather quickly while the number of messages received increase linearly in time. We've shown only one plot since we have the same trend regardless of the number of threads. The actual number of incoming messages that *Agora* is able to manage, despite being more than sufficient during the experiments, could be limiting with micro-kernels in systems with high parallelism. Although that matter has not been further investigated, the main reason we can associate this issue with is the I/O management system. CSV files despite being simple and easy, require a thread-safe implementation and hence a high number of lock/unlock primitives.

As a final thought, on each machine the *same* application was executed. This means that with a more heterogeneous pool of applications, *Agora* is capable of releasing the pressure on internal locks primitives and experience a speedup which could grow linearly with respect to the number of different applications.

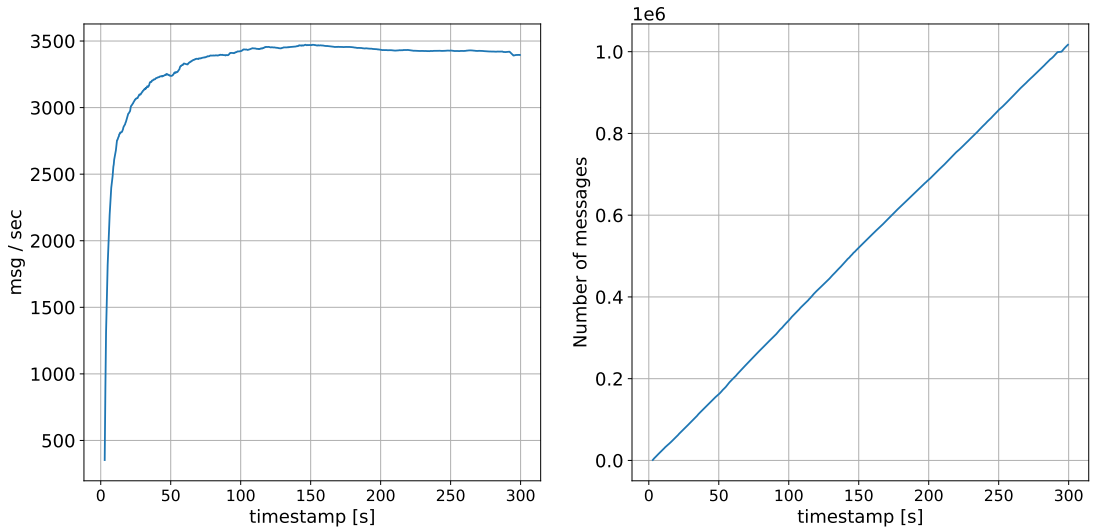


Figure 5.4: *The throughput and number of messages trend during reception.*

5.3 Target Applications

This section gives the reader an overview over the benchmarking application that have been integrated with *Agora* in order to test the learning process approach. They are taken from the *Princeton Application Repository for Shared-Memory Computers* (PARSEC) benchmark suite [54]. As stated on the documentation page, PARSEC differs from other benchmark suites in the following ways:

- *Multithreaded*: PARSEC is one of few benchmark suites that are parallel.
- *Emerging Workloads*: the suite includes emerging workloads which are likely to become important applications in the near future but which are currently not commonly used.
- *Diverse*: The selection of included programs is wide and tries to be as representative as possible.
- *Employ State-of-Art Techniques*: The PARSEC suite not only represent emerging applications but also use state-of-art techniques.
- *Research*: the suite is primarily intended for research. It can also be used for performance measurements of real machines, but its original purpose is insight, not numbers.

The following programs have been chosen to be evaluated. The same objective function is defined for each of them and that is the *minimization of their execution time*. Their description is primarily based on this paper [55]. A small summary that lists the software-knobs exploited during DSE and their input characteristics is provided at the end of each subsection.

5.3.1 Freqmine (Data Mining)

The `freqmine` application employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI). It is an Intel RMS benchmark which was originally developed by Concordia University. FIMI is the basis of Association Rule Mining (ARM), a very common data mining problem which is relevant for areas such as protein sequences, market data or log analysis. `freqmine` was included in the PARSEC benchmark suite because of the increasing demand for data mining techniques which is driven by the rapid growth of the volume of stored information.

FP-growth stores all relevant frequency information of the transaction database in a compact data structure called FP-tree (Frequent Pattern-tree). An FP-tree is composed of three parts. First, a prefix tree encodes the transaction data such that each branch represents a frequent itemset. The second component of the FP-tree is a header table which stores the number of occurrences of each item in decreasing order of frequency. The third component is a lookup table which stores the frequencies of all 2-itemsets. A row in the lookup table gives all occurrences of items in itemsets which end with the associated item. In order to mine the data for frequent itemsets, the FP-growth method traverses the FP-tree data structure and recursively constructs new FP-trees until the complete set of frequent itemsets is generate.

`freqmine` has been parallelized with *OpenMP*. It employs three parallel kernels:

- **Build FP-tree header**: scans the transaction database and counts the number of occurrences of each item. This kernel has *one* parallelized loop.
- **Construct prefix tree**: builds the initial tree structure of the FP-tree. It performs the second and final scan of the transaction database necessary to build the data structures which will be used for the actual mining operation. The kernel has *four* parallelized loops.
- **Mine data**: uses the data structures previously computed and mines them to recursively obtain the frequent itemset information. *Two* parallelized loops are employed.

The software-knobs and the input size are listed in the following table:

Name	Type	Values
`num_threads`	int	[1 - 8]
`threshold`	int	[100, 1000, 5000, 11000]
Input Size	990,000 transactions	

5.3.2 Blackscholes (Financial Analysis)

The `blackscholes` application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE)

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is an option on the underlying S with volatility σ at time t if the constant interest rate is r. There is no closed-form expression for the Black-Scholes equation and as such it must be computed numerically. The `blackscholes` benchmark was chosen to represent the wide field of analytic PDE solvers in general and their application in computational finance in particular. The program is limited by the amount of floating-point calculations a processor can perform.

`blackscholes` stores the portfolio with *numOptions* derivatives in array *OptionData*. The program includes *fileoption-Data.txt* which provides the initialization and control reference values for 1,000 options which are stored in array *datainit*. The initialization data is replicated if necessary to obtain enough derivatives for the benchmark.

The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and calls function *BlkSchlsEqEuroNoDiv* for each of them to compute its price.

The software-knob and the input size are listed in the following table:

Name	Type	Values
`num_threads`	int	[1 - 8]
Input Size	65,536 options	

5.3.3 Swaptions (Financial Analysis)

The `swaptions` application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a no-arbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. `swaptions` therefore employs MonteCarlo (MC) simulation to compute the prices. The workload was included in the benchmark suite because of the significance of PDEs and the wide use of Monte Carlo simulation.

The program stores the portfolio in the *swaptions* array. Each entry corresponds to one derivative. `swaptions` partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. Each thread iterates through all swaptions in the work unit it was assigned and calls the function *HJMSwaptionBlocking* for every entry in order to compute the price. This function invokes *HJMSimPathForwardBlocking* to generate a random HJM path for each MC run. Based on the generated path the value of the swaption is computed.

The software-knobs and the input size are listed in the following table:

Name	Type	Values
`num_threads`	int	[1 - 8]
`trial_multiplier`	int	[1 - 7]
Input Size	64 swaptions, 20,000 simulations	

5.3.4 Bodytrack (Computer Vision)

The `bodytrack` computer vision application is an Intel RMS workload which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence. `bodytrack` employs an annealed particle filter to track the pose using edges and the foreground silhouette as image features, based on a 10 segment 3D kinematic tree body model. These two image features were chosen because they exhibit a high degree of invariance under a wide range of conditions and because they are easy to extract. An annealed particle filter was employed in order to be able to search high dimensional configuration spaces without having to rely on any assumptions of the tracked body such as the existence of markers or constrained movements. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

For every frame set Z_t of the input videos at time step t, the `bodytrack` benchmark executes the following steps:

1. The image features of observation Z_t are extracted. The features will be used to compute the likelihood of a given pose in the annealed particle filter.
2. Every time step t the filter makes an annealing run through all M annealing layers, starting with layer $m = M$.
3. Each layer m uses a set of N unweighted particles which are the result of the previous filter update step to begin with.

