Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Self-Prioritized Modular Adaptations using Bidirectional Transformations

Duchesne, Jérémy; Lombat, Quentin

Award date:
2018

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. Jul. 2025

Université de Namur
Faculty of Computer Science
Academic Year 2017–2018

Self-Prioritized Modular Adaptations using Bidirectional Transformations

Jérémy Duchesne Quentin Lombat

Internship mentor: Zhenjiang Hu

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Pierre-Yves Schobbens

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

As self-adaptive software systems get complex, it is desirable to modularize the monitoring-analysis-planning-execution (MAPE) feedback loop into ones that manage the changes related to individual goals. However, decomposing into modules requires resolving conflicts when the shared information is updated simultaneously. To address the challenge, we propose self-prioritization on top of Kramer et al.'s three-layered architecture of self-managing systems. A complex MAPE loop is first broken down into model-based Analysis-Planning (AP) process pairs, formulating an M(AP)$^+$E loop template while sharing the monitor and execution interfaces to the managed system. Then, bidirectional transformations (BXs) are used to propagate changes between the modularized AP views, using BiGUL, a well-behaved put-based bidirectional language. Finally, conflicts between views are avoided by executing the AP pairs sequentially in a certain ordering, giving the maximum priority to the last one, and allowing it to overwrite the changes made by the previous ones. An important feature of our approach is that it frees users from specifying the changing priorities by employing a rule-based synchronizer that self-prioritizes the conflict-free execution of AP modules with respect to dynamic contexts. Our approach has been implemented and illustrated on an Infrastructure as a Service (IaaS), the Amazon Web Services (AWS) API. Its evaluation highlights the efficiency of using BiGUL, and the usefulness of the self-prioritized modular adaptation system.

Résumé

Au fur et à mesure que la complexité des logiciels auto-adaptifs augmente, il est intéressant de modulariser la boucle monitoring-analysis-planning-execution (MAPE) en plusieurs, qui gèrent les changements de façon individuelle. Cependant, la décomposition en modules demande de résoudre les conflits apparaissant lorsque l'information partagée est mise à jour simultanément. Pour résoudre ce problème, nous proposons d'ajouter le fait de donner de la priorité automatique à l'architecture en trois couches de Kramer et al. sur les systèmes qui s'auto-gèrent. Une boucle MAPE complexe est découpée en pairs d'analyse et de plan (AP), devenant ainsi une boucle M(AP)$^+$E, où les pairs d'AP partagent les mêmes étapes de monitoring et d'exécution. Ensuite, les transformations bidirectionnelles sont utilisées pour propager les changements entre les vues d'AP. Cette propagation est réalisée par BiGUL, un langage bidirectionnel dit "basé put". Enfin, les conflits entre les vues sont évitées par l'exécution séquentielle des pairs dans un ordre précis. La priorité maximum est donnée à la dernière vue exécutée, lui permettant de remplacer par son propre changement ceux effectués par de précédentes vues. Une importante caractéristique de notre approche et qu'elle libère l'utilisateur de spécifier lui-même les priorités des vues, via l'utilisation d'un synchroniseur basé sur des règles. Ce synchroniseur donne la priorité de façon automatique lors de l'exécution sans conflit des modules d'AP, tout en respectant le contexte courant. Notre approche a été implémentée et est illustrée sur l'API Amazon Web Services, un IaaS (Infrastructure as a Service). Son évaluation souligne l'efficacité de l'utilisation de BiGUL, ainsi que l'utilité de notre système.

Acknowledgements

We would like to thank Prof. Pierre-Yves Schobbens, who sent us in Tokyo and without whom we would have never been able to work at the National Institute of Informatics. We also also owe a grateful thank to Prof. Zhenjiang Hu, who warmly welcomed us and carefully followed our work all our internship long. Prof. Hu, his two secretaries, Zirun Zhu, Hsiang-Shang Ko, Yongzhe Zhang and all the other members of the lab have always been available when we needed help.

This work would never have existed without the previous work of Lionel Montrieux on the modular adaptations using bidirectional transformations. Finally, we would like to thank Prof. Yijun Yu, who gave us valuable advice during the GRACE Symposium held at the NII.

Contents

Acronyms

AMI Amazon Machine Image.

API Application Programming Interface.

Availability Probability of a system to be available when needed.

AWS Amazon Web Service.

CPU Central Processing Unit.

DSLs Domain-Specific Languages.

IaaS Infrastructure as a Service.

IP Internet Protocol.

Modifiability The cost of a change in a system.

Performance Timing taken by the system to respond to an event.

RAM Random Access Memory.

Security Ability of the system to resist unauthorized usage.

Testability Easiness of the system to show its faults.

Usability Easiness of the user to accomplish a specific task in the system.

Introduction

Software systems nowadays tend to grow bigger, and their supporting infrastructures are also increasingly complex. Therefore, it is complicated to maintain and improve software qualities (i.e., availability, performance, etc.) [6]. Separating those concerns into subsystems is a good way to facilitate their individual treatment [13].

Consider the self-adaptive software systems [2, 11, 23], where a monitoring-analysis-planning-execution (MAPE) loop [37] is often adopted to handle the system. The loop makes decisions according to the evolving environment to reach a goal. However, as self-adaptive software systems get more complex with bigger goals, a single MAPE loop becomes too complicated to design and maintain. It is then ideal to refine the monitoring-analysis-planning-execution (MAPE) feedback loop into one that can handle the modularization of the system, by applying the Analysis-Planning pair on each module.

One natural idea for the above modularization is to refine the MAPE loop into a set of subsystems with simpler MAPE loops, and coordination among them. Since the Monitor-Execute pair is the same for all the MAPE loops, we could represent a subsystem in terms of Analysis-Plan pairs, and use a new form for composing simpler MAPE loops, the M(AP)$^+$E loop form, which allows multiple Analysis and Planning pairs for one Monitoring and one Execution.

However, adopting this approach to modularize the MAPE loop induces two new difficulties. The first one is the propagation of the changes applied in one subsystem towards the other subsystems. Secondly, when subsystems are sharing information, if both of them want to change a shared data at the same time, a conflict may arise. Those two issues are addressed in this thesis, thanks to bidirectional transformations (BX) [16, 10, 15] and a rule-based synchronizer [33].

BX is a new technique for change propagation in a consistent manner, being able to (i) properly divide a large system called the *source*, into smaller subsystems called the *views*, and (ii) guarantee that any change in one of the subsystems is passed accurately in the rest of a large system. To achieve that, each BX supports two transformations: the *forward transformation*, known as the *get* function, and the *backward transformation*, known as the *put* function. The first one, get, can return the view as output if the source was given as input. The second one, put, can return an updated source as output if the original source and the view were given as input. As we will see it in details in Section 1.1, this updated source is the result of the changes made in the view, back in the source. If it possesses several different views (ie. if the main system has several subsystems), the changes made inside one view are propagated to the source with the *backward transformation*. From that point, the updated source can spread the modifications to the other views thanks to their respective *forward transformation*, and the modularity of the system is then ensured.

The execution of the MAPE loop associated to the BXs can propagate the self-adaptive changes, but it does not solve the potential conflicts between concurrent data modifications. Conflict resolution has already been studied in depth. In this thesis, we do not propose a way

to resolve conflicts, but to avoid them. If the execution of the views was sequential, it could avoid all the conflicts. Indeed, if *View A* is executed before *View B* and they share some data, even if *View A* has updated the source, the planning of *View B* can overwrite the changes, and the *backward transformation* of *View B* will update the source accordingly. Thus, the views must be prioritized. Of course, executing the views always in the same sequential order would lose flexibility and adaptability. To remedy this situation, we designed and implemented a rule-based synchronizer that can automatically find, according to predefined rules and at each iteration, the best order in which the views must be executed, transforming the model from "prioritized" to "self-prioritized". The "best" ordering is defined as the order in which the views should be executed according to the current context and the predefined rules related to it, without breaking any of them.

To evaluate those ideas, they have been brought together into a practical example. The purpose of the created system is to handle a collection of servers, which are available through the cloud. The cloud is a paradigm that allows access to resources and services, over the Internet. In this case, the cloud provides us resources, which are the servers our system needs to manage. It means that those servers are not reachable physically, but the services they provide are available thanks to the Internet. Compared to real servers lease, when the physical structure is rented and the use is personal, servers providers like Amazon give their servers to several people. Their clients are paying not for a physical server, but for a power of calculus and a storage space. The use of those servers is then shared amongst all their clients.

To manage those servers, our system requires as much information as possible about them. Most of the time, resources on the cloud provide a way to access specific data. In our case, the system uses the servers available on Amazon, thanks to Amazon Web Services (AWS). AWS possess a complete API with which it is possible to communicate information about the servers (an API is an interface supplying some tools to access the resources). Once the system has all the needed data, we modularize it into four different subsystems. The purpose of each of them is related to a specific quality [12]: the security, the modifiability, the availability and the performance. The division of the system according to qualities highlights the benefits of modularization, by restraining their access solely to the information they need. A subsystem must change the data, following its own strategy, to respect the constraints of the quality it represents. For example, the subsystem standing for the availability has to ensure that the system is reachable all the time, whereas the subsystem representing the security handles the access policies of the servers. Clearly, those two subsystems do not need to access the data of the whole system. Actually, they do not need to access the same data at all. This is where modularization shows all its usefulness.

The four modules are created, from the main system, thanks to bidirectional transformations (BXs). The changes they will apply on the data will also be propagated thanks to BXs. To address the conflict issue, when two (or more) subsystems are changing information they share at the same time, we propose an implementation of the rule-based synchronizer. Our example of implementation is quite simple. According to the current context (subsubsection 4.2.1.2) and to predefined rules (subsubsection 4.2.1.3), our algorithm orders the subsystems. The context represents internal information, like the number of servers currently running, and external information, like the hour of the day. The rules are based on those elements, and give specific constraints on the order to be found by the algorithm. Executed sequentially in the chosen order, the subsystems avoid conflicts between them when they apply their respective changes.

Once all the modules have been executed, and the changes have been propagated with the BXs on the main system, the updated data are sent back to the AWS API. Amazon is configured to automatically adjust its servers according to those data. Thanks to the *status*

concept, if the status is requested by our system to shut a specific server down, Amazon will shut it down. It works similarly when a server needs to be started.

As the implementation of our example is operable, experiments have been executed on the architecture. A part of the results are detailed in the chapter related to experimentations, but the whole set of graphics and collected data are readable in the appendices.

The main technical contributions of this thesis are threefold. First, it provides a novel way to propagate properly the changes inside a modular system with several subsystems. Secondly, it gives a new approach to avoid the conflicts when shared data change inside different subsystems. Thirdly, it presents a framework able to respond accurately to unforeseen situations and giving the best answer to solve them. We have created a practical model, using those three contributions. This thesis aims to prove the usefulness of the model, and its performances with the used technologies. Moreover, it wants to demonstrate the efficiency of BiGUL while showing one of its practical uses. For the reader to concretely see the model, the code is available (and runnable, following the ReadMe) at this address: `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations`.

The work is presented as follows: Chapter 1 introduces bidirectional transformations. Chapter 2 details the other notions of the model, the MAPE loop template and the rule-based synchronizer. Chapter 3 is devoted to the $M(AP)^+E$ loop, which solves the propagation issue. Chapter 4 explains in details the synchronizer, which avoids the conflicts between the modules. Finally, Chapter 5 tries to prove the efficiency and the performances of the model, thanks to several experimentations.

Chapter 1

State of the art

Before presenting our work, we first introduce the concepts on which we base our work, among which the language used throughout this thesis. The first section is thus a brief explanation of bidirectional transformations (BXs). Then is presented BiGUL (pronounced "Beagle") [24], the bidirectional programming language used in this thesis.

1.1 Bidirectional Transformations

BX is a new technique for change propagation in a consistent manner, being able to (i) properly divide a large system called the source, into smaller subsystems called the views, and (ii) guarantee that any change in one of the subsystems is passed accurately in the rest of a large system without any unexpected side-effect. To achieve that, each BX supports two transformations: the forward transformation, known as the *get* function, and the backward transformation, known as the *put* function. The first one, *get*, can return the view as output if the source was given as input. The second one, *put*, can return an updated source as output if the original source and the view were given as input. This updated source is the result of the changes made in the view, back in the source. If it possesses several different views (ie. if the main system has several subsystems), the changes made inside one view are propagated to the source with the backward transformation. From that point, the updated source can spread the modifications to the other views thanks to their respective forward transformation, and the modularity of the system is then ensured.

Let us imagine, as a teacher, we have to manage information about a group of students. However, we do not need all the available information about those students. Ideally, we should access only the information we need. Thus, we have the *source*, which is the entire information available, and a *view*, which is only the information we need. Both share information, but the *source* has information that the *view* does not need. As a teacher, we might change information about students. Those changes should not only change for our view, but also in the source. BXs guarantee that if something has changed in the view, the source is updated accordingly.

A lot of different areas are interested in using BXs, and researchers are actively working on combining their domain with BXs. Amongst those areas, we can mention:

- "Model-Driven Software Development: to compute and synchronize views of software models";
- "Graphical User Interfaces: to maintain the consistency of a GUI and the underlying application model in the model-view-controller paradigm";

- "Visualization With Direct Manipulation: to visualize abstract data and animate algorithms";
- "Relational Databases: to construct updatable views";
- "Data Transformation, Integration, and Exchange: to map data across paradigms, merge it from multiple sources, and exchange it between sources";
- "Data Synchronizers: to bridge the gap between replicas in different for- mats";
- "Macro Systems: to give feedback to the programmer (e.g., from a type checker or a debugger) in terms of the original program elements prior to macro expansion";
- "Domain-Specific Languages (DSLs): to translate between run-time values of the object language (the DSL) and the corresponding values of the host language in embedded interpreters";
- "Structure Editors: to provide convenient interfaces for editing complicated data sources";
- "Serializers: to mediate between external data (binary or sequential data representations on the wire or the file system) and structured objects in memory" [10].

To support BXs, programming languages have already been developed. The most popular is *Query/Views/Transformation (QVT)*. QVT is a standard that specifies model transformations, and is very popular since model transformations are used to manipulate models [26]. Although it was not created for BXs, it can be used to define BXs between models [31]. Another language is the *Janus Transformation Language (JTL)*. Unlike QVT, JTL has specifically been created to support non-bijective transformations and change propagation between models [8].

Like in many other fields, BXs could be specified in a wrong way, resulting in transformations that are not inverses of each other. *Lenses* solve this issue. A *lens* is a bidirectional program that satisfies two properties [15] such a transformation is called "well-behaved". Those properties are:

$$put\ s\ (get\ s) = s \qquad \textsc{GetPut}$$
$$get\ (put\ s\ v) = v \qquad \textsc{PutGet}$$

Here, the s is the source, and the v is its associated view. As a reminder, the *get* transformation returns a view when a source is given; the *put* transformation returns an updated source when a source and an updated view are given. If a program satisfies the GetPut and PutGet properties, then it is "well-behaved", and thus called a *lens*. Let us look more precisely on what those properties mean. The GetPut property enforces that if the view (recovered with the forward transformation) was not changed, the backward transformation will give exactly the same source as the original one. The PutGet property imposes that any change in the view has to be correctly reflected in the updated source. It is then possible to recover exactly the same view by applying the *get* function on the updated source [21].

As explained earlier, many programming languages already help the developer to write well-behaved bidirectional transformations. However, all of them are *get-based* languages [20]. It means that the developer writes the *get* function, and the *put* comes for free. It is important to spotlight that for each *get*, many *puts* can be derived. The programmer

cannot choose which *put* is automatically created, and has to formally prove that the chosen *put* satisfies the GetPut and PutGet properties.

The *put-based* languages are the exact opposite. The developer has to write the *put* function, then the *get* comes for free. The main difference between those languages relies on the automatic derivation: for each *get* function, many possible *put* functions could be chosen, whereas at most one *get* function can be derived from a given *put*. A program that implements only the *put* direction and that compiles in *put-based* languages can directly be called a *lens* because it automatically satisfies the GetPut and PutGet properties. It is also possible to implement both the *put* and the *get* in put-based languages. However, it is a last resort: it forces to formally prove that the bidirectional transformation is well-behaved, and removes the advantage of using a put-based language. In fact, it is exactly what the programs coded with the *get-based* languages must do. To be called *lenses*, they must formally demonstrate that the *put*, which has been derived from the *get*, is correct and satisfies the properties. There is no way to control which *put* will be generated from a given *get*. Using put-based bidirectional languages, yet more complicated to code, is a much more reliable way to assure that the needed transformation is "well-behaved".

In this thesis, as the BXs aggregate the information contained in a large system into smaller subsystems and update the large system with potential changes occurring in the subsystems, the focus is upon the *lenses*. Moreover, the BXs are implemented using BiGUL, the only existing *put-based* language [24], and illustrated in the next section.

1.2 BiGUL

As the following pages will contain pieces of code, an explanation of BiGUL's syntax is mandatory. This section will first introduce a simple BiGUL signature, and then show some patterns often used in other chapters.

1.2.1 A simple BiGUL signature

A BiGUL function typically has a signature like this:

$$simpleSignature :: BiGUL\ Source\ View \tag{1.1}$$

Following 1.1, with the *get*, the input will be the Source, and the output will be the View. But with the *put* function, the input will be the old Source and an updated View, and the output will be the updated Source. Moreover, as BiGUL is a *put-based* language, the function only defines the backward transformation. Regarding the forward transformation, it is automatically derived by the language.

1.2.2 Useful BiGUL patterns

This section introduces three particular patterns frequently used in the code of this thesis.

1.2.2.1 Rearrangement

The first pattern is the rearrangement. It is very helpful when the structure of the source and the view are different. But first, here is an explanation of the *Prod* BX constructor,

which is used in the example below. It is a good practice to use this BiGUL function when both the source and the view are pairs. The signature of *Prod* is:

$$Prod :: BiGUL\ s1\ v1 -> BiGUL\ s2\ v2 -> BiGUL\ (s1, s2)\ (v1, v2) \tag{1.2}$$

If the source is *(s1, s2)* and the view is *(v1,v2)*, the corresponding BiGUL program is: *BiGULFunction1 'Prod' BiGULFunction2*. The first argument of *Prod* will thus be applied to *s1* and *v1*, and its second argument will link *s2* and *v2*.

Now for the rearrangement, let us imagine that the source contains three attributes of a car (the name of the manufacturer, the number of doors and the color), but the view only needs two of them (the name of the manufacturer and the color). In that case, a simple BiGUL program could look like this:

```
1 simpleRearr :: BiGUL (String, (Int, String)) (String,String)
2 simpleRearr =
3     $(rearrV [| \(v1,v2) -> (v1,((),v2)) |])$
4     Replace 'Prod' (Skip (const ()) 'Prod' Replace)
```

Listing 1.1: Sample code for Rearrangement

In this example, the source contains three pieces of information and the view contains only two. At the third line, the view is rearranged from a pair to another pair, where the first element of the second part does not matter. If the source needed to be rearranged, the same structure could be applied using *rearrS*. BiGUL imposes to have the same number of arguments on each side of the rearrangement. As *v1* and *v2* were defined on the left side of the anonymous function, both had to be on the right part as well. After making the structure of the source and the view identical, it is easy to replace each element with its associated element. Line 4, the first element is replaced. Then, the first element of the second part must be skipped because the view does not need it. Finally, the last element is replaced.

1.2.2.2 Case and Branches

The second pattern is the use of branches inside a Case. There are four different kinds of branches, represented in this example:

```
1  data Complete = Letter Char Complete | ComNull deriving Show
2
3  deriveBiGULGeneric ''Complete
4
5  pTransform :: BiGUL Complete [Char]
6  pTransform = Case [
7      $(normalSV [p| ComNull |] [p| [] |] [p| ComNull |])
8          ==> $(update [p| ComNull |] [p| [] |] [d||]),
9      $(normalSV [p| Letter _ ComNull |] [p| [_] |] [p| Letter _ ComNull |])
10         ==> $(update
11                 [p| Letter c ComNull |]
12                 [p| [c] |]
13                 [d| c = Replace |]),
14     $(normal [| \(Letter _ _) v -> length v > 1 |] [| \s -> True |])
15         ==> $(update
16                 [p| Letter c cs |]
17                 [p| (c:cs) |]
18                 [d| c = Replace; cs = pTransform |]),
```

```
19      $(adaptiveSV [p| _ |] [p| _:_ |])
20          ==> \_ v -> charToCompl v,
21      $(adaptive [| \s v -> length v == 0 |])
22          ==> \_ _ -> ComNull]
23          where
24              charToCompl :: [Char] -> Complete
25              charToCompl [] = ComNull
26              charToCompl (v:vs) = Letter v (charToCompl vs)
```

Listing 1.2: Sample code for Case and Branches

A possible source of the program could be "*Letter 'b' (Letter 'a' (Letter 'c' ComNull))*", and its corresponding view would be "*bac*". The third line, *deriveBiGULGeneric*, informs BiGUL that a new Haskell data type has been created. The normal branches handle the happy cases when everything goes well, whereas the adaptive ones take care of all the others. Let us have a closer look at each branch.

The first one, line 7, is a *normalSV*. It means that the condition is made upon both the source and the view by pattern matching, respectively the first and the second brackets. The third brackets is the exit condition, generally the same condition as the source one. Its major purpose is to optimize the *put* transformation [21]. If this case is matched, the source and the view are empty, and so the update function does not have to do anything. In the example, the source would be "ComNull" and the view " ".

The second branch is also a *normalSV*, and handles the basic case where the source and the view contain only one element. In our example, it corresponds to "*Letter 'c' ComNull*" and "*c*". Thus, the update has to create the bridge between these two elements. Once this branch has been taken, the update function is called, and the pattern matching can be made in both the source and the view. It is represented by the variable *c* (for character), used inside the Replace BiGUL function.

The last normal case, line 14, is represented with a *normal* branch. Here, the condition is again upon both the source and the view, but is defined with an anonymous function to handle conditions more complicated than pattern matching. The second brackets represent the exit condition. This case pattern matches when there is still at least one element in the source, and more than one in the view. In such case, the first element is replaced, and the *pTransform* function is applied recursively on the rest of the lists.

The last branches are adaptive ones. Their purpose is to adapt the source when the source and the view have a different number of elements.

The first adaptive branch is an *adaptiveSV*, and defines its condition with a pattern matching on both the source and the view. If no match has been made before and the program arrives here, it means that either the source is empty, but not the view, or the view is empty, but not the source. This adaptive branch takes care of the empty source case. The function *charToCompl* will thus convert the rest of the characters inside the view into a *Complete* data type in the source.

Finally, the last branch, line 21, is an *adaptive* one. Just as the *normal*, it makes its condition with an anonymous function. The last possibility is when the source is not empty, but the view is. In other words, this case happens when one or more elements have been deleted in the view. As the view must update the old source into an updated source, if the view is empty, the source has to be empty as well. Thus, the program sends an empty *Complete* data type.

1.2.2.3 The align function

The third and last pattern explained in this thesis is *align*, a function provided by BiGUL. This function is very often used in our code. It consists of the alignment of two lists, the source and the view. To align those lists, the BXs have to synchronize each element between them. It must be able to insert, delete or update elements when the synchronization is applied. For example, the source could be a list of people, each element containing information like the name, the national number ID, the number of children, etc. The view could represent those people, but for a particular service which needs only a part of the information. In that case, the *align* fonction is used to synchronize the two lists.

This alignment issue is already explained and detailed by Barbosa et al.'s matching lens [4]. There are several alignment strategies but the following explanation will focus only on the one already implemented in BiGUL.

To fully understand this function, the explanation is divided in three parts. Firstly, the Haskell signature of align is explained to show the needs of the function. Secondly, the entire implementation is illustrated. Thirdly, a little example using align is described.

```
1 align :: forall s v. (Show s, Show v)
2     => (s -> Bool) -- view condition
3     -> (s -> v -> Bool) -- matching condition
4     -> BiGUL s v -- update function
5     -> (v -> s) -- creation function
6     -> (s -> Maybe s) -- deletion function
7     -> BiGUL [s] [v]
```

Listing 1.3: Signature of align

The align function needs 5 arguments to work properly. The first one is the view condition. This is the condition on a element of the source to determine if it must be represented in the view. In the example of people, it could filter all the people with less than 1 child. In the view, only the people with children would be represented. It is logical that the function needed for this argument takes s (ie. an element of the source) as input, and returns a boolean as output.

The second argument is the matching condition. It is a one-to-one relation which links the elements in the source and in the view. In our example, this condition could be the national number ID. The function for this argument needs s, an element of the source, and v, an element of the view, as input. It returns a boolean (true if those elements match).

The third argument is the BiGUL *update* function on elements. This function gives the way to synchronize matched elements of the source and the view. As the explanation about the implementation of the align function will show, the update function is called only on the elements that satisfy the matching condition (ie. the second argument). In the example of lists of people, the *update* function gives the information needed by the view, and tells which information must be removed from the view.

The fourth argument is the creation function. It describes how to create an element in the source when a complete new element is found in the view. That is why the function takes an element of the view (v) and produces an element of the source (s). In our example, if the view has created a new person, this person must be represented in the source as well. The function describes how to derive a person in the source from the same person in the view.

The last argument is the deletion. This function is triggered when an element is found in the old source but is not in the view anymore. Returning a *Maybe*, it allows the

programmer to choose between the complete deletion from the source (the function returns *Nothing*), or a soft deletion (the function returns an updated element that does no longer satisfy the source condition (ie. first argument of the *align* function), and thus will not be represented in the view anymore).

```
1 align p match b create conceal = Case [
2     $(normalSV [p| [] |] [p| [] |] [p| [] |])
3         ==> $(update [p| _ |] [p| [] |] [d| |]),
4     $(normal [| \(s:_) (v:_) -> p s && match s v |] [| \(s:_) -> p s |])
5         ==> $(update
6                 [p| x:xs |]
7                 [p| x:xs |]
8                 [d| x = b; xs = align p match b create conceal |]),
```

Listing 1.4: Implementation of align - normalSV and normal

The align function uses the pattern *Case and Branches* already highlighted. It is divided in six branches. The first is the classic one (line 2 in Listing 1.4). When the source and the view are both empty, the function does nothing.

The second case is a little bit more complicated (line 4 in Listing 1.4). This case is the one that handles the match between the source and the view. In the condition of this branch, the p is the predicate to determine if the source element is not deleted and the match function checks if the source and the view are linked. In English, this condition can be translated as "if the element of the source is not deleted and both the elements are linked". Note that the *normal* branch is used and not the *normalSV*, because we have to make a condition on both the source and the view simultaneously. When this case is triggered, the function makes a simple update where the current element is transformed, with the function b given by the programmer, showing the way from the source to the view and vice versa. The align function is called recursively with the remaining elements.

```
1     $(adaptive [| \(s:_) [] -> p s |])
2         ==> \ss _ ->
3                 let (prefix, remaining) = span p ss
4                     in catMaybes (map conceal prefix) ++ remaining,
5     $(normal [| \(s:_) _ -> not (p s) |] [| \(s:_) -> not (p s) |])
6         ==> $(update
7                 [p| _:xs |]
8                 [p| xs |]
9                 [d| xs = align p match b create conceal |]),
```

Listing 1.5: Implementation of align - adaptive and normal

The third branch is an adaptive one (line 1 in Listing 1.5). It catches the situation when the source contains non deleted elements and the view is empty. In this case the function has to delete the remaining elements in the source. To make this deletion, the *span* function is used to separate the non deleted elements (*prefix*) and the others (*remaining*). Thereafter, the elements in *prefix* are deleted and concatenated with the remaining list.

The fourth branch (line 5 in Listing 1.5) is the normal case that handles the situation where the current element is considered as deleted. The function simply skips this element and calls recursively the align function with the remaining elements.

```
1     $(adaptive [| \ss (v:_) -> isJust (findFirstMatch v ss) |])
2         ==> \ss (v:_) -> uncurry (:) (fromJust (findFirstMatch v ss)),
3     $(adaptiveSV [p| _ |] [| \(v:_) -> p (create v) |])
4         ==> \ss (v:_) -> create v : ss
```

```
5       ]
6           where
7               findFirstMatch :: v -> [s] -> Maybe (s, [s])
8               findFirstMatch v [] = Nothing
9               findFirstMatch v (s:ss) | p s && match s v = Just (s, ss)
10                                      | otherwise = do
11                                          (s', ss') <- findFirstMatch v ss
12                                          return (s', s:ss')
```

Listing 1.6: Implementation of align - adaptive and adaptiveSV

The fifth branch is the case when the view and the source are not linked (line 1 in Listing 1.6). The function has to find the element in the source linked with the current element in the view. To do this task, the align function uses the *findFirstMatch* function. It retrieves a pair where the first part contains the first occurrence that match with the current view, and the second part contains the remaining elements. Then, it transforms this pair into a list with the first element linked to the view.

The sixth and last branch (line 3 in Listing 1.6) creates an element in the source from the view if there is not any element in the source linked with the current element of the view. Even if this case had not any condition, as the last case, it works like a Haskell *otherwise*.

To illustrate *align*, the implementation below shows an example using it.

```
1   type Identifier = Int
    type Name = String
    type Deleted = Bool

5   type Source = (Identifier, (Name, Deleted))
    type View = (Identifier, Name)

    myAlign :: BiGUL [Source] [View]
    myAlign = align
10      -- the view condition
        (\(_,(_, deleted)) -> not deleted)
        -- the matching condition
        (\(idS,(_, _)) (idV,_) -> idS == idV)
        -- the update function
15      $(update
            [p|(identifier, (name, _))|]
            [p|(identifier, name)|]
            [d|identifier = Replace; name = Replace|])
        -- creates an item in the source from an item in the view
20      (\(identifier, name) -> (identifier, (name, False)))
        -- how to delete an item from the source when it is not in the view
        (\(identifier, (name, _)) -> Just (identifier, (name, True)))
```

Listing 1.7: Example using align

The source is a list of tuples containing an identifier, a name and a status (*deleted*) and the view is a list of pairs containing an identifier and a name.

The function *myAlign* calls the align with some parameters. The first one is the condition on the source to know which element must be considered in the view. In this example, an element is considered if its status (*deleted*) is equals to *False*. The second parameter is the matching condition. This condition is true when the identifier in the source is equal to the identifier in the view. The third parameter is the update function,

that describes how to transform an element in the source into an element in the view and inversely. In our example, the view aggregates the source by removing information (the status). The two other elements, the identifier and the name, must be replaced. The fourth parameter gives the way to create a new element in the source from the current element in the view. In this example, it keeps the identifier and the name, and it assigns the status to *False* by default. The last parameter describes how to delete an element. In the example, the function puts the status (*deleted*) to False. Thus, it will not be represented in the view anymore, because it will not match again the view condition (ie. the first argument of *align*).

Bidirectional transformations has been introduced. The next chapter highlights the other notions used in our thesis: the MAPE loop template and the rule-based synchronizer.

Chapter 2

Basic preliminary notions

Here are presented two other notions: the MAPE loop template and the rule-based synchronizer, which both work with the bidirectional transformations to form our model. Then, the idea behind an unpublished paper called *Modular Hierarchical Self-Adaptation using Bidirectional Transformations* [29] is highlighted. As a matter of fact, our thesis uses this unpublished paper as foundations, but our model describes a much more complex and extensive system.

2.1 The MAPE loop template

The MAPE loop template is a well-known and often used concept, especially as a control model for autonomic and self-adaptive systems [2, 11, 23, 35]. Its purpose is to allow a system to independently make decisions and automatically adapt itself, according to a changing environment. As a matter of fact, as time goes by, the complexity of the management of the computing systems significantly increases, and the manual control of those systems becomes more and more complicated. This is where autonomic computing takes action [22], and more precisely the MAPE loop template.

Divided into four steps, it facilitates the structure creation of adaptive systems. Fanzhang Li specifies those steps as follows:

1. "Monitor provides the mechanisms that collect, aggregate, filter, manage, and report details (for example, metrics and topologies etc) which are collected from a managed resource to the Analyzer.";
2. "Analyzer takes charge of analyzing the information transmitted from Monitor and correlating and modeling complex situation.";
3. "Planner provides the mechanisms to structure the action needed to control the behavior of the managed resource.";
4. "Executer executes the action structured by Planner to control the part of managed resource" [37].

All together, those four steps are able to examine a system and make decisions according to the evolving environment.

2.2 The rule-based synchronizer

As explained earlier, the rule-based synchronizer has the goal to find an execution order of the views at each iteration of the system. There are a lot of possible representations for

a knowledge base (logical, philosophical, computational, etc) [30]. Thus, finding a way to put human knowledge into a program has been studied for a long time [7]. Amongst all the existing techniques, the rule-based representation is an established approach [33] that offers a clear and easy way for the user to define accurately its expertise. The fact that a rule can associate a boolean condition (the trigger of the action) with an expected result (the action itself) is a renowned way to describe human knowledge. At a particular time, the boolean conditions of all the rules are evaluated. Only the rules whose boolean condition is *True* are taken into account in the knowledge base. The expected result of those rules can then be used to apply what a human would have done in the same situation, but in a much more efficient way.

2.3 Modular Hierarchical Self-Adaptation using Bidirectional Transformations

To let the reader understand this thesis, an explanation of another paper is required. The base of our subject relies on an idea described on an unpublished paper called *Modular Hierarchical Self-Adaptation using Bidirectional Transformations* [29]. The authors want to solve a widespread issue: nowadays, most of the systems are very complex. To solve this, the use of hierarchical modules, resulting from the decomposition of the big system, is a time-honored strategy. The paper proposes an approach based on the bidirectional transformations [24] and the *MAPE loop*, illustrated in Figure 2.1.

The monitoring generates the source from the AWS API. This is an Infrastructure as a Service (IaaS), a model of cloud computing which lets the user choose an amount of computing resources and automatically scales up or down those resources to meet the demand. It allows to manage a set of servers (in this case, owned by Amazon). The clients of those services can thus rent some servers and run their own computer applications [5]. Then, each view is created with a *forward transformation* (get) and runs its own analysis and planning steps. Those steps apply changes, which are send back into the source with their *backward transformation* (put). The execute step finally sends the results back to the API [29]. Our interest in this paper is the upper side, with the bidirectional transformations and the analysis and planning steps. The exchanges (ie. the monitoring and execute steps) between the AWS API and the cloud infrastructure configuration model (ie. the main source) are mostly present to test the performances of the upper part. This model contains some substantial disadvantages, explained in the next section.

2.4 From hierarchical to self-prioritized adaptation

From the previous paper [29] and its drawbacks, some ideas came up to improve the whole model and to transform it into an extensive and more significant system. Two main disadvantages are highlighted here.

Firstly, each view represents a specific quality attribute of the system. The purpose of those qualities is to ensure that the system follows a number of policies, each of them related to a specific quality. The main qualities of a system are usually the Availability, the Modifiability, the Performance, the Security, the Testability and the Usability [12]. In the actual case, the goal of the views is to represent a quality attribute. In that respect, the autoscaling view is related to the performance, the redundancy to the availability, and the firewall rules to the security. A set of three qualities is too light to represent a concrete example. Our goal is to show the use of BXs in a realistic example, and one of the most important concern of companies is the resulting cost of a system. To be more

Figure 2.1: M(AP)$^+$E loop [29]

accurate with a real situation, the modifiability concern, which allows the decrease (and the increase) of the costs, has to be implemented. Besides, the modifiability concern will enter in conflict with the redundancy and availability concerns, which will show in a more efficient way how our approach handles those conflicts, and why ordering them is a real asset.

Secondly, the order in which the views are executed is hard-coded and never changes. The main interest in this order relies on the priority of a view amongst the others. It means that the first view executed is the less important, because if it changes something in the source, the second executed view can overwrite those changes, so can the third, and so on. Thanks to this order, even if the views are sharing the same data, there will not be any conflicts anymore. As each view can erase the changes made by the previous views, the last one will always have the prerogative to enforce its own changes. However, it should be possible to change the order of the views dynamically, according to the inner and outer context of the system. The inner context covers all the data inside the source, such as the number of running servers, whereas the outer context contains potentially every other data, such as the actual hour of the day. Indeed, a concern could take the advantage in one situation, but not in another one. For instance, the Availability concern could be more important than the Cost during the day, but not during the night in order to save money. Therefore, this thesis proposes a rule-based approach that can automatically change, at each iteration of the system, the order in which the views are executed. This improvement transforms the hierarchical adaptation model into a self-prioritized adaptation model:

The shape of Figure 2.2 is directly related to the Three Layer Architecture Model for Self-Management described by Kramer and Magee [25], which is based itself on an architecture defined by Gat for robotic systems [19]. Just as described in those two architectures, our model is mainly divided into three layers: the infrastructure, the BXs, and the synchronizer. Kramer and Magee named the layers as follow:

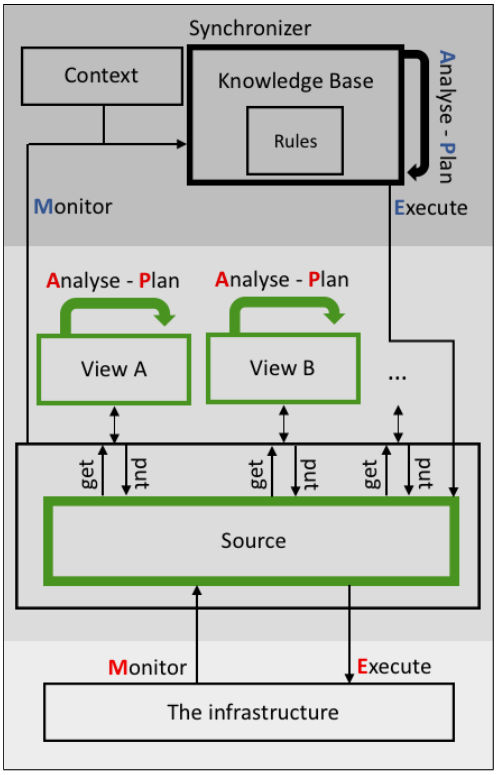


Figure 2.2: Self-Prioritized Modular Adaptations using BXs - Theoritical

1. Component Control : "It consists of sensors, actuators and control loops. The bottom layer of a self-managed system consists of the set of interconnected components that accomplish the application function of the system".

2. Change Management : "In a self-managed system, this layer is responsible for effecting changes to the underlying component architecture in response to new states reported by that layer or in response to new objectives required of the system introduced from the layer above".

3. Goal Management : "This layer produces change management plans in response to requests from the layer below and in response to the introduction of new goals" [25].

As a matter of fact, the infrastructure in figure 2.2 is our Component Control: it contains components that can be monitored in order to provide information for the next layer. Then, our Change Management is the Source and its views. Each view can change the data according to plans, and the BXs propagate those changes back in the Source. Finally, our Goal Management is the synchronizer. Even though it is based on rules and not on goals, its purpose is identical: it produces a change in the management of the previous layer (ie. the source and its BXs), by applying a new execution order of the views when a new context is met.

Thus, figure 2.2 shows how the three notions explained, from Section 1.1 to Section 2.2, are brought together. The main idea is a combination of two MAPE loops. The first one, in red in Figure 2.2 and refined as a M(AP)$^+$E loop, must propagates the changes, performed inside one view, to the others thanks to the analysis and planning steps located on each view. The BXs are inside this loop as well, and have the aim of associating data inside the source (ie. the large system) with the data inside the several views (ie. the subsystems). The second loop is the synchronizer, that must find the best order in which the views are executed. As a reminder, if the views are executed sequentially, the potential

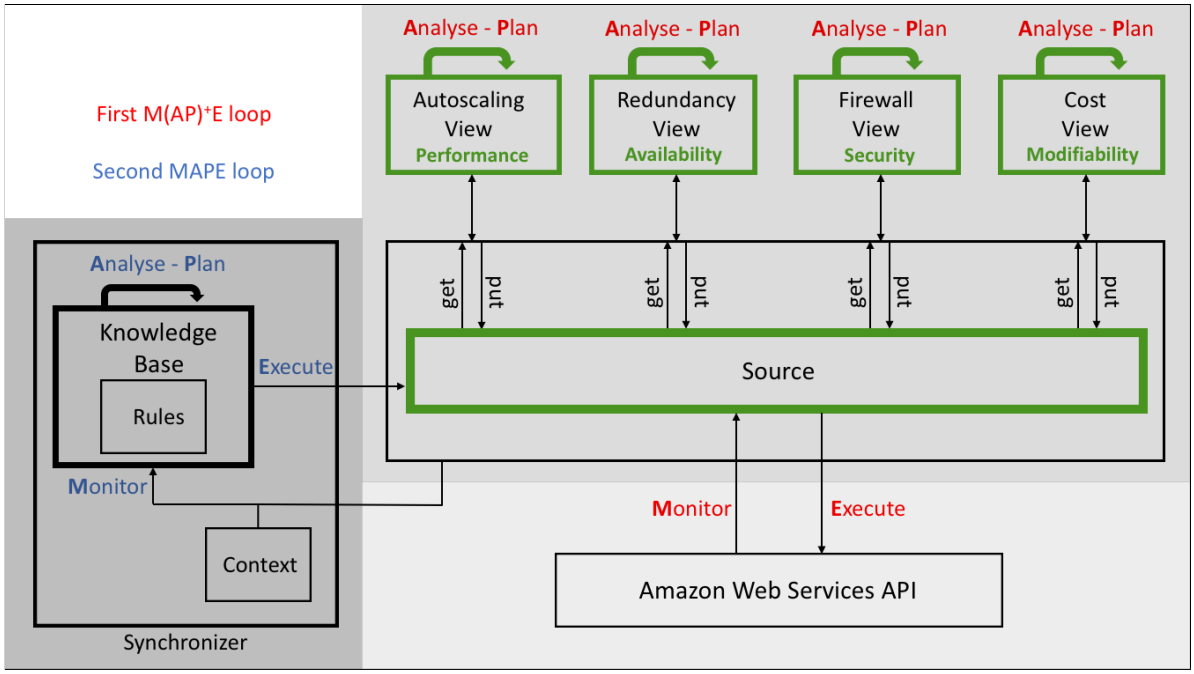


Figure 2.3: Self-Prioritized Modular Adaptations using BXs - Practical

conflicts between them are avoided. Indeed, each view is then able to overwrite the changes of previously executed views. To find the best execution order of the views, the second loop, in blue in Figure 2.2, contains a rule-based knowledge base. The boolean conditions of those rules reference the current context of the infrastructure.

Obviously, figure 2.2 is not a concrete example. A real implementation of our model, which is evaluated in this thesis, is the illustration of Figure 2.3. The next two chapters explain in details how the two loops are working together to form the self-prioritized modular adaptations system.

Chapter 3

First MAPE loop : Modular Adaptations using Bidirectional Transformations

The main idea behind this thesis is a combination of two *MAPE loops*. For a better clarity, the combination of the loops is illustrated in a concrete example throughout this thesis. The chosen example is illustrated in Figure 2.3, with each of the two loops represented with a different color.

The first one, quite similar to the one explained earlier in Montrieux's paper, will *monitor* the AWS API to create the source. This source represents the large and complex system that must be handled and decomposed into four modules. Those modules can be synchronized with the source thanks to the bidirectional transformations. For each one, the forward transformation (ie. *get*) selects the data needed by the view (ie. the module) in the source. With those data, each view can *analyse* them and *plan* to do some changes. Once the changes have been made, the backward transformation (ie. the *put*) updates the source with the data modified by the view. When all the views have finished their changes, the system will *execute* the update to the AWS API. This first loop handles the *modular* part of the thesis. The *self-prioritized* part, where the priority of each view is set, is located in the second *MAPE loop*. The current chapter will focus on the first loop and how all its mechanisms work, and the second loop will be detailed in the next chapter. Moreover, the code of our system is runnable and can be downloaded here: `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations`. Before going into our solution, we are first introducing what has already been done by other studies, and why it does not fit our subject.

MAPE loops, serving as the basis for a lot of self-adaptive systems, turned up in a lot of research and they have been applied in many different shapes. Weyns et al. [35] listed diverse types of MAPE loops for distributed systems, like the master/slave pattern (ie. M^+APE^+), or the regional planner pattern (ie. MAP^+E). Each described MAPE loop pattern has its own favourable situation in which it is better to use it. However, the $M(AP)^+E$ pattern, used in this paper, does not appear in them.

Barna et al. [5] implemented Hogna, a self-adaptive solution using MAPE-K loop, to auto-scale web-applications on the cloud. Basically, Hogna is working like our model, but without the BXs. Moreover, it only handles one concern, the auto-scaling. As there is only one concern to handle, the need to divide the system into subsystems does not appear. In doing so, as it is impossible to add other concerns, it removes the opportunity to entirely and self-adaptively manage the servers taking all the concerns into account. Moreover, our

model is not closed to managing web-applications on the cloud. It can be adapted according to what the user wants to administer.

Garlan et al. have developed an architecture-based infrastructure for self-adaptation, called Rainbow. It provides "a framework which uses software architectures and a reusable infrastructure to support self-adaptation of software systems. The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns" [18]. This requires predefined strategies or architectural plans to work properly, whereas our model only needs configurations parameters of an IaaS (such as AWS), which is transparent to the users.

Although recent, the use of bidirectional programming in software engineering has been the subject of some research [28, 38]. However, most of the studies use get-based languages in contrast to a put-based language like BiGUL.

The decomposition of large systems into smaller ones has already been treated by several studies. Finkelstein et al. [14] have introduced good software engineering practices by treating multiple aspects as viewpoints. They highlighted the profit of modularization in managing the inconsistency. The real added-value of modularization in software engineering, Sullivan et al. [32] and Lopes et al. [27] demonstrate it mathematically. They based their work on a theory of modular design by Baldwin and Clark [3], with their Net Options Value, a quantitative approach to evaluate the benefits of modularization. Regarding the views themselves, Fradet et al. [17] introduced a formal approach based on typed graphs to verify the consistency of views, providing a simple yet powerful algorithm to check it. While all of those papers praise the benefits of modularization, none of them is concerned with the propagation issue, when multiple views are sharing the same data.

Obviously, a lot of work has already been done in the subjects studied in this thesis, but the added-value of our work lies on the composition of those subjects, and the model created when they are combined together.

3.1 Model of the first loop : M(AP)$^+$E

As shown in Figure 3.1, all the data used here come from the AWS API. It is an IaaS, which is a category of Cloud Computing. It means that Amazon takes care of the servers, the network and the storage whereas the client just has to handle the application software. This API provides a large range of data, and many of which are unnecessary in this study. Therefore, the *Monitoring* step of the loop serves not only to recover the data but also to select them. Then, each view may execute some computation inside the *Analysis* and *Planning* steps. As several *Analysis* and *Planning* steps occur between only one *Monitoring* and one *Execute*, the traditional *MAPE loop* becomes a *M(AP)$^+$E loop*. The behavior of the *Analysis* and *Planning* steps will be explained in a much more precise way in the following sections. Regarding the *Execute* step, it will simply transmit to the AWS API the changes made by the different views, thanks to Ansible[1]. Once the changes of the views have been made, the set of data is parsed into a YAML format, and sent to Ansible. It will take charge of updating the AWS API.

The following sections explains in details all the parts of the first loop, one by one.

[1] `https://www.ansible.com`

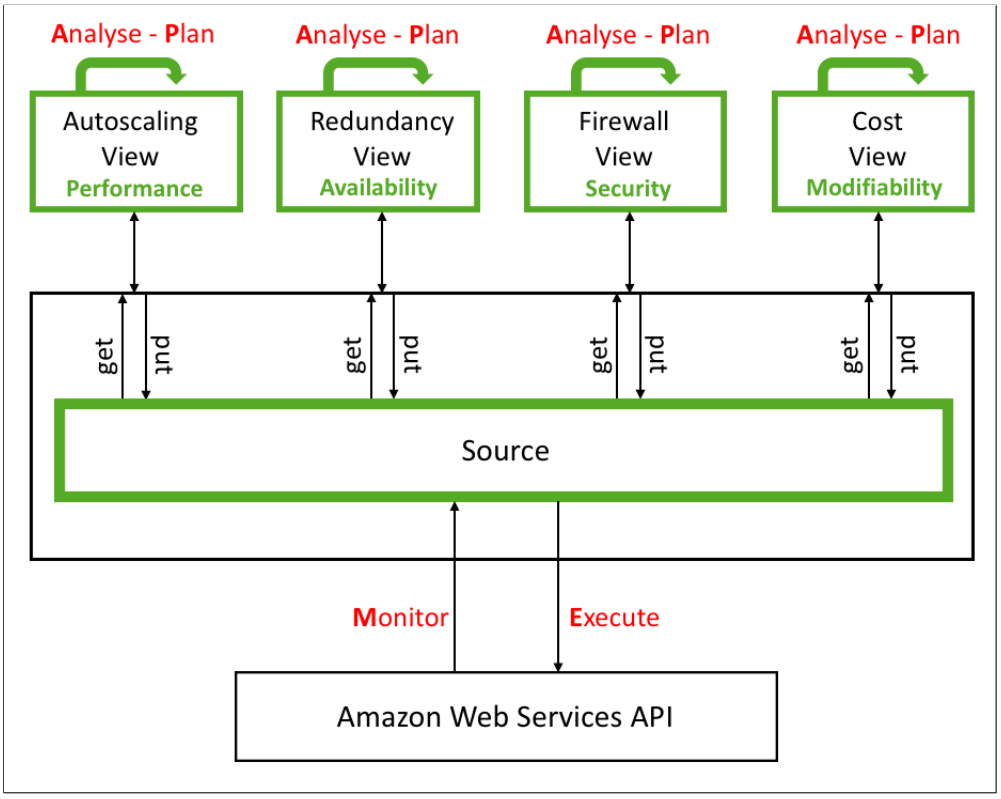


Figure 3.1: M(AP)+E loop

3.2 The Source

This portion of the thesis describes the data structure of the *Source*. This data structure is a derivation of the data sent by the AWS API, but defined in Haskell. The structure is shared by the source and its four views.

3.2.1 The Source - Data structure

The *Monitor* step recovers the data from the AWS API. Conceptually, the source looks like this:

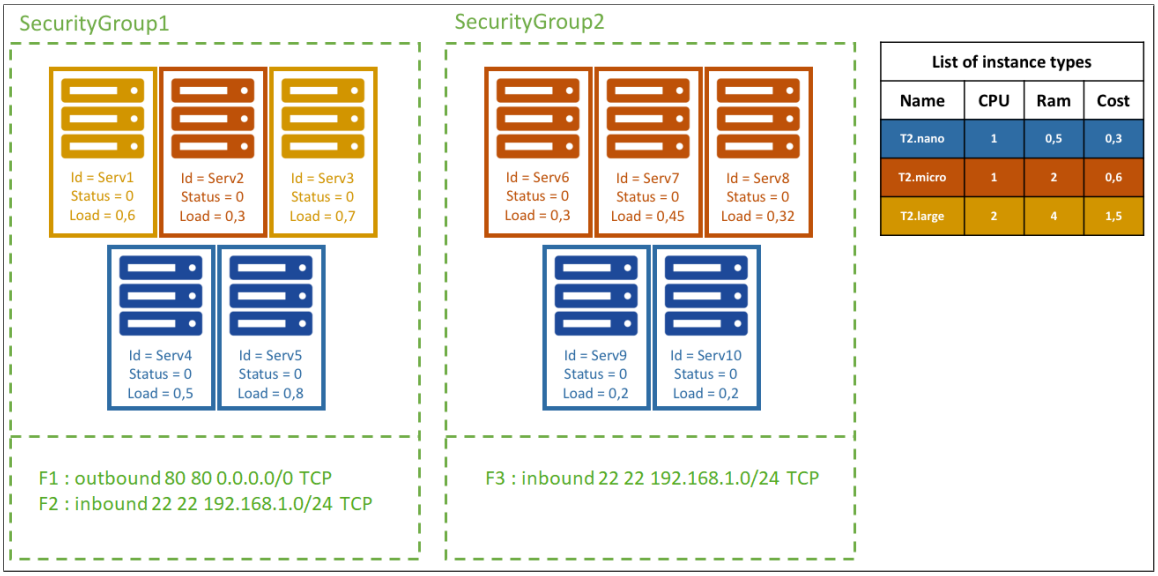


Figure 3.2: Example of a Source

A source possesses a set of *Instances*, which are the Amazon's virtual servers; a set of *Instance Types*, which identify the features of the servers; and a set of *Security Groups*, which specify the access policies of all the servers related to each security group. Moreover, each *Security Group* contains a set of *Firewall Rules*, which list the allowed and denied accesses. Together, these three sets can describe a concrete system, able to show how the modular self-prioritized approach, described in this thesis, works.

