Auto-Scaling Techniques for Spark Streaming

Master-Thesis von Seyedmajid Azimi Gehraz aus Iran
Tag der Einreichung:

1. Gutachten: Prof. Dr. rer. nat. Carsten Binnig
2. Gutachten: Thomas Heinze

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Data Management

Auto-Scaling Techniques for Spark Streaming

Vorgelegte Master-Thesis von Seyedmajid Azimi Gehraz aus Iran

1. Gutachten: Prof. Dr. rer. nat. Carsten Binnig
2. Gutachten: Thomas Heinze

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den October 28, 2018

____

(Seyedmajid Azimi Gehraz)

Contents

List of Figures

List of Tables

List of Code Snippets

List of Algorithms

Abstract

Many modern cloud-based applications face ever changing dynamic workloads. The rise of IOT with devices around the globe getting online and offline with no particular pattern, makes the situation even worse. This problem is magnified in streaming applications that are sensitive to latency. Since the workload pattern is partially unpredictable, application developers inevitably over-provision resources in order to meet customers' requirements which leads to cost inefficiencies. To resolve this issue, researchers have come up with a new class of resource managers that automatically determine workload characteristics and adopt the application to fit the workload.

In this thesis a dynamic resource manager is developed for Spark Streaming. Q-Learning is used as a foundation algorithm. To overcome the slow learning problem of Q-Learning, numerous optimizations are applied. Namely, an optimized state space initializer that is independent of workload is developed. Furthermore, Value Iteration algorithm is developed to help initialize the state table by applying minimal training period. The solution can be used by any Spark Streaming job without applying any modification to Spark.

The proposed implementation is evaluated under an extensive list of experiments based on two real world workloads. It is able to outperform the default Spark's dynamic resource manager. However, it also illustrates the fragility of the approach by changing a single configuration parameter. As a consequence, the problem of dynamic resource management still hasn't been solved. It has been moved to configuration space. Application developers have to run many manual experiments to find out the optimal configuration.

1 Introduction

Cloud computing has been on rise over the last decade. Parts of this popularity is due to its inherent features. It lets application developers to run their applications on virtual infrastructure. Virtual infrastructure lays the foundation of on-demand infrastructure. Developers acquire and release resources as required by workload. Examples of cloud providers are Amazon AWS [1], Microsoft Azure [50] and Google Cloud [25]. Today's cloud infrastructure is widely used by many customers for different purposes such as batch processing, serving static content, storage servers and alike.

As cloud environment brings up *elasticity* [32], it also introduces a new set of challenges and problems. Modern applications face fluctuating workloads. Typically, if a workload is *predictable*, resources are allocated ahead of time before load-spike starts. However, in many other scenarios predicting even near future workload is a not so easy task. Even though running an application in cloud environments helps to overcome a long standing problem of *over-provisioning*, low utilization is still one of the major problems of cloud applications. This has been confirmed by multiple studies [19] [54].

The root of the problem is originated from the fact that, most developers do not have enough insight about bottom and peak workload of their application. Thus, they fail to define an effective scaling strategy. Therefore, they end up with conservative strategies which in turn leads to low utilization. Hence, we need a system that automates the process of resource allocation. Auto-Scaling has been well studied in the context of web application. [28] [21] [33] are just a few examples. Chapter 6 explores more techniques and strategies.

The ultimate goal of an Auto-Scaling system is to automate the process of acquiring and releasing *resources* in order to minimize the *cost* with minimum violation of *service level objectives* (SLO). The definition of *resource* depends on the context. As an example, for a stateless web applications it means virtual machines or containers that run web server software. For an Auto-Scaling system to adjust required resources, it shall consider different aspects of the application and its environment. Additionally, the term *cost* is also defined in the context. As an example, it might mean monetary cost or just numerical value of resources. SLOs are predefined rules that shall not be violated during application runtime. These rules are also defined in the context of application.

Data Stream Processing Systems are data processing systems that process *unbounded* stream of data unlike their *batch-oriented* counterparts. With the ever increasing adoption of IoT applications, it is critical to design stream processing systems that handles the incoming messages with high throughput and low latency. With static workloads, these problems could be solved by dominating stream processing systems like Apache Spark [7], Apache Storm [56] and Apache Flink [2]. However, the problem of low utilization still holds in data stream processing systems. This leads us to a new generation of stream processing systems called *Elastic Data Stream Processing Systems* that applies elasticity concepts to stream processing system.

Prior to this thesis a number of studies [14], [29] have been performed on elastic stream processing system. One of the dominating stream processing systems is Apache Spark which supports both batch and stream processing workloads. In order to support both workloads, Apache Spark has a unique architecture that partitions the input workload into predefined window of batches – an architecture known as micro batching. With this common architecture as a foundation, number of interesting challenges arise that need to be considered for elastic workloads. For example, micro-batching introduces a small latency before each record gets processed. How is record processing latency affected by Auto-Scaling decisions? How does length of micro-batch interact with Auto-Scaling system? How much micro-batch recomputation is caused by Auto-Scaling decisions?

This thesis will focus on dynamic resource allocation in the context of Apache Spark. An extensible framework will be developed based on a prior work by Kielbowicz [42]. This thesis will extend the existing prototype and implement multiple Auto-Scaling techniques for Apache Spark Streaming and will evaluate these techniques using real-world workloads. The ultimate goal is to identify how the architecture of Spark Streaming influences the performance of the different Auto-Scaling techniques.

1.1 Requirements of Thesis

Since Auto-Scaling is a quite wide realm, this thesis focuses on Apache Spark Streaming with a couple of predefined requirements. Any proposed solution must adhere to the following requirements and limitations to a large extent.

- **Online Decision Making**. Since Spark Streaming is an online data stream processing system, scaling decisions shall be applied in an online manner without causing any downtime on target system.
- **QoS Objectives**. Scaling decisions shall avoid violating predefined SLOs as much as possible. Understandably, violating SLOs is an inevitable incident. However, this should be kept under an acceptable level.
- **Extendibility**. Proposed solution shall provide some degree of extendibility such that, future modifications can be applied without any major code refactoring.
- **Configurability**. Configurability is an important aspect of any complex software. It should be easy for a system administrator or *DevOps* team to configure the Auto-Scaling system.
- **Workload Independence**. Aforementioned Auto-Scaling system should not impose any assumption on type of workload running under Apache Spark Streaming.
- **Computationally Feasible**. As mentioned in first requirement, Auto-Scaling system should make computationally feasible actions. In other words, it should generate results in order of seconds.
- **Without Spark Core Modification**. In order to make the solution extensible as much as possible, it is not allowed to modify `spark-core` or `spark-streaming` packages.

1.2 Summary

As mentioned this thesis will focus on dynamic resource allocation in the context of Apache Spark. The thesis is organized as follows. Chapter 2 introduces and explains basics of Auto-Scaling techniques. Chapter 3 introduces the architecture of Apache Spark Streaming. Chapter 4 explains structure and design considerations of this thesis including implementation details. Chapter 5 evaluates the implementation under different workloads using different set of configuration parameters. Chapter 6 includes discussion of prior and related work. Finally, chapter 7 concludes.

2 Auto-scaling Techniques

As mentioned in Chapter 1 the key characteristic of cloud environments is *elasticity* behavior. However, manually adjusting resources are is not an effective approach to exploit this feature. Hence, we need to automate this procedure with minimal human intervention. This chapter introduces foundations of Auto-Scaling techniques. Different techniques and architectures will be discussed from a high level standing point. A detailed introduction into the topic is given by Lorido-Botran, Miguel-Alonso, and Lozano [48]. This chapter is organized as follows. Section 2.1 introduces basic concepts of Auto-Scaling. Section 2.2 defines general architecture of an Auto-Scaler. Section 2.3 clarifies which sort of actions can be applied by an Auto-Scaler. Section 2.4 classifies different techniques and briefly explains each category. Finally, Section 2.5 concludes.

2.1 Basic Concepts

The ultimate goal of an Auto-Scaling system is to automate the process of acquiring and releasing *resources* in order to minimize the *cost* with minimum violation of *service level objectives* (SLO). However, *resource* is a broad and context-dependent term. It refers to any form processing unit – engine – that provides application developers some form of computation power. This general purpose definition is broad enough to capture different kinds. In most cases, it means virtual machines allocated by cloud provider. In more modern distributed systems, it may refer to *container*s like Docker Container Engine [20]. However, a resource might be as simple as a single process or thread.

The term *cost* refers to any form of expenditure that users pay in order to acquire a resource. It doesn't necessarily mean *monetary* cost. It can also refer to numerical values of resources, like number of virtual machines or number of running processes. Although minimizing cost is the ultimate goal of any Auto-Scaling system, not in all cases cost reduction is desirable. It should be achieved with respect to defined *Service Level Objectives* (SLO).

Service Level Objectives are any predefined rules that shall be respected during application runtime. The following defines a couple of SLOs for different applications:

- 99 percentile round-trip latency of requests in a web application should be less than 150 milliseconds.
- All committed records in master database must be replicated with a maximum delay of 5 milliseconds.
- All messages pushed by a producer, should be processed by respective consumers in less than 5 minutes.
- At least 95% of images published in the last 24 hours should be served by cache servers.

Defining effective and meaningful SLOs is a challenge on its own. Typically, it requires coordination with business team. Having profound understanding of system requirements is critical to define appropriate SLOs [61]. However, it is out of the scope of this thesis. In this thesis, it is assumed that there is a well established set of pre-defined SLOs.

From a high level point of view, Auto-Scaling is a *trade-off* amongst cost and violation of SLOs. Resource *under-provisioning* will degrade performance and leads to SLO violations, while on the other hand, resource *over-provisioning* leads to idle resources, hence unnecessary costs. To make an effective decision, an Auto-Scaler needs to consider application and its environment:

- **Infrastructure pricing model**. Some infrastructure providers charge customers on hourly basis. That is, if customer acquires a resource at 10:30AM and releases it at 11:30AM, the customer is charged for two hours. Some other service providers might charge on minute basis. Pricing model has a huge impact on aggressiveness of the Auto-Scaler system, since it makes some decisions pointless.

- **Service level objectives**. Each application has its own set of objectives that should be adhered by Auto-Scaling system. These SLOs might be defined and applied at *soft* and *hard* levels. Violating an SLO at soft level is not critical, albeit alarming. However, violating a resource at hard level is a critical issue and is a negative point for an Auto-Scaler.
- **Acquire/Release delay**. Depending on type of the resource, it might take some time for the resource to become responsive and ready to process user requests. For example, booting a virtual machine typically takes couple of minutes, whereas launching a container takes time in order of seconds. An Auto-Scaler shall consider whether acquiring and releasing a heavy weight resource in a *zig-zag* manner worth the overhead or not.
- **Unit of allocation**. In some cases, it might be beneficial to allocate multiple instances of a same resource at once. This might be due to the startup and initialization overhead or it might be the case that Auto-Scaler predicted a huge load spike in near future.

2.2 Generic Auto-Scaler Architecture

Figure 2.1 illustrates generic architecture of an Auto-Scaler. This architecture is broad enough to capture different kinds of applications.

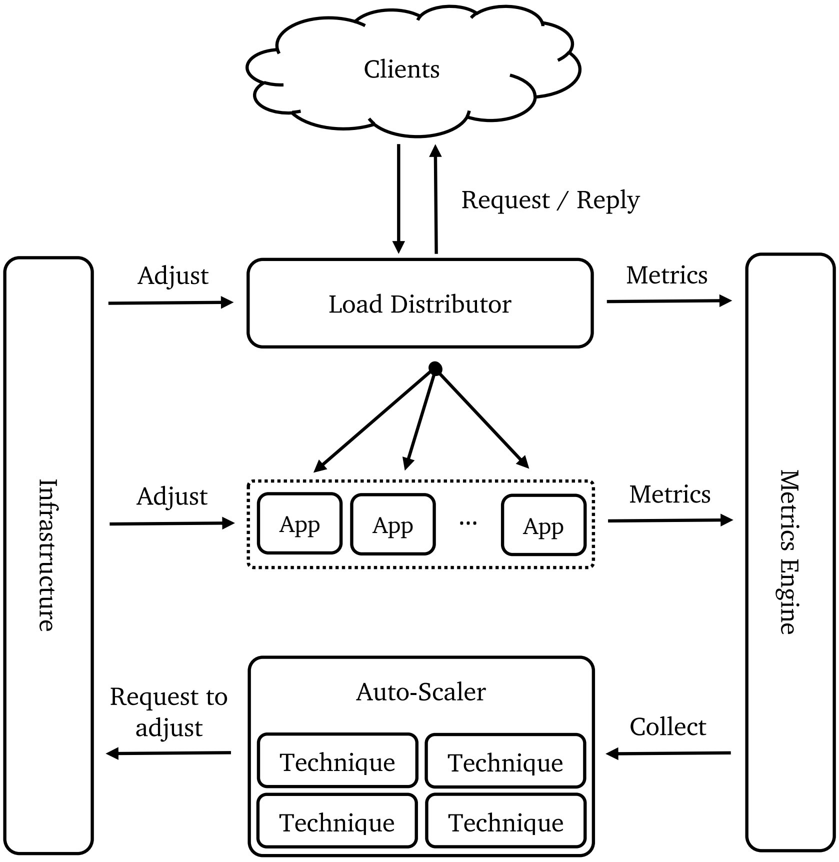


Figure 2.1: Architecture of a Generic Auto-Scaler

An Auto-Scaling system typically consists of following sub-components. Table 2.1 summarizes all components of the system.

Clients Most kinds of applications, typically have some form of client that sends requests to the system and either waits to get a reply or operates under *fire-and-forget* strategy. It shall be noted that, a client is not necessarily an end user. In today's modern distributed applications, an application might be client of another application.

Component	Description
Clients	Users of the application which might be applications on their own
Load Distributor	Distributes incoming requests to application instances
Application	Runs business logic defined by developer
Metrics Engine	Monitors and collects metric from application and provides it to other components
Infrastructure	Provides API to adjust (acquire/release) resources
Auto-Scaler	Runs the auto-scaling algorithm based on metrics collected by Metrics Engine

Table 2.1: Summary of Auto-Scaler Components

Load Distributor In order to provide some degree of *transparency*, usually clients connect to a Load Distributor component. It is the responsibility of the Load Distributer that *proxies* client's request to application. Load Distributor is also a generic component and might represent different technology in read-world applications. In the context of a web application, it might be an HTTP load balancer. Even a *Message Broker* can also be represented as a form of Load Distributer. It shall be noted that Load Distributor itself can be replicated or sharded for *high availability* or *scalability* reasons.

Application The application component runs the business logic. This architecture doesn't impose any limitation on application architecture. It might be a simple stateless web application. It might access to a back-end cache or database service. It might push some messages to a message broker, as a result of client request. It might be just a simple process consuming messages provided by a message broker. It might forward client requests to other applications for further processing.

Infrastructure Typically, when an Auto-Scaler system decides to take any action, it doesn't touch the application directly. In order to provide *separation of concerns*, this responsibility is handed over to infrastructure via an API provided by resource/service provider. An important issue that shall be noted here is that, service provider may schedule resource changes and execute them some time later. Thus, resource changes might not take effect immediately at the moment Auto-Scaler requests them. This fact shall be considered by Auto-Scaler system.

Metric Engine Auto-Scaler system needs to have a good insight on current status of the application and incoming requests. Metric engine – known as *monitoring engine* – takes the responsibility of measuring and collecting different aspects of the application. The term *metric* refers to any form of measurable aspect of an application or its environment. Ghanbari et al. [24] has proposed a list of different types of metrics that could be exploited for different purposes.

- **Hardware** dependent metrics such as CPU usage, disk access time, memory usage, network bandwidth usage, network latency.
- **Operating System** provided metrics such as CPU-time, page faults, real memory.
- **Load balancer** provided metrics such as size of request queue length, session rate, number of current sessions, transmitted bytes, number of denied, requests, number of errors.
- **Web server** provided metrics such as transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, . . .).
- **Application server** provided metrics such as total threads count, active threads count, used memory, session count, processed requests, pending requests, dropped requests, response time.
- **Database server** provided metrics such as number of active threads, number of transactions in a particular state (e.g. write, commit, roll-back, . . .).

Since storing and reporting metrics has its own overhead, typically metrics engine aggregates collected values at different scale depending on how fresh it is. Rationally, fresh values are more important for Auto-Scaling system. As an example, it might provide near real time values for about 15 minutes. Then, for the last 5 hours, collected values are aggregated by a window of one minute. For last two days, it is aggregated by a window of 15 minutes and finally, for any record older than last two days, it is aggregated on hourly basis. Whether these aggregated values are sufficient for Auto-Scaler to make an accurate decision is out of the scope of this thesis. However, it may be a good idea to empirically adjust this system until it fits the requirements of Auto-Scaler system.

Auto-Scaler This component is the core of Auto-Scaling system. Typically, an Auto-Scaler is triggered periodically. When triggered, it collects metrics from Metrics Engine and offloads them to one or multiple technique implementations. It shall be noted that, an Auto-Scaler might utilize different set of Auto-Scaling techniques simultaneously – for different stages of the application as an example. Even it might utilize a single implementation under different configurations. This architecture does not impose any limitation on the order of techniques. Then, based on some preferences or ordering mechanism, it chooses the *final decision*. Finally, it requests the Infrastructure API to change the number of resources. For the sake of understandability, Algorithm 1 describes this procedure.

Algorithm 1: General Work-Flow of an Auto-Scaler

```
// different implementations of techniques
implementations ← []
// decision of each implementation
decisions ← []
// final decision of Auto-Scaler
finalDecision ← null
// instantiate as many techniques as required
for i ← 0 to n do
    implementations[i] ← InstantiateTechnique(i)
end
repeat
    // load monitoring data from metrics engine
    currentValue ← GetCurrentMetricsFromMetricsEngine()
    // initialize decisions
    decisions ← []
    for i ← 0 to n do
        impl ← implementations[i]
        decisions[i] ← GetDecision(impl, currentValue)
    end
    // calculate final decision based on some weight or ordering mechanism
    finalDecision ← GetFinalDecision(decisions, currentValue)
    // request infrstructure API to adjust resources
    RequestInfrastructureAPI(finalDecision)
until Auto-Scaler is running
```

2.3 Actions

In case Auto-Scaler decides to take an action, it notifies Infrastructure via an API regarding its decision. Note that Infrastructure either commits the action *synchronously* when the request is made by the Auto-Scaler or schedules the action*asynchronously* for future. This thesis, assumes three possible actions. Table 2.2 summarizes feasible actions for an Auto-Scaler.

Action	Description
Scale-In	Remove/Release one or more resources
Scale-Out	Acquire/Add one or more resources
No-Action	Do nothing

Table 2.2: Feasible Actions of an Auto-Scaler

It's noteworthy that in all cases, Auto-Scaler is allowed to store any history of actions taken so far. In fact, it is a special category of Auto-Scalers known as *stateful* Auto-Scalers. Refer to Section 2.4 for further discussion and explanation on taxonomy of Auto-Scalers.

Another aspect of taking an action is that, Auto-Scaler is allowed to Scale-In/Out *horizontally* or *vertically* in each round independent of previous rounds. Horizontal Scale-In/Out refers to a category of actions that acquires or releases resources in parallel to each other. For example, in the context of a web application, adding or removing one or more virtual machines is considered as a horizontal scaling action. While on the other hand, Auto-Scaler might decide to just scale by adding hardware resources. For example, it might decide to add more RAM or remove couple of CPU cores in one specific virtual machine. This kind of scaling action is considered as vertical scaling action.

Actions are not necessarily applied at a constant rate. Auto-Scaler is in full charge of taking actions at *exponential* rates. For example, in consecutive rounds, an Auto-Scaler can decide to acquire resources by a power of two. This also applies for Scale-Out actions. Nothing hampers an Auto-Scaler from changing rate of scaling actions in each round. It might even decide to apply different rates for different stages of the application like *startup* phase, or *near-ending* phase, etc.

Last but not least, an Auto-Scaler might decide to apply a *grace period* after taking an action independent of previous rounds. A grace period is a time frame, in which Auto-Scaler does not take any further action in order to let cluster of resources stabilize. Similar to action rates, grace period can also be applied at different rates. For example, Auto-Scaler might increase grace period by a multiple of two.

2.4 Taxonomy of Auto-Scaling Techniques

Auto-Scalers can be modeled and classified in different categories. In a sense, it is a multi-dimensional space of features and characteristics. Table 2.3 lists and summarizes different dimensions of Auto-Scaling. In the rest of this chapter each dimension is described and explored in details.

Dimension	Description
Schedule-based versus *Rule-based*	Whether Auto-Scaler applies decisions manually or based on predefined rules.
Reactive versus *Proactive*	Whether Auto-Scaler reacts to workload changes or predict future workload ahead of time.
Execution mode	Whether Auto-Scaler is central component or operating in a distributed fashion.
Algorithm family	Which family of algorithms is applied to make the decision.

Table 2.3: Dimensions of Auto-Scaling

2.4.1 Schedule-Based versus Rule-Based

Some applications have a very basic cyclic workload pattern on daily basis which could be manually modeled as *cron* style jobs. Schedule-based systems can not adopt to unplanned workload changes. Since this type of Auto-Scalers are in conflict with thesis requirements, it won't be studied in this thesis.

On the other side of the extreme, there exists *rule-based* Auto-Scalers. These Auto-Scalers take actions based on set of rules defined by application developers inspired by business requirements. Each rule is based on one or multiple *constraints*. Each constraint consists of one or a set of conditions around some variable. Variables can be defined by *application* – number of tweets per second, as an example – or by its environment – CPU utilization, for example. For example, if average network bandwidth is more than 80% of maximum available bandwidth for 10 minutes, then take Scale-Out action.

The criteria for calculating these variables vary from application to application. Some applications might define minimum or maximum values for variables. In some other cases, average values might be useful. Furthermore, average/minimum/maximum values might be defined for a window of time or number of occurrences of specific event. Time or event windows in turn could be defined as a *static* window – where it moves by a fixed interval – or as a *sliding* window – where it smooths over elements.

2.4.2 Reactive vs Proactive

In another dimension, Auto-Scalers can be partitioned into two categories. *Reactive* approaches monitor the workload in order to find a meaningful change. Thereafter, they apply some algorithm to figure out the final decision. To view different family of algorithms, refer to Section 2.4.4. It's noteworthy to express that, in some applications by the time a reactive Auto-Scaler decides to take some action, it might be too late. In other words, taking an action in an already overloaded application might not be desirable in some cases. Taft et al. [58] argues that applying reactive approaches in the context of OLTP databases is not desirable. This leads us to another class of Auto-Scalers known as *proactive* Auto-Scalers.

Proactive Auto-Scalers try to predict *future* workload ahead of time before facing workload spikes. Hence, they are known as *predictive* approaches. Whether the term *future* is defined as near future or long term future depends on the type of Auto-Scaler. This has the advantage that, by the time load spike occurs, the required resources are already available, warmed-up and ready to respond user requests. As mentioned in Section 2.1, allocating some resources takes some time to become fully initialized. For example, adding a database replica is not an immediate action. It takes some time to re-replicate database records and validate the replicas. Thus, for some scenarios, even though reactive approaches might seem to be sufficient, but due to initialization latency it is not an applicable approach.

2.4.3 Execution Mode

The generic architecture which is described in Section 2.2 does not impose any limit on the operation and execution mode of the Auto-Scaler. In other words, Auto-Scaler itself might run as a *multi-instance* application. Many execution modes haven been proposed in literature. But they can be classified in following generic groups without loosing generality. Table 2.4 summarizes operation modes.

Execution Mode	Description
Global Controller	At a specific point in time, a single controller is responsible of taking actions.
Distributed Without Coordination	Auto-Scalers operate independent of each other without performing any form of coordination.
Distributed Coordinated	Auto-Scalers perform in distributed mode, but they are allowed to communicate and cooperate with each other.
Hierarchical Controllers	A hierarchy of controllers cooperate with each other to make final decision.

Table 2.4: Execution Modes of an Auto-Scaler

2.4.3.1 Global Controller

In this architecture, Auto-Scaler runs and controls the application as a *central* or *master* component. One or multiple *backup* or *slave* controllers might also accompany master node. In case master controller fails, one of the backup controllers kicks in and takes over the responsibility. In this model, it's the responsibility of master node to make decisions and execute actions. To detect failure of master controller, well established *cluster coordination* utilities like Apache Zookeeper [8] or CoreOS Etcd [17] exist. Figure 2.2 depicts a variation of original Auto-Scaler architecture that performs under master-slave model coordinated by a Zookeeper ensemble.

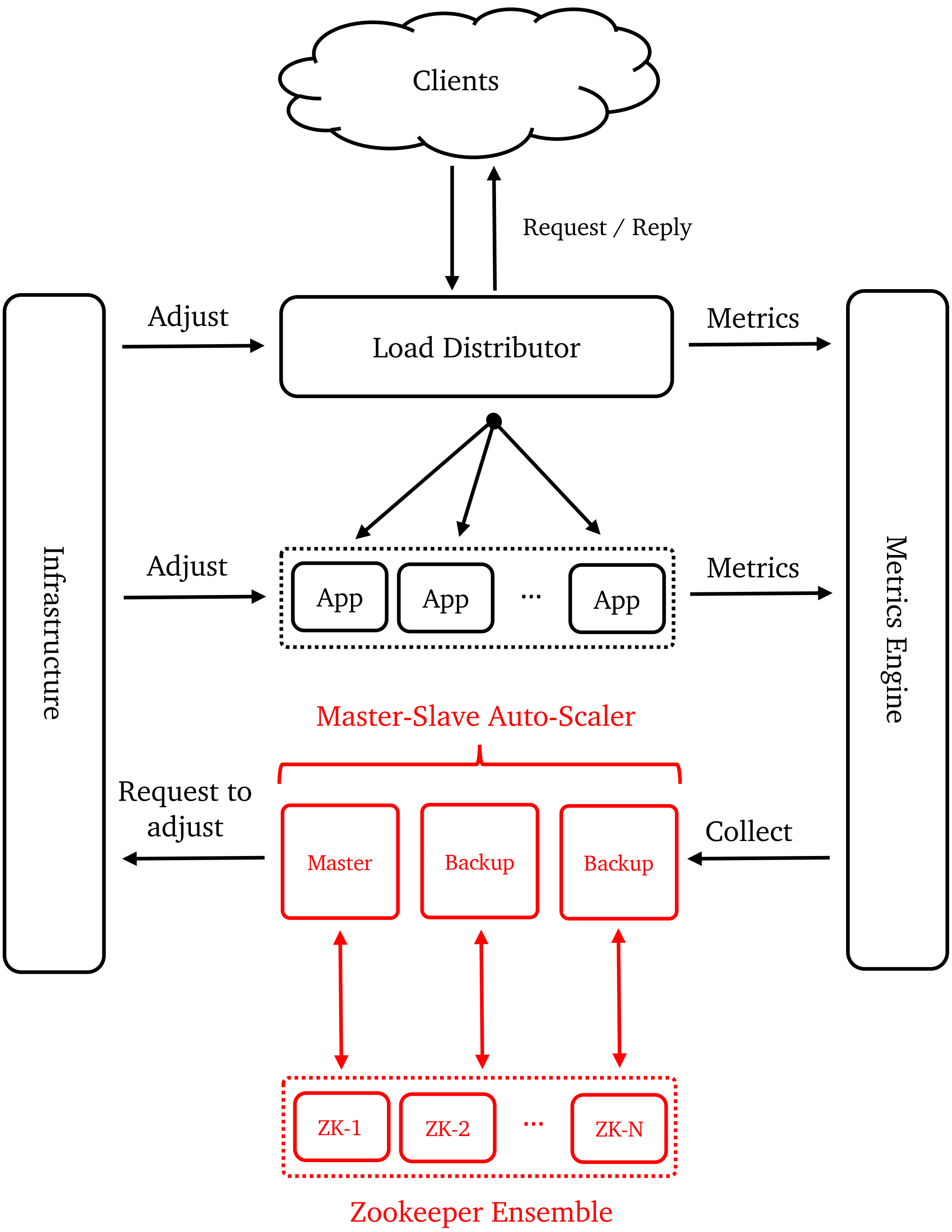


Figure 2.2: Architecture of Master-Slave or Global Controller

2.4.3.2 Distributed Without Coordination

In this architecture, typically Auto-Scaler runs alongside the application instances and each Auto-Scaler instance only *controls* the *local* application. The major difference against global controller is that, in this model each Auto-Scaler makes decision on its own without any form of coordination with other Auto-Scalers. Note that, since each Auto-Scaler instance only controls the local application independent of other application instances, it might lead to sub-optimal decisions because local controllers lack global view of the cluster. Figure 2.3 illustrates this architecture.