   $$S_{t,m} = \{(s_{t,m}^{(1)})...(s_{t,m}^{(N)})\}$$

   Each particles $s_{t,m}^{(i)}$ is an instance of the multi-variate model configuration X which encodes the location and state of the tracked body.
4. Each particles $s_{t,m}^{(i)}$ is then assigned a weight $\pi_{t,m}^{(i)}$ by using weighting function $w(Z_t, X)$ corresponding to the likelihood of X given the image features in Z_t scaled by an annealing level factor. The result is the weighted particle set:

   $$S_{t,m}^{\pi} = \{(s_{t,m}^{(1)}, \pi_{t,m}^{(1)})...(s_{t,m}^{(N)}, \pi_{t,m}^{(N)})\}$$

5. N particles are randomly drawn from set $S_{t,m}^{\pi}$ with a probability equal to their weight $\pi_{t,m}^{(i)}$ to obtain a temporary weighted particle set. Each particle $\overline{s}_{t,m}^{(i)}$ of the temporary set is then used to produce particle

   $$\overline{s}_{t,m-1}^{(i)} = \overline{s}_{t,m}^{(i)} + B_m$$

   where B_m is a multi-variate Gaussian random variable. The result is particle set $S_{t,m-1}^{\pi}$ which is used to initialize layer $m - 1$.
6. The process is repeated until all layers have been processed and the final particle set $S_{t,0}^{\pi}$ has been computed.
7. $S_{t,0}^{\pi}$ is used to compute the estimated model configuration X_t for time step t by calculating the weighted average of all configuration instances:

   $$X_t = \sum_{i=1}^{N} s_{t,0}^{(i)} \pi_{t,0}^{(i)}.$$

8. The set $S_{t+1,M}$ is then produced from $S^{\pi}_{t,0}$ using

$$s^{(i)}_{t+1,M} = s^{(i)}_{t,0} + B_0.$$

In the subsequent time step $t + 1$ the set $S_{t+1,M}$ is used to initialize layer M.

The likelihood $w(Z_t, s^{(i)}_{t,m})$ which is used to determine the particle weights $\pi^{(i)}_{t,m}$ is computed by projecting the geometry of the human body model into the image observations Z_t for each camera and determining the error based on the image features. The likelihood is a measure of the 3D body model alignment with the foreground and edges in the images. The body model consists of conic cylinders to rep-resent 10 body parts 2 for each limb plus the torso and the head. Each cylinder is represented by a length and a radius for each end. The body parts are assembled into a kinematic tree based upon the joint angles. Each particle represents the set of joint angles plus a global translation.

`bodytrack` has a persistent thread pool. The main thread executes the program and sends a task to the thread pool with method *SignalCmd* whenever it reaches a parallel kernel. It resumes execution of the program as soon as it receives the result from the worker threads. The program has three parallel kernels:

- **Edge detection (Step 1)**: employs a gradient based edge detection mask to find edges. The result is compared against a threshold to eliminate spurious edges. The output of this kernel will be further refined before it is used to compute the particle weights.
- **Edge smoothing (Step 1)**: a separable Gaussian filter of size 7x7 pixels is used to smooth the edges. The result is remapped between 0 and 1 to produce a pixel map in which the value of each pixel is related to its distance from an edge. The kernel has two parallel phases, one to filter image rows and one to filter image columns.
- **Calculate particle weights (Step 4)**: valuates the foreground silhouette and the image edges produced earlier to compute the weights for the particles. This kernel is executed once for every annealing layer during every time step, making it the computation ally most intensive part of the body tracker.

The parallel kernels use tickets to distribute the work among threads balance the load dynamically.

The software-knobs and the input size are listed in the following table:

Name	Type	Values
`num_threads`	int	[1 - 8]
`annealing`	int	[1 - 8]
`particles`	int	[1000, 2000, 3000, 4000]
Input Size	4 frames, 4,000 particles	

5.3.5 Other Target Applications

The following applications are not part of the PARSEC benchmark suite and have been included as additions.

Stereomatch

The `stereomatch` application is used in image-processing to compute disparities between a pair of stereo images (the same scene observed by two cameras). The disparity information can then be processed to calculate the depth of the objects in the scene, given that objects close to the cameras are characterized by a higher disparity. The output of this application is required for estimating the depth of the objects in the scene. The application is based on the *OpenCL* library.

The algorithm derived from [56] builds adaptive-shape support regions for each pixel of an image, based on colour similarity, and then it tries to match them on the other image, computing its disparity value. The algorithm implementation [57] exposes three application-specific parameters to modify the effort spent on building the support regions and on matching them in the second image to trade off the accuracy of the disparity image (the Stereomatching output) and the execution time (and thus the reachable application throughput). The application-specific parameters are listed below:

- **Confidence**: defines a threshold for color similarity with respect to the anchor pixel when building a support region.
- **Max_arm_length**: the support region for each pixel is encoded in a cross structure and all pixels covered by the cross arms (left, right, up and down) are similar in color to the anchor pixel. This parameter represents the maximum arm length and can thus limit the size of support windows.
- **Hypo_step**: this parameter defines the set of integer disparities that the algorithm should test for each pair of pixels. Each disparity value is a "hypothesis" that is actually evaluated by the algorithm. The disparity hypotheses (in pixels) will range from 0 to a fixed max value, with a given hypo_step step. The latter parameter acts as the "resolution" of the algorithm, by setting the step between consecutive disparity hypotheses.

In addition, the application has been parallelized by using OpenMP, making available as fourth parameter the number of threads used for the computation. For each pair of stereo images, the following five OpenCL kernels are executed.

- **WinBuild (left and right cameras)**: two instances of the same OpenCL kernel (one for the left and one for the right image) can be submitted in parallel to the OpenCL command queue to build the local support regions for each pixel in the image. The search span is limited by the application parameter *max_arm_length* and the arms cover all the pixels that satisfy the condition of color similarity specified by *confidence*.
- **Matching-cost aggregation**: this kernel evaluates all the disparity hypotheses by computing the matching-cost associated with support regions of pixels on the same line in the two reference images. A Winner-Takes-All (WTA) decision is then taken for selection of the "cheapest" disparity hypothesis in terms of matching-cost.

- **FinalDecision**: this kernel simply considers the results of all the workgroups involved in the previous step and decides the global disparity result. It can be seen as a reduction step.
- **Refinement**: it performs a regularization (smoothing) of disparities in the support region.

The software-knobs and the input size are listed in the following table:

Name	Type	Values
`num_threads`	int	[1 - 8]
`confidence`	int	[14, 24, 34, 44, 54, 64]
`max_arm_length`	int	[1 - 17]
`hypo_step`	int	[1, 2, 3]
Input Size	15 image pairs (Left & Right)	

K-means

The `k-means` application implements an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. We have already discussed about this algorithm in Sub-Section 2.5.1 so we are not going in further details.

This application have been chosen manly because it enables us to enforce the input features sensitivity of *Agora*. The input dataset is indeed heterogeneous and composed of matrices of different size representing different sets of random points in two dimensions. Unlike previous target applications, `k-means` defines, aside from the execution time, an *error* metric which is a measure of the result accuracy with respect to a reference scenario (i.e. another run on the same dataset using maximum number of iterations available). The error E is defined as:

$$E = \frac{\sum_{i}^{N_{rows}} \sqrt{\sum_{j}^{N_{cols}} (a_{ij} - r_{ij})^2}}{N_{rows}}$$

where A is the input matrix, R is the reference matrix and N_{rows}, N_{cols} are respectively the number of rows and columns. The input feature monitored during execution is `size` that corresponds to the current input matrix size ($num\_rows \cdot num\_columns$).