Figure 3.2 shows our implemented source. This source is made up of ten *Instances*, three *Instance Types* and two *Security Groups*, in which there are respectively two and one *Firewall Rules*.

Now that the schema has been explained, let us see the code behind the model:

```
1  type Id = String
2  type Description = String
3  type Type = String
4  type Ami = String
5  type SecurityGroupRef = Id
6  type InstanceRefs = [Id]
7  type FromPort = Maybe Int
8  type ToPort = Maybe Int
9  type Ip = String            -- Format: 0.0.0.0/0 for IPv4 or ::/0 for IPv6
10 type Protocol = String
11 type State = Int
12 type Status = Int
```

```
13 type TypeCPUs = Int
14 type FirewallStatus = Int
15 type Load = Double         -- Unit: %
16 type TypeRAM = Double      -- Unit: Gio
17 type TypeCost = Double     -- Unit: $/hour
18 type Outbound = Bool
19
20 data Instance = Instance Id Type Ami State Status SecurityGroupRef Load
21                 deriving (Show, Eq, Read)
22 deriving instance NFData Instance
23 deriveBiGULGeneric ''Instance
24 deriveJSON defaultOptions ''Instance
25
26 data FirewallRule = FirewallRule Outbound FromPort ToPort Ip Protocol
27                 deriving (Show, Eq, Read)
28 deriving instance NFData FirewallRule
29 deriveBiGULGeneric ''FirewallRule
30 deriveJSON defaultOptions ''FirewallRule
31
32 data SecurityGroup = SecurityGroup Id Description InstanceRefs [FirewallRule]
33                 deriving (Show, Read)
34 deriving instance NFData SecurityGroup
35 deriveBiGULGeneric ''SecurityGroup
36 deriveJSON defaultOptions ''SecurityGroup
37
38 data InstanceType = InstanceType Id TypeCPUs TypeRAM TypeCost
39                 deriving (Show, Eq, Read)
40 deriving instance NFData InstanceType
41 deriveBiGULGeneric ''InstanceType
42 deriveJSON defaultOptions ''InstanceType
43
44 data Source = Source [Instance] [SecurityGroup] [InstanceType]
45                 deriving (Show, Read)
46 deriving instance NFData Source
47 deriveBiGULGeneric ''Source
48 deriveJSON defaultOptions ''Source
```

Listing 3.1: Data structure of the Source

Each instance contains several values: an Id, a Type, an Ami, a State, a Status, a SecurityGroupRef and a Load. The Id is just the Amazon's identifier. The Type is a reference to the instance type, to recover the features of the server. The Amazon Machine Image (AMI) represents the virtual server in the cloud where the instance is launched[2]. Regarding the State, it shows if the instance is running or not. The Status contains an integer (0, 1 or 2), where 0 means that nothing has to be done with the instance, 1 that the instance must be created, and 2 that the instance must be terminated. SecurityGroupRef is a reference to a security group that defines the access policy of the server. Finally, the Load is a value between 0 and 1, representing the percentage of use of the server.

An instance type is composed of an identifier, the number of Central Processing Units (CPUs)'s inside this type of instances, the amount of Random Access Memory (RAM) and finally its cost.

A security group contains an identifier, a description (required by Amazon when the data is sent back to the API), the references to all the instances that must follow the access

[2] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html` (Accessed on 04/22/2018)

policy described by the security group, and finally a list of firewall rules, which define the policy.

A firewall rule possesses an "outbound" that shows if the current rule allows or denies the access, the interval of ports affected by the rule, an Internet Protocol (IP) address and a protocol (for example, http).

3.3 The Auto-scaling View

The aim of the auto-scaling view, representing the Performance concern, is to ensure that the load of the servers is balanced amongst all of them, and that it stays below a predefined limit. First we explain the data structure of this view, then the bidirectional transformation between the source and the view, and finally its analysis and planning steps.

3.3.1 Auto-scaling View - Data structure

This view is a subset of the Source:

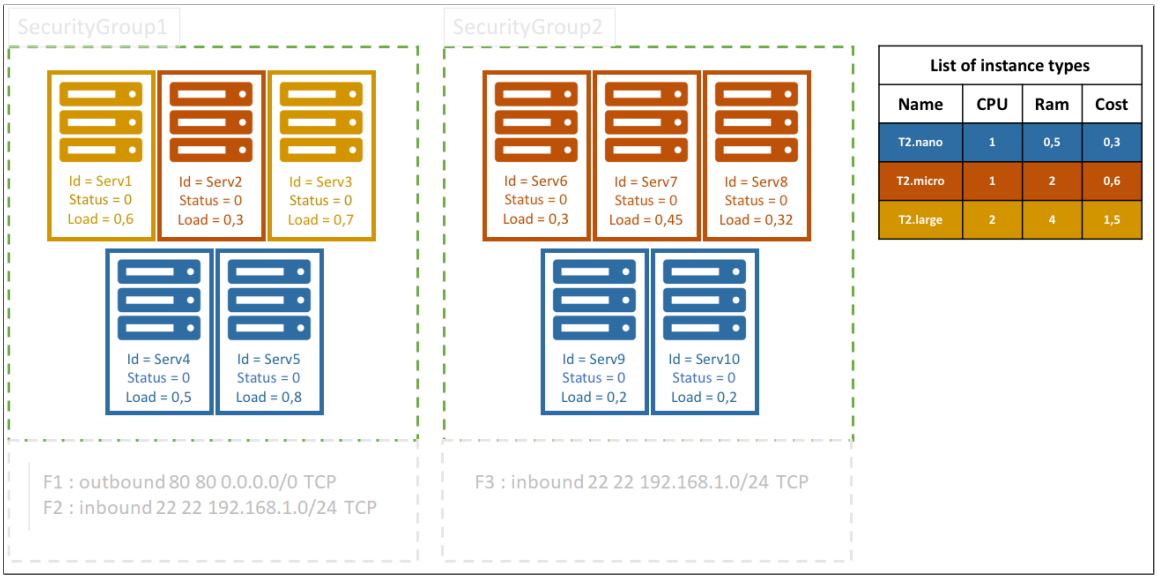


Figure 3.3: Auto-scaling view

It is the same schema than the one for the source, but all the data not needed by the view have been shaded. As a matter of fact, the purpose of the auto-scaling view is to adjust the number of CPU's according to the load-balancing, thus it doesn't need any information about the Security Groups. The main interest lies in the *Load* attribute of each instance. Here is the code of the auto-scaling view structure, where the name convention takes the first letter of the view before the name of the original list (here, auto-scaling becomes AS):

```
1 data ASInstance = ASInstance Id Type State Status SecurityGroupRef Load
2                 deriving (Show, Eq)
3 deriving instance NFData ASInstance
4 deriveBiGULGeneric ''ASInstance
5 deriveJSON defaultOptions ''ASInstance
6
7 data ASInstanceType = ASInstanceType Id TypeCPUs TypeRAM TypeCost
8                 deriving (Show, Eq)
```

```
deriving instance NFData ASInstanceType
deriveBiGULGeneric ''ASInstanceType
deriveJSON defaultOptions ''ASInstanceType

data ASView = ASView [ASInstance] [ASInstanceType]
                deriving (Show)
deriving instance NFData ASView
deriveBiGULGeneric ''ASView
deriveJSON defaultOptions ''ASView
```

Listing 3.2: Data structure of the Auto-scaling view

As showed above, the autoscaling needs all the information in the source, excepted the list of security groups. Indeed, as the computation performed inside this view wants to scale up or down the servers (according to a specific percentage that the view must reach), it must possess all the information about the servers and their types (excluding the *ami* of the instances, as it is a value handled solely by the AWS API). The Haskell data type are not repeated in this document because of the space. However, in the code, all the types are defined again in the view's data structure. To separate properly the source and its different subsystems, we create new Haskell data types for each view. Even if the information is the same, the concern can evolve and change during the lifetime of the software. That is why we duplicate the data types. Moreover, each concern will certainly be executed on a remote and independently server. Finally, in our case we chose to implement the Analysis and Plan of concerns in Haskell like the rest of the code, but we could use another language where we should have to define the data in this language. To summarize, we assume that all concerns are completely independent from each other and from the source. The only links between the main system and its subsystems have to be the bidirectional transformations.

3.3.2 Auto-scaling View - Bidirectional transformation

In the four sections about the bidirectional transformations, one for each view, will be exposed the bidirectional transformations used to synchronize the changes made inside the views, back to the source. Those transformations are one of the two main interests of this thesis, with the self-prioritized execution of the views. They allow the propagation of the information from one module to the whole system, and they ensure that these changes will be properly propagated, thanks to *BiGUL*, which guarantees that the bidirectional transformations are well-behaved.

The bidirectional transformation between the source and the *auto-scaling view* works as follows:

```
autoScalingUpdate :: BiGUL S.Source AS.ASView
autoScalingUpdate =
    $(rearrS [| \(Source insts sgs instTypes) -> (insts, instTypes, sgs)|])$
    $(rearrV [| \(ASView insts instTypes) -> (insts,instTypes,())|])$
    $(update
        [p|(insts,instTypes,sgs)|]
        [p|(insts,instTypes,sgs)|]
        [d| insts = alignInstances;
            instTypes = alignInstanceTypes;
            sgs = Skip (const ())|])
```

Listing 3.3: autoScalingUpdate

As shown in the data structure of the auto-scaling view, it needs the list of instances, the list of instance types, but not the list of security groups. Moreover, regarding the

instances and instance types, it does not need the *ami* (To remember the data structure of the model, please refer to section 3.2.1).Thus, the *autoScalingUpdate* function rearranges the source and the view into tuples, where x is the list of instances, y is the list of instance types, and z is the list of security groups. Once both of them have the same structure, it is simple to call BiGUL's functions to synchronize the instances and the instance types. The third part of the function, *update*, takes the source as first argument, the view as second, and applies the synchronization in the third argument. Indeed, the x calls the *alignInstances* function to align the instances, the y calls the *alignInstanceTypes* to align the instance types, and the z calls BiGUL's *Skip* function to warn that the list of security groups in the source does not have any correspondance in the view.

```
1  alignInstances :: BiGUL [Instance] [ASInstance]
2  alignInstances = align
3       (\_ -> True)
4       (\(Instance s _ _ _ _ _ _) (ASInstance v _ _ _ _ _) -> s == v)
5       $(update
6          [p|Instance identifier instType _ state status sg load|]
7          [p|ASInstance identifier instType state status sg load|]
8          [d|identifier = Replace; instType = Replace; state = Replace; status=
                Replace; sg= Replace; load= Replace|])
9       (\(ASInstance vId vType _ _ vSG vLoad) -> Instance vId vType "" 0 1 vSG vLoad)
10      (\(Instance sId sType sAmi sState _ sSecurityGroupRef sLoad) -> Just (Instance
             sId sType sAmi sState 2 sSecurityGroupRef sLoad))
```

Listing 3.4: Auto-scaling - alignInstances

The above BiGUL's function possesses a list of *Instances* as source, and a list of *ASIstances* as view. It corresponds to the data structure in section 3.2.1. The body of the function simply calls the BiGUL's *align* function explained in section 1.2.2. The first argument of *align*, line 3, represents the elements in the source that do not have to be in the view, but still remain in the source. Here, we need all the instances in the system, so the function has to return *True* all the time. The second argument, line 4, makes the match between the elements of the source and the elements of the view (ie. those which possess the same identifier). The third argument, lines 5 to 8, shows how to update the data of each element. The fourth argument, line 9, explains how to create a new instance in the source with an instance in the view. We drop the state and the status, because the values of those fields in the view does not matter: the new instance in the source must have its state to 0, and its status to 1. Finally, the last argument, line 10, describes how an instance in the source but not in the view has to be handled (ie. totally removed or particularly treated). Here, we set its status to 2, saying that the instance must be terminated.

```
1  alignInstanceTypes :: BiGUL [InstanceType] [ASInstanceType]
2  alignInstanceTypes = align
3       (\_ -> True)
4       (\(InstanceType s _ _ _) (ASInstanceType v _ _ _) -> s == v)
5       $(update
6          [p|InstanceType identifier typeCPUs typeRAM typeCost|]
7          [p|ASInstanceType identifier typeCPUs typeRAM typeCost|]
8          [d|identifier = Replace; typeCPUs = Replace; typeRAM= Replace; typeCost=
                Replace|])
9       (\(ASInstanceType vId vTypeCPUs vTypeRAM vTypeCost) -> InstanceType vId
            vTypeCPUs vTypeRAM vTypeCost)
10      (\_ -> Nothing)
```

Listing 3.5: Auto-scaling - alignInstanceTypes

In the same way, the code above describes the alignement between the instance types of the source and the view. The first argument accepts all the instance types, because the views needs all of them. The second matches the elements in the source with those in the view, with their identifier. The third argument synchronizes the possible changes in the data. The fourth shows how to create a new instance in the source with a new instance type in the view. Finally, the last argument sends always *Nothing*, but this will never occur because it is not allowed to delete an instance type, even in the view.

3.3.3 Auto-scaling View - Analysis and Planning

The aim of the four analysis and planning steps sections, one for each view, is to explain in details the strategies used inside each step of *Analysis* and *Planning*. As a reminder, the *Analysis* step goes over the view and provides some information to the *Planning* step, which may change the view attributes. Those changes will then update the main Source, thanks to the *put* transformation.

The analysis step of the auto-scaling is quite simple. Its purpose is to determine whether the planning must increase or decrease the number of instances to keep a predefined load level. This concern is applied on each security group independently because for our example system we use those groups to separate the type of instance (Web and database). Before getting into the explanation, it's important to define some useful functions.

```
1 groupBySecurityGroup :: [ASInstance] -> [(String, [ASInstance])]
2 groupBySecurityGroup [] = []
3 groupBySecurityGroup (x:xs) = (findSecurityGroups x, filter
4      (\y -> findSecurityGroups x == findSecurityGroups y)
5      (x:xs)):(groupBySecurityGroup (filter (\y-> findSecurityGroups x /=
           findSecurityGroups y) xs))
6
7 findSecurityGroups :: ASInstance -> String
8 findSecurityGroups (ASInstance _ _ _ _ sg _) = sg
```

Listing 3.6: Analyse and plan : groupBySecurityGroup

The *groupBySecurityGroup* function returns a list of pairs, where the first element is the name of a security group and the second element is the list of instances in this security group. As explained earlier, this function is important because the auto-scaling concern is applied on each security group, one by one.

```
1 findNumberOfCPU :: [ASInstanceType] -> ASInstance -> Int
2 findNumberOfCPU instTypes (ASInstance _ instType _ _ _ _) =
3     foldl
4         (\acc (ASInstanceType iden cpu _ _) -> if iden == instType then cpu else
              acc)
5         0 instTypes
6
7 nbrCPU :: [ASInstanceType] -> [ASInstance]  -> Int
8 nbrCPU instTypes = foldl (\acc inst -> (findNumberOfCPU instTypes inst) + acc ) 0
```

Listing 3.7: Analyse and plan : findNumberOfCPU

The first function determines the number of CPU's owned by an instance. It is quite simple: the function finds the linked instance type and retrieves the number of CPUs. Regarding the *nbrCPU* function, it is an extension of *findNumberOfCPU* but applied to a

list of instances.

```
findLoadOfInstance :: [ASInstanceType] -> ASInstance -> Double
findLoadOfInstance instTypes (ASInstance iden instType state status sg l) = l * (
    fromIntegral (findNumberOfCPU instTypes (ASInstance iden instType state status
    sg l)))
```

Listing 3.8: Analyse and plan : findLoadOfInstance

The above function is used to know the exact load of an instance. Indeed, the load value represents the load of each CPU, but an instance can possess several CPU's. Thus, the function computes the total load of an instance.

```
runInstance :: ASInstance -> ASInstance
runInstance (ASInstance iden instType state _ sg l)
     | (state == 16) = (ASInstance iden instType state 0 sg l)
     | otherwise = (ASInstance iden instType state 1 sg l)
```

Listing 3.9: Analyse and plan : runInstance

The *runInstance* function is used to start an instance. Two cases are possible. The first one is when the instance is really started. In this case, the previous status does not matter and the function reset it to 0. The second one is when the instance has another state than started (it could be pending, shutting-down, stopping, stopped, ...). In this case, the function puts the status at 1 to notify that the system has to restart the instance.

```
instancesRunning :: [ASInstance] -> [ASInstance]
instancesRunning = filter (\(ASInstance _ _ state status _ _) -> (state == 16 &&
    status /= 2) || (state /= 16 && status == 1))

instancesStopped :: [ASInstance] -> [ASInstance]
instancesStopped = filter (\(ASInstance _ _ state status _ _) -> (status == 0 &&
    state == 80) || (state == 16 && status == 2))
```

Listing 3.10: Analyse and plan : instancesRunning and instancesStopped

These two functions are used to filter a list of instances. The first one retrieves the running instances. It means those that are started in the system and not stopped by another concern, or those stopped in the system but started by another concern. The second function retrieves the stopped instances. It means those that are stopped in the system and not started by another concern, or those started in the system but stopped by another concern. Note that both ignore all instances with an intermediate state to simplify the analysis and plannnig steps.

After having reviewed all those useful functions, the strategy used for the analysis can be explained. This step calculates, for each *Security Group*, the number of CPU's needed to reach a certain level of load, assuming a perfect distribution of the load. The calculation does not take into account the instances which must be stopped (as asked by a previous view), because the charge they are currently absorbing will not absorb anything after stopping them.

```
autoScalingAnalysis :: Double -> AS.ASView -> [(String, Int)]
autoScalingAnalysis avrLoadExpected (ASView instances instTypes) =
```

```
3     map
4         (\(a,b) -> (a, ceiling (
5             (foldl
6                 (\acc inst ->  (findLoadOfInstance instTypes inst) + acc)
7                 0.0 (instancesRunning b)
8             ) / avrLoadExpected))
9         )
10        (groupBySecurityGroup instances)
```

Listing 3.11: Analyse of the auto-scaling view : autoScalingAnalysis

The first step splits the list of instances with the *groupBySecurityGroup* function. After that, for each *Security Group*, it computes the number of CPU's required to reach the average load expected. The result of this function looks like a list of pairs, where each of them links a *Security Group* with its expected number of CPU's. The analysis is over. The next step is the planner.

```
1  autoScalingPlan :: [(String, Int)] -> AS.ASView -> AS.ASView
2  autoScalingPlan analyses (ASView instances instanceTypes) =
3  ASView (foldl (\acc (sg, insts) -> case find (\(sg2, i) -> sg == sg2) analyses of
4      Just (_,cpuExpected) ->
5          if (cpuExpected - (nbrCPU instanceTypes (instancesRunning insts))) >= 0
6          then
7                  acc ++ (increaseWithNewInstances cpuExpected sg instanceTypes (
                       increaseWithOffInstances cpuExpected instanceTypes insts))
8          else
9                  acc ++ (decrease cpuExpected instanceTypes insts)
10     Nothing -> error "Analyse is inconsistent with the planner data"
11     ) [] (groupBySecurityGroup instances)) instanceTypes
```

Listing 3.12: Planning of the auto-scaling view : autoScalingPlan

It uses the result of the analysis to add or remove some instances. As a reminder, the main purpose of the auto-scaling view is to absorb peak loads and making sure that the system is working properly. The strategy is divided in two parts. Firstly, it groups all the instances by their *Security Group*. Secondly, it checks if it is needed to add or remove instances, according to the number of CPUs calculated by the analysis step. The first part uses the *groupBySecurityGroup* function in the same way as before and the second part uses *increaseWithNewInstances* and *increaseWithOffInstances* depending on the number of CPUs requested:

```
1  increaseWithOffInstances :: Int -> [ASInstanceType] -> [ASInstance] -> [ASInstance]
2  increaseWithOffInstances cpu instTypes instances =
3      if (nbrCPU instTypes ((instancesStopped instances) ++ (instancesRunning
           instances))) < cpu
4          then (instancesRunning instances) ++ map runInstance (instancesStopped
               instances)
5          else snd (foldl (\(acc1, acc2) inst -> if acc1 < cpu
6              then (acc1 + (findNumberOfCPU instTypes inst)  ,(runInstance inst):acc2
                   )
7              else (acc1, inst:acc2)) ((nbrCPU instTypes (instancesRunning instances)
                   ),(instancesRunning instances)) (instancesStopped instances))
```

Listing 3.13: Planning of the auto-scaling view : increaseWithOffInstances

```
increaseWithNewInstances ::  Int -> String -> [ASInstanceType] -> [ASInstance] -> [
    ASInstance]
increaseWithNewInstances cpu currentSG instTypes instances
     | cpu > (nbrCPU instTypes (instancesRunning instances)) =
        increaseWithNewInstances cpu currentSG instTypes (((\(ASInstanceType iden _
             _ _) -> (ASInstance "" iden 0 1 currentSG 0)) bestInstanceType):
            instances)
     | otherwise = instances
    where
        bestInstanceType =
            foldl (\(ASInstanceType iden1 tCPU1 tRam1 tCost1) (ASInstanceType iden2
                 tCPU2 tRam2 tCost2) ->
                if (cpu - tCPU2) > 0 && (cpu - tCPU1) > (cpu - tCPU2)
                then (ASInstanceType iden2 tCPU2 tRam2 tCost2)
                else (ASInstanceType iden1 tCPU1 tRam1 tCost1))
            (head instTypes) instTypes
```

Listing 3.14: Planning of the auto-scaling view : increaseWithNewInstances

If the system needs to increase the number of CPU's, the strategy works as follows: The *increaseWithOffInstances* function starts the stopped instances linked to the same *Security Group*, trying to reach the right number of CPU's. Then, if it is not enough, the *increaseWithNewInstances* function creates new instances based on the instances types available.

```
compareWeightOfSubSeq :: ([ASInstance], (Int, Int)) -> ([ASInstance], (Int, Int))
    -> Ordering
compareWeightOfSubSeq (_,(cpu1, runningInsts1)) (_, (cpu2, runningInsts2))
     | cpu1 < cpu2 = LT
     | cpu1 == cpu2 && runningInsts1 < runningInsts2 = LT
     | otherwise = GT

decrease :: Int -> [ASInstanceType] -> [ASInstance] -> [ASInstance]
decrease cpu instTypes instances =
    (removeInstances ((nbrCPU instTypes instances) - cpu) instTypes (
        instancesRunning instances)) ++ (instancesStopped instances)
    where
        removeInstances :: Int -> [ASInstanceType] -> [ASInstance] -> [ASInstance]
        removeInstances nbToRemove instTypes [] = []
        removeInstances nbToRemove instTypes (x:xs)
              | (findNumberOfCPU instTypes x) <= nbToRemove =
                (stopInstance x):(removeInstances (nbToRemove - (findNumberOfCPU
                    instTypes x)) instTypes xs)
              | otherwise = x:(removeInstances nbToRemove instTypes xs)
```

Listing 3.15: Planning of the auto-scaling view : decrease

If the system needs to decrease the number of CPU's, the strategy is to stop some instances. The *decrease* function generates all the possible subsequences of instances. For example, if the list of instances contains two items (a and b), the list of all subsequences will be *[[],["a"],["b"],["a","b"]]*. After that, the strategy is to determine the weight of each subsequence. In those functions, the weight represents the difference between the number of CPU's expected and the number of CPU's contained inside this subsequence. Only those with a weight higher than 0 will be kept, to get those which possess at least the number of CPU's required. Thereafter, the subsequences are sorted following a bottom-up approach

(and according to their weight). Finally, it takes the head of the list which is the best composition of instances with the expected number of CPU's.

This ends the explanations of the *auto-scaling view*. The next one is the *redundancy view*.

3.4 The Redundancy View

Related to the Availability concern, the redundancy view must keep the system available at any time. Thus, it has to keep at least two running instances inside each security group. If one goes down, the other one guarantees the access and keeps the security group available. The further sections show the data structure of the view, followed by the bidirectional transformation between the source and the redundancy view, to finish with the analysis and planning steps.

3.4.1 Redundancy View - Data structure

Following the structure of the source (Section 3.2.1) the schema for the redundancy view looks like:

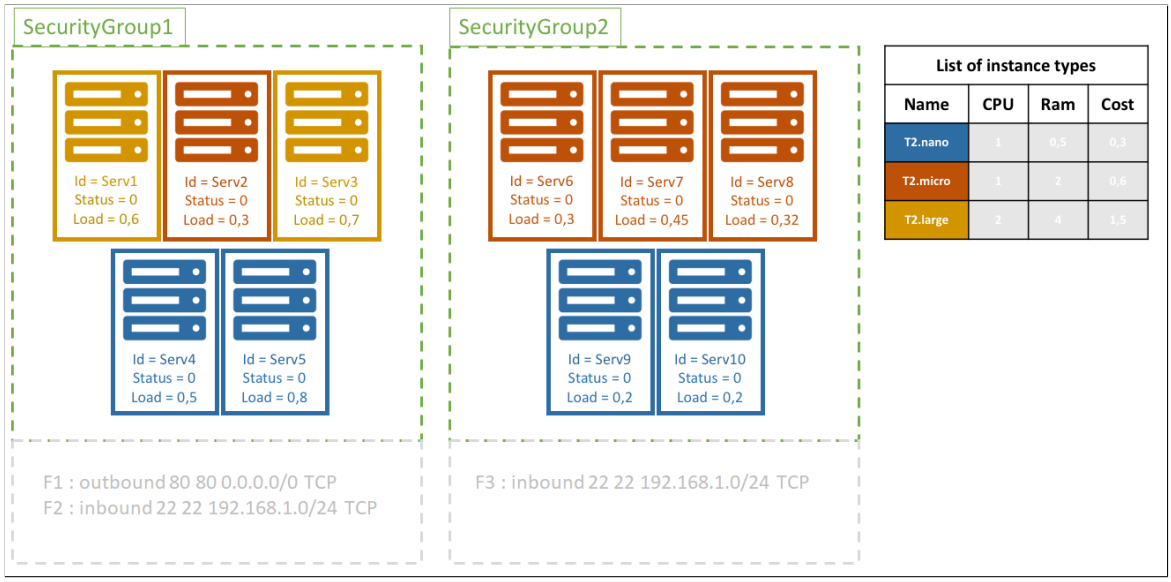


Figure 3.4: Redundancy view

The only thing that matters to the redundancy view is to be sure that, every time a user needs the system, it is operational and functional. Therefore, this view simply requires the number of CPUs (ie. running instances) available in each *Security Group*, to be able to guarantee the Availability at all times. The name convention is the same than the auto-scaling view, and will be the same for the two last views. The code related to this schema is as follows:

```
1 data RInstance = RInstance Id Type State Status SecurityGroupRef
      deriving (Show, Eq)
  deriving instance NFData RInstance
  deriveBiGULGeneric ''RInstance
5 deriveJSON defaultOptions ''RInstance

  data RSecurityGroup = RSecurityGroup Id InstanceRefs
```

```
    deriving (Show)
deriving instance NFData RSecurityGroup
deriveBiGULGeneric ''RSecurityGroup
deriveJSON defaultOptions ''RSecurityGroup

data RInstanceType = RInstanceType Id
    deriving (Show, Eq)
deriving instance NFData RInstanceType
deriveBiGULGeneric ''RInstanceType
deriveJSON defaultOptions ''RInstanceType

data RView = RView [RInstance] [RSecurityGroup] [RInstanceType]
    deriving (Show)
deriving instance NFData RView
deriveBiGULGeneric ''RView
deriveJSON defaultOptions ''RView
```

Listing 3.16: Data structure of the Redundancy view

Obviously, the AMI is not relevant for this view, as well as the firewall rules inside each security groups. It only needs the type of the instance, its state and status (to know if it is already running), and its security group. The list of security groups is needed to take into account those which are not related to any instances. Regarding the list of instance types, we need them in the planning strategy, explained later.

3.4.2 Redundancy View - Bidirectional transformation

The code for the bidirectional transformation between the *redundancy view* and the source is:

```
redundancyUpdate :: BiGUL S.Source R.RView
redundancyUpdate =
    $(rearrS [| \(Source insts sgs instTypes) -> (insts,sgs,instTypes)|])$
    $(rearrV [| \(RView insts sgs instTypes) -> (insts,sgs,instTypes)|])$
    $(update
        [p|(insts,sgs,instTypes)|]
        [p|(insts,sgs,instTypes)|]
        [d|insts = alignInstances; sgs = alignSecurityGroups; instTypes =
            alignInstanceTypes|])
```

Listing 3.17: redundancyUpdate

As the view needs the three lists (ie. the instances, the instance types and the security groups), the rearrangement only converts the data structure of the source and the view into tuples, to be able to associate the instances in the source with the instances in the view, and the same for the instances type and the security groups. These conversions are handled by three BiGUL's functions: *alignInstances*, *alignSecurityGroups* and *alignInstanceTypes*.

```
alignInstances :: BiGUL [Instance] [RInstance]
alignInstances = align
    (\(Instance _ _ _ state status _ _) -> (state /= 48))
    (\(Instance s _ _ _ _ _ _) (RInstance v _ _ _ _) -> s == v)
    $(update
        [p|Instance identifier instType _ instState instStatus instGroupRef _|]
        [p|RInstance identifier instType instState instStatus instGroupRef |]
        [d|identifier = Replace; instType = Replace; instState = Replace;
            instStatus = Replace; instGroupRef = Replace|])
```

```
9     (\(RInstance vId vType vState vStatus vGroupRef) ->
10        Instance vId vType "" vState vStatus vGroupRef 0)
11    (\(Instance sId sType sAmi sState _ sSecurityGroupRef sLoad) ->
12        Just (Instance sId sType sAmi sState 2 sSecurityGroupRef sLoad))
```

Listing 3.18: Redundancy - alignInstances

The *alignInstances* function calls the *align* function already implemented inside BiGUL. The *align* function requires five arguments. The first one describes the elements that must not be taken into account in the view. In this case, the redundancy concern needs all the instances of the system, except the terminated ones (state equals 48) because we cannot restart them. Indeed, it needs the running instances to know which security group does not satisfy the redundancy, and the stopped ones to restart them if needed. The second argument of the function matches the elements in the source with their associated element in the source, thanks to their identifier. The third argument, line 5, characterizes the update function. Clearly, it only needs to *replace* each information to propagate the potential changes in both directions, and according to the data structure of Section 3.2.1. The fourth argument of the *align* function describes how to produce in the source a new element created in the view. The function reuses the information contained in the view, and put basic values for all the other attributes. The last argument shows how to delete an element in the source when the view asked for it. The strategy of this thesis is to pass the *Status* attribute to 2 when the element must be deleted. As a reminder, a status of 0 means that no change has to be made on the instance, and a status of 1 says that the instance must be restarted if it exists, or created if not. The deletion will not violate the GetPut property, as it will delete it only if the view requested it.

```
1  alignSecurityGroups :: BiGUL [SecurityGroup] [RSecurityGroup]
2  alignSecurityGroups = align
3      (\_ -> True)
4      (\(SecurityGroup s _ _ _) (RSecurityGroup v _) -> s == v)
5      $(update
6          [p|SecurityGroup identifier _ instanceRefs _|]
7          [p|RSecurityGroup identifier instanceRefs|]
8          [d|identifier = Replace; instanceRefs = Replace|])
9      (\(RSecurityGroup vId vRefs) -> SecurityGroup vId "" vRefs [])
10     (\_ -> Nothing)
```

Listing 3.19: Redundancy - alignSecurityGroups

Once again, the *align* function of BiGUL is used. Its five arguments are very comparable to those depicted above for the *alignInstances* function. As all the security groups must be taken into account by the concern, the first argument sends always *True*. Indeed, the redundancy has to be sure that all the security groups possess at least two running instances. Thus, all of them must be represented in the view. The second argument, as usual, matches the elements in the source with their associated element in the view. The update function, line 5, follows the data structure in Section 3.2.1 to link accurately the information with the *Replace* function. The fourth argument still describes how to create in the source a possibly new security group in the view. The list of firewall rules is empty, because it is AWS that knows the specific ID of the instances. As we want to create it, this security group does not currently exist on AWS. Thus, it is AWS that will link automatically the security group with its instances at its creation. Finally, the fifth argument, which is supposed to show how to delete a security group in the source, sends back *Nothing*. As a matter of fact, the *redundancy concern* creates or restarts instances in

the security groups, but can not in any circumstances remove one of them.

```
alignInstanceTypes :: BiGUL [InstanceType] [RInstanceType]
alignInstanceTypes = align
     (\_ -> True)
     (\(InstanceType s _ _ _) (RInstanceType v) -> s == v)
     $(update
        [p|InstanceType identifier _ _ _|]
        [p|RInstanceType identifier|]
        [d|identifier = Replace|])
     (\(RInstanceType vId) -> InstanceType vId 0 0 0)
     (\_ -> Nothing)
```

Listing 3.20: Redundancy - alignInstanceTypes

Still with the *align* function, its arguments for the *alignInstanceTypes* function are as follows. The first one allows all the instance types to be taken into the view. As the redundancy concern needs to know the types to restart (or create) the instances, all of them are mandatory. The second argument matches the types in the source with the types in the view. The third one, the update function, easily binds the information of related elements, invariably thanks to the *Replace* and according to Section 3.2.1. The penultimate argument shows how to create a new instance type in the source from the view. As the information represented in the view is minimal, there are a lot of basic values used to create the type in the source. The last argument depicts eventually how to delete an instance type, and sends *Nothing* all the time because it is forbidden for the view to delete a type.

Thanks to those functions, the redundancy view can be properly created with the right data from the source, and the potential changes in this view can be correctly propagated back to the source.

3.4.3 Redundancy View - Analysis and Planning

The chosen strategy for this analysis is the following one. It has to return a list of pairs, whose first element is the id of a *Security Group* which does not satisfy our redundancy contraint, and the second one is the number of running *RInstances* within this group (ie. either 0 or 1). The first part of the *redundancyAnalysis* function, (ie. before the "++" operator), returns all the security groups with several instances but none, or only one, is currently running. The second part handles the security groups without any related instances:

```
redundancyAnalysis :: [RInstance] -> [RSecurityGroup] -> [(RSecurityGroup, Int)]
redundancyAnalysis instances securityGroups =
    (
        map (\(groupName, occurrence) -> case find (\(RSecurityGroup i _) ->
            groupName == i) securityGroups of
            Just s -> (s, occurrence)
            Nothing -> error ("Source inconsistent"))
        (findSecurityGroups instances)
    )
    ++ (map (\securityGroup -> (securityGroup, 0)) (filter (\(RSecurityGroup _ ref)
         -> (length ref == 0)) securityGroups))
```

Listing 3.21: Analyse of the Redundancy view : redundancyAnalysis

To achieve the first part, the program applies the *countRunningInstances* function on each instance of the list. This function returns a pair whose first element is the id of the security group of the current instance, and the second is the number of running instances related to this security group. To increment this counter, an instance must have either its state to 16 and its status not equal to 2 (meaning that it is running (ie. state equals 16) but none of the previous views requested to stop it (ie. status not equal to 2)), either its status to 1 (meaning that a previous view requested to restart the instance). With the result of the *countRunningInstances* function, the program filters the pairs to keep only those with their second element below 2, thus those which do not respect the redundancy constraint.

```
1 findSecurityGroups :: [RInstance] -> [(String, Int)]
2 findSecurityGroups instances = nub (filter
3     (\(_,occ) -> occ < 2)
4     (map
5         (\(RInstance _ _ _ _ ref) -> (countRunningInstances (filter
6             (\(RInstance _ _ _ _ ref1) -> ref == ref1)
7             instances)))
8         instances))
```

Listing 3.22: Analyse of the Redundancy view : findSecurityGroups

```
1 countRunningInstances :: [RInstance] -> (String, Int)
2 countRunningInstances (inst@(RInstance i t state status ref):instances) =
3     (
4       ref,
5       length (L.filter (\(RInstance _ _ state status _) ->
6           (state == 16 && status /= 2) || (status == 1)) (inst:instances))
7     )
```

Listing 3.23: Analyse of the Redundancy view : countRunningInstances

```
1 frequency :: (Ord a) => [a] -> [(a, Int)]
2 frequency xs = toList (fromListWith (+) [(x, 1) | x <- xs])
```

Listing 3.24: Analyse of the Redundancy view : frequency

The second part (ie. after the "++" operator, line 9 in Listing 3.21) returns all the *Security Groups* without any reference to instances.

Thanks to the pairs returned by the analysis, the *Planning* knows exactly on which *Security Groups* it has to work. This step is divided in three parts. First, inside the *redundancyPlan* function, the analysis is performed on the view. Then, in the *completeInstances* function, each pair returned by the analysis is send separately to update the list of *RInstances* in the *handleCompleteness* function, which forms the last part.

```
1 redundancyPlan :: RView -> RView
2 redundancyPlan inst@(RView instances securityGroups instanceTypes) =
3     completeInstances
4         inst
5         (redundancyAnalysis instances securityGroups)
```

Listing 3.25: Planning of the Redundancy view : redundancyPlan

```
completeInstances :: RView -> [(RSecurityGroup, Int)] -> RView
completeInstances view [] = view
completeInstances (RView instances securityGroups instanceTypes) (incompleteSG:ss)
    =
    completeInstances
        (RView (handleCompleteness instances instanceTypes incompleteSG)
            securityGroups instanceTypes)
        ss
```

Listing 3.26: Planning of the Redundancy view : completeInstancesPlan

The *handleCompleteness* function works differently depending on the number of running *RInstances* inside the security group (ie. either 0 or 1). If there is already one running instance, the strategy is to launch another instance of the same type. Indeed, two instances with different types (and so different computing capacity) could lead to a problem if the strongest one falls. In order to start such an instance, the program has to verify if there is a stopped instance with the same type and in the same security group already in the system before deciding to start a new one. It is the purpose of the second *find* function, line 4. If it does not find such an instance, it launches a brand new instance of the correct type. Otherwise, it restarts it by either setting its status to 0 (if the previous status was 2, another view planned to stop the instance but it is still running, and so the plan does not have to do anything) or to 1 to really restart it. Regarding the first *find* function, line 3, retrieves the sole running instance of the security group.

If there is no running instance, the goal is still to possess in the end two instances of the same type. To do so, it tries to find the most present instance type inside the stopped instances related to the security group, line 8. From there, three possibilities can occur.

Firstly, the security group is not related to any instance, and so the list of instances is empty. Thus, it checks among all the instances of all the security groups to find the most used in the system. If it comes across one, it launches a new instance of that type. If it does not identify any type, it means that there is not any instance in the whole system, and so launches the first proposed type inside the instance types list. It is decided by our strategy to take the first one by default, because the first one is the smallest. As there is currently zero running instance, it means that the charge is null. The smallest instance type is then enough.

Secondly, there is only one instance of that type in the security group. In that case, the program restarts the instance and creates a new instance of the same type.

Finally, when at least two instances have been found, the program simply restarts them.

```
handleCompleteness :: [RInstance] -> [RInstanceType] -> (RSecurityGroup, Int) -> [
    RInstance]
handleCompleteness instances ((RInstanceType typeId _ _ _):_) ((RSecurityGroup
    secId refs), occurrence) = case occurrence of
    1 -> case find (\(RInstance instId _ state status _) -> (state == 16 || status
        == 1) && (elem instId refs)) instances of
        Just (RInstance instId1 instType1 _ _ _) -> case find (\(RInstance instId2
            instType2 _ _ _) -> (instId1 /= instId2) && (instType1 == instType2) &&
             (elem instId2 refs)) instances of
            Just (RInstance i t ste stus sRef) -> (delete (RInstance i t ste stus
                sRef) instances) ++ [(RInstance i t ste (if (stus == 2) then 0 else
                 1) sRef)]
            Nothing -> instances ++ [(RInstance "" instType1 0 1 secId)]
        Nothing -> error ("Source inconsistent")
```

```
8     0 -> case filter (\(RInstance _ instType _ _ _) -> instType == findMostUsedType
           (filter (\(RInstance _ _ _ _ sgRef) -> sgRef == secId) instances))
          instances of
9         [] -> case filter (\(RInstance _ t _ _ _) -> t == findMostUsedType
              instances) instances of
10            ((RInstance _ mostUsedType _ _ _):[]) -> instances ++ [(RInstance ""
                  mostUsedType 0 1 secId), (RInstance "" mostUsedType 0 1 secId)]
11            otherwise -> instances ++ [(RInstance "" typeId 0 1 secId), (RInstance
                  "" typeId 0 1 secId)]
12        ((RInstance i t ste stus sRef):[]) -> (delete (RInstance i t ste stus sRef)
              instances) ++ [(RInstance i t ste (if (stus == 2) then 0 else 1) sRef)
              , (RInstance "" t 0 1 sRef)]
13        ((RInstance i1 t1 ste1 stus1 sRef1):(RInstance i2 t2 ste2 stus2 sRef2):_)
              ->
14            (delete (RInstance i2 t2 ste2 stus2 sRef2) (delete (RInstance i1 t1
                  ste1 stus1 sRef1) instances)) ++ [(RInstance i1 t1 ste1 (if (stus1
                  == 2) then 0 else 1) sRef1), (RInstance i2 t2 ste2 (if (stus2 == 2)
                   then 0 else 1) sRef2)]
```

Listing 3.27: Planning of the Redundancy view : handleCompleteness

```
1 findMostUsedType :: [RInstance] -> String
2 findMostUsedType [] = ""
3 findMostUsedType instances = fst (maximumBy (comparing snd) (frequency (map (\(
      RInstance _ instType _ _ _) -> instType) instances)))
```

Listing 3.28: Planning of the Redundancy view : findMostUsedType

These explanations finish the *redundancy view* section. The next one is devoted to the *firewall view*.

3.5 The Firewall View

The goal of the view, standing for the Security concern, is to ensure that all security groups follow the basic access policies according to their respective function, so that the security of each instance inside the security groups is relevant. As usual, first comes the data structure, then the bidirectional transformation associated to this view, to end with the analysis and planning steps.

3.5.1 Firewall View - Data structure

As you can see in Figure 3.5, the structure of the view is the same as the source. All along this section, you will see some data already explained in the source but with a different name due to our naming convention.

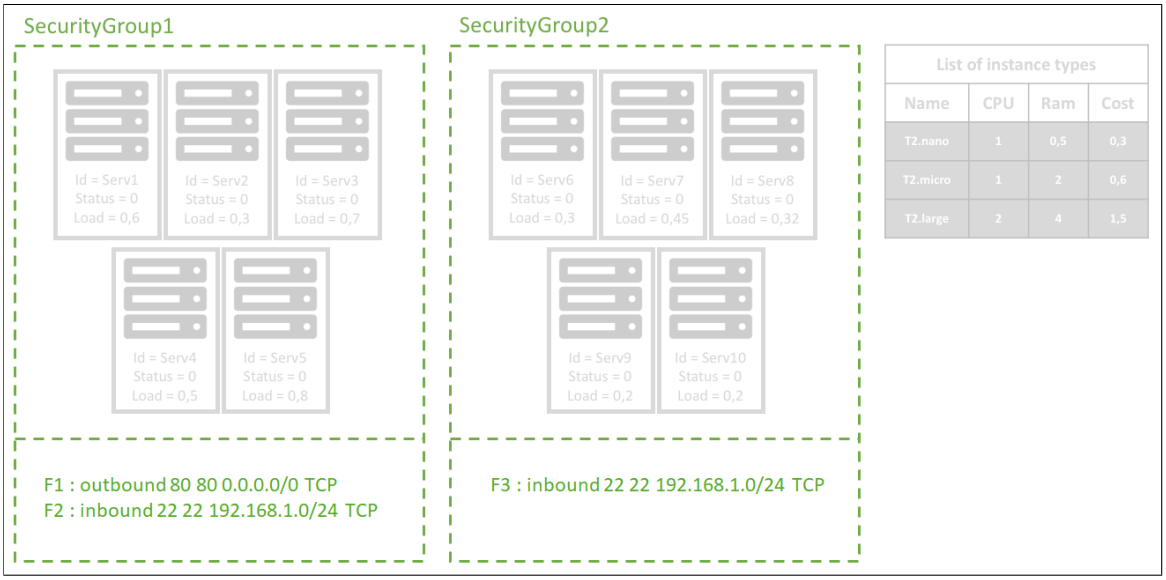


Figure 3.5: Firewall view

As shown above, in contrast with the three other views, the firewall view does not need the instances neither the instance types, but only the security groups and their associated firewall rules. Indeed, this view is just concerned by the access policies, and thus does not care about what is inside the instances or the instance types. Before continuing, let's explain the concept of firewall rules and how these rules are composed. By default, all the traffic is closed in a security group. The rule let the user specify explicitly the traffic that the security group allows. As you saw in Figure 3.5, the firewall rule starts with a key word (inbound or outbound) to signify the type of rule. If it is inbound, the rule will be applied to the traffic coming from the external. If it is outbound, the rule will be applied to the outgoing traffic. The second part of the rule is the port range. In other words, there are two values that specify the ports handled by this rule. For example, to manage the HTTP port (80), you have to specify 80 for the first value and also 80 for the second value.To manage all the ports from 8080 to 8082, you have to specify 8080 for the first value and also 8082 for the second value. Then, the next parameter is the IP address handled by this rule. It is a formatted string containing the IP and the mask like *0.0.0.0/0* for IPv4 and like *::/0* for IPv6. Thus, if it is an inbound rule, the IP is the source of the traffic. If it is an outbound rule, the IP is the destination of the traffic. Finally, the last value is the name of the protocol handled by the rule. Basically, this field is filled with TCP or UDP. The summarize, the rule *outbound 80 80 0.0.0.0/0 TCP* means in english that all the outgoing HTTP traffic is allowed and the rule *inbound 22 22 192.168.1.0/24 TCP* means that all the incoming SSH traffic is allowed for the IPs from 192.168.1.0 to 192.168.1.255. The code looks like this:

```
data FRule = FRule Outbound FromPort ToPort Ip Protocol
    deriving (Show, Eq)
deriving instance NFData FRule
deriveBiGULGeneric ''FRule
deriveJSON defaultOptions ''FRule

data FSecurityGroup = FSecurityGroup Id [FRule]
    deriving (Show)
deriving instance NFData FSecurityGroup
deriveBiGULGeneric ''FSecurityGroup
deriveJSON defaultOptions ''FSecurityGroup
```

```
data FView = FView [FSecurityGroup]
    deriving (Show)
deriving instance NFData FView
deriveBiGULGeneric ''FView
deriveJSON defaultOptions ''FView
```

Listing 3.29: Data structure of the Firewall view

As mentioned above, only the security groups and their associated firewall rules are worth it in this view and so they are the only data kept by the view from the source. Also, the data are the same as the source but, as already explained, to properly separate the view and the source we chose to redefine them for the subsystem.

3.5.2 Firewall View - Bidirectional transformation

The bidirectional transformation implemented for this view is:

```
firewallUpdate :: BiGUL Source FView
firewallUpdate =
    $(rearrS [| \(Source insts sgs instTypes) -> (sgs, (insts, instTypes)) |])$
    $(rearrV [| \(FView sgs) -> (sgs,()) |])$
        alignSecurityGroups `Prod` (Skip (const ()))
```

Listing 3.30: firewallUpdate

As the only list that matters for the Firewall view is the list of security groups, the rearrangement puts it as the first element of the pair for the source, and set the two other lists in the second element. For the view, the rearrangement places the associated list as the first element, and nothing as the second. With those structures, the *alignSecurityGroups* function can synchronize properly the lists of security groups in the source and in the view, while skipping the two other lists of the source.

```
alignSecurityGroups :: BiGUL [SecurityGroup] [FSecurityGroup]
alignSecurityGroups = align
     (\_ -> True)
     (\(SecurityGroup s _ _ _) (FSecurityGroup v _) -> s == v)
     $(update
        [p| SecurityGroup identifier _ _ rules |]
        [p| FSecurityGroup identifier rules |]
        [d| identifier = Replace; rules = alignRules |])
     (\(FSecurityGroup vId _) -> SecurityGroup vId "" [] [])
     (\_ -> Nothing)
```

Listing 3.31: Firewall - alignSecurityGroups

The *align* function associates the security groups in the source and the view accurately. The first argument, which filters the security groups from the source to the view, allows all of them to be represented in the view. The second matches the correct security groups between them with their id. The third argument synchronizes the data of the related security groups, thanks to the *update* function. To handle the list of rules inside each security group, the *alignRules* function is used. The fourth argument shows how to create a new security group, create in the view, in the source. Finally, the last should describe how to delete a security group in the source when it is not present anymore in the view. As the deletion of a security group is not allowed in the view, this argument is empty.

```
alignRules :: BiGUL [FirewallRule] [FRule]
alignRules = align
    (\_ -> True)
    (\(FirewallRule sOutBound sFrom sTo sIp sProtoc) (FRule vOutBound vFrom vTo vIp
         vProtoc) -> (sOutBound == vOutBound) && (sFrom == vFrom) && (sTo == vTo)
        && (sIp == vIp) && (sProtoc == vProtoc))
    $(update
        [p| FirewallRule outbound from to ip protoc |]
        [p| FRule outbound from to ip protoc |]
        [d| outbound = Replace; from = Replace; to = Replace; ip = Replace; protoc
            = Replace |])
    (\(FRule outbound from to ip protoc) -> FirewallRule outbound from to ip protoc
        )
    (\_ -> Nothing)
```

Listing 3.32: Firewall - alignRules

The rules are synchronized thanks to the *alignRules* function and its five arguments. The first argument sends always *True* because all the rules must be taken into account. The second matches the rules. As a rule does not possess an id, the rules are matched with all their data. The third argument synchronizes the data of associated rules with the BiGUL's *Replace* function. The fourth explains the creation in the source of a rule created in the view. The last argument returns always *Nothing* because it not allowed to delete a rule in the firewall view.