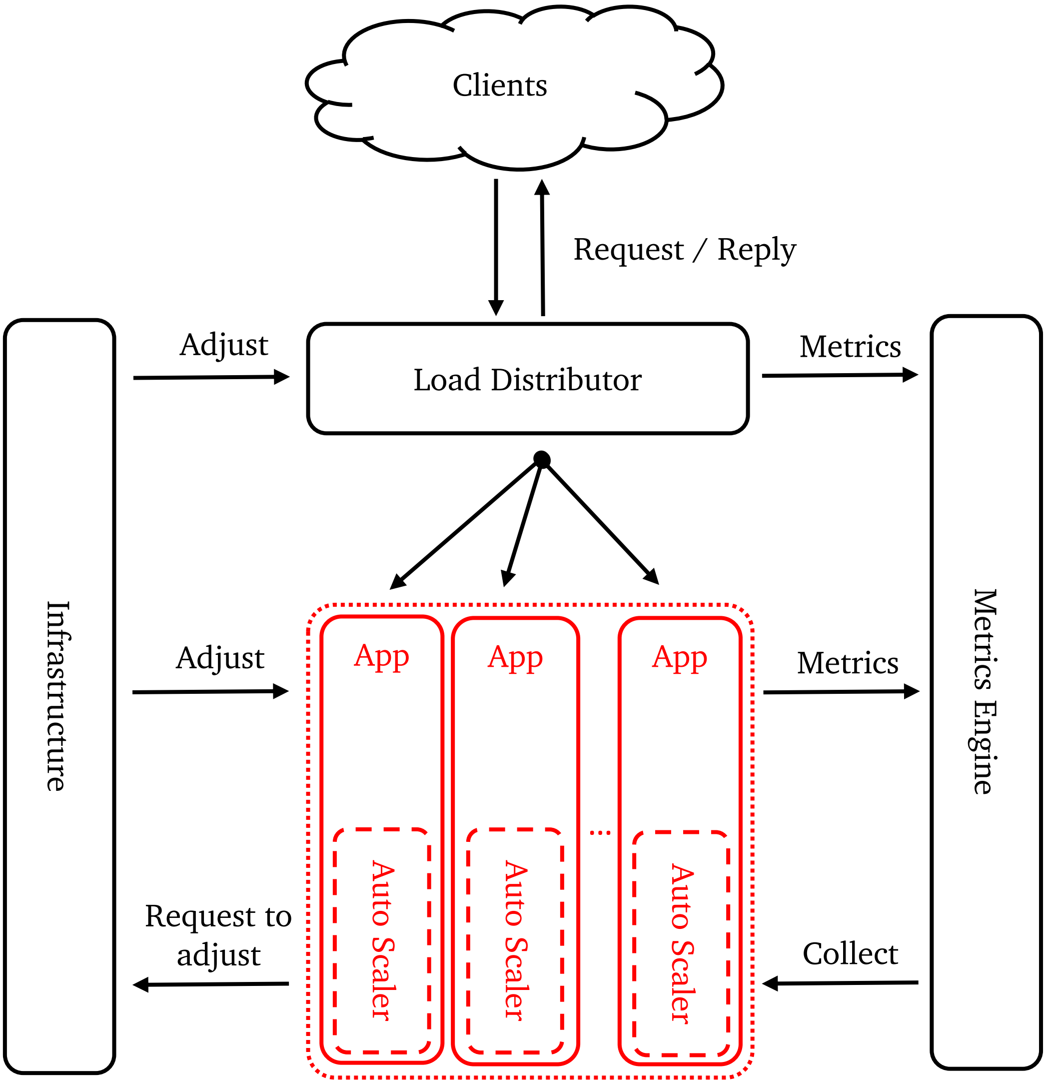


Figure 2.3: Architecture of Distributed Non-Coordinated Controller

2.4.3.3 Distributed Coordinated

This architecture is similar to previous model. However the difference is that, Auto-Scaler components coordinate with each other over a communication channel to gather more information. The amount of communicated information heavily depends on underlying algorithm. In some scenarios only neighbor nodes are contacted. In other cases, all nodes might communicate with each other in order to achieve consensus. Figure 2.4 depicts this architecture.

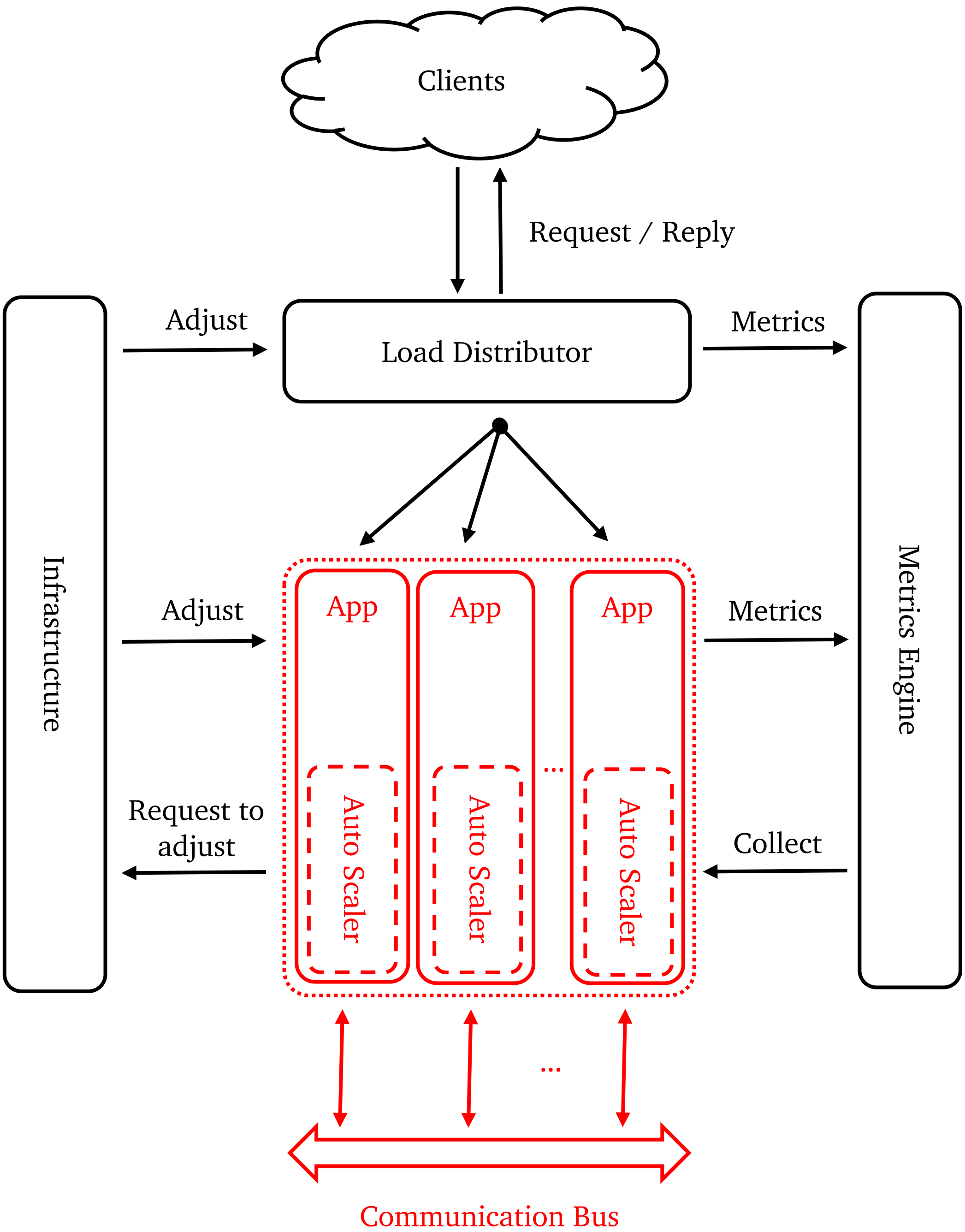


Figure 2.4: Architecture of Distributed Coordinated Controller

2.4.3.4 Hierarchical Controllers

This architecture is the mixture of previous models. It's a combination of *Global Controller* and *Distributed Coordinated* model. Local Auto-Scalers communicate with a global controller in order to perform necessary actions. Local Auto-Scalers might also communicate with each other. Additionally, one or more backup controller might also accompany global controller. Furthermore, there might be no limit on who actually performs the actions. Each Auto-Scaler – whether it is operating in local or global model – might perform actions independent of the others. Figure 2.5 illustrates this architecture.

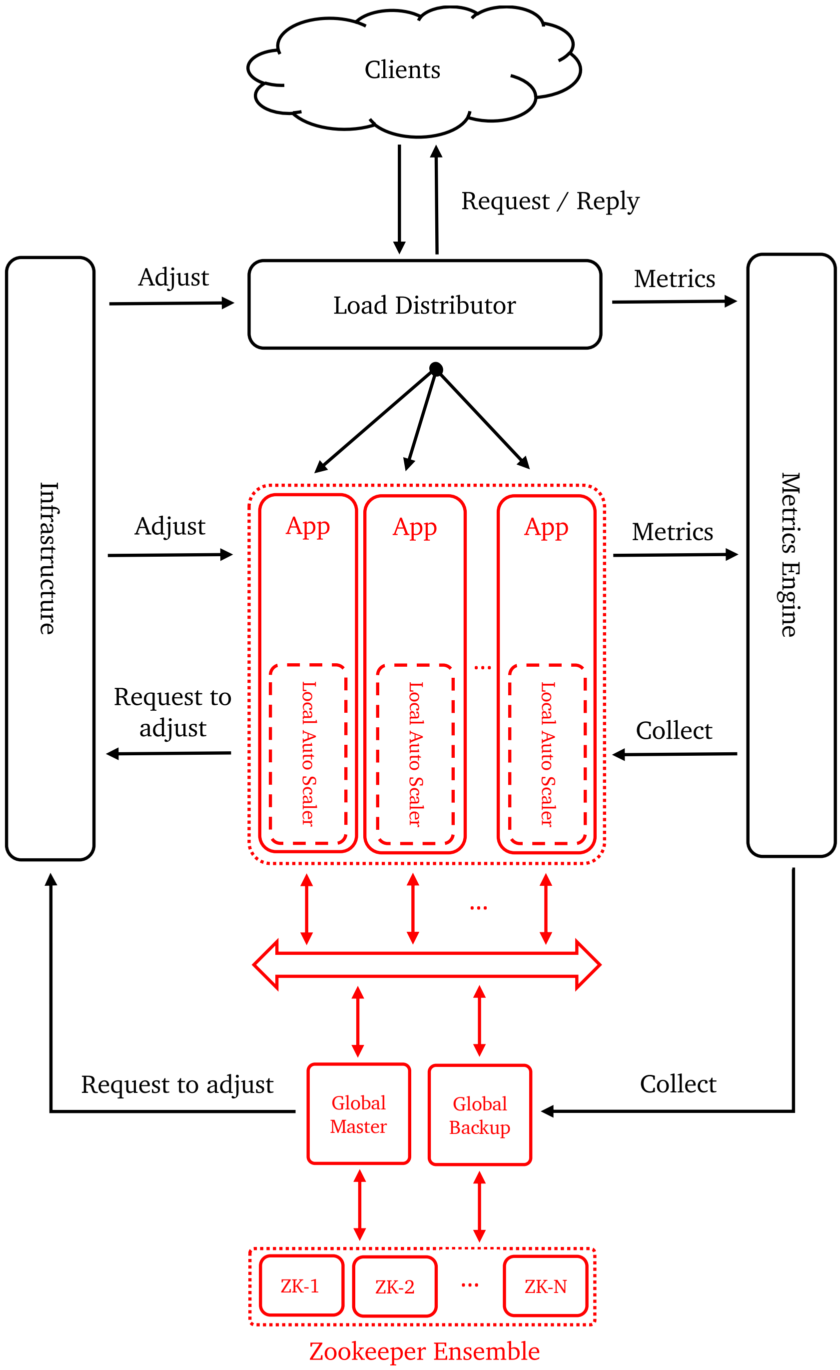


Figure 2.5: Architecture of Hierarchical Controllers

2.4.4 Algorithm Family

From algorithmic point of view, Auto-Scalers can be classified into 5 different categories.

- **Threshold-Based Policies**
- **Time-Series Analysis**
- **Reinforcement Learning**
- **Queuing Theory**
- **Control Theory**

In this section, each category is explained to some extent.

2.4.4.1 Threshold-Based Policies

According to Section 2.4.2, threshold-based approaches follow a purely reactive approach. It lacks any form of future workload prediction. Threshold-based techniques usually involve a set of constraints which monitors performance data gathered from Metrics Engine. In its naive form, each rule defines an *upper* and/or *lower* bound plus two time periods that define how long that specific metric was above the upper threshold or bellow the lower threshold.

Algorithm 2 better describes this approach in a pseudocode. Refer to Section 6 to review list of proposed solutions.

Algorithm 2: General Work-Flow of Threshold-Based Techniques

```
// upper threshold
UpperThreshold ← LoadFromConfig()
// time period that performance metric was above upper threshold
UpperPeriod ← LoadFromConfig()
// lower threshold
LowerThreshold ← LoadFromConfig()
// time period that performance metric was lower than lower threshold
LowerPeriod ← LoadFromConfig()
// number of resources to acquire in each round
NumberOfResourcesToAcquire ← LoadFromConfig()
// number of resources to release in each round
NumberOfResourcesToRelease ← LoadFromConfig()
repeat
    currentValue ← GetCurrentMetricValue()
    if currentValue > UpperThreshold for UpperPeriod seconds then
        // acquire resource
        AcquireResource(NumberOfResourcesToAcquire)
        DoNothingDuringGracePeriod()
    end
    if currentValue < LowerThreshold for LowerPeriod seconds then
        // release resource
        ReleaseResource(NumberOfResourcesToRelease)
        DoNothingDuringGracePeriod()
    end
until Auto-Scaler is running
```

2.4.4.2 Time-Series Analysis

In contrast to threshold-based techniques, time-series analysis approaches are purely proactive or predictive approaches. A *time-series* is a collection of data points sampled and ordered iteratively at uniform time intervals [48]. Time-series analysis typically requires to store a history of data points. Hence, storage and computation requirements are more intensive than threshold-based approaches. Refer to Section 6 for evaluation of prior works in time-series. According to theory of time-series analysis [11] [33], time-series can be decomposed into multiple sub components.

- **Season** The season component captures recurring patterns that are composed of one or more frequencies, e.g. daily, weekly or monthly patterns. These dominant frequencies can be determined by using a *Fast Fourier Transformation* (FFT) or by *Auto-Correlation* algorithms.
- **Trend** The trend component can be described by a *monotonically* increasing or decreasing function that can be approximated using common regression techniques.
- **Noise** The noise component is unpredictable outliers of various frequencies with different amplitudes. The noise can be absorbed to some extent by applying smoothing techniques like *Weighted Moving Averages* (WMA), by using *Lower Sampling Frequency* or by a *Low-Pass Filter* that eliminates high frequencies.

A time-series has a couple of important characteristics that reveals more information about the values in it. The following represents some of these characteristics.

- **Burstiness** The burstiness defines the impact of fluctuations within the time series and usually calculated by the ratio of the *Maximum Observed Value* to the *Median* within a sliding window.
- **Length** The length of the time series mainly influences the accuracy of approximations for the components mentioned above. It can be modeled as *static* or *sliding* window of time.
- **Relative Monotonicity** The relative monotonicity is the maximum number of consecutive monotonic values either *increasing* or *decreasing* within a window and indirectly determines the influence of the noise and seasonal components.
- **Mean, Median, Standard Deviation and Quartiles** These values are important indicators for the distribution of values – how values are spread – in time series.
- **Frequency** The frequency of a time series represents the number of values that form a period of interest. This value is an important input as it defines the starting point to find seasonal patterns.

2.4.4.3 Reinforcement Learning

Reinforcement Learning is a technique that relies on direct experience from environment. The decision maker – known as *agent* – tries to learn the best possible action for each *state* of the environment. The final goal is to maximize the returned *reward*. In the context of Auto-Scaling problem, the agent is the Auto-Scaler component that gets current system state (any group of performance metrics) and tries to minimize/maximize some aspect of the application (response times, throughput, etc.) by performing Scale-In, Scale-Out or No-Action.

In order to formally define Reinforcement Learning environments, some basic terminology [57] shall be defined. At each time step t, where $t = 0, 1, 2, \dots$ is a sequence of discrete time steps, the agent receives a representation of the environment's state $s_t \in S$, where S is the set of possible states, and based on that, selects an action, $a_t \in A(s_t)$, where $A(s_t)$ is the set of possible available actions in state s_t. One time step later, as a consequence of performing action a_t, the agent receives a numerical reward r_{t+1} and lands in a new state s_{t+1}. At each time step, the agent implements a *mapping*

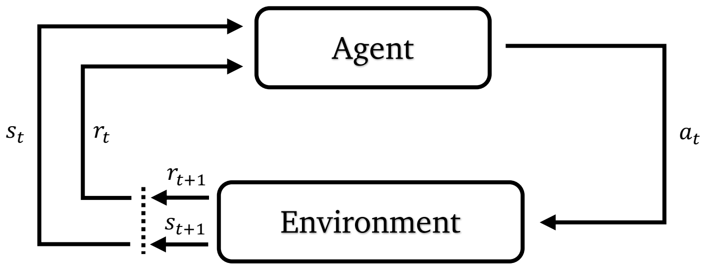


Figure 2.6: Reinforcement Learning Agent[1]

from states to probabilities of selecting each possible action. This mapping is called the agent's *policy* and is denoted as π_t, where $\pi_t(s,a)$ is the probability of taking action a at state s at time t. Figure 2.6 depicts this procedure.

In Reinforcement Learning environments future states can be determined only with the current state, regardless of the past history. This is known as *Markov Property*, which formally states that the probability of a moving to a new state s_{t+1} only depends on the current state s_t and action a. In other words, it is *independent* of all previous states and actions [57] [48]. Equation 2.1 defines this property.

$$P(s_{t+1}=s',r_{t+1}=r|s_t,a_t)=P(s_{t+1}=s',r_{t+1}=r|s_t,a_t,r_t,s_{t-1},a_{t-1},r_{t-1},\ldots,s_1,a_1,r_1,s_0,a_0) \tag{2.1}$$

A stochastic process that satisfies the Markov property is called a *Markov Decision Process* (MDP). It is typically represented as a 5-tuple consisting of *states*, *actions*, *transitions probabilities*, *reward* and *policy* function.

- Set S which represents the state space of the environment.
- Set A which represents the total possible actions.
- Reward function R defined as $S \times A \to R$ which represents the reward for each state-action pair. $R(s,a)$ is the reward received after executing action a at state s.
- Probability distribution T defined as $S \times A \to P(S)$ which specifies a probability distribution over the set S. $P(s'|s,a)$ specifies the probability of landing in state s' assuming that the agent is in state s and performs action a.
- Policy function π defined as $S \to A$ that maps state s to some action a. A policy function that maps state s to *best* possible action – with highest expected reward – is called *optimal* policy denoted as π^*.

An important aspect of reward function is that, reward must be accumulated with a *Discount Rate* denoted as γ. This is introduced to prevent *infinite* reward accumulation and force reward values to converge. Equation 2.2 defines reward function in combination with discount rate.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_0^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

Last but not least, under policy π, *value-state* function $V^{\pi}(s)$ defines an estimate value of expected reward for state s. Equation 2.3 defines value-state function where E_π defines the expected reward value. The value-state function is the core of most Reinforcement Learning techniques. In Section 4, two of the techniques used in this thesis will be further discussed. Furthermore, Section 6 evaluates prior work regarding Reinforcement Learning approaches.

$$V^{\pi}(s) = E_\pi(R_t|s_t=s) = E_\pi\Big(\sum_0^{\infty} \gamma^k r_{t+k+1}|s_t=s\Big) \tag{2.3}$$

[1] The figure has been taken from Sutton and Barto [57]

2.4.4.4 Queuing Theory

Queuing theory has been thoroughly used in the context of packet processing to estimate the average wait time until a packet could be routed. The same principle can be applied in the context of Auto-Scaling problem. Clients send requests at a *Mean Arrival Rate* denoted as λ. Requests are enqueued until they are processed with at a *Mean Service Rate* denoted as μ. Figure 2.7 portrays two architecture of queuing systems. Furthermore, multi-tiered applications are modeled as multiple queues attached *serially* or in *parallel*, depending on the architecture of the application.

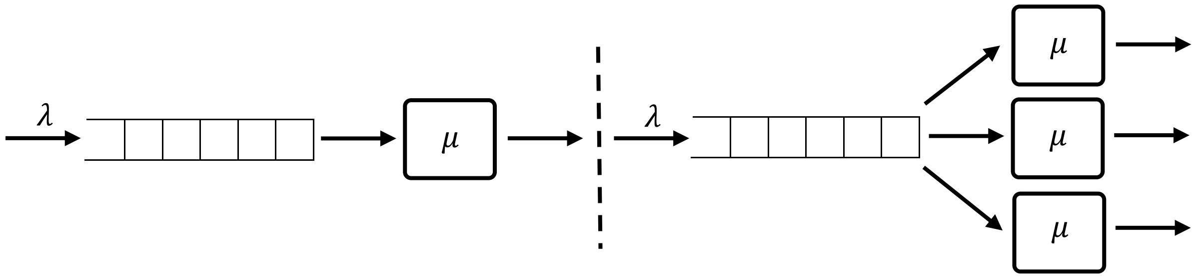


Figure 2.7: Queuing Theory with Single Processor (Left), with Multiple Processors (Right)[1]

Queuing systems are modeled with *Kendall*'s notation [41]. A queue is modeled as $A/B/C/K/N/D$. Each variable is defined and further explained in the following.

- **A** Request inter-arrival time distribution.
- **B** Service time distribution.
- **C** Number of request processors.
- **K** Maximum length of the queue. This is an optional parameter and is set to unlimited if not specified.
- **N** Calling population. This is an optional parameter and is set to unlimited if not specified. This parameter controls the population of requests entering the system. In case of an *open queuing* system, this value is unlimited. While in a *closed queuing* system the population of customers is a finite value.
- **D** Priority order which defines in which order requests will be processed by request processors. This parameter is optional and is set to *First Come First Served* (FCFS) if not specified.

Queuing theory is mostly capable of modeling stationary systems with constant parameters defined above. In case of Auto-Scaling problems, these values should be periodically recomputed, since request inter-arrival distribution, service time and number of request processors are changing constantly. These parameters can be re-calculated mostly based on two approaches. First, by applying *analytic* techniques – suitable for simple queuing systems. Second, by applying *Discrete Event Simulation* (DES) techniques – suitable for more complex models.

2.4.4.5 Control Theory

Control Theory is categorized as mixture of reactive and proactive systems and decomposed intro three major categories.

- **Open Loop Systems** These systems do not consider any feedback returned by underlying systems. They only consider current state of the system in order to make decision. The output of the system is not considered to check whether it is in stable and desired state.

[1] The figure has been taken from Lorido-Botran, Miguel-Alonso, and Lozano [48]

- **Feedback Systems** These system monitor the output of the system in order to check whether it is complying to defined business conditions. These systems are mainly reactive systems. Figure 2.8 depicts these controllers.

- **Feed-Forward System** These systems pro-actively try to predict the system's future state and behavior.

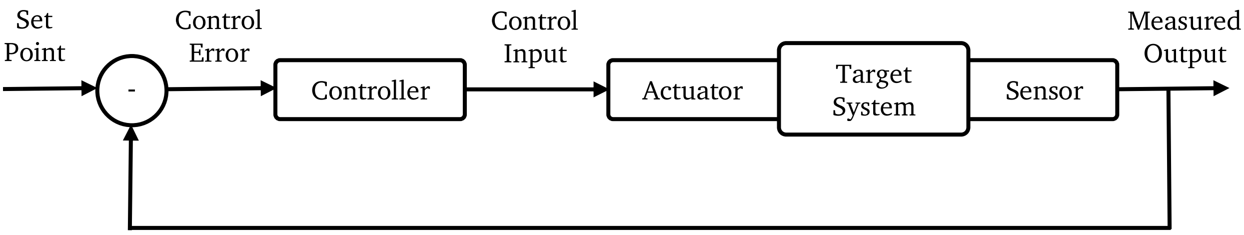


Figure 2.8: Feedback Control System[1]

Feedback driven control systems are further divided into multiple categories [51].

- **Fixed Gain Controllers** The most simple type of feedback control systems. However, the configuration parameters of the controller are fixed during system runtime.

- **Adaptive Controller** These controllers allow to change some of the configuration parameters during runtime which makes them an applicable option for *slowly-changing* environments, but not for suitable for *bursty* workloads.

- **Reconfiguring Controller** In contrast to Adaptive Controllers in which only configuration parameters can be changed at runtime – controller itself does not change – in this model the controller itself can also be changed at runtime. With this enhancement, Reconfiguring Controllers are more resilient to bursty workloads.

- **Model Predictive Controller** These controllers mix some *predicative* features into feedback controllers which make them even more resilient to unanticipated workloads.

2.5 Conclusion

In this chapter, different architectural aspects and major design considerations of Auto-Scalers have been discussed. As mentioned, it is a multi-dimensional space of features and capabilities. Many of the current proposals do not map to one category, but combine divergent set of features. Section 6 compares and evaluates prior research on different systems.

[1] The figure has been taken from Patikirikorala and Colman [51]

3 Apache Spark and Spark Streaming

Apache Spark [7] is a general purpose data processing framework that supports wide range of applications from *batch* processing, *stream* processing, *graph* processing to *machine learning*, etc. In this chapter the architecture of Apache Spark and Spark Streaming – as one of its sub projects – will be explored and discussed. Throughout the chapter, minor examples will also be presented to further simplify the concepts. This chapter is organized as follows. Section 3.1 explains the basic concepts of Apache Spark. Then, Section 3.2 explains *Resilient Distributed Datasets* (RDDs) which is the most integral component of the Apache Spark. Section 3.3 explains *Discretized Streams* (DStreams) as the fundamental building block for data stream processing applications. Finally, Section 3.4 concludes this chapter.

3.1 Basic Concepts

Map-Reduce [18] and its derivative projects have been widely used by *data-oriented* applications to process and crunch huge datasets over last the decade. In traditional Map-Reduce environments, developers typically create *acyclic* data flow graphs to process input data. However, as confirmed by Zaharia et al. [66], there are two categories of applications that are not well suited for this architecture.

- **Iterative Jobs** Many machine learning applications process the same input *iteratively*. In traditional model, each iteration should be defined as a separate MapReduce job. After an iteration is finished, the produced output is saved to a distributed file system and then next iteration is started by feeding in the output of last iteration. While this is feasible, but for each job, input data has to be loaded from disk which leads to serious performance issues.

- **Interactive Analysis** MapReduce derivative projects like Hive [4] and Pig [6] have been extensively used to run SQL queries on top of massive datasets. Whenever a user submits different queries over the same dataset, the ideal solution would be to load all datasets into memory once and then execute different queries on top of it. However, with traditional model of execution each query shall be defined as a separate job which reads input data from disk.

Apache Spark has been designed from ground up to resolve these issues. It provides a large stack of tools to facilitate processing large datasets. Figure 3.1 depicts tools provided by Spark.

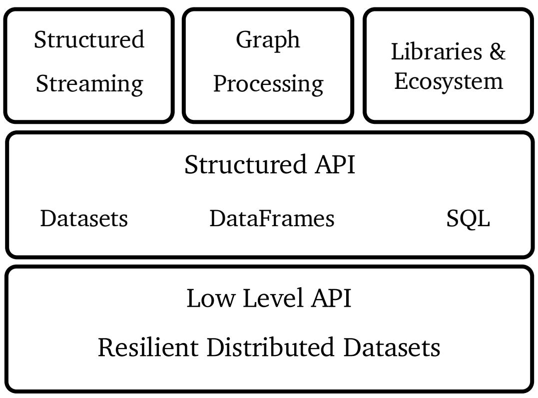


Figure 3.1: Apache Spark Stack

3.1.1 Spark Runtime Architecture

Any Spark application consists of multiple components at runtime. The following lists the relevant components from a high level point of view. Figure 3.2 depicts coarse grained architecture of a Spark application and Table 3.1 summarizes these components.

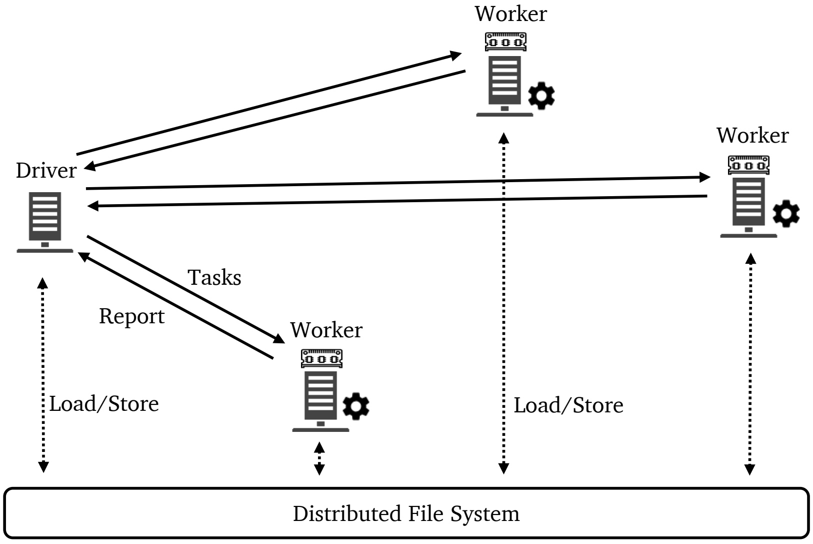


Figure 3.2: Architecture of Spark Runtime[1]

Driver Process It is the core component of every Spark application. It is a single process and runs on one of the nodes in the cluster. It maintains all the critical information such as user's program, location of input and output files plus data flow of the application. The driver process should run during the runtime the of the application. In case driver process fails, the whole application is considered as *dead*.