The software-knob and the input size are listed in the following table:

Name	Type	Values
`num_iterations`	int	[3, 5, 10, 30, 50, 70, 90, 100]
Input Size	Heterogeneous data-points matrices with 20 features each	

5.4 Evaluating the Iterative Learning Approach

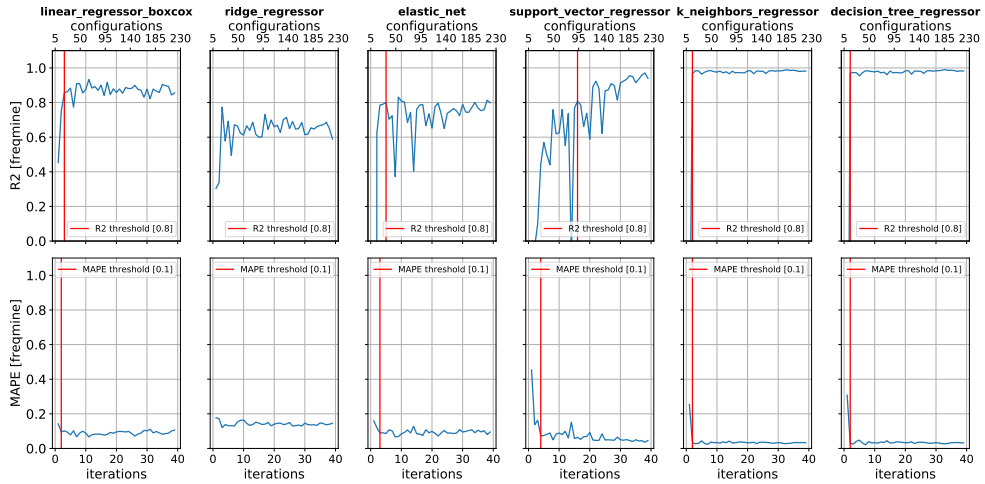
This section aims at experimentally assessing the ability of *Agora* to learn the application-knowledge at runtime using the applications described previously in Section 5.3. To address this, the following experiments evaluate out-of-sample predictions during the learning process by varying the number of iterations and therefore increasing the fraction of design space considered over time. The bandwidth on the network and the HDD play a large role in performance. Certain applications perform markedly different when running locally versus having the executables and input files served over the target platform (different bandwidth and I/O delays due to the NFS protocol). They might indeed exhibit significant variations in execution time across multiple runs. For this reason, each of the following experiments was repeated 10 times, taking the average of the results.

5.4.1 Models Accuracy

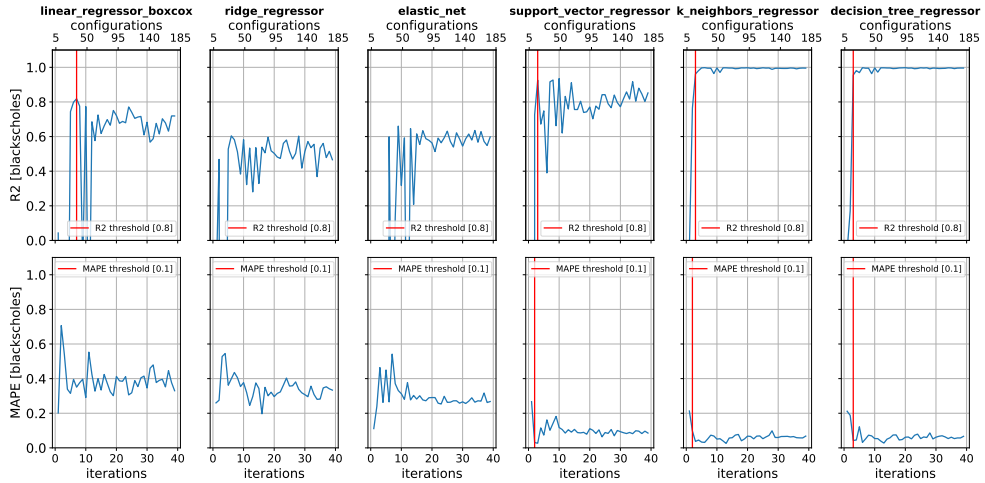
For each application, Figure 5.5 shows the evaluation of all the available models outlined in Sub-Section 3.2.2. The y-axis represents the model quality in terms of the coefficient of determination R^2 and the mean absolute percentage error $MAPE$ computed. The x-axis shows the number of iterations in the bottom part and the number of explored configurations in the upper part. Every application reports a single metric which is the **execution time** (in *milliseconds*). We omitted the `k-means` error metric for graphical reasons since it displayed the same trend as the other. As stated in 3.2.2, the end-user has the ability to specify a $MAPE$ threshold with a percentage value. This quality metric has general applicability and the end-user is not required to know the application behaviour (i.e. the EFP output space) in advance anymore in order to quantify the goodness of a model. Ideally the best model is expected to reach an R^2 score of 1 and a $MAPE$ score of 0.

Every application has the `max_number_of_iterations` capped to 40 except for `stereomatch` which was set to 80 due to a much larger Design Space. These values have proved to be a good restriction in order to show the trend even beyond the threshold limits. The `number_of_observations_per_configuration` was set to 2 and that enables the framework to have more accuracy over the same configuration. Moreover, in case of input features, this allows to potentially explore the same configuration on different inputs. To explore the DS in a reasonable manner and to produce an output which could be graphically interesting, the `number_of_configurations_per_iteration` ranged between 5 and 30 depending on the application DS size. Big DSs like `bodytrack` or `stereomatch` are set with higher values in order to explore more efficiently.

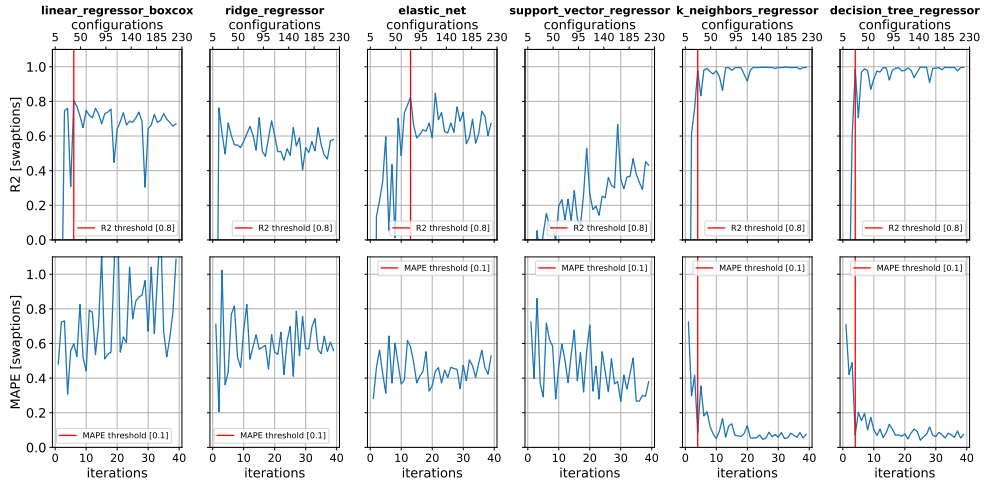
We start by quickly addressing the results obtained by each model separately reminding that the objective of this thesis is not comparing different modelling techniques but to provide a framework that exploits them in order to learn the application-knowledge at runtime.