3.5.3 Firewall View - Analysis and Planning

As basic example for the Firewall view, the strategy implemented consists of assuring that the security groups with "web" and "database" in their name obey to the elementary rules for web and database groups. Thus, the security groups related to the web must be opened on port 80, and the security groups related to the databases must close the port 3306. Moreover, all of the security groups must be always accessible with SSH, and so their port 22 must stay open. The analysis scans the rules of each security groups and sends to the planning all of those which does not satisfy the basics explained above. This view is a simple example of a firewall analysis that checks only a few rules. It is thus hard-coded. Obviously, this implementation is simple and can be greatly improved by giving the ability to add firewall rules on the fly, using a configuration file. Keep in mind that the purpose of this work is to demonstrate the feasibility of a modular system using BiGUL. Then, we will not dwell on safety rules.

```
firewallAnalysis :: FView -> [(FSecurityGroup, Int, Int)]
firewallAnalysis (FView ls) = map handleSecurityGroup ls
```

Listing 3.33: Analyse of the Firewall view : firewallAnalysis

```
handleSecurityGroup :: FSecurityGroup -> (FSecurityGroup, Int, Int)
handleSecurityGroup (FSecurityGroup name rs) =
    if (isInfixOf "web" name) then
        case checkPort22 of
            0 -> case checkPort80 of
                0 -> (FSecurityGroup name rs, 0, 0)
                otherwise -> (FSecurityGroup name rs, 0, 1)
            1 -> case checkPort80 of
                0 -> (FSecurityGroup name rs, 1, 0)
                otherwise -> (FSecurityGroup name rs, 1, 1)
```

```
            otherwise -> case checkPort80 of
                0 -> (FSecurityGroup name rs, 2, 0)
                otherwise -> (FSecurityGroup name rs, 2, 1)
    else
        if (isInfixOf "database" name) then
            case checkPort22 of
                0 -> case checkPort3306 of
                    0 -> (FSecurityGroup name rs, 0, 2)
                    otherwise -> (FSecurityGroup name rs, 0, 3)
                1 -> case checkPort3306 of
                    0 -> (FSecurityGroup name rs, 1, 2)
                    otherwise -> (FSecurityGroup name rs, 1, 3)
                otherwise -> case checkPort3306 of
                    0 -> (FSecurityGroup name rs, 2, 2)
                    otherwise -> (FSecurityGroup name rs, 2, 3)
        else case checkPort22 of
            0 -> (FSecurityGroup name rs, 0, 4)
            1 -> (FSecurityGroup name rs, 1, 4)
            otherwise -> (FSecurityGroup name rs, 2, 4)

    where
        checkPort22 = (length (filter (\(FRule _ from to _ _) -> ((from <= Just 22)
             && (to >= Just 22))) rs))
        checkPort80 = (length (filter (\(FRule o from to _ _) -> ((not o) && (from
            <= Just 80) && (to >= Just 80))) rs))
        checkPort3306 = (length (filter (\(FRule o from to _ _) -> ((not o) && (
            from <= Just 3306) && (to >= Just 3306))) rs))
```

Listing 3.34: Analyse of the Firewall view : handleSecurityGroup

The response of the analysis is a list of triplet, whose first element is the name of the security group and the second and third are respectively a number that will define the behavior of the planning with the SSH access, and the web/database access. With this result, the planning can apply the right strategy to end up with the appropriate access policies.

```
firewallPlan :: FView -> FView
firewallPlan fv = FView (addSSH (firewallAnalysis fv))
```

Listing 3.35: Analyse of the Firewall view : firewallPlan

```
addSSH :: [(FSecurityGroup, Int, Int)] -> [FSecurityGroup]
addSSH ls =
    map (\(FSecurityGroup i rs, port22, webOrDB) -> case (port22, webOrDB) of
        (0, 0) -> (FSecurityGroup i (addSSHRules (addWebRule rs)))
        (0, 1) -> (FSecurityGroup i (addSSHRules rs))
        (0, 2) -> (FSecurityGroup i (addSSHRules rs))
        (0, 3) -> (FSecurityGroup i (addSSHRules (removeDBAccess rs)))
        (0, 4) -> (FSecurityGroup i (addSSHRules rs))
        (1, 0) -> (FSecurityGroup i (addWebRule (addSSHRules (removeSSHRule rs))))
        (1, 1) -> (FSecurityGroup i (addSSHRules (removeSSHRule rs)))
        (1, 2) -> (FSecurityGroup i (addSSHRules (removeSSHRule rs)))
        (1, 3) -> (FSecurityGroup i (removeDBAccess (addSSHRules (removeSSHRule rs)
            )))
        (1, 4) -> (FSecurityGroup i (addSSHRules (removeSSHRule rs)))
        (2, 0) -> (FSecurityGroup i (addWebRule rs))
        (2, 1) -> (FSecurityGroup i rs)
        (2, 2) -> (FSecurityGroup i rs)
        (2, 3) -> (FSecurityGroup i (removeDBAccess rs))
```

```
18          (2, 4) -> (FSecurityGroup i rs))
19      ls
20      where
21          addSSHRules ls = ls ++ [FRule True (Just 22) (Just 22) "0.0.0.0/0" "TCP",
                FRule False (Just 22) (Just 22) "0.0.0.0/0" "TCP"]
22          removeSSHRule ls = filter (\(FRule _ from to _ _) -> not ((from == Just 22)
                 && (to == Just 22))) ls
23          addWebRule ls = ls ++ [FRule False (Just 80) (Just 80) "0.0.0.0/0" "TCP"]
24          removeDBAccess ls = concat (map (\(FRule o from to ip pr) -> if ((from <=
                Just 3306) && (to >= Just 3306)) then [FRule o from (Just 3305) ip pr,
                FRule o (Just 3307) to ip pr] else [(FRule o from to ip pr)]) (filter
                (\(FRule _ from to _ _) -> not ((from == Just 3306) && (to == Just
                3306))) ls))
```

Listing 3.36: Analyse of the Firewall view : addSSH

According to the integers sent by the analysis, the firewall rules of the currently treated security group are handled. This is a very simple strategy, but it is a basis for a rudimentary security. Of course, it is easy to add more policies depending on the user. In this implementation and for our evaluation of the model, this elementary plan is enough.

3.6 The Cost View

The cost view, related to the Modifiability concern, did not exist in the original hierarchical self-adaptations model [29]. On the top of the fact that the cost is one of the most important concern for compagnies, it is particularly interesting to add it to our model because the view shares a lot of data with the redundancy view and the auto-scaling view. Considering that the purpose of this thesis is to suggest a multi-viewing data sharing system, it is worth to possess at least three views that will enter into conflict, to show the real added-value of our approach. The order in which the views are executed will define which one takes the precedence over the others. The most important will be the last, that overwrites the changes of the other views. As usual, after explaining the data structure, the bidirectional transformation between the source and the cost view is explained. The analysis and planning steps end this section.

3.6.1 Cost View - Data structure

Here is the schema of the cost view, still following the same structure as the source in Section 3.2.1:

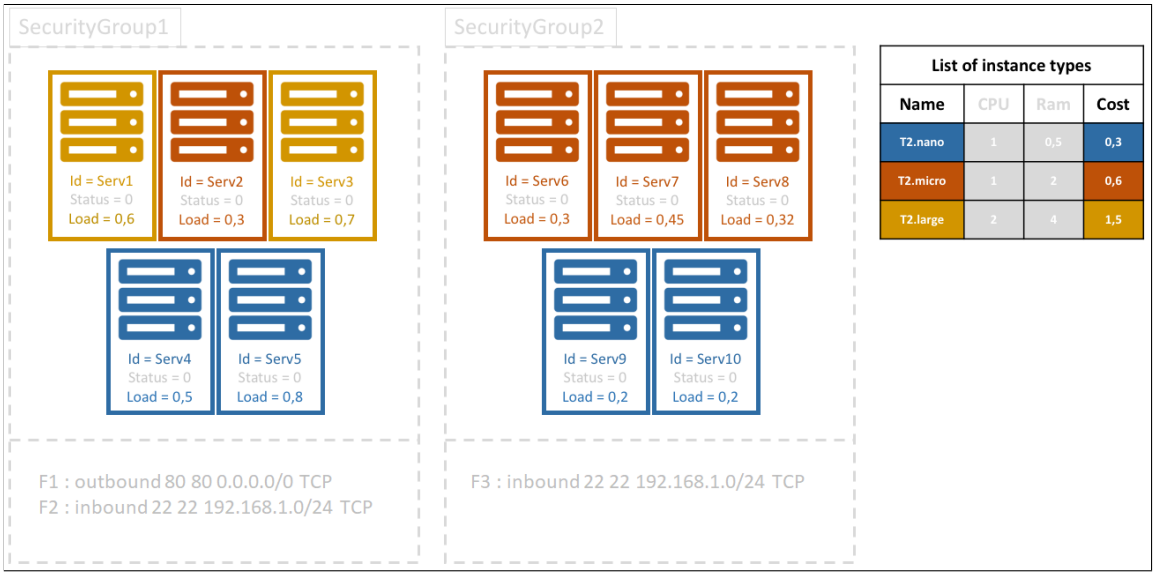


Figure 3.6: Cost view

Clearly, the only attribute taken into account is the cost of each instance. Based on this sole element, the view will start or shut down some instances. The code of this view is:

```
data CInstance = CInstance Id Type Load
    deriving (Show, Eq)
deriving instance NFData CInstance
deriveBiGULGeneric ''CInstance
deriveJSON defaultOptions ''CInstance

data CInstanceType = CInstanceType Id TypeCost
    deriving (Show, Eq)
deriving instance NFData CInstanceType
deriveBiGULGeneric ''CInstanceType
deriveJSON defaultOptions ''CInstanceType

data CView = CView [CInstance] [CInstanceType]
    deriving (Show)
deriving instance NFData CView
deriveBiGULGeneric ''CView
deriveJSON defaultOptions ''CView
```

Listing 3.37: Data structure of the Cost view

In addition to the type attribute, the load has also been kept. This attribute is taken into account in the analysis and planning steps of the cost view when servers of the same price are compared.

The attentive reader may have noticed that the *State* and *Status* attributes, located in the instances definitions, are present in every view, except the cost view. As the views are going to launch (or create) and to shut down (or delete) instances, it is mandatory to know if the instance is running or not. While the cost view is going to shut down instances as well, it does not need those attributes because only the running instances will be picked up from the source to this view, all the others will not appear. The state and the status attributes are thus useless for this specific view.

3.6.2 Cost View - Bidirectional transformation

The bidirectional transformation between the *cost view* and the source is implemented like this:

```
costUpdate :: BiGUL Source CView
costUpdate =
    $(rearrS [| \(Source insts sgs instTypes)  -> (insts,instTypes,sgs) |])$
    $(rearrV [| \(CView insts instTypes) -> (insts,instTypes,()) |])$
      $(update
        [p| (insts,instTypes,sgs) |]
        [p| (insts,instTypes,sgs) |]
        [d| insts = alignInstances; instTypes = alignInstanceTypes; sgs = Skip (
            const ()) |])
```

Listing 3.38: costUpdate

Once again, the list of security groups is not required for this view. As a matter of fact, the only interesting attributes for the view to calculate the total cost of the system are the type of each instance, and the cost associated to each type. Thus, this BiGUL function reorganizes the source and the view so that the list of instances and the list of instance types can be at the same position in the tuple. After that, the *alignInstances* and *alignInstanceTypes* functions handle the synchronization between the lists of instances and the lists of instance types respectively.

```
alignInstances :: BiGUL [Instance] [CInstance]
alignInstances = align
    (\(Instance _ _ _ state status _ _) ->
    (state == 16 && status /= 2) ||
    (state /= 16 && status == 1))
    (\(Instance s _ _ _ _ _ _) (CInstance v _ _) -> s == v)
    $(update
        [p| Instance identifier instType _ _ _ _ instLoad |]
        [p| CInstance identifier instType instLoad |]
        [d| identifier = Replace; instType = Replace; instLoad = Replace |])
    (\(CInstance vId vType vLoad) -> Instance vId vType "" 0 1 "" vLoad)
    (\(Instance sId sType sAmi sState _ sSecurityGroupRef sLoad) -> Just (Instance
        sId sType sAmi sState 2 sSecurityGroupRef sLoad))
```

Listing 3.39: Cost - alignInstances

As usual, the basic *align* function of BiGUL is used. The first argument filters the element in the source and sends to the view only those which satisfy the condition. The aim of the cost view is to calculate the whole cost of the system. There are two categories of instances that cost money. Either the instances are currently running (i.e. state equals to 16) and previous concerns did not extinguish them (i.e. status not equals to 2) or they are currently not running (i.e. state not equals to 16) but previous concerns ask to start them (i.e. status equals to 1). Indeed, if a previously executed view wants to shut down an instance, it must be taken into account by the cost view, and the associated cost of this instance must not be into the whole future cost. However, an instance with a status of 1 means that a previous view (and so, with a smaller priority) wants to start a stopped instance. Even if the instance is not currently running, its cost must be calculated because it will be launched by the AWS API at the *execute* step.

The second argument of the *align* function matches the element in the source with the element in the view, thanks to their identifier. The third argument is the update function.

As the other update functions, it only uses the BiGUL's *Replace* function to synchronize the attributes on both sides. The fourth argument describes how to create an existing instance in the view, but not in the source, by giving basic values to the constructor. Finally, the last argument shows how to delete an instance in the source when it is not existing in the view, just by putting its status to 2.

```
1  alignInstanceTypes :: BiGUL [InstanceType] [CInstanceType]
2  alignInstanceTypes = align
3       (\_ -> True)
4       (\(InstanceType s _ _ _) (CInstanceType v _) -> s == v)
5       $(update
6          [p| InstanceType identifier _ _ typeCost |]
7          [p| CInstanceType identifier typeCost |]
8          [d| identifier = Replace; typeCost = Replace |])
9       (\(CInstanceType vId vTypeCost) -> InstanceType vId 0 0 vTypeCost)
10      (\_ -> Nothing)
```

Listing 3.40: Cost - alignInstanceTypes

For the last time, the *align* function is employed. The first argument always returns true, because all the instance types must exist in the view, in order to find the cost of the running instances. The second argument links the element in the source with those in the view. The third is the common update function, which synchronizes the attributes on each side. The fourth argument shows how to create an instance present in the view but not in the source. Ultimately, the last argument returns *Nothing* because it is not permitted to delete any instance types in the view. Basically, both last parameters are not used because the instance type is provided by AWS and it is not possible to create or delete one of them. However, to have a lens, the transformation must be well-behaved. We thus have to handle the cases where we remove or create an instance type, even if finally during the execute step of the system those changes are not taken into account.

3.6.3 Cost View - Analysis and Planning

The code which analyses the *Cost View* is quite simple:

```
1  costAnalysis :: CView -> Double
2  costAnalysis (CView instances instanceTypes) =
3      sum (map (\(CInstanceType ident cost) -> calculateCost instances ident cost)
           instanceTypes)
4
5  calculateCost :: [CInstance] -> String -> Double -> Double
6  calculateCost instances typeName cost =
7          (fromIntegral (length (filter (\(CInstance _ typeInstance _) -> typeName ==
               typeInstance) instances))) * cost
```

Listing 3.41: Analyse of the Cost view

The idea behind this code is as follows. First, we want to transform each *CInstanceType* into an integer, which represents the cost of the *CInstances* of this type, in order to *sum* this new list to obtain the total cost of the system. To do so, the *map* is used to apply the *calculateCost* function to every *CInstanceType*.

The *calculateCost* function filters all the *CInstances* of the system, and keeps only the one whose type are the same as the type currently examined. Once the list only contains

the right *CInstances*, the last thing to do is taking its length, and multiply it with the right cost.

Now that the analysis has returned the cost of the system, the *Planning* can adjust the number of instances, to remain below a limited price defined by the user.

```
1  costPlan :: Double -> CView -> CView
2  costPlan limitCost cView = handlePlan (costAnalysis cView) limitCost cView
3
4  handlePlan :: Double -> Double -> CView -> CView
5  handlePlan currentCost limitCost (CView instances instanceTypes)
6      | currentCost > limitCost = handlePlan (costAnalysis newIntermediateSource)
           limitCost newIntermediateSource
7      | otherwise = CView instances instanceTypes
8          where
9              newIntermediateSource = CView sortedList instanceTypes
10
11             sortedList = tail (sortBy sortByCost instances)
12
13             sortByCost :: CInstance -> CInstance -> Ordering
14             sortByCost (CInstance _ type1 load1) (CInstance _ type2 load2)
15                 | load1 < load2 = LT
16                 | load1 == load2 = case
17                     ((find (\(CInstanceType typeName1 _) -> typeName1 == type1)
                           instanceTypes),
18                     (find (\(CInstanceType typeName2 _) -> typeName2 == type2)
                           instanceTypes))
19                     of
20                         (Just (CInstanceType _ cost1), Just (CInstanceType _ cost2)
                               ) -> if cost1 > cost2 then LT else GT
21                         (Nothing,_) -> error ("Source inconsistent")
22                         (_,Nothing) -> error ("Source inconsistent")
23                 | otherwise = GT
```

Listing 3.42: Planning of the Cost view

Obviously, the *Planning* only has to change something if the cost calculated by the analysis is above the limit. Otherwise, it simply returns the unchanged view. In case of high cost, the list of *CInstances* in the view must be reduced. The strategy is to remove the head of the list, after having sorted it. The *sortByCost* function ensures that its output will follow the rules describe within. The first rule puts first the little used, because removing one whose load is high could be problematic for the system. Then, if the load attribute is the same, it puts the most expensive before. The function which removes the head is applied recursively until the current price is below the limit. This strategy assures that the most used instances run as long as possible, and if they have to be shut down, the most expensive is first removed. This view does not care about the number of CPU's, because it is the main purpose of the *auto-scaling view*, and it is the whole point of separating the concerns in different views.

This ends the explanations of the propagation of the changes, made inside one view, into the source and propagated to the other views thanks to the four bidirectional transformations. It is now possible to make a change in one of the modules, sends this change back to the source with a *backward transformation*, and the source eventually propagates the change to the other modules with the *forward transformations*.

3.7 The Amazon Web Services API

We have presented all the modular systems, each in charge of its analysis and planning (AP). Let us introduce the monitor (M) and execute (E) steps, which will end the explanation of the first MAPE loop. As a reminder, our use case manages a cloud infrastructure and we chose to work with Amazon Web Services (AWS) because it is one of the most used IaaS around the world. AWS is an Infrastructure as a Service (IaaS) offering cloud computing. Basically, it allows the user to rent servers and plenty of services that we could need to manage a cloud infrastructure. The other reason why we chose to use AWS is the fact that a lot of other works use it. That makes the comparison easier. Typically, they are two ways to manage our infrastructure with AWS: either we use the web interface to run, stop or buy something; or we can use the AWS API [3], that allows us to make requests when we want to do something in our infrastructure. For our automatic system, the API is better suited. Thus, we use the AWS API to monitor and execute. As shown all along this section, we use tools to facilitate the communication of our system with AWS API.

3.7.1 AWS API - Monitoring

The monitoring step consists of retrieving all the information about our infrastructure. In other words, the monitor creates the source (explained in section 3.2). Because there is no fairly complete Haskell API available, to handle this task, our system requires to be run on a machine with the AWS API CLI[4] installed, which is a tool to manage your AWS services. Thanks to it, we can run simple console commands to interact with the AWS API. We will not highlight the installation of those tools here but you can find our working system with some documentation on the github. Moreover, you also can find some installation instructions according to your operating system in the official AWS API CLI documentation. We will not detail how to obtain Amazon API credentials information. Therefore, we assume that the system in which our system is running has AWS API CLI installed and that you have all the information in order to be connected to Amazon Web Services.

3.7.1.1 Communicate in JSON

As you will see during the configuration of AWS Cli, we want it to communicate in JSON. When we query AWS for the information about our infrastructure, it retrieves this information in this format. We need to translate this data in Haskell to work on it. To do that, we use a Haskell library called *Aeson*[5]. To make it easy, this library allows us to create Haskell data and making a simple Haskell derivation or instantiation to work with JSON. Typically, there are two Haskell instances for a Data type. One instance to specify how we can go from Haskell data to JSON string and another to go from JSON string to Haskell data. These two instances are respectively called *ToJson* and *FromJson*. In their basic use, both are simple to understand. A little example using a Haskell derivation will explain each of them. Obviously, we can override the default behavior to handle specific cases. In this thesis, we only explain what we have done with this library. The rest of the library is irrelevant for our use.

The translation of Haskell data into JSON is the first illustrated. Here is a simple example, with the data of a person.

```
1 {-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
2 import GHC.Generics
```

[3] http://docs.aws.amazon.com/cli/latest/index.html (Accessed on 04/22/2018)
[4] https://aws.amazon.com/cli/ (Accessed on 04/22/2018)
[5] https://hackage.haskell.org/package/aeson (Accessed on 04/22/2018)

```haskell
import Data.Aeson

data Person = Person {
  name :: String,
  age  :: Int }
  deriving (Generic, ToJSON, FromJSON)

me :: Person
me = Person {name="Quentin", age="24"}

main = encode me // return this string : {"name":"Quentin","age":24}
```

Listing 3.43: Simple Example of JSON encode using Aeson

Thanks to *Aeson*, we just had to declare a Haskell data deriving *ToJSON* to be ready to encode data in JSON. That is what the main function shows. Note that we had to specify some directives to let the compiler derive any class in Generic, ToJSON and FromJSON. It is quite easy to transform Haskell data in JSON using a simple derivation.

However, we do not need that in this chapter but in the following one. In this chapter, we only need the other way, from JSON to Haskell. Translating data in JSON is as easy as in Haskell data. The trick is having a Haskell data with the same structure as the JSON. It means that the names of JSON labels have to be the same as in the Haskell data type. The type of each JSON data must also be consistent with the Haskell data type. In our example, the generated JSON has exactly the same structure. We can then directly use the *decode* function to get Haskell data. The following code shows this situation:

```haskell
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
import GHC.Generics
import Data.Aeson

data Person = Person {
  name :: String,
  age  :: Int }
  deriving (Generic, ToJSON, FromJSON)

json :: String
json = "{\"name\":\"Quentin\",\"age\":24}"

main = decode $ json :: Maybe Person // return : Just Person {name="Quentin", age="
    24"}
```

Listing 3.44: Simple Example of JSON decode using Aeson

The example above uses the same data as the first one, but in the opposite way. We just have to use the *decode* function on a JSON string to transform it in a Haskell data. The function returns a *Maybe* to handle errors and returns Nothing when something bad happened (for example, a wrong JSON format). Of course, more complete data can be handled with the API. That is why it is possible to override the default behavior of *FromJson*. With AWS, we had to do this because the API retrieves data with the identifier in CamelCase (capitalizing the first letter of each word), whereas our naming convention is lowerCamelCase (capitalizing the first letter of each word except the first one). In our example, instead of our JSON in lower case, we would have *"Name":"Quentin","Age":24*. Our data declaration to handle this specific problem would be the following one:

```haskell
capitalized :: String -> String
capitalized (x:xs) = Char.toUpper x : xs
```

```
3  capitalized [] = []
4
5  data Person = Person {
6    name :: String,
7    age  :: Int }
8    deriving (Generic, ToJSON)
9  instance FromJSON AWSInstance where
10   parseJSON = genericParseJSON defaultOptions {
11               fieldLabelModifier = capitalized }
```

Listing 3.45: Example of JSON decode using Aeson

In this way, before trying to match the JSON with our data, *Aeson* modifies the label of our data with the *capitalized* function that capitalizes the first letter. The last things to highlight before explaining the monitoring is that the translation between JSON and Haskell is recursive. It means that if we have data containing other data, Aeson will parse those data recursively. Moreover, it is not required to specify all the data of the JSON. If a part of the data is not needed, a field that can be matched can be skipped. For example, the JSON *"Name":"Quentin","Age":24, "sex":"male"* will give exactly the same result as the Listing 3.44.

3.7.1.2 Retrieving data from AWS

Let's explain the monitor step. To communicate with the AWS API, some credentials must be specified. Thus, the monitor function takes three arguments. The first two are respectively the AWS access key part and the AWS secret access key of the credentials. The last one is the AWS region where the system is executed. Then the call of the *monitor* function with those three arguments retrieves a source representing our current AWS infrastructure. Before explaining, line by line, the code created to monitor, it is important to reinforce that when this thesis was written, AWS did not provide a way to get the information about the type of available instances in a specific region. For that reason, we had to hard-code a list of available instance types in the region where we ran the system, during our internship in Tokyo. It is not a complete list, but only the list of instance types that we use.

```
1  instanceTypes :: [InstanceType]
2  instanceTypes = [
3      InstanceType "t2.nano" 1 0.5 0.0058,
4      InstanceType "t2.micro" 1 1 0.0116,
5      InstanceType "t2.small" 1 2 0.023,
6      InstanceType "t2.medium" 2 4 0.0464,
7      InstanceType "t2.large" 2 8 0.0928,
8      InstanceType "t2.xlarge" 4 16 0.1856,
9      InstanceType "t2.2xlarge" 8 32 0.3712]
```

Listing 3.46: List of instance types used

```
1  monitor :: String -> String -> String -> IO Source
2  monitor access secret region = do
3      -- configure AWS
4      _ <- readProcess "aws" ["configure"] (access ++ "\n" ++ secret ++ "\n" ++
           region ++ "\n" ++ "json\n")
5
6      -- retrieve instances
7      resInstances <- readProcess "aws" ["ec2", "describe-instances", "--filter", "
           Name=tag:BiGUL,Values=CloudBx"] []
8      jsonInstances <- return $ fromJust (decode (Char8.pack resInstances) :: Maybe
           AWSDescribeInstancesResponse)
```

```
9     instances <- updateLoadOfInstances (fromAWSToSourceInstances jsonInstances)
10
11    --retrieve security Groups
12    resSG <- readProcess "aws" ["ec2", "describe-security-groups", "--filter", "
          Name=tag:BiGUL,Values=CloudBx"] []
13    jsonSG <- return $ fromJust (decode (Char8.pack resSG) :: Maybe
          AWSDescribeSecurityGroupsResponse)
14    securityGroups <- return (linkInstancesToSecurityGroup instances (
          fromAWSToSourceSecurityGroups jsonSG))
15
16    return $ Source instances securityGroups instanceTypes
```

Listing 3.47: Function of monitoring

The heart of the algorithm is now explained. As already said, the first thing to do is to specify the credentials information to the AWS API Cli. The first line of code (line 4) accomplishes that. We use the function *readProcess* provided by the *system.Process* library of Haskell [6]. This function allows the developer to execute a command just like inside a terminal. It takes three parameters. The first one is the file path of the executable, in our case the executable can be called directly as "aws". The second parameter is an array of Strings, corresponding to the arguments given to the executable. The last parameter is the standard input corresponding to the information that the developer would have written with the keyboard, if this command was executed in the terminal. This listing represents this command, executed by the function *readProcess*.

```
1 $ aws configure
2     AWS Access Key ID [None]: <access>
3     AWS Secret Access Key [None]: <secret>
4     Default region name [None]: <region>
5     Default output format [None]: json
```

This command returns nothing relevant. Thus, we do not keep the result of the *readProcess* function. The new line (line 7) runs a command like the previous one. The program remains the same, but the parameters differs. It uses the EC2 part of the AWS API Cli. EC2 is the name given by AWS to their cloud computing service. The command asks then to describe the instances of the AWS infrastructure. However, to avoid any side effect on the infrastructure, we filter the instances to those with the tag *BiGUL=CloudBx*. In this way, only these will be managed by the system. It gives to the user the possibility to manage only a part of the infrastructure, without any effect on the other resources. In our system, the instances without this tag are hidden. The line can be translated in bash as follows:

```
1 $ aws ec2 describe-instances --filter "Name=tag:BiGUL,Values=CloudBx"
2     [JSON]
```

As shown in the official documentation[7], the response to this command is a JSON (because during the configuration step, line 4 of Listing 3.47, we asked to communicate in this format) containing a lot of information about the current situation of our instances. We had created specific Haskell data matching with the JSON retrieved, to manipulate that information. To clarify the code here, we removed the implementation of *FromJSON* for all the data. It is significantly the same as in the Listing 3.45, because we have to capitalize the first letter of the field to be consistent with our naming convention.

[6] https://hackage.haskell.org/package/process-1.6.3.0/docs/System-Process.html (Accessed on 04/22/2018)

[7] https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instances.html (Accessed on 04/22/2018)

```
1  data AWSDescribeInstancesResponse = AWSDescribeInstancesResponse
2      { reservations :: [AWSReservation]
3      } deriving (Show, Generic)
4  instance FromJSON AWSDescribeInstancesResponse where [...]
5
6  data AWSReservation = AWSReservation
7      { instances :: [AWSInstance]
8      } deriving (Show, Generic)
9  instance FromJSON AWSReservation where [...]
10
11 data AWSInstance = AWSInstance
12     { instanceId :: String,
13     instanceType :: String,
14     state :: AWSState,
15     securityGroups :: [AWSInstanceSecurityGroup],
16     imageId :: String
17     } deriving (Show, Generic)
18 instance FromJSON AWSInstance where [...]
19
20 data AWSState = AWSState
21     { code :: Int,
22     name :: String
23     } deriving (Show, Generic)
24 instance FromJSON AWSState where [...]
25
26 data AWSInstanceSecurityGroup = AWSInstanceSecurityGroup
27     { groupName :: String,
28     groupId :: String
29     } deriving (Show, Generic)
30 instance FromJSON AWSInstanceSecurityGroup where [...]
```

Listing 3.48: Data of instances - AWS API

The code above is explained here. Where we ask the description of instances, AWS API returns a list of reservations (line 1). A reservation is composed by instances reserved at the same time. Then, those reservations contain a list of instances (line 6) with several useful properties (line 11). An instance has two sub data type, corresponding to the state of the instance (line 20) and the information about the security groups (line 26).

We have the data corresponding to the JSON retrieved by AWS API, the line 8 of the Listing 3.47 decodes it in the Haskell type AWSReservation. In addition to the decode function, we use the *pack* function to properly handle the UTF-8 encoding. The next line uses two functions: the first one, called *fromAWSToSourceInstances*, translates the AWS Data into our source data (section 3.2). The second one is the function called *updateLoadOfInstances* that retrieves the load of each instance. Indeed, this information is not provided when we ask the details about instances. The code below explains both functions:

```
1 fromAWSToSourceInstances :: AWSDescribeInstancesResponse -> [Instance]
2 fromAWSToSourceInstances d = Prelude.foldl (\acc o -> acc ++ (createInstance o) )
      [] (reservations d)
3     where
4         createInstance :: AWSReservation -> [Instance]
5         createInstance o = L.map (\inst -> Instance (instanceId inst) (instanceType
              inst) (imageId inst) (code (state inst)) 0 (instSg inst) 0) (instances
              o)
6         instSg :: AWSInstance -> String
7         instSg o    | L.length (securityGroups (o)) > 0 = (groupName (L.head (
              securityGroups (o))))
8                     | otherwise = ""
```

Listing 3.49: Translate AWS data into a Source

Thus, *fromAWSToSourceInstances* goes through all the AWS reservations with a *foldl* (line 2). It merges the current accumulator, initialized as an empty list of instances, with the results of the sub function *createInstance*, which is a list of instances contained in the reservation. This latter uses a *map* to generate this list of instances, based on the AWS data.

```
1
2  data AWSGetMetricsResponse = AWSGetMetricsResponse
3      { datapoints :: [AWSDatapoint]
4      } deriving (Show, Generic)
5  instance FromJSON AWSGetMetricsResponse where [...]
6
7  data AWSDatapoint = AWSDatapoint
8      { timestamp :: String,
9      average :: Double,
10     unit :: String
11     } deriving (Show, Generic, Eq)
12 instance FromJSON AWSDatapoint where [...]
13
14 updateLoadOfInstances :: [Instance] -> IO [Instance]
15 updateLoadOfInstances instances = do
16     res <- foldM fn [] instances
17     return res
18     where
19         fn :: [Instance] -> Instance -> IO [Instance]
20         fn acc (Instance identifier insttype ami state status sg _)
21             | state == 16 = do
22                 currentTime <- getCurrentTime
23                 timeEnd <- getCurrentTime
24                 timeStart <- return (addUTCTime (-3600) currentTime)
25                 r <- readProcess "aws" ["cloudwatch","get-metric-statistics", "--
                       namespace", "aws/EC2", "--metric-name","CPUUtilization", "--
                       statistics","Average", "--dimensions", "Name=InstanceId,Value="
                        ++ identifier, "--start-time", (iso8601 timeStart), "--end-
                       time", (iso8601 timeEnd), "--period","300"] []
26                 son <- return $ fromJust (decode (Char8.pack r) :: Maybe
                       AWSGetMetricsResponse)
27                 mostRecent <- return (getMostRecent (datapoints json))
28                 return (case mostRecent of
29                     Just x -> (acc ++ [Instance identifier insttype ami state
                           status sg ((average x)/100)])
30                     Nothing -> (acc ++ [Instance identifier insttype ami state
                           status sg 0]))
```

```
31              | otherwise = do
32                  return (acc ++ [Instance identifier insttype ami state status sg
                        0])
33          getMostRecent :: [AWSDatapoint] -> Maybe AWSDatapoint
34          getMostRecent x = L.foldl (\acc o -> if acc == Nothing then Just o else if
                (parseISO8601 (timestamp o)) > (parseISO8601 (timestamp (fromJust acc))
                ) then Just o else acc) Nothing x
35
36  iso8601 :: UTCTime -> String
37  iso8601 = formatTime defaultTimeLocale "%FT%T%QZ"
38
39  parseISO8601 :: String -> Maybe UTCTime
40  parseISO8601 t = parseTimeM True defaultTimeLocale "%FT%T%QZ" t
```

Listing 3.50: updateLoadOfInstances

The *updateLoadOfInstances* function is quite long and we will not explain it line by line. Its role is to get the load of each instance. To do so, it goes through the list of instances and calls the cloudwatch part of the AWS Cli program (line 25). It then retrieves a list of loads (CPUUtilization) by period of 5 minutes (300 seconds) between the start time and the end time. As you can see at the line 24, the start time begins 1 hour earlier than the current time. Normally, we should have a list of 12 measuring points of the load (every 5 minutes). However, sometimes AWS does not start to watch the instance immediately. Moreover, after some experiments, we saw that if we put a smaller time interval than 1 hour AWS, AWS no longer gives any measurement points. Finally, we only keep the most recent data and update the current instance to generate a list of updated instances.

The list of instances and the list of instance types are now available. The last part to retrieve is the last part of the source: the list of security groups. The lines 12 and 13 of the monitor function (Listing 3.47) are respectively similar to the lines 7 and 8 previously explained, except that we ask to describe the security groups [8] tagged with BiGUL=CloudBx. Therefore, like the previous explanation, we use other Haskell data to handle the new JSON:

```
1   data AWSDescribeSecurityGroupsResponse = AWSDescribeSecurityGroupsResponse
2       { securityGroups :: [AWSSecurityGroup]
3       } deriving (Show, Generic)
4   instance FromJSON AWSDescribeSecurityGroupsResponse where [...]
5
6   data AWSSecurityGroup = AWSSecurityGroup{
7       groupName :: String,
8       description :: String,
9       ipPermissions :: [AWSRule],
10      ipPermissionsEgress :: [AWSRule]
11      } deriving (Show, Generic)
12  instance FromJSON AWSSecurityGroup where [...]
13
14  data AWSRule = AWSRule{
15      fromPort :: Maybe Int,
16      toPort :: Maybe Int,
17      ipRanges :: [AWSIpRange],
18      ipProtocol :: String
19      } deriving (Show, Generic)
20  instance FromJSON AWSRule where [...]
21
22  data AWSIpRange = AWSIpRange
```

[8] https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-security-groups.html (Accessed on 04/22/2018)

```
23      { cidrIp :: String
24      } deriving (Show, Generic)
25  instance FromJSON AWSIpRange where [...]
```

Listing 3.51: Data of security groups - AWS API

The next line of the Listing 3.47 uses the *fromAWSToSourceSecurityGroups* function that translates the AWS Data into our source data (section 3.2). It also uses the function called *linkInstancesToSecurityGroup* that links each instance with its securityGroup. Indeed, every instance knows which security is linked to it, but the security group does not have this information. Let us describe a little bit those both functions.

```
1  fromAWSToSourceSecurityGroups :: AWSDescribeSecurityGroupsResponse -> [
       SecurityGroup]
2  fromAWSToSourceSecurityGroups d = Prelude.foldl (\acc o -> acc ++ [SecurityGroup (
       groupName o) (description o) [] (rules o)]) [] (securityGroups d)
3      where
4          rules :: AWSSecurityGroup -> [FirewallRule]
5          rules o = L.map (fromAWSRuleToRules False) (ipPermissions o) ++ L.map (
               fromAWSRuleToRules True) (ipPermissionsEgress o)
6          fromAWSRuleToRules :: Outbound -> AWSRule -> FirewallRule
7          fromAWSRuleToRules outbound o = FirewallRule outbound (fromPort o) (toPort
               o) (cidrIp (L.head (ipRanges o))) (ipProtocol o)
```

Listing 3.52: Translate AWS security group data into a Source

The function above has the same purpose that the one described in Listing 3.49. It must transform the types (used to parse the AWS JSON) into a security group list, in order to complete the source of the system. Then, the function goes through the securityGroup given by AWS, with a *foldl*, and it extracts some information (group name, description and rules) to create a new list of SecurityGroup, compliant with our source. The sub-function *rules* generates a list of firewall rules used by the current securityGroup. Clearly, when the function creates the new securityGroup, it puts the list of instances as empty (third parameter of the data *SecurityGroup*). Indeed, the data retrieve by AWS do not contain any information about the instances linked with the securityGroup. That is why we need to use another function, called *linkInstancesToSecurityGroup*, and described below:

```
1  linkInstancesToSecurityGroup :: [Instance] -> [SecurityGroup] -> [SecurityGroup]
2  linkInstancesToSecurityGroup [] sgs = sgs
3  linkInstancesToSecurityGroup ((Instance identifier _ _ _ _ sgIden _):xs) sgs = case
       L.find (\(SecurityGroup iden _ _ _) -> iden == sgIden ) sgs of
4      Just sg@(SecurityGroup iden desc insts rules) -> linkInstancesToSecurityGroup
           xs ((SecurityGroup iden desc (identifier:insts) rules):(L.filter (\(
           SecurityGroup iden _ _ _) -> iden /= sgIden ) sgs))
5      Nothing -> linkInstancesToSecurityGroup xs sgs
```

Listing 3.53: Translate AWS security group data into a Source

The *linkInstancesToSecurityGroup* function is used to go through all the instances and checks what security group is linked to it. When this security group is found, the function updates it by adding the current instance in its list of instances.

Now that each part of the source is computed, the function has to create a complete *Source*. The last line of the monitor function (Listing 3.47) does that, by returning a Source containing the information received.

3.7.2 AWS API - Execution

The execution step of the MAPE loop consists of executing something to update the handled system, according to the changes made by the analysis and planning steps. In our case, after running our multiple subsystems, we had to communicate with AWS to create, stop, remove or update instances or security groups. To achieve this task, two choices are possible: using the API of AWS, or using a tool that makes the execution easier. We chose to use a tool called Ansible [9]. It is a software that automates software provisioning, configuration management, and application deployment [1]. To simplify and summarize, it saves us from making calls to the API manually. Its configuration files used to deploy the infrastructure are in YAML (Yet Another Markup Language) format, much more readable than XML, CSV or JSON. The use of Ansible greatly facilitates the deployment of the changes. The execute step simply generates, from the updated source, the YAML files which are run by Ansible, to adapt the information on AWS. You can find an example of YAML configuration file in Appendix A

The thesis is not intended to explain the installation of tools such as Ansible. Thus, we suppose that, all along this section, Ansible is correctly installed and configured. You can easily find the installation and configuration instructions on the official documentation of the tool[10]. Let us detail how the execute function works in our system.

```
1 executeAWS :: String -> String -> String -> String -> String -> String -> Source ->
      IO ()
2 executeAWS access secret region key pair image lb src = do
3     template <- return (generateMainTask access secret region key pair image lb)
4     tmpFilePath <- writeSystemTempFile "ansible.yaml" (template ++ (
          sourceToAnsibleTasks src))
5     _ <- readProcess "ansible-playbook" [tmpFilePath] []
6     return ()
```

Listing 3.54: Execution function

Remember that the goal of this function is to create a configuration file readable for Ansible, and to execute it on our infrastructure. The *executeAWS* function takes 6 parameters. The first two are respectively the AWS access key part and the AWS secret access key of your credentials, just like the *monitor* function. The third parameter is about the region of the infrastructure. The fourth one is the name of the key pair used to accomplish some tasks. This key pair must be configured on AWS, during the configuration of Ansible. It is used while an instance is created. The fifth parameter is the OS image name, also used when a new instance is created. The sixth one is the identifier of the load balancer, used for new instances. Finally, the last one is the updated source. Obviously, this function as the *monitor* function is just an example. Of course, we can create a much more complicated function to handle different situations.

```
1 generateMainTask :: String -> String -> String -> String -> String -> String ->
      String
2 generateMainTask access secret region key pair image lb = "- name: Main task\n" ++
3  indent ("gather_facts: False\nhosts: localhost\nvars:\n" ++
4     indent ("key pair: "++ key pair ++
5         "\nimage: "++ image ++
6         "\nregion: "++ region ++
7         "\naws_access_key: "++ access ++
8         "\naws_secret_key: "++ secret ++
```

[9] https://www.ansible.com/ (Accessed on 04/22/2018)

[10] http://docs.ansible.com/ansible/latest/scenario_guides/guide_aws.html#introduction (Accessed on 04/22/2018)

```
9         "\naws_load_balancer: "++ lb ++
10        "\naws_tags: {\"BiGUL\":\"CloudBx\"}\n") ++
11    "tasks:\n")
```

Listing 3.55: Execution function

The first line of the *executeAWS* function creates a basic template of the Ansible file, by calling the *generateMainTask* function already described above. This function is not really complicated. It creates a correctly indented string, corresponding to variables used all along the file. Those variables represent the parameters of the execute function.

```
1  - name: Main task
2      gather_facts: False
3      hosts: localhost
4      vars:
5          key pair: <key pair>
6          image: <image>
7          region: <region>
8          aws_access_key: <access>
9          aws_secret_key: <secret>
10         aws_load_balancer: <lb>
11         aws_tags: {"BiGUL":"CloudBx"}
12     tasks:
```

Listing 3.56: "Ansible file generated (Part 1)"

This default template is generated. The execution part has to create Ansible tasks depending on the updated source. The second line of code achieves that goal. First, this line calls *sourceToAnsibleTasks*, explained below. Then, it merges the result with the default template to generate the content of the Ansible file. Finally, it creates a temporary file with the Haskell *writeSystemTempFile*. The *sourceToAnsibleTasks* function is now explained. Because this function is quite long, it will not be fully described, but we will explain the outlines with only a part of the code.

```
1  sourceToAnsibleTasks :: Source -> String
2  sourceToAnsibleTasks (Source instances sgs _) =
3      securityGroupsStr ++
4      instancesToRunStr ++
5      newWebInstancesStr ++
6      newOtherInstancesStr ++
7      idsInstancesToStopStr
8      where
9          [...]
10         securityGroupsStr = L.concat (
11             L.map (\sg ->
12                 (indentN 2 ("- name: Security group\n" ++
13                     (indent ("ec2_group:\n" ++
14                         (indent (securityGroupToAnsibleTask sg)))))) sgs)
15         [...]
```

Listing 3.57: sourceToAnsibleTasks

This function is composed of several sub-functions, that create the Ansible tasks useful to update the infrastructure according to the source. As their names suggest it, the first function creates the tasks relative to the securityGroup. Then, it creates the tasks to run the existing instances. After, the new instances are created and finally, the instances to delete are shut down. Pay attention, there are two functions to create new instances. Indeed, as you will see during the experimentation (chapter 5), the use case has two security groups : one for the databases and another one for the web workers. the *newWebInstancesStr* function creates instances related to the web by linking them to a load balancer. The *newOtherInstancesStr*

function creates other instances. To summarize, typically the infrastructure has several web workers and one or two fixed instances of the database. To distribute the requests between all the web workers, we use a load balancer. Each of these functions returns an indented string to complete the Ansible file. For our explanation, the generation of the tasks related to the security group will be shown. The other function works quite in the same way. We create a new task called "Security group" with "ec2 group" as type and for each *securityGroup*. Then, we call the *securityGroupToAnsibleTask* function to generate this task.

```
1  securityGroupToAnsibleTask :: SecurityGroup -> String
2  securityGroupToAnsibleTask (SecurityGroup name description instsRef fwRules) =
3      "name: " ++ name ++ "\n" ++
4      "description: "++ description ++"\n" ++
5      "tags: \"{{aws_tags}}\"\n" ++
6      "aws_access_key: \"{{aws_access_key}}\"\n" ++
7      "aws_secret_key: \"{{aws_secret_key}}\"\n" ++
8      "region: \"{{region}}\"\n" ++
9      "rules: \n" ++
10     indent(L.concat (L.map rulesToAnsibleTask (fst (rules fwRules)))) ++
11     "rules_egress: \n" ++
12     indent(L.concat (L.map rulesToAnsibleTask (snd (rules fwRules))))
13     where
14         rules :: [FirewallRule] -> ([FirewallRule],[FirewallRule])
15         rules = L.foldl (\(inbound, outbound) rule@(FirewallRule x _ _ _ _) -> if x
               then (inbound, rule:outbound) else (rule:inbound, outbound)) ([],[])
16         rulesToAnsibleTask ::FirewallRule -> String
17         rulesToAnsibleTask (FirewallRule _ fromPort toPort ip protocol) =
18             "- proto: " ++ protocol ++"\n" ++
19             case fromPort of
20                 Just p -> "  from_port: "++ show p ++"\n"
21                 Nothing -> ""
22             ++
23             case toPort of
24                 Just p -> "  to_port: "++ show p ++"\n"
25                 Nothing -> ""
26             ++
27             "  cidr_ip: "++ ip ++"\n"
```

Listing 3.58: securityGroupToAnsibleTask

The function to create a securityGroup task is a little bit long, but not really complicated. First, we put the basic information about the current security group. Then, we add the firewall rule thanks to the *rulesToAnsibleTask* function. The inbounds have the prerogative over the outbounds. Finally, we obtain:

```
[...]
        - name: Security group
            ec2_group:
                name: <name>
                description: <description>
                tags: "{{aws_tags}}"
                aws_access_key: "{{aws_access_key}}"
                aws_secret_key: "{{aws_secret_key}}"
                region: "{{region}}"
                rules:
                    - proto: <protocol>
                        from_port: <port>
                        to_port: <port>
                        cidr_ip: <ip>
                rules_egress:
                    - proto: <protocol>
                        from_port: <port>
                        to_port: <port>
                        cidr_ip: <ip>
```

Listing 3.59: "Ansible file generated (Part 2)"

The rest of the code of *sourceToAnsibleTasks* works almost in the same way. The entire implementation can be found on GitHub. When the *sourceToAnsibleTasks* function has generated all the tasks, we merge it with the basic template already generated. Then, we store it into a temporary file. Finally, we call *ansible-playbook*, installed on our system, to run the newly generated file. In the end, our system has updated all the infrastructure according to the updated source.

Chapter 4

Second MAPE loop : Self-Prioritized Views using Bidirectional Transformations

This chapter is devoted to the second *MAPE loop*. Before explaining how this loop works, it's important to understand clearly why we need it. Thus, we begin with an explanation of the problem we want to solve in this part of our work. To do so, we use simple examples. After that, we explain the implementation of our solution and we illustrate this implementation with our main study case as we did during the chapter 3.

4.1 Necessity of a Synchronizer

As you already know, the aim of this work is to see whether the bidirectional transformations can be used to simplify a big system into several smaller ones. This modularization comes with several issues: how can we solve the conflicts between those subsystems? How can we give importance to a subsystem over the others? In a more general way, the main question is : how can we synchronize several subsystems avoiding conflicts?
All along this section, we will try to explain, step by step, the solutions we found to solve the above questions. In order to be as general as possible, let us forget for a moment the example system we have been using since the beginning of this thesis. Keep in mind what a typical modular system is (Figure 4.1). It contains a big system, working on a set of data, and a collection of smaller systems working on a subset of those data. Trivial examples are used to illustrate the issues of this section. It allows us to simply and intuitively show the problems we are trying to resolve with our synchronizer.

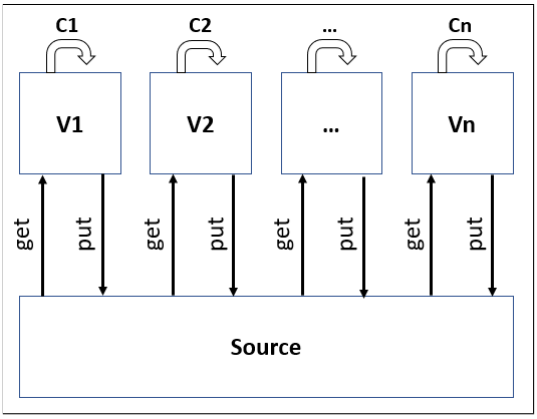


Figure 4.1: A typical modular system

As you will see, the solutions provided in this thesis to synchronize subsystems are relatively basics. Indeed, this research does not fit properly in a very specific field of research: it is complicated to find referencing articles or paper talking about how to avoid conflicts as we would like to do. Even when a paper uses a conflict resolution technique in a modular system, the explanation usually comes down to only a few sentences. This is difficult to use in our work and it is even rarer when it comes to using a synchronizer to avoid these conflicts. Moreover, most of the time, human intervention is needed to resolve the most delicate conflicts. That's why we decided to develop a solution that can serve as a basis for future works. Therefore, this part is based on basic intuitions and tries to find an intuitive solution to the problems that state, not the best one. Note also that, because of a lack of time, we could not continue our research on this subject. The section 6.3 lists the works which we thought relevant during the development session. Thus, this part of the work is essentially used to prune the elementary problems that a synchronizer produces.

However, even if we do not really use these works, it is important to note that basics notions exist about the conflict resolution. Indeed, these notions come from other fields of research, like the multithreading programmation. We can cite some notions like the mutual exclusion, deadlock and also a modular system which requires some kind of coordination between all submodules or a human intervention [36]. We can also think about version control systems to avoid conflicts but it is not really relevant in our situation because it requires merging potential conflicts and does not really provides a solution for the automatic resolution of those ones. It must be kept in mind that we do not want human intervention during the execution. We have to abandon the idea of merging models based on instructions given by the user. As explained in the future works, we could have tried to replace the human intervention by artificial intelligence techniques, but this was not our field of research. Instead, we decided to avoid conflicts by running the different subsystems one by one. This way of doing things allows subsystems to know the modifications made by the previous ones and therefore to react to a partially updated situation, contrary to the classical model merging which does not allow to know, even partially, the modifications of the other models.

4.1.1 Problem of subsystems conflicts

At this point of the research, we know how we can generate the subset of data and how to put them into the big system, thanks to BiGUL and its BXs. However, with multiple subsystems, everyone can work on one shared part of data. Thus, if several subsystems change a shared part, conflicts can appear. Let us imagine a simple situation where we manage printers of a company. We have two subsystems: the first one called *EnergySaverSystem* tries to save energy and stop each useless printer (ie. printers that did not print in the last hour

anymore); the second subsystem called *KeepAliveSystem* starts printers when a job is in the queue. We thus have the following information about printers:

```
{"printers" : [
        {
            "name" : "PrinterOne",
            "available_sheets" : 200,
            "state" : "stopped",
            "queued_printing" : 2,
            "number_printing_last_hour" : 5
        },{
            "name" : "PrinterTwo",
            "available_sheets" : 156,
            "state" : "stopped",
            "queued_printing" : 0,
            "number_printing_last_hour" : 6
        },{
            "name" : "PrinterThree",
            "available_sheets" : 563,
            "state" : "started",
            "queued_printing" : 1,
            "number_printing_last_hour" : 0
        }
]}
```

Listing 4.1: Data structure of the Source

The code above describes three printers. We have five attributes for each printer: the name, the number of sheets available, the state (started or stopped), the number of jobs in the queue and finally, the number of printings during the last hour. All of those are available for subsystems. The main source must be updated depending on the subsystems' results. For the *EnergySaverSystem*, we need information about the state and the number of printings during the last hour. The subset of data for this system is the following one:

```
{"printers" : [
        {"state" : "stopped","number_printing_last_hour" : 5},
        {"state" : "stopped", "number_printing_last_hour" : 6},
        {"state" : "started", "number_printing_last_hour" : 0}
]}
```

Listing 4.2: View of the first subsystem (*EnergySaverSystem*)

After its execution, the subsystem must stop the last printer and keep the first two in the same state. We have underlined the change. Now, we have the following data about printers:

```
{"printers" : [
        {"state" : "stopped", "number_printing_last_hour" : 5},
        {"state" : "stopped", "number_printing_last_hour" : 6},
        {"state" :  "stopped", "number_printing_last_hour" : 0}
]}
```

Listing 4.3: Updated view of the first subsystem (*EnergySaverSystem*)

For the second subsystem (*KeepAliveSystem*), we also need the state but we need the number of jobs in the queue. The subset of data for this system is the following one:

```
{"printers" : [
        {"state" : "stopped", "queued_printing" : 2},
        {"state" : "stopped", "queued_printing" : 0},
        {"state" : "started", "queued_printing" : 1}
]}
```

Listing 4.4: View of the second subsystem (*KeepAliveSystem*)

After its execution, the subsystem must start the first printer and keep the last one running. Thus, we have the following data:

```
{"printers" : [
        {"state" :  "started"], "queued_printing" : 2},
        {"state" : "stopped", "queued_printing" : 0},
        {"state" : "started", "queued_printing" : 1}
]}
```

Listing 4.5: Updated view of the second subsystem (*KeepAliveSystem*)

We now have to merge both the subsystems to update the shared source. How can we do that? The solution that we found is to associate a priority to each subsystem. In our example, the most important thing is to keep the printers started when a job is in the queue. The second subsystem (*KeepAliveSystem*) is more important than the other and when a conflict occurs, the result of the second subsystem (*KeepAliveSystem*) gets a higher priority. The updated source will be the following one, where we start the first and last printers and keep the second one stopped:

```
{
    "printers" : [
        {
            "name" : "PrinterOne",
            "available_sheets" : 200,
            "state" : "started",
            "queued_printing" : 2,
            "number_printing_last_hour" : 5
        },
        {
            "name" : "PrinterTwo",
            "available_sheets" : 156,
            "state" : "stopped",
            "queued_printing" : 0,
            "number_printing_last_hour" : 6
        },
        {
            "name" : "PrinterThree",
            "available_sheets" : 563,
            "state" : "started",
            "queued_printing" : 1,
            "number_printing_last_hour" : 0
        }
    ]
}
```

Listing 4.6: Printers data (Source updated)

4.1.2 Problem of subsystems' prioritization

Priority between subsystems is a way to solve conflicts. In section section 4.1 we saw that it is difficult to find lot of other works about synchronizer and there are unanswered questions, like the implementation of the prioritization in BiGUL. One solution of this question is quite easy and was already addressed in part at the end of the subsection 4.1.1. As we have seen in this section, the execution order of each subsystem is important because the last executed subsystem can modify the changes of the previous subsystems. Note as the system is modular, a subsystem may not be aware of the changes made by previous ones. Thus, the order of execution is even more important that the result of a subsystem can override previous ones without knowing. If we come back to the example of printers, we saw that the priority needed can be implemented by the execution order. Indeed, if we execute firstly the subsystem that stops all useless printers and update the source with the result and after that, we execute the second subsystem that starts printers with at least one job in its queue on the updated source, this second subsystem will overwrite the result of the previous one and have the priority.