Distributed File System It provides a shared file system accessible by any node in the cluster. There is no limitation on type of the file system but typically *Hadoop Distributed File System* (HDFS) [3] is used.

Worker Processes A collection of worker processes known as *executors* that run on cluster nodes. Executors run user-defined code. During the runtime of the application each worker consume any number of input records, processes it based on user-defined code and emits any number of output records. Similar to driver process, liveness of the worker processes shall by monitored. However, the role of worker processes are not as critical as driver process, even though they may fail for any reason. Workers have access to load/store data files from/to distributed file system or local memory of the machine. During lifetime of the application, status of the workers will be reported to driver process.

User-Defined Code Provided by user and caries the main logic of the application.

Component	Description
Driver	Maintains all relevant information to process input data and produce output.
Distributed File System	Provides shared file system accessible by all nodes in cluster.
Worker Process	Gets necessary information from driver and executes user defined code.
User-Defined Code	Caries the main application logic.

Table 3.1: Summary of Spark Runtime Components

[1] The figure has been partially taken from Zaharia et al. [65]

3.1.2 Spark Cluster Manager

Section 3.1.1 described the coarse grained architecture of Spark. However, from a more fine grained point of view, there is missing component known as *Cluster Manager*. Cluster Manager controls the *assignment* of executors to cluster nodes. It monitors liveness of executors during lifetime of the Spark application. *Spark Session* is the entry point for all Spark applications and has the responsibility to communicate with Cluster Manager to distribute tasks and collect task progress reports from executors. It also distributes user defined code on Worker nodes. Figure 3.3 illustrates the role of Cluster Manager in Spark.

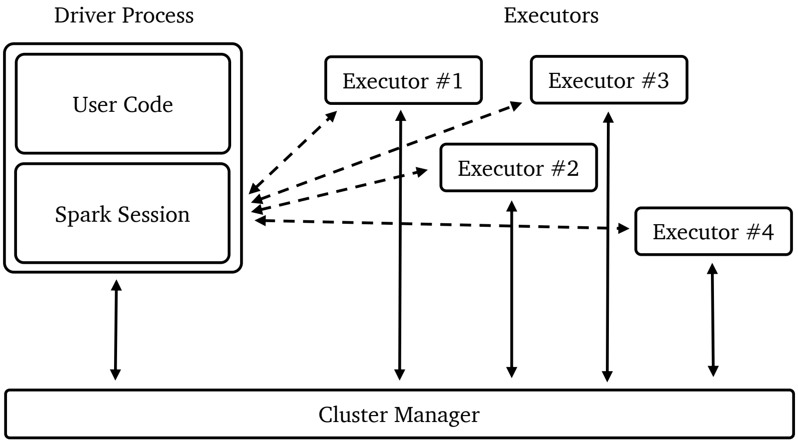


Figure 3.3: Architecture of Spark Cluster Manager[1]

Note that, the term *executor* is a conceptual term and implementation details may differ among different resource managers. Spark supports multiple Cluster Manager implementations. The following describes the most prominent implementations. It shall be noted that this list is a growing over time due to Spark's pluggable resource manager architecture.

Apache YARN [60] It is referred as *Yet Another Resource Negotiator* and is the default resource manager for modern Hadoop workloads. It has a master-slave architecture and consists of a single global *Resource Manager* – most probably accompanied by a backup as well – and several *Node Managers* that run on each worker node. The concept of executor is implemented as *containers* in YARN. Each container can hold a specific amount of node's processing power (CPU, RAM, Network, etc.). In YARN, any application is modeled with two components. First, *Application Master* that maintains the necessary information to run the application – Spark driver process in this case and second, several *Workers* that run the user defined code. Both Application Master and Workers are executed in the context of YARN containers. Fault-tolerance is provided at multiple levels. First, several backup nodes monitor the master resource manager via Zookeeper. In case master resource manager fails, one of the backup masters takes over the responsibility. Second, Applications Master reports its status to the global resource manager. In case, application master fails, the global resource manager launches a new Application Master on another node. Third, Workers provide different progress reports to Application Master and Node Managers. In case any Worker container fails, a new container will be allocated and the old one will be killed by the local Node Manager.

Apache Mesos [35] It is another dominant cluster scheduler for Spark. It is a master-slave resource manager. A global resource manager – known as *Mesos Master* – has cluster level view. On each node runs a single *Mesos Slave* process. Mesos has a pluggable architecture for different class of application schedulers. That is, a single cluster can run a mixture of Spark, MPI, etc. jobs with different priority for each application type. Each *Framework Scheduler* handles corresponding jobs. For example, Spark scheduler, maintains multiple driver processes or MPI scheduler maintains multiple MPI

[1] The figure has been partially taken from Chambers and Zaharia [15]

applications. Free resources on each Mesos Slave are represented as empty *slots* – very much like containers in YARN – and are allocated to one of the currently running jobs. Fault-tolerance is provided by a similar hierarchical approach like YARN. Mesos Master is monitored by several backups through Zookeeper. Mesos Slaves as well as Framework Schedulers are in turn monitored by Mesos Master. Each job is further monitored by corresponding Framework Scheduler. And finally, executors are monitored by Mesos Slaves.

Spark Standalone It is the default resource manager for Spark and will be used throughout this thesis. It also follows the master-slave model. *Spark Master* is the global cluster level resource manager. On each cluster node runs a *Spark Slave* process. Spark Slaves are responsible to run and monitor worker executors. It is possible to set default number of CPU cores and available memory for each executor through configuration, either *statically* which is applied to all jobs or at *submission* time for a specific job. Fault-tolerance is achieved by several measures. Figure 3.4 depicts fault-tolerance architecture of Spark. Similar to YARN and Mesos, a multi-level failure detection approach is exploited. Multiple backup nodes monitor the Spark Master via Zookeeper and in case of master failure, one of backups will take over (Case 1). Spark Slaves and Driver processes are monitored by Spark Master (Case 2). Progress of each executor – whether the assigned task is done or not – is monitored by its corresponding Driver process (Case 3). Initial input and final output of Spark jobs are stored in a distributed file system like HDFS (Case 4). In case any executor fails to store its final output in DFS, it will be relaunched by Spark Master on another node. Intermediate results are stored locally without any fault-tolerance in mind. However, *checkpoints* which is stored on distributed file system can be used in this case to recover from executor failures (Case 4). Refer to Section 3.2 for more information on checkpointing process. Liveness of each executor – whether it is running or not – is monitored by corresponding local Spark Slave process (Case 5)

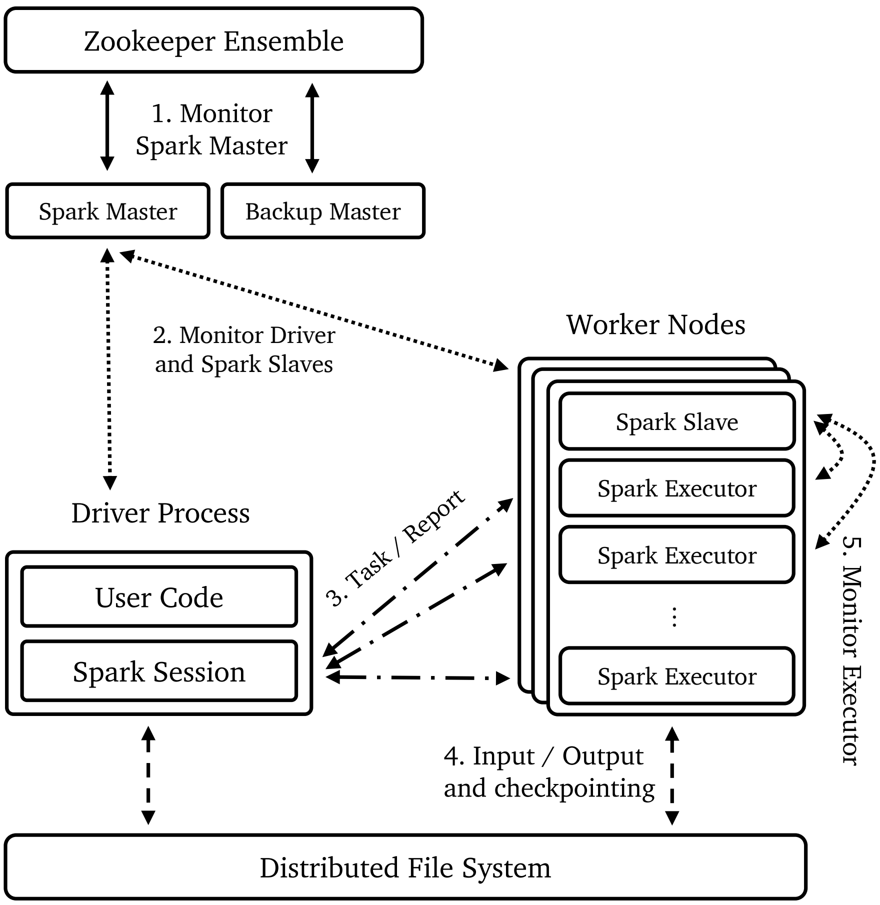


Figure 3.4: Architecture of Spark Fault-Tolerance[1]

[1] The figure has been partially taken from Chambers and Zaharia [15]

Resilient Distributed Dataset (RDD) [65] is the basic building block of Spark's data processing pipeline. It is inspired by the same concepts behind *shared memory* and enables *iterative algorithms* and *interactive analytics* to perform common operations directly on memory. RDDs are fault-tolerant and parallel data structures to let intermediate results of a Spark application to be stored in memory during multiple iterations. Existing distributed shared memory abstractions for cluster computing provide *fine-grained* interface – typically key/value – to modify state. Unfortunately, with this approach the only way to provide fault-tolerance is the *log shipping* approach that is already offered by distributed database systems.

Whereas, RDDs offer *coarse-grained* interface based on *immutable transformations*. Each transformation (`map`, `filter`, `foreach`, `groupByKey`, etc.) applies same application logic to all records that it contains. A dataset's *lineage graph* is the *chain* – *sequence* – of multiple transformations that produced the aforementioned dataset. With this approach only the transformation itself needs to be logged and shipped to other nodes to provide fault-tolerance. Since RDDs are already immutable, recomputing the lineage on other node is a trivial task. The RDD abstraction is useful in data intensive processing applications, since same operation is applied to many records. Table 3.2 summarizes the differences between RRDs and common distributed shared memory technologies.

Dimension	RDD	Distributed Shared Memory
Reads	Per record or per input file	Per key/value
Writes	Coarse grained	Per key/value
Consistency	Easy (Immutable RDDs)	Application dependent
Fault-Tolerance	Lineage recomputation	Application dependent
Straggler Mitigation	Parallel backup tasks	Application dependent
Operator Placement	Based on data locality	Application dependent
Behavior if not enough memory is available	Controlled serialization to disk	OS managed swapping

Table 3.2: High-Level Comparison of RDDs and Distributed Shared Memory Systems[1]

An RDD is a *read-only*, *partitioned* collection of records [65]. RDDs can only be created through deterministic transformations from either *stable storage* or *other RRDs*.

- **Stable storage**. In this case dataset is typically stored on a shared file system like HDFS and accessible from all nodes of the cluster.
- **Other RDDs**. In this case subsequent RDDs *depend* on each other forming the lineage graph.

An RDD has enough information about how it was derived from other RDDs. This means it doesn't have to be materialized in every step of the pipeline. In other words, an RDD cannot reference another RDD that it cannot reconstruct after a failure. Additionally, users are able to control *persistency* and *partitioning* of RDDs.

- **Persistency**. Users can define which RDDs will be reused in next steps of the pipeline. They choose a storage strategy – in-memory or disk-based – to persist RDDs.
- **Partitioning**. Some transformations like `groupByKey` or `join` partition the original RDD into multiple secondary partitions. RDDs allow application developers to perform partitioning – hash, range or any custom partitioning strategy – such that relevant records are *co-partitioned* on same machine. This is particularly useful for *operator placement* optimizations.

In order to simplify the concept of lineage graph, Listing 3.1 and Figure 3.5 illustrate a simple log processing pipeline and its corresponding generated lineage graph.

[1] The table has been taken from Zaharia et al. [65]

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startsWith("ERROR"))
errors.persist()

// Case 1 -- Count total number of errors
val errorCount = errors.count()

// Case 2 -- MySQL error
val mysqlErrors = errors.filter(_.contains("MySQL"))
val mysqlErrorCount = mysqlErrors.count()

// Case 3 - Return forth field of MySQL logs that contain SLOW QUERY
val slowQueries = mysqlErrors.filter(_.contains("SLOW QUERY"))
                             .map(_.split(',')(3))
                             .collect()
```

Listing 3.1: Parsing Errors in Log Files From HDFS

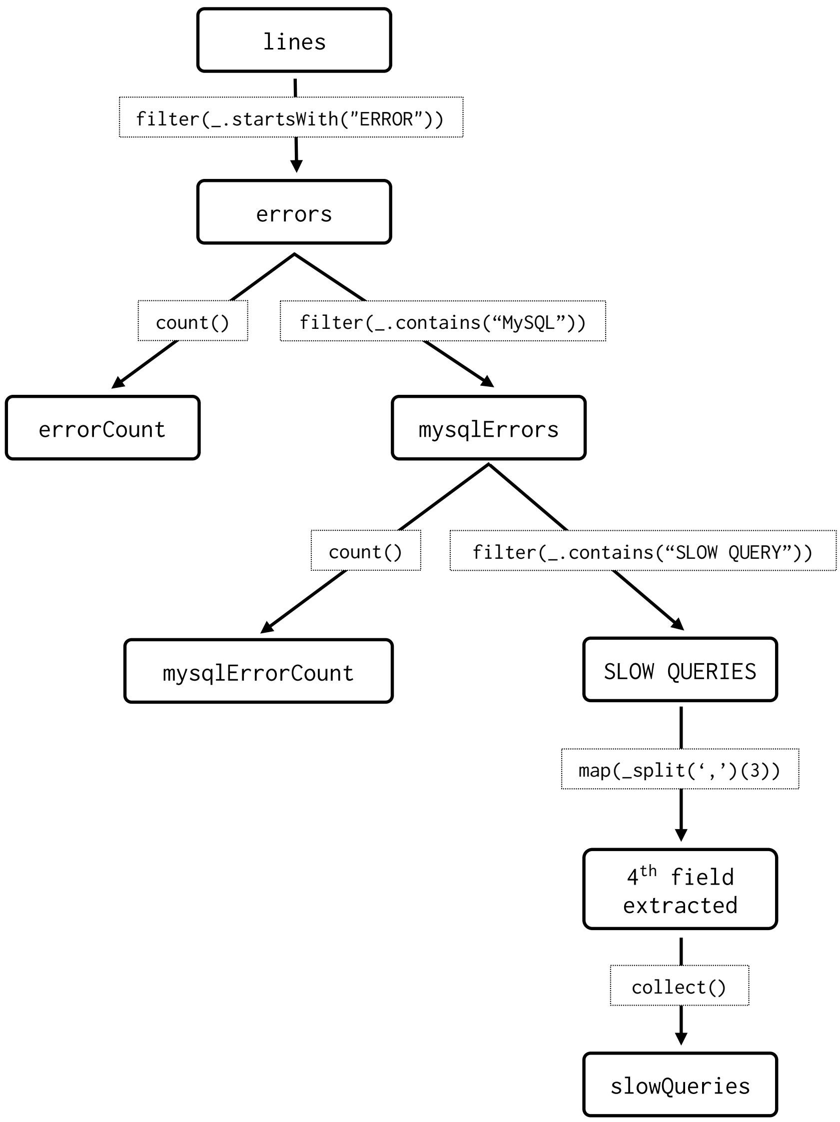


Figure 3.5: RDD Transformations and Corresponding Lineage Graph[1]

[1] The figure has been partially taken from Zaharia et al. [65]

3.2.1 RDD Transformations and Dependencies

As mentioned, RDDs are the result of *lazy* transformations derived from one another. In other words, RDDs are not materialized unless it is really needed. However, not all transformations are lazy. There are three major types of transformations.

- **Transformations without shuffling**. These are lazy transformations that doesn't cause any sort of shuffling among different nodes. That is, transformed records can still be further processed on the same node. For example, `map` and `filter` are classified in this group.

- **Transformations with shuffling**. These are also lazy transformations but they will trigger shuffling process between nodes. The behavior of the shuffling process is influenced by the implementation of *partitioners*. As an example, `groupByKey`, `reduceByKey` and `join` belong to this group.

- **Actions**. These transformations trigger RDD materialization process. That is, the actual computation of RDDs does not realize until this type of transformation is met in lineage graph. For example, `count`, `collect` and `save` are classified in this group.

In a lineage graph, RDDs are derived from one another. This makes each RDD depend on one or several RDDs. There are two types of dependencies among RDDs.

- **Narrow**. When one partition of the parent RDD is used by at most one partition of the child RDD, then the dependency is considered as narrow. For example `map`, `filter`, `union` and joins with *co-partitioned* inputs create narrow dependencies. Narrow dependencies allow to process RDDs without triggering shuffling process – a phenomenon known as *pipelined execution*. Additionally, failure recovery is easier in case of narrow dependencies because only lost parent RDDs need to be recomputed. Figure 3.6 depicts narrow dependencies.

- **Wide**. When one partition of the parent RDD is used by more than one partition of the child RDD, then the dependency is considered as wide. For example, `groupByKey` and joins with inputs *not co-partitioned* create wide dependencies. Wide dependencies trigger the shuffling process. In contrast to narrow dependencies, recovering from node failures are less efficient since a failed node might cause loss of some partitions from all ancestors which requires full recomputation [65]. Figure 3.7 depicts wide dependencies.

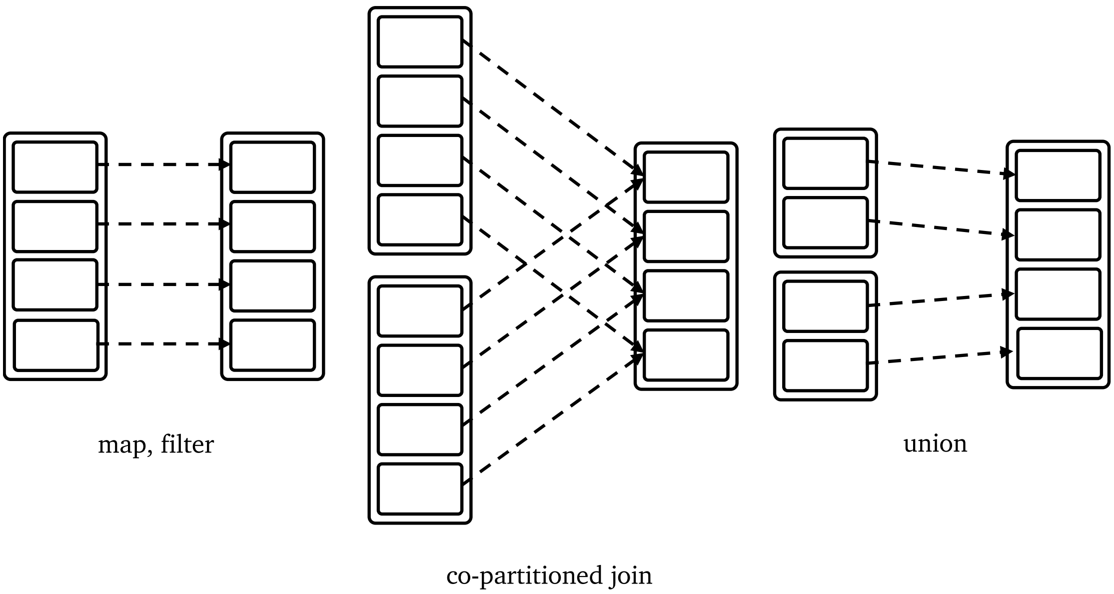


Figure 3.6: RDD Narrow Dependencies[1]

[1] The figure has been partially taken from Zaharia et al. [65]

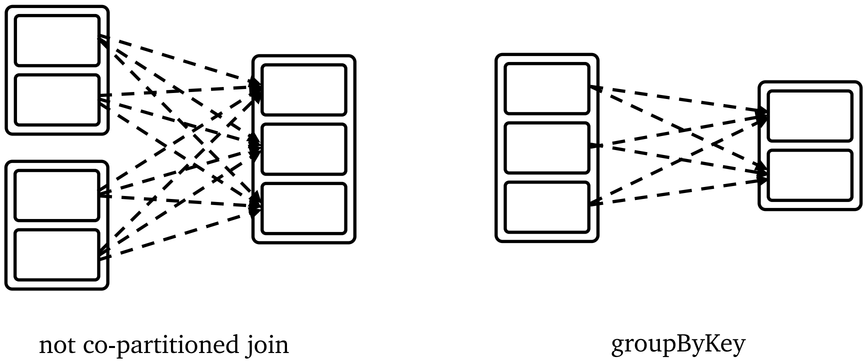


Figure 3.7: RDD Wide Dependencies[1]

3.2.2 Fault-Tolerance with Checkpointing

The idea of the lineage graph and recomputable transformations significantly simplifies fault-recovery. However, in long chains recomputing failed RDDs can be a very time/process intensive operation. Hence, in some cases it is useful to checkpoint some RDDs to stable storage – like shared file system. In general checkpointing is useful for applications with deep lineage graphs containing wide dependencies, since a node failure in parent RDDs leads to full recomputation in child RDDs. If the lineage graph is small or most dependencies are narrow, checkpointing tends to be less useful.

In order to provide full control to application developers, Spark exposes checkpointing behavior though its API – via `REPLICATE` flag in `persist()` function – and leaves the decision up to developers. There are some scenarios where automatic checkpointing without intervention of developers may seem a feasible choice. Because job scheduler is completely aware of the size all RDDs and naturally it should be able to implement more effective checkpointing strategy compared to developers. However, this feature is not currently available in Spark.

With checkpointing in-place, providing fault-tolerance becomes a fairly straightforward process. Note that, RDDs are immutable set of records. This feature makes it easy to flush them in a background process, since there is no *concurrency* and *consistency* concerns involved. Noteworthy, there is an additional fault-tolerance mechanism known as *WAL-based fault-tolerance* for streaming pipelines. This topic is further discussed in Section 3.3.

3.2.3 RDDs and Spark Job Stages

Spark's Job scheduler takes RDD locations into account to take any decisions, since some partitions of RDDs are cached into memory or persisted to stable storage. When a job is submitted, Spark's job scheduler does the following.

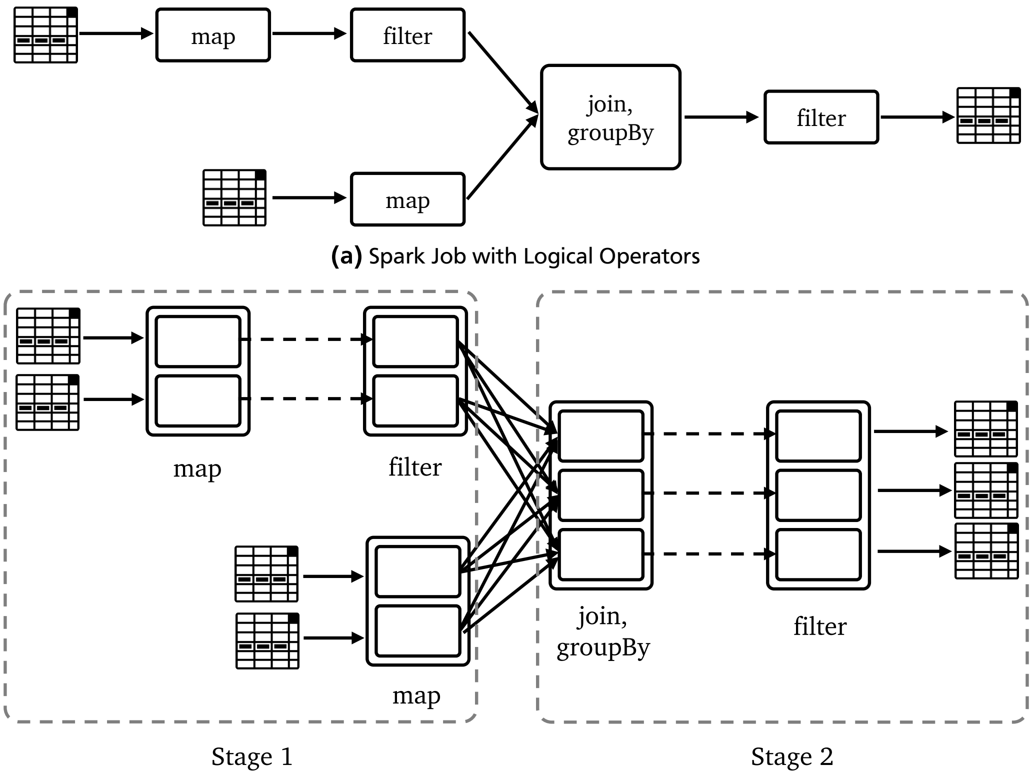
- **Calculating Lineage Graph**. In this step, Spark calculates the lineage graph of RDDs with corresponding transformations and dependencies.

- **Defining Stages**. In this step Spark defines the stages required to run the job. From developer's point of view a job is a sequence of RDDs. However, at runtime a job is modeled as *Direct Acyclic Graph* (DAG) of stages. A *stage* is basically the longest possible chain of RDD transformations with shuffling operations in the boundaries. Additionally, different partitions of RDDs in their corresponding stages will also be computed.

- **Launching Tasks**. In this step, Spark launches *tasks* which is going to be run by executors. Each task processes one or multiple partitions of RDDs. Depending on the location of RDDs, tasks will be launched on machines with data-locality awareness to reduce network transmission. In case a task fails, it can be run as long as parent stages are still available. If one or more parent stages become unavailable, missing partitions from parent stages shall be recomputed according to lineage graph.

[1] The figure has been partially taken from Zaharia et al. [65]

Figure 3.8a shows a sample job with logical operators and Figure 3.8b depicts its corresponding DAG with two stages at runtime.



(a) Spark Job with Logical Operators

(b) Spark Job Stages at Runtime

3.3 Discretized Streams

Many modern big data applications process data in real-time as they enter the processing pipeline. Online machine learning, fraud detection, spam detection, etc. are just few examples of applications that require near real time processing of data. In order to process incoming records in near real-time, new generation of data processing frameworks has been emerged which is known as *stream processing systems*. Apache Storm [56] is an example of stream processing systems. Although these systems provide stream processing pipelines, but they also come with a couple of deficiencies. First, Section 3.3.1 explores the architecture of traditional stream processing systems and its problems. Then, Section 3.3.2 describes the new stream processing model known as *Discretized Streams* (D-Streams) [64] that is built on top of RDDs in order to resolve the problems of traditional model.

3.3.1 Continuous Operator Processing Model

Although the concept of stream processing is nothing new and there has been many proposals and implementations like Apache Storm [56] and Apache Flink [2], but most of these systems are based on a processing model known as *Continuous Operator Processing Model*. In this model streaming computations are divided into a set of *long-lived stateful* operators that process messages in a loop:

1. Get one or more messages from previous operators.
2. Apply the computation – business logic – on newly received messages. Query the internal state if required.
3. Update internal state if necessary.

4. Produce any number of messages as the result of computation. These messages will be sent to next operators down the pipeline.

5. Go back to Step 1

While continuous processing model minimizes latency, the stateful architecture of operators and nondeterminism that comes from record interleaving on the network, makes it hard to provide fault tolerance efficiently. In particular, recovering from a *failed* or slow node – *stragglers* – is challenging. The are usually two standard approaches to overcome these issues.

Replication In replication model which is borrowed from database systems, there are two copies of processing graphs. Produced messages are sent as duplicates to down stream operators. However, just replicating messages is not sufficient. A *consensus protocol* should exist in-place to ensure that both operator replicas see incoming messages in the same order that is sent by upstream operators. Even though replication is a costly operation but it recovers from failures very fast, since both replicas are processing message online in synchronized steps.

Upstream Backup The basic idea behind this model [36] is to checkpoint the internal state every once in a while to a stable shared storage. In case an operator – or node – fails, a backup operator takes over and reloads the last successfully written checkpoint. Then backup operator rebuilds the state by replying new messages and reproducing lost messages as necessary. Although this model is more efficient in terms of replication costs, but suffers from high fail-over time. Backup node should replay newly published messages from the last checkpoint and apply them to the internal state. Depending on checkpointing interval, this might be a time consuming operation.

Besides the fault-tolerance costs of this model, dealing with straggler nodes are even more challenging. Replication model provides no solution at all, since two replicas are processing messages synchronously. The only way to resolve this issue in Upstream Backup model is to fail – kill – the slow operator and let the backup operator takes over the responsibility which will undergo slow recovery process. Figure 3.9 depicts these two approaches with additional synchronization and replication messages involved.

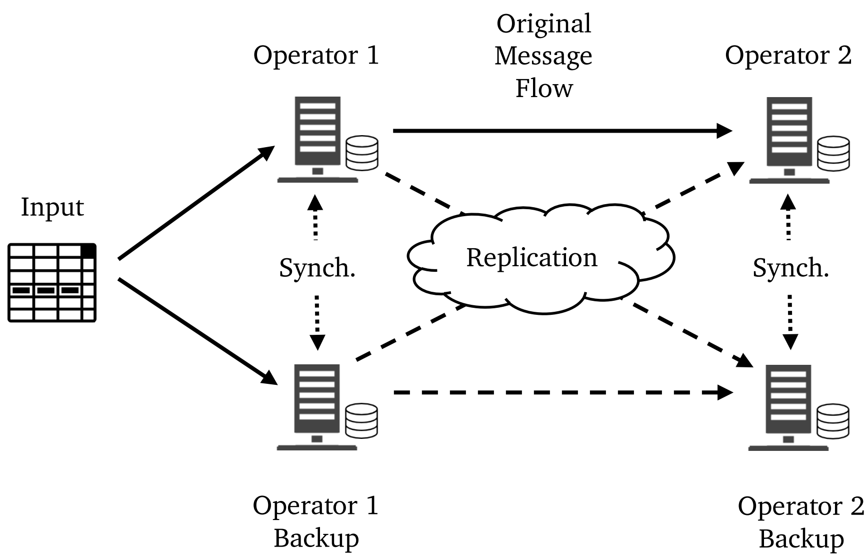


Figure 3.9: Continuous Operator Processing Model[1]

[1] The figure has been partially taken from Chambers and Zaharia [15]

3.3.2 D-Stream Processing Model

Discretized Streams (D-Streams) [64] is a novel technique to overcome the issues the Continuous Operator Processing Model. The basic idea behind D-Streams is to separate the computation from its state and keep the state as RDDs. This separation provides the following benefits. First, computation of messages becomes a set of *stateless* and *deterministic* tasks operating on RDDs. Second, RDDs already provide resiliency through replication and lineage recomputation. There is no need for further mechanisms to provide fault-tolerance. Third, it provides unified processing model for batch and stream processing workloads.

As a consequence of storing operator intermediate state as RDDs, a streaming computation can be modeled as series of *deterministic batch computations* on small intervals – known as micro-batch. Messages received in each batch interval is stored reliably across the cluster to form an input dataset for that interval. Once the time interval completes, this dataset is processed just like traditional batch processing using `map`, `filter`, `groupByKey`, etc. Formally, a D-Stream is a sequence of *immutable*, *partitioned* datasets (RDDs) that can be acted on by deterministic transformations [64]. These transformations produce new D-Streams, and may create intermediate state represented as RDDs.

In order to clarify D-Stream API, Listing 3.2 shows a simple streaming word count example. It creates a `pageViews` D-Stream by reading event stream over TCP, and groups them into 1-second batch intervals. Then, it transforms the event stream to a new D-Stream of `(URL,1)` pairs called `ones`. Finally, it performs a running count with a *stateful* `runningReduce` operator. Figure 3.10 depicts the corresponding lineage graph and how the streams are divided into batch intervals. Note that smaller rectangles illustrates RDD partitions.