(a) `freqmine`

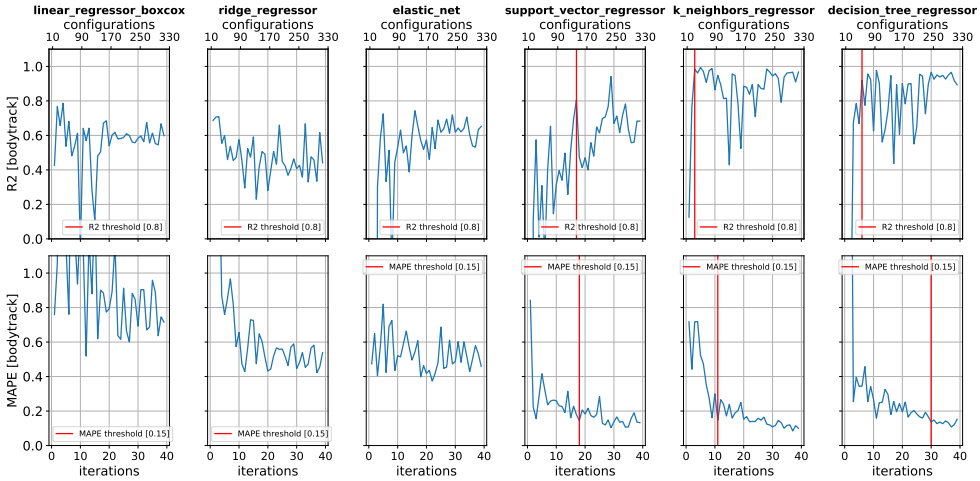


(b) `blackscholes`

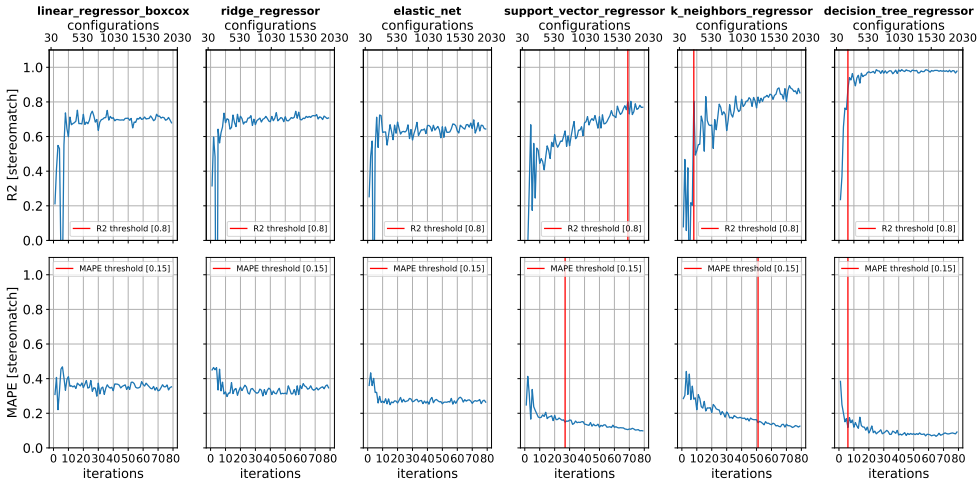


(c) `swaptions`

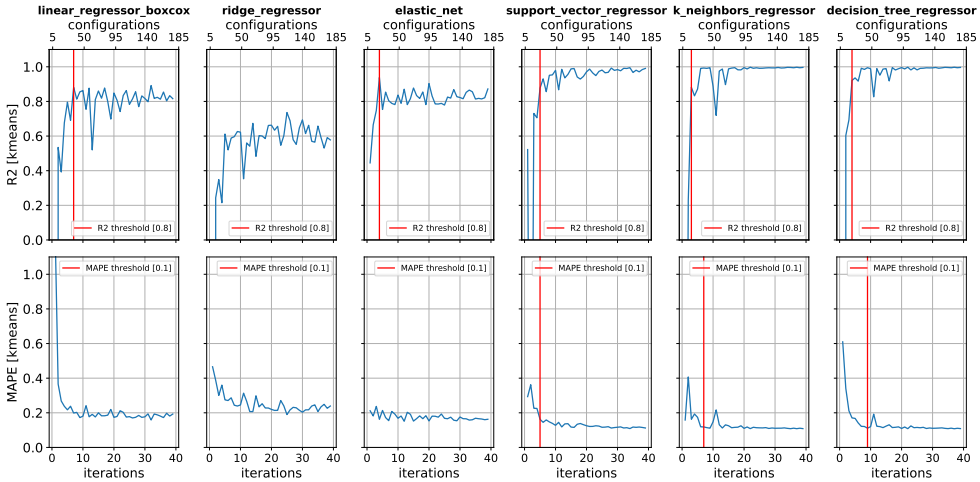
Figure 5.5: *Coefficient of determination R^2 and mean absolute percentage error $MAPE$ of the target applications' EFP models, by varying the number of iterations and the number of explored software-knobs configurations during DSE.*



(d) `bodytrack`



(e) `stereomatch`



(f) `k-means`

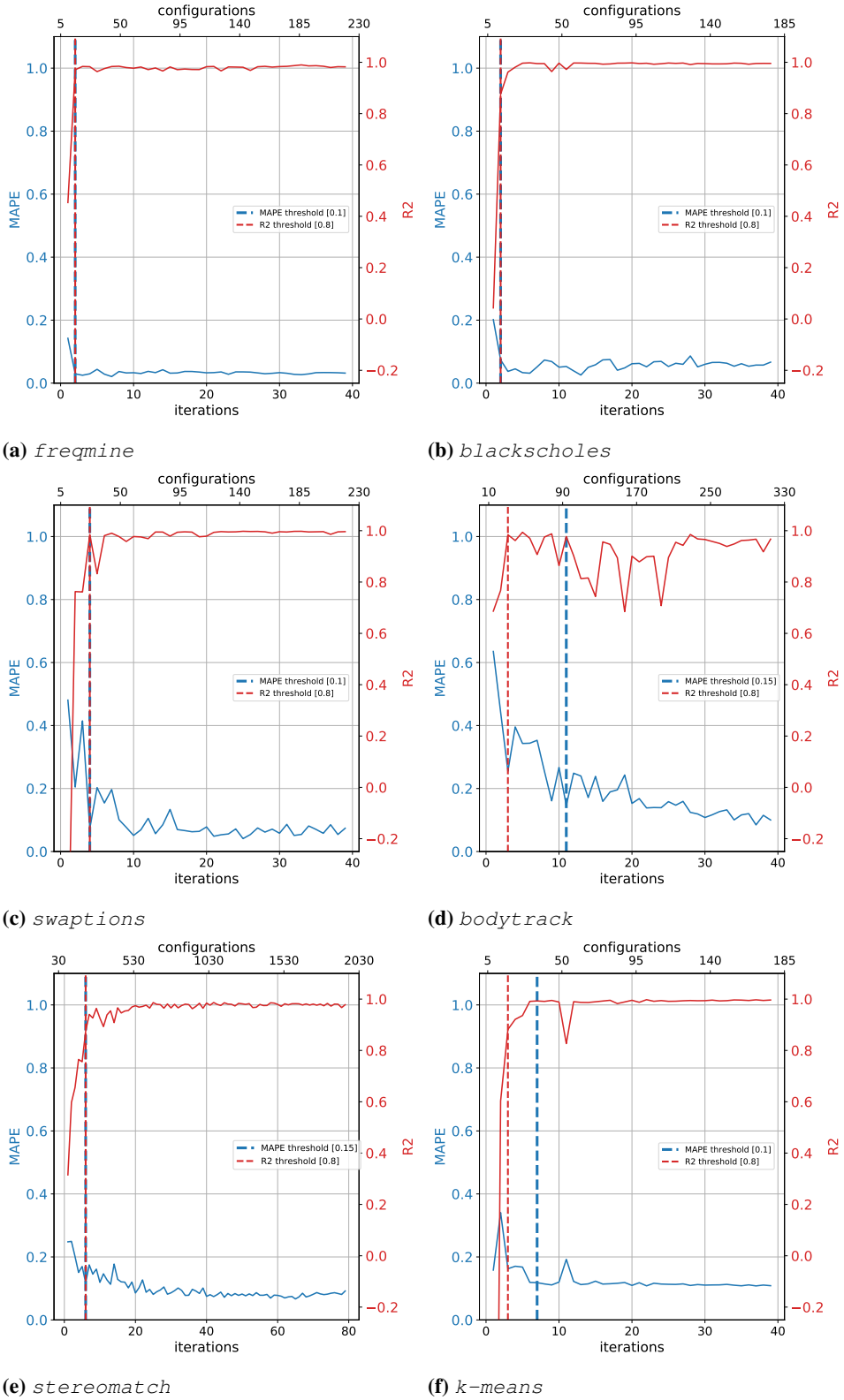
Figure 5.5: *(cont.)*

From the experimental results, a trend on both the R^2 and the $MAPE$ metrics is noticeable. It respectively increases and decreases with an increasing number of iterations and hence an increasing number of configurations explored. A red vertical line identifies the iteration number which produced an acceptable model depending on the predefined limit threshold (set by default to $R^2 = 0.8, MAPE = 0.10$ *or* 0.15). A model is deemed acceptable if both metrics have verified the threshold limits. If a modelling technique is able to achieve $R^2 > 0.8$ it doesn't mean that the $MAPE$ will be verified as well at the same iteration.

We can notice that the linear models perform, as expected, generally worse than the others. For instance, in `blackscholes` and `swaptions` during the initial phase there are notable negative spikes due to a probable insufficient number of data samples that makes the validation phase behave irregularly. The SVR model while generally maintaining the trend of the error below the linear ones, in some cases have a worst R^2 score in comparison. This is an important aspect to highlight since SVR internally runs a kernel called *Radial Basis Function (RBF)*. Without going into details, the kernel is subject to two important parameters: the regularization parameter C and the kernel coefficient $gamma$. Proper choice of C and $gamma$ is critical to the SVM's performance. That's why in this scenario the end-user should be required to find and set a proper value for both but consequently losing the concept of *auto*-tuning itself. An alternative could be exploiting a different plugin that performs a *GridSearch* cross-validation which objective is to find the optimal hyperparameters of the model. However, this approach is costly in terms of computational effort and was not investigated further during this thesis' work.