Finally, the last question we have to answer is how to determine the priority of the subsystems. There are many ways to do this. The simplest is to fix an order and never change it. It is equivalent to hard-coding the sequence of execution of each subsystem. Even if this approach can be used, its inflexibility is a drawback. Indeed, there is lot of modular systems where the flexibility of the executions order is important. Let's show it on a simplified example. Suppose a system managing a hydraulic dam. The retention pond can not be drained because it also serves the reserve in case of drought. The hydraulic dam has at least 2 subsystems: one to manage the amount of electricity produced according to the demand, and another to manage the water level in the retention pond to stay above the limit. Even if the real system is more complexe and has more subsystems, we can imagine few cases to show why the flexibility is important in a modular system. First example, in case of a very dry summer, suppose that the needs of electricity is low. The system of water level management is more important than the quantity of electricity produced. In fact, we can suppose that it is more important to keep the level of the retention pond as high as possible to overcome a lack of water, and producing just the needed electricity. In the worst case, not enough electricity is fine, but water is vital. Second example, suppose that we are in winter and the needs of electricity is high because people needs to heat their homes. Generally during this season, it is often raining and there is no problem of water supply. Thus, the system of electricity management si more important than the level of water in the retention pond. Clearly, this kind of systems is more complexe but even with only two subsystems, the both previous examples show us the need of flexibility in the execution order.

Our idea is to make a self-adaptive system that changes the priority at runtime, depending on external and internal variables. Thus, we have chosen to name these variables a context and react according to it. Basically, the context is a list of key/value pairs that describe the real context in which the system is. It can contain what the users want. We can imagine an IOT system where the context contains the value of multiple sensors like the temperature, humidity and so on. Now that the context is defined, we have to determine the priority. There a two main approaches: one called rule-based and the second called goal-based.

The first one is quite simple. It is a list of rules that determines the priority. In English, we could have for example:

- If temperature is less than 20C°, then the first subsystem is more important then the others. In other words, it can override the change made by others.
- If temperature is less than 20C° and the humidity is more than 70%, then the second subsystem has the priority.
- If the time is between 7:00pm and 6:00am then the third subsystem is less important than others.

The second one is a bit more complicated. In this case, the goal is important and the order of the execution will depend on the context that we would like to have. For example, a robot that has to catch something on a table will generate the sequence of operations to accomplish this goal. Because we do not use this approach, we will not detail it more at this point.

Note that for our study, we have implemented a rule-based approach and a good future work could be the implementation of the goal-based approach. The reason why we choose this approach is simply the fact that, when we had to implement this part, the rule-based approach seems to be the best solution with the time we had left. This list of rules can be

viewed as a knowledge base where we can find the knowledge to prioritize the subsystems. However, the rules could be contradictory. For example, a rule that say :"if the temperature is above 20C° then the first subsystem is more important than others" and another rules say : "if the time is between 7:00pm and 6:00am then the third subsystem is more important than others" we have a contradictory. In fact, we did not investigate this point in detail and chose a simple solution. Indeed, as you will see deeper in subsubsection 4.2.1.5 we avoid this by defining three simple strategies. The first one assumes that the list of rules contains no contradiction. The second one removes contradictory rules by removing the oldest one when one conflict happens. The third one removes contradictory rules by removing the newest one.

4.1.3 Theoretical Solution of the Problems

At this point of the thesis, we would like to generalize the use of all previous concepts to create a system that works in as many situations as possible. Here is what we have in a typical modular system. As we already said, the main system is composed of data and each subsystem can read and/or write a part of these data. To go from the main system to the subsystems we have a *Lens*, and more precisely, we have one BiGUL bidirectional transformation per subsystem. In addition to that, we have the list of rules and the context to determine the priority of the subsystems. We can represent schematically the system like this:

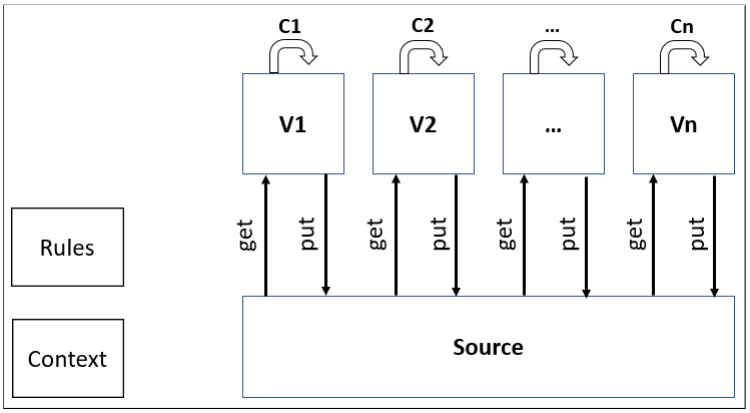


Figure 4.2: Modular system, rules and context

With this schema, we have to think about the most efficient way to synchronize the subsystems. Even if BiGUL already solves the synchronization of subsystems independently, we have to find a way to create a framework, making our system self-adaptive. Theoretically, we have chosen to use the same pattern as the first loop, namely a MAPE loop. In this kind of loop we have four steps:

- Monitor : Its goal is to retrieves (monitor) the information about the context
- Analysis : This step has to determine the rules that will be used to determine the subsystem order
- Planning : It determines according to the context and the chosen rules the order of subsystems.
- Execute : Its role is to execute the subsystems in the order chosen to update the data of the main system.

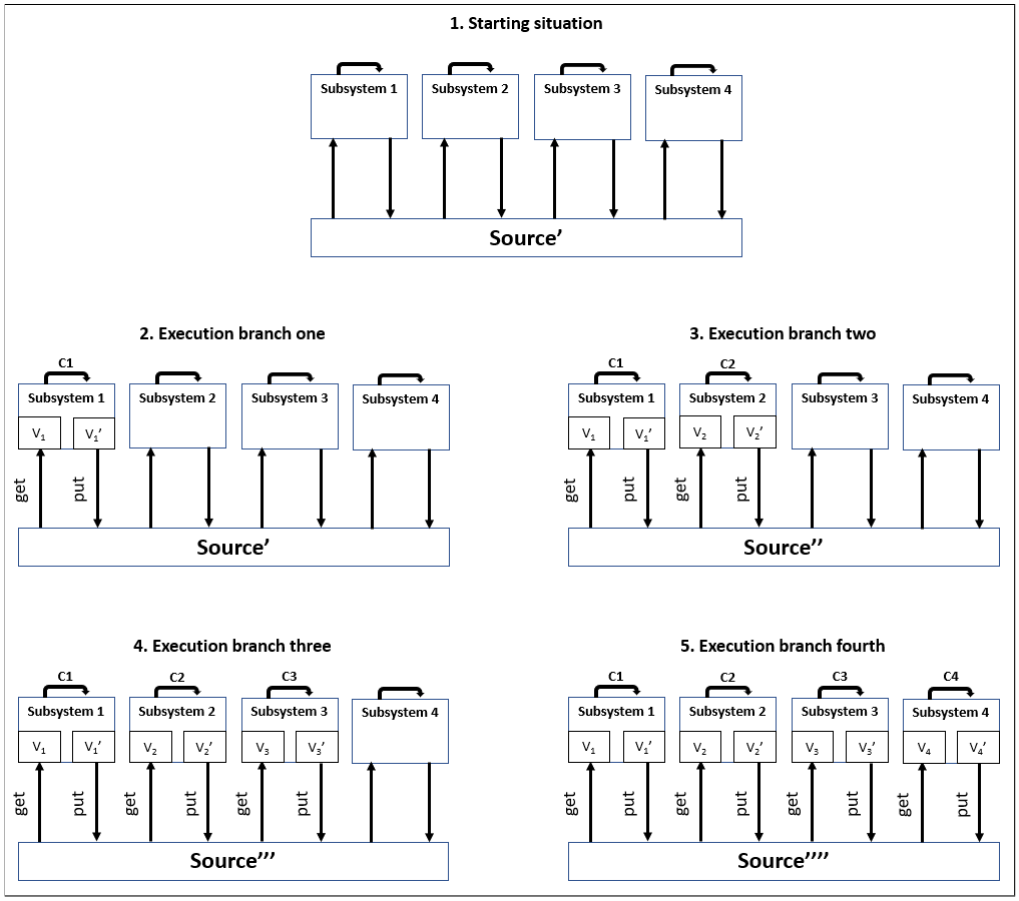


Figure 4.3: Sequential execution of concerns

4.1.3.1 Details about the execution

Those steps will be explained later when we describe the implementation (section 4.2). Here, we emphasize the execution step. Suppose that we have the priority of each subsystem and we just have to run each of them. There are two main ways to execute a system like this. In software engineering, we can do either sequential executions or parallel executions. Even if we have implemented the second one, we think it's important to analyze both and determines the benefits and drawbacks. With a sequential execution, it involves executing each subsystem one by one. The execution steps are described in Figure 4.3. As we can see, we execute the get of the first subsystem to obtain its data, called V_1. After some work on those data (ie. the analysis and planning steps) we obtain this updated data, called V_1'. Then, we execute the put to update the source and obtain $Source'$. After that, we reiterate for the next subsystem but we use the updated source ($Source'$) for the get and we obtain an updated source again ($Source''$), after the put. We apply this strategy for all the subsystems from the less important to the most important and finally, we obtain the completely updated source. Using this approach allows us to make the calculation of each subsystem only once but, in some cases, we are wasting time because two (or more) independent subsystems could work simultaneously. Indeed, if two systems do not share any common data, they can work simultaneously without any conflict and could optimize the CPU usage.

The second way to execute our subsystems is in parallel. As we said before, when two subsystems do not share any common data we can execute them simultaneously. The hard work here is to determine if two subsystems share data. We thought about a solution to optimize the time of the whole execution by using parallelism. All the steps to explain

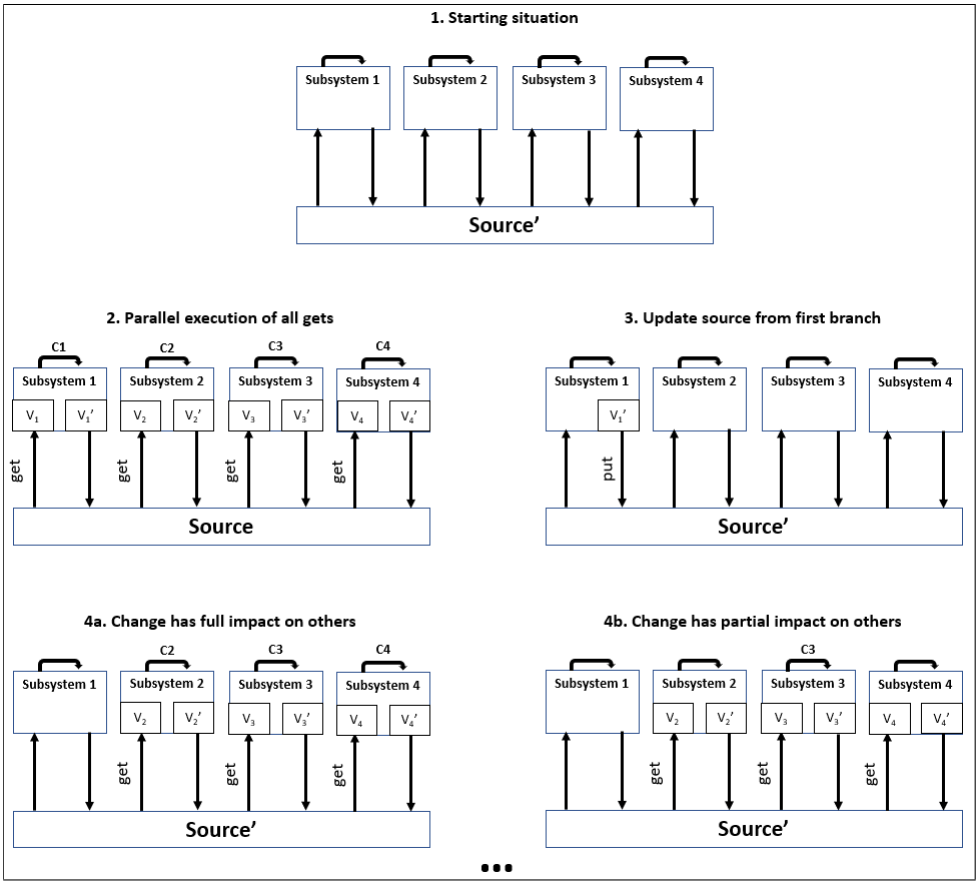


Figure 4.4: Parallel execution of concerns

that are shown in Figure 4.4. To summarize, we start with our system. In the schema, it possesses four subsystems and a pair of functions for each of them. The first step applies all get functions to obtain all subsets of data, called V_x. In this way, all concerns can work in parallel (independently) to update their own data to V'_x. The second step consists of applying the put of the first subsystem to obtain the updated source ($Source'$). After that, we reapply all gets of the remaining subsystems, but there is a difference with step 1. When we have the result of the get V_x, we compare it to the results already obtained when we call the previous (V_x^{t-1}). Thus, if the data needed by a subsystem is the same ($V_x = V_x^{t-1}$), it means that the change made by the previous subsystem has no effect on the current subsystem, and a new calculation to update its own data is useless. However, if V_x is different from V_x^{t-1}, we have to apply the calculation again and generate a new V'_x. In both cases, we have to run the put to obtain the updated source again ($Source''$). Now, we have the source updated by the two first subsystems. The next steps follow the same pattern, we called all remaining gets (Subsystem 3 and 4), we check if we need to make the calculation again (if $V_{3,4}$ is not equals to $V_{3,4}^{t-1}$) and we call the put of the next subsystem to update the source with the updated view (V'_3), and so on.

As you can see in Figure 4.4, the step 4a describes a situation where the changes made by the first branch require a new calculation. In other words, the first branch and the others share the same data and they have been updated by the first branch. The step 4b describes the situation where some other subsystems are not impacted by the changes made by the first branch. In this case, the calculation is not necessary. Of course, it might be impacted by another branch. This is why we apply the get in a parallel way, but each put is executed sequentially.

We thought about these two strategies but, as you will see, we only implemented the first one. There are multiple reasons why we did that. The first one is a reason of programming time. Indeed, during our intership we only had the time to implement one of them. The second reason is a question of efficiency. Thus the question is, why did we choose to implement this one. Here is, mathematically, what these two approaches mean:

Assume the function $t(x)$ gives us the time required to execute x. For the sequential approach, we have:

$$\begin{aligned} Wall-clock\ time_{sequential} = \\ & t(get_1) + t(C_1) + t(put_1) + \\ & t(get_2) + t(C_2) + t(put_2) + \\ & t(get_3) + t(C_3) + t(put_3) + \\ & t(get_4) + t(C_4) + t(put_4) \end{aligned}$$

$$\begin{aligned} Total\ calculation\ load_{sequential} = \\ & get_1 + C_1 + put_1 + \\ & get_2 + C_2 + put_2 + \\ & get_3 + C_3 + put_3 + \\ & get_4 + C_4 + put_4 \end{aligned}$$

The generalization of those formulas for n subsystems is:

$$Wall-clock\ time_{sequential} = \sum_{i=1}^{n} t(get_i) + t(C_i) + t(put_i)$$

$$Total\ calculation\ load_{sequential} = \sum_{i=1}^{n} get_i + C_i + put_i$$

For the parallel execution, the development is more complicated. The result changes if the subsets of data are strongly overlapping. The best and the worst cases are calculated here. The best situation is when the changes made by previous subsystems have no impact on the data of the others (Figure 4.6), and the worst situation is when a change in the previous subsystem has an impact on all the others (Figure 4.5). The calculation for the time in the best and the worst case is:

$$\begin{aligned} Best\ wall-clock\ time_{parallel} = \\ & \max[t(get_1) + t(equality\ check_1) + t(C_1) + t(put_1), \\ & \quad t(get_2) + t(equality\ check_2) + t(C_2), \\ & \quad t(get_3) + t(equality\ check_3) + t(C_3), \\ & \quad t(get_4) + t(equality\ check_4) + t(C_4)] + \\ & \max[t(get_2) + t(equality\ check_2) + t(put_2), \\ & \quad t(get_3) + t(equality\ check_3), \\ & \quad t(get_4) + t(equality\ check_4)] + \\ & \max[t(get_3) + t(equality\ check_3) + t(put_3), \\ & \quad t(get_4) + t(equality\ check_4)] + \\ & t(get_4) + t(equality\ check_4) + t(put_4) \end{aligned}$$

The first *max* is to wait the first execution of each concern. This way, we run gets, equality checks and calculations all together in the same time and we execute the put of

the first concern. At that point, there are 3 concerns left. The second max is the execution of those three concerns but, in the best case, we do not need to do the calculation again because concerns are totally independent. The last max is the step with 2 concerns left. And finally, the last line of the equation is the application of the last concern. Now, let's see what is happening when the concerns strongly share the same parts of data.

$$
\begin{aligned}
worst\ wall-clock\ time_{parallel} = \\
&\max[t(get_1) + t(equality\ check_1) + t(C_1) + t(put_1),\\
&\quad t(get_2) + t(equality\ check_2) + t(C_2),\\
&\quad t(get_3) + t(equality\ check_3) + t(C_3),\\
&\quad t(get_4) + t(equality\ check_4) + t(C_4)] +\\
&\max[t(get_2) + t(equality\ check_2) + t(C_2) + t(put_2),\\
&\quad t(get_3) + t(equality\ check_3) + t(C_3),\\
&\quad t(get_4) + t(equality\ check_4) + t(C_4)] +\\
&\max[t(get_3) + t(equality\ check_3) + t(C_3) + t(put_3),\\
&\quad t(get_4) + t(equality\ check_4) + t(C_4)] +\\
&t(get_4) + t(equality\ check_4) + t(C_4) + t(put_4)
\end{aligned}
$$

The *worst wall-clock time* has the same pattern than the *best wall-clock time* but we have to calculate again the new data of each subsystem (C_n). Now we have the calculation for 4 concerns. Let's generalize for n subsystems:

$$
\begin{aligned}
Best\ wall-clock\ time_{parallel} = \\
&\max[t(get_1) + t(equality\ check_1) + t(C_1) + t(put_1),\\
&\quad t(get_2) + t(equality\ check_2) + t(C_2),\\
&\quad ...\\
&\quad t(get_n) + t(equality\ check_n) + t(C_n)] +\\
&\sum_{i=2}^{n}(\max[t(get_i) + t(equality\ check_i) + t(put_i),\\
&\quad t(get_{i+1}) + t(equality\ check_{i+1}),\\
&\quad t(get_n) + t(equality\ check_n)])
\end{aligned}
$$

$$
\begin{aligned}
Worst\ wall-clock\ time_{parallel} = \\
&\max[t(get_1) + t(equality\ check_1) + t(C_1) + t(put_1),\\
&\quad t(get_2) + t(equality\ check_2) + t(C_2),\\
&\quad ...\\
&\quad t(get_n) + t(equality\ check_n) + t(C_n)] +\\
&\sum_{i=2}^{n}(\max[t(get_i) + t(equality\ check_i) + t(C_i) + t(put_i),\\
&\quad t(get_{i+1}) + t(equality\ check_{i+1}) + t(C_{i+1}),\\
&\quad t(get_n) + t(equality\ check_n) + t(C_n)])
\end{aligned}
$$

The best and worst calculation time is easier to calculate. We have to sum all the components of each branch.

$$Best\ total\ calculation\ load_{parallel} = \sum_{i=1}^{n}(\sum_{j=i}^{n}(get_j + equality\ check_j) + C_i + put_i)$$

$$Worst\ total\ calculation\ load_{parallel} = \sum_{i=1}^{n}(\sum_{j=i}^{n}(get_j + equality\ check_j + C_i) + put_i)$$

To make the equations simpler, we can assume that both put and get transformations and the equality checks are constant, and that those values are negligible. Thus, we can remove them from the equations:

$$\begin{aligned}
Total\ time_{sequential} &= \sum_{i=1}^{n} t(C_i) \\
Total\ calculation\ load_{sequential} &= \sum_{i=1}^{n} C_i \\
Best\ wall-clock\ time_{parallel} &= \max[t(C_1), ..., t(C_n)] \\
Worst\ wall-clock\ time_{parallel} &= \max[t(C_1), ..., t(C_n)] + \sum_{i=2}^{n}(\max[t(C_i), ..., t(C_n)]) \\
Best\ total\ calculation\ load_{parallel} &= \sum_{i=1}^{n} C_i \\
Worst\ total\ calculation\ load_{parallel} &= \sum_{i=1}^{n}(\sum_{j=i}^{n} C_i)
\end{aligned}$$

The choice of the sequential approach is pretty obvious. Even if, in some cases, the parallel approach is more rapid, the worst case also needs to be avoided. The parallel approach could be useful for problems with limited interference but it requires more investigation and analysis of the system. Remember that the goal of our synchronizer is to provide an efficient way to execute a modular system and this synchronizer must be a library.

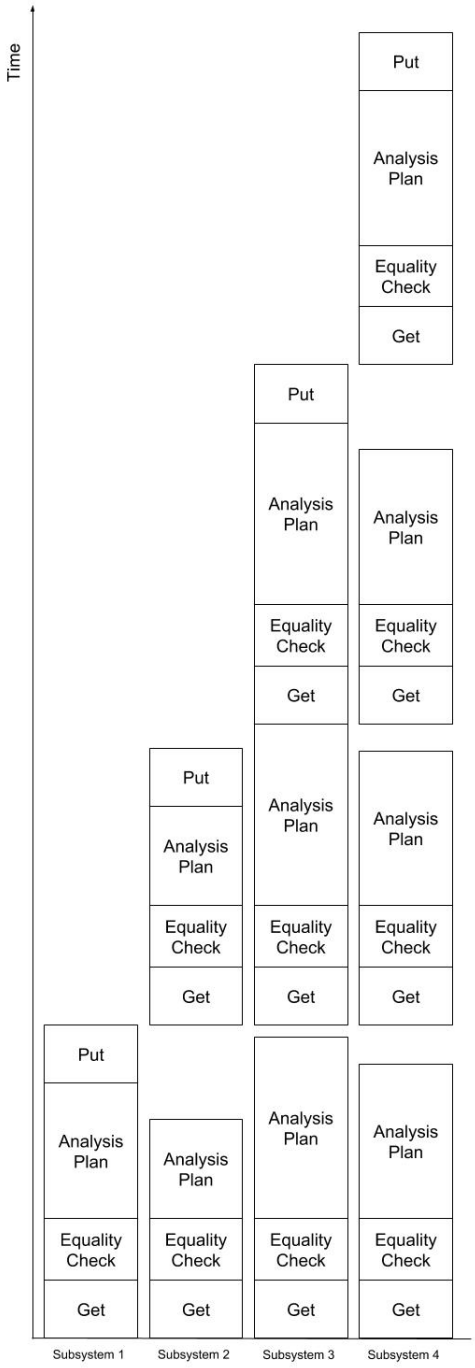


Figure 4.5: Parallel execution : worst case

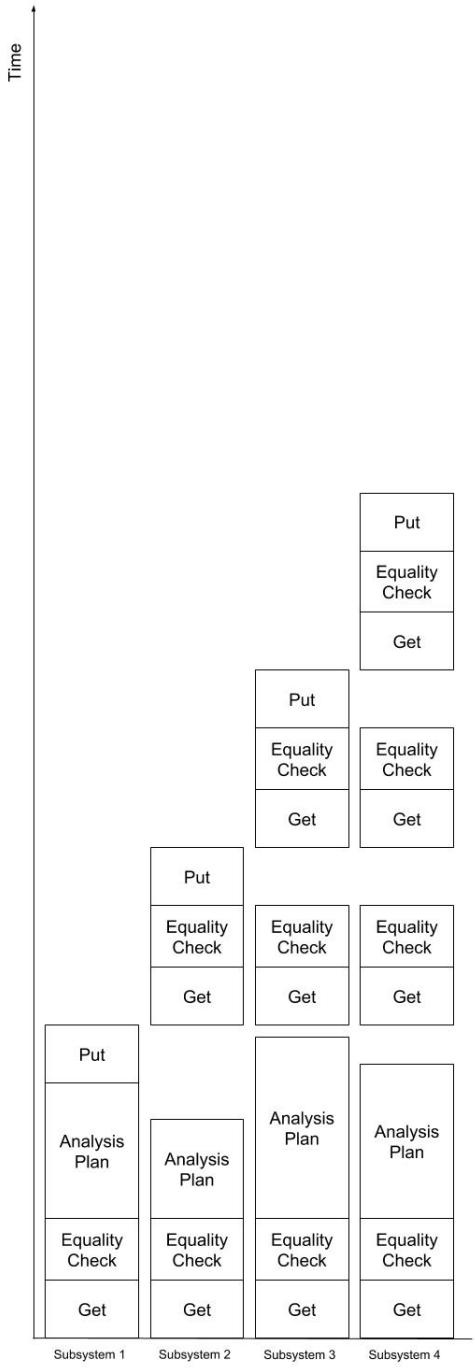


Figure 4.6: Parallel execution : best case

4.2 Development of the Haskell Solution

The integration of our idea in Haskell is explained in this section. To put together the whole model, we use the modular system explained in chapter 3. As explained in this chapter, the model contains views. From now on, we will use the word "view" to speak about the data on which the subsystem works, and the word "concern" (subsubsection 4.2.1.1) to speak about the subsystems themselves. Remember, we wanted the prioritizing *MAPE loop* to be implemented like a library that can be used independently of the system handled by the first *MAPE loop*. It provides to the programmer several functions to synchronize different submodules. As a reminder, here is the schema focusing on the second loop which we had already briefly mentioned in chapter 2 (Figure 2.2 and Figure 2.3), also called the *synchronizer*. As you can see, it is an evolution of the Figure 4.2 :

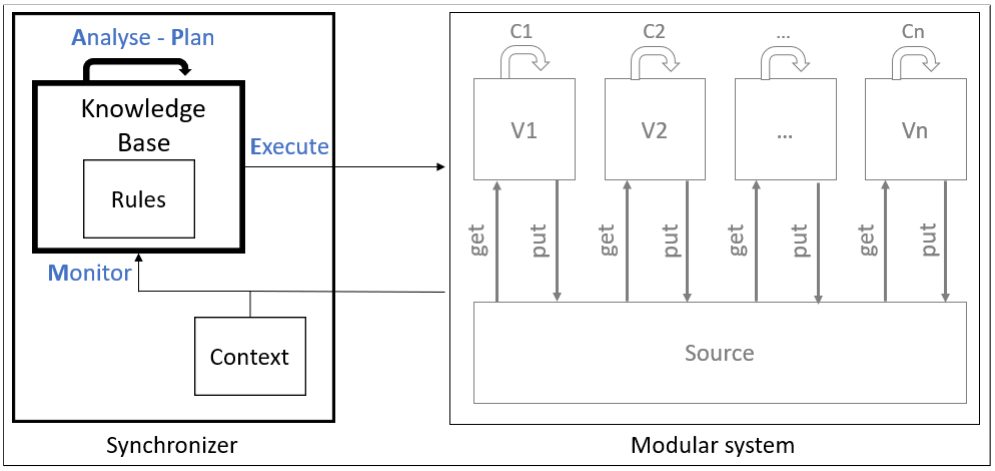


Figure 4.7: Synchronizer

First are explained concepts needed to be created in Haskell in order to develop our library. These concepts are used all along our synchronizer and must be clearly defined. After that, we describe how we have adapted the theoretical solutions using a MAPE loop in Haskell.

4.2.1 Concepts and utilities

4.2.1.1 The concerns

The synchronizer needs to know information about the modular system it is currently managing. First, it needs the source of data already explained in section 3.2, but it also needs to know some information about each subsystem. We have created the concept of "concern" to handle this information.

Concretely, a concern is one of the four branches of our first *MAPE loop* (chapter 3). In our case, those concerns are: Performance (AutoScaling view), Security (Firewall view), Availability (Redundancy view) and Modifiability (Cost view). As you can see in the Figure 3.1, each branch is made up of a BiGUL function and an analyser and planner. In order to make our system as adaptable as possible, we decided to create a Haskell type called *Concern*. Basically, a concern is represented by a identifier name (for example: Performance, Availability, Security, ...) [1], a BiGUL function to switch from the source to the view and

[1] Note that sometimes in the code we use the name of the view instead of the name of the concern. In fact, the name does not matter as long as it clearly identifies what it is.

vice versa, and finally, a way to execute the analysis and planning steps. Concretely, our implementation handles two types of situations.

The first situation is when the analysis and planning steps of the concern are executed in Haskell on the same machine as the synchronizer. In this case, we have to define which BX to use, but also the function of analysis and planning. You can see the implementation of this type of concern in the Listing 4.7 from lines 8 to 11.

The second situation is when the analysis and planning steps are implemented as a web service available though the Internet. To handle this situation, we have created a Haskell data called *ConcernRemote* in the Listing 4.7, from lines from lines 13 to 18. In this case, we need more information. Indeed, we chose to communicate with servers through the HTTP Protocol. Thus, we must specify the information about the server (URL, port and secured value). Moreover, we have to communicate with these servers. Then, we provide a way for the programmer to write two functions to specify how the view of the concern has to be sent to the server (line 17) and how the response of the server has to be transformed in a view (line 18).

```
1  type ConcernName = String
2  type Url = String
3  type Port = Int
4  type Secured = Bool
5  type Serializer v = (v -> String)
6  type Deserializer v = (String -> v)
7  data Concern s  = forall v. (ToJSON v, Show v, NFData v) =>
8                       Concern
9                         ConcernName
10                        (BiGUL s v)
11                        (v -> v)
12                | forall v. (ToJSON v, Show v, NFData v) =>
13                      ConcernRemote
14                        ConcernName
15                        (BiGUL s v)
16                        Url Port Secured
17                        (Serializer v)
18                        (Deserializer v)
```

Listing 4.7: Haskell data to represent a Concern

The *Concern* data takes one argument. As all the other concerns, it must be based on the same source, whereas the view is generally different. Each type of *Concern* has a name to identify it and a *BiGUL* BX to make the link between the source and its view. The specify data about the two types of concern comes after the *BiGUL* function. Note that we use the existential quantification to allow all kind of views.

The data structure is specified. The explanation of the branches in terms of *Concerns* remains. The following code is an example to define the list of concerns for our use case. Therefore we just give you the signature of the function used. Notice that all serialize and unserialize functions transform views to JSON and vice versa and are automatically generated by an Haskell library called *Aeson* already used in Section 3.7.1.1. Thus, showing the implementation is not necessary. However, you can see the definition of the four BiGUL functions (lines 7 to 10) in the chapter 3. If you want to see a complete example, you can go to chapter 5, about experimentation, or in the github where the code of our system is runnable (`https://github.com/qlombat/Self-Prioritized-Modular-Adaptations`).

```
1  serialize :: ToJSON a => a -> String
2  unseralizeC :: String -> CView
```

```
unseralizeAS :: String -> ASView
unseralizeF :: String -> FView
unseralizeR :: String -> RView
autoScalingUpdate :: BiGUL S.Source AS.ASView
costUpdate :: BiGUL Source CView
firewallUpdate :: BiGUL Source FView
redundancyUpdate :: BiGUL S.Source R.RView
[
  (ConcernRemote "Cost" costUpdate
      "http://mywebsite.tld/myscript" 80 False serialize unseralizeC),
  (ConcernRemote "Firewall" firewallUpdate
      "http://mywebsite.tld/myscript" 80 False serialize unseralizeF),
  (ConcernRemote "Redundancy" redundancyUpdate
      "http://mywebsite.tld/myscript" 80 False serialize unseralizeR),
  (ConcernRemote "AutoScaling" autoScalingUpdate
      "http://mywebsite.tld/myscript" 80 False serialize unseralizeAS)
]
```

Listing 4.8: Haskell data to represent a Concern

4.2.1.2 The Context

The context describes the current situation in which the system is. It is given by the programmer and it contains all the information judged useful to establish a priority order for the system. For example, a context can contain some simple values like the date or the time stamps. It can also contain more complex values like the advertising revenue, or an aggregation of data to determine if the system is in an emergency situation or not.

The context has a predefined structure that the programmer has to follow. As we already said in subsection 4.1.2, a context is theoretically a list of key/value pairs. The translation of this requirement in Haskell is a map of two types. The first is the key, a *String*. The second is the value related to the key. For our experiments, we only need 4 types of values, which are integers, doubles, strings and booleans.

```
data CtxValue = I Int | D Double | S String | B Bool deriving (Eq, Ord, Generic)
type CtxKey = String
type Context = Map CtxKey CtxValue
```

Listing 4.9: Data structure of the context

Usually, the context is just a Haskell file with a function that returns a *Context* value. By convention, this file provides a function called *makeContext* that retrieves a *IO Context*. It returns an *IO* because the context is mainly based on external data.

```
getHourOfDay :: IO Int
getHourOfDay = do
    now <- getCurrentTime
    timezone <- getCurrentTimeZone
    let (TimeOfDay hour minute second) = localTimeOfDay $ utcToLocalTime timezone
        now
    return hour
getTimestamp :: IO Int
getTimestamp = (read <$> formatTime defaultTimeLocale "%s" <$> getCurrentTime) ::
    IO Int
makeContext :: IO Context
makeContext = do
    hour <- getHourOfDay
    timestamp <- getTimestamp
```

```
13      emergency <- return True
14      return (fromList [
15        ("Emergency", B emergency),
16        ("HourOfDay", I hour),
17        ("Timestamp", I timestamp)])
```

Listing 4.10: Example of context

The simple example above shows a context with 3 different data items. The first value describes if the system is in emergency mode. It can be produced by sensors or information about a security breach, but here we just put a placeholder. The next values represent the current information about the time.

As you will see in the following sections, the rules are based on the context. We can see the context as a snapshot of the situation in which the system is and the rules as the behavior to adopt.

4.2.1.3 The Rules

The rules are described by the user, so that the system can know, according to changing elements, in which order it is better to execute the views at that precise time. First is detailed the data structure. Then is explained the parser used to transform the content of a *.txt* file into the data structure.

As the rest of the model, the rules have got their own Haskell data structure:

```
1  type Rules = [Rule]
   type Rule = (RuleOperator, [RuleView])
   data RuleOperator =
       Equals CtxKey CtxValue |
5      LessThan CtxKey CtxValue |
       LessOrEqualsThan CtxKey CtxValue |
       MoreThan CtxKey CtxValue |
       MoreOrEqualsThan CtxKey CtxValue |
       Not RuleOperator |
10     And RuleOperator RuleOperator |
       Or RuleOperator RuleOperator |
       T | F
   data RuleView = Anything | V String
```

Listing 4.11: Data structure of the rules

As shown above, the *Rules* of the system are a list of *Rule*. A *Rule* is a pair, whose first element is a *RuleOperator*, and the second is a list of *RuleView*. A *RuleOperator* is equivalent to a boolean condition. If the computation of the *RuleOperator* returns *true*, the order of the views describes in the associated list of *RuleView* must be taken into account. If it returns *false*, the associated list is ignored. Obviously, *RuleOperator* contains all the operators needed for a boolean condition to be exhaustive. Regarding *RuleView*, it is either *V String*, where the string is the name of a view, or *Anything*, which represents any other list of views. It means that, every time *Anything* appears between two names in a list, it does not matter if the second name is not right behind the first one in the final execution order, as long as it stays behind it. To facilitate the display of the data structure, we have also implemented a way to show them:

```haskell
instance Show RuleOperator where
    show (Equals k v) = k ++ " == " ++ show v
    show (LessThen k v) = k ++ " < " ++ show v
    show (LessOrEqualsThen k v) = k ++ " <= " ++ show v
    show (MoreThen k v) = k ++ " > " ++ show v
    show (MoreOrEqualsThen k v) = k ++ " >= " ++ show v
    show (Not x) = "not( " ++ show x ++ ")"
    show (And r1 r2) = "(" ++ show r1 ++ ") and (" ++ show r2 ++ ")"
    show (Or r1 r2) = "(" ++ show r1 ++ ") or (" ++  show r2 ++ ")"
    show (T) = "true"
    show (F) = "False"
```

Listing 4.12: Display function of RuleOperator

```haskell
data RuleView = Anything | V String
instance Show RuleView where
    show (Anything) = "*"
    show (V v) = show v
```

Listing 4.13: Display function of RuleView

With those functions, a basic rule in the system, which looks like: *(And (LessThen "HourOfDay" I 22) (MoreThan "HourOfDay" I 6), [V Auto-scaling, V Cost, Anything, V Redundancy])* then becomes for the user: *("HourOfDay" < 22) and ("HourOfDay" > 6) : Auto-scaling, Cost, *, Redundancy.* The data structure of the rules allows us to manipulate them in the program, but does not tell how to stock them. All the rules are defined inside a *.txt* file, and written for the user to read it easily if he wants to add a rule at a special position. To convert the rules in the *.txt* file into the Haskell data format, we use a parser. BiYacc [39] could have been used here. But it as been designed by the same team as BiGUL. Its creators told as that BiYacc was not supported anymore and did not work at all. Thus, we created our own parser.

4.2.1.4 The Parser

The parser uses the syntax as described by *Shown*. Its code looks like this:

```haskell
ruleParser :: String -> IO [Rule]
ruleParser fileName = do
    listOfString <- fileIntoList fileName
    return (parseListStringToListRules listOfString)
```

Listing 4.14: ruleParser

The function takes the name of the *.txt* file as input, and returns all the rules inside as a list. *ruleParser* is composed of two functions. The first one, *fileIntoList*, reads the rules line by line and returns them as a list of strings. The other function, *parseListStringToListRules*, transforms the list of strings into a list of *Rules*, the correct Haskell data type.

```haskell
fileIntoList :: String -> IO [String]
fileIntoList fileName = fmap lines (readFile fileName)
```

Listing 4.15: fileIntoList

As mentioned before, *fileIntoList* returns the list of rules in the *.txt* file as a list of strings. To do so, it simply calls 3 functions already defined in Haskell. *readFile* takes the

path to a file, and returns the content of the file as a single *String*. From this result, the combination of *fmap* and *lines* cuts the single string at each new line character, to send the list of strings. As a matter of fact, we ask the user to write one rule by line in the *.txt* file.

```
1 parseListStringToListRules :: [String] -> [Rule]
2 parseListStringToListRules ls = map (\s -> parseStringToRule s) ls
```

Listing 4.16: parseListStringToListRules

Obviously, the *parseStringToRule* function is applied at each string of the list, thanks to the *map* Haskell function, to transform it into a list of rules.

```
1 parseStringToRule :: String -> Rule
2 parseStringToRule line = (parseCondition (unpack (res !! 0)), parseRulesOrder (
      unpack (res !! 1)))
3     where
4         res = splitOn ":" (removeWhiteSpaces line)
```

Listing 4.17: parseStringToRule

```
1 removeWhiteSpaces :: String -> Text
2 removeWhiteSpaces s = strip (pack (filter (\c -> c /= ' ') s))
```

Listing 4.18: removeWhiteSpaces

A *Rule* is just a pair, where the first element is a *RuleOperator*, and the second is a list of *RuleView*. Thus, for each line of the list, *parseStringToRule* returns a pair with the correct types. As a reminder, a basic rule looks like: *("HourOfDay" < 22) and ("HourOfDay" > 6) : Auto-scaling, Cost, *, Redundancy*. The line, from which the white spaces are removed, is split into two parts at the suspension points. The first part represents the boolean condition of the rule, while the second part is the order in which the views must be executed. *parseCondition* and *parseRulesOrder* handle those parts respectively. All along the code of the parser, there will be the *pack* and *unpack* functions. Those two functions are used to transform a *String* into a *Text*, and the reverse. The *Text* Haskell type is demanded for a lot of functions, like *splitOn*. Here is the code for *parseRulesOrder*:

```
1 parseRulesOrder :: String -> [RuleView]
2 parseRulesOrder rules = map (\name -> replaceViewNames name) (map unpack (splitOn "
      ," (pack rules)))
3     where
4     replaceViewNames [] = error ("Inconsistent rules")
5     replaceViewNames "*" = Anything
6         replaceViewNames n = V n
```

Listing 4.19: parseRulesOrder

From a string like *Auto-scaling, Cost, *, Redundancy*, the *splitOn* function returns a list of strings by cutting at each coma. Then, *replaceViewNames* actually transforms a string (the name of each view), into the correct Haskell data type.

```
1 parseCondition :: String -> RuleOperator
2 parseCondition ('(':ss) = case handleParenthesis "" ss 1 of
3     (cond, ('a':'n':d':ss)) -> And (parseCondition cond) (parseCondition ss)
4     (cond, ('o':'r':ss)) -> Or (parseCondition cond) (parseCondition ss)
```

```
5       (cond, "") -> parseCondition cond
6       (_, _) -> error ("Inconsistent rules")
7   parseCondition ('n':'o':'t':'(':ss) = Not (parseCondition ('(':ss))
8   parseCondition condition = case handleCondition "" condition of
9       (key, ('<':'=':value)) -> LessOrEqualsThen key (handleValue value)
10      (key, ('<':value)) -> LessThen key (handleValue value)
11      (key, ('>':'=':value)) -> MoreOrEqualsThen key (handleValue value)
12      (key, ('>':value)) -> MoreThen key (handleValue value)
13      (key, ('=':'=':value)) -> Equals key (handleValue value)
14  handleParenthesis :: String -> String -> Int -> (String,String)
15  handleParenthesis treated rest 0 = (init treated, rest)
16  handleParenthesis treated ('(':ss) int = handleParenthesis (treated ++ "(") ss (int
        + 1)
17  handleParenthesis treated (')':ss) int = handleParenthesis (treated ++ ")") ss (int
        - 1)
18  handleParenthesis treated (s:ss) int = handleParenthesis (treated ++ [s]) ss int
19  handleParenthesis _ _ int = error ("Inconsistent rules")
20  handleCondition :: String -> String -> (String,String)
21  handleCondition treated ('<':'=':ss) = (treated, '<':'=':ss)
22  handleCondition treated ('<':ss) = (treated, '<':ss)
23  handleCondition treated ('>':'=':ss) = (treated, '>':'=':ss)
24  handleCondition treated ('>':ss) = (treated, '>':ss)
25  handleCondition treated ('=':ss) = (treated, '=':ss)
26  handleCondition treated (s:ss) = handleCondition (treated ++ [s]) ss
27  handleCondition _ _ = error ("Inconsistent rules")
28  handleValue :: String -> CtxValue
29  handleValue value
30      | (isDigit (head value)) = intOrDouble value
31      | (value == "True") = B True
32      | (value == "False") = B False
33      | otherwise = S value
34  intOrDouble :: String -> CtxValue
35  intOrDouble value = if (elem '.' value) then D (read value) else I (read value)
```

Listing 4.20: parseCondition and its inner functions

From a string like *("HourOfDay" < 22) and ("HourOfDay" > 6)* (the parenthesis are mandatory), the *parseCondition* function returns *And LessThan "HourOfDay" I 22 MoreThan "HourOfDay" I 6*. To do so, it works like a tree. Firstly, it handles the parenthesis, by reading one by one each character and applying the "+1/-1" strategy to find out which closing parenthesis belongs to which opening parenthesis. If the result, at a point, equals 0, it means that a pair of matching parenthesis has been identified. Once the content of the correct parenthesis has been found, *handleCondition* detects which operator is used in this boolean condition, so that the program can properly create the right constructors, from line 9 to 13. Finally, the last thing to parse are the values. *handleValue* checks if there is a digit inside the treated string. If it does, it is either an *Integer* or a *Double*. If it does not, it is either one of the two *booleans*, or a *string*.

4.2.1.5 Priority Policy

Clarifying the rules was mandatory before explaining what a priority policy is. As explained earlier, the user defines the rules in a text file. By definition, a rule can be true or false. As showed above, two rules that have opposite demands cannot be true at the same time. If it was the case, the set of rules would be inconsistent. We decided to define the list of rules as follows: "A list of rules is consistent when, no matter the context, two or more rules

cannot be true together if they are contradictory in terms of the execution order". To avoid inconsistency, we put in place several policies. They are threefold:

```
1 data PriorityPolicy = Neutral | Safety | Last  deriving Generic
```

Listing 4.21: PriorityPolicy

- Neutral : this strategy assumes that the user is expert in his domain and all rules are consistent. The case where one rule contradicts another cannot happen or the system is not able to make a choice. If we take two conflicting rules, with this strategy, the rule that asks to put A before B cannot be true at the same time that the rule asking to put B before A. If this situation occurs, the system will not be able to work properly and an error will be triggered.
- Safety : this strategy gives the advantage to the earliest rules. If a conflict occurs, it takes the first rule that appears in the file. With our example, this strategy choose to keep the first rule and the correct order returned by the function will be A before B.
- Last : this strategy is the complete opposite of the previous one. As the user just added the rule, it assumes that it is more reliable. In this case, the system does not take into account the oldest rules if a conflict occurs. Thus, the result obtained will be B before A because the rule kept is the last one.

the use and implementation of different policies will be seen later in Listing 4.28.

4.2.2 Monitoring

The *Monitoring* is the first step of our second MAPE loop and gets information from the main source and from the context. To idea behind the monitoring of these two elements is to separate the things related to the modular system (source) (like the number of instances in our case) and the things related to the outside system (context) (like the current time, the weather or the time stamp). As you already know, the second part of our system is theoretically a MAPE loop. To make our library easier to use, we created an interface as simple as possible. It is the reason why we chose to create a unique function called *executionSeq* (the name means that the execution is sequential) that handles the monitoring and execution steps. Our model is conceptually the same but during the implementation we use the parameters as the monitor and the result of the function as execution. To understand clearly how our synchronizer works, here is the signature of the function.

```
1 executeSeq :: Context -> Rules -> PriorityPolicy -> s -> [Concern s] -> IO (s, [
    MasterView])
```

Listing 4.22: Signature of the function used for the monitoring

As explained above, the monitoring step is a part of the execution function. To be more specific, the monitoring is represented by the parameters of the function *Context -> Rules -> PriorityPolicy -> s -> [Concern s]*). Remember, the conceptual schema (Figure 4.7) shows that our synchronizer monitors the context and our first MAPE loop. The context is then represented in the signature by the first parameter, typed Context (subsubsection 4.2.1.2). The first MAPE loop is represented by the fourth and fifth parameters. The first one is the source used by our first loop and the second is the list of concerns that we could see in subsubsection 4.2.1.1. The other parameters are about the rules used in our system (subsubsection 4.2.1.3) (the second argument) and the strategy used to determine the order of the rules (subsection 4.2.3) (the third argument).