```
val pageViews = readStream("tcp://...", "1s")
val ones      = pageViews.map(event => (event.url, 1))
val counts    = ones.runningReduce((a, b) => a + b)
```

Listing 3.2: Streaming Word Count using D-Stream API

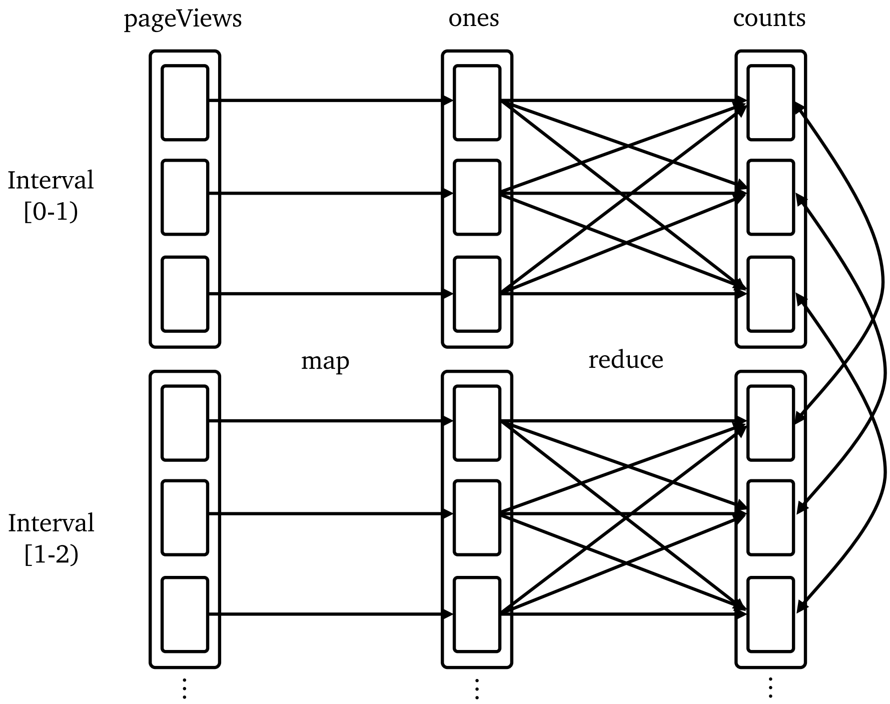


Figure 3.10: Streaming Word Count Using D-Stream API[1]

[1] The figure has been partially taken from Zaharia et al. [64]

3.3.3 Fault-Tolerant Message Consumption

Since RDDs provide the basic building block of Spark Streaming, achieving fault-tolerance in streaming applications is a straightforward process. In case of any node failure, failed RDD partitions can be reproduced easily by evaluating lineage graph and determining which transformations need to be recomputed. Furthermore, for stateful transformations, Spark already provides checkpointing. However, there is one case that needs more attention.

Naturally, in streaming applications, live stream of records are coming from some data sources. For example, in log processing application it may originate from web servers or in live monitoring applications it comes from sensors. If one of the RDD transformations crashes during message consumption, two questions arise:

- Are the messages that were consumed before crashing, be redelivered after relaunching failed transformations?
- In case lost messages are redelivered, will they be delivered with the same order as consumed before crashing?

The answer to these two questions leads us to *three* types of data sources.

Reliable Ordered With this type of data sources, the data source provides ordering and reliable message consumption capabilities. Messages will be kept in data source until its reliably confirmed – ACKed – by consumer. In case of consumer crash, messages will be delivered with the same order as before. Many of publish-subscribe message brokers such as Apache Kafka [5] provide this feature.

Reliable Unordered With this type of data sources, the data source provides reliable message store but redelivery is not guaranteed to be in the same order as before consumer crash. For cases where ordered message consumption is crucial for business logic of the application, Spark Streaming provides *WAL-based* consumers. With WAL-based consumers, each consumer appends the received message to a log file before starting to consume the message. These log files are stored in a distributed file system like HDFS. In case of consumer failure, another consumer re-opens the log file and re-processes the messages with same order as before.

Unreliable This type of data source does not provide any form of reliability, ordering and fault-tolerance. Hence, WAL-based message consumption shall be enabled by application developer.

3.4 Conclusion

In this chapter different aspects and features of Spark has been explored. Built on top of unique features of Resilient Distributed Datasets (RDDs), Spark offers a unified approach for batch and stream processing. As explained during the chapter, deficiencies and problems of traditional systems have been addressed to some extent by Spark.

4 Design and Implementation Detail

In this chapter design and project structure of implemented techniques will be introduced and discussed. Two techniques have been selected to be implemented. This chapter is organized as follows. Section 4.1 explains selected techniques and the reasoning behind it. In this section, selected techniques are explained at theoretical level. Section 4.2 describes the structure of the implementation for Spark Streaming in detail. Section 4.3 describes the configuration parameters required to launch the implementation. Finally, Section 4.4 concludes this chapter.

4.1 Selection of Implemented Auto-Scaling Techniques

As discussed in Section 2.4.4, Auto-Scalers can be categorized into 5 groups.

Threshold-Based In this approach, one or multiple threshold values are defined [28] which specifies the behavior of the Auto-Scaler when the system load goes beyond these thresholds. However, manually defining thresholds is a tricky process [21]. Spark's default dynamic resource manager [7] and Online Parameter Optimization [30] are partially based on this approach and they will be evaluated in Section 5.

Time-Series Analysis In this approach, a history window of workload – depending on prediction accuracy – is considered to predict future workload. However, in case the workload is changing in an unpredictable way – which is not an unusual phenomena in streaming applications – then this approach becomes less useful. As confirmed by DEBS grand challenges [39] [40] [27], the general assumption for streaming applications is that the workload is unpredictable. Due to this limited applicability, no time-series technique is implemented.

Queuing Theory In this approach, the system is modeled as a queue network. Since queuing parameters – message inter-arrival and service rates – are static, they have to be re-calculated in a timely manner. One of the novel implementations is done by Lohrmann, Janacik, and Kao [45]. However, authors made two unconvincing assumptions. First, worker nodes shall be homogeneous in terms of processing power and network bandwidth. Second, there should be an effective partitioning strategy in place in order to load balance outgoing messages between stages. In reality both assumptions rarely occur. Large scale stream processing clusters are built incrementally. Depending on workload, data skew does exist and imperfect hash functions are widely used by software developers. Furthermore, as confirmed by Rajarshi, Tesauro, and Walsh [52], queuing models are very complicated to build and less adaptive to changing environments. As a consequence, no queuing theory technique is implemented.

Reinforcement Learning Reinforcement Learning has shown its capabilities to adapt ever changing environments. However, some of the proposed solutions are conflicting with the requirement of this thesis as discussed in Section 1. The following summarizes the reasoning why each proposal is accepted or rejected by this thesis.

- Herbst et al. [31] proposed a solution based *Bayesian Networks*. There are two major problems with this proposal. First, it needs sampling and offline training which is inapplicable for changing streaming workloads. Second, in case the model is complex, the training phase is long and computationally infeasible for streaming workloads. As mentioned, both issues are conflicting with thesis requirements.
- Tesauro et al. [59] proposes a hybrid approach to resolve performance issues of online training which consists of two components. First an online component based on queuing theory. Second, Reinforcement Learning component that is trained offline. This proposal models each node as a queue. However, the only way to apply this technique is to modify `spark-core` package. Thus, this solution is not implemented.

- Rao et al. [53] proposed a solution to manage virtual machine resources. However, it is also based on offline training and sampling which is obtained from a separate supervised training phase. Thus, it's not applicable for dynamic streaming workloads.
- Enda, Enda, and Jim [23] proposed a parallel architecture to Reinforcement Learning without any global controller involved. Nodes (RL agents) have two tables. Local table is trained by each node separately. Global table contains values learned by other nodes. When an agent learns anything new, it broadcasts it to other nodes. From theoretical point of view, this solution might seem feasible, since parallel learning speeds up initialization process. However, Spark has a single global controller. Applying this technique to Spark requires heavy modification to `spark-core` package.
- Heinze et al. [29] implemented Reinforcement Learning in the context of FUGU [26]. Each node, maintains its own Q-Table and imposes local policy without coordinating with other nodes. Although this architecture is not applicable for Spark, but its core idea – *Temporal Difference* [57] algorithm– is applicable. Thus, this thesis implemented this proposal by adopting it to Spark architecture. Refer to Section 4.1.1 for theoretical background and Section 4.2 for implementation detail.
- Cardellini et al. [12] proposed a two level hierarchical architecture for resource management in Apache Storm [56]. Local controller applies local policy on each node and coordinates with the global controller for confirmation of its actions. Although, this architecture seems to be a promising approach, however it has been implemented by modifying Storm's internal components. As mentioned above, this is in conflict with thesis's requirements.
- Dutreilh et al. [22] proposed a model-based Reinforcement Learning approach for resource management which is based on a global controller. In order to overcome the slow convergence of model-free learning, authors proposed to estimate environment dynamics based on collected samples at runtime. Then it switches to *Value Iteration* [57] algorithm instead of *Temporal Difference*. This approach has also been partially adopted by this thesis. Refer to Section 4.1.2 for theoretical background and Section 4.2 for implementation details.

Control Theory Techniques based on Control Theory are also promising for elastic data streaming. Because it monitors input and output of the application, it can respond to workload changes very fast and adapt the system if necessary. Thus it is perfectly capable of handling dynamic environments. The comparison between Reinforcement Learning and Control Theory techniques is left for future work.

As mentioned, two techniques – *Temporal Difference* and *Value Iteration* – will be implemented in this thesis. In next two sections, theoretical foundation of these algorithms will be laid out. For an extensive introduction of Reinforcement Learning refer to Sutton and Barto [57]. Furthermore, Section 6 discusses more techniques and the reasoning why they have not been considered by this thesis.

4.1.1 Temporal Difference

Section 2.4.4 explains the basics of Reinforcement Learning. Temporal Difference (TD) learning is one of the foundational algorithms of Reinforcement Learning. It can learn from applying experience without having prior knowledge about environment's dynamics. This property is potentially useful for data stream processing systems in which the incoming workload is changing without any particular pattern.

Before starting to dig into details, some formal notions shall to be explained. An *episode* is series of experiences that is taken by the agent. At each time step t, the agent moves from one *state* S_t to another state S_{t+1}. A *policy* π defines the action A_t that should be taken by the agent at each state. Followed by each action, the agent receives a *reward* R_{t+1} from the environment. Note that, the agent will receive reward for its corresponding action at next state S_{t+1}. Thus, an episode can be viewed as Equation 4.1.

$$\dots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, A_{t+2}, R_{t+3} \dots \tag{4.1}$$

An episode does not necessarily have a *terminal* state. In some environments – like data stream processing – an episode is a never ending sequence of experiences. Prior to taking an action, the agent has an estimate of *expected reward* V which specifies future reward if it follows the same actions returned by policy π from that specific state. Thus, it is referred as V_π which translates to expected future reward under policy π. During the episode, the agent tries to *maximize* the reward that it received from the environment. Any policy that leads to maximum possible reward is called the *optimal* policy π^* and its corresponding estimate of future reward is referred to as V^*_π.

Unlike Monte Carlo [57] methods that requires agent to wait until the end of episode, TD approaches only need to wait until the end of next time step to get a feedback from environment. Then, the agent updates corresponding estimate of previous state. Thus, the most simple TD approach can be formulated as Equation 4.2. Equation 4.2 contains two important parameters of TD.

$$V(S_t) \longleftarrow (1-\alpha)V(S_t) + \alpha\big[R_{t+1} + \gamma V(S_{t+1})\big] \tag{4.2}$$

- $\boldsymbol{\alpha}$ denotes *Learning Factor*. It specifies, how much the agent shall learn from new experiences. A higher α means learning with a faster pace. It also leads to forgetting history faster. On the other hand, a lower α leads to slower learning process which means the agent trusts its experience history more – it gives more weight to history rather than new experiences. α is defined as a number between (0,1].

- $\boldsymbol{\gamma}$ denotes *Reward Factor* or *Discount Factor*. It specifies, whether the agent shall optimize for *future* or *immediate* reward. A higher γ leads to optimizing for future reward, whereas a lower γ leads to optimizing for immediate reward. It is defined as a number between (0,1). As defined by Equation 4.3, reward is cumulating at each time step which causes the sum to grow indefinitely. In order to prevent infinite reward problem, it is crucial to define γ as a number less than one to force convergence.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{0}^{\infty} \gamma^k r_{t+k+1} \tag{4.3}$$

Since expected future reward is defined per action, Equation 4.2 changes slightly and becomes as Equation 4.4. This update is applied after every transition from a nonterminal state S_t. In case S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is set to zero. As a consequence, each experiment can be defined with a quadruple of (State, Action, Reward, State′) or $(S_t, A_t, R_{t+1}, S_{t+1})$. This method is known as *Sarsa* algorithm and named *Q-Learning* because of the Q-Table used in the equation. Algorithm 3 describes this procedure.

$$Q(S_t, A_t) \longleftarrow (1-\alpha)Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})\big] \tag{4.4}$$

Algorithm 3: Q-Learning Work-Flow[1]

Algorithm parameters: $\alpha \in (0,1]$, $\gamma \in (0,1)$
Initialize $Q(s, a)$ for all $s \in \{S - terminal\}$, $a \in A(s)$ arbitrarily and $Q(terminal, a) = 0$
repeat
  Take action A based on policy π derived from Q table
  Observe reward R and land in state S'
  Choose A' from S' using the same policy π
  $Q(S,A) \leftarrow (1-\alpha)\, Q(S,A) + \alpha\, \big[R + \gamma\, Q(S', A')\big]$
  $S \leftarrow S'$ and $A \leftarrow A'$
until *End of episode or S is terminal*

[1] The pseudocode has been taken from Sutton and Barto [57]

An agent takes action based on a policy function π which in turn is derived – partially or completely – from the Q-Table. However, it shall be noted that in the beginning of episode the Q-Table is initialized to zero. In such situations, it takes very long time for agent to discover new states which is particularly harmful for data stream processing systems. This problem is usually solved by introducing small degree of randomness. The policy function decides randomly with a small probability ϵ and in any other case $(1-\epsilon)$, it behaves based on Q-Table. In some cases it is desirable to initialize the Q-Table with a random value or based on some *heuristic* function that is partially derived from domain knowledge.

Furthermore, if two or more actions have equal expected reward, then policy function has to decide on one of them by breaking the tie. There are different strategies to solve this issue. Choosing a random action is one of them. However, in some domains this might be dangerous or harmful. Rather than choosing randomly, a simple heuristic inspired by domain knowledge can be helpful too.

4.1.2 Value Iteration

In case the workload pattern is unpredictable, environment dynamics is unperceived. In other words, reward values of each action and probability of landing in a specific state after taking an action is unknown. However for the sake of discussion, let us assume environment dynamics is *known*. In such situations, Reinforcement Learning can be solved using *Dynamic Programming* techniques. In this section, the problem will be solved for environments with known dynamics. Then, some of the core ideas will be applied to adopt the solution for environments with unknown dynamics.

Finding an optimal policy function, when environments dynamics is known can be *recursively* solved by applying dynamic programming. As mentioned previously, the agent is trying to maximize expected reward by taking optimal actions. Here expected reward is defined with $\mathbb{E}$ and $\max_a$ selects maximum value based on specified action.

$$\begin{aligned} V^*(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma V^*(S_{t+1}) \,|\, S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma V^*(s')\right] \end{aligned} \tag{4.5}$$

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \,|\, S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \,|\, s, a)\left[r + \gamma \max_{a'} Q^*(S_{t+1}, a')\right] \end{aligned} \tag{4.6}$$

For scenarios where p and r are unknown – which is the case for this thesis – these values can estimated by collecting samples. As the agent runs for some period of time, it collects samples and then estimates p and r. Given that p and r are estimated to some degree, it is possible to initialize Q-Table with more useful values – compared to zero or random values – and speed up learning process. Whether this estimation is accurate or not – how many samples shall be collected for a precise estimation – is another question which will be evaluated in Section 5.

The following procedure is applied for collecting samples which is inspired by Dutreilh et al. [22]. As agent runs and observes a quadruple of (s, a, r, s'), the following counters are updated.

$$\begin{aligned} P[s,a,s'] &= P[s,a,s'] + 1 \\ R[s,a] &= R[s,a] + r \\ C[s,a] &= C[s,a] + 1 \end{aligned} \tag{4.7}$$

Given these statistics, estimators $\overline{P}$, $\overline{R}$ are calculated to replace original p, r in Equation 4.6 respectively.

$$\begin{aligned} \overline{P}[s,a,s'] &= \frac{P[s,a,s']}{C[s,a]} \\ \overline{R}[s,a] &= \frac{R[s,a]}{C[s,a]} \end{aligned} \tag{4.8}$$

After estimating p, r, Algorithm 4 can be used to produce optimal policy. Not the difference between **max** and **argmax** functions. The former returns the *maximum value* and the later returns the *action that has the maximum value*. Value Iteration that described in Algorithm 4 can be used independent of Temporal Difference. However, in this thesis it is used to get a useful initialized Q-Table. Thus, when sample collection period is over and Q-Table is initialized, normal Q-Learning approach is used thereafter. The reason is that in Value Iteration, after sampling period, Q-Table is never updated again. However, by combining Value Iteration – for initializing Q-Table – and Q-Learning, Q-Table is also updated at runtime which improves decision accuracy. [2]

Algorithm 4: Value Iteration for Estimating $\pi \approx \pi^{*}$[1]

```
A small threshold θ > 0 determining accuracy of estimation
Initialize V(s) for all s ∈ {S − terminal} arbitrarily and V(terminal) = 0
repeat
    Δ ← 0
    foreach s ∈ S do
        v ← V(s)
        V(s) ← max_a Σ_{s',r} p(s', r | s, a) [r + γ V(s')]
        Δ ← max(Δ, |v − V(s)|)
    end
    Output policy π ≈ π*, such that
    π(s) = argmax_a Σ_{s',r} p(s', r | s, a) [r + γ V(s')]
until Δ < θ
```

4.2 Design

In this section technical design and structure of the thesis will be explained and discussed. As mentioned in previous section, two Reinforcement Learning techniques – namely Q-Learning (Temporal Difference) and Value Iteration – have been implemented. Both of these methods require defining *state space*, *policy* and *reward* functions. Typically, Reinforcement Learning agents move from one state to another by taking an action which is proposed by policy function, getting feedback – known as reward – from environment and landing into destination state. In next sections each of these components will be discussed.

4.2.1 State Space

State space defines – models – the environment where the agent is living. Usually, it is combination of environment's properties. As an example, For a typical web application, a sample state space could be defined as a tuple of [Number of web servers, Total workload defined as requests per second, Average round trip delay of requests]. For a Hypervisor, a sample state space can be defined as a tuple of [Number of running virtual machines, Total CPU utilization, Total used

[2] The pseudocode has been taken from Sutton and Barto [57]

RAM]. For a data stream processing system, a sample state space can be defined as a tuple of [Number of worker nodes, Total workload defined as messages per second, Average latency of messages].

As state space contains more precise information, it gets bigger and bigger. Thus, in most systems it is *discretized* to some degree to reduce its size. State space reduction has a couple of benefits.

Change Observation For example, in most systems a 1% increase/decrease in CPU utilization is not considered as a noticeable change but 5% is.

Absorbing Sudden Bursts In some applications, there might be a sudden change in one aspect of the system for a slightly short time. For example 20% increase in CPU utilization for 5 seconds may not be considered as sustained workload increase.

Preventing Zig-Zag Decisions If state space is discretized to fine-grained states, it might lead to consecutive contradictory decisions – A phenomena known as *Zig-Zag* decisions. For example, Auto-Scaler may decide to perform a scale-in action in one step and scale-out action in next step. A reasonable way to prevent this never ending loop is to discretize the state space into coarse-grained states.

However, it shall be noted that *over discretizing* the state space may have negative impacts on target system. Section 5 contains an experiment to evaluate discretization impact. In this thesis, a combination of latency and direction of workload is incorporated in to state space.

Latency Each micro-batch is associated with two latencies. First, *Processing Latency* – time that takes to process a single micro-batch from start to end. second, *Scheduling Latency* – time that each micro-batch stays in a queue *waiting* to be processed. In this project, total latency (processing latency + scheduling latency) is stored as an average value calculated over a window of time.

Workload Direction It is reasonable to consider average number of messages that are being processed in state space. However, in order to reduce state space only its *direction* is stored. That is, instead of storing a pure number as average number of messages processed per window of time, only a boolean value indicating that incoming messages are increased to decreased – compared to last window – is stored.

As mentioned in Section 2 each state is associated with three actions – No-Action, Scale-In and Scale-Out. Each action has corresponding reward – Q-Value– attached to it. Figure 4.1 depicts this relationship.

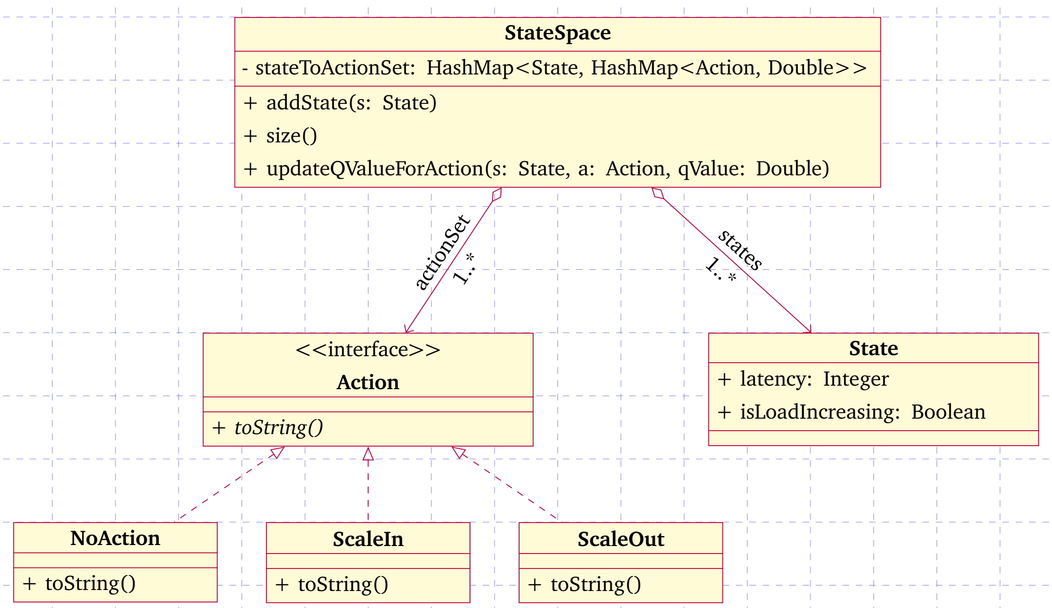


Figure 4.1: State Space Class Diagram

4.2.2 State Space Initialization

A typical workload either looks like the ones depicted by Figure 4.2, or they can be built by combining multiple graphs in Figure 4.2.

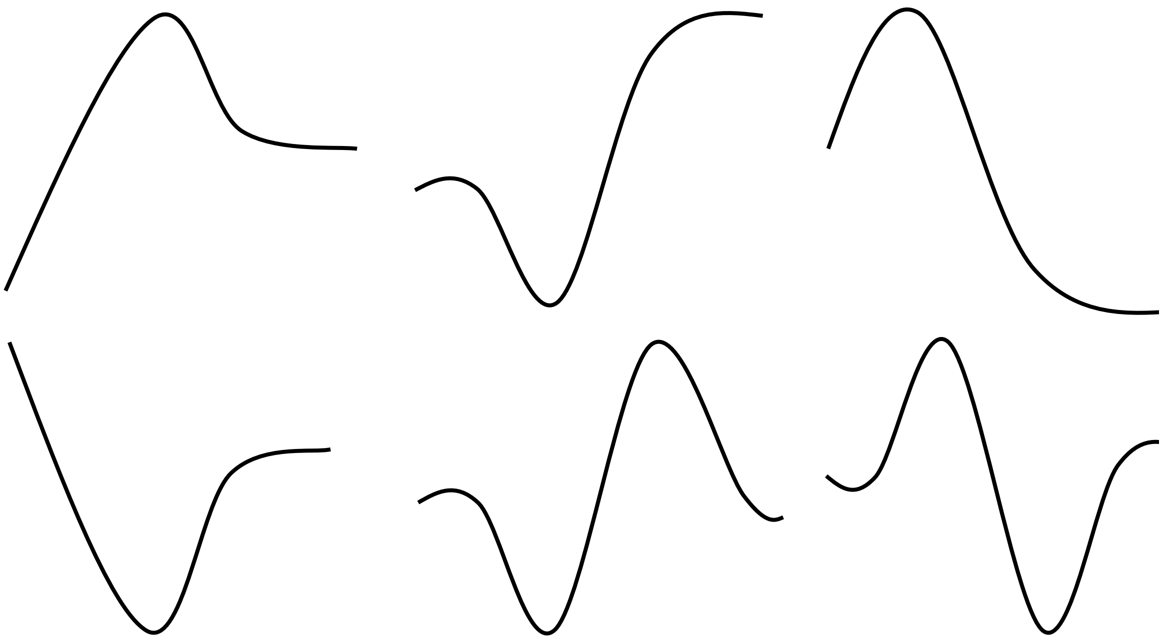


Figure 4.2: Workload Categories

However they have one thing in common. It rises from a base state, goes to a peak and then returns back. Without loosing generality and with a sufficiently large sliding window, workload-to-latency graph can be modeled as Figure 4.3. This graph is divided into four regions. As workload increases, latency is approaching target latency (Region 1). In case the workload keeps increasing, the agent goes beyond the target latency which leads to violating SLOs (Region 2). Assuming the agent takes some reasonable action, latency starts to decrease (Region 3). Finally, at some point it will drop below the target latency line (Region 4).

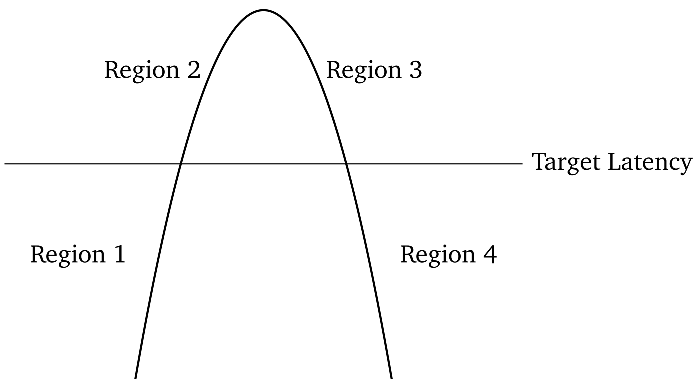


Figure 4.3: Different Regions of a Typical Workload

Typically when the agent starts, its Q-Table is empty – initialized with zero. Over time, as the agent takes actions and discovers its environment, it updates its Q-Table. Since Q-Table is initialized with zero, the agent usually takes some random actions from time to time with a small probability. This helps to discover the environment. In some applications, this behavior is sufficient. However, in fast changing environments like streaming systems this process takes very long time to converge. Thus developers usually try to initialize Q-Table beforehand to speedup the learning process. In this thesis, three different initializers have been implemented. Figure 4.4 illustrates class diagram of state space initializers.

Zero Initializer This is the default implementation which initializes all the Q-Values – reward values – to zero.
Random Initializer This initializer assigns a random value between (-1,1) to Q-Values.
Optimized Initializer This initializer uses a *heuristic* to initialize the Q-Table. Typically in most systems, there is a well defined set of SLOs defined by users that should be respected by Auto-Scaling system. This initializer exploits a normalization function defined in Equation 4.9. For each region, Q-Table is initialized with different values.

$$n = \frac{\text{current latency}}{\text{target latency}} \tag{4.9}$$

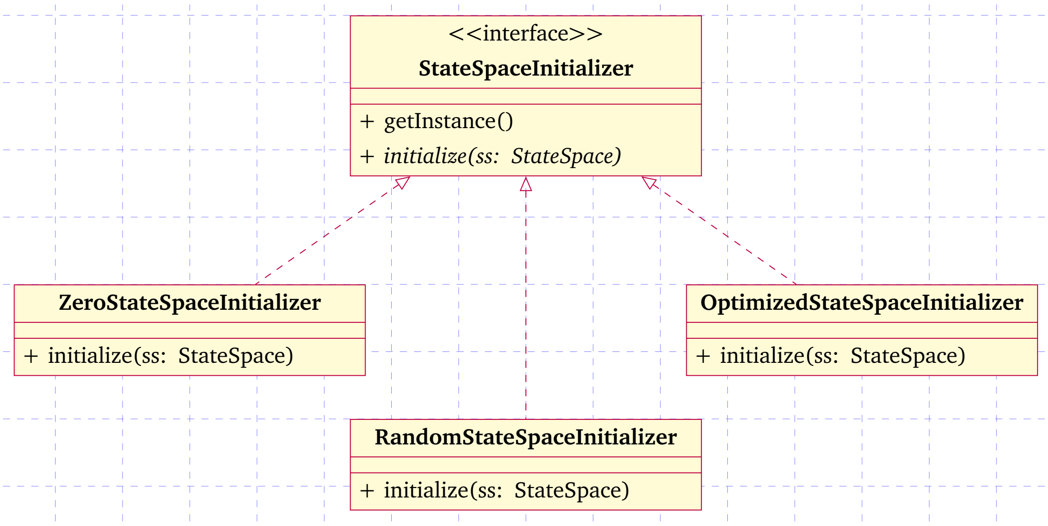


Figure 4.4: State Space Initializer

The following describes the method that optimized initializer applies to four regions defined in previous section. Table 4.1 summarizes each region and describes preferred action in each region.
Region 1 In this region, latency is increasing as workload is increasing. So it is reasonable to give more reward to Scale-Out as the agent approaches target latency line. This is called *Early Scale-Out* action which helps to add worker tasks in order to prevent violating SLOs. Thus the reward for each action is initialized as Equation 4.10. In this region, Scale-Out and No-Action rewards are positive but less than one.

$$\text{Scale-Out} = \frac{\text{current latency}}{\text{target latency}} \qquad \text{No-Action} = 1 - \frac{\text{current latency}}{\text{target latency}} \qquad \text{Scale-In} = 0 \tag{4.10}$$

Region 2 In this region, the agent has fall behind and latency has been increased up to the point that it is more than target latency. At this point Scale-Out is still the preferred action. It is reasonable to initialize Scale-In with a negative reward to prevent the agent from taking unjustifiable action. Reward for each action is initialized as Equation 4.11. In this region, Scale-Out reward is larger than one but No-Action reward is less than one.

$$\text{Scale-Out} = \frac{\text{current latency}}{\text{target latency}} \qquad \text{No-Action} = \frac{\text{current latency}}{\text{target latency}} - 1 \qquad \text{Scale-In} = -1 \tag{4.11}$$

Region 3 In this region, either the workload is decreasing or the agent has taken a presumably reasonable action that has led latency to decrease. Usually, it is not required to take any further action here. Thus, to let latency stabilize, No-Action is a preferred action in this region. Additionally, Scale-In is still prohibited.

$$\text{Scale-Out} = 0 \qquad \text{No-Action} = 1 \qquad \text{Scale-In} = -1 \tag{4.12}$$

Region 4 In this region, the latency has dropped below target latency and is decreasing gradually. Thus, as the agent gets farther away from target latency line, we encourage the agent to take Scale-In action. However, near target latency line, we still prefer to take No-Action, since it is still dangerous to take Scale-In.

$$\text{Scale-Out} = 0 \qquad \text{No-Action} = \frac{\text{current latency}}{\text{target latency}} \qquad \text{Scale-In} = 1 - \frac{\text{current latency}}{\text{target latency}} \tag{4.13}$$

Region	Preferred Behavior
Region 1	Early Scale-Out is preferred to avoid hitting target latency.
Region 2	Target latency is already violated, so Scale-Out is still preferred.
Region 3	Although latency is higher than target – but decreasing, No-Action is preferred to let latency stabilize.
Region 4	Latency is stable again. If latency is sufficiently low, then Scale-In is preferred.

Table 4.1: Summary of Latency Regions

4.2.3 Policy Functions

Policy π is a function that guides the agent to take an action in each state. Different strategies can be implemented but at its core, a policy function is either extracted from Q-Table. An optimal policy π^* is a policy that returns actions that lead to maximum expected reward. Before getting into details about implemented policies of this thesis, *monotonicity constraint* [34] shall be explained. Monotonicity constraint mandates:

- If Scale-Out was the chosen action in last state and latency has been increased – compared to last state, then it is not possible to take Scale-In in current state. The reason is that, even by applying Scale-Out, latency has been increased. So, for a even higher latency, there is no reason that Scale-In could ever help.

- If Scale-In was the chosen action in last state and latency has been decreased – compared to last state, then it is not possible to take Scale-Out in current state. The reason is that, latency is decreasing without taking Scale-Out. So, for a even lower latency, there is no need for Scale-Out. Even though Scale-Out is a safe action in this case, but in Spark Streaming any Scale-In/Out action cause reshuffling of intermediate results. Thus, it is crucial to avoid unnecessary – although safe – actions.

As confirmed by Heinze et al. [29], monotonicity constraint improves learning process. Algorithm 5 shows the pseudocode of this procedure.

Algorithm 5: Monotonicity Constraint