The selection of an heterogeneous pool of applications also wants to show how the framework is able to adapt to different scenarios, going from applications that require from 1 to 5 iterations in order to find a suitable application-knowledge (e.g. `blackscholes`, `freqmine` or `swaptions`) and others that need a higher number (e.g. `bodytrack` or `stereomatch`). Concerning the first type, one can notice how the coefficient of determination and the mean absolute percentage error quickly achieve a reasonable value. On the contrary the other type of applications struggles during the initial phase, given most likely a nonlinear relationship between software-knobs and EFP. For instance the R^2 metric of `bodytrack` has an irregular progression even with the best models and the $MAPE$ takes much more time to reach the predefined quality threshold. The `stereomatch` application despite having a huge Design Space, registered quite good results overall and even if only the k-neighbors and decision-tree regressors were able to achieve optimal results, the other linear models performed decently as well. This means that the relationship between software-knobs and the EFPs is much more linear and regular although having a number of configurations that big.

Finally, another aspect to highlight is that the learning process may lead to an improvement beyond the threshold limit but the framework is designed to stop as soon as the target quality is achieved for each of the available EFPs models. Being aware of this behaviour is important because as a consequence the end-user is capable of managing the trade-off between the learning phase duration and the quality of the results, by using higher (or lower) threshold values.

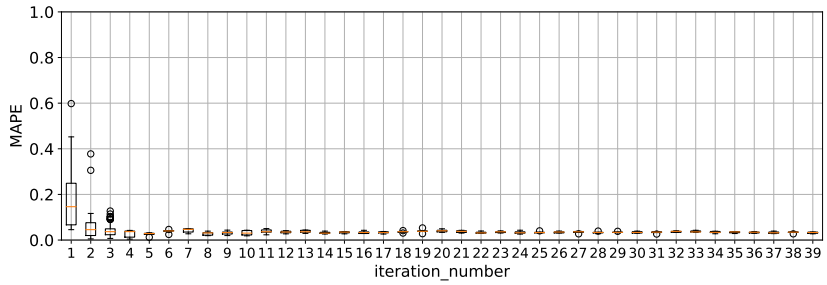


(a) `freqmine` **(b)** `blackscholes`

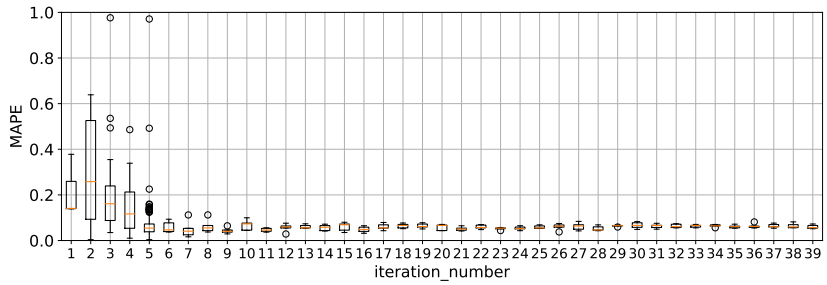
(c) `swaptions` **(d)** `bodytrack`

(e) `stereomatch` **(f)** `k-means`

Figure 5.6: *Coefficient of determination R^2 and mean absolute percentage error $MAPE$ of the best model found, by varying the number of iterations and the number of explored software-knobs configurations during DSE.*



(a) `freqmine`



(b) `blackscholes`

Figure 5.7: *The distribution of the mean absolute percentage error $MAPE$ of the target applications' EFPs models.*

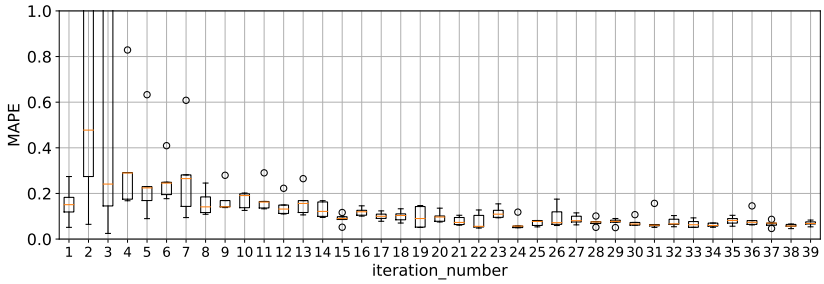
In Figure 5.6 we report for each application a trend of the quality metrics recorded by the best performing model found at each iteration. The coefficient of determination and the mean absolute percentage error are put together in the same graph in order to show how they create a sort of horizontal "V" shape with an increasing number of iterations. This proves on one hand how the same trend previously highlighted is reflected on the final model as well, on the other hand that selecting the best model at each iteration gives us a general approach that is flexible on different kind of applications/data and performs significantly better during the whole phase with respect to every listed model taken individually.

5.4.2 Error Distribution

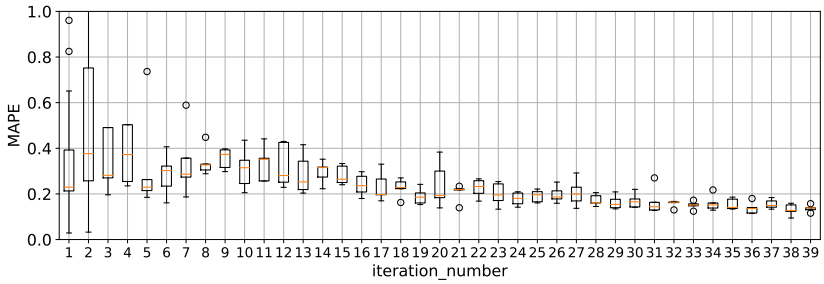
In order to get a better grasp on the model accuracy, in Figure 5.7 we show a box-plot of the error distribution registered during the best model validation for each application.

The box-plots show how the first iterations are subject to the presence of *outliers*. An outlier is a value that lies in a data series on its extremes, which is either very small or large and thus can affect the overall validation. The box-plot is a method typically depicted by quartiles and inter quartiles that helps in defining the upper limit and lower limit beyond which any data lying will be considered as outliers. The very purpose of this diagram is to identify and discard them from the data series so that the conclusion made from the study gives more accurate results not influenced by any extremes or abnormal values.

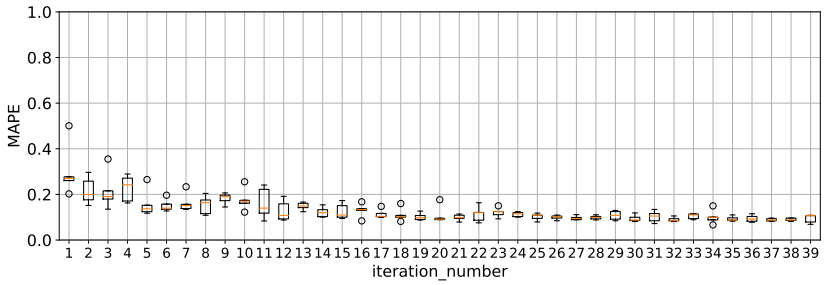
As we can see, every application displays a trend of convergence, reducing the boxplot height with an increasing number of iterations.



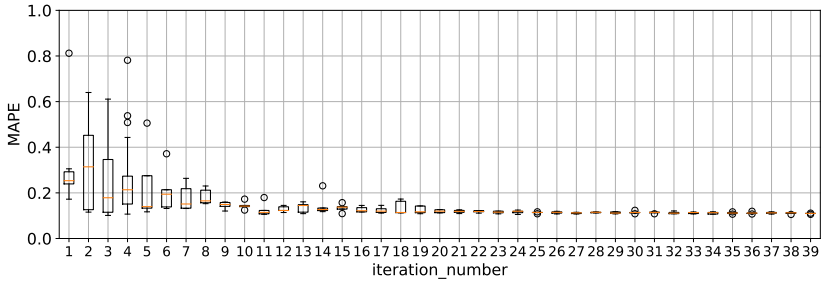
(c) `swaptions`



(d) `bodytrack`



(e) `stereomatch`



(f) `k-means`

Figure 5.7: *(cont.)*

5.4.3 Prediction quality with respect to the percentage of explored Design Space

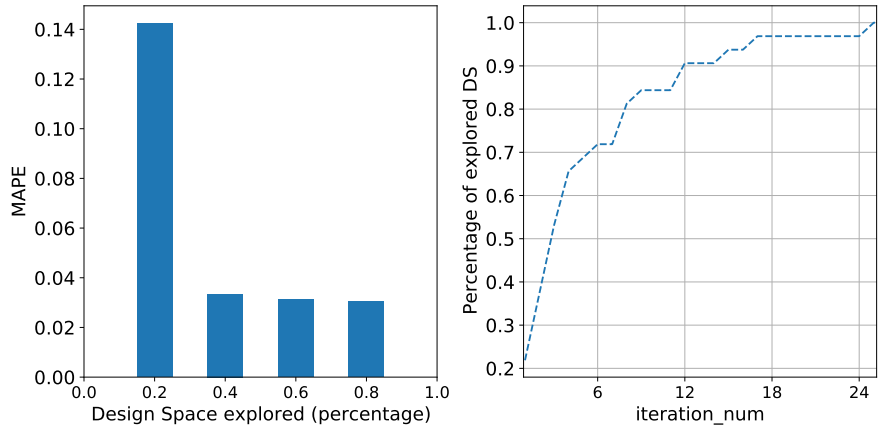
Another important aspect that must be considered is measuring the model quality with respect to the Design Space *explored percentage*. Figure 5.8 aims at showing this matter. The percentage value is calculated as:

$$\frac{EC_i}{TotConfigs}$$

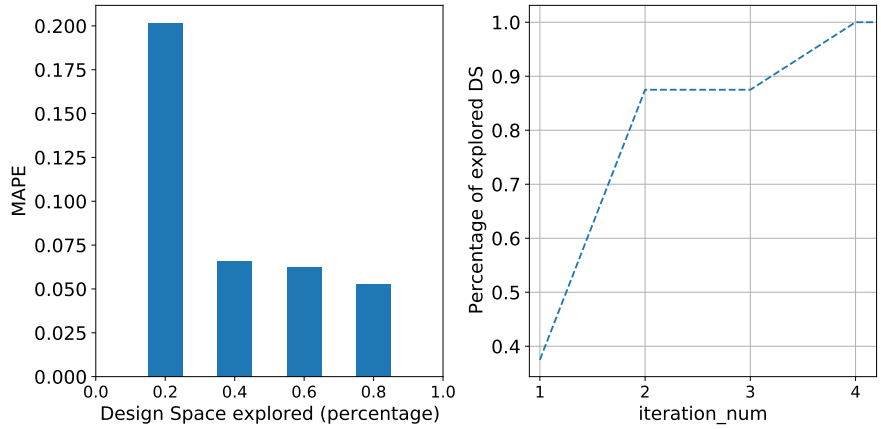
where EC_i is the number of distinct configurations explored at iteration i and *TotConfigs* is the total number of configuration inside the DS (i.e. the number of data points in a full-factorial design). On the left the reader is presented with the mean absolute percentage error trend registered at four levels: 20%, 40%, 60% and 80%; on the right a line plot explains how fast *Agora* was able to cover the whole DS, which ideally is achieved by reaching a value of 100%. As we identified already earlier, there is a common trend among the applications: a decreasing $MAPE$ with increasing number of configurations explored (hence a bigger percentage of DS). One can notice how after exploring just the 40% of the space we generally achieve a $MAPE$ that verifies the thresholds and consequently gives us satisfying prediction results. Continuing the DSE, the error keeps decreasing but without manifesting a huge gap like between 20% and 40%.

The target applications which were tested are substantially monolithic programs created to run on a different number of threads. Because they are simple, their corresponding DS is small and *Agora* is generally capable of covering it in a small fraction of time, depending of course on each application average execution time. However this doesn't generally hold with more complex applications (e.g. `bodytrack` or `stereomatch`). As a matter of fact the number of configurations is so high that the required number of iterations to explore the DS as a whole tends to grow a lot larger. Even `k-means` requires more iterations due to the fact that we're also considering unbounded input-features which contributes in expanding the design space even more. The real efficiency of the learning method proposed relies here. In a purely automatic way *Agora* is able to provide a suitable application-knowledge by exploring just a small percentage of the whole design space, gaining a clear advantage with respect to a full exploration. Indeed, many other approaches consequently force the user to decide on having a longer but useless exploring process or a short but inaccurate one.

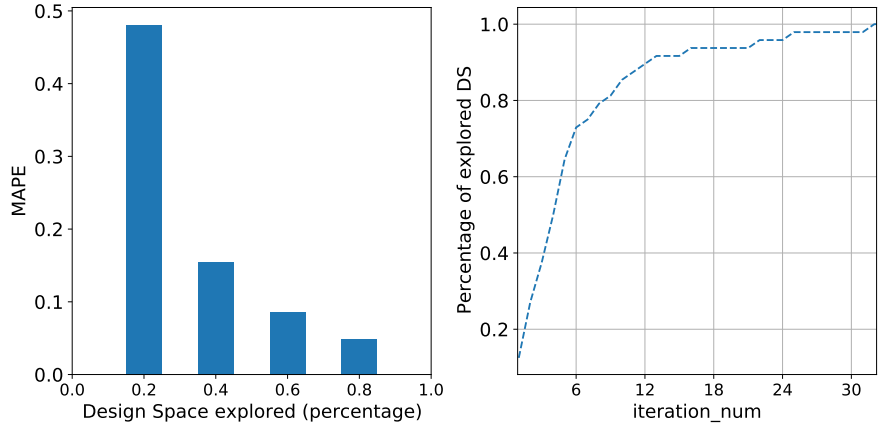
Moreover, even by reaching almost a 100% percentage value, the $MAPE$ is still greater than zero (which ideally is what we could expect after exploring the whole DS). This outlines that there is still a strong dependency on the analyzed data and that the model could be fitting a function that is non-deterministic. This is especially true for input-sensible applications like `k-means`. Therefore to increase the accuracy, one should even go beyond the 100% and retrieve more observation of the same software-knobs configuration. This is a fundamental aspect that give us an indication in order to make a related decision. Dealing with non-critical applications, every decision taken at runtime (e.g. the selection or understanding of which are the best performances) can't be achievable during design-time. Reaching the 100% doesn't tell the whole story. Collecting every point still remains sensitive to the dataset, to the system noise, etc. and that's why just one iteration is not enough.