4.2.3 Analysis and Planning

The goal of the analysis and planning steps is to determine the rules to use (Analysis) and find a compatible order for the execution of the concerns (Planning). We have a function called *executeSeq*, which is called with the information to monitor as input. Its output should be the execution step. The analysis-planning pair is located inside the function. Here are the explanations, line by line, of the function. Analysis and planning steps go from line 6 to line 8.

```
1  executeSeq :: Context ->
2                Rules -> PriorityPolicy ->
3                s -> [Concern s] ->
4                IO (s, [MasterView])
5  executeSeq ctx rules priorityPolicy source concerns = do
6      let concernNames = map getConcernName concerns
7      let order = determineOrder ctx rules concernNames priorityPolicy
8      let concernsOrdered = reverse (orderConcern order concerns)
9      res <- foldM fn (source, []) concernsOrdered
10     return res
11     where
12         fn (accS, accV) concern = do
13             branchResult <- execBranch accS concern
14             return (fst branchResult, accV ++ [(snd branchResult)])
```

Listing 4.23: Implementation of function executeSeq

4.2.3.1 Analysis: Determine the execution order

Firstly, we need to know all the names of the concerns. Line 6 of Listing 4.23 handles that point. It is a map applying the function *getConcernName* to the concerns. As shown in the code below, the function returns a string corresponding to the name of the concern. In the end, we have a list of string (list of concerns' names), called *ConcernNames*. This list is mandatory because later we only use the name of the concern to determine the order of the executions. It allows us to work on less heavy data.

```
1  getConcernName :: Concern s -> String
2  getConcernName (Concern name _ _) = name
3  getConcernName (ConcernRemote name _ _ _ _ _ _) = name
```

Listing 4.24: Implementation of function getConcernName

Secondly, we need to determine the execution order and store it into *order*. To accomplish this task we use the function *determineOrder*, that needs the context, the list of rules, the list of concerns' names and the priority policy. As shown below, we have the *lstViews*, which is the list of concerns' names transformed into a view type. Then, we have *lstRulesView* (that calls the function *evalRules* to determine the rules to keep) and after that, with a map, we extract the rule views from the rules with a true condition.

```
1  determineOrder :: Context -> Rules -> [String] -> PriorityPolicy -> [String]
2  determineOrder ctx rules lstConcern priorityPolicy = map (\(V a) -> a) (
       determineViewPriority priorityPolicy lstViews lstRulesView)
3      where
4          lstViews =  map (\a -> V a) lstConcern
5          lstRulesView = map (\(_, a) -> a) (evalRules ctx rules)
```

Listing 4.25: determineOrder function

The function *evalRules* is a filter on the boolean condition of the rule. Then, *evalBooleanCondition* returns true when the rule is true in the current context and false otherwise. The function seems to be a little bit heavy but in fact, this feeling is caused by the number of pattern matching we have to make.

```
-- Describe how to evaluate a rule
evalRules :: Context -> Rules -> Rules
evalRules ctx = filter (\(a,b) -> evalBooleanCondition ctx a)
-- Return true when the boolean conditions are true according to the context, false
     otherwise
evalBooleanCondition :: Context -> RuleOperator -> Bool
evalBooleanCondition ctx (Equals k val) =
    case (Map.lookup k ctx) of
        Just v -> val == v
        Nothing -> False
evalBooleanCondition ctx (LessThen k val) =
    case (Map.lookup k ctx) of
        Just v -> v < val
        Nothing -> False
evalBooleanCondition ctx (LessOrEqualsThen k val) =
    case (Map.lookup k ctx) of
        Just v -> v <= val
        Nothing -> False
evalBooleanCondition ctx (MoreThen k val)   =
    case (Map.lookup k ctx) of
        Just v -> v > val
        Nothing -> False
evalBooleanCondition ctx (MoreOrEqualsThen k val)   =
    case (Map.lookup k ctx) of
        Just v -> v >= val
        Nothing -> False
evalBooleanCondition ctx (Not r) = not (evalBooleanCondition ctx r)
evalBooleanCondition ctx (And r1 r2) =  (evalBooleanCondition ctx r1) &&
                                        (evalBooleanCondition ctx r2)
evalBooleanCondition ctx (Or r1 r2) = (evalBooleanCondition ctx r1) ||
                                      (evalBooleanCondition ctx r2)
evalBooleanCondition ctx T = True
evalBooleanCondition ctx F = False
```

Listing 4.26: evalRules function

The list of rules is filtered and has kept only those with a true condition. *lstRulesView* is generated and contains a list of all the potential execution orders asked by the true rules. Actually we have this type of list:

```
lstRulesView = [
    [Anything, V "AutoScaling", Anything, V "Redundancy", Anything],
    [V "Cost", V "Redundancy", Anything],
    [Anything, V "AutoScaling", V "Redundancy", Anything],
]
```

Listing 4.27: Example of lstRulesView

The remaining part of *determineOrder* is now explained. The function calls another function called *determineViewPriority* and applies a map on the result to retrieve only the name of the views returned by the function. The function below tries to determine the order of the concerns based on the true rules and according to the *PriorityPolicy*. At that point, we have the list of all views handled by the system (as a reminder, the listView parameter

contains the name of all the concerns transformed in view type) and a list of the orders asked by the true rules. The output of the function must return a simple list of *RuleView*.

```
1 determineViewPriority :: PriorityPolicy -> [RuleView] -> [[RuleView]] -> [RuleView]
2 determineViewPriority Neutral listView lstRuleView =
3     head (findCorrectSituations listView lstRuleView)
4 determineViewPriority Safety listView lstRuleView =
5     head (determineViewPrioritySafety listView lstRuleView)
6 determineViewPriority Last listView lstRuleView =
7     head (determineViewPrioritySafety listView (reverse lstRuleView))
```

Listing 4.28: determineViewPriority function

In the case of the *Neutral* policy, the function *findCorrectSituations* takes directly *lstRuleView* without any filter, where the two others (Safety and Last) apply *determineViewPrioritySafety*. In the following explanation, we explain those two functions to understand clearly the usefulness of our strategy. But before, we need a function which compares two lists of RuleView and knows if those lists are consistent:

```
1  goodSituation :: [RuleView] -> [RuleView] -> Bool
2  goodSituation [] [] = True
3  goodSituation _ [Anything] = True
4  goodSituation [] (Anything:ys) = goodSituation [] ys
5  goodSituation [] _ = False
6  goodSituation _ [] = False
7  goodSituation x (Anything:Anything:ys) = goodSituation x (Anything:ys)
8  goodSituation (x:xs) (Anything:y:ys)
9      | (x == y) = (goodSituation xs ys)
10     | otherwise = (goodSituation xs (Anything:y:ys))
11 goodSituation (x:xs) (y:ys) = (x == y) && (goodSituation xs ys)
```

Listing 4.29: goodSituation function

The function *goodSituation* takes as first argument a concrete situation and as second parameter a pattern. It returns true if the situation fits to the pattern and false otherwise. A situation and a pattern are almost the same, except that a situation does not contain the Anything value and contains all concerns exactly once. For example, with this pattern *[V "A", Anything, V "B", Anything]* the following situations return:

- *[V "A", V "B"]* : True
- *[V "B", V "A"]* : False, because the order is not good
- *[V "A", V "C", V "B", V "D"]* : True
- *[V "C", V "A", V "B", V "D"]* : False, because the situation does not start by A

After those explanations, the function *determineViewPriority* described in Listing 4.28 can be fully explained. As detailed before, the neutral case tries to find a situation with the given rules, the safety case filters the rules and also tries to find a situation with *determineViewPrioritySafety*. The newest case works as the previous one but the list of rules is reverted because the function *determineViewPrioritySafety* applies the rules from the first one. The first strategy uses a function called *findCorrectSituations*. As we can see below, this function takes a list of views and a list of consistent rule views. With those parameters, it tries to generate a situation in which the views can be executed. The implementation of this function is:

```haskell
findCorrectSituations :: [RuleView] -> [[RuleView]] -> [[RuleView]]
findCorrectSituations listView lstRuleView = do
    x <- (perms (nub [x | x <- listView, x /= Anything]))
    guard $ isNothing (find (\y -> not (goodSituation x y)) lstRuleView)
    return x
        where
            perms :: Eq a => [a] -> [[a]]
            perms [] = [[]]
            perms xs = [ i:j | i <- xs, j <- perms $ delete i xs ]
```

Listing 4.30: findCorrectSituations function

To find a correct situation, it uses a mechanism called backtracking. It is an algorithm trying to satisfy some goals by finding alternative paths [9, 34]. Our goal is to find a situation that can fit all the rules. Here is a small example: we have five views respectively called A, B, C, D and E. We also have three constraints given by the rules:

- *[Anything, V "A", Anything, V "B", Anything]* : The view A must be before the view B.
- *[V "C", Anything]* : The view C must be the first
- *[Anything , V "B", V "E", Anything]* : The views B and E must immediately follow each other in this order.

With those three constraints, 4 situations could fit the rules. For example, we could have those following orders : ["C","A","B","E","D"], ["C","A","D","B","E"], etc. The goal of *findCorrectSituations* is to find these solutions. To do so, the first step is cleaning the list of views. It removes all the "Anything", and the *nub* function removes all duplicate elements. After that, the *perms* function generates all the permutations of the views. For our example, the function *perms* will generate 120 lists that contains five views. Thanks to the lazy evaluation of Haskell, the solutions of *perms* are not computed if they are not used. In the function *findCorrectSituations*, x will take each one of the generated values. Then, if this x passes the guard, it means that the situation fits the constraints and this value of x is returned. This implementation constructs the list of the situations that pass the guard.

```haskell
determineViewPrioritySafety :: [RuleView] -> [[RuleView]] -> [[RuleView]]
determineViewPrioritySafety listView [] =
    perms (nub [x | x <- listView, x /= Anything])
determineViewPrioritySafety listView (x:xs) =
    findMaxModel (perms (nub [x | x <- listView, x /= Anything])) x xs
        where
            findMaxModel sol [] [] = sol
            findMaxModel sol currentRule [] =
                case filter (\s -> (goodSituation s currentRule)) sol of
                    [] -> sol
                    res -> res
            findMaxModel sol currentRule (x:xs) =
                case filter (\s -> (goodSituation s currentRule)) sol of
                    [] -> findMaxModel sol x xs
                    res -> findMaxModel res x xs
```

Listing 4.31: determineViewPrioritySafety function

DetermineViewPrioritySafety uses another function called *findMaxModel*. First, the function *determineViewPrioritySafety* generates all the possibilities, just like in the *findCorrectSituations* function (Listing 4.30). Then, if the list of rules is empty, it means that the list must not be filtered, because all possibilities are corrects. If the list of rules is not empty, it calls the function *findMaxModel* with the list of all possibilities, the first rules et the remaining rules. *FindMaxModel* is a recursive function where the basic case is when no more rule has to be managed. In this case, it returns the remaining solutions. The recursive filters the list of solutions with the current rule. If there is no solution, it means that the rules are too strong and/or not consistent with the previous ones. Thus, we skip this rule (by keeping the previous list of solutions) and apply the following rules. If the filter gives a sublist of non-empty solutions, we apply the following rules to this sublist. Intuitively, it reduces the list of solutions by applying the rules one by one and skipping the ones that are not consistent with previous ones.

Let us come back to the function *determineViewPriority*, described in Listing 4.28. The Haskell *head* function is used to keep the first situation that fits the guard. Thanks to the lazy evaluation, it stops the generation of situations.

4.2.3.2 Planning: Ordering our concerns

The analysis has generated an execution order. *executeSeq* must now sort the concerns according to the order asked by the rules and it is the purpose of the planning step. It consists of a call to the function *orderConcern* that takes in parameters the order we want (a list of concerns' name) and the list of the concerns. This function will just retrieve the list of concerns, ordered according to the list of concerns' name.

```
1 orderConcern :: [String] -> [Concern s] -> [Concern s]
2 orderConcern [] [] = []
3 orderConcern (x:xs) (concern:ys) | x == (getConcernName concern) =
4                                      concern:(orderConcern xs ys)
5                                  | otherwise = case find (\c -> (getConcernName c)
                                       == x) ys of
6                                      Just x -> x:(orderConcern xs (concern:(delete
                                           x ys)))
7                                      Nothing -> error ("The concern " ++ (
                                           getConcernName concern) ++ " is not
                                           implemented")
8 orderConcern x y = error ("orderConcern x y : impossible case " ++ show x ++ " " ++
      show (map getConcernName y))
```

Listing 4.32: orderConcern function

Three cases are possible: there is no concern at all, in that case the list of concern names is empty, like the list of concerns. The second case is when the list must be ordered. Either the current concern is already in the right place, then we can keep it and call the function recursively with the remaining list, or the current concern is not at its correct place. Than, we have to find the correct concern and place it accurately. The last case is when the two other cases are not triggered. It means that the list of concerns' names and the list of concerns are not linked. This can happen when a rule asks a concern not given by the programmer, for example. In that case, it sends an error.

4.2.4 Execution

As explained at the beginning of this section, the function *executeSeq* represents the monitoring part and the execution part. The monitoring has already been explained. This

section is devoted to the execution step. *executeSeq* must return a pair, containing the updated source and a list of updated views. The attentive reader noticed that the type of the returned view list is a bit special. Indeed, in Haskell, a list has to be homogenous. It means that each element of a list must be of the same type. Basically, a list containing a string, an integer and a boolean is not possible in Haskell. It is a problem for us, because we would like to retrieve a list of updated views but each concern has its own view type.

After a lot of research, we found a way to bypass this problem (`https://wiki.haskell.org/Heterogenous_collections`). The trick consists of creating a master type and use the existential types of Haskell. This way, we hide to the Haskell compiler the real type of the list. Thus, the MasterView data says: no matter the type of v, as long as it respects some properties (showable, ...). That is why v is embedded in a data called MasterView. For the compiler, the list is homogenous: all items are typed as MasterView. However, it is not the case because each master view can contain any view thanks to existential quantification. We need to add an instruction for the compiler to notify the use of ExistentialQuantification.

```
1 {-# LANGUAGE ... , ExistentialQuantification, ... #-}
2 ...
3 data MasterView = forall v. (ToJSON v, Show v, NFData v) => MasterView v
4 deriving instance Show MasterView
5 ...
```

Listing 4.33: findCorrectSituations function

Now we have an ordered list of concerns (*concernsOrdered*) and we can really execute our modular system. In the Listing 4.23, the execution part goes from line 9 to line 14. The *foldM* function is used. It works like a basic fold but with monads. As a reminder, we work with an IO monad. This fold starts with a pair composed by the source and an empty list of views as accumulator. It applies the function *fn* to each concern of the list *concernsOrdered* (line 9). The function *fn* takes the current state of the accumulator and the current concern. Then, it calls the function *execBranch* with the current source (first part of the accumulator) and the current concern as parameters (line 13). The function *execBranch* retrieves a pair where the first part is the updated source and the second part is the updated view of this concern. To complete the function *fn*, the source of the accumulator must be updated and the view of the current concern must be added to the list of views in the accumulator (line 14). Here is the implementation of *execBranch*:

```
1  execBranch :: s -> Concern s -> IO (s,MasterView)
2  execBranch source (Concern _ bx analysisAndPlan) = do
3          let view = get bx source
4          viewUpdated <- return (case view of
5              Just x -> analysisAndPlan x
6              otherwise -> error "Impossible to generate the view with the
                   bidirectional tranformation given")
7          sourceUpdated <- return (case put bx source viewUpdated of
8              Just x  -> x
9              otherwise -> error "Impossible to generate the updated source with the
                   bidirectional tranformation given")
10         return (sourceUpdated, MasterView viewUpdated)
11 execBranch source (ConcernRemote _ bx url port secured serialize unserialize) = do
12         let view = get bx source
13         viewUpdatedStr <- case view of
14             Just x -> postRequest url port secured (serialize x)
15             otherwise -> error "Impossible to generate the view with the
                   bidirectional tranformation given"
16         viewUpdated <- return (unserialize viewUpdatedStr)
17         sourceUpdated <- return (case put bx source viewUpdated of
```

```
18              Just x  -> x
19              otherwise -> error "Impossible to generate the updated source with the
                    bidirectional tranformation given")
20          return (sourceUpdated, MasterView viewUpdated)
```

Listing 4.34: execBranch function

The *execBranch* function takes the source and the current concern as input. Two possibilities can occur, according to the type of concern. Indeed the behavior of the function changes if the concern is remote or not.
The first case is when the concern is local. The BiGUL function with get is called to obtain the view of the concern. When it is done, the analysis and planning steps obtain an updated updated view. Finally, the BiGUL function is called with put and retrieves pair containing the updated source and the updated view.
The second case is when the concern is remote. The first step is the same: the BiGUL get function obtains the view. But to recover the updated view, we have to send a HTTP POST request. To make this server request, we use the function postRequest that takes the information about the server and the content of the request. In our case, the content of the POST is the serialized view (*serialize x*). This function retrieves a string to unserialize. It is the updated view sent by the server. It replaces the analysis and planning steps of the first case. The last step is the same as before: the BiGUL put function is called to obtain the updated source and retrieves the pair.

```
1  postRequest :: String -> Int -> Bool -> String -> IO String
2  postRequest url port secured body = do
3      request' <- parseRequest ("POST " ++ url)
4      let request
5              = setRequestMethod "POST"
6              $ setRequestBody (RequestBodyLBS (Char8.pack body))
7              $ setRequestSecure secured
8              $ setRequestPort port
9              $ request'
10     mgr <- newManager tlsManagerSettings
11     let req = request { responseTimeout = responseTimeoutNone }
12     response <- Network.HTTP.Conduit.httpLbs req mgr
13     return $ Char8.unpack (getResponseBody response)
```

Listing 4.35: postRequest function

The function to create a post request uses two well-known libraries called Network.HTTP.Simple (`https://hackage.haskell.org/package/http-conduit/docs/Network-HTTP-Simple.html`) and Network.HTTP.Conduit (`https://hackage.haskell.org/package/http-conduit/docs/Network-HTTP-Conduit.html`). From lines 3 to 9, it creates the request with its properties. Line 11 created a request manager to handle a secured server. Line 12 removed the timeout of the request (this line has to be replaced in production to avoid infinite requests). Line 13 executes the request and the last line retrieves the content of the response.

Chapter 5

Experiments

This chapter is about the experimentation and evaluation of the whole system described during the previous chapters. Where our implementation attempts to answer some research questions, the evaluation part attempts to determine whether the answers provided actually solve the problems. Let's remember the question we tried to solve in this thesis. The first issue is how to propagate properly the changes inside a modular system with several subsystems. The second issue is about the resolution of the conflicts in a modular system. Finally, we wanted to create a framework able to respond accurately to unforeseen situations and to give the best answer to solve them.

The first issue is quickly solved by using BXs. Indeed, by the mathematical definition of BiGUL, if a function is correctly implemented and if it is a lens (see section GetPut and section PutGet) then the transformation between the source (Main System) and the view (Sub Systems) do not produce any side effect (pay attention, we suppose that the put is correctly implemented by the programmer to ensure the correctness of the BX). In other words, the get always produces the same view with a given source and the put produces the same source if there is no change in the view [21]. Based on this observation, we did not create experiments to prove the first issue.

The second issue is more complicated to analyze. Indeed, the evaluation requires to run our system without any conflict avoidance mechanisms, and then count the number of conflicts that occur during the tests. After that, running the system again with conflict avoidance mechanisms, and compare the results. However, during the explanation of our solution, we changed a little bit the problem. To avoid the conflicts, the system executes each subsystem sequentially where the last one overrides the previous ones. Thus, it is easy to understand that all conflicts are avoided with this solution. Although the provided solution avoids conflicts, it is not sure that the behavior of the system is still coherent and provides a correct solution. To evaluate this point, we chose to run the system on a real case (explained later) and see if the behavior seems to be coherent and correct. Clearly, this experiment does not *prove* anything, but shows if the system reacts in a way that we can justify. Moreover, this experiment depends on the chosen use case, and the implementation of each subsystem. As shown in section 5.2, we suppose that the entire system is correctly implemented and that each subsystem has a coherent behavior.

The last issue is to provide an efficient framework which reacts to unforeseen situations. We provide a framework called *the synchronizer* that avoids conflict by reordering the concerns, according to a base of rules and the current context. Thus, in a certain way, our system can handle unforeseen conflicts between subsystems by reordering the concerns. The result of each concern does not matter for the synchronizer. In other words, the system can also respond to unforeseen situation about the context in which the system is. No matter the context, the synchronizer is able to reorder and execute the system according to the

rules. However, a good improvement would be to enrich the base of rules by some kind of machine learning algorithms, to increase the number of rules automatically (see section 6.3). To evaluate this point, we have to check if the behavior of the system is correct no matter the situation (even with unforeseen situations). Unfortunately, we did not have enough time to expand the experimentation and validate this point. However, we have made an evaluation of the performance to show the effectiveness of this system to react to a given situation (section 5.3).

The following sections will provide an experimental setup and describe in a more detailed way the evaluation already addressed.

5.1 Experimental setup

Firstly, all the experimentations are run on AWS, in the region of Tokyo. The backend architecture where our system runs is made up of six server instances of type *t2.micro*. One for the controller/synchronizer, another to simulate the load, and each of the four others dedicated to one of the concerns. All of the instances are inside the same security group, to allow the communication with each other. As you can see, our system runs only on five servers and we use a sixth to simulate the load. Thus, each of the five servers contains our system, but the controller/synchronizer has the AWS CLI and Ansible installed and configured, to communicate with AWS. To let the reader easily understand, we named each server as follow :

- $Server_{Controller}$ for the server that executes the monitor and executes parts of the first MAPE loop and also runs the synchronizer.
- $Server_{Load}$ for the server that simulates the load of the test infrastructure.
- $Server_{AutoScaling}$ for the server that executes the auto scaling analysis and planning steps (subsection 3.3.3).
- $Server_{Redundancy}$ for the server that executes the redundancy analysis and planning steps (subsection 3.4.3).
- $Server_{Firewall}$ for the server that executes the firewall analysis and planning steps (subsection 3.5.3).
- $Server_{Cost}$ for the server that executes the cost analysis and planning steps (subsection 3.6.3).

The communication between the servers works as follows: the $Server_{Controller}$ monitors the AWS API to recover the data in JSON. Then, they are parsed to obtain the source, and the synchronizer (always $Server_{Controller}$) can find the best order in which the views must be executed, thanks to the rules and the context. Once the synchronizer has finished, the $Server_{Controller}$ parses the views in JSON and sends them to their respective server ($Server_{(AutoScaling,Redundancy,Firewall,Cost)}$) in the correct order. Each concern can now make its analysis and planning steps (which may or may not change the view), and transfer it to the $Server_{Controller}$, to finally update the whole source. Finally, Ansible is used to submit all the changes of the concerns to Amazon. All the communication are made by HTTP. For example, when the $Server_{Controller}$ sends the view to $Server_{AutoScaling}$, in fact the first one makes a simple HTTP GET request to the $Server_{AutoScaling}$ that analyzes and plans some changes and returns the updated view as HTTP response.

Now that the installation of our system is explained, let's describe the infrastructure that the system manages. The front-end architecture is composed of two security groups: database-securitygroup, for the database, and web-securitygroup, for the web workers. Usually, both allow SSH. The first one allows also MySQL, and the second one allows HTTP. The first one is always composed of a single server of type *t2.micro*, while the composition of the second varies between the different experiments. However, all the web workers (instances of the web-securitygroup) contain the last version of Apache[1] by default, and run the default Wordpress[2] website configured to communicate with the database. In addition to all this, we use a load balancer provided by AWS' Elastic Load Balancer Classic, to distribute the incoming traffic to the running web workers.

The four concerns are applied to the web-securitygroup, whereas only the security concern (ie. Firewall view) is applied to the database-securitygroup. If we had to run every concern on this last security group, we would have been obliged to synchronize every MySQL databases and reconfigure the MySQL server dynamically when a change is made. We have decided to simplify it by having only one instance in the database-securitygroup that is why concerns, that could decrease or increase the number of instances, are not run on this security group.

The $Server_{Load}$ has jMeter[3] installed to simulate a real use of the system. This software creates fake requests on the infrastructure. The number of requests per second is changed manually. It has the effect to change the load of the servers over time. Note that even if the number of requests is changed manually, we can determine the exact number of requests made by minute thanks to AWS Load Balancer Monitoring. This way, we can reproduce the same number of requests over time for a future experimentation.

Finally, we use the *safety* strategy, defined in subsubsection 4.2.1.5 to find the execution order of the views. The experimentation is also executed with the following base of rules:

- AdvertisingRevenue == 0 : Cost, Redundancy, AutoScaling, , Firewall
- FireEmergency == True : Redundancy, Firewall, *
- SecurityEmergency == True : Firewall, *
- ((DayOfWeek == 6) and (HourOfDay > 18)) or ((DayOfWeek == 6) or (DayOfWeek == 7)) : Redundancy, AutoScaling, *
- (WeekOfYear >= 50) or (WeekOfYear == 1) : Redundancy, AutoScaling, *, Cost
- (Month == 1) or (Month == 7): Redundancy, AutoScaling, *, Cost
- ((HourOfDay > 6) and (HourOfDay < 10)) or ((HourOfDay > 17) and (HourOfDay < 22)) : AutoScaling, Redundancy, Firewall, *
- AdvertisingRevenue <= 200 : Cost, *
- PeriodOfReduction == True : Redundancy, AutoScaling, *, Cost
- (HourOfDay > 6) and (HourOfDay < 20) : AutoScaling, *, Cost
- (HourOfDay <= 6) and (HourOfDay >= 20) : Redundancy, Cost, Firewall, *

[1] https://httpd.apache.org
[2] https://wordpress.com
[3] http://jmeter.apache.org

The analysis and planning steps of each concern for this evaluation are set up with specific strategies already explained in chapter 3. As a reminder :

- The autoscaling needs one argument to work properly: the average load that each security group has to impose to its associated instances. In our case, this expected value is 40%. Of course, this is configurable and entirely depends on the user. The planning step thus adds instances when the calculated average is higher than the expected value, and shuts down instances when the calculated average is lower.
- The analysis of the redundancy is simpler and goes through each security group independently to find those which do not possess at least two running instances. The planning then starts servers on those groups, while paying attention to have always two instances of the same type. Indeed, if it starts a very weak instance next to a big one. If the big one fails, the small will not be able to handle the charge of requests.
- The analysis and planning steps of the firewall ensures that all the instances are reachable with a SSH access, that the traffic from the Internet to the web-securitygroup must be allowed on port 80 (http), and finally that the traffic from the Internet to the database-securitygroup must not be allowed on port 3306 (MySQL).
- The analysis of the cost reviews the whole system to compute the total cost. The associated planning step then shuts down some servers, to go under a predefined limit, and following this strategy: first will be shut down the instances with the lowest load. If the load is equal between instances, the most expensive is ended first. It allows us to handle the servers that a previous view wants to start, whose load is null, and prevent them to be launched before trying to concretely shut down instances that already run. This implementation strategy is quite simple and could be widely improved by allowing the cost to not only shut down instances but also trying to meet the demands of previous views. For example, if the autoscaling demands to start a big server, judged too expensive by the cost, it could try to launch a smaller server instead of completely preventing the big instance to be started.

To complete the experimental setup, you can also go to the GitHub of the project to see exactly and precisely all the configurations of each submodules.

5.2 Behavior evaluation

As already explained, the behavior evaluation tries to analyse the behavior of a running infrastructure when some stimulus are made. In our case, those stimulus are the change of the number of requests. The goal of this section is to see how the system reacts according to different contexts that begets a different execution order of the concerns. To accomplish this evaluation, we will change the context manually and see if what we expected is what the system does.

At the beginning, we have two webworkers typed *t2.nano* inside the web-securitygroup. Moreover, we have the following context on which the rules described above can apply: *("SecurityEmergency", B False), ("FireEmergency", B False), ("HourOfDay", I 14), ("DayOfWeek", I 3), ("WeekOfYear", I 20), ("Month", I 5), ("AdvertisingRevenue", I 250), ("PeriodOfReduction", B True)*. With this context and the rules described in section 5.1, the order in which the concerns are executed is: Cost, Firewall, Autoscaling, Redundancy. As a reminder, it means that the Redundancy concern is the most important. The autoscaling concern is configured to keep the average load at 40%, and the cost concern ensures that the cost stays under $0.5 per hour. Finally, our system is launched every 15 minutes.

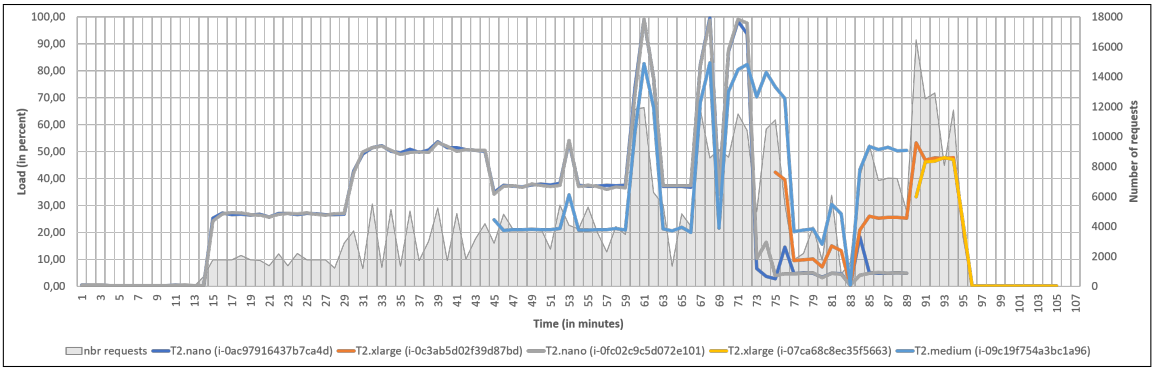


Figure 5.1: Behavior experiment - Autoscaling advantage

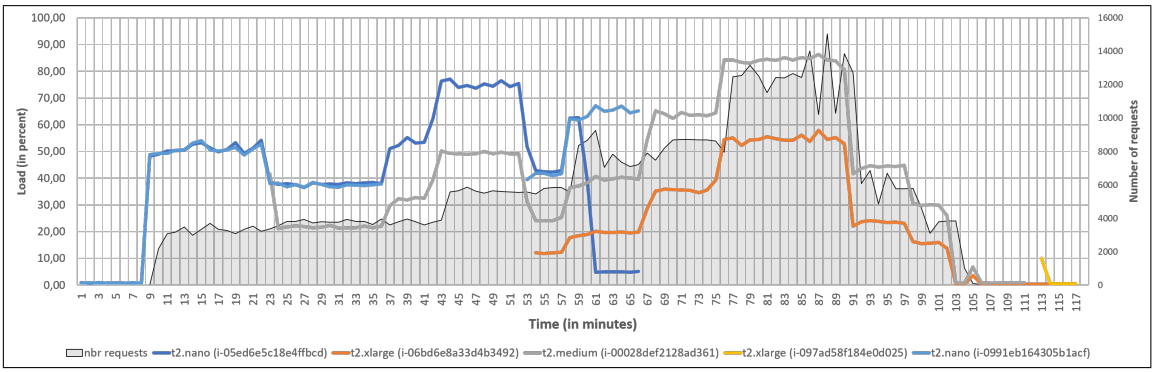


Figure 5.2: Behavior experiment - Cost advantage

Figure 5.1 shows the results obtained with all those data. The left axe represents the load in percentage; the bottom one is the time in minutes, and the right axe is the number of requests. The lines, which illustrate the instances, refer to the left, whereas the shaded shape refers to the right.

The expected behavior of this experiment is that the system keeps the load close to 40 percents because the AutoScaling is the most important concern after the Redundancy. That is why, even if the load is under 40 percent, the system will have to keep 2 running instances. Now, let's describe step by step how the system reacts.

- Minute 1 to 11 represent the launch of the system, without any load.
- At t-11, our system is launched and the situation does not need any changes.
- At t-13, the first requests arrive until t-26, where our system is launched and does nothing, because even if the load is higher than before, it is still below the threshold.
- T-30 sees a jump in the number of requests and the average load of the two servers is around 50%.
- At t-41, our system is executed and decides to create a new instance of type *t2.medium*. It is ready and started at t-45, which leads to a fall in the load of the two old instances.
- Between t-45 and t-59, the number of requests is constant, and despite the fact that our model is run at t-56, it does not do anything because the 3 running instances can handle the current charge and keep the load close to the expected one.
- From t-59 to t-64, there is a peak of requests absorbed by the three instances, with very high loads.

- As the system has been launched at t-71, during a sequence of peaks from t-65 to t-77, it starts a new *t2.xlarge* instance to handle this amount of queries.
- From that point, the load decreases drastically, because of two factors: the first one is the drop of the number of requests, and the second is the fact that the new big server takes over a lot of work.
- Thereafter, and until t-83, the number of requests is so low that the load is almost null.
- From t-83 to t-96, the load increases again due to a new peak of queries.
- Our system is launched at t-86 and decides to shut down the three smallest instances (2 *t2.nano* and 1 *t2.medium*), and to start a new *t2.xlarge*. It leads to a situation where we have only two running *t2.xlarge* instances. What is interesting is that it detects the large gap between the load of the instances, and smartly resolved the issue to obtain a situation where the instances possess the same expected load.
- In the end, despite the stop of the requests, the instances stay activated because the Redundancy concern requires always at least two running instances.

As you can see, the system follows the expected behavior by keeping the load close to 40 percents and also keeping at least 2 instances. The Cost is overwritten by the AutoScaling and Redundancy, because this result is obtained with the order mentioned earlier. However, changing this order can widely impact the behavior of the system. For example, if the context evolves and the *AdvertisingRevenue* is now null, the new calculated order becomes Firewall, Autoscaling, Redundancy, Cost (according to the rules described in section 5.1). The expected behavior of the experiment changes a little bit. Indeed, the system will work similarly but the total cost has to be under \$0.5. In other words, even if the redundancy or the AutoScaling request a new server, if the total cost with this change is above the \$0.5, the cost concern will erase this change.

Let's see the behavior of our system. The Figure 5.2 shows a similar situation but with that new order (ie. Firewall, Autoscaling, Redundancy, Cost). The beginning is the same and the reaction of the system works as expected. However, at t-65, the model is run and decides to shut down the two *t2.nano* instances. Once shut down, we have decided to manually remove them from AWS. With a user intervention, it highlights a real-case situation where the administrator removes completely some servers or where a server completely crashes. From t-75, the system notices an increase of the requests, and so an increase of the load. The system is executed again at t-80. Despite the fact that the autoscaling wants to start a new *t2.2xlarge* instance to handle the charge, the cost overwrites this decision and cancel the launch of this server, because the current system already runs two instances whose cost is \$0.232 per hour. If the system had launched a *t2.2xlarge* server, whose cost is \$0.3712 per hour, the total amount would have been \$0.6032, and that exceeds the limit predefined by the user.

We can see that the system works and follows what we expected. However, we also see that the cost concern could be improved. Indeed, the cost just shuts down instances already in our infrastructure, but it could use some new instances to approach the limit cost. In our experiment, the kept cost is \$0.232 per hour. There is a gap between this cost and \$0.5, that could be filled by running some new instances that keep the cost of the infrastructure under the limit. Unfortunately, the cost concern cannot run any instance but just shutting them down. Thus, the experiment shows that the behavior is correct (ie. follows the defined strategies) but is not really efficient.

Table 5.1: Efficiency experiment (in ms)

Instances	Controller/Synchronizer		Cost			AutoScaling			Redundancy			Firewall			Total
	Parsing	Ordering	Get	A&P	Put	Get	A&P	Put	Get	A&P	Put	Get	A&P	Put	
100 Stopped	0.296	0.040	1.080	16.500	1.640	0.431	32.500	0.459	0.420	23.200	0.363	0.029	13.300	0.158	90.416
50 Stopped	0.298	0.040	1.110	14.900	0.164	0.428	25.200	1.750	0.250	19.000	0.927	0.030	12.800	0.139	77.036
50 Running	0.297	0.040	0.237	19.600	1.100	0.429	26.400	1.580	0.256	19.000	0.699	0.029	13.600	0.139	83.406

Let us change once again, without graphs, the context to see what happens when the Redundancy concern is less important. The day was productive, the *AdvertisingRevenue* is higher than zero. It is now 18 o'clock. The new calculated order is Cost, Firewall, Redundancy, Autoscaling. The difference with the other order is that when there is no more request, and the autoscaling is more important than the redundancy, all the servers are shut down. The decisive importance of the execution order of the views can be seen in this circumstances, where the final situation of the servers can completely change, depending on which view has the priority. Indeed, if the rules define an order without being very careful, it can lead to issues in the system. However, those problems come from the definition of the rules by the user, not from our model.

Note that the setup and the results of this experiment are available in our public GitHub `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations/tree/master/Experiments/Exp1`

5.3 Efficiency evaluation

The efficiency evaluation aims to analyze the system performances in some situation. To accomplish this purpose, we executed benchmarks to measure each part of our system. Even if this evolution depends exclusively on the machines where it is executed, we tried to reduce this trouble by using AWS. This way, the machine is well-known and the evaluation can be reproduced by anyone who wants it. Moreover, we use a Haskell library (called *criterion*) that allows us to get some statistics about our system. Because our work is not about the performances and statistics of a system ,we will limit ourselves to show and explain the mean execution time of each part of the software. Of course, *criterion* gives us lots of information and statistics. Thus, all the reports are available in the appendices, and all along this section we will reference specific ones.

Let's identify the most interesting parts of the system to benchmark. We chose to avoid the part of monitoring and execute of the first MAPE loop, because it is closely linked to the used Cloud services and it is not the real added value of our solution. Thus, we decided to focus on the four concerns, by the measuring of the Get, Put and analysis/planning steps. Moreover, we also evaluate the controller/Synchronizer by measuring the parsing of the rules and the ordering of the concerns (called "Synchronizer/Determine Order Last" and "Synchronizer/Determine Order Safety" in the report of *criterion*).

Clearly, table 5.1 represents the total time taken by each major step of the model, in milliseconds. Regarding the names of the columns, the *Parsing* is the parsing of the rules (taken from a .txt file) into their Haskell format. The *Ordering* is the whole process of choosing the best order in which the views will be executed. Finally, the *A&P* is the Analysis and Planning process.

The experiments are composed by three benchmark sets. The first situation[4] that we

[4]Setup, data and results : `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations/tree/master/Experiments/Exp2`

analyze is when the web security group contains 100 stopped instances (T2.Nano) (report in Appendix B). The database security group contains 1 instance (T2.micro). The configuration of the context is made to obtain the execution order of the concerns as follows (from less important to more important) : Cost (max 1,5/h), Firewall, Redundancy, Autoscaling (avr load 40%). The rest of the setup is exactly the same as the one explained during the section 5.1. The second situation[5] is the same then the previous one except that the number of stopped instances is 50 (report in Appendix C). Finally, the last situation[6] is the same then both previous ones except that all instances are in a running state (report in Appendix D). This way, we see if the state of the instances has any kind of impact on the performance of the system (and thus, if launch or shut them down has an impact).

Here are the explanations of the results. The reordering step of the views, while capital in the model, is extremely fast and does not impact the total time of the execution no matter the situation. As expected using the BiGUL programming language, the time taken by the bidirectional transformations is minimal: together, the forward and the backward transformations never last longer than 3 milliseconds. Even more: the *put* of the Autoscaling and the Cost are the only one exceeding 1.5 milliseconds. Obviously, most of the time is taken by the analysis and planning steps, which is pretty normal knowing that it is where the intelligence of the system is, and where the decisions must be made. Finally, the total time for the model to determine what to do with the servers is, in general, only a little bit less than 100 milliseconds, which can be interpreted as very efficient for a self-managed model that handles 100 servers. Moreover, it's easy to see that the state of the instances has no significative impact on the performance, even if we notice a tiny increase of the total time. Indeed, this little gap could be produced by other processes on the instances, that may have impacts on the performances of the instances. It is also important to notice that even if the number of instances doubles, the increasing of the total time is minor. It means that we could increase the number of instances without increasing much more the time of the execution.

5.3.1 Validity of the results

A particular attention must be paid to those results. Indeed, they can be affected by several internal and external factors. Firstly, this evaluation depends on the IaaS on which it is executed. Even if AWS is used again, the results may differ depending on the region chosen by the user (this evaluation is made with the Asia Pacific (Tokyo) region), but also on the potential updates of the Amazon infrastructure and, of course, on the use of the Amazon servers by other clients. Generally speaking, the results can change every time a version of one of the tools used is upgraded. Using those tools is also a proof to have faith in our evaluation. AWS is a worldwide service, employed by hundreds of companies, as well as Ansible.

[5] Setup, data and results : `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations/tree/master/Experiments/Exp3`

[6] Setup, data and results : `https://github.com/qlombat/Self-Prioritized-Modular-Adaptations/tree/master/Experiments/Exp4`

Chapter 6

Conclusion and Future Works

This chapter ends the thesis. It summarizes the essential results and the lessons to be learned from our work. We will also discuss new issues that our work has raised and possible future works.

6.1 A Modular System Using Bidirectional Transformations (BiGUL)

As we saw, the modularization is a common strategy for self-adaptive systems. This work has presented a system using bidirectional transformations to synchronize the subsystems. We introduced some useful bidirectional patterns suppose to help us during the development. We also explained what is a *lense* and how the BXs work. Then, we detailled some previous work about modular systems to situate the thesis in the domain of modular self-adaptive systems. The main issue of this part of the thesis is on the utility of bidirectional transformations in a modular system. Our work consisted initially to develop a modular system using a put-based bidirectional language. Because we did an internship at the National Institute Informatics in Tokyo, we chose to use BiGUL, the language developed by the team we were working with. Although it was not too difficult for us to develop our example, we encountered difficulties with the language. Indeed, apart from the fact that bidirectional programming can be difficult to grasp, the syntax of BiGUL is sometimes problematic and does not allow a quick understanding of the problem. Moreover, developing bidirectionally takes time and it is sometimes superfluous. Indeed, on a very large system where we need to be sure that the synchronization is done correctly, the use of languages such as BiGUL can be a real advantage for the maintenance of the software. However, on a smaller system, the use of this type of language can be more complicated since the principle is to encode only one of the two transformations, the analysis of the problem is generally more complicated and the debugging is sometimes complicated and long.

As for our example strictly speaking, the solution we provided to manage a cloud infrastructure is sometimes too simple. Indeed, due to the lack of time during the development, some concerns are limited and not totally realistic. For example, the concern about the security is largely hard coded with only a few rules. However, even with those limited concerns, the example system that we created is enough to illustrate the usefulness of the bidirectional programming. Moreover, it runs on AWS, which is one of the most used cloud platforms.

All along this work, we have personally seen the interest of BiGUL. The use of bidirectional programming, and especially BiGUL, has been a real advantage. As you already saw, it provides useful tools like *align*, *rearrS* or *rearrV* to synchronize data. During the devel-

opment step, the language is very constraining in order to obtain a correct synchronization, but when the system is developed the maintenance time is greatly reduced.

6.2 A Way to Orchestrate a Modular System

In a second step and once the modular system was done, our job was to improve it to make it self-adaptive. After a small review of other fields of research, we tried to elaborate our own system based on context and rules. The first one is a snapshot of the situation in which the system is, and the second one is the knowledge of our system, based on rules choosing the way to order each subsystem. As already explained, we chose a rule-based approach, and the main issues of this second part of the thesis was:

- Is it possible to make a modular self-adaptive system using BiGUL (1)?
- How to avoid conflicts between different views (2)?
- How to execute different subsystems in different orders (3)?

The answers to these three questions were seen in details during chapter 4. Indeed, all along this thesis we have worked on an example system, and at the end of this work (1) we managed to run a system that adapts to the environment through a context and a knowledge base. Even if the knowledge base and the context was simple, we were able to show a concrete example of a self-adaptive system managing a modular system using BiGUL. During this work, to make our system self-adaptive we faced a problem when data is shared between different submodules. To solve it, we were inspired by non-self-adaptive systems, in which the static execution order of the subsystems simulates a priority among the modules. As explained earlier, we have chosen to change dynamically the execution order of subsystems such that the last executed subsystem can override the changes made by previous ones. In this way, each subsystem has a priority and conflicts are avoided (2). Finally, we thought about how to execute subsystems and explained the sequential approach and the parallel approach (3). We then have been able to show that the parallel approach strongly relies on the interdependence of the subsystems. Thus, it can sometimes be very beneficial in terms of time for a quantity of calculus compared to the sequential approach, but it can also be catastrophic if the data are strongly shared.

Even if we found a solution to those questions, the answers are not necessarily the best. Indeed, we showed that it is possible to have a modular self-adaptive system using BiGUL avoiding conflicts but some of our algorithms are not optimized and can be improved. For example, the way we determine the order of execution of concerns uses backtracking and could be widely optimized. Moreover, we flew over the parallel approach and further investigation might be interesting. As for the evaluation of our solution, due to the lack of time, we could not be more specific about some essential measures. For example, it might have been interesting to determine the performance of our solution if the number of rules, the number of subsystems or the number of variables in the context increased. Also, the proposed implementation can be simplified and improved to avoid recalculations or to choose other methods, more efficient.

In spite of this, all the questions we wanted to answer at the beginning of the internship have found their own answer. Our system, publicly available on GitHub, is running and has demonstrated its results in an efficient manner.

6.3 Future Works

Here are presented ways of enhancements. We had those ideas during the internship, but they were unrealistic to implement according to the time we had.

6.3.1 Maintenance on the fly

One of the goals of a modular system is to make maintenance easier. One interesting future work could focus on changes in sub-modules done on the fly. Indeed, what would be the impact of a modular system if one of its subsystems is in maintenance or change of behavior. How to avoid side effects and what exactly are the effects that this could produce. By extension, how to integrate a new sub-module into an executing system.

These questions are interesting due to the nature of a modular system whose purpose is to breaking up the complexity of the system and make it easier to maintain and more scalable.

6.3.2 Automatically learn the rules

Another interesting area of research is how to build the knowledge base. In fact, during this work we added manually the rules that seemed to us relevant. However, systems of this kind already exist and it may be interesting to learn from previous experiences using for example machine learning. Thus, the system could, on one hand, increase its knowledge base with rules that humans were not aware of, and on the other hand, it would make our system more self-adaptive by allowing it to learn from its own experiences. Thus, to react better to new situations. This improvement could completely remove the human from the process, and making the system completely autonomic.

6.3.3 Reverse the order of execution (from the highest priority to the smallest)

A bias of this work was the fact that each submodule can rewrite on the changes made by the previous ones, which gives it a higher priority. A possible future work would be to think the other way around, namely that the highest priority submodule is executed first. Then the following modules adapt according to the results of the previous ones in order to respect what they asked, while applying the changes of the current subsystem.

6.3.4 Merge rather than synchronize

We choose to synchronize the subsystems. In other words, we had to give a priority to the subsystems and execute the whole system according to those priorities. However, an interesting future work would be to no longer use priority but to try to merge the results obtained by the subsystems, thanks to artificial intelligence techniques for example. In this way, we could try to find a way to react to a conflict only a human could have solved, but totally automated.

6.3.5 Concerns with same priority

All along this thesis, we assumed that each concern has different priorities. In the real world, several subsystems can have the same priority. In this case, we must think of finding a way to merge and resolve conflicts. We could also improve the rules notations to specify which subsystems have the same priority. A simple solution to this problem would be to say that the execution order of subsystems does not matter, and that if such a case arises

we can choose any order of execution. However, this solution is not good because it is not deterministic: it could give different results with the same input data.

6.3.6 Improve the way to find a good situation

As you have seen, to find a situation that suits a given context, we generate the possible solutions and test if they are compatibles with the rules. Work could be done on how to determine this situation. Indeed, instead of starting from a set of situations, we could deduce a situation directly from the rules. Or, we could also simplify the rules thanks to simplifying patterns. For example, we intuitively know that if we have the rule $*, A, *, B, *$ and the rule $*, B, *, C, *$, it could be reduce by the rule $*, A, *, B, *, C, *$ that has the same meaning, but reduces the number of rules to check when we have a situation.

Bibliography

[1] Ansible. How ansible works | ansible.com. `https://www.ansible.com/overview/how-ansible-works`. (Accessed on 04/22/2018).

[2] Arcaini, P., Riccobene, E., and Scandurra, P. Modeling and analyzing mape-k feedback loops for self-adaptation. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Piscataway, NJ, USA, 2015), SEAMS '15, IEEE Press, pp. 13–23.

[3] Baldwin, C. Y., and Clark, K. B. *Design Rules, Vol. 1: The Power of Modularity*, vol. 1. The MIT Press, March 2000.

[4] Barbosa, D. M. J., Cretin, J., Foster, N., Greenberg, M., and Pierce, B. C. *Matching lenses: alignment and view update.* University of Pennsylvania, École Polytechnique, 2010.

[5] Barna, C., Ghanbari, H., Litoiu, M., and Shtern, M. Hogna: A platform for self-adaptive applications in cloud environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Piscataway, NJ, USA, 2015), SEAMS '15, IEEE Press, pp. 83–87.

[6] Boehm, B. Improving and balancing software qualities. In *Proceedings of the 38th International Conference on Software Engineering Companion* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 890–891.

[7] Chandrasekaran, B. Generic tasks in knowledge-based reasoning: High-level building blocks for expert system design.

[8] Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. Jtl: A bidirectional and change propagating transformation language. In *Software Language Engineering* (2011), B. Malloy, S. Staab, and M. van den Brand, Eds., Springer Berlin Heidelberg, pp. 183–202.

[9] Clocksin, W. F., and Mellish, C. S. *Programming in Prolog, Fifth Edition.* Springer, 2012.

[10] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations* (Berlin, Heidelberg, 2009), R. F. Paige, Ed., Springer Berlin Heidelberg, pp. 260–283.

[11] de Lemos, R., Giese, H., Muller, H. A., and Shaw, M. Software engineering for self-adaptive systems: A second research roadmap.

[12] Englebert, V. Software architectures engineering : Technologies and methods. Academic lesson, University of Namur, Namur, Belgium, 2016.

[13] Filman, R., Elrad, T., Clarke, S., Aksit, M., and Unknown, U. *Aspect-Oriented Software Development*. Addison Wesley, 2004.

[14] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering.

[15] Fischer, S., Hu, Z., and Pacheco, H. The essence of bidirectional programming.

[16] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems 29*, 3 (2007), 17.

[17] Fradet, P., Le Métayer, D., and Périn, M. Consistency checking for multiple view software architectures. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (London, UK, UK, 1999), ESEC/FSE-7, Springer-Verlag, pp. 410–428.

[18] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow: Architecture-based self adaptation with reusable infrastructure.

[19] Gat, E. Artificial intelligence and mobile robots. MIT Press, Cambridge, MA, USA, 1998, ch. Three-layer Architectures, pp. 195–210.

[20] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K. Groundtram: An integrated framework for developing well-behaved bidirectional model transformations. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2011), ASE '11, IEEE Computer Society, pp. 480–483.

[21] Hu, Z., and Ko, H.-S. *Principles and Practice of Bidirectional Programming in BiGUL*. Springer International Publishing, Cham, 2018, pp. 100–150.

[22] Huebscher, M. C., and McCann, J. A. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv. 40*, 3 (Aug. 2008), 7:1–7:28.

[23] Iglesia, D. G. D. L., and Weyns, D. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst. 10*, 3 (Sept. 2015), 15:1–15:31.

[24] Ko, H.-S., Zan, T., and Hu, Z. Bigul: A formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New York, NY, USA, 2016), PEPM '16, ACM, pp. 61–72.

[25] Kramer, J., and Magee, J. Self-managed systems: An architectural challenge. In *2007 Future of Software Engineering* (Washington, DC, USA, 2007), FOSE '07, IEEE Computer Society, pp. 259–268.

[26] Kurtev, I. State of the art of qvt: A model transformation language standard. In *Applications of Graph Transformations with Industrial Relevance* (Berlin, Heidelberg, 2008), A. Schürr, M. Nagl, and A. Zündorf, Eds., Springer Berlin Heidelberg, pp. 377–393.

[27] Lopes, C. V., and Bajracharya, S. K. An analysis of modularity in aspect oriented design. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development* (New York, NY, USA, 2005), AOSD '05, ACM, pp. 15–26.

[28] Mazaneka, S., and Hanus, M. Constructing a bidirectional transformation between bpmn and BPEL with a functional logic programming language.

[29] Montrieux, L., Yu, Y., Wermelinger, M., and Hu, Z. Modular hierarchical self-adaptation using bidirectional transformation. Unpublished, 2016.

[30] Sowa, J. F. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, 2000.

[31] Stevens, P. Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling 9*, 1 (Dec 2008), 7.

[32] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. The structure and value of modularity in software design. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2001), ESEC/FSE-9, ACM, pp. 99–108.

[33] Suwa, M., Scott, A. C., and Shortliffe, E. H. *An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System*, vol. 3. AI Magazine, Stanford, California, 1982.

[34] Vanhoof, W. Functional and logical programming. Academic lesson, University of Namur, Namur, Belgium, 2014.

[35] Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., and Goschka, K. M. On patterns for decentralized control in self-adaptive systems.

[36] William D., HurleyKyle D., H. Collaborative model for software systems with synchronization submodel with merge feature, automatic conflict resolution and isolation of potential changes for reuse, 01 2004.

[37] Yan, S. Review of dynamic fuzzy logic and its applications by fanzhang. *SIGACT News 42*, 4 (Dec. 2011), 48–49.

[38] Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., and Montrieux, L. Maintaining invariant traceability through bidirectional transformations. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 540–550.

[39] Zhu, Z., Ko, H.-S., Martins, P. M. R., Saraiva, J. A., and Hu, Z. Biyacc: Roll your parser and reflective printer into one. CEUR-Ws, Ed., 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations, pp. 43–50.