```
feasibleActions ← GetActionsFromStateSpace(currentState)
if currentState.latency > lastState.latency AND lastAction == ScaleOut then
    feasibleActions ← feasibleActions − ScaleIn
end
if currentState.latency < lastState.latency AND lastAction == ScaleIn then
    feasibleActions ← feasibleActions − ScaleOut
end
```

Three different policies have been implemented in this thesis. Figure 4.5 depicts class diagram of these polices. Table 4.2 summarizes the behavior of each policy.

Greedy Policy This policy takes action in two phases. First, it applies monotonicity property. After that, it selects the best possible action. Best action here means the action with largest/lowest Q-Value, depending on how reward function has been implemented.

One Minus Epsilon Policy This policy wrap any other policy and adds some degree of randomness to it. That is, with a probability of ϵ it returns a random action and with a probability of $1-\epsilon$ it hands over the process to underlying policy. Underlying policy is not necessarily an optimal – greedy – policy.

Decreasing One Minus Epsilon Policy This policy is similar to previous policy. The different is that randomness (ϵ) is decreasing over time until it reaches zero after which it unconditionally hands over to underlying policy.

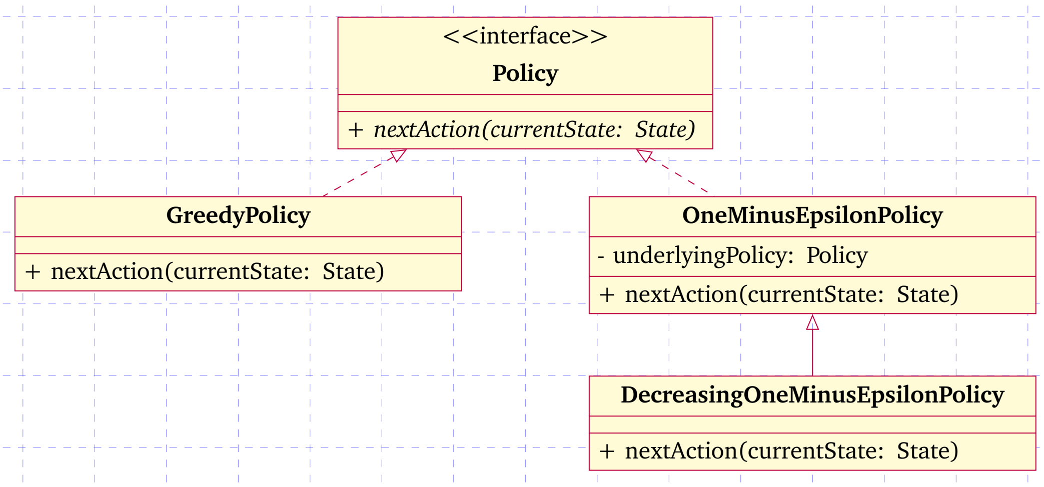


Figure 4.5: Policy Functions

Policy	Behavior
Greedy	Always stick to action with highest/lowest Q-Value.
One Minus Epsilon	Take random action with a probability of ϵ, otherwise stick to underlying policy.
Decreasing One Minus Epsilon	Same as $1-\epsilon$ policy, but ϵ is decreasing over time.

Table 4.2: Summary of Policy Functions

4.2.4 Reward Function

Reward function is used as a mechanism to return environment's feedback to agent. Reward is usually abstracted as floating point number. Depending on how a *good action* is defined, the agent might get a negative or positive reward. Both reward functions are based on the same behavior as state space initializers 4.2.2. Specifically, rewards are assigned based on general rules.

1. If the taken action didn't improve the latency – compared to last state, generally negative reward should be returned. There are exceptions to this rule.

2. If latency and workload are increasing simultaneously, then rule 1 does not apply. Thus, Scale-Out gets positive reward.

3. If the agent is at region 3 – refer to Figure 4.3, then No-Action gets positive reward to help the agent stabilize.

4. When the agent is in region 4, reward function prioritizes Scale-In. That is, when latency and workload are decreasing, No-Action gets negative reward to encourage the agent apply Scale-In action. We call this **Prefer Scale-In When Load is Decreasing** reward function in the rest of thesis.

4.2.5 Executor Strategy

Deciding when to add/remove resources is one problem – which is solved by policy function – and deciding *how many* resources to add/remove is another problem. The strategy heavily depends on workload's characteristics. As a consequence multiple strategies have been implemented in this thesis. Figure 4.6 depict the class diagram and Table 4.3 summarizes them.

Static Strategy This strategy adds or removes executors by a static – fixed – number. No further measure is considered.

Linear Strategy This strategy adds or removes executors, with a linear coefficient. The ration behind this strategy is that, if multiple consecutive Scale-Out decision is taken, then most probably the workload has been increased considerably. Thus, at each step the number of added executors are increased by a fixed factor. As an example, if in case of consecutive Scale-Out actions, this strategy adds 1, 2, 4, 6, 8, . . . executors at each step. Additionally, if a Scale-In occurs between consecutive Scale-Out actions, then the coefficient factor is reset to one. The same also applies when consecutive Scale-In actions occur.

Queue Aware Strategy As mentioned in Section 4.2.2, all micro-batches are associated with scheduling latency. This strategy builds a queue of batches that are waiting to be processed. Addition of executors depends upon the length of waiting queue. Thus, if cluster is struggling to process batches and many of them are queued in waiting queue, then this strategy adds more executors in one decision – the precise number depends on queue length. However, removing executors is done with caution. That is, removing executors is done using static strategy with a fixed number. The result is exponentially increasing - statically decreasing strategy.

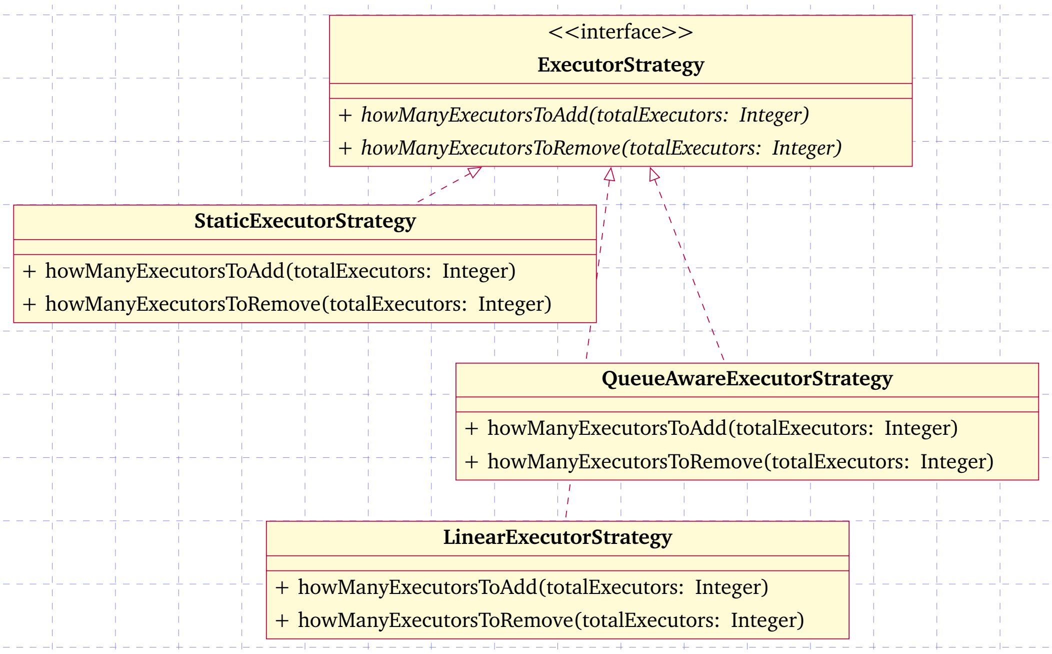


Figure 4.6: Executor Strategies

Executor Strategy	Behavior
Static	Add or remove at a fixed rate.
Linear	Add or remove with a fixed coefficient for same consecutive actions.
Queue Aware	Add based on waiting queue length. Remove statically at a fixed rate.

Table 4.3: Summary of Executor Strategies

4.3 Configuration

In this section configuration parameters that are being utilized by the project will be explained. As mentioned in Section 1, the proposed solution should be highly configurable. Table 4.4 lists all the configuration parameters and describes them briefly. Note that all the parameters should be prefixed with `spark.streaming.dynamicAllocation`.

Parameter	Description
`minExecutors`	Minimum executors for Auto-Scaler.
`maxExecutors`	Maximum executors available for Auto-Scaler to use.
`executorGranularity`	Number of executors to add/remove at each step.
`maxLatency`	Maximum latency to limit the size of state space.
`targetLatency`	Latency that is acceptable for user.
`latencyGranularity`	Granularity of latency that is used for state space discretization.
`incomingMessageGranularity`	Granularity of workload that is used for state space discretization.
`windowSize`	Sliding window size that is utilized for making decision.
`learningFactor`	Learning factor of the agent.
`discountFactor`	Discount or reward factor of the agent.
`initializationMode`	Different modes of state space. One of `zero`, `random` or `optimal`.
`epsilon`	The value of ϵ used for randomized policy.
`epsilonStep`	ϵ is reduced by this amount when used in decreasing random policy.
`valueIterationInitializationCount`	Iterations that Value Iteration algorithm performs to initialize state space.
`valueIterationInitializationTime`	Time to let Value Iteration algorithm collect metrics and samples.
`policy`	Defines the policy function.
`reward`	Defines the reward function.
`decisionInterval`	The interval of time that the agent makes decision.
`executorStrategy`	Defines the executor strategy. One of `static`, `linear`, `relative`.
`seed`	Seed value to control the randomized behavior in consecutive runs.