(a) `freqmine`

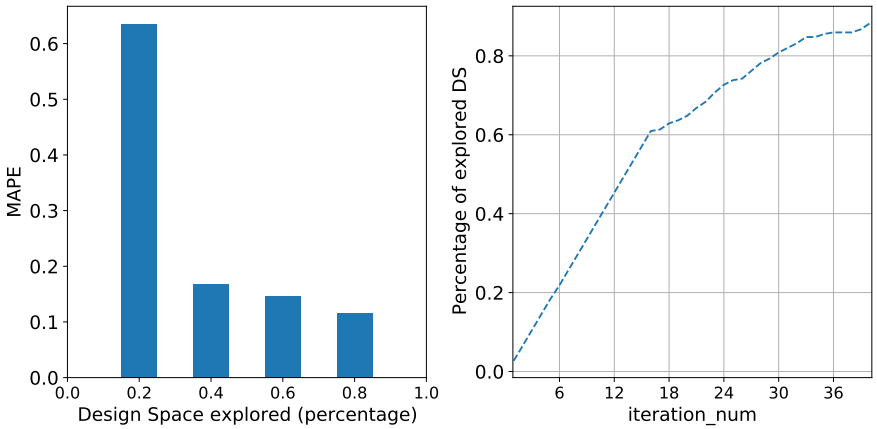


(b) `blackscholes`

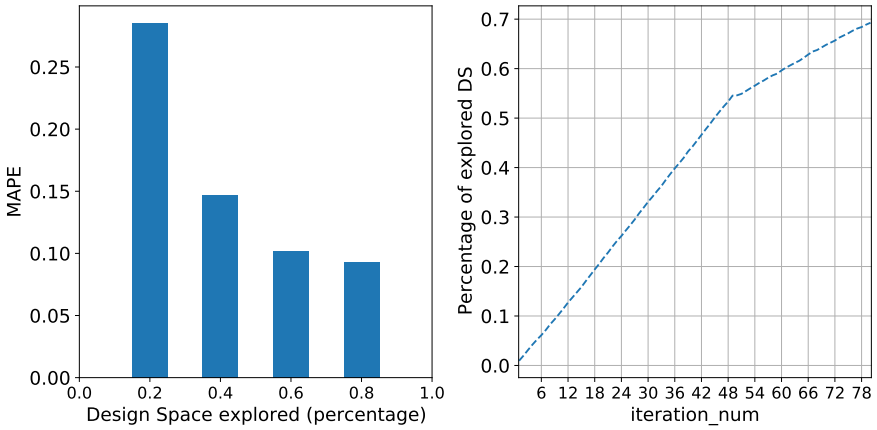


(c) `swaptions`

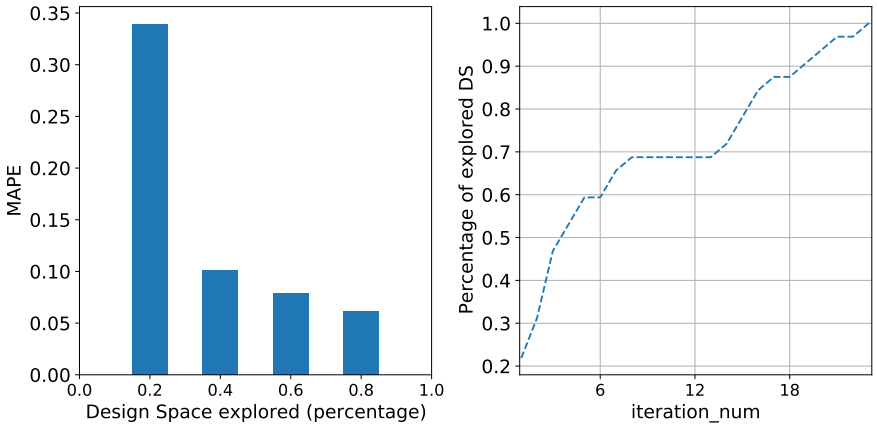
Figure 5.8: *On the left: mean absolute percentage error $MAPE$ trend with respect to the percentage of explored DS. On the right: the percentage of explored DS with an increasing the number of iterations.*



(d) *bodytrack*

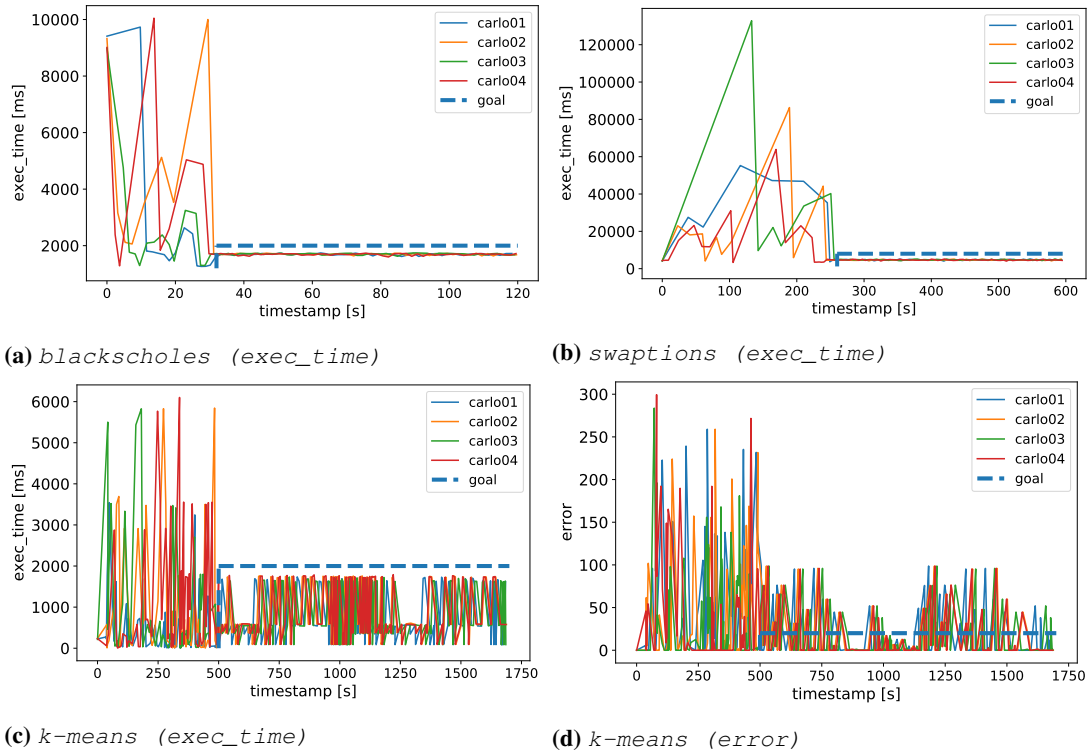


(e) *stereomatch*



(f) *k-means*

Figure 5.8: *(cont.)*



(a) `blackscholes (exec_time)`

(b) `swaptions (exec_time)`

(c) `k-means (exec_time)`

(d) `k-means (error)`

Figure 5.9: *Execution trace showing the initial learning phase and the subsequent tuning phase for each of the four application instances running on the carlo cluster. The blue line marks the separation between the two phases and highlights the application goal (e.g. minimizing the execution time).*

5.4.4 Execution Trace

To end this section, Figure 5.9 shows an execution trace for some of the target applications running simultaneously on the four computing nodes over the *carlo* cluster. We omitted the rest of the applications since they exhibit a similar trend. Each sub-figure represents the EFP behaviour of the four processes. The trace can be divided in two phases: the *training* phase and the *tuning* phase. The length of the training phase is determined by the model convergence time and on the input characteristics (i.e. the `k-means` execution). During the first phase *Agora* performs the Design Space Exploration and each client is forced with a different configuration. Once the training phase is terminated then the application-knowledge is produced. *mARGOt* receives the knowledge and sets each application instance with a software-knob configuration that verifies the user-defined objective function. It is worth to notice that on one hand the same software-knob configuration is set on each instance because all of them are part of the same experiment; on the other hand the training phase length is almost the same for all the clients thanks to the configuration distribution performed by the Remote Application Handler.

Concerning the `k-means` execution, one can appreciate how *mARGOt* exploits the application-knowledge, which in that case is composed of multiple clusters of Operating Points, by setting different configurations each time a new input feature is observed and trying to minimize the error while keeping the execution time below 2 seconds.

5.5 Summary

This chapter assessed the proposed framework by evaluating the iterative learning approach, the features exposed and by measuring the introduced overheads inside the communication channel. From the experimental results, it is possible to notice how the proposed methodology is able to provide an application-knowledge that has been correctly judged before broadcasting it, meaning that a good model is found and it is able to make good quality predictions. Moreover, we've shown that the major benefit is that *Agora* is able to drastically reduce the time required to perform a Design Space Exploration, which can grow exponentially when considering several and usually unbounded software-knobs.

In the final chapter we will summarize the achievements and provide recommendations for future works.

CHAPTER 6

Conclusions

Application autotuning is a promising path investigated in literature to improve computation efficiency. The relationship between an application configuration and the extra-functional properties might depend on the underlying architecture, on the system workload and on features of the current input. The application-knowledge, which autotuning frameworks rely on to drive their workflow, is typically produced off-line and depends on an expensive phase called Design Space Exploration (DSE) whose fruition requires significant effort in order to reduce its overhead. In this thesis, we addressed the problem on how to contain the exponential growth of the Design Space when considering several, and usually unbounded, application-specific parameters coupled with the input features. This chapter summarizes the contributions and provides recommendations for future works.

6.1 Main Contributions

The outcome of this thesis is a framework exploiting a model-driven approach to learn the application-knowledge at runtime and in a distributed fashion. In order to minimize the learning time, the framework applies known machine-learning techniques to determine the relationship between software-knobs, extra-functional properties and input features, leveraging an iterative procedure that samples the Design Space until the computed models reach the target quality. We have experimentally evaluated the proposed approach on some well-known benchmark applications, studying the consequent benefits and limitations in terms of overhead and model quality. Considering the features introduced by the proposed framework, their main results can be summarized as follows:

1. Since the framework is designed to be deployed mainly in a distributed context, we evaluated its scalability and throughput inside a cluster environment with four

virtual machines and one server node, powered by *PoliCloud*. Experimental results show how by varying the work-group size we're able to manage up to $\approx 20k$ requests per second.

2. The framework capabilities have been evaluated by integrating it inside a pool of applications taken from the *PARSEC* benchmark suite and some additional (`stereomatch` and `k-means`). First, by evaluating the models accuracy in terms of the coefficient of determination R^2 and the mean absolute percentage error $MAPE$, we have shown how the best resulting model was capable of learning the EFPs relationships in a reasonable number of iterations, without exploring the application design space in its entirety. By comparing the errors with respect to the percentage of explored Design Space we addressed the real benefits introduced by the iterative learning approach, showing how sampling the Design Space in an efficient way, exploiting Design of Experiments methods and machine-learning techniques, leads to significant benefits.

3. The execution trace monitored for some of the tested applications shows a clear distinction between the learning phase and the tuning phase, demonstrating how, after exploring different software-knobs configurations, the application-knowledge is generated and the *mARGOt* autotuner is able to exploit it by setting the best configuration that verifies the user-defined objective function. In particular, `k-means`' trace shows how the input-features are considered during the application-knowledge creation and, depending on the current input, the framework is able to dynamically tune it setting different configurations based on each cluster representative.