Appendices

Appendix A

Example of Ansible file

```
- name: Main task
  gather_facts: False
  hosts: localhost
  # Variables of the ansible file that can be used later in the file
  vars:
    keypair: Quentin
    image: ami-1955cc7f
    region: ap-northeast-1
    aws_access_key: AKIAJ5VAHO5CNO77AABA
    aws_secret_key: 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
    aws_load_balancer: BiGUL-CloudBx
    aws_tags : {"BiGUL":"CloudBx"}
  # Tasks that ansible will permform
  tasks:
    # Task to update the web security group
    - name: Security group
      ec2_group:
        name: web-securitygroup
        description: web-securitygroup
        tags: {{aws_tags}}
        aws_access_key: "{{aws_access_key}}"
        aws_secret_key: "{{aws_secret_key}}"
        region: "{{region}}"
        # Inbound rules (allow port 22 and 80 for webworkers)
        rules:
          - proto: TCP
            from_port: 22
            to_port: 22
            cidr_ip: 0.0.0.0/0
          - proto: TCP
            from_port: 80
            to_port: 80
            cidr_ip: 0.0.0.0/0
        # Outbound rules (allow port 22 for webworkers)
        rules_egress:
          - proto: TCP
            from_port: 22
            to_port: 22
            cidr_ip: 0.0.0.0/0
          - proto: -1
            cidr_ip: 0.0.0.0/0
    # Task to update the database security group
    - name: Security group
      ec2_group:
        name: database-securitygroup
        description: database-securitygroup
        tags: {{aws_tags}}
        aws_access_key: "{{aws_access_key}}"
        aws_secret_key: "{{aws_secret_key}}"
        region: "{{region}}"
        # Inbound rules (allow port 22 and 3306 for databases)
        rules:
          - proto: TCP
            from_port: 22
            to_port: 22
```

```
            cidr_ip: 0.0.0.0/0
          - proto: TCP
            from_port: 3306
            to_port: 3306
            cidr_ip: 0.0.0.0/0
        # Outbound rules (allow port 22 for databases)
        rules_egress:
          - proto: TCP
            from_port: 22
            to_port: 22
            cidr_ip: 0.0.0.0/0
          - proto: -1
            cidr_ip: 0.0.0.0/0
    # Task to stop instances
    - name: stop instances
      ec2:
        aws_access_key: "{{aws_access_key}}"
        aws_secret_key: "{{aws_secret_key}}"
        region: "{{region}}"
        instance_ids: [i-041c273779852aff6,i-05b13366fb09a86d3,i-0db0dbe01fce299ff]
        state: stopped
        wait: yes
        monitoring: yes
```

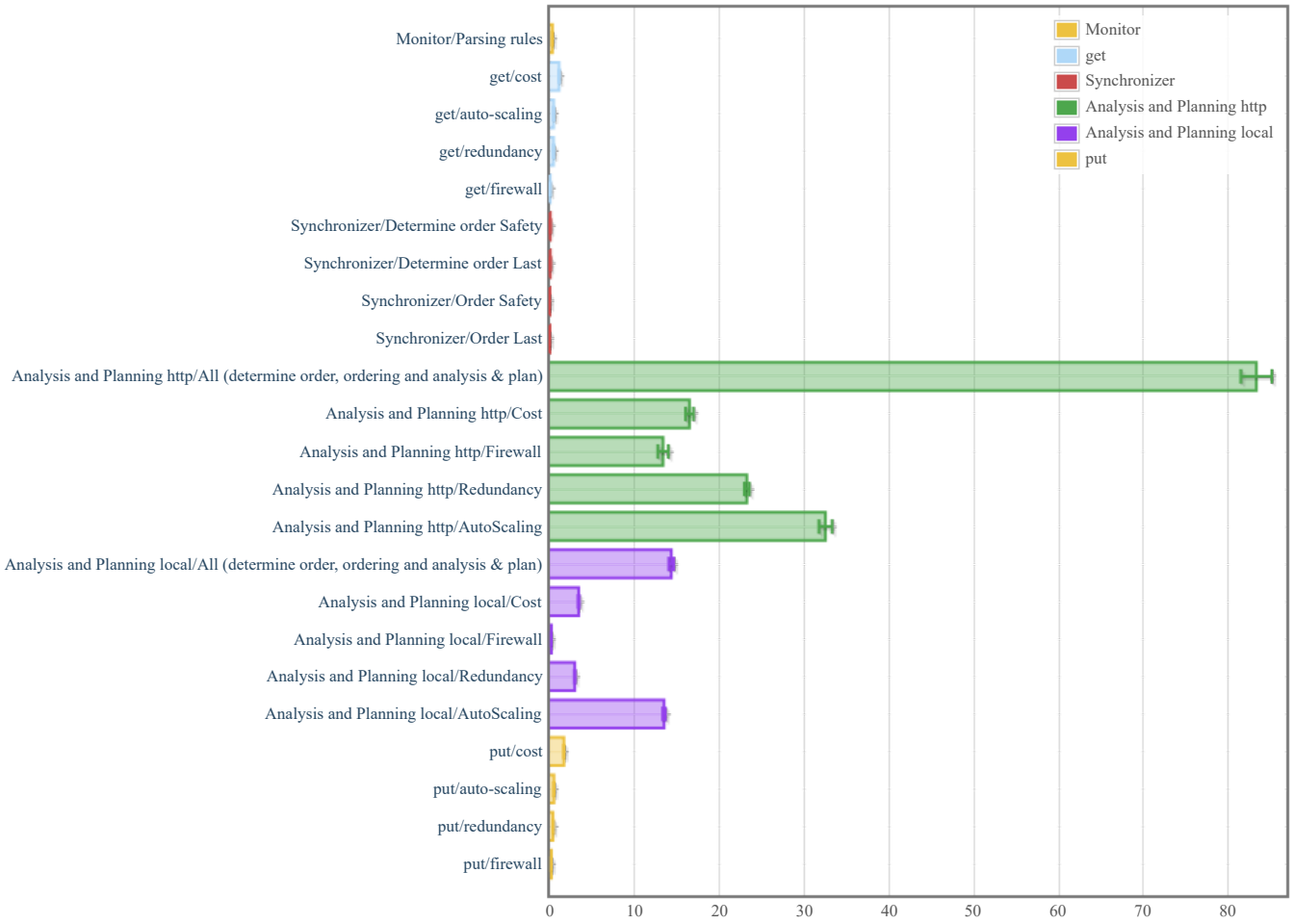
Appendix B

Experiment 2

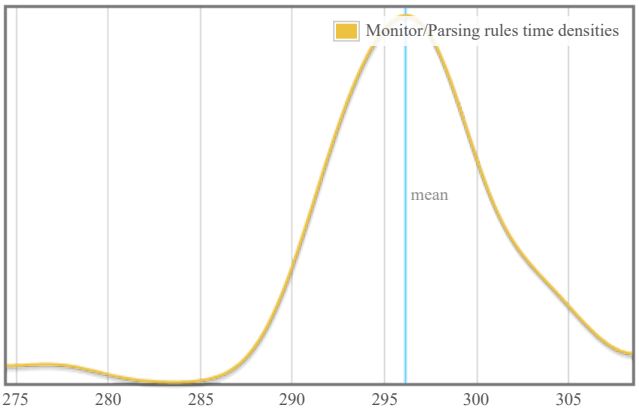
criterion performance measurements

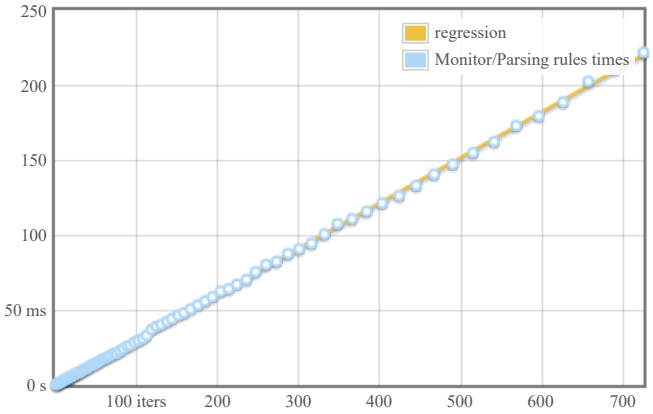
overview

want to understand this report?



Monitor/Parsing rules

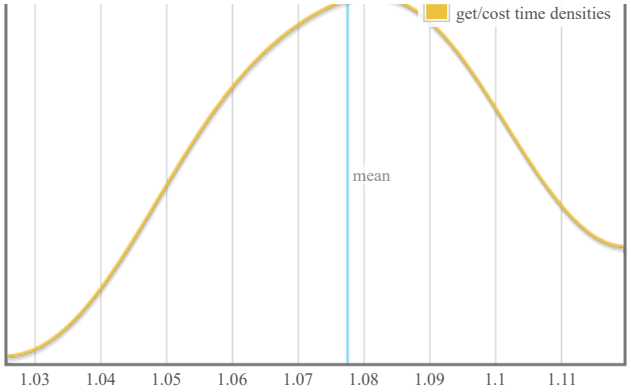


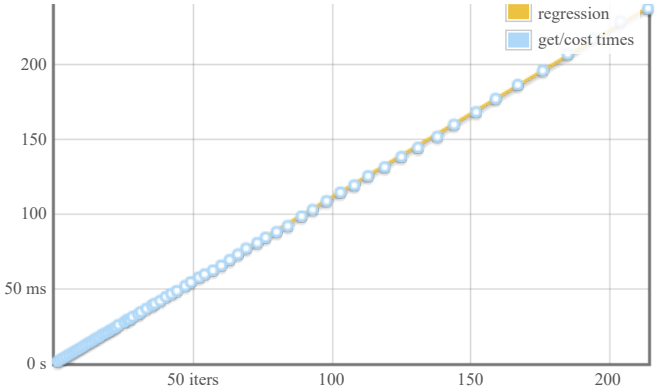


	lower bound	estimate	upper bound
OLS regression	302 µs	304 µs	306 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	295 µs	296 µs	297 µs
Standard deviation	3.33 µs	4.75 µs	8.17 µs

Outlying measurements have slight (8.1%) effect on estimated standard deviation.

get/cost

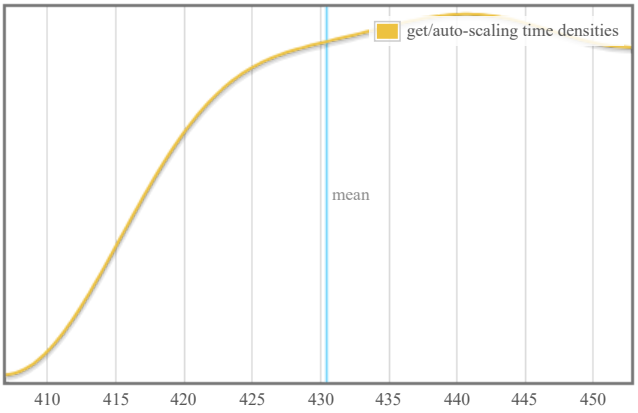


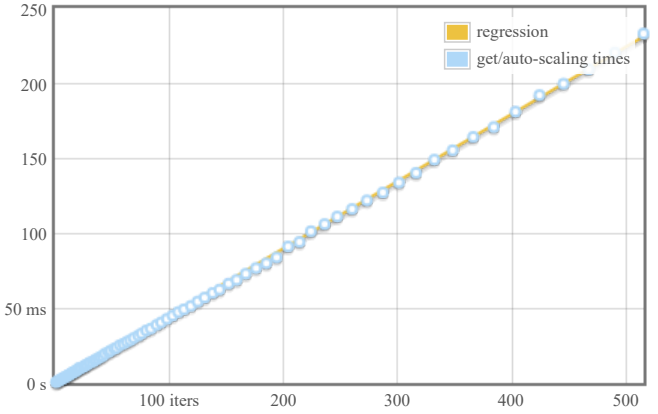


	lower bound	estimate	upper bound
OLS regression	1.11 ms	1.11 ms	1.12 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.07 ms	1.08 ms	1.08 ms
Standard deviation	16.4 µs	19.1 µs	22.8 µs

Outlying measurements have slight (8.3%) effect on estimated standard deviation.

get/auto-scaling

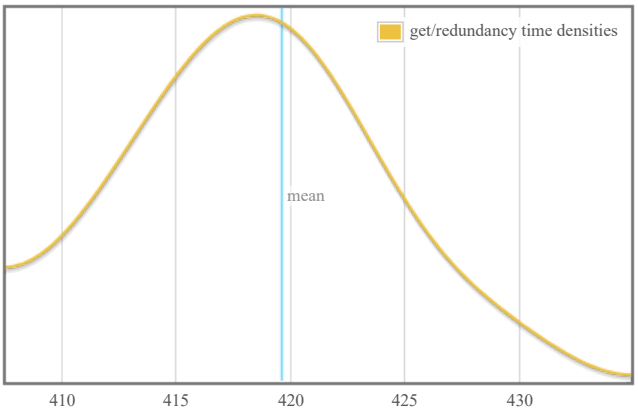


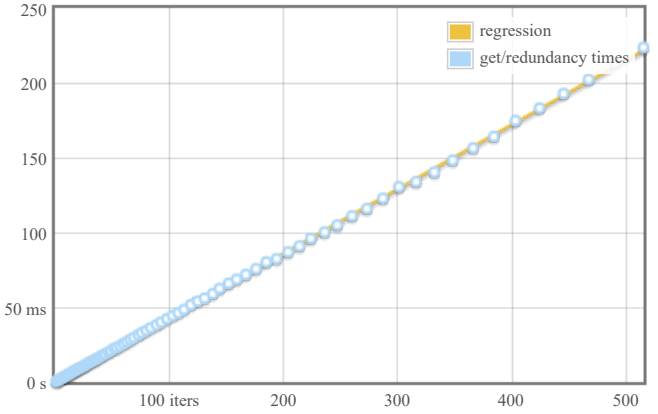


	lower bound	estimate	upper bound
OLS regression	446 µs	449 µs	450 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	427 µs	431 µs	434 µs
Standard deviation	10.8 µs	11.9 µs	13.6 µs

Outlying measurements have moderate (19.9%) effect on estimated standard deviation.

get/redundancy

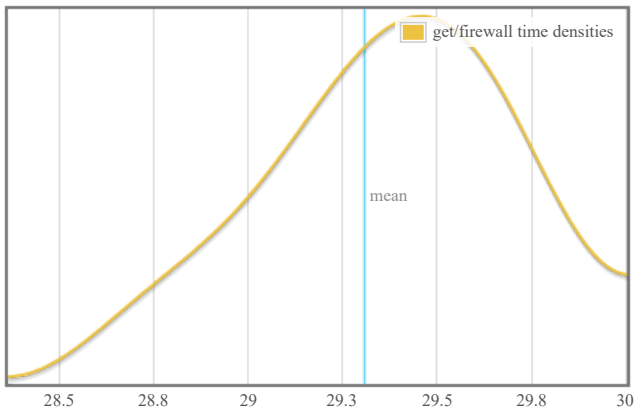


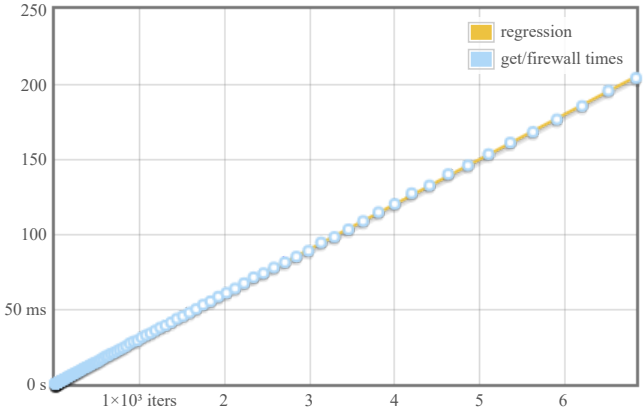


	lower bound	estimate	upper bound
OLS regression	428 µs	431 µs	433 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	418 µs	420 µs	422 µs
Standard deviation	4.93 µs	6.01 µs	7.14 µs

Outlying measurements have slight (6.5%) effect on estimated standard deviation.

get/firewall

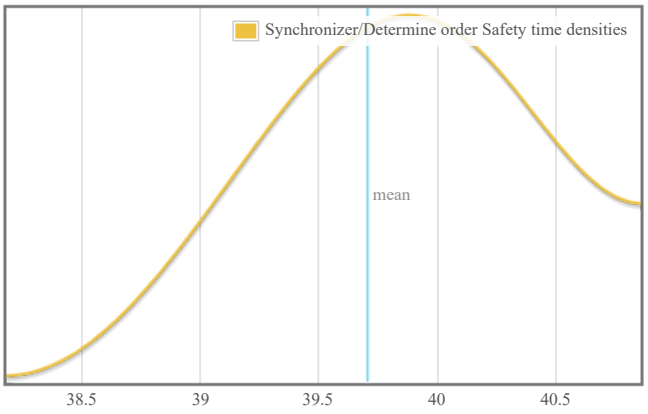


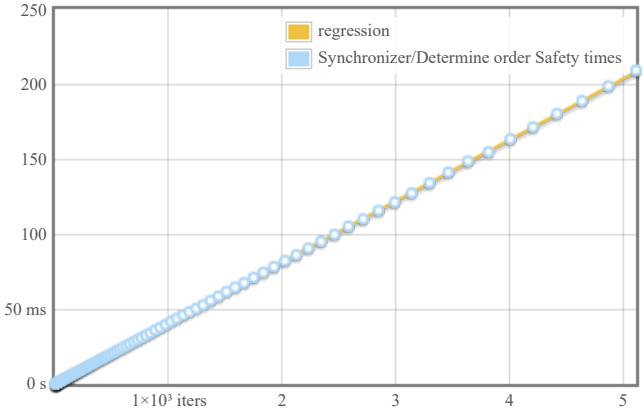


	lower bound	estimate	upper bound
OLS regression	29.9 µs	30.0 µs	30.0 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	29.2 µs	29.3 µs	29.4 µs
Standard deviation	295 ns	351 ns	431 ns

Outlying measurements have slight (6.8%) effect on estimated standard deviation.

Synchronizer/Determine order Safety

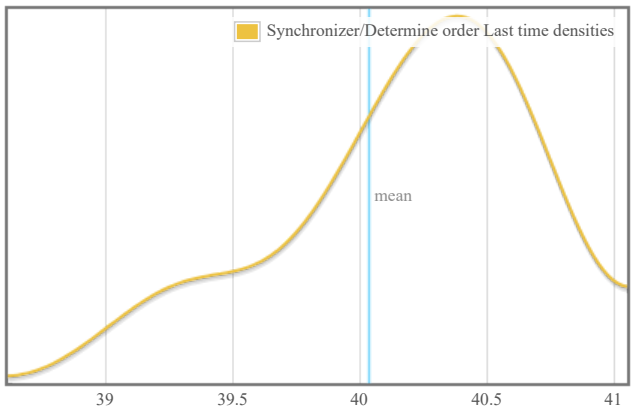


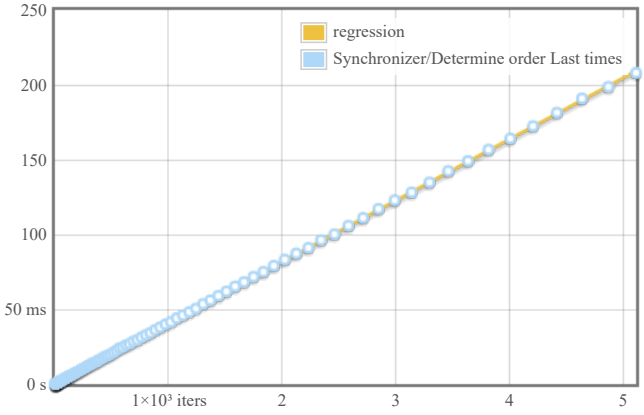


	lower bound	estimate	upper bound
OLS regression	40.7 µs	40.8 µs	40.8 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	39.5 µs	39.7 µs	39.9 µs
Standard deviation	488 ns	582 ns	718 ns

Outlying measurements have slight (9.3%) effect on estimated standard deviation.

Synchronizer/Determine order Last

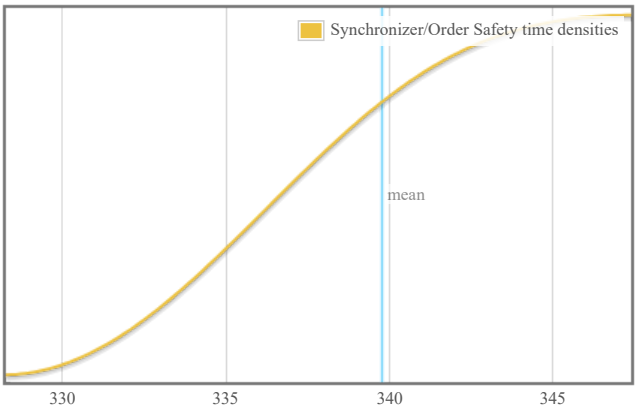


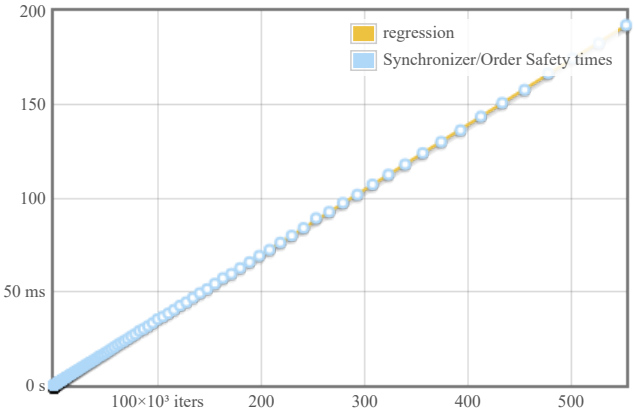


	lower bound	estimate	upper bound
OLS regression	40.9 µs	41.0 µs	41.1 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	39.8 µs	40.0 µs	40.2 µs
Standard deviation	480 ns	574 ns	690 ns

Outlying measurements have slight (9.2%) effect on estimated standard deviation.

Synchronizer/Order Safety

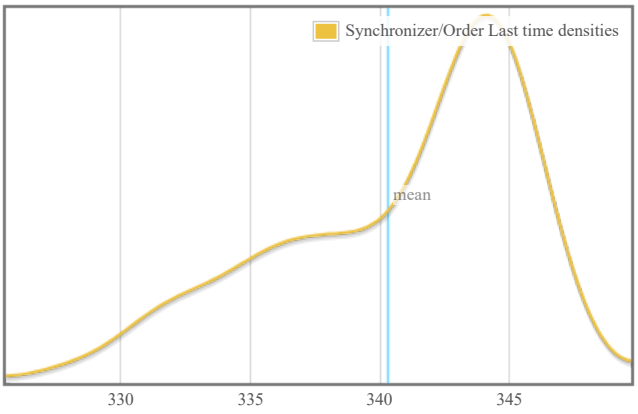


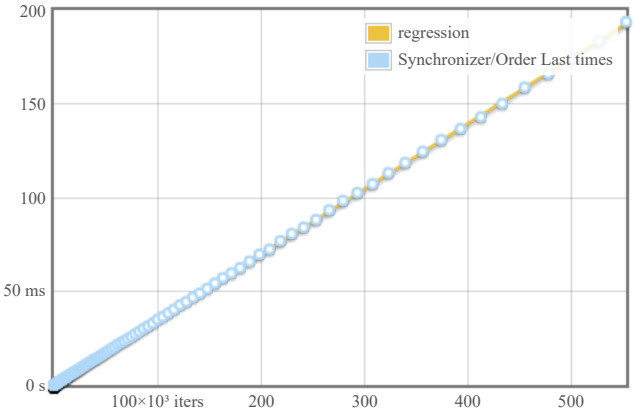


	lower bound	estimate	upper bound
OLS regression	348 ns	348 ns	349 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	338 ns	340 ns	341 ns
Standard deviation	3.76 ns	4.54 ns	5.43 ns

Outlying measurements have moderate (12.8%) effect on estimated standard deviation.

Synchronizer/Order Last

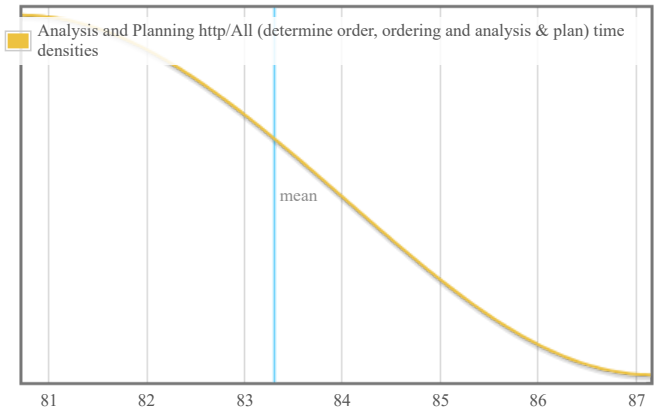


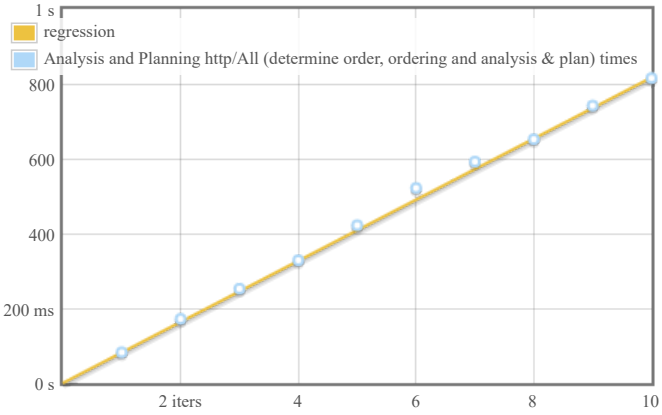


	lower bound	estimate	upper bound
OLS regression	348 ns	349 ns	350 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	338 ns	340 ns	342 ns
Standard deviation	4.28 ns	5.15 ns	6.35 ns

Outlying measurements have moderate (16.2%) effect on estimated standard deviation.

Analysis and Planning http/All (determine order, ordering and analysis & plan)



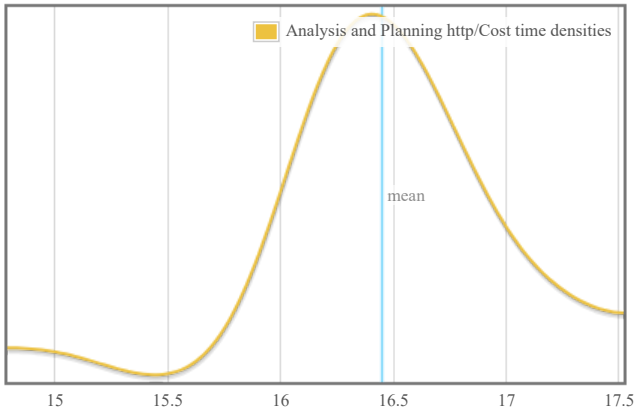


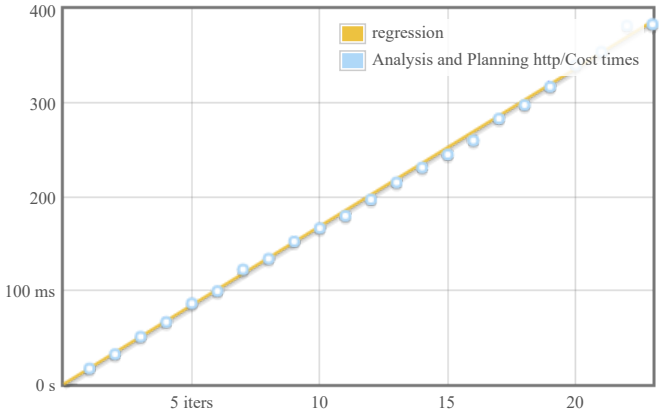
	lower bound	estimate	upper bound
OLS regression	79.4 ms	81.7 ms	85.4 ms
R² goodness-of-fit	0.995	0.998	1.000

Mean execution time	82.4 ms	83.3 ms	84.7 ms
Standard deviation	1.31 ms	1.83 ms	2.58 ms

Outlying measurements have slight (9.0%) effect on estimated standard deviation.

Analysis and Planning http/Cost

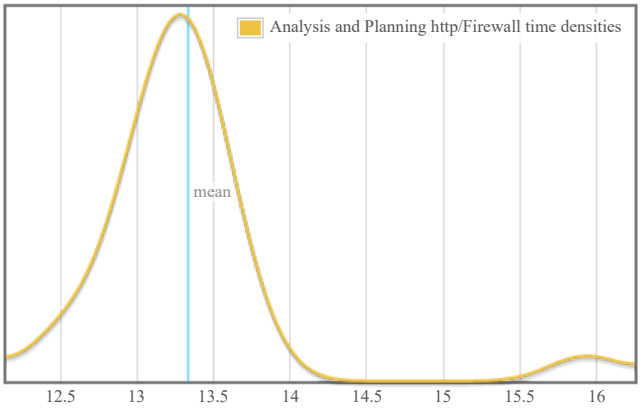


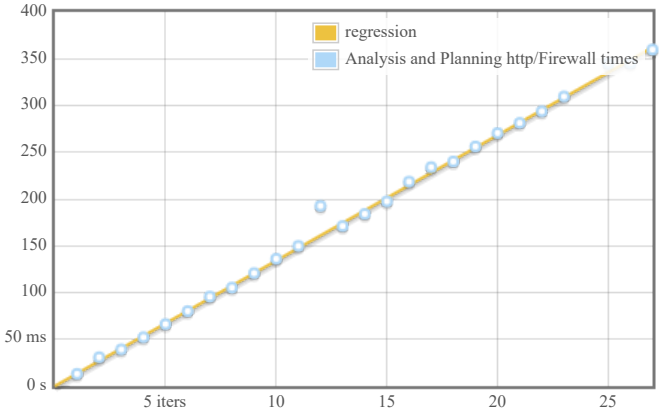


	lower bound	estimate	upper bound
OLS regression	16.5 ms	16.8 ms	17.2 ms
R² goodness-of-fit	0.997	0.998	1.000
Mean execution time	16.2 ms	16.5 ms	16.6 ms
Standard deviation	300 µs	462 µs	757 µs

Outlying measurements have slight (8.0%) effect on estimated standard deviation.

Analysis and Planning http/Firewall

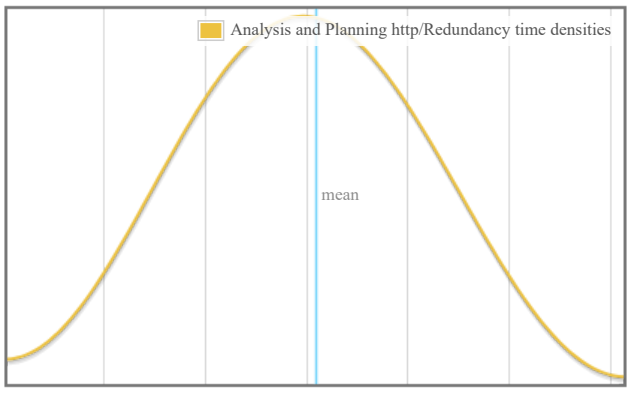


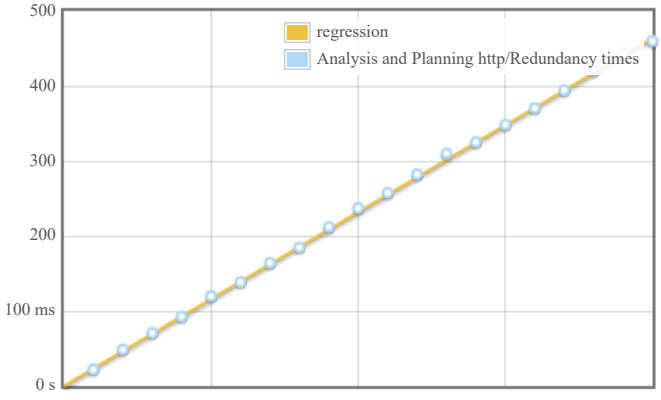


	lower bound	estimate	upper bound
OLS regression	13.2 ms	13.4 ms	13.6 ms
R² goodness-of-fit	0.986	0.996	1.000
Mean execution time	13.2 ms	13.3 ms	13.8 ms
Standard deviation	273 µs	623 µs	1.18 ms

Outlying measurements have moderate (18.2%) effect on estimated standard deviation.

Analysis and Planning http/Redundancy

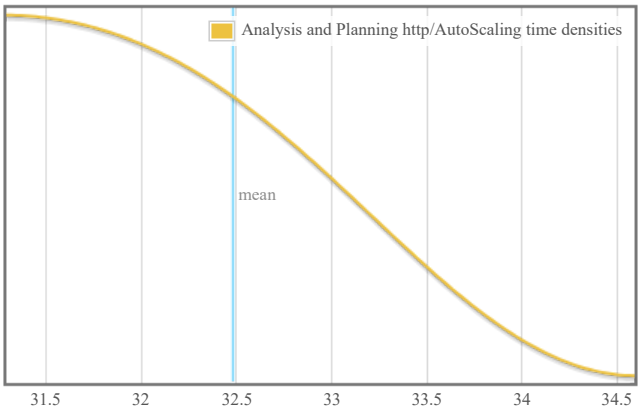


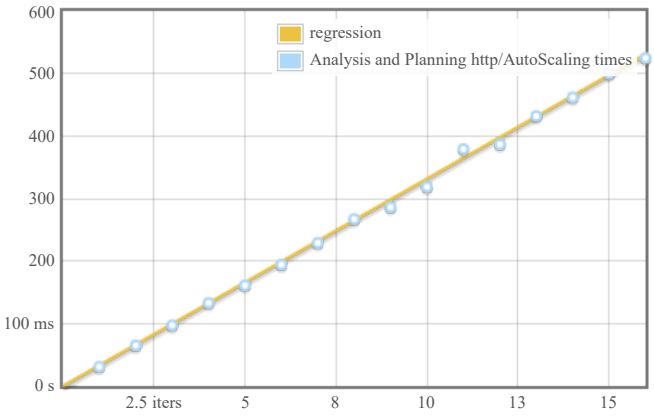


	lower bound	estimate	upper bound
OLS regression	23.0 ms	23.2 ms	23.4 ms
R² goodness-of-fit	0.999	1.000	1.000
Mean execution time	23.1 ms	23.2 ms	23.3 ms
Standard deviation	220 µs	286 µs	371 µs

Outlying measurements have slight (4.8%) effect on estimated standard deviation.

Analysis and Planning http/AutoScaling

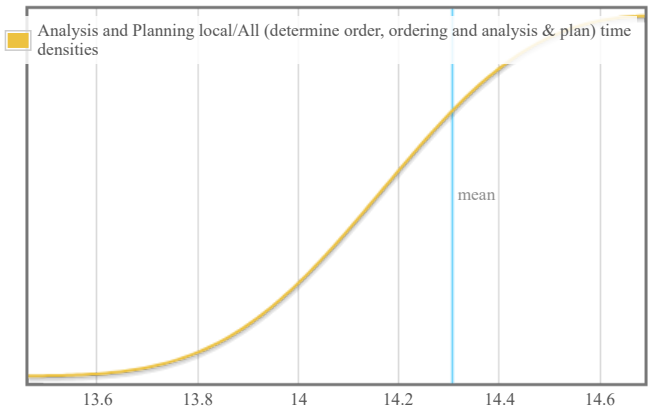


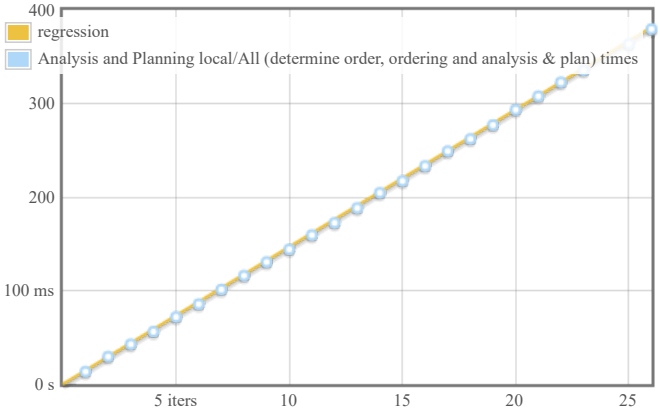


	lower bound	estimate	upper bound
OLS regression	32.6 ms	33.2 ms	33.7 ms
R² goodness-of-fit	0.995	0.998	1.000
Mean execution time	32.1 ms	32.5 ms	32.9 ms
Standard deviation	569 µs	776 µs	1.15 ms

Outlying measurements have slight (5.9%) effect on estimated standard deviation.

Analysis and Planning local/All (determine order, ordering and analysis & plan)

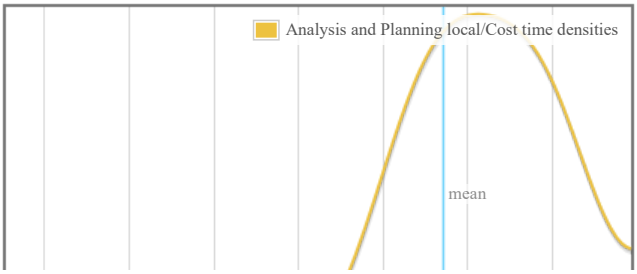


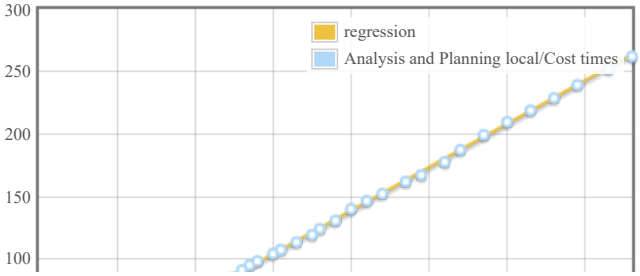


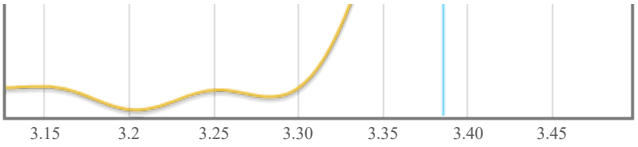
	lower bound	estimate	upper bound
OLS regression	14.6 ms	14.7 ms	14.7 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	14.2 ms	14.3 ms	14.4 ms
Standard deviation	184 µs	288 µs	403 µs

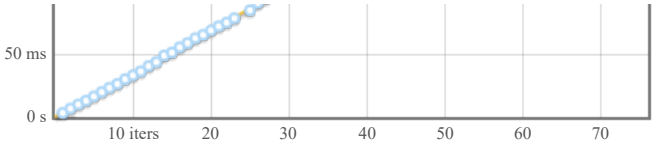
Outlying measurements have slight (3.8%) effect on estimated standard deviation.

Analysis and Planning local/Cost





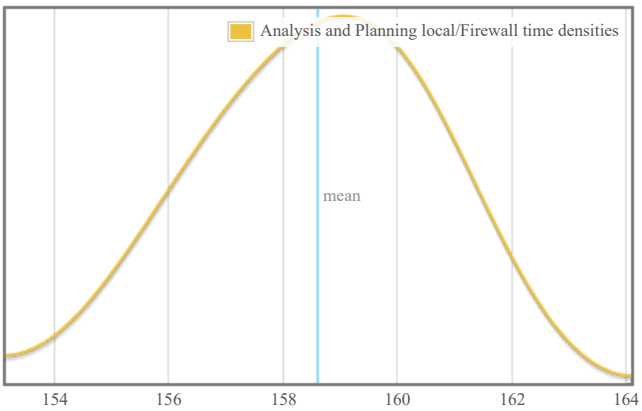


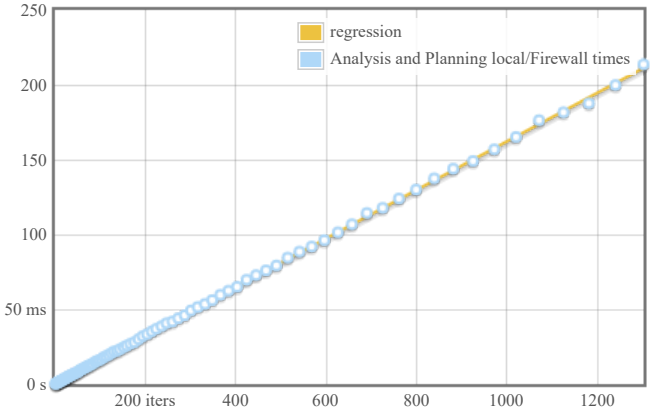


	lower bound	estimate	upper bound
OLS regression	3.46 ms	3.47 ms	3.49 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	3.36 ms	3.39 ms	3.40 ms
Standard deviation	51.7 µs	74.0 µs	101 µs

Outlying measurements have slight (8.0%) effect on estimated standard deviation.

Analysis and Planning local/Firewall

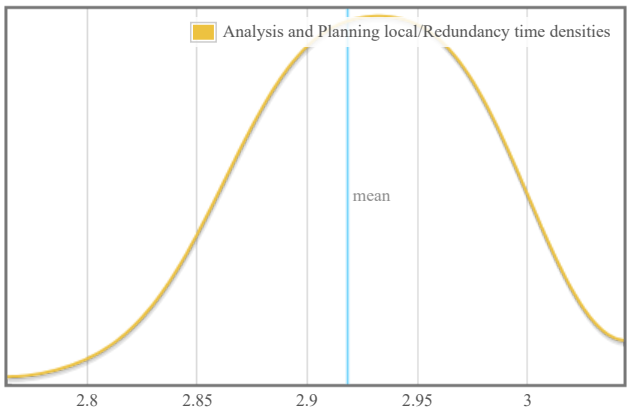


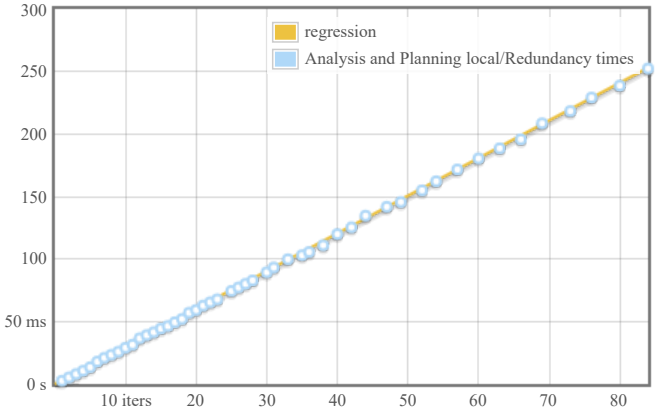


	lower bound	estimate	upper bound
OLS regression	161 µs	162 µs	163 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	158 µs	159 µs	159 µs
Standard deviation	1.98 µs	2.36 µs	2.82 µs

Outlying measurements have slight (8.1%) effect on estimated standard deviation.

Analysis and Planning local/Redundancy

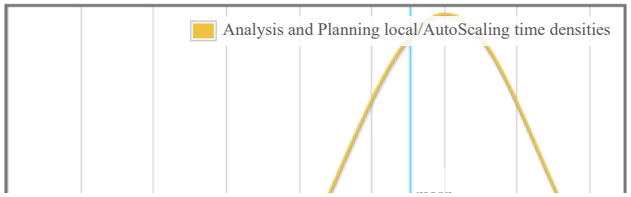


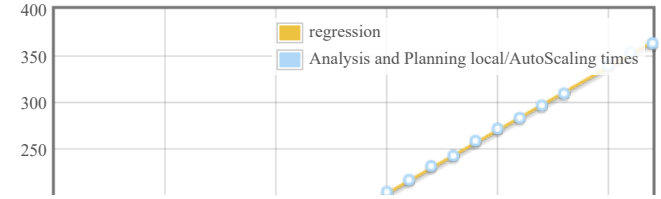


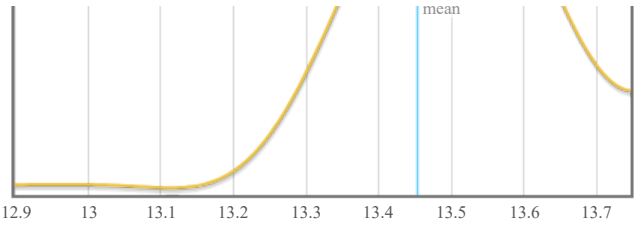
	lower bound	estimate	upper bound
OLS regression	3.00 ms	3.01 ms	3.02 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	2.90 ms	2.92 ms	2.94 ms
Standard deviation	48.1 µs	58.5 µs	73.5 µs

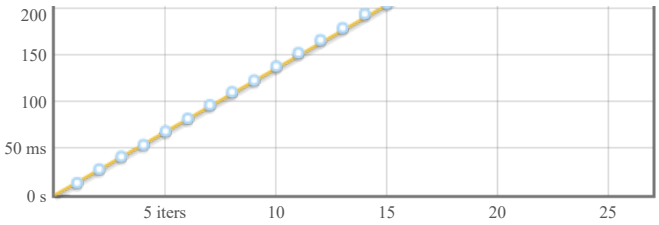
Outlying measurements have slight (7.6%) effect on estimated standard deviation.

Analysis and Planning local/AutoScaling





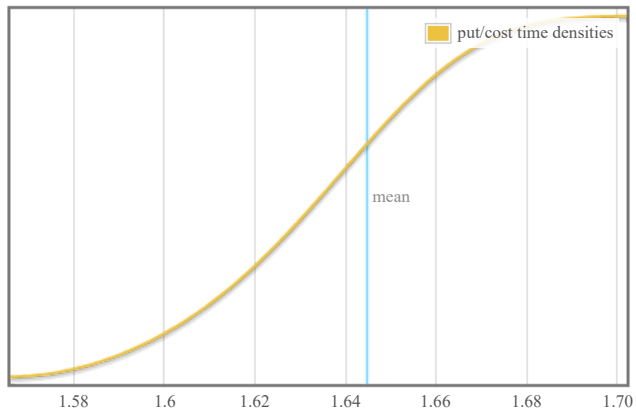


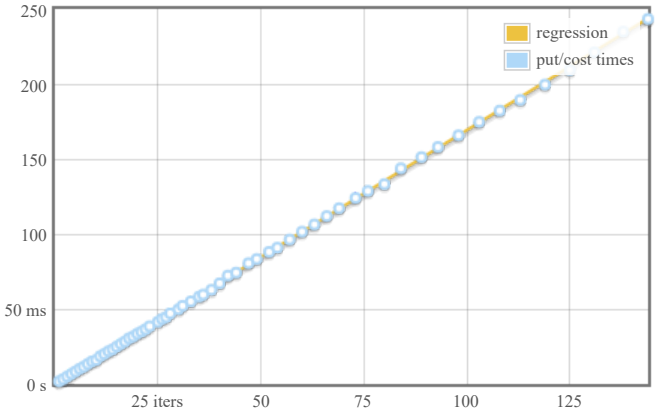


	lower bound	estimate	upper bound
OLS regression	13.4 ms	13.5 ms	13.6 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	13.4 ms	13.5 ms	13.5 ms
Standard deviation	106 µs	163 µs	243 µs

Outlying measurements have slight (3.7%) effect on estimated standard deviation.

put/cost

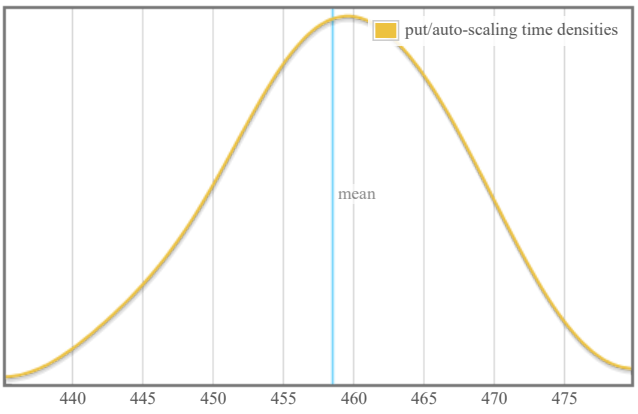


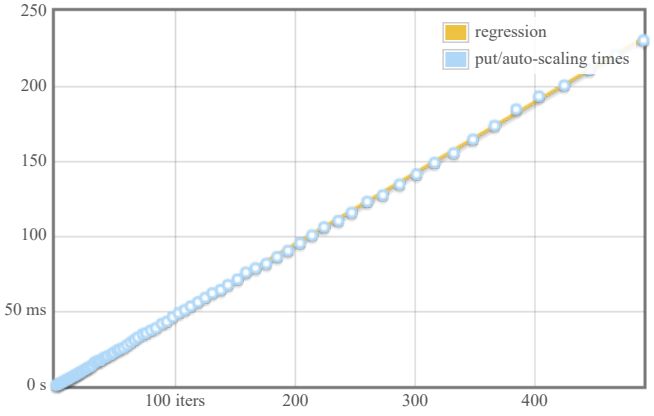


	lower bound	estimate	upper bound
OLS regression	1.69 ms	1.69 ms	1.70 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.63 ms	1.64 ms	1.66 ms
Standard deviation	30.7 µs	35.3 µs	40.9 µs

Outlying measurements have slight (9.4%) effect on estimated standard deviation.

put/auto-scaling

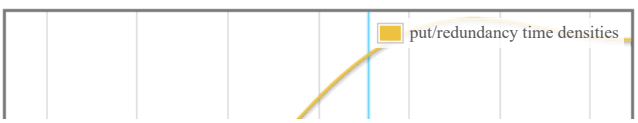


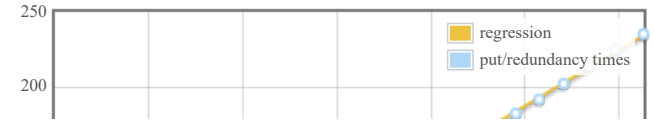


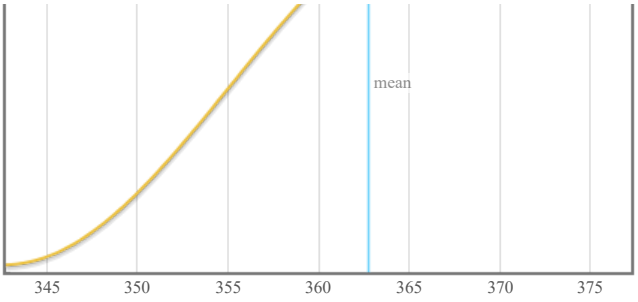
	lower bound	estimate	upper bound
OLS regression	472 µs	474 µs	476 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	456 µs	459 µs	461 µs
Standard deviation	7.37 µs	8.93 µs	10.8 µs

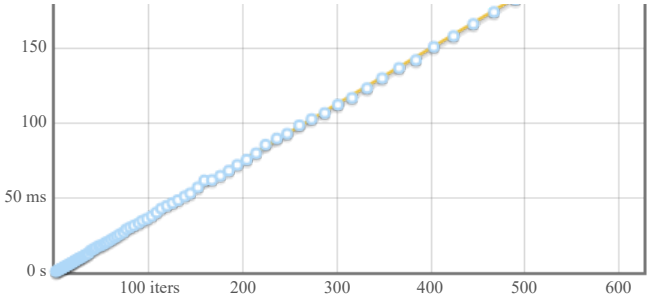
Outlying measurements have moderate (11.1%) effect on estimated standard deviation.

put/redundancy





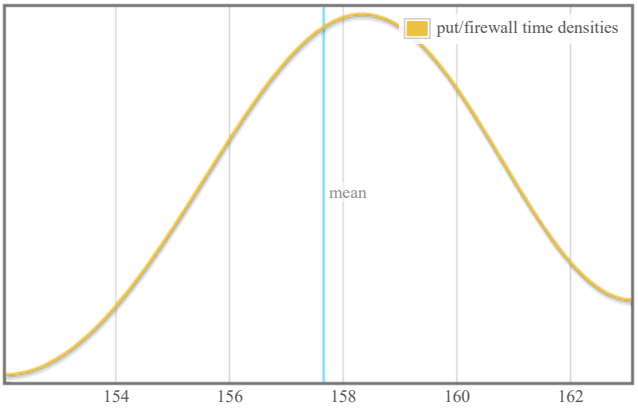


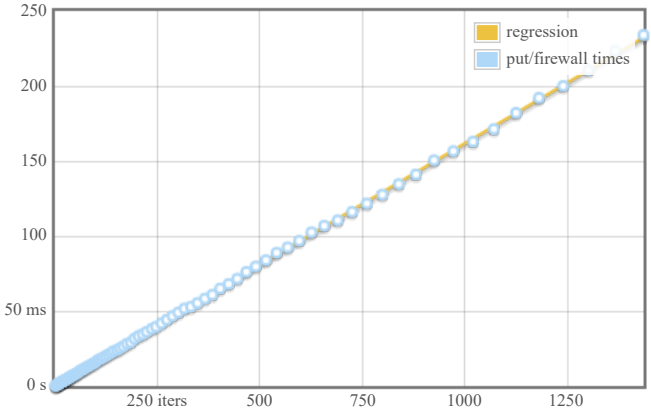


	lower bound	estimate	upper bound
OLS regression	373 µs	375 µs	376 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	360 µs	363 µs	365 µs
Standard deviation	6.85 µs	8.05 µs	9.70 µs

Outlying measurements have moderate (13.7%) effect on estimated standard deviation.

put/firewall





	lower bound	estimate	upper bound
OLS regression	161 µs	162 µs	162 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	157 µs	158 µs	158 µs
Standard deviation	2.29 µs	2.66 µs	3.10 µs

Outlying measurements have moderate (10.6%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own. The charts in each section are active; if you hover your mouse over data points and annotations, you will see more details.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x* axis indicates the number of loop iterations, while the *y* axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression prediction of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R² goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R² should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be. (Hover the mouse over the table headers to see the confidence levels.)