Table 4.4: Summary of Configuration Parameters

4.4 Conclusion

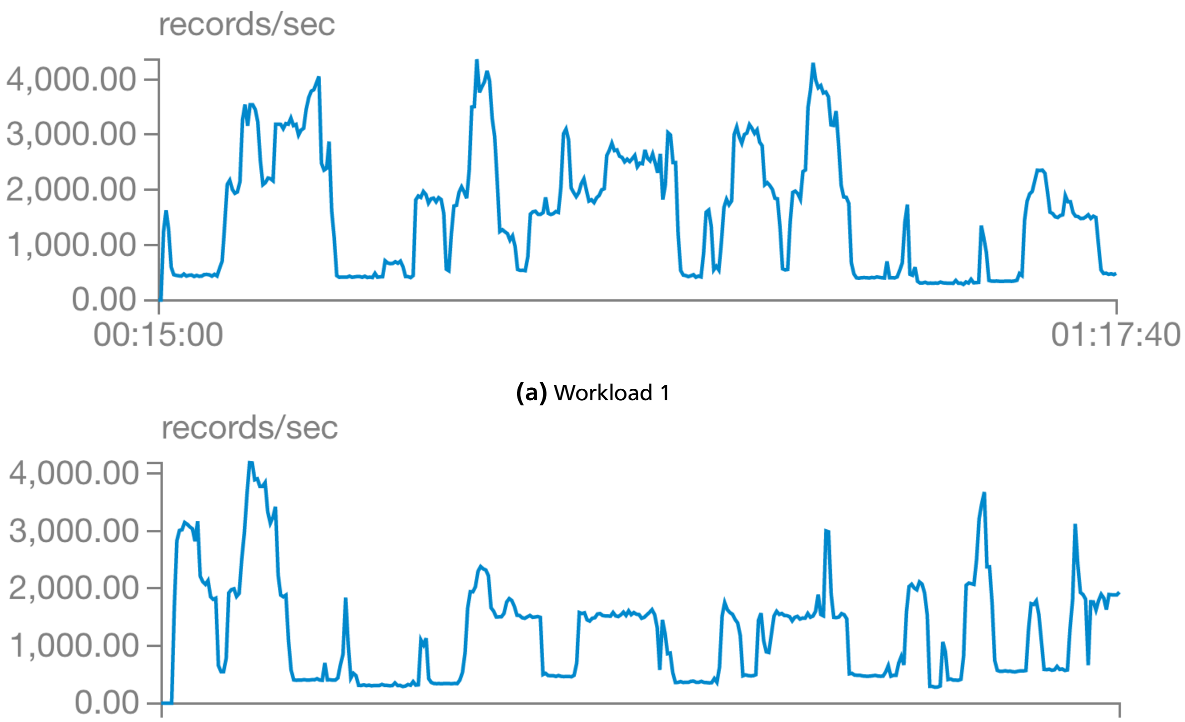
In this chapter design and structure of this thesis has been explored. Since Reinforcement Learning approaches are capable of adapting to ever changing streaming workloads, two techniques, namely Temporal Difference and Value Iteration, have been selected and implemented. State space have been designed to contain enough information of the environment and at the same time be small enough. Multiple state space initializers have been designed to show the initialization effect on learning process. Furthermore, in order to make the implementation highly flexible against different environments, multiple policy, reward and executor strategies have been implemented. Section 5 evaluates the effect of changing these parameters.

5 Evaluation

In this chapter the two implemented techniques will be evaluated under three read world workloads. In order to show the effect of each configuration parameter, multiple experiments have been designed and deployed. Table 4.4 defines the configuration space. First, characteristics of workload will be discussed in Section 5.1. Thereafter, there is separate section for each experiment.

5.1 Workload Characteristic

DEBS 2014 [39] has been chosen as a real world workload to test the implementation. Each workload contains data from a random location of the original workload and replayed to feed Spark cluster. All experiments were run for one hour. Figure 5.1 shows distribution of messages in two workloads which have been captured from Spark UI.



(a) Workload 1

(b) Workload 2

Figure 5.1: Two Workloads of DEBS 2014

The workload involves sensors that measure energy consumption of devices. Each device is connected to one household which in turn is located in one house. This creates a hierarchy from house as parent of households and household as parent of devices. The workload asks to predict energy consumption per device, per household, and per house for next window of time. The prediction runs over a sliding window of historical measurements which contains n elements. Assuming $window$ variable contains n elements of the history, Equation 5.1 defines how the predicted value is calculated.

$$\text{predicted value} = \frac{\text{average(window)} + \text{median(window)}}{2} \tag{5.1}$$

Note that intensity of the workload does not solely depend on number of incoming messages. It also depends on uniqueness of device IDs. A unique device ID inside a micro batch leads to recalculation of Equation 5.1 which on its own depends on sort operation to calculate median value. That is, the following cases might occur in one micro batch:

- Number of incoming messages is high and uniqueness of devices is also high.
- Number if incoming messages is high but uniqueness of devices is low.
- Number of incoming messages is low but uniqueness of devices is high.
- Number of incoming messages is low and uniqueness of devices is also low.

As an example, consider these two batches:

1. A batch of 500 records belonging to 500 hundred devices – one record for each device – which leads to 500 sort operations. Bare in mind that, it is *not* 500 hundred sort operation on a 1-element list. Each record is accumulated with previous records of the sliding window and then sorted.
2. A batch of 2000 records belonging to 5 devices – 400 hundred records for each device – which leads to 5 sort operations.

Case 1 is much more CPU intensive than case 2. In order to make each workload CPU intensive enough, window size is changed for each workload. Table 5.1 defines the history window size for each workload.

Workload	**Window Size** = n
Workload 1	1650
Workload 2	1900

Table 5.1: Workload Window Size

In all experiments *batch size* is set to 10 seconds. The cluster uses 24 executors with minimum of 4 executors that should be respected by Auto-Scaler. All experiments start with minimum number of executors – 4 in this case – and run for *one* hour. In case any training is required to run the experiment, training data set is separated from the original workload dataset. In all experiments – except the last one – four charts are illustrated. Two charts for latency and two charts for number of executors. Candlestick charts depicts minimum, 10 percentile, average, 90 percentile and maximum values. Furthermore, all experiments were run *two* times and the average of them is included in charts.

This experiment has been designed to illustrate the strategy of adding/removing executors when taking Scale-In or Scale-Out actions. Table 5.2 shows the configuration of this experiment.

#	Experiment	Configuration
1	Executor Strategy = Static	Executor Granularity: 1 History Window: 2 Minutes Target Latency: 30 Seconds Decision Interval: 1 Minute Table Initializer: Optimized Latency Granularity: 10 Seconds Learning Factor: 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Executor Strategy = Linear	
3	Executor Strategy = Queue Aware	

Table 5.2: Executor Strategy Configuration Parameters

Figure 5.2, 5.3 depicts latency charts for first workload. Furthermore, Figure 5.4, 5.5 depicts the behavior of strategies regarding number of executors. Similarity, Figure 5.6, 5.7, 5.8 and 5.9 illustrates latency and executor charts for second workload.

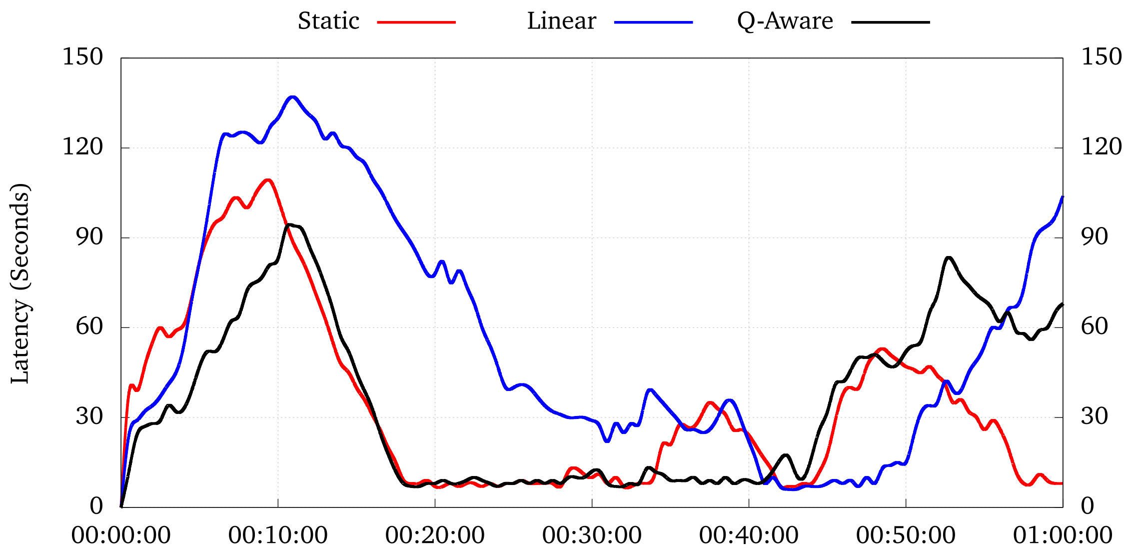


Figure 5.2: Executor Strategy – Workload 1 – Latency

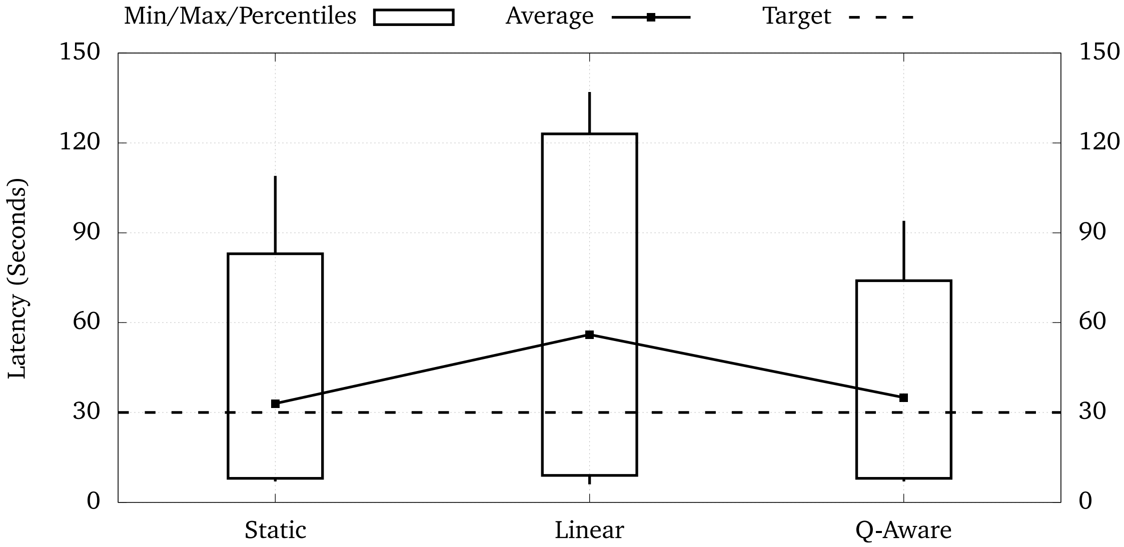


Figure 5.3: Executor Strategy – Workload 1 – Latency

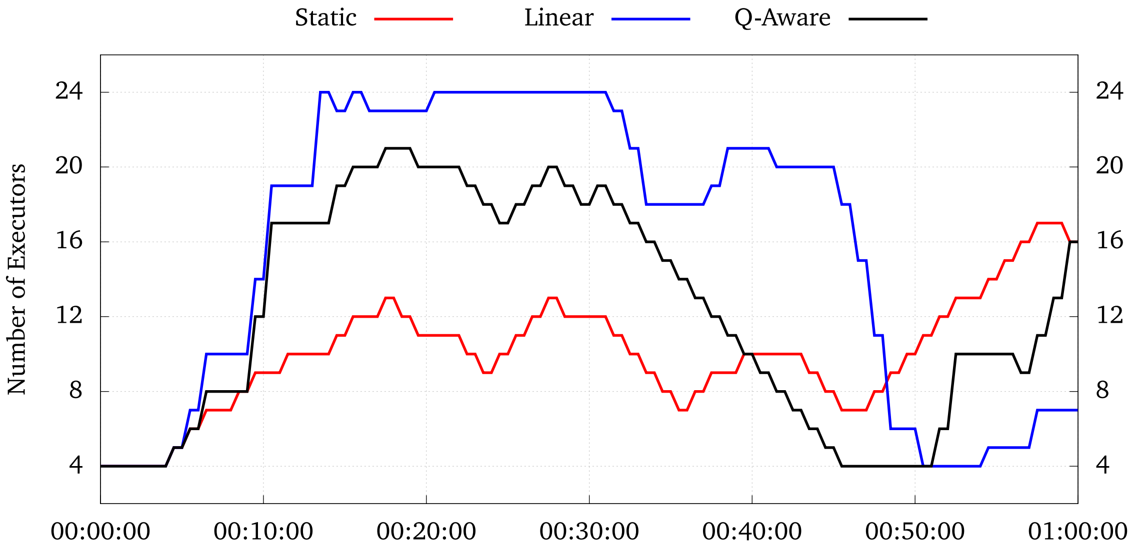


Figure 5.4: Executor Strategy – Workload 1 – Number of Executors

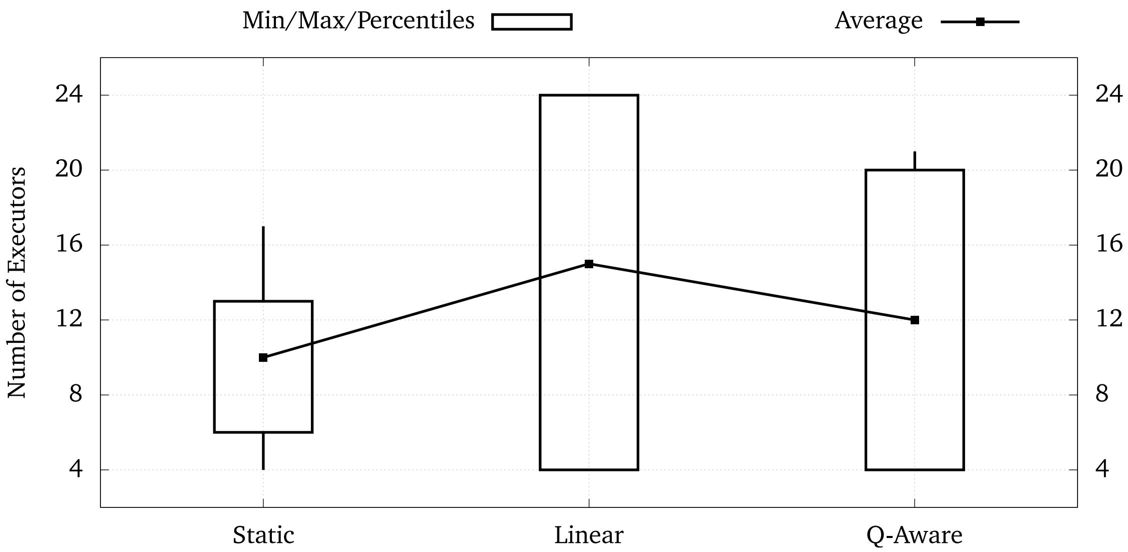


Figure 5.5: Executor Strategy – Workload 1 – Number of Executors

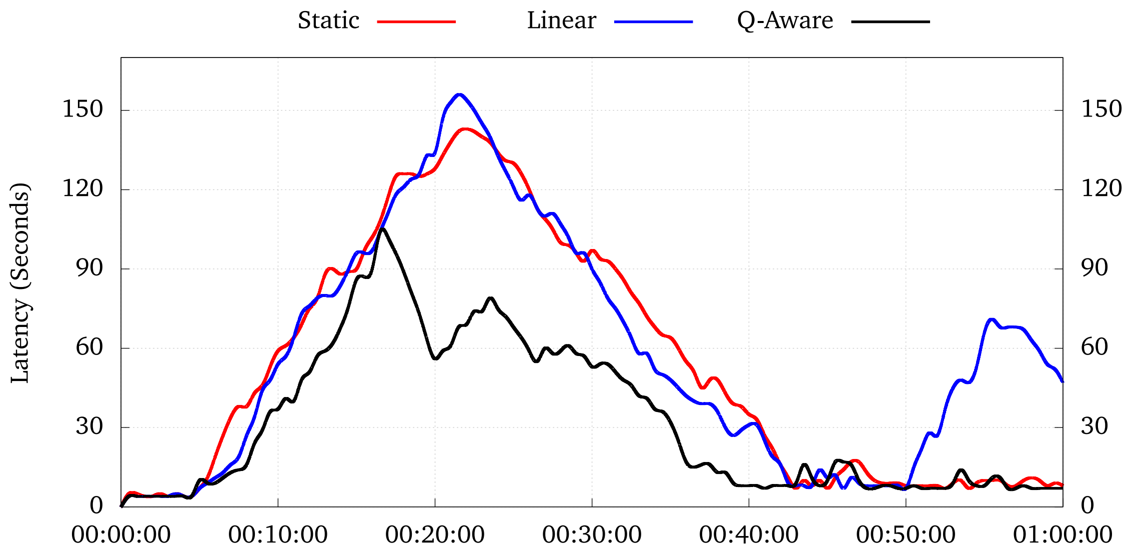


Figure 5.6: Executor Strategy – Workload 2 – Latency

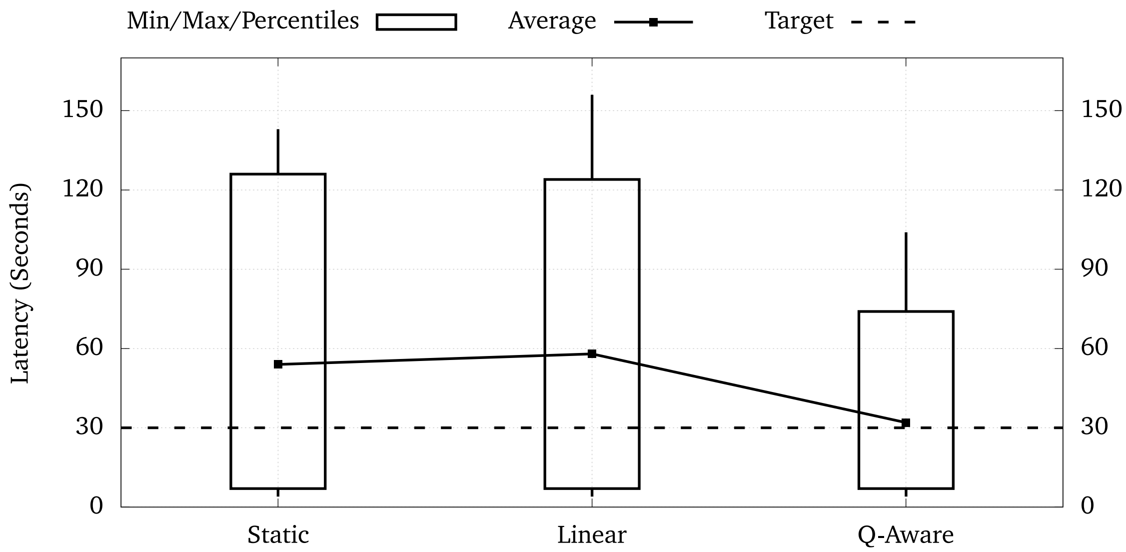


Figure 5.7: Executor Strategy – Workload 2 – Latency

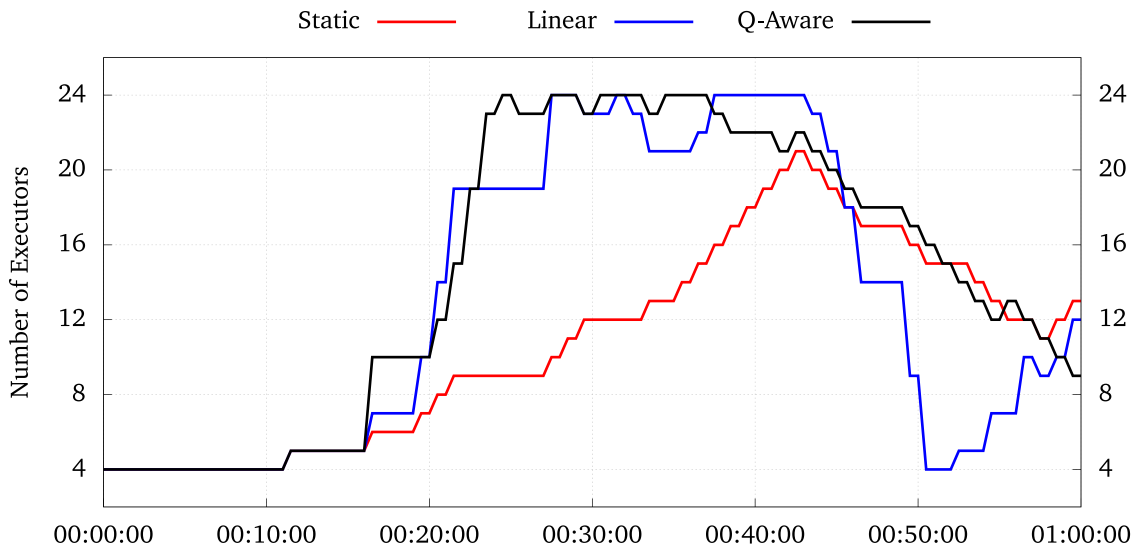


Figure 5.8: Executor Strategy – Workload 2 – Number of Executors

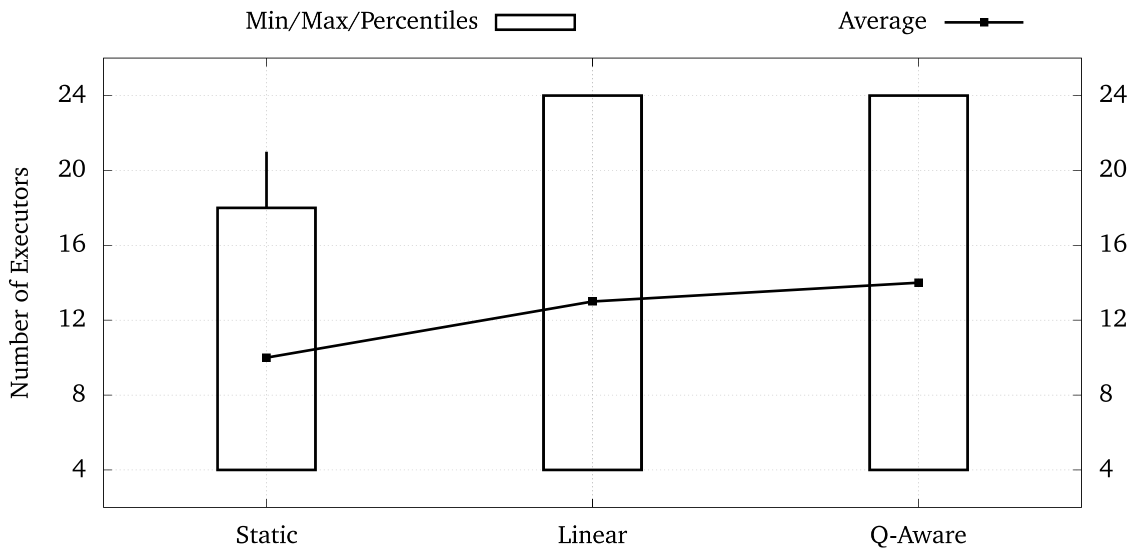


Figure 5.9: Executor Strategy – Workload 2 – Number of Executors

5.2.1 Conclusion

As depicted in last section, for Workload 1, Static strategy performs slightly better than Queue Aware strategy. Both of them were able reach target latency to some degree, though they are still above target latency. However, for Workload 2 Queue Aware strategy outperforms all other strategies, since Static and Linear strategies violate target latency.

It can be speculated that Queue Aware strategy is the best strategy amongst three strategies. However, it can't be proven now. More experiments shall be done. Thus, for next set of experiments Queue Aware strategy is chosen to be evaluated extensively. Furthermore, the conclusion is only based on two workloads and it can't be generalized.