6.2 Future Works

Although experimental evaluations have shown promising results, there are still limitations and open questions that needs to be investigated. The most challenging points are the following:

1. The throughput measurements reach a significant value but could remain a limiting factor with micro-kernels running inside highly parallelized systems. To this end, we believe that the main bottleneck resides in the type of storage that *Agora* exploits. CSV tables are simple and easy to manage but inside a distributed context they lead to a high number lock/unlock primitives calls. A potential improvement could be integrating *Agora* inside a database context which should release a lot of I/O pressure. For example, Cassandra seems to be a promising path to follow thanks to its properties of fault tolerance, decentralization and scalability.

2. The design of experiment algorithms considered in this thesis rely on implementations taken from the PyDOE package [45]. The package offers other good approaches which integration and comparison could be an interesting study to perform (e.g. Box-Behnken design 2.3.2). As a negative note, they all remain *non-sequential* design techniques. A DoE technique is deemed sequential if, by providing an older design as input, is capable of producing a new design "on top" of that input, meaning that only new unseen experiments are added while still maintaining the old ones. The design of new techniques or modifications of

existing ones exploiting this property may improve even more the experiments selection prior to the DSE phase.

3. *mARGOt* can distinguish between deterministic EFPs and not. By specifying a confidence level, the end-user is able to indicate the number of time that the Operating Point standard deviation must be considered. Since *Agora* doesn't consider this aspect, a future improvement would be modifying the prediction system in order to derive a standard deviation for each of the OPs before packing them into the final application-knowledge.

4. The modelling phase is the core of the learning approach and also the most susceptible to changes. We have tested the framework on a relatively small pool of target applications that, despite being as diverse and representative as possible, are kind of the tip of the iceberg. There are lots of scenarios which could be interesting to test *Agora* on, in the context of smart cities or drug discovery processes for instance. This may need a more robust machine-learning procedure that makes the current version insufficient.

We have decided to omit other minor improvements and fixes that could be performed in future developments. They remain all listed and documented inside the project repository [9], which is publicly disclosed.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, June 2011.

[2] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in High-Performance Computing Applications," *Proceedings of the IEEE*, vol. 106, pp. 2068–2083, Nov. 2018.

[3] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 199–212, Mar. 2011.

[4] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, vol. 48, pp. 62:1–62:33, Mar. 2016.

[5] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," Sept. 2009.

[6] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, (New York, NY, USA), pp. 324–334, Association for Computing Machinery, June 2006.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan. 2003.

[8] D. Gadioli, E. Vitali, G. Palermo, and C. Silvano, "mARGOt: A Dynamic Autotuning Framework for Self-Aware Approximate Computing," *IEEE Transactions on Computers*, vol. 68, pp. 713–728, May 2019.

[9] B. Menicagli, "mARGOt framework with Agora support git repository," Feb. 2021.

[10] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology." /paper/Autonomic-Computing%3A-IBM%27s-Perspective-on-the-State-Horn/1ad1c619a9b3ba5a3ac597f51c8d15011a83423b, 2001.

[11] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster Computing*, vol. 12, pp. 1–15, Mar. 2009.

[12] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, "Fulfilling the Vision of Autonomic Computing," *Computer*, vol. 43, pp. 35–41, Jan. 2010.

[13] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing Sequential Programs with Statistical Accuracy Tests," *ACM Transactions on Embedded Computing Systems*, vol. 12, pp. 88:1–88:26, May 2013.

[14] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 35–50, Association for Computing Machinery, Feb. 2014.

[15] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, "Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs," *IEEE Transactions on Software Engineering*, vol. 45, pp. 219–236, Mar. 2019.

[16] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications," in *Euro-Par 2009 Parallel Processing* (H. Sips, D. Epema, and H.-X. Lin, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 9–20, Springer, 2009.

[17] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 303–316, Association for Computing Machinery, Aug. 2014.

[18] A. Rasch, M. Haidl, and S. Gorlatch, "ATF: A Generic Auto-Tuning Framework," in *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 64–71, Dec. 2017.

[19] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 38–38, Nov. 1998.

[20] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, pp. 216–231, Feb. 2005.

[21] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, pp. 521–530, Jan. 2005.

[22] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," *The International Journal of High Performance Computing Applications*, vol. 18, pp. 21–45, Feb. 2004.

[23] C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-Tuner for OpenCL Kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip*, pp. 195–202, Sept. 2015.

[24] J. Shen, A. L. Varbanescu, H. Sips, M. Arntzen, and D. G. Simons, "Glinda: A framework for accelerating imbalanced applications on heterogeneous platforms," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, May 2013.

[25] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 676–687, May 2011.

[26] S. A. Kamil, "Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages," tech. rep., CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, Jan. 2013.

[27] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, (New York, NY, USA), pp. 198–209, Association for Computing Machinery, June 2010.

[28] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 13–24, Association for Computing Machinery, Dec. 2013.

[29] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," *ACM SIGPLAN Notices*, vol. 50, pp. 379–390, June 2015.

[30] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive Control of Approximate Programs," *ACM SIGPLAN Notices*, vol. 51, pp. 607–621, Mar. 2016.

[31] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 161–176, Association for Computing Machinery, June 2016.

[32] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2017.

[33] M. Cavazzuti, "Design of Experiments," in *Optimization Methods: From Theory to Design Scientific and Technological Aspects in Mechanics* (M. Cavazzuti, ed.), pp. 13–42, Berlin, Heidelberg: Springer, 2013.

[34] NIST, "1. Exploratory Data Analysis," . *Exploratory Data Analysis*, p. 2333.

[35] T. S. Madhulatha, "An Overview on Clustering Methods," *arXiv:1205.1117 [cs]*, May 2012.

[36] L. Rokach and O. Maimon, "Clustering Methods," in *Data Mining and Knowledge Discovery Handbook* (O. Maimon and L. Rokach, eds.), pp. 321–352, Boston, MA: Springer US, 2005.

[37] H. C. Hwang, J. Park, and J. G. Shon, "Design and Implementation of a Reliable Message Transmission System Based on MQTT Protocol in IoT," *Wireless Personal Communications*, vol. 91, pp. 1765–1777, Dec. 2016.

[38] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi, "Internet of Things: Survey and open issues of MQTT protocol," in *2017 International Conference on Engineering MIS (ICEMIS)*, pp. 1–6, May 2017.

[39] D. Gadioli, "Dynamic Application Autotuning for Self-aware Approximate Computing," in *Special Topics in Information Technology* (B. Pernici, ed.), pp. 91–102, Cham: Springer International Publishing, 2020.

[40] Apache, "Apache Cassandra database," Feb. 2021.

[41] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and Computing*, vol. 14, pp. 199–222, Aug. 2004.

[42] "Paho MQTT C Client Library." https://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/index.html.

[43] "Boost C++ Libraries." https://www.boost.org/.

[44] S. Kumar, "Python-dotenv: Add .env support to your apps in development and deployments."

[45] M. Baudin, M. Christopoulou, Y. Collette, and J.-M. Martinez, "PyDOE package for experimental designs," 2013.

[46] D. Cournapeau, "The scikit-learn package homepage," Dec. 2020.

[47] Joblib developers, "JobLib package homepage," 2008.

[48] "Pandas - Python Data Analysis Library." https://pandas.pydata.org/.

[49] "NumPy." https://numpy.org/.

[50] "Environment Modules documentation." https://modules.readthedocs.io/en/latest/.

[51] "Slurm Workload Manager - Documentation." https://slurm.schedmd.com/.

[52] "Eclipse Mosquitto." https://mosquitto.org/, Jan. 2018.

[53] T. Robitaille, "Psrecord: Python package to record activity from processes."

[54] "The PARSEC Benchmark Suite." https://parsec.cs.princeton.edu/index.htm.

[55] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, Association for Computing Machinery, Oct. 2008.

[56] K. Zhang, J. Lu, and G. Lafruit, "Cross-Based Local Stereo Matching Using Orthogonal Integral Images," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, pp. 1073–1079, July 2009.

[57] E. Paone, G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, and T. Lepley, "An exploration methodology for a customizable OpenCL stereo-matching application targeted to an industrial multi-cluster architecture," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, (New York, NY, USA), pp. 503–512, Association for Computing Machinery, Oct. 2012.