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.

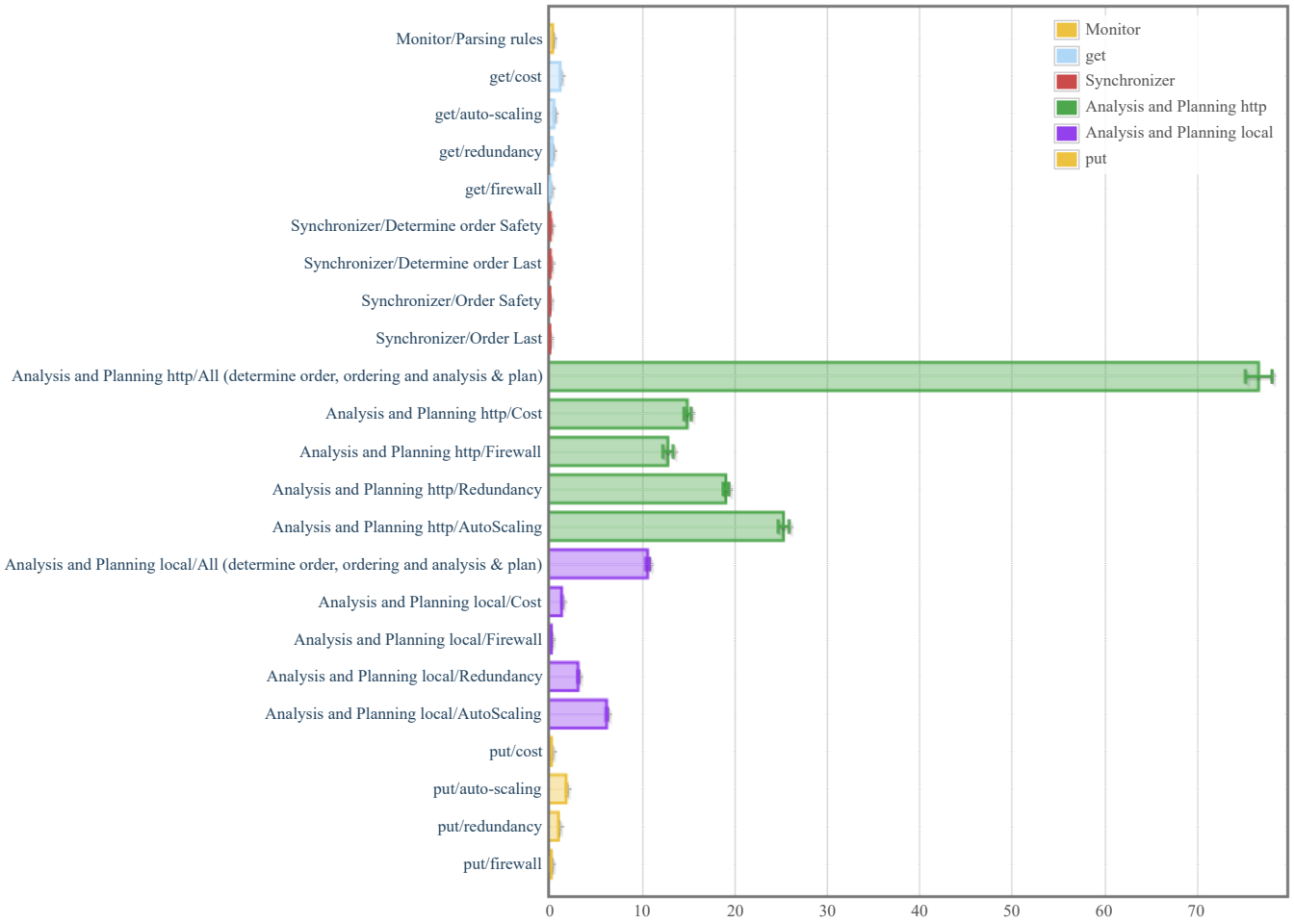
Appendix C

Experiment 3

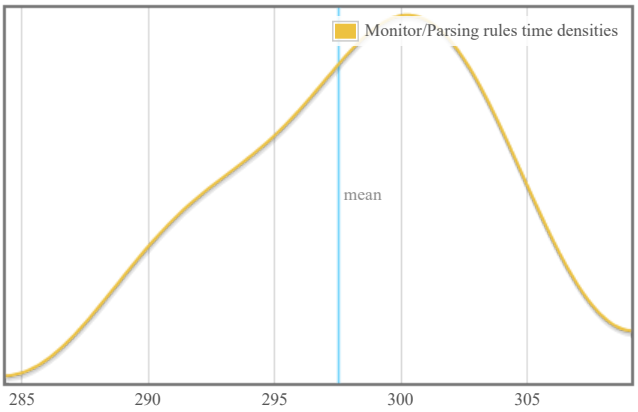
criterion performance measurements

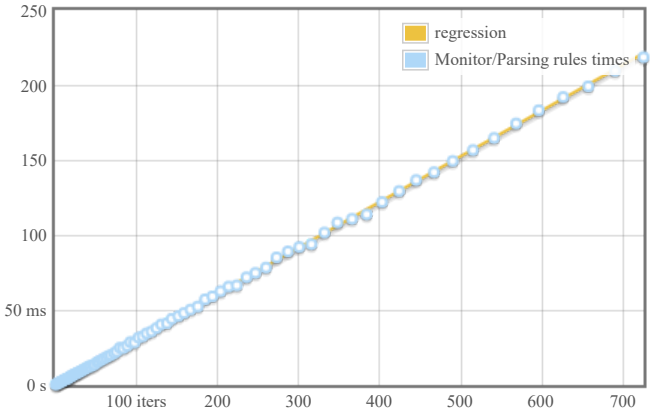
overview

want to understand this report?



Monitor/Parsing rules

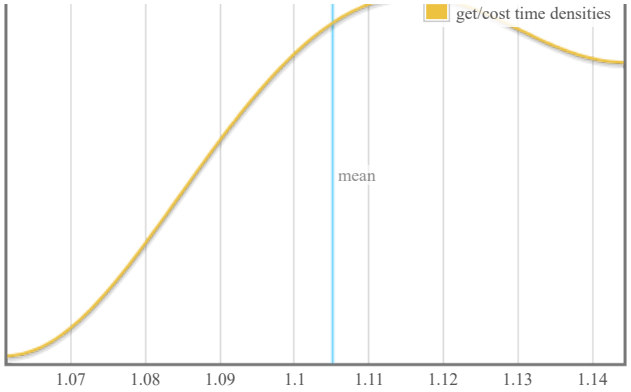


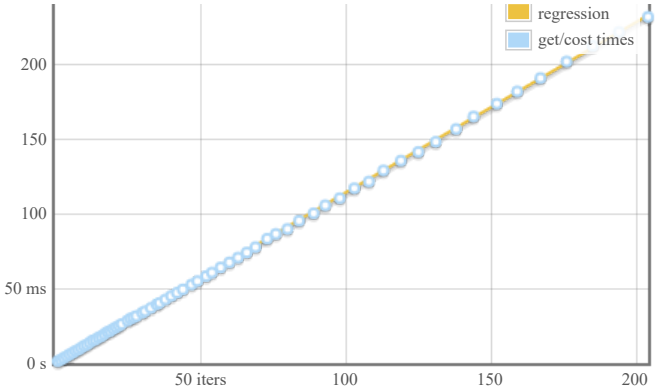


	lower bound	estimate	upper bound
OLS regression	304 µs	305 µs	306 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	296 µs	298 µs	299 µs
Standard deviation	4.77 µs	5.62 µs	6.64 µs

Outlying measurements have moderate (11.2%) effect on estimated standard deviation.

get/cost

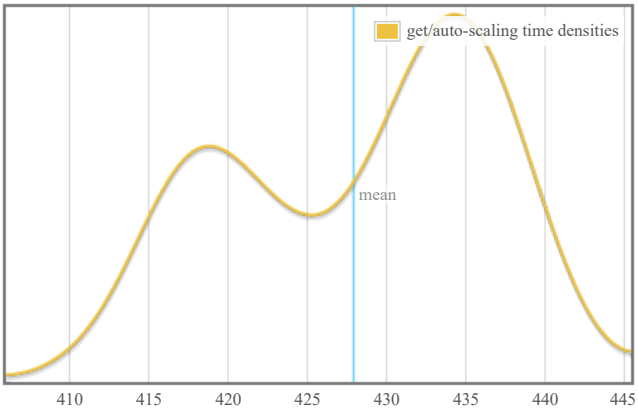


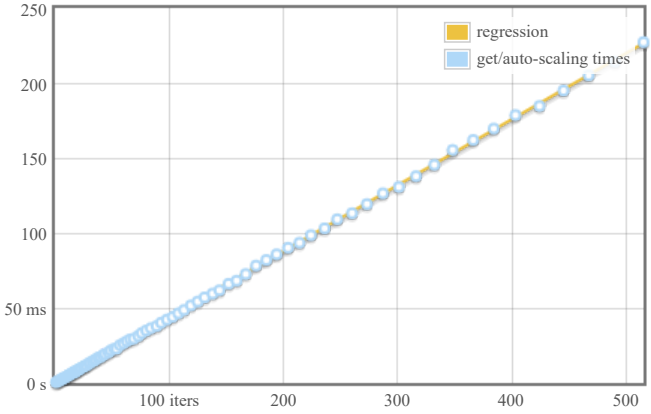


	lower bound	estimate	upper bound
OLS regression	1.13 ms	1.14 ms	1.14 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.10 ms	1.11 ms	1.11 ms
Standard deviation	18.7 µs	21.3 µs	24.6 µs

Outlying measurements have slight (8.4%) effect on estimated standard deviation.

get/auto-scaling

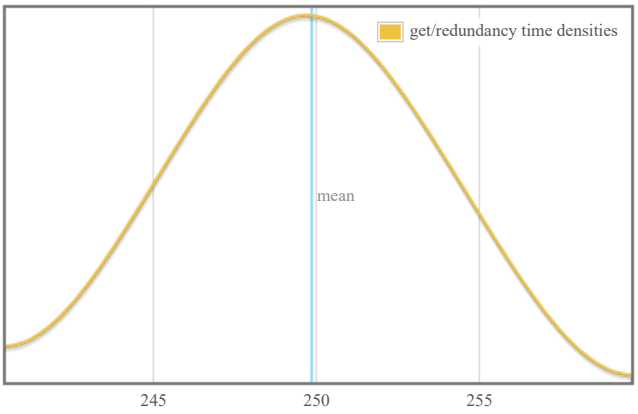


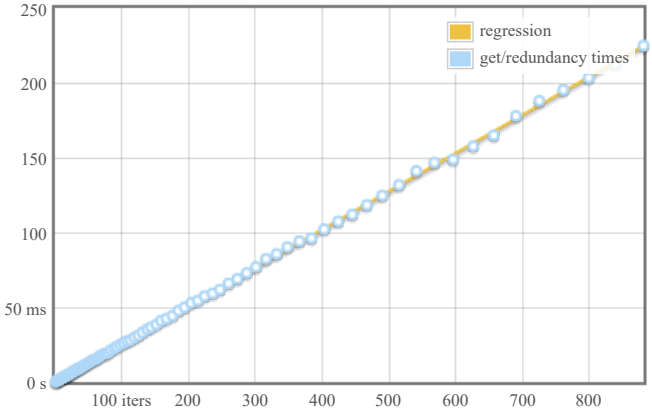


	lower bound	estimate	upper bound
OLS regression	438 µs	440 µs	441 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	425 µs	428 µs	431 µs
Standard deviation	7.58 µs	8.63 µs	10.1 µs

Outlying measurements have moderate (12.1%) effect on estimated standard deviation.

get/redundancy

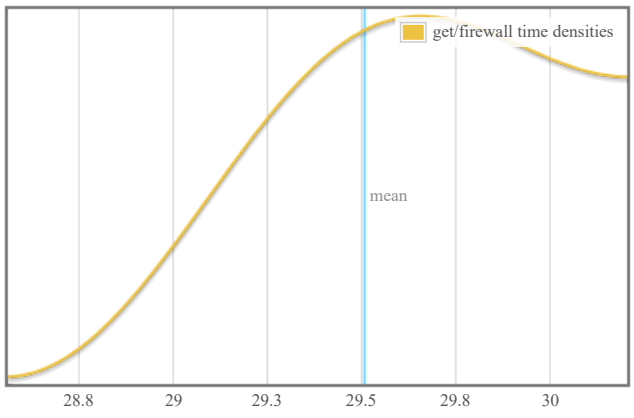


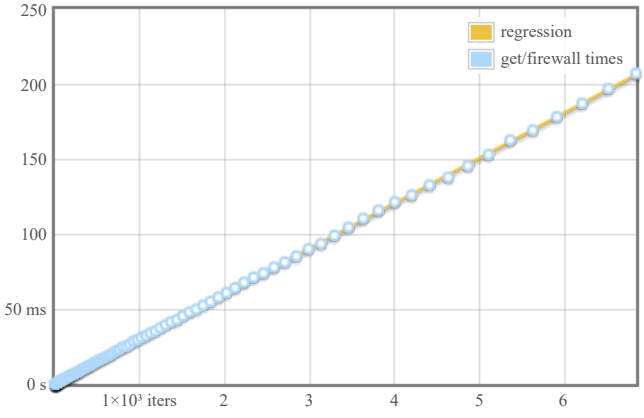


	lower bound	estimate	upper bound
OLS regression	254 µs	255 µs	257 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	249 µs	250 µs	251 µs
Standard deviation	3.45 µs	4.14 µs	4.88 µs

Outlying measurements have slight (8.8%) effect on estimated standard deviation.

get/firewall

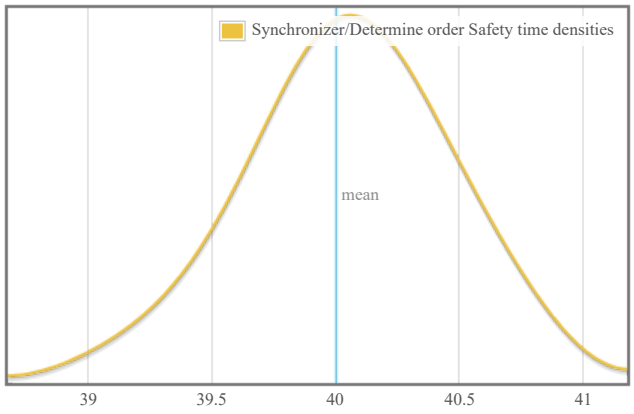


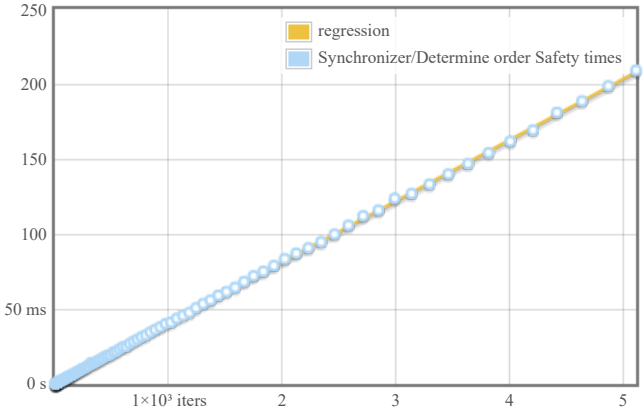


	lower bound	estimate	upper bound
OLS regression	30.1 µs	30.1 µs	30.2 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	29.4 µs	29.5 µs	29.6 µs
Standard deviation	314 ns	377 ns	458 ns

Outlying measurements have slight (7.5%) effect on estimated standard deviation.

Synchronizer/Determine order Safety

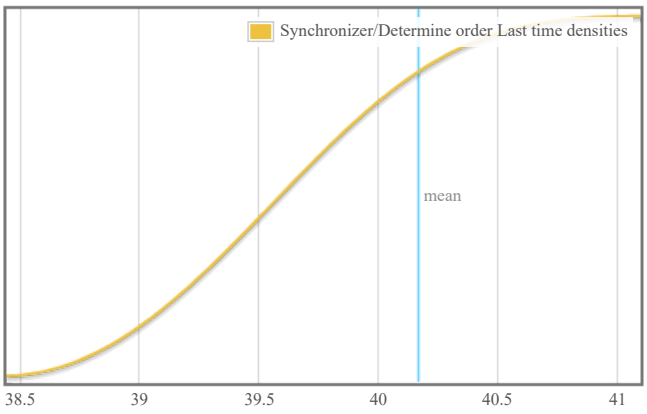


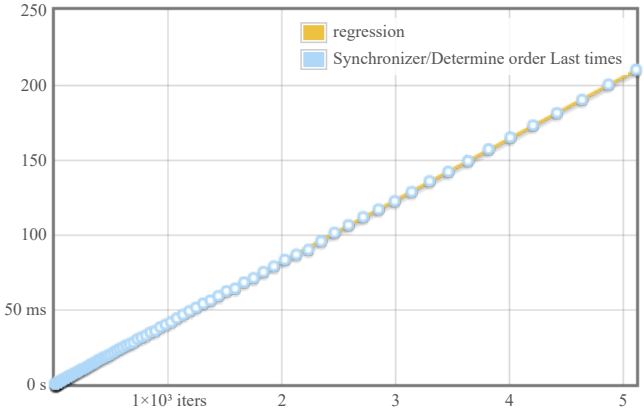


	lower bound	estimate	upper bound
OLS regression	40.6 µs	40.7 µs	40.9 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	39.8 µs	40.0 µs	40.1 µs
Standard deviation	389 ns	497 ns	604 ns

Outlying measurements have slight (7.1%) effect on estimated standard deviation.

Synchronizer/Determine order Last

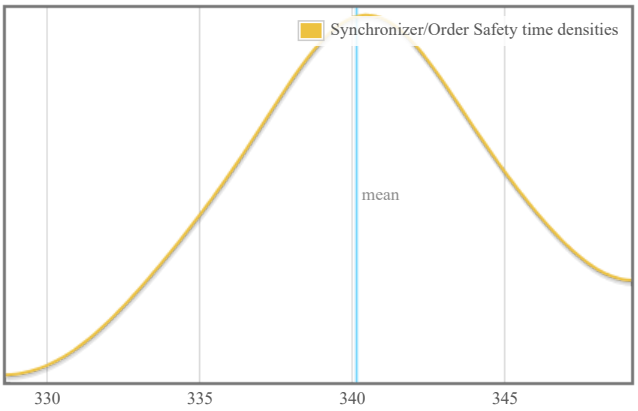


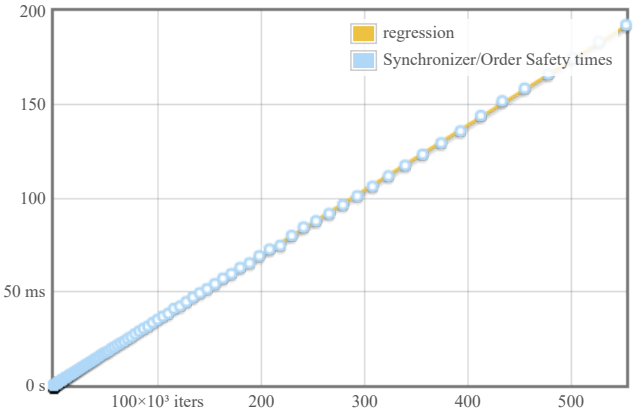


	lower bound	estimate	upper bound
OLS regression	41.0 µs	41.1 µs	41.1 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	40.0 µs	40.2 µs	40.4 µs
Standard deviation	478 ns	580 ns	753 ns

Outlying measurements have slight (9.2%) effect on estimated standard deviation.

Synchronizer/Order Safety

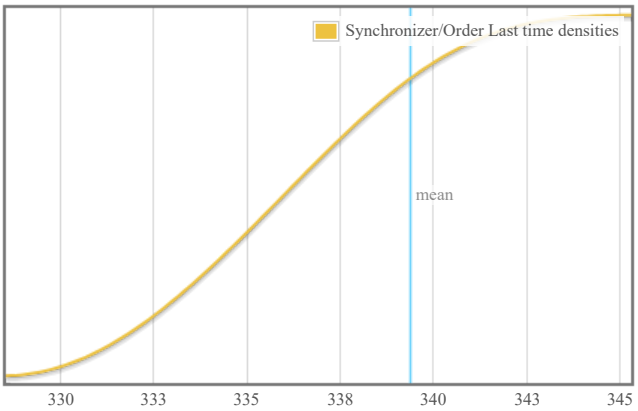


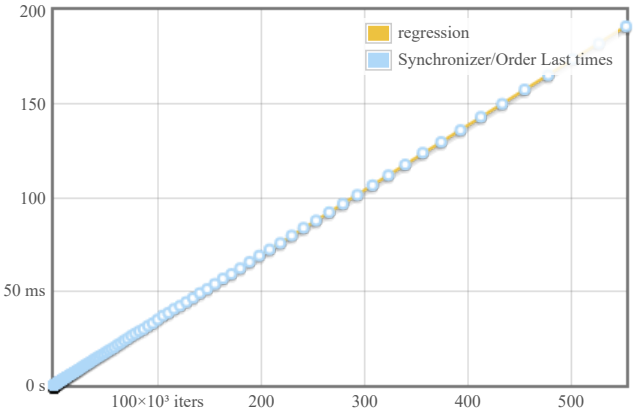


	lower bound	estimate	upper bound
OLS regression	347 ns	348 ns	348 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	339 ns	340 ns	341 ns
Standard deviation	3.39 ns	4.06 ns	5.04 ns

Outlying measurements have moderate (10.7%) effect on estimated standard deviation.

Synchronizer/Order Last

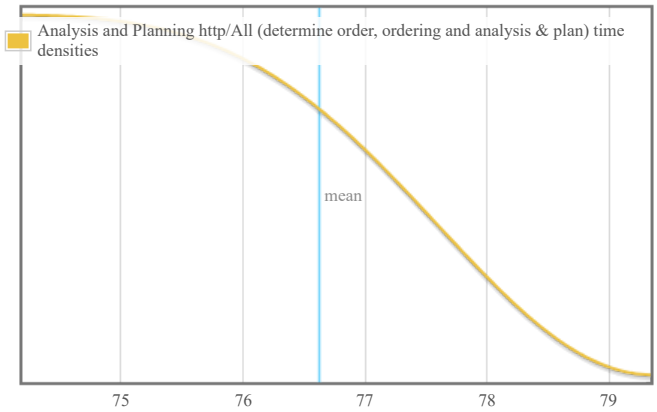


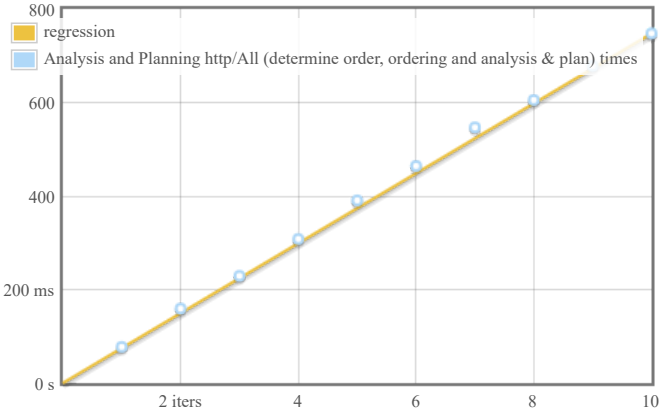


	lower bound	estimate	upper bound
OLS regression	346 ns	347 ns	347 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	338 ns	339 ns	341 ns
Standard deviation	3.02 ns	3.67 ns	4.61 ns

Outlying measurements have slight (9.1%) effect on estimated standard deviation.

Analysis and Planning http/All (determine order, ordering and analysis & plan)



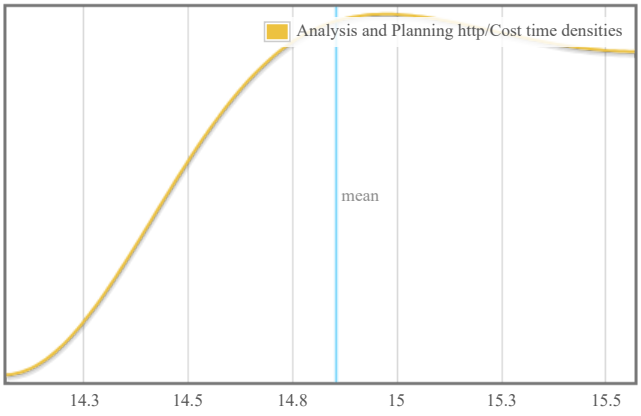


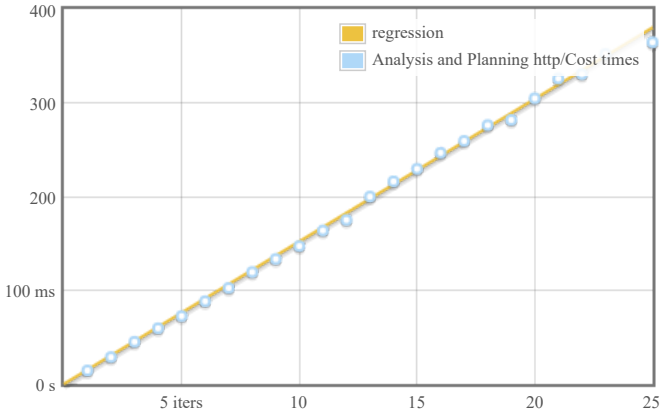
	lower bound	estimate	upper bound
OLS regression	72.8 ms	74.7 ms	77.4 ms
R² goodness-of-fit	0.998	0.999	1.000

Mean execution time	75.8 ms	76.6 ms	77.6 ms
Standard deviation	1.08 ms	1.44 ms	2.04 ms

Outlying measurements have slight (9.0%) effect on estimated standard deviation.

Analysis and Planning http/Cost

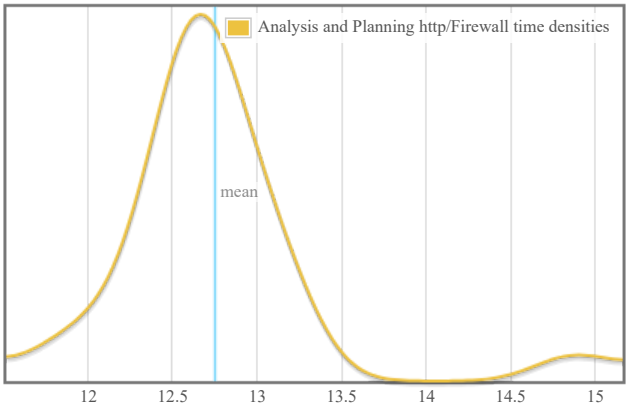


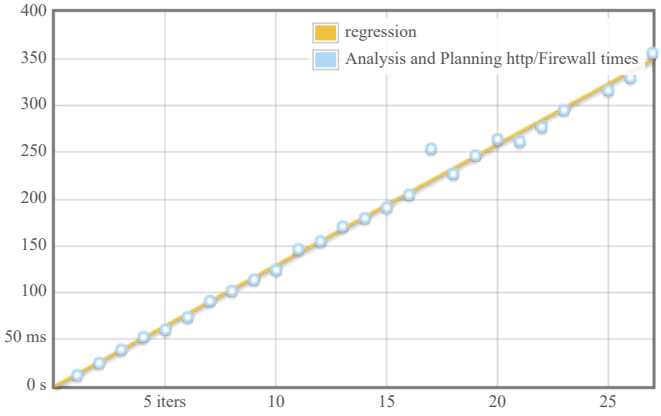


	lower bound	estimate	upper bound
OLS regression	14.9 ms	15.2 ms	15.6 ms
R² goodness-of-fit	0.997	0.998	0.999
Mean execution time	14.7 ms	14.9 ms	15.0 ms
Standard deviation	331 µs	381 µs	462 µs

Outlying measurements have slight (4.8%) effect on estimated standard deviation.

Analysis and Planning http/Firewall

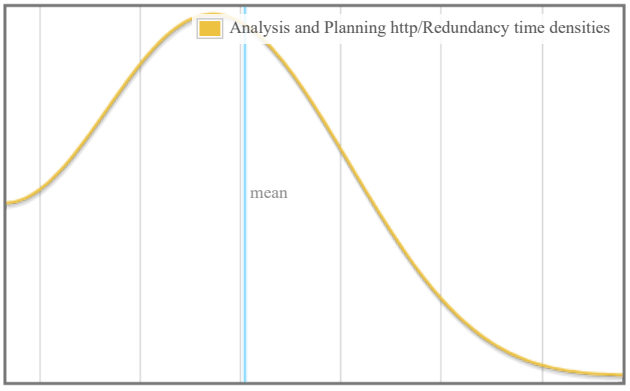


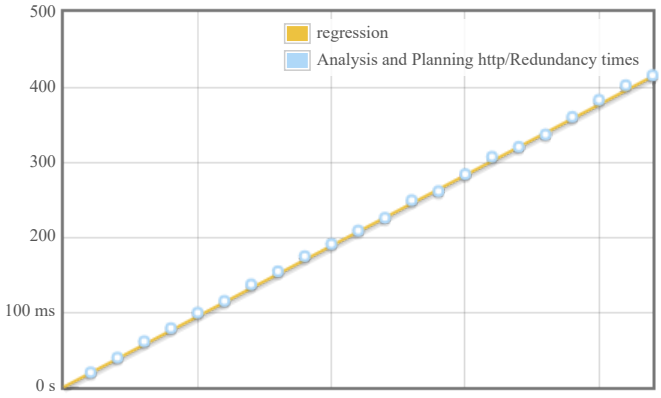


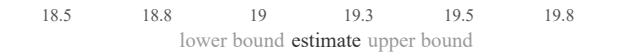
	lower bound	estimate	upper bound
OLS regression	12.7 ms	12.9 ms	13.3 ms
R² goodness-of-fit	0.982	0.994	0.999
Mean execution time	12.6 ms	12.8 ms	13.1 ms
Standard deviation	288 µs	551 µs	991 µs

Outlying measurements have moderate (18.0%) effect on estimated standard deviation.

Analysis and Planning http/Redundancy



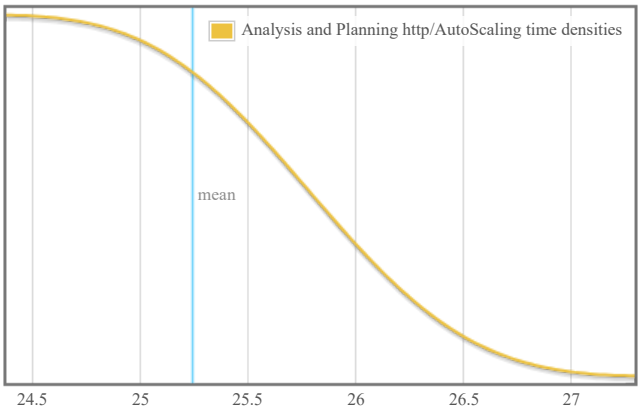


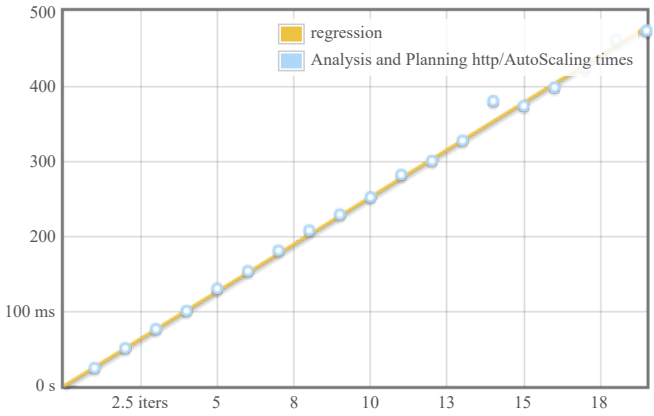


	lower bound	estimate	upper bound
OLS regression	18.6 ms	18.8 ms	19.0 ms
R² goodness-of-fit	0.999	1.000	1.000
Mean execution time	18.9 ms	19.0 ms	19.2 ms
Standard deviation	219 µs	305 µs	471 µs

Outlying measurements have slight (4.3%) effect on estimated standard deviation.

Analysis and Planning http/AutoScaling

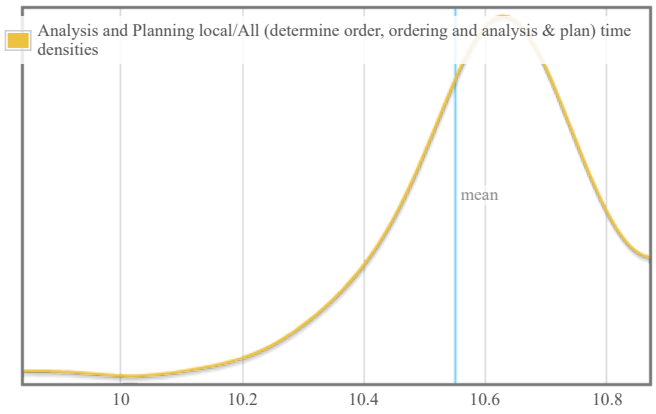


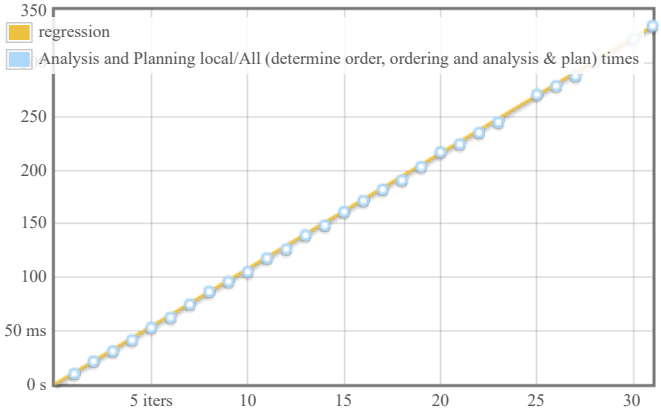


	lower bound	estimate	upper bound
OLS regression	24.7 ms	25.2 ms	25.9 ms
R² goodness-of-fit	0.993	0.997	1.000
Mean execution time	25.0 ms	25.2 ms	25.6 ms
Standard deviation	348 µs	578 µs	1.02 ms

Outlying measurements have slight (5.0%) effect on estimated standard deviation.

Analysis and Planning local/All (determine order, ordering and analysis & plan)

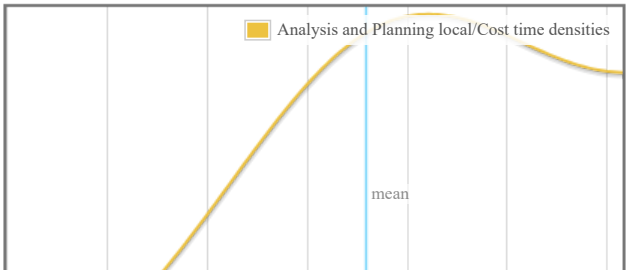


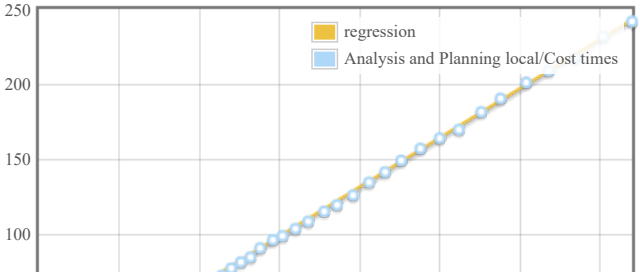


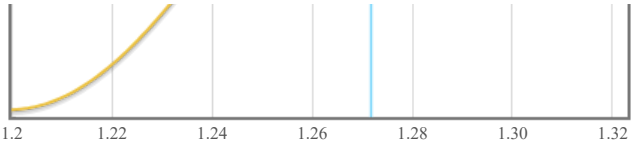
	lower bound	estimate	upper bound
OLS regression	10.8 ms	10.8 ms	10.9 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	10.5 ms	10.6 ms	10.6 ms
Standard deviation	134 µs	201 µs	316 µs

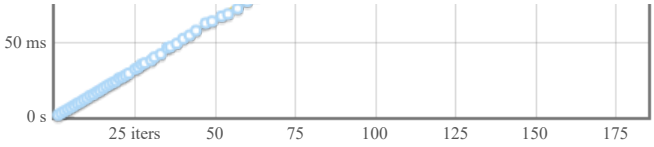
Outlying measurements have slight (3.3%) effect on estimated standard deviation.

Analysis and Planning local/Cost





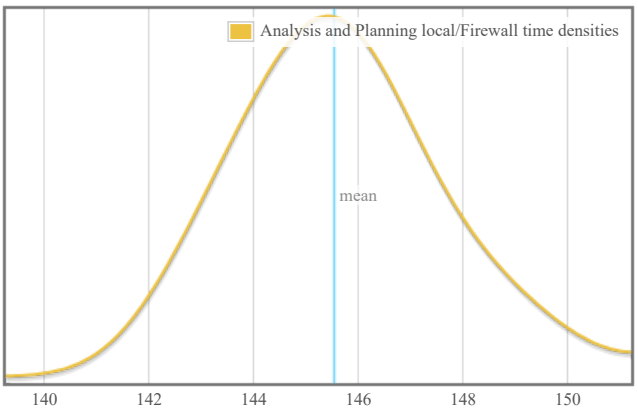


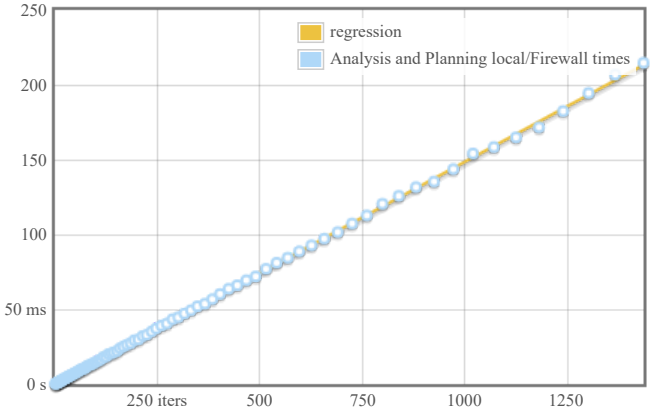


	lower bound	estimate	upper bound
OLS regression	1.31 ms	1.31 ms	1.32 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.26 ms	1.27 ms	1.28 ms
Standard deviation	23.6 µs	28.3 µs	33.8 µs

Outlying measurements have moderate (11.6%) effect on estimated standard deviation.

Analysis and Planning local/Firewall

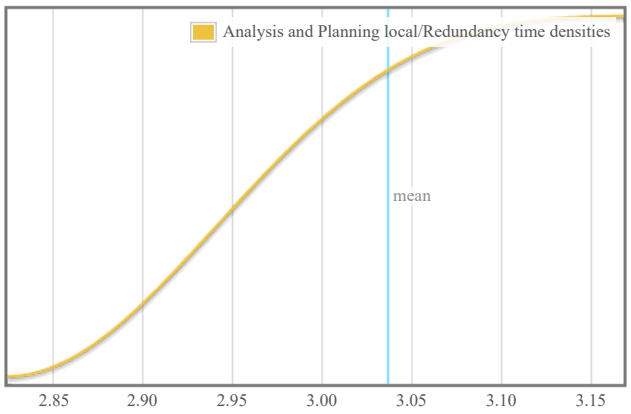


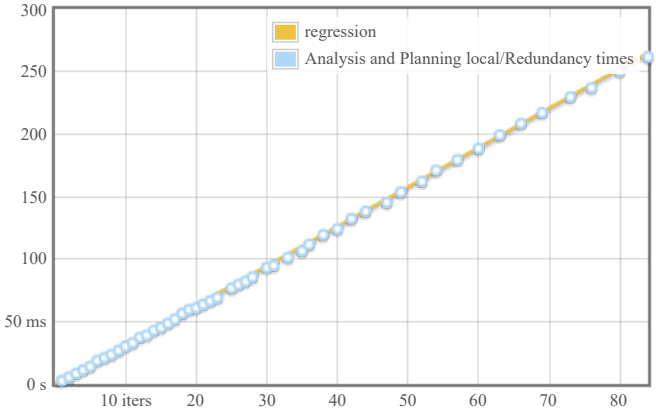


	lower bound	estimate	upper bound
OLS regression	148 µs	149 µs	150 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	145 µs	146 µs	146 µs
Standard deviation	1.66 µs	2.04 µs	2.63 µs

Outlying measurements have slight (7.1%) effect on estimated standard deviation.

Analysis and Planning local/Redundancy

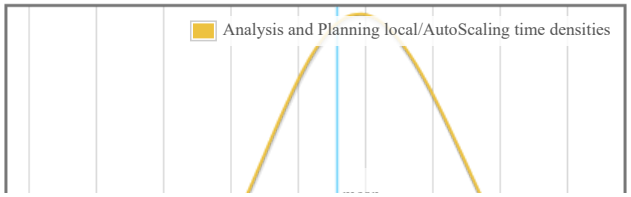


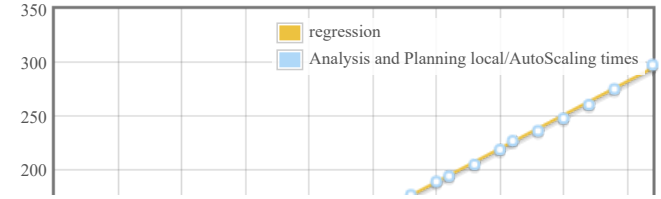


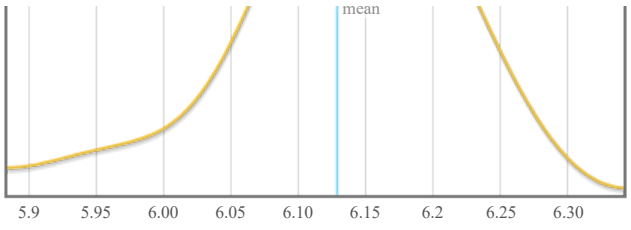
	lower bound	estimate	upper bound
OLS regression	3.14 ms	3.16 ms	3.17 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	3.01 ms	3.04 ms	3.06 ms
Standard deviation	66.4 µs	77.0 µs	94.7 µs

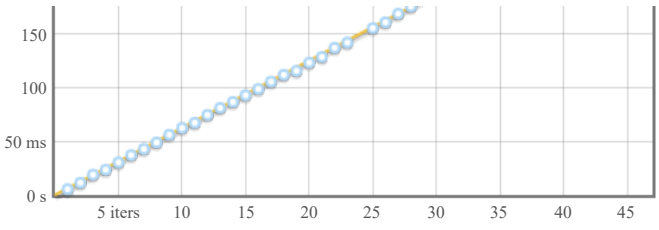
Outlying measurements have moderate (11.5%) effect on estimated standard deviation.

Analysis and Planning local/AutoScaling





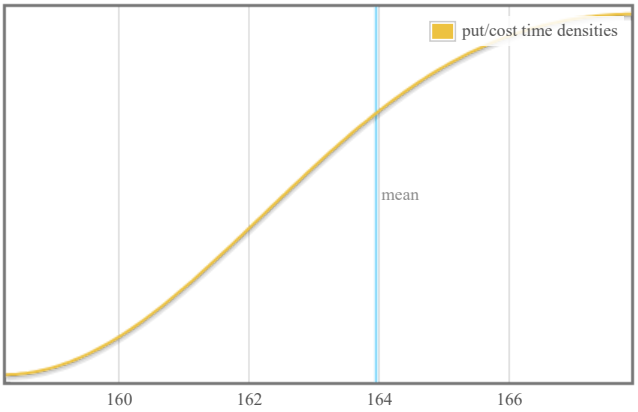


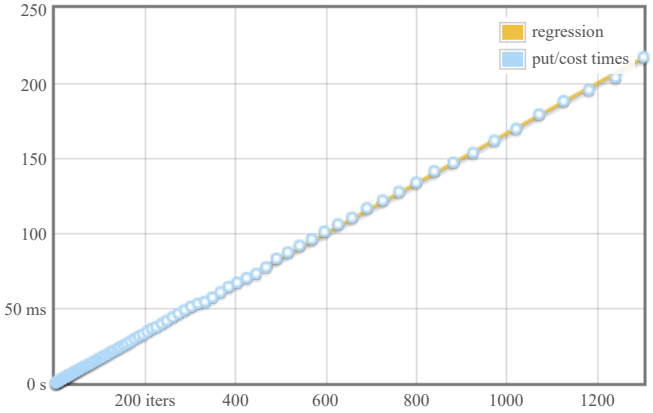


	lower bound	estimate	upper bound
OLS regression	6.23 ms	6.27 ms	6.32 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	6.10 ms	6.13 ms	6.16 ms
Standard deviation	69.1 µs	89.5 µs	115 µs

Outlying measurements have slight (2.6%) effect on estimated standard deviation.

put/cost

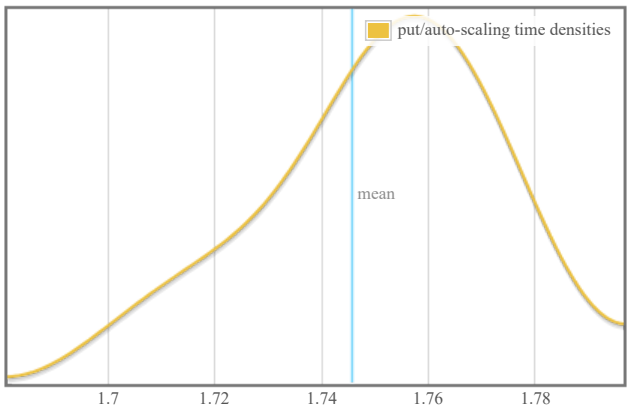


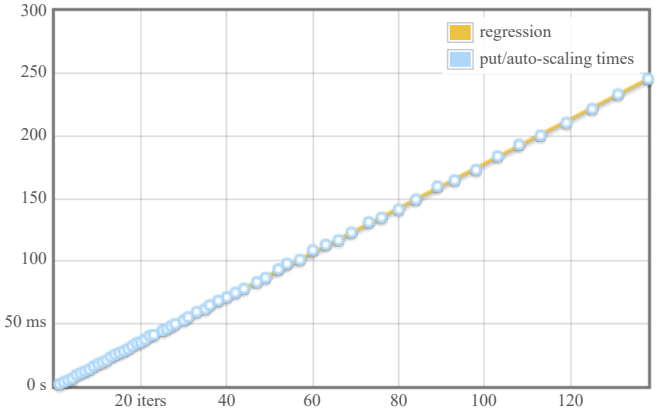


	lower bound	estimate	upper bound
OLS regression	166 µs	167 µs	167 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	163 µs	164 µs	165 µs
Standard deviation	2.03 µs	2.35 µs	2.77 µs

Outlying measurements have slight (7.2%) effect on estimated standard deviation.

put/auto-scaling

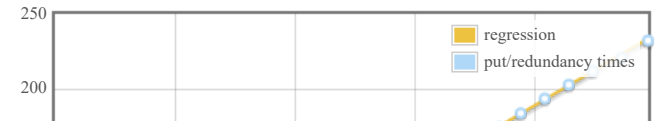


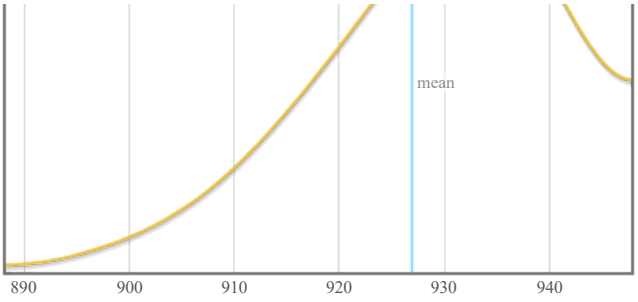


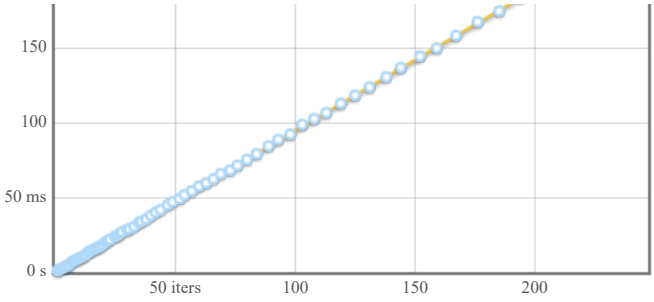
	lower bound	estimate	upper bound
OLS regression	1.78 ms	1.78 ms	1.79 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.74 ms	1.75 ms	1.75 ms
Standard deviation	21.3 µs	25.7 µs	30.5 µs

Outlying measurements have slight (1.7%) effect on estimated standard deviation.

put/redundancy



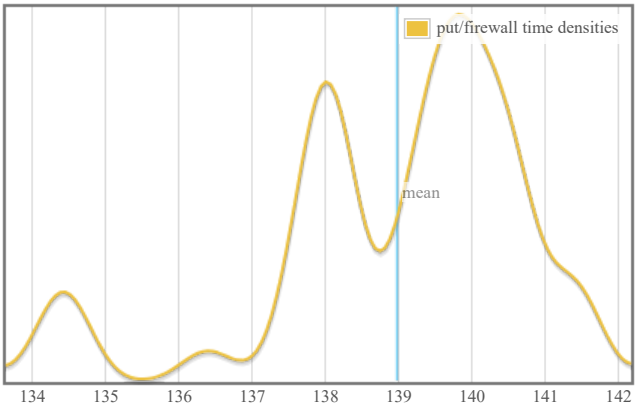


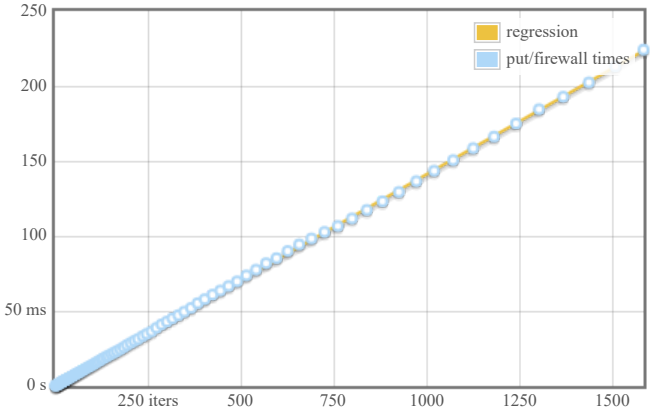


	lower bound	estimate	upper bound
OLS regression	943 µs	945 µs	949 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	923 µs	927 µs	930 µs
Standard deviation	9.85 µs	12.0 µs	16.5 µs

Outlying measurements have slight (1.4%) effect on estimated standard deviation.

put/firewall





	lower bound	estimate	upper bound
OLS regression	141 µs	141 µs	142 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	138 µs	139 µs	139 µs
Standard deviation	1.32 µs	1.75 µs	2.30 µs

Outlying measurements have slight (6.1%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own. The charts in each section are active; if you hover your mouse over data points and annotations, you will see more details.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x* axis indicates the number of loop iterations, while the *y* axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression prediction of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R² goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R² should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be. (Hover the mouse over the table headers to see the confidence levels.)

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.