5.3 Experiment 2: History Window

This experiment has been designed to illustrate the effect of history window on quality of decision made by Auto-Scaler. Table 5.3 shows the configuration of this experiment.

#	Experiment	Configuration
1	History Window = 1 Minute	Executor Granularity: 1 Executor Strategy: Queue Aware Target Latency: 30 Seconds Decision Interval: 1 Minute Table Initializer: Optimized Latency Granularity: 10 Seconds Learning Factor: 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	History Window = 2 Minutes	
3	History Window = 3 Minutes	

Table 5.3: History Window Configuration Parameters

Figures 5.10, 5.11, 5.12 and 5.13 illustrates latency and executor charts for Workload 1. Figures 5.14, 5.15, 5.16 and 5.17 illustrates latency and executor charts for Workload 2.

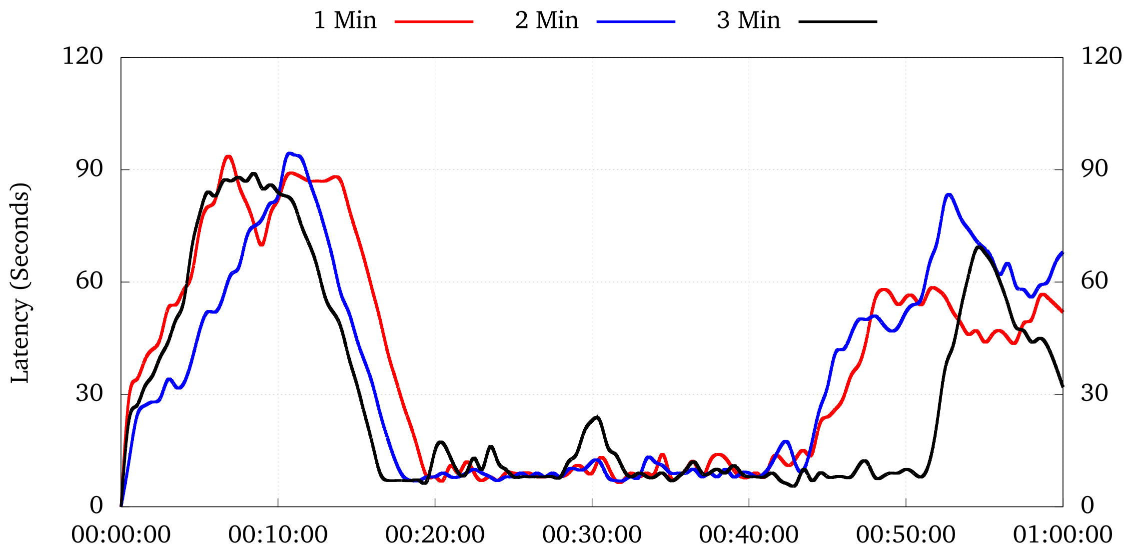


Figure 5.10: History Window – Workload 1 – Latency

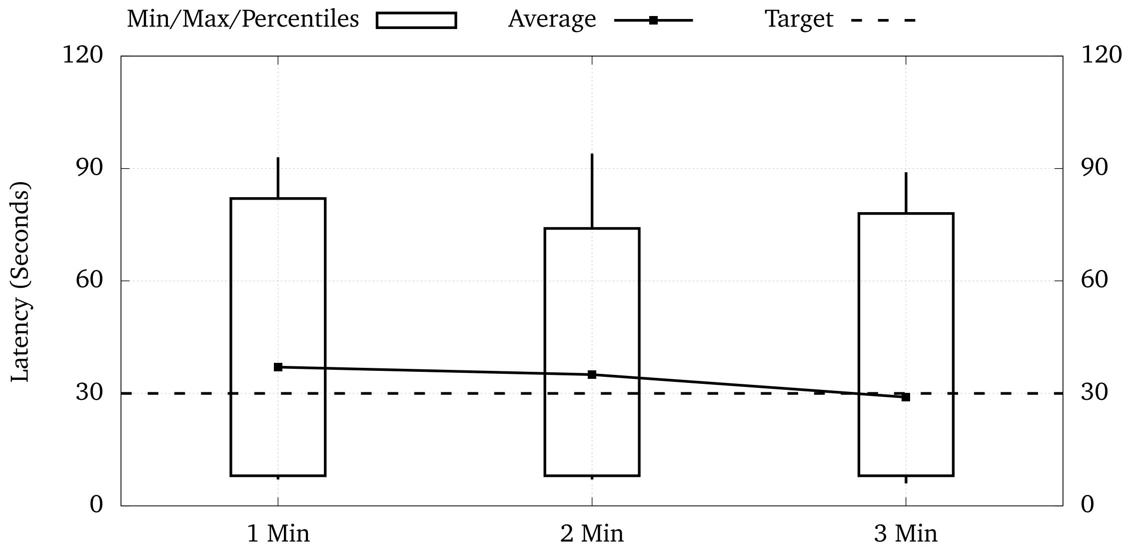


Figure 5.11: History Window – Workload 1 – Latency

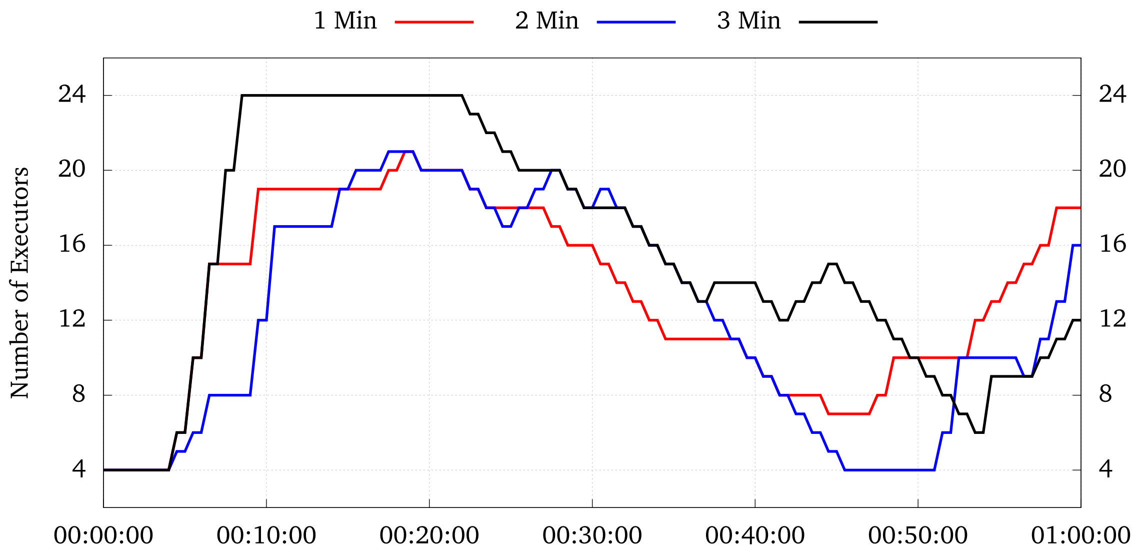


Figure 5.12: History Window – Workload 1 – Number of Executors

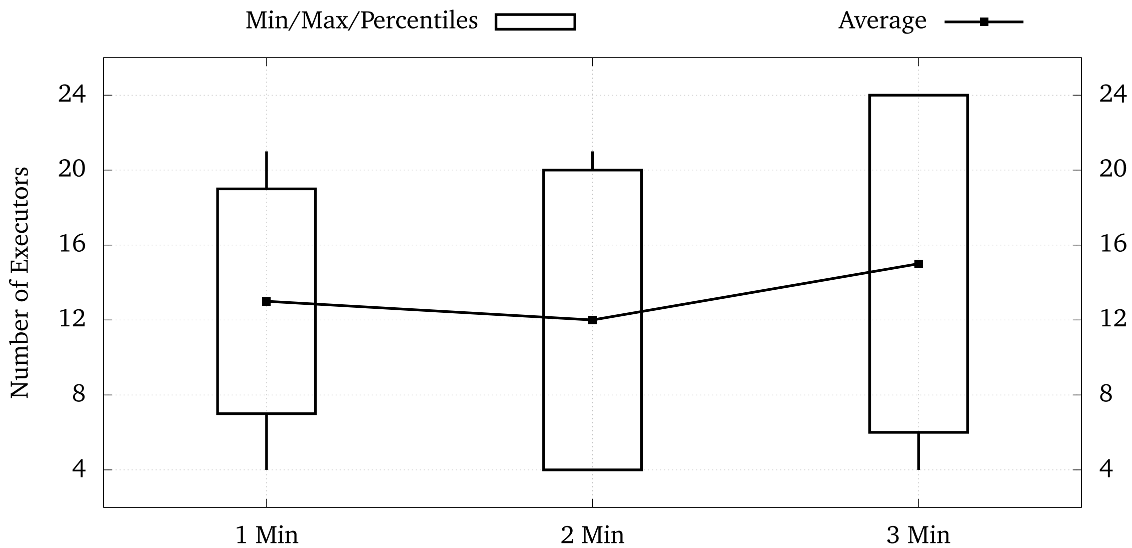


Figure 5.13: History Window – Workload 1 – Number of Executors

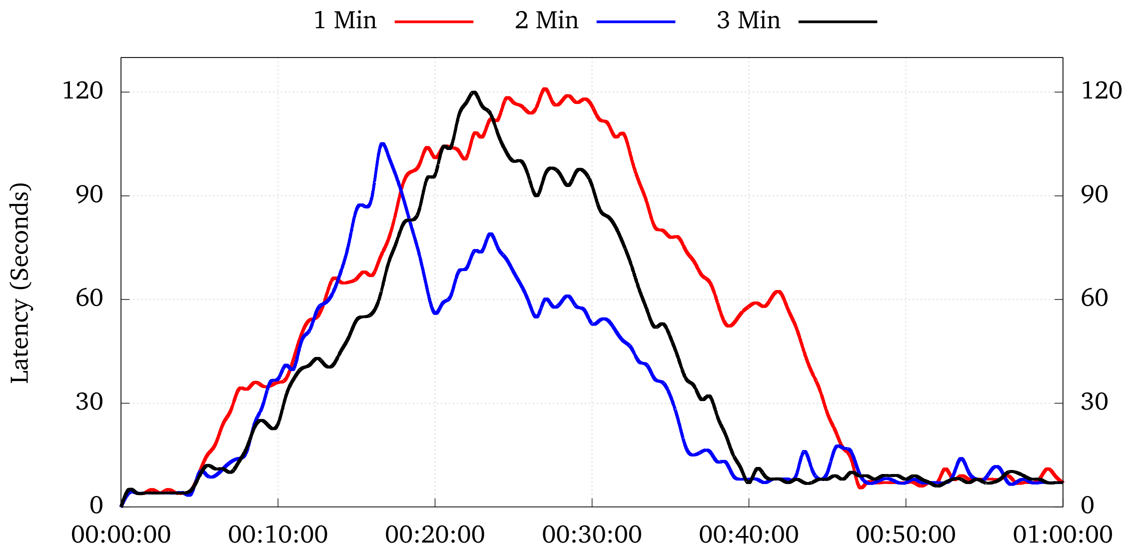


Figure 5.14: History Window – Workload 2 – Latency

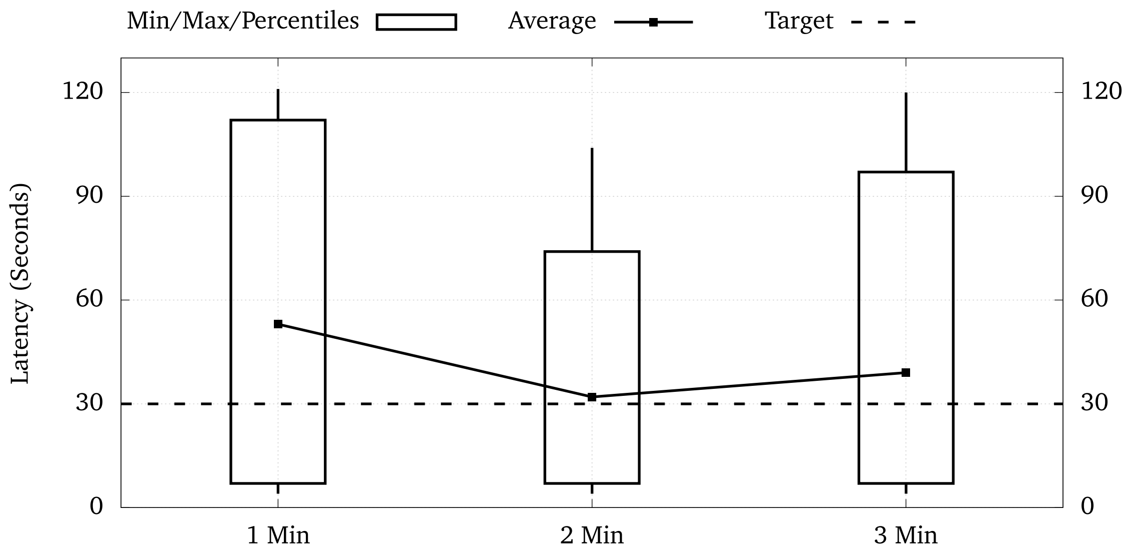


Figure 5.15: History Window – Workload 2 – Latency

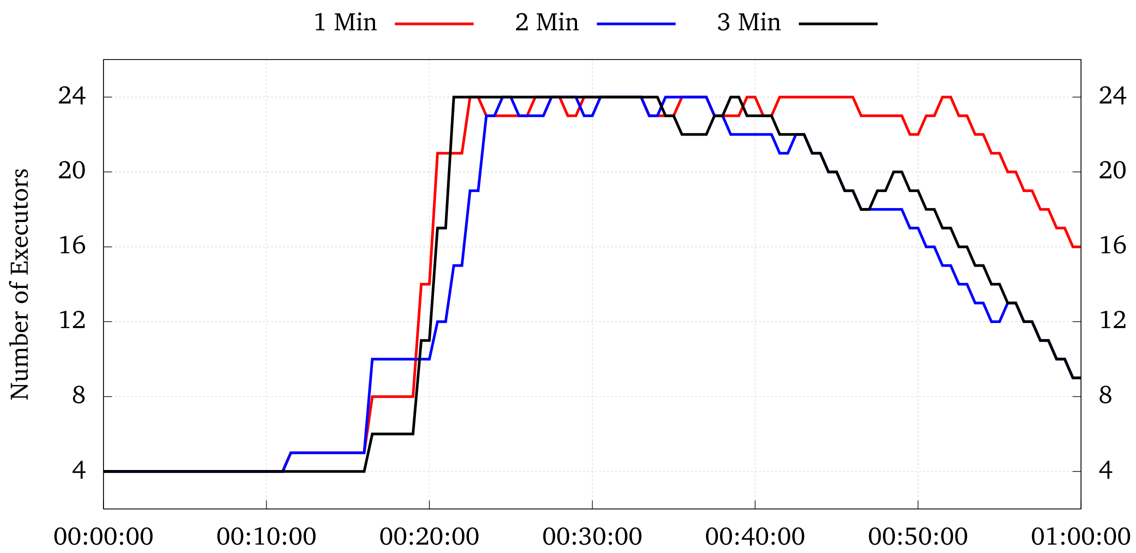


Figure 5.16: History Window – Workload 2 – Number of Executors

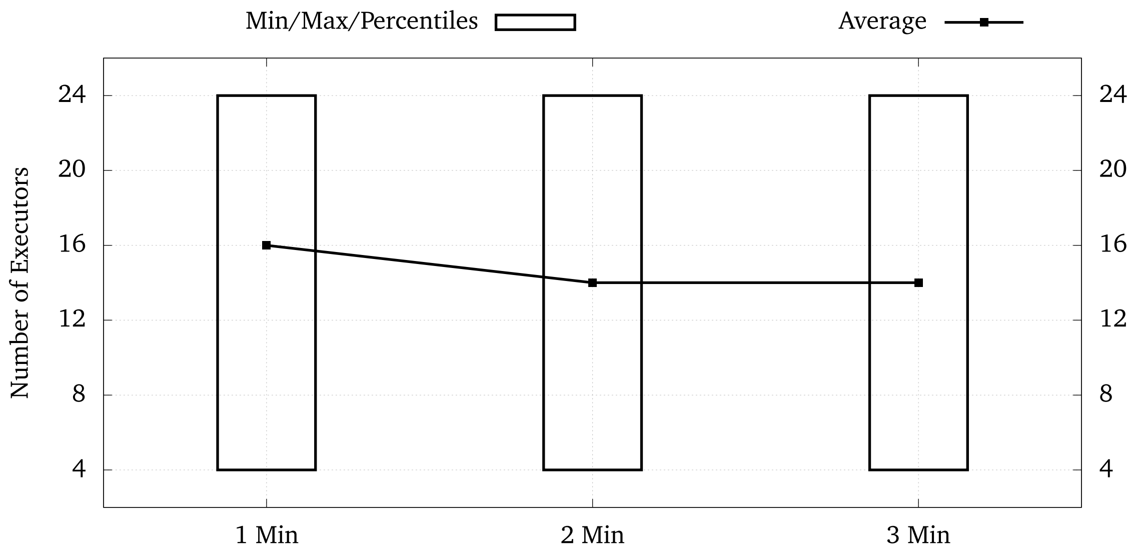


Figure 5.17: History Window – Workload 2 – Number of Executors

5.3.1 Conclusion

For Workload 1, three-minute history window leads to better latency albeit with slightly more executors. However, for second workload two-minute window outperforms both one and three minute windows.

Also note that, short history window (like one-minute) makes Auto-Scaler too sensitive to noisy workloads. This can be confirmed by Figure 5.16. As depicted, one-minute history window suffers from zig-zag decisions when it reaches 24 executors. Two-minute window to some degree suffers from this issue as well. However, three-minute window is resistant to this issue.

In general, history window of two minutes shows better results than the others. Thus, for rest of the experiments this window size will be used.

5.4 Experiment 3: Decision Interval

The Auto-Scaler implemented in thesis makes decisions in intervals defined by `decisionInterval` parameter. This experiment has been designed to illustrate the effect of this parameter on behvaior of the Auto-Scaler. Table 5.4 shows the configuration of this experiment.

#	Experiment	Configuration
1	Decision Interval = 1 Minute	Executor Granularity: 1 Executor Strategy: Queue Aware Target Latency: 30 Seconds History Window: 2 Minutes Table Initializer: Optimized Latency Granularity: 10 Seconds Learning Factor: 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Decision Interval = 2 Minutes	
3	Decision Interval = 3 Minutes	

Table 5.4: Decision Interval Configuration Parameters

Figures 5.18, 5.19, 5.20 and 5.21 illustrates latency and executor charts for Workload 1. Figures 5.22, 5.23, 5.24 and 5.25 illustrates latency and executor charts for Workload 2.

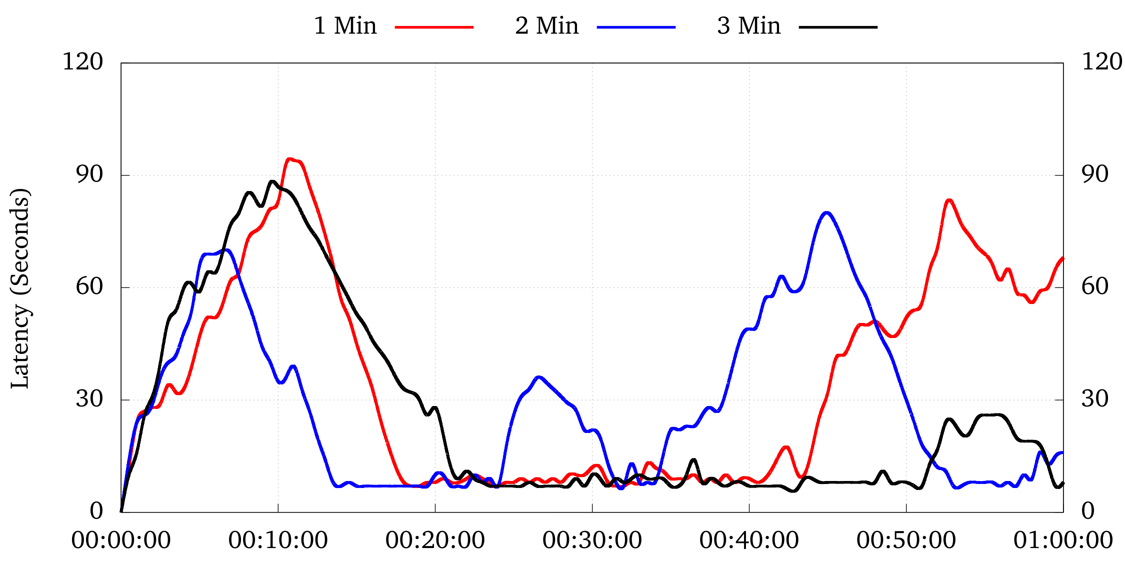


Figure 5.18: Decision Interval – Workload 1 – Latency

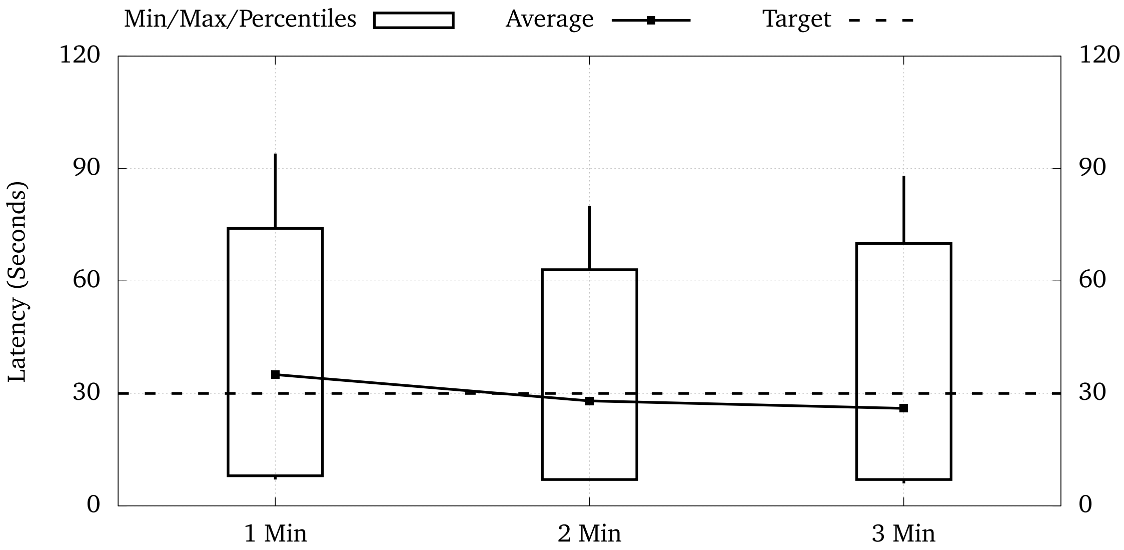


Figure 5.19: Decision Interval – Workload 1 – Latency

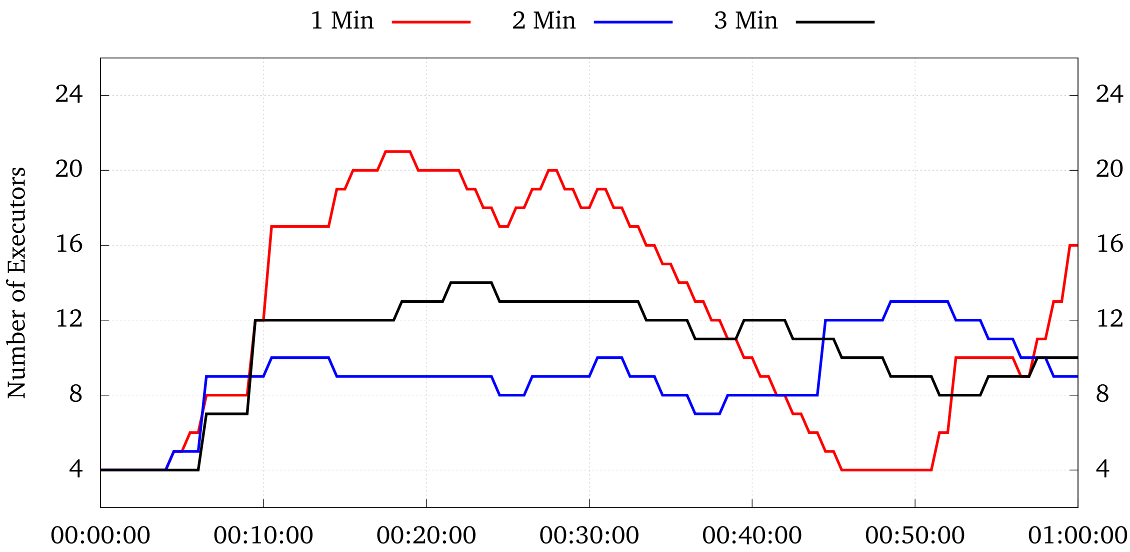


Figure 5.20: Decision Interval – Workload 1 – Number of Executors

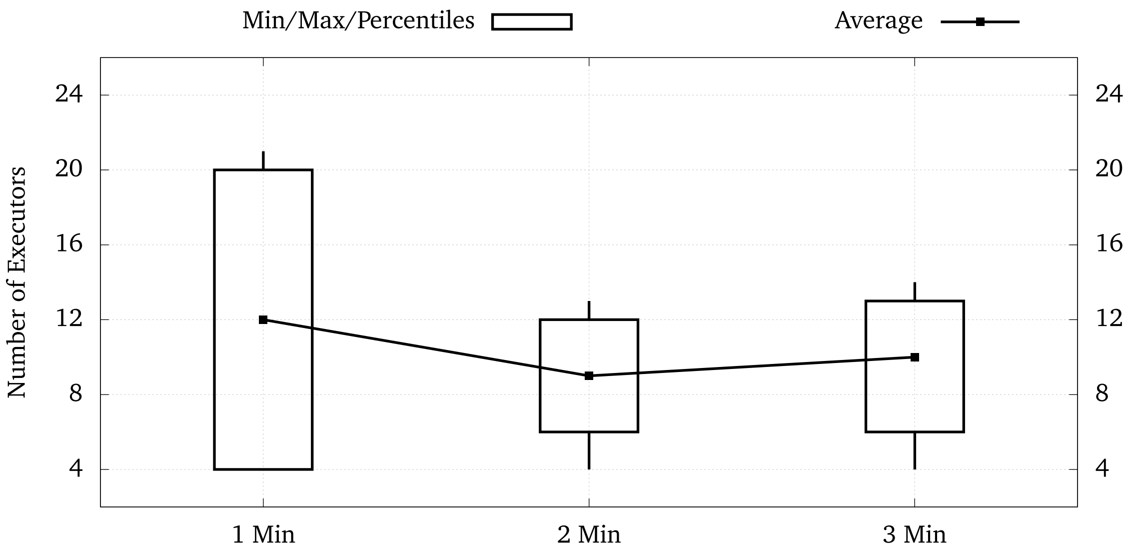


Figure 5.21: Decision Interval – Workload 1 – Number of Executors

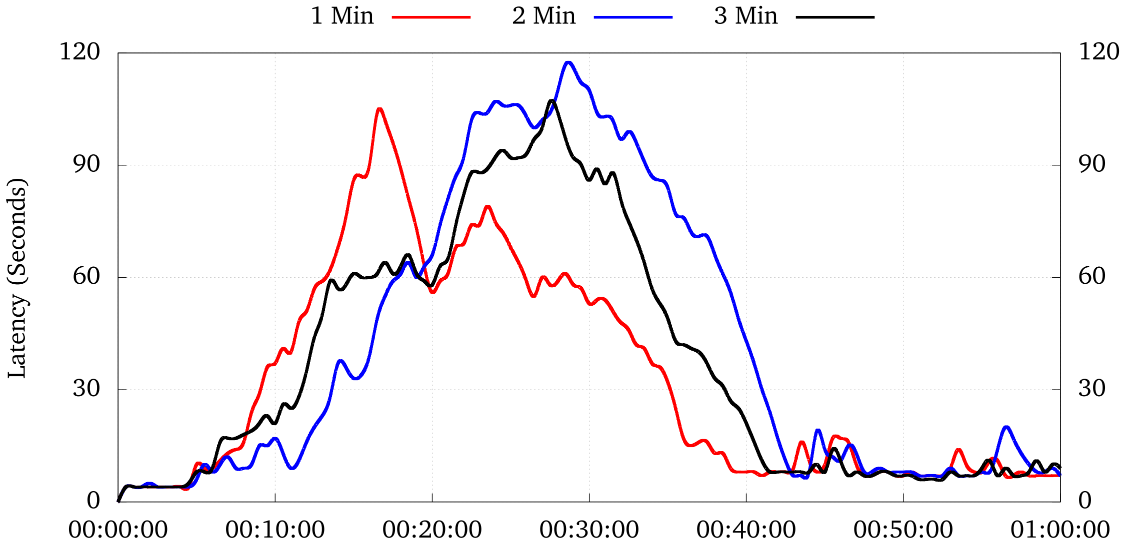


Figure 5.22: Decision Interval – Workload 2 – Latency

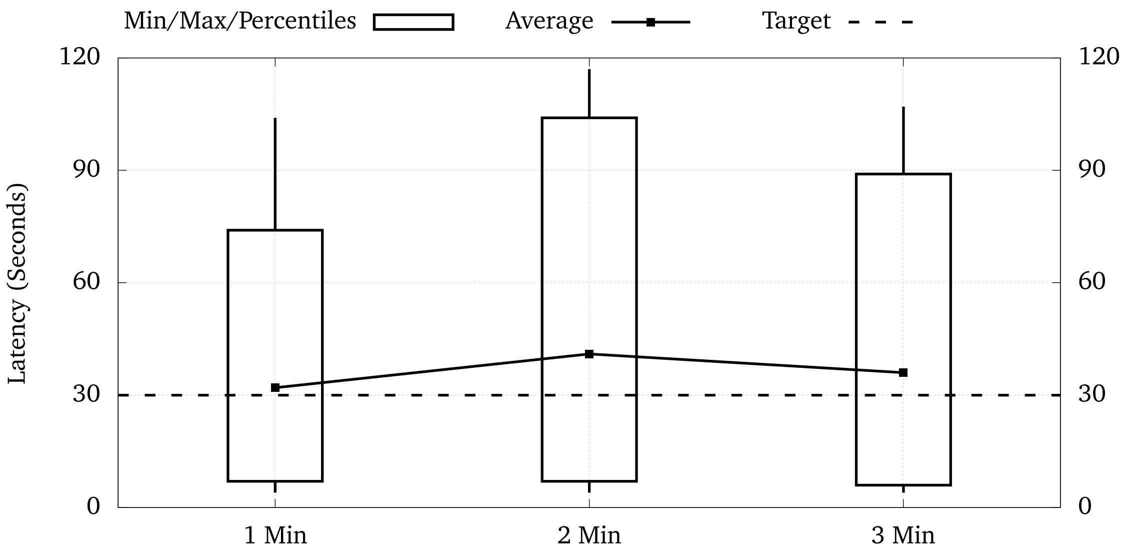


Figure 5.23: Decision Interval – Workload 2 – Latency

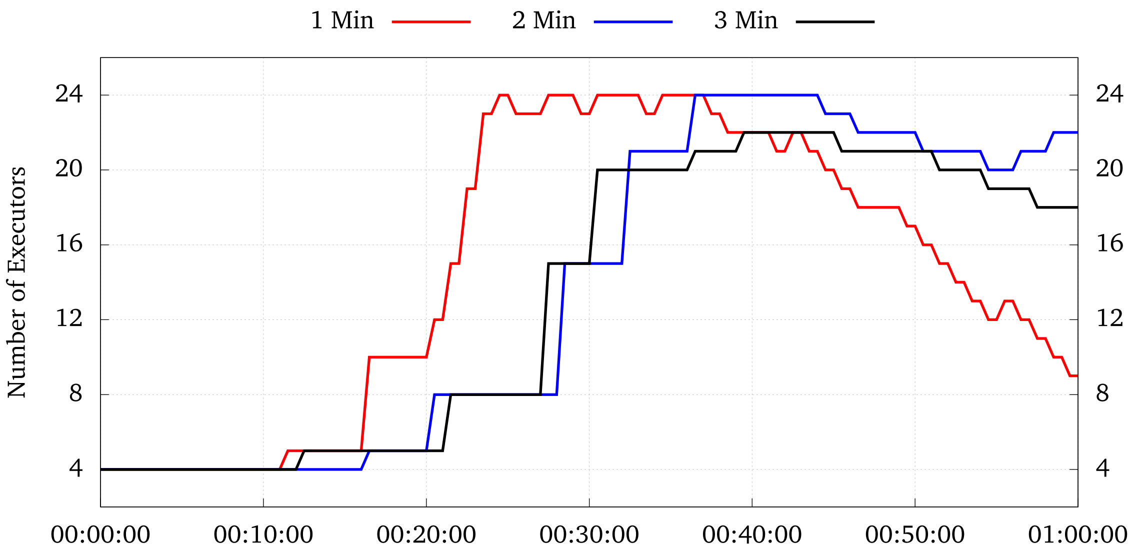


Figure 5.24: Decision Interval – Workload 2 – Number of Executors

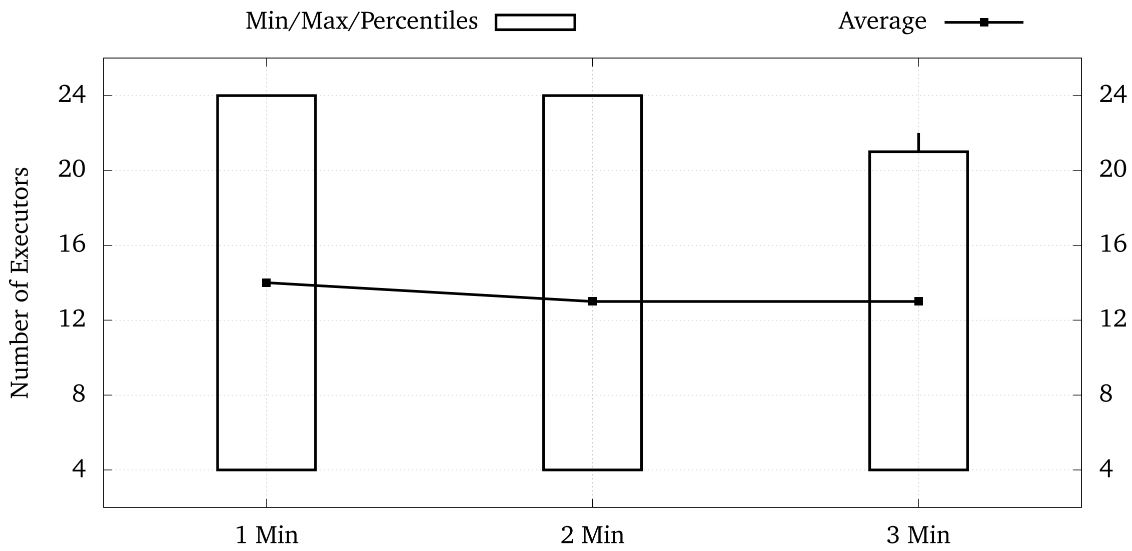


Figure 5.25: Decision Interval – Workload 2 – Number of Executors

5.4.1 Conclusion

For Workload 1, two and three-minute decision interval leads to better latency with much fewer executors (around half of one-minute decision interval) and they lead to even better latency than defined target latency. However, for second workload two and three-minute decision intervals have worse characteristics than one-minute window as they violate target latency more.

At this point selecting the best configuration becomes more difficult. Note that two and three-minute decision intervals perform extremely well for Workload 1 and poor for Workload 2. On the other hand one-minute decision interval performs reasonable in both workloads. Thus, this configuration is still preferred.

5.5 Experiment 4: State Space Initializer

As mentioned in Section 4, in order to speedup learning process, state space can be initialized using different techniques. This experiment has been designed to illustrate the effect of different state space initializers. In particular, it evaluates effectiveness of optimized initializer in comparison to random and zero initializers. Table 5.5 shows the configuration of this experiment. Note that, Queue Aware executor strategy is optimized to complement Auto-Scaler's decisions. In order to eliminate positive effect of Queue Aware strategy, Static strategy is used instead to better illustrate the negative effects of random actions.

#	Experiment	Configuration
1	Table Initializer: Optimized Policy: Greedy	Executor Granularity: 1 Executor Strategy: Static Target Latency: 30 Seconds Decision Interval: 1 Minute History Window: 2 Minutes Latency Granularity: 10 Seconds Learning Factor: 0.7 Discount Factor: 0.9 Reward: Prefer Scale-In When Load Is Decreasing
2	Table Initializer: Zero Policy: One Minus Epsilon Epsilon: 0.2	
3	Table Initializer: Random Policy: Greedy	

Table 5.5: State Space Initializer Configuration Parameters

Figures 5.26, 5.27, 5.28 and 5.29 illustrates latency and executor charts for Workload 1. Figures 5.30, 5.31, 5.32 and 5.33 illustrates latency and executor charts for Workload 2.

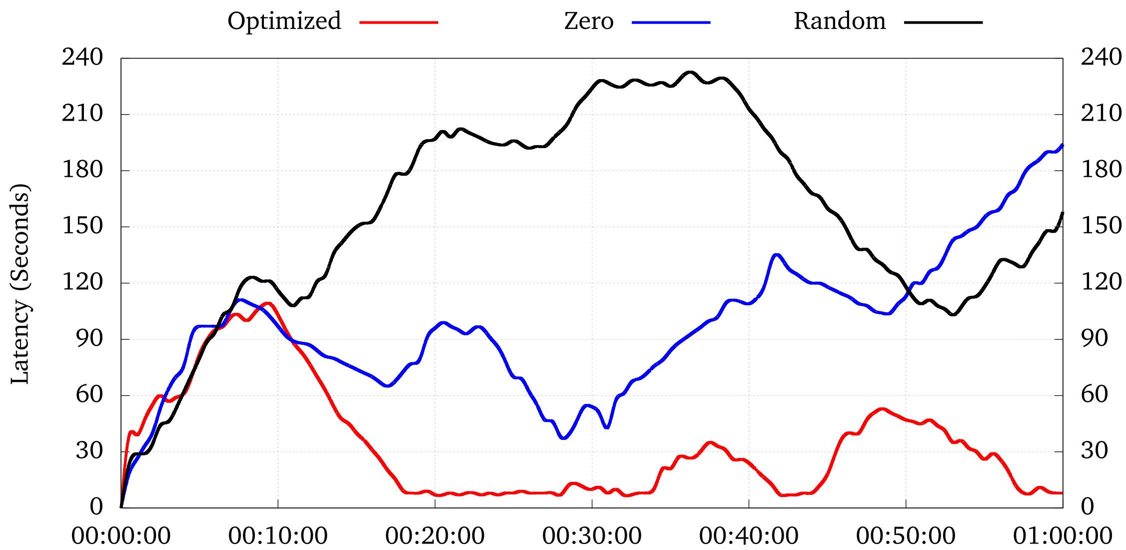


Figure 5.26: State Space Initializer – Workload 1 – Latency

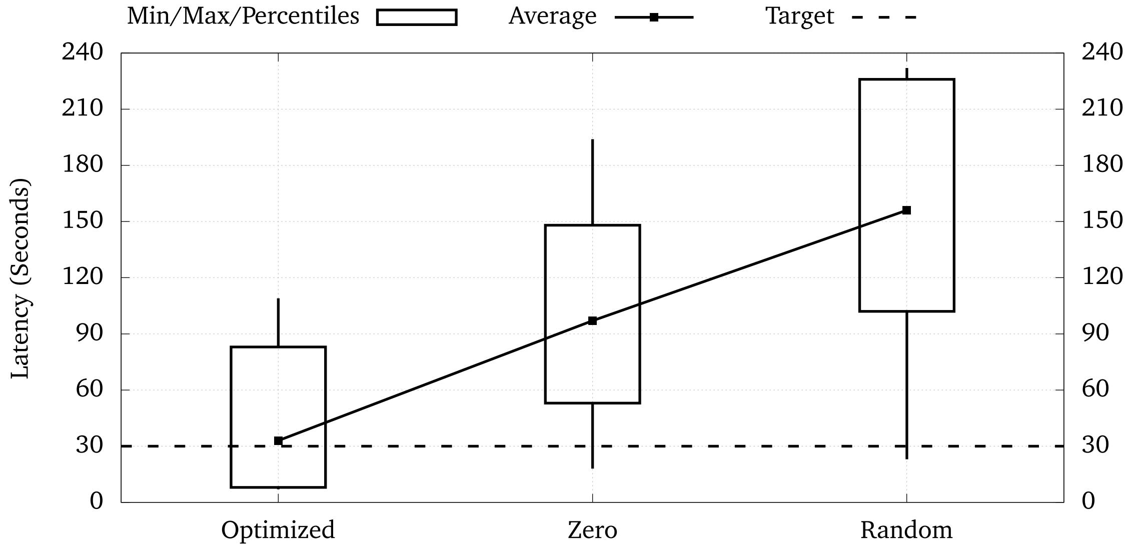


Figure 5.27: State Space Initializer – Workload 1 – Latency

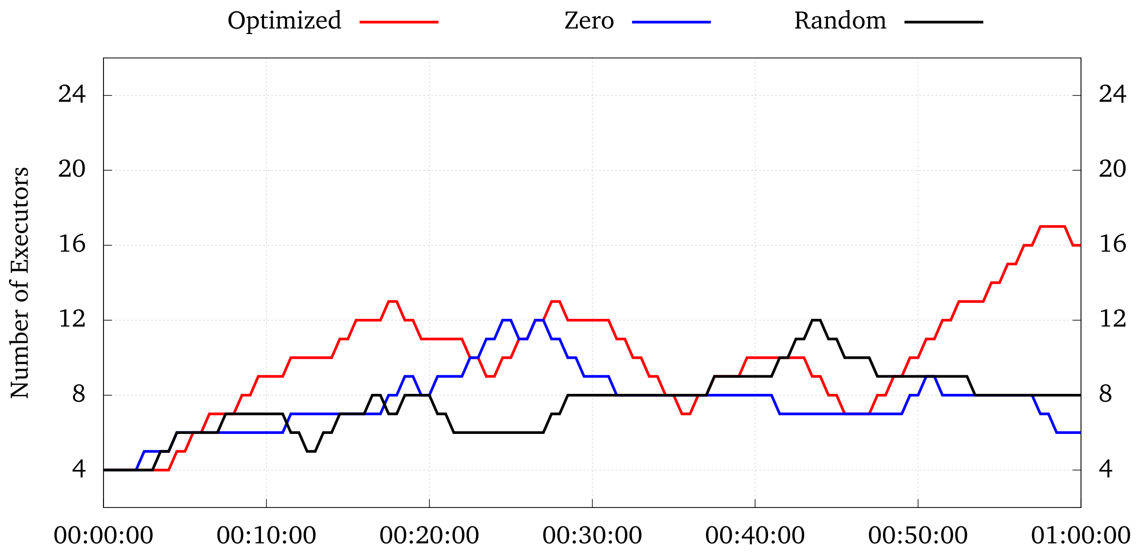


Figure 5.28: State Space Initializer – Workload 1 – Number of Executors

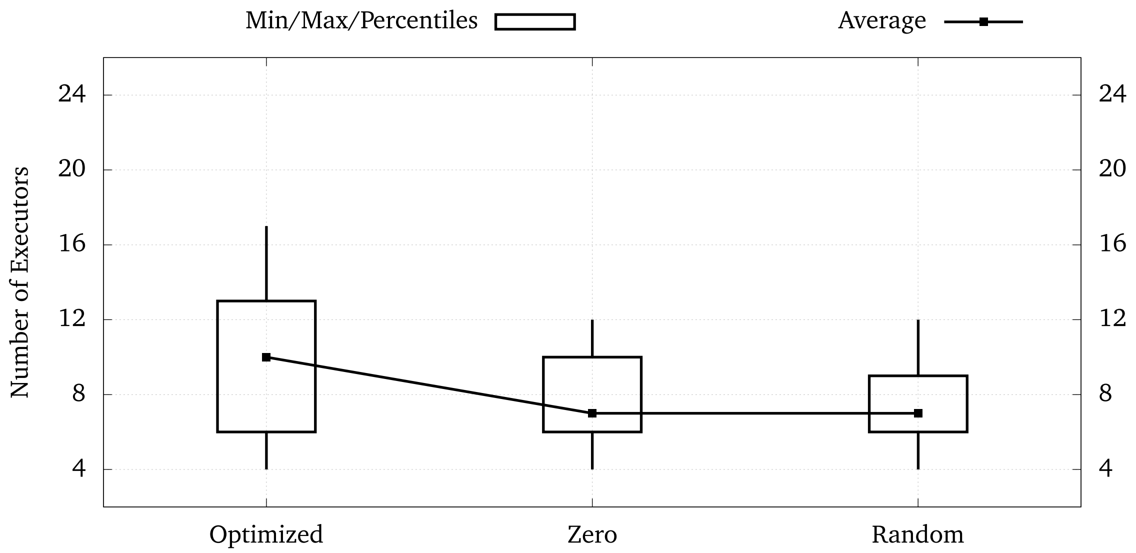


Figure 5.29: State Space Initializer – Workload 1 – Number of Executors

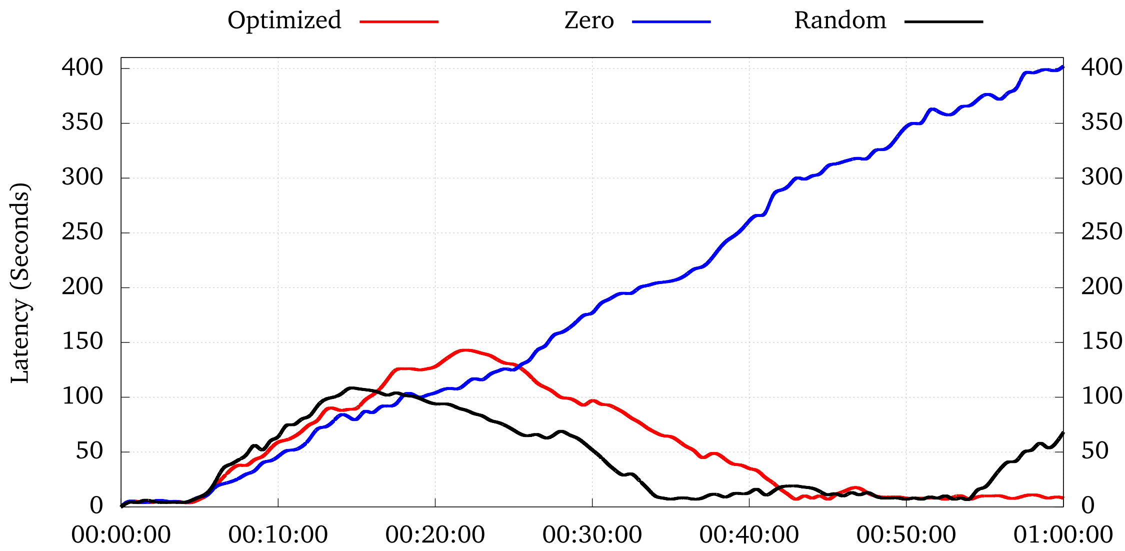


Figure 5.30: State Space Initializer – Workload 2 – Latency

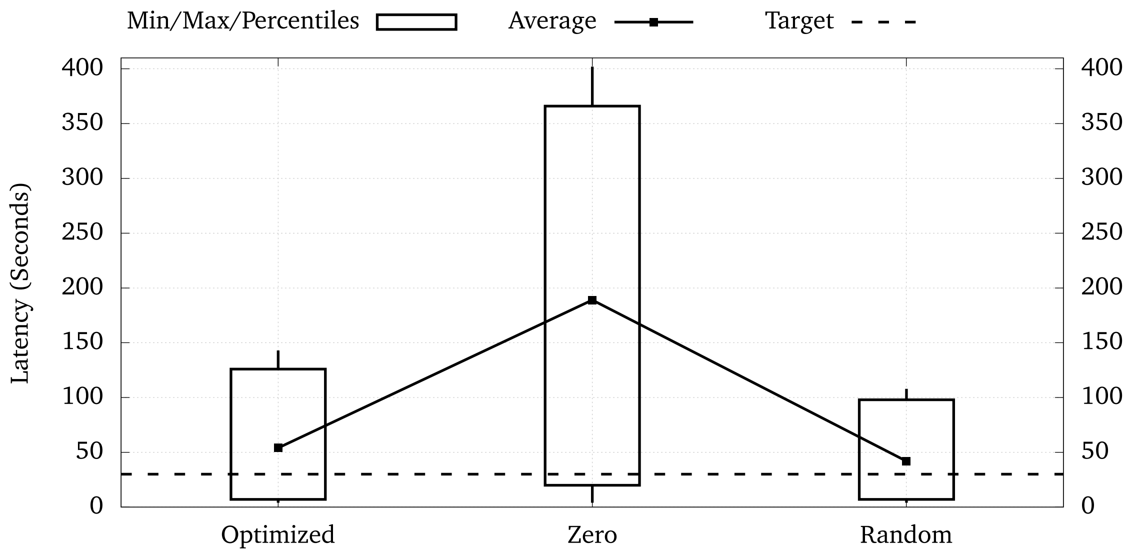


Figure 5.31: State Space Initializer – Workload 2 – Latency

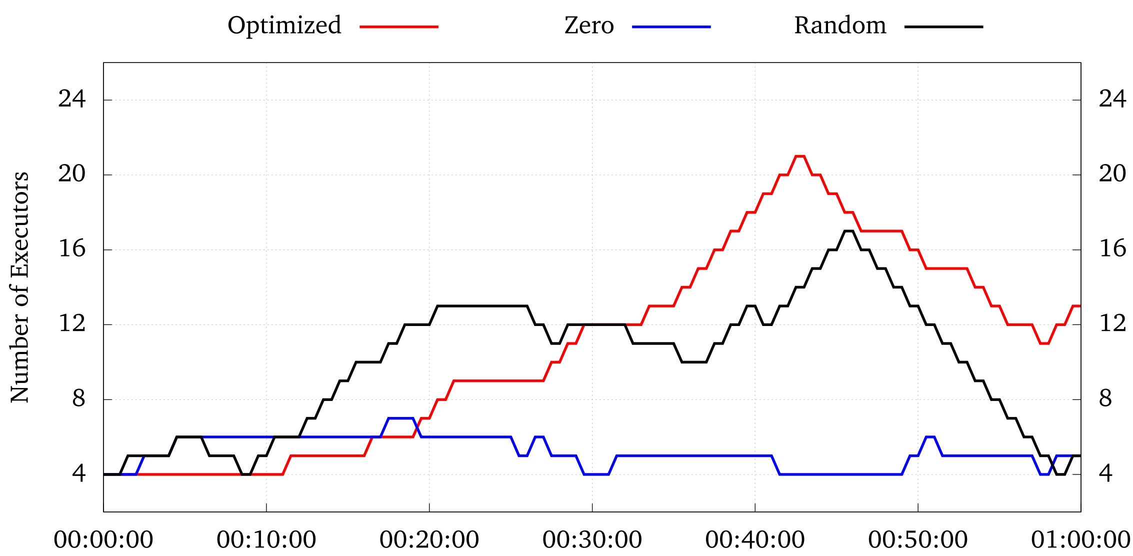


Figure 5.32: State Space Initializer – Workload 2 – Number of Executors

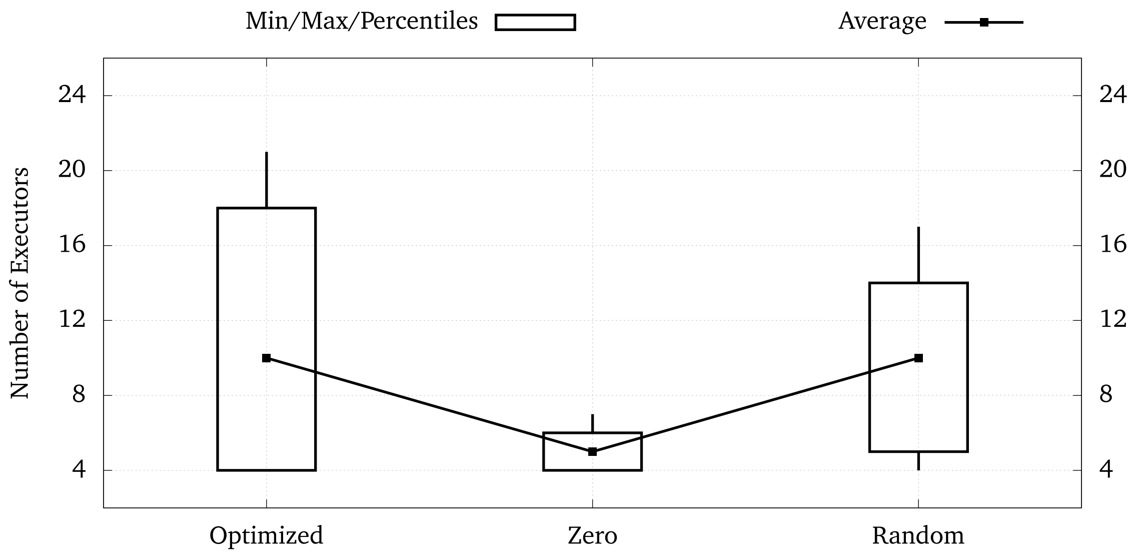


Figure 5.33: State Space Initializer – Workload 2 – Number of Executors

5.5.1 Conclusion

For Workload 1 as depicted, optimized initializer beats zero and random initializers and to a large degree it is able to respect target latency. For Workload 2, random initializer performs similar to optimized initializer. But this is only achieved by chance. Looking at Figure 5.32 reveals two pieces of information.

- Since the first part of Workload 2 is not intensive and Auto-Scaler is running with minimum executors, random initializer performs early Scale-Out – there is no other option with minimum executors. Since this action helps to improve the latency, Auto-Scaler learns this bit of information and performs another Scale-Out which in turn improves latency further. Up until the workload reaches critical point – in terms of latency – Auto-Scaler has already performed consecutive Scale-Out actions which helps to diverge faster. Note that if the workload begins with a peak – just like Workload 1, there is little chance that random initializer could win the game.
- Considering final stage of workload, where Auto-Scaler has removed so many executors that the latency started to increase fast – Figure 5.30 – which indicates state space still contains wrong information. Note that optimized initializer removes executors at a slower rate.

Looking at behavior of zero initializer reveals an interesting point. Since Auto-Scaler has started with minimum executors, it wasn't able to process the workload. On the other hand all states are initialized to zero. Thus, Auto-Scaler will stick to No-Action – with a probability of ϵ will perform random action. Staying with No-Action leads to even higher latency which is a new state initialized with zero. Again, Auto-Scaler will stick to No-Action which in turn increases latency and causes Auto-Scaler to land in another new state. This process keeps going – latency increases leading Auto-Scaler to land in a new state. This means, Auto-Scaler never sees the previous states to utilize learned information. This is the point where Queue-Aware executor strategy could potentially help to complement Auto-Scaler's actions. In case Auto-Scaler is behind schedule and by any chance takes a random Scale-Out action, Queue-Aware strategy adds a batch of executors at once which would considerably improve latency.

In general, if an Auto-Scaler starts with zero or random initializer, it is better to start with maximum – or at least reasonable – number of executors such that initial random actions don't cause serious SLO violations.

As mentioned in previous chapters, A Q-Learning agent learns new information and merges it with its own knowledge. The speed of learning is controlled by learning factor which defined by `learningFactor` in this thesis. This experiment has been designed to show the effect of different learning factor values. Note that learning factor affects Auto-Scaler's behavior in long run. Thus, for a 1-Hour experiment extreme values should be chosen to make difference observable. Furthermore, since there is no random action involved in this experiment, Queue Aware executor strategy can be used again. Table 5.6 describes the configuration parameters of this experiment.

#	Experiment	Configuration
1	Learning Factor = 0.7	Executor Granularity: 1 Executor Strategy: Queue Aware Table Initializer: Optimized Target Latency: 30 Seconds Decision Interval: 1 Minute History Window: 2 Minutes Latency Granularity: 10 Seconds Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Learning Factor = 0.07	
3	Learning Factor = 0.007	

Table 5.6: Learning Factor Configuration Parameters

Figures 5.34, 5.35, 5.36 and 5.37 illustrates latency and executor charts for Workload 1. Figures 5.38, 5.39, 5.40 and 5.41 illustrates latency and executor charts for Workload 2.

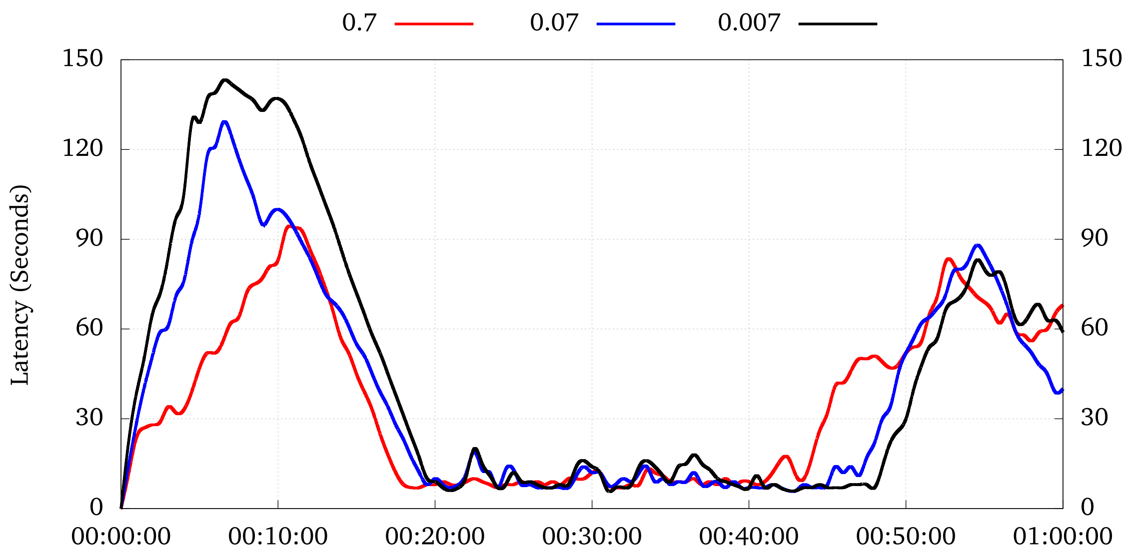


Figure 5.34: Learning Factor – Workload 1 – Latency

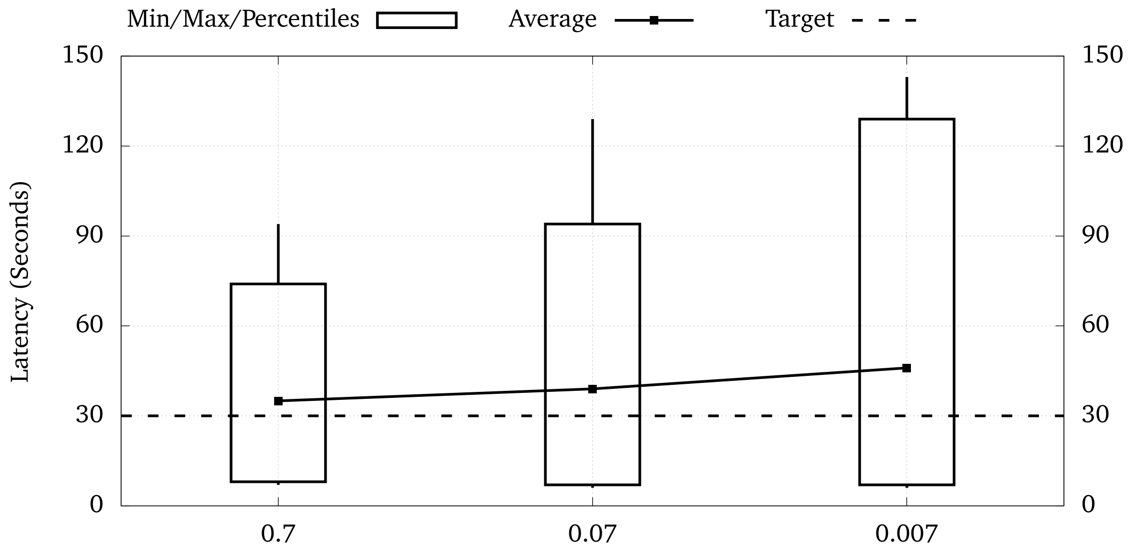


Figure 5.35: Learning Factor – Workload 1 – Latency

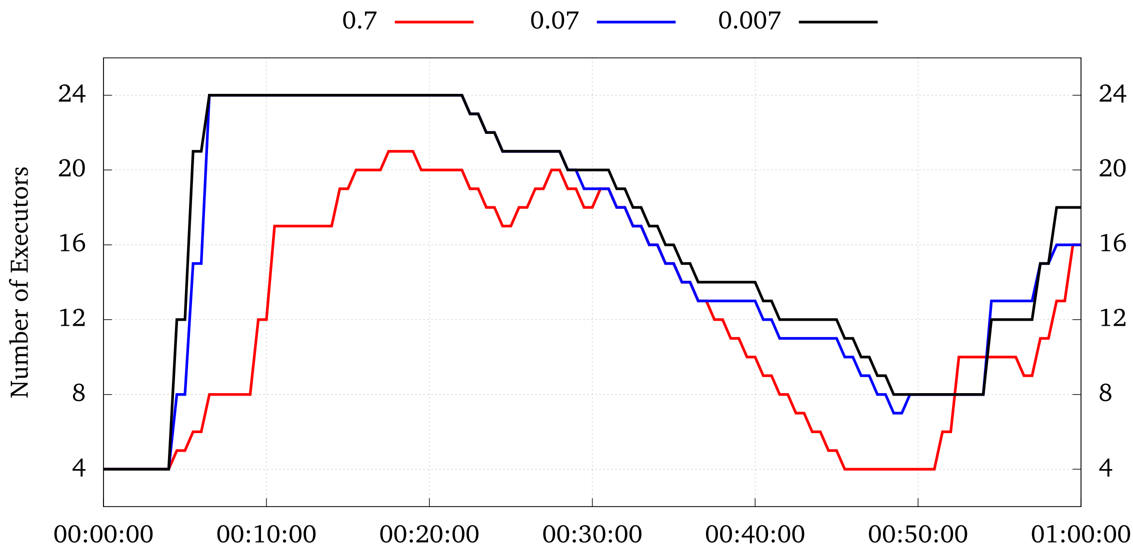


Figure 5.36: Learning Factor – Workload 1 – Number of Executors

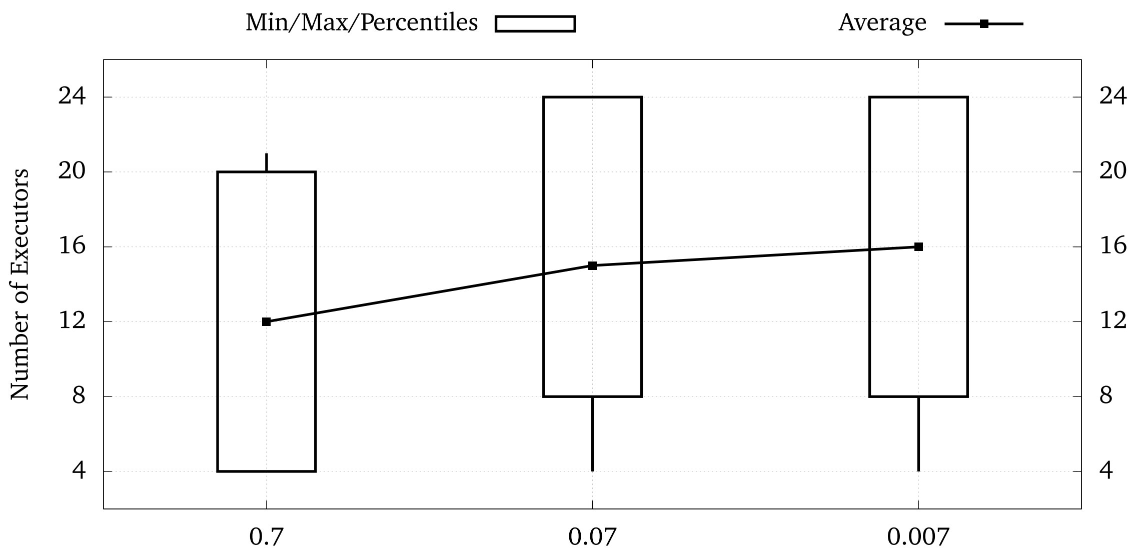


Figure 5.37: Learning Factor – Workload 1 – Number of Executors

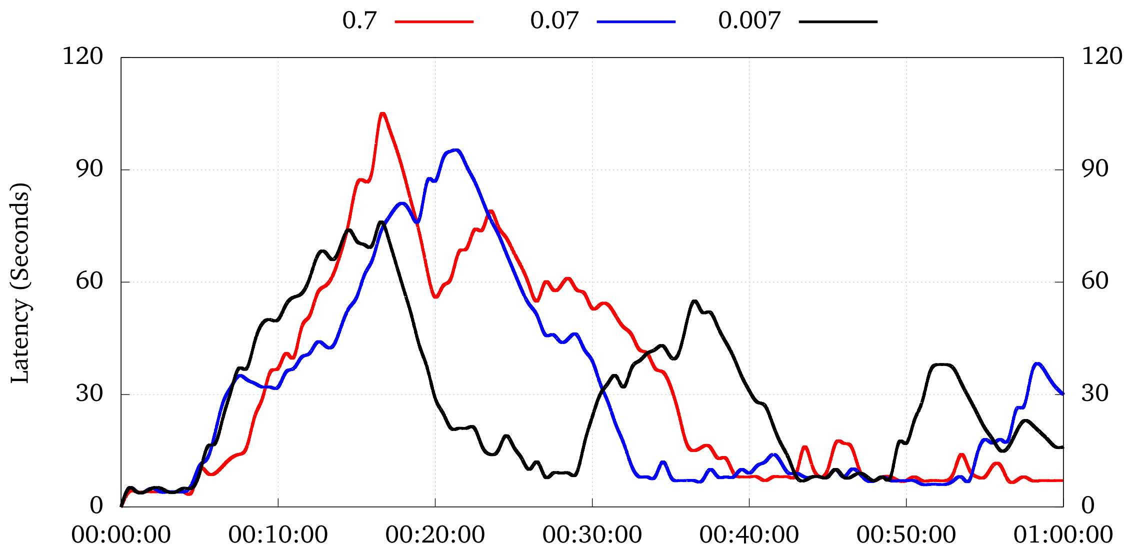


Figure 5.38: Learning Factor – Workload 2 – Latency

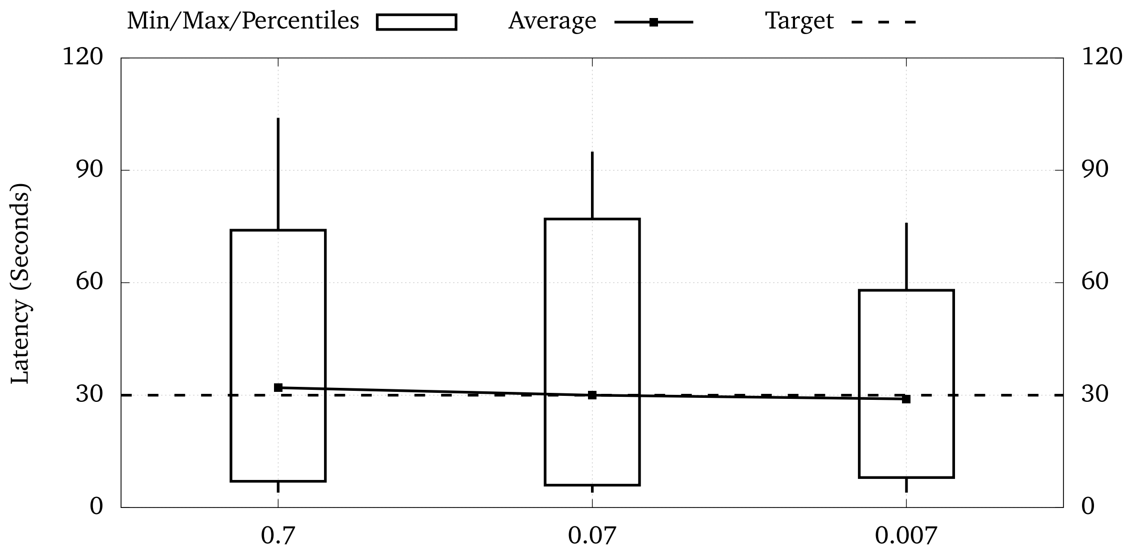


Figure 5.39: Learning Factor – Workload 2 – Latency

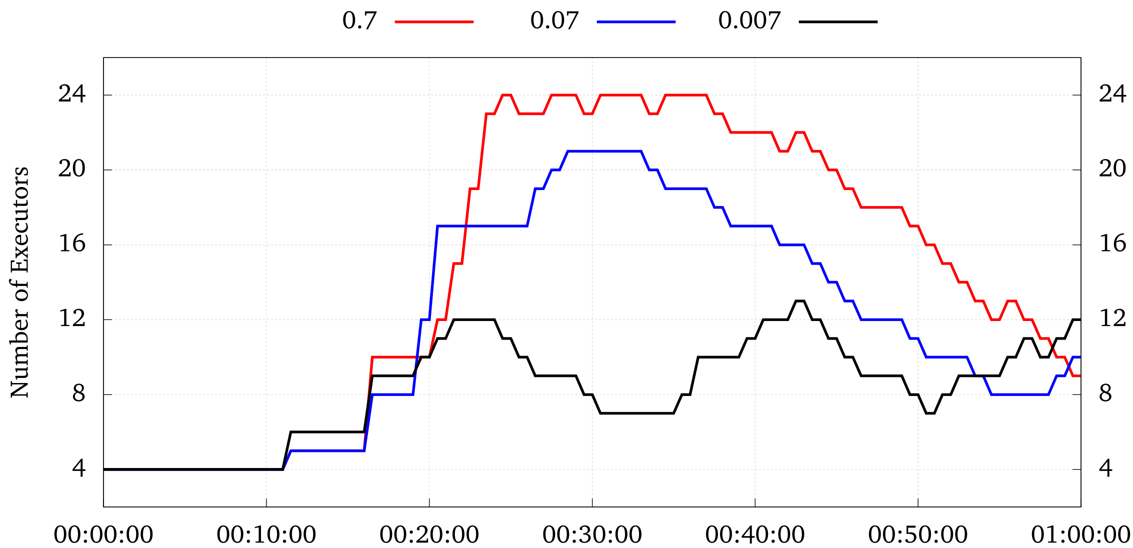


Figure 5.40: Learning Factor – Workload 2 – Number of Executors

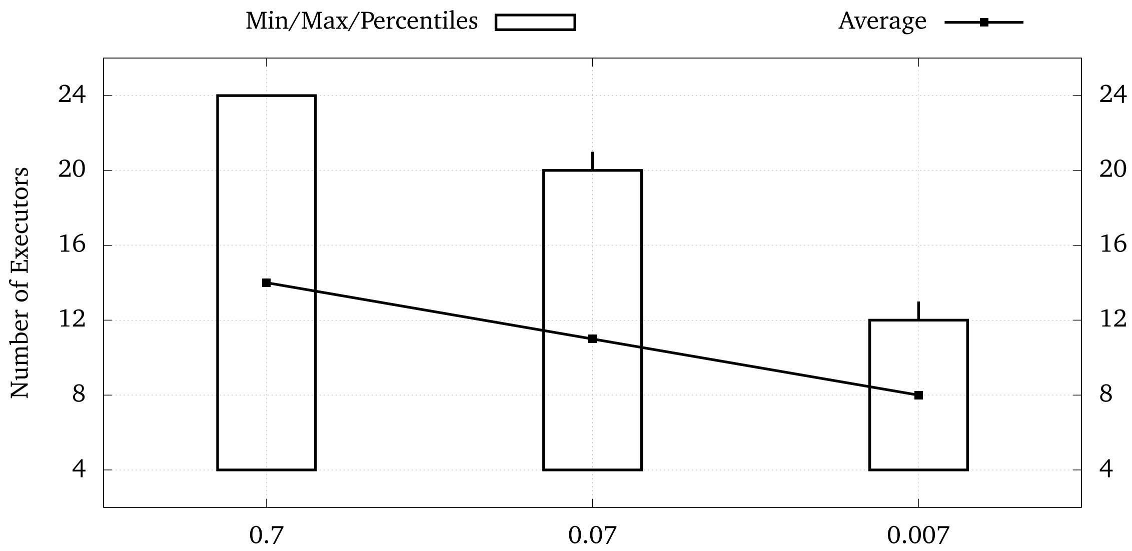


Figure 5.41: Learning Factor – Workload 2 – Number of Executors

5.6.1 Conclusion

For Workload 1, learning factor of 0.7 achieves a better result both in terms of latency and number of executors. It is tempting to conclude that for unpredictable workloads, it is better to apply a higher learning factor.

But the above conclusion doesn't hold when we look at the result of Workload 2. In terms of latency, 0.007 is slightly below target latency which is better than 0.7 and 0.07. In terms of number of executors, it achieves an outstanding result of 8 executors on average. Combining latency and number of executors this is the best result that has been achieved for Workload 2.

It seems that the effect of learning factor is totally workload dependent and no generalized conclusion can be made based on these two workloads.

5.7 Experiment 6: Discretization

This experiment has been designed to evaluate the effect of discretization on Auto-Scaler's decisions. Discretization is applied to both latency and incoming messages. Table 5.7 describes the configuration parameters of this experiment.

#	Experiment	Configuration
1	Latency Granularity = 5 Seconds	Executor Granularity: 1 Executor Strategy: Queue Aware Table Initializer: Optimized Target Latency: 30 Seconds Decision Interval: 1 Minute History Window: 2 Minutes Learning Factor = 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Latency Granularity =10 Seconds	
3	Latency Granularity = 15 Seconds	

Table 5.7: Discretization Configuration Parameters

Figures 5.42, 5.43, 5.44 and 5.45 illustrates latency and executor charts for Workload 1. Figures 5.46, 5.47, 5.48 and 5.49 illustrates latency and executor charts for Workload 2.

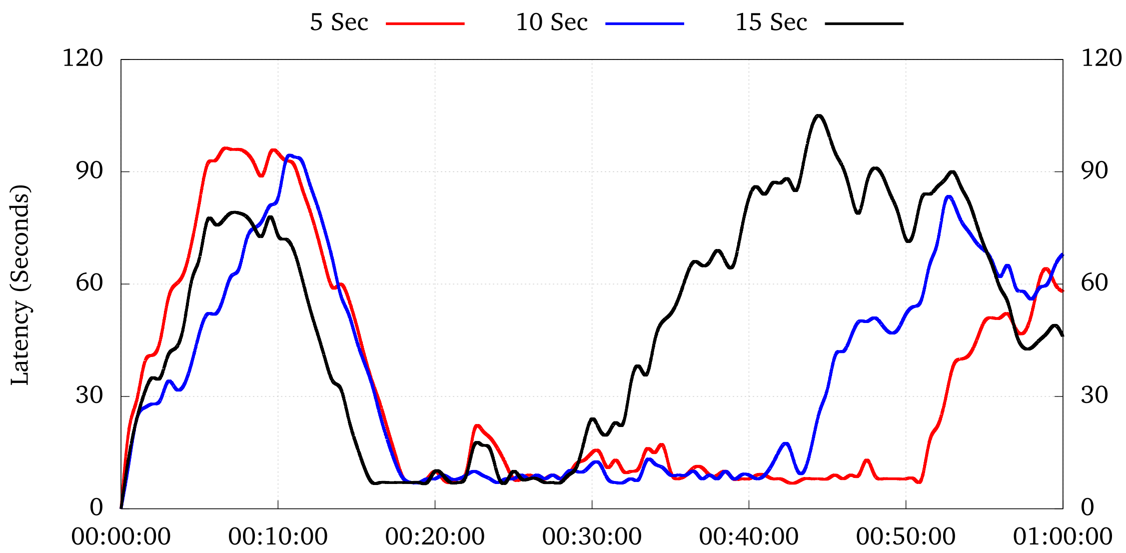


Figure 5.42: Discretization – Workload 1 – Latency