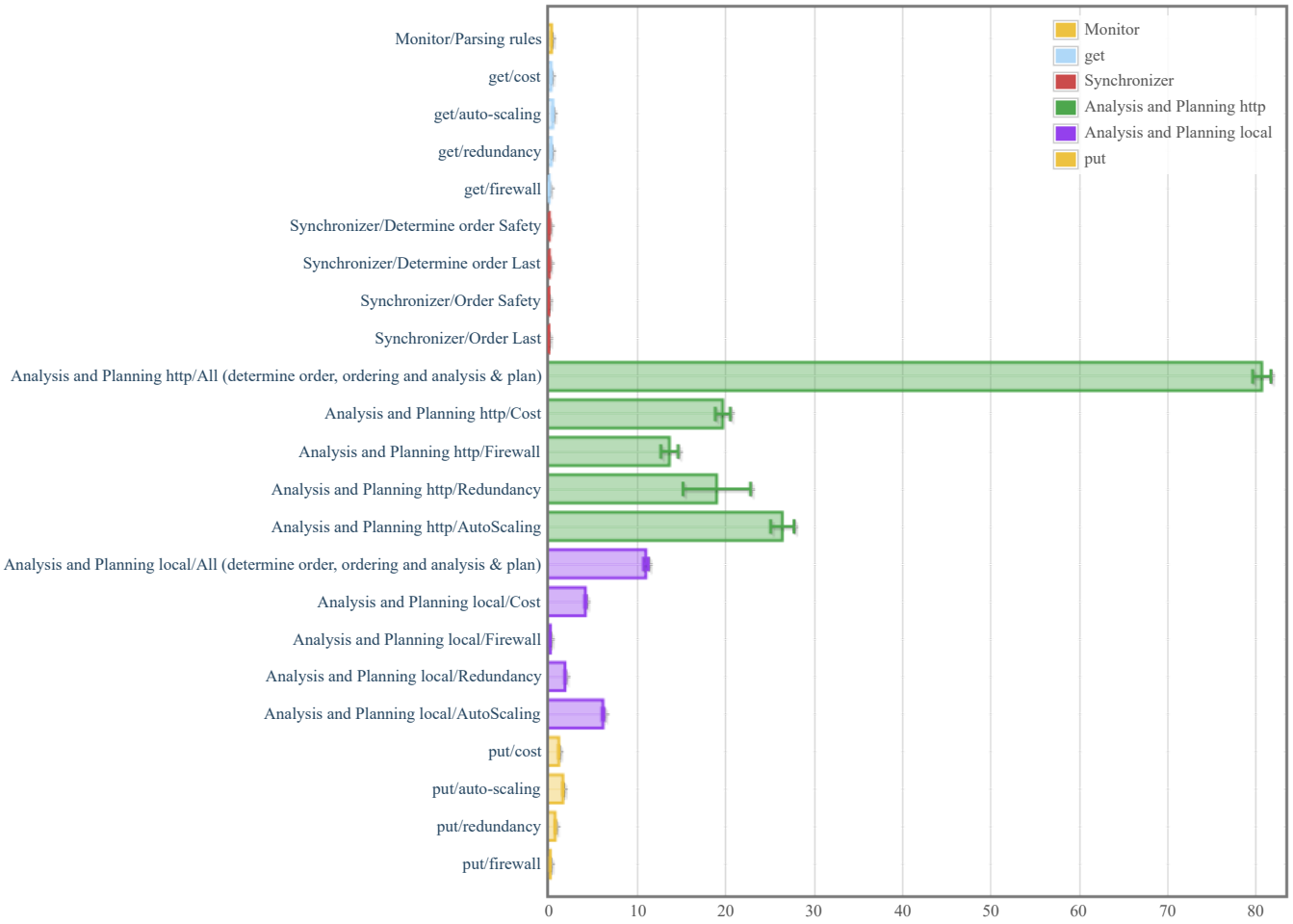
Appendix D

Experiment 4

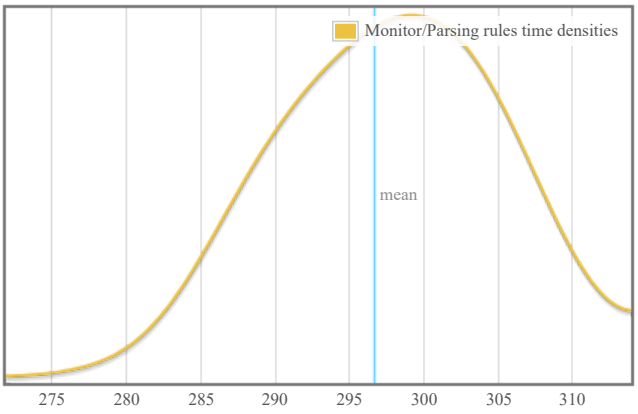
criterion performance measurements

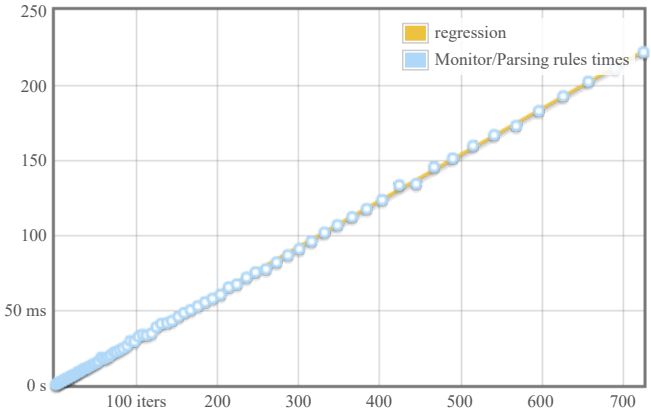
overview

want to understand this report?



Monitor/Parsing rules

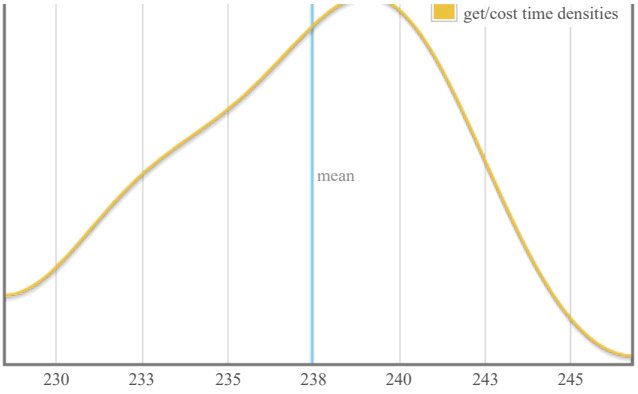


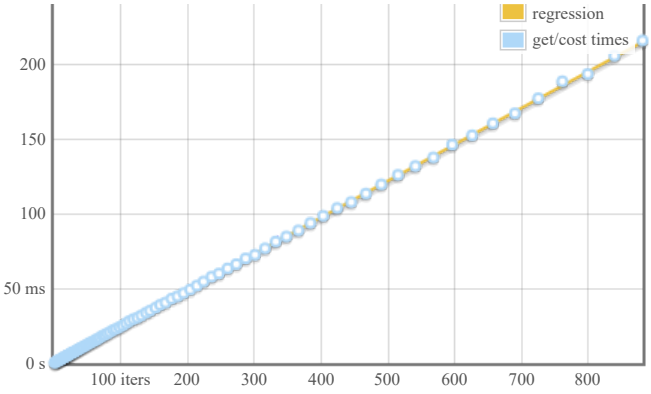


	lower bound	estimate	upper bound
OLS regression	306 µs	307 µs	309 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	295 µs	297 µs	299 µs
Standard deviation	6.47 µs	7.76 µs	9.98 µs

Outlying measurements have moderate (19.4%) effect on estimated standard deviation.

get/cost

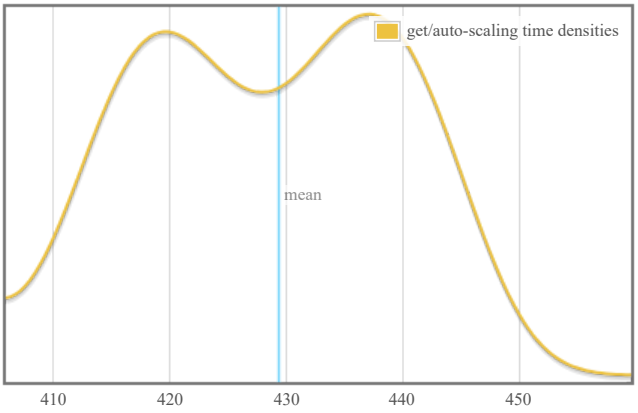


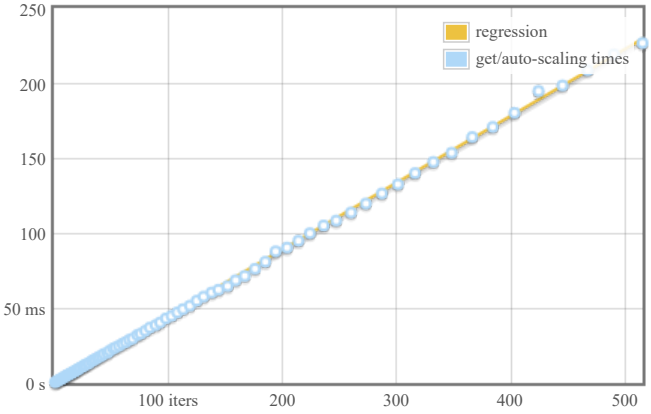


	lower bound	estimate	upper bound
OLS regression	243 µs	244 µs	244 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	236 µs	237 µs	239 µs
Standard deviation	3.21 µs	3.78 µs	4.49 µs

Outlying measurements have slight (8.7%) effect on estimated standard deviation.

get/auto-scaling

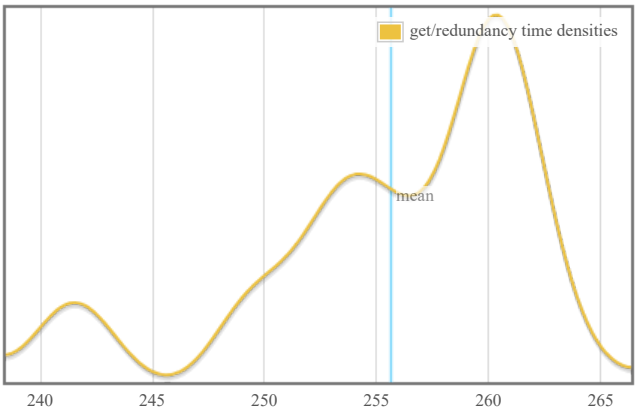


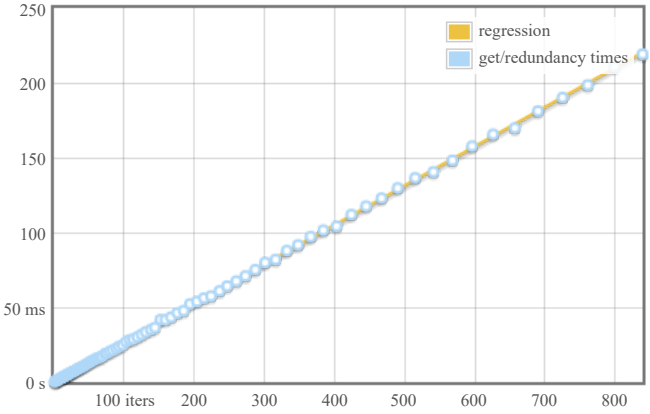


	lower bound	estimate	upper bound
OLS regression	443 µs	445 µs	448 µs
R² goodness-of-fit	0.999	1.000	1.000
Mean execution time	426 µs	429 µs	433 µs
Standard deviation	9.72 µs	11.0 µs	13.3 µs

Outlying measurements have moderate (17.6%) effect on estimated standard deviation.

get/redundancy

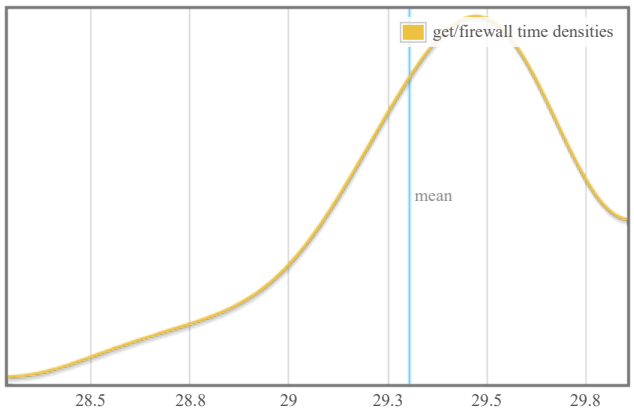


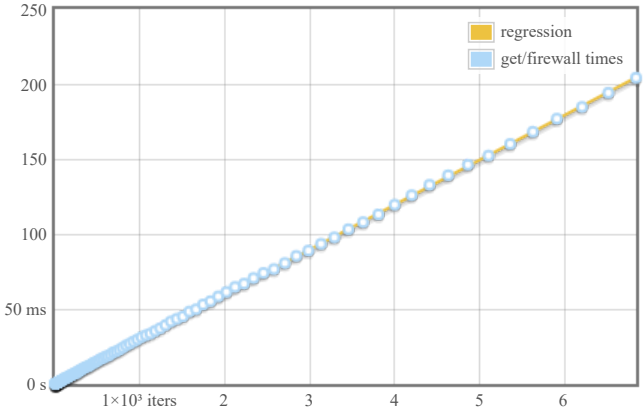


	lower bound	estimate	upper bound
OLS regression	262 µs	263 µs	264 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	254 µs	256 µs	257 µs
Standard deviation	4.95 µs	6.14 µs	7.66 µs

Outlying measurements have moderate (16.8%) effect on estimated standard deviation.

get/firewall

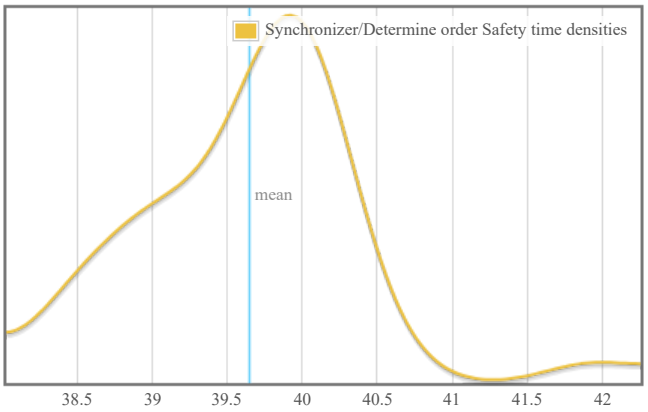


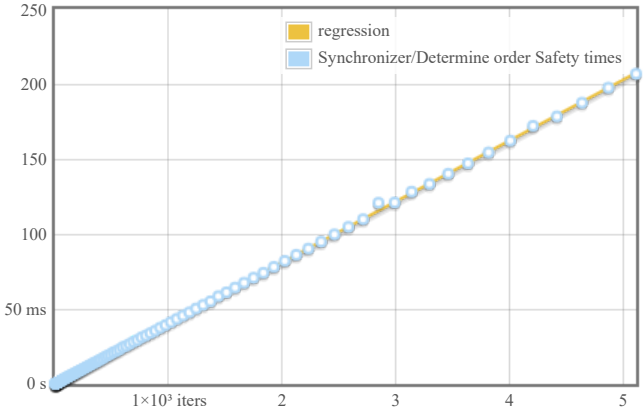


	lower bound	estimate	upper bound
OLS regression	29.8 µs	29.9 µs	29.9 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	29.2 µs	29.3 µs	29.4 µs
Standard deviation	261 ns	331 ns	425 ns

Outlying measurements have slight (6.1%) effect on estimated standard deviation.

Synchronizer/Determine order Safety

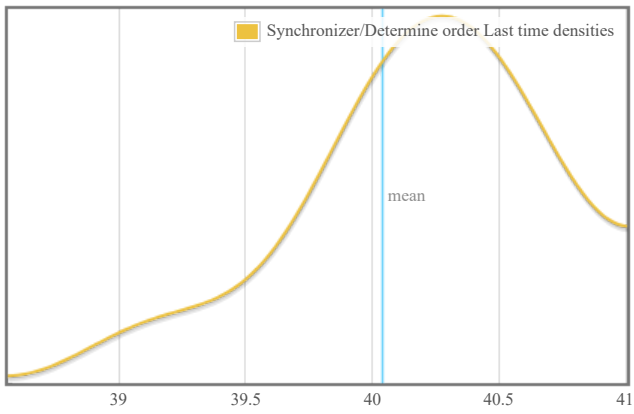


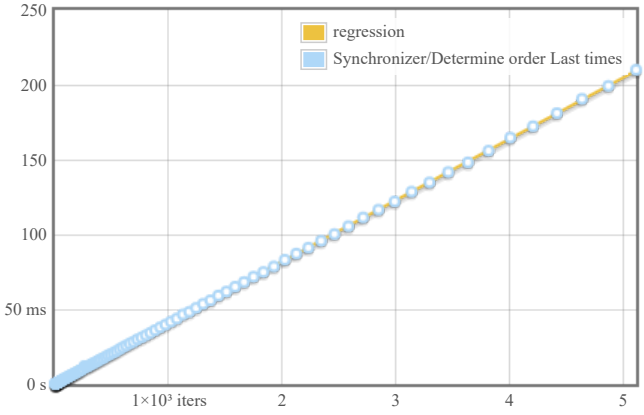


	lower bound	estimate	upper bound
OLS regression	40.5 µs	40.6 µs	40.8 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	39.4 µs	39.7 µs	39.9 µs
Standard deviation	534 ns	680 ns	993 ns

Outlying measurements have moderate (12.8%) effect on estimated standard deviation.

Synchronizer/Determine order Last

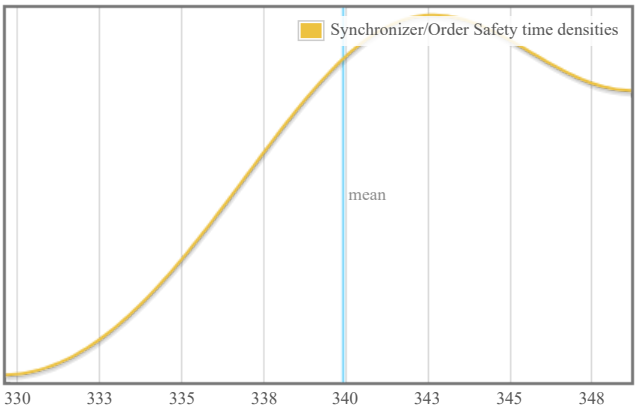


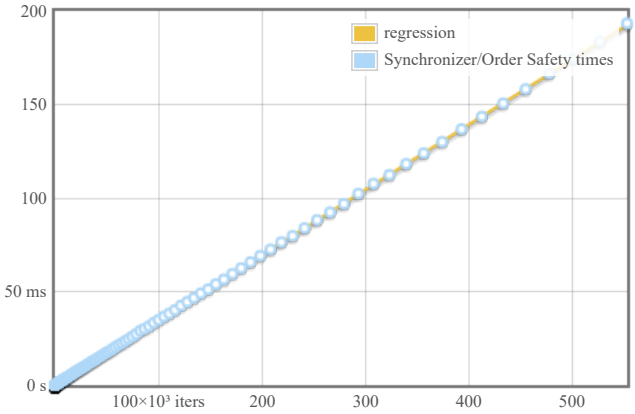


	lower bound	estimate	upper bound
OLS regression	41.0 µs	41.0 µs	41.1 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	39.9 µs	40.0 µs	40.2 µs
Standard deviation	436 ns	553 ns	663 ns

Outlying measurements have slight (8.5%) effect on estimated standard deviation.

Synchronizer/Order Safety

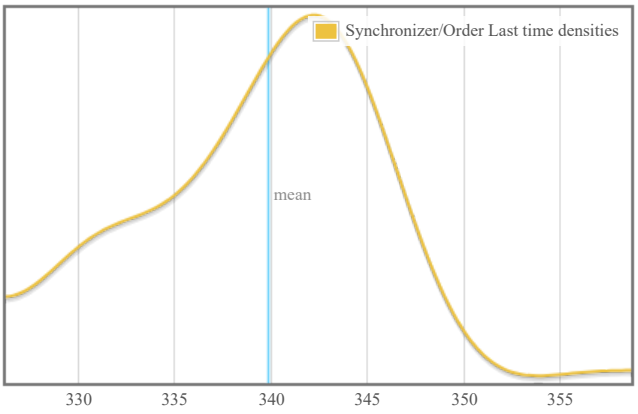


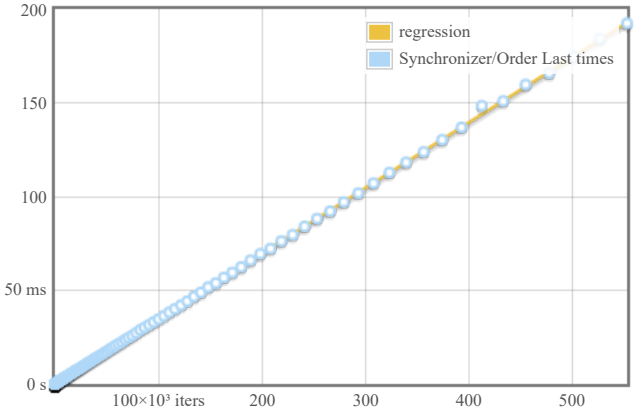


	lower bound	estimate	upper bound
OLS regression	348 ns	349 ns	349 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	338 ns	340 ns	341 ns
Standard deviation	4.02 ns	4.73 ns	5.58 ns

Outlying measurements have moderate (13.7%) effect on estimated standard deviation.

Synchronizer/Order Last

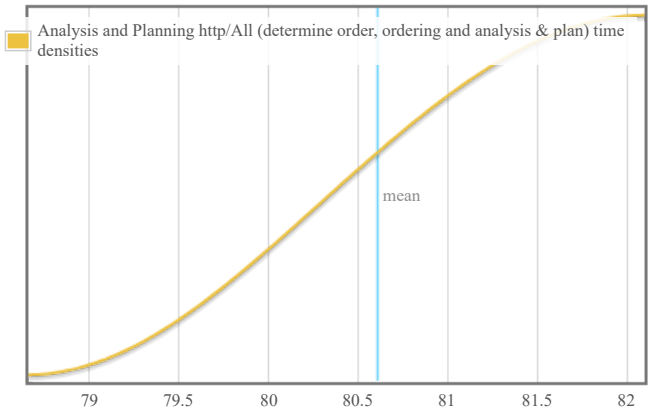


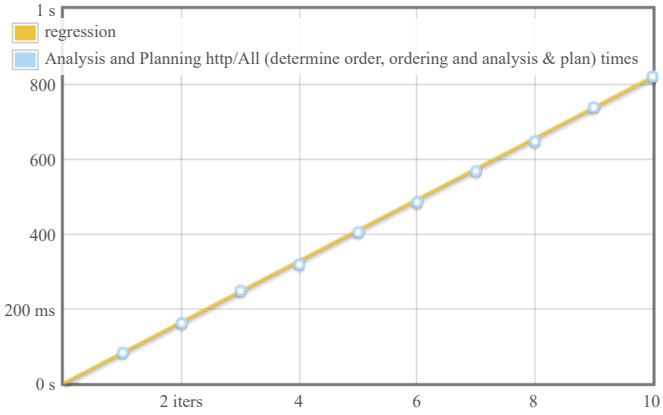


	lower bound	estimate	upper bound
OLS regression	348 ns	349 ns	351 ns
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	338 ns	340 ns	342 ns
Standard deviation	4.82 ns	5.91 ns	8.12 ns

Outlying measurements have moderate (20.0%) effect on estimated standard deviation.

Analysis and Planning http/All (determine order, ordering and analysis & plan)



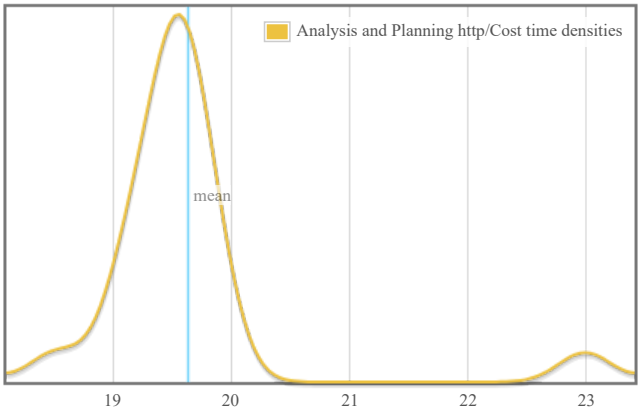


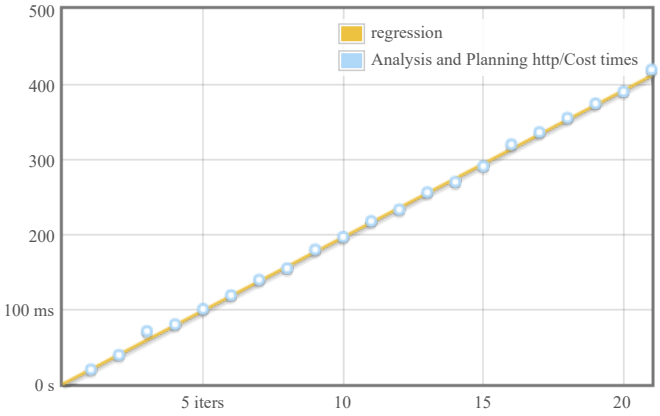
	lower bound	estimate	upper bound
OLS regression	80.6 ms	81.8 ms	83.3 ms
R² goodness-of-fit	0.999	1.000	1.000

Mean execution time	79.9 ms	80.6 ms	81.2 ms
Standard deviation	730 µs	1.03 ms	1.43 ms

Outlying measurements have slight (9.0%) effect on estimated standard deviation.

Analysis and Planning http/Cost

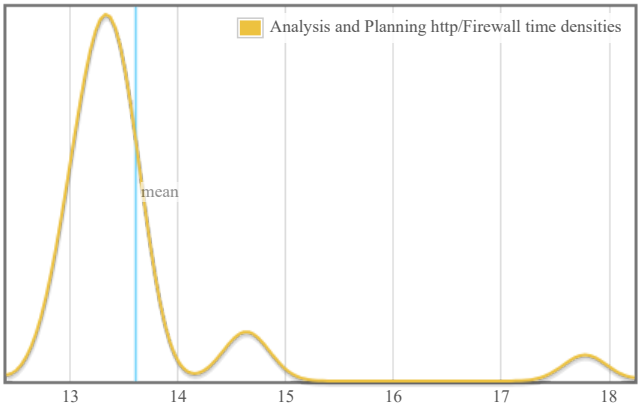


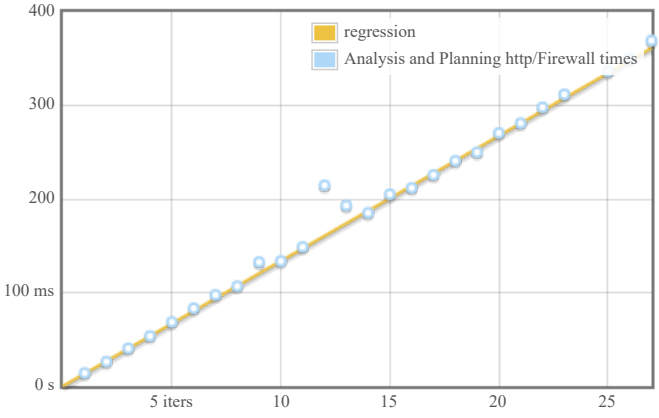


	lower bound	estimate	upper bound
OLS regression	19.3 ms	19.6 ms	19.9 ms
R² goodness-of-fit	0.998	0.999	1.000
Mean execution time	19.4 ms	19.6 ms	20.2 ms
Standard deviation	261 µs	840 µs	1.55 ms

Outlying measurements have moderate (13.5%) effect on estimated standard deviation.

Analysis and Planning http/Firewall

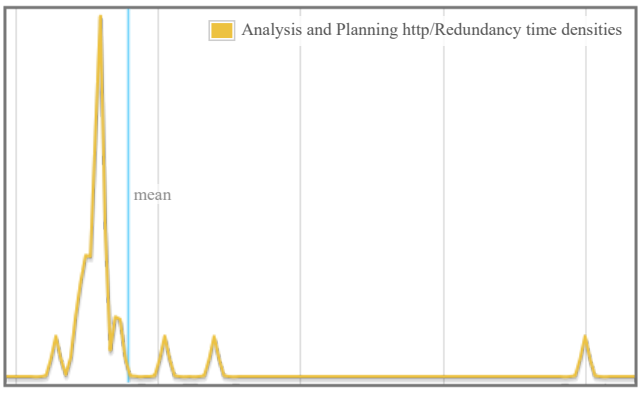


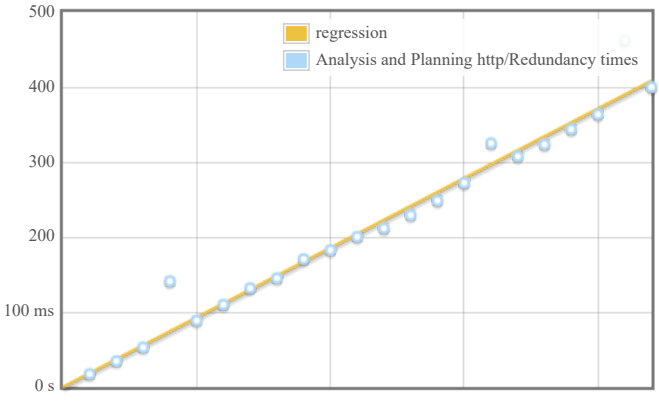


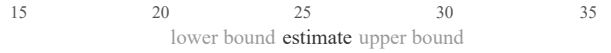
	lower bound	estimate	upper bound
OLS regression	13.1 ms	13.4 ms	13.6 ms
R² goodness-of-fit	0.962	0.989	0.999
Mean execution time	13.4 ms	13.6 ms	14.6 ms
Standard deviation	355 µs	978 µs	1.86 ms

Outlying measurements have moderate (33.2%) effect on estimated standard deviation.

Analysis and Planning http/Redundancy



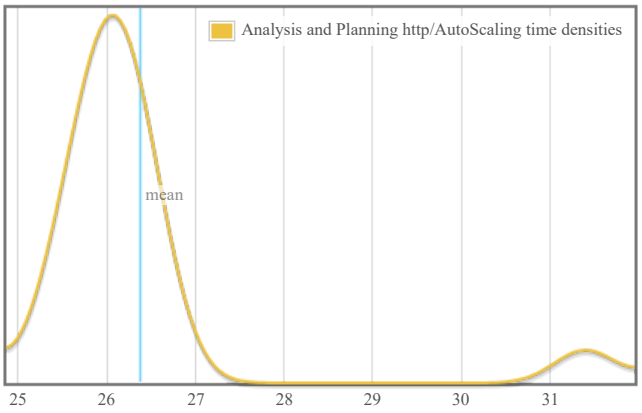


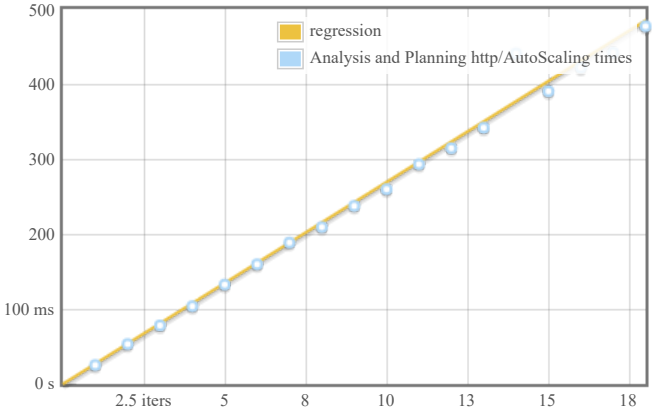


	lower bound	estimate	upper bound
OLS regression	16.7 ms	18.6 ms	20.5 ms
R² goodness-of-fit	0.932	0.964	0.999
Mean execution time	18.0 ms	19.0 ms	22.2 ms
Standard deviation	956 µs	3.81 ms	7.47 ms

Outlying measurements have severe (80.6%) effect on estimated standard deviation.

Analysis and Planning http/AutoScaling

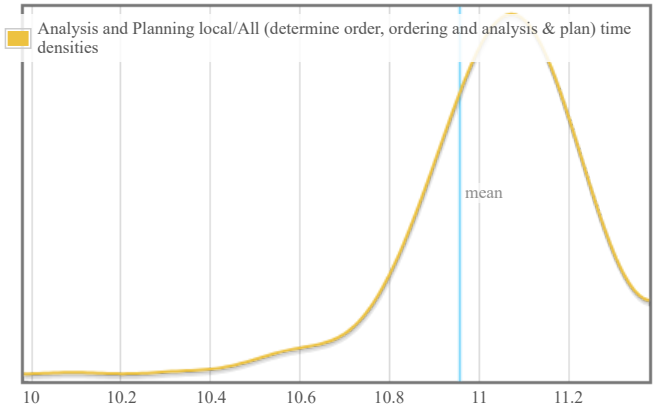


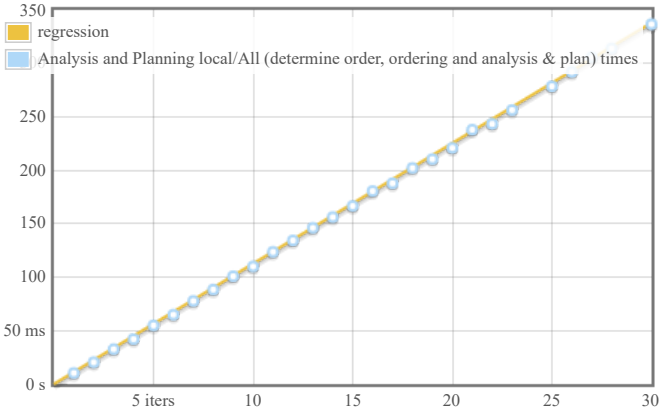


	lower bound	estimate	upper bound
OLS regression	26.0 ms	26.9 ms	28.9 ms
R² goodness-of-fit	0.962	0.986	1.000
Mean execution time	26.0 ms	26.4 ms	27.6 ms
Standard deviation	295 µs	1.33 ms	2.54 ms

Outlying measurements have moderate (15.7%) effect on estimated standard deviation.

Analysis and Planning local/All (determine order, ordering and analysis & plan)

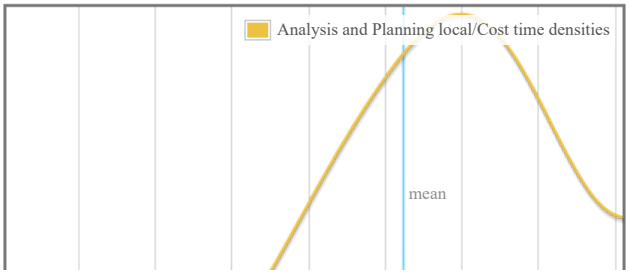


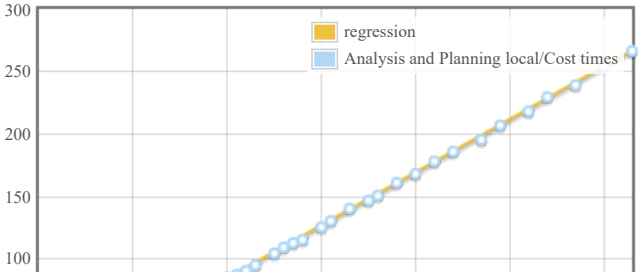


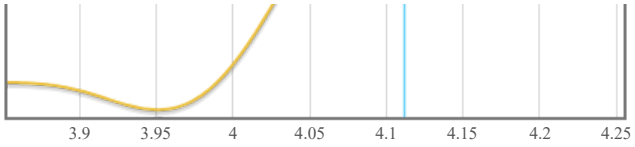
	lower bound	estimate	upper bound
OLS regression	11.2 ms	11.3 ms	11.3 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	10.8 ms	11.0 ms	11.0 ms
Standard deviation	179 µs	277 µs	406 µs

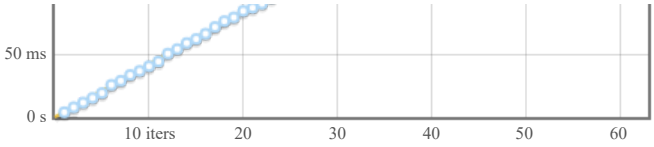
Outlying measurements have slight (6.7%) effect on estimated standard deviation.

Analysis and Planning local/Cost





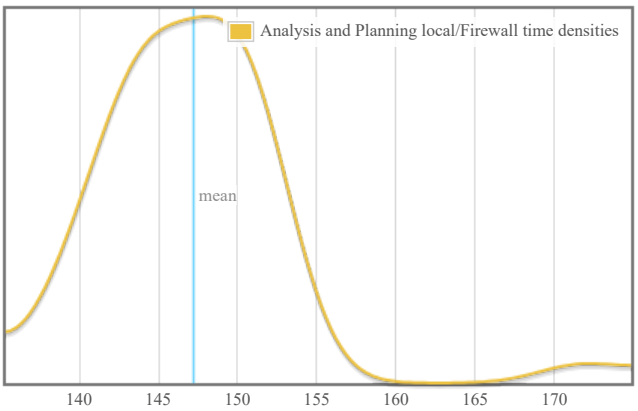


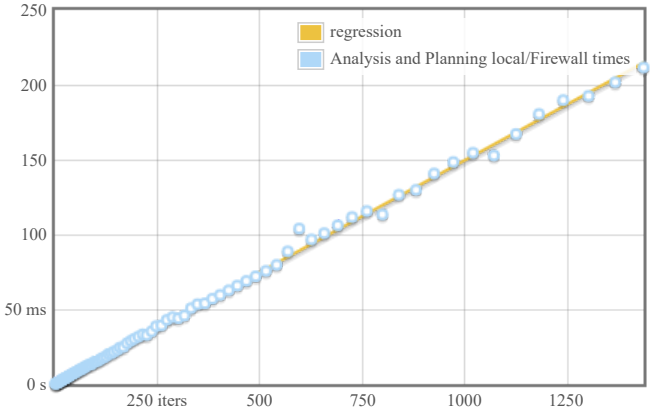


	lower bound	estimate	upper bound
OLS regression	4.22 ms	4.24 ms	4.25 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	4.08 ms	4.11 ms	4.14 ms
Standard deviation	65.8 µs	86.5 µs	115 µs

Outlying measurements have slight (6.6%) effect on estimated standard deviation.

Analysis and Planning local/Firewall

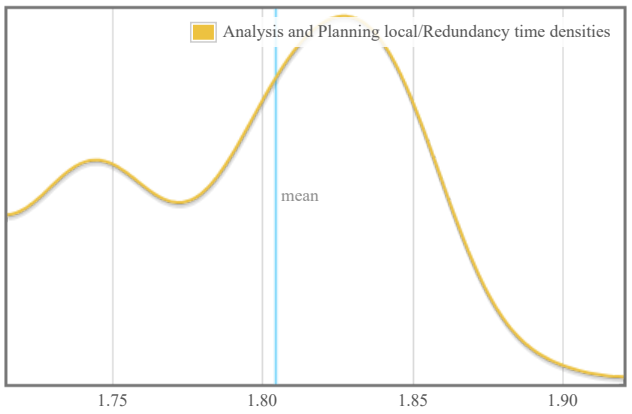


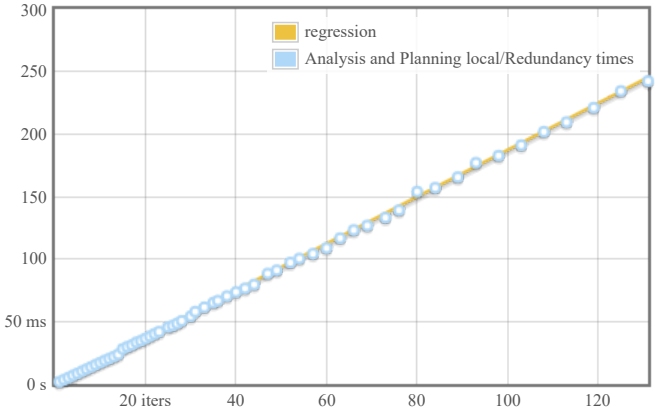


	lower bound	estimate	upper bound
OLS regression	148 µs	150 µs	152 µs
R² goodness-of-fit	0.997	0.998	0.999
Mean execution time	146 µs	147 µs	150 µs
Standard deviation	3.74 µs	5.52 µs	9.31 µs

Outlying measurements have moderate (35.9%) effect on estimated standard deviation.

Analysis and Planning local/Redundancy

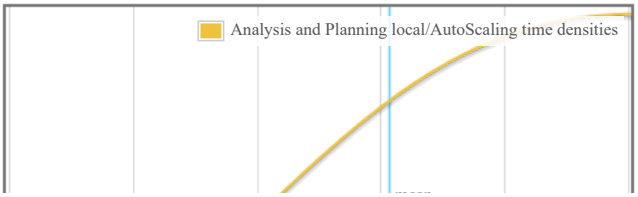


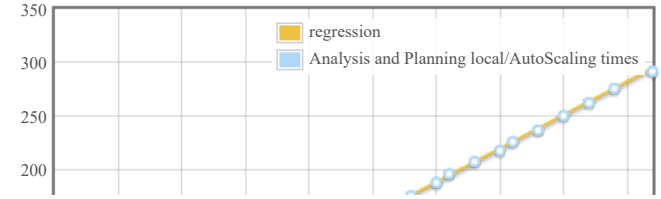


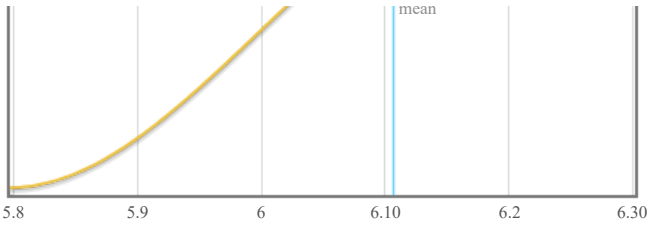
	lower bound	estimate	upper bound
OLS regression	1.86 ms	1.87 ms	1.88 ms
R² goodness-of-fit	0.999	1.000	1.000
Mean execution time	1.79 ms	1.80 ms	1.82 ms
Standard deviation	39.5 µs	45.1 µs	54.0 µs

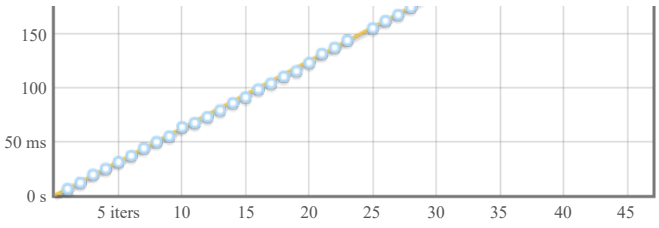
Outlying measurements have moderate (13.0%) effect on estimated standard deviation.

Analysis and Planning local/AutoScaling





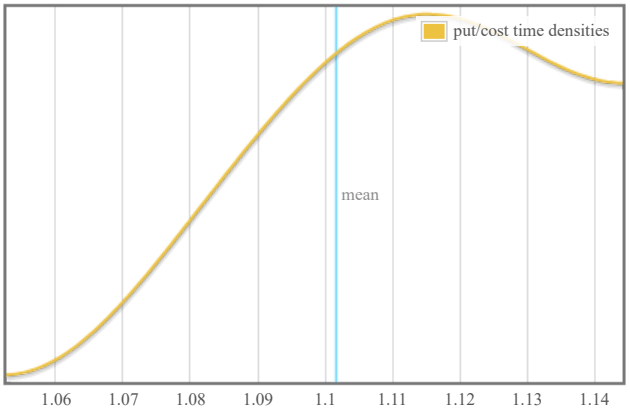


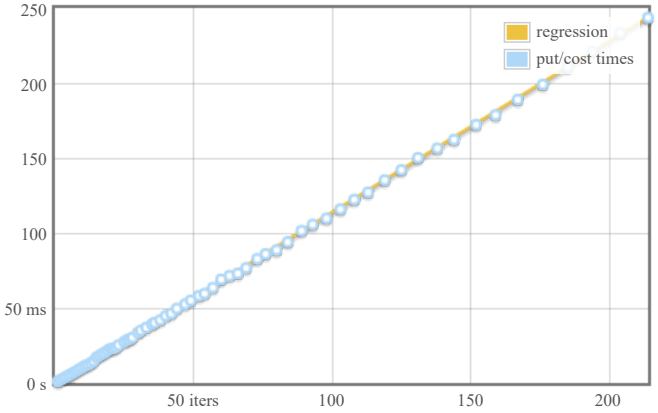


	lower bound	estimate	upper bound
OLS regression	6.25 ms	6.27 ms	6.30 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	6.06 ms	6.11 ms	6.14 ms
Standard deviation	97.9 µs	115 µs	139 µs

Outlying measurements have slight (2.6%) effect on estimated standard deviation.

put/cost

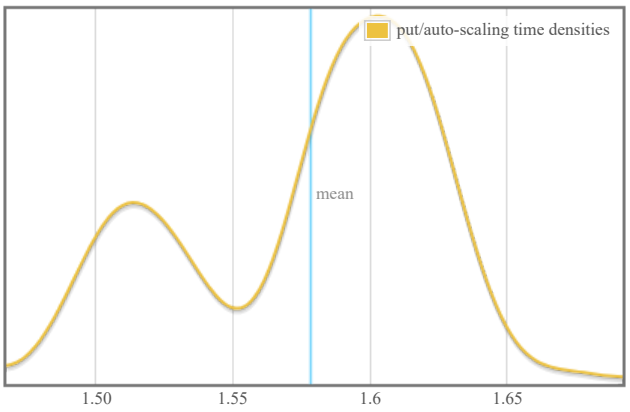


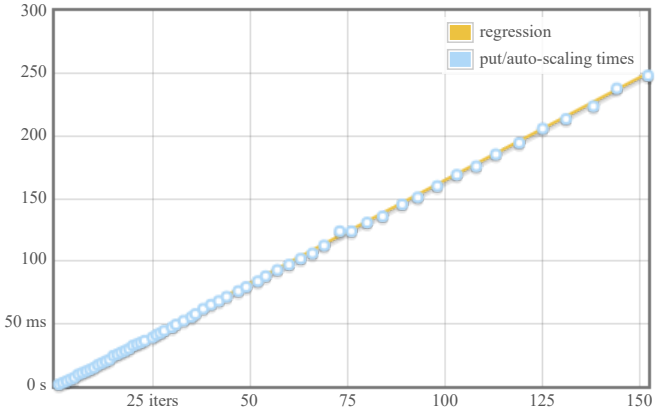


	lower bound	estimate	upper bound
OLS regression	1.14 ms	1.14 ms	1.14 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.09 ms	1.10 ms	1.11 ms
Standard deviation	20.2 µs	22.7 µs	25.9 µs

Outlying measurements have slight (9.7%) effect on estimated standard deviation.

put/auto-scaling

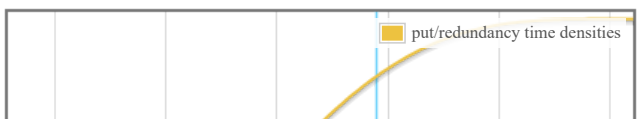


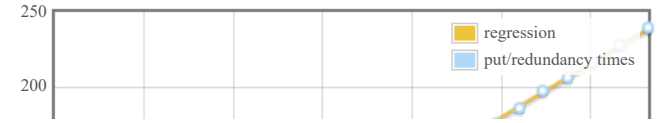


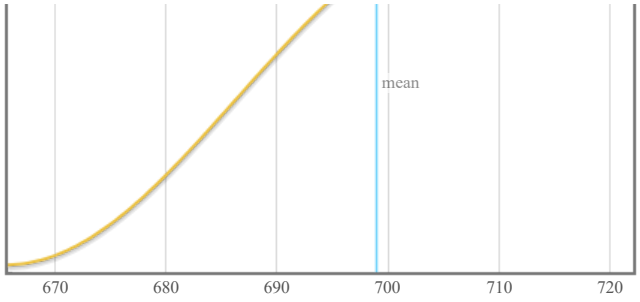
	lower bound	estimate	upper bound
OLS regression	1.64 ms	1.65 ms	1.66 ms
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	1.56 ms	1.58 ms	1.59 ms
Standard deviation	39.8 µs	46.4 µs	55.1 µs

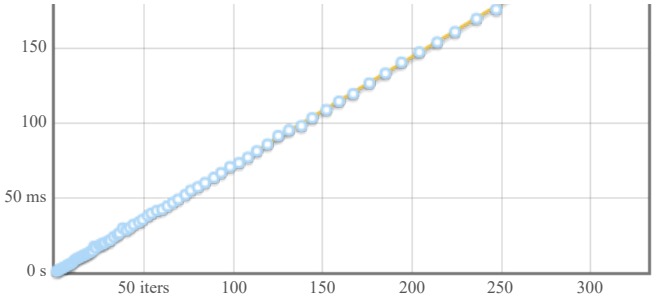
Outlying measurements have moderate (17.1%) effect on estimated standard deviation.

put/redundancy





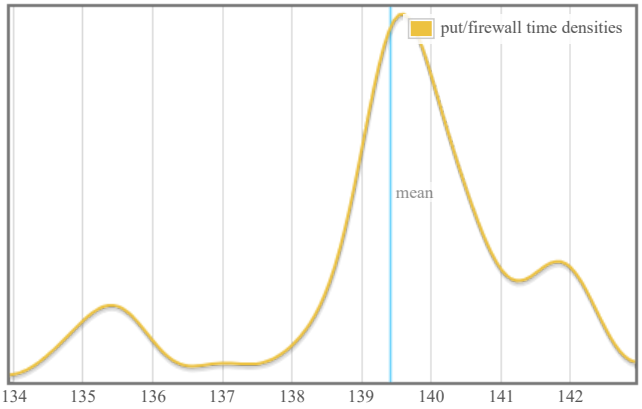


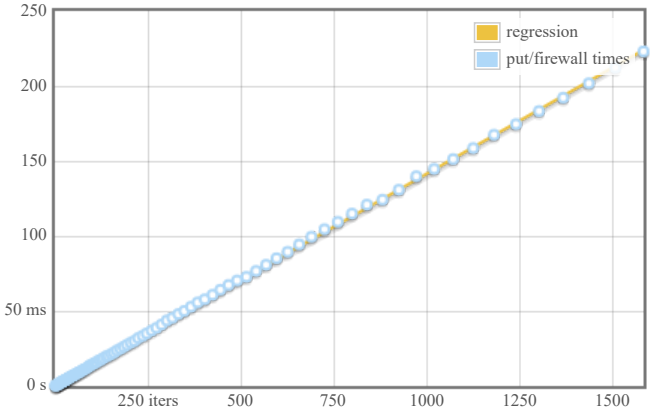


	lower bound	estimate	upper bound
OLS regression	718 µs	720 µs	722 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	695 µs	699 µs	703 µs
Standard deviation	11.3 µs	13.2 µs	15.5 µs

Outlying measurements have slight (9.8%) effect on estimated standard deviation.

put/firewall





	lower bound	estimate	upper bound
OLS regression	141 µs	142 µs	142 µs
R² goodness-of-fit	1.000	1.000	1.000
Mean execution time	139 µs	139 µs	140 µs
Standard deviation	1.43 µs	1.88 µs	2.35 µs

Outlying measurements have slight (6.9%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own. The charts in each section are active; if you hover your mouse over data points and annotations, you will see more details.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x* axis indicates the number of loop iterations, while the *y* axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression prediction of execution time for a given number of loop iterations. Ideally, all measurements will be on (or very near) this line.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R² goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R² should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be. (Hover the mouse over the table headers to see the confidence levels.)

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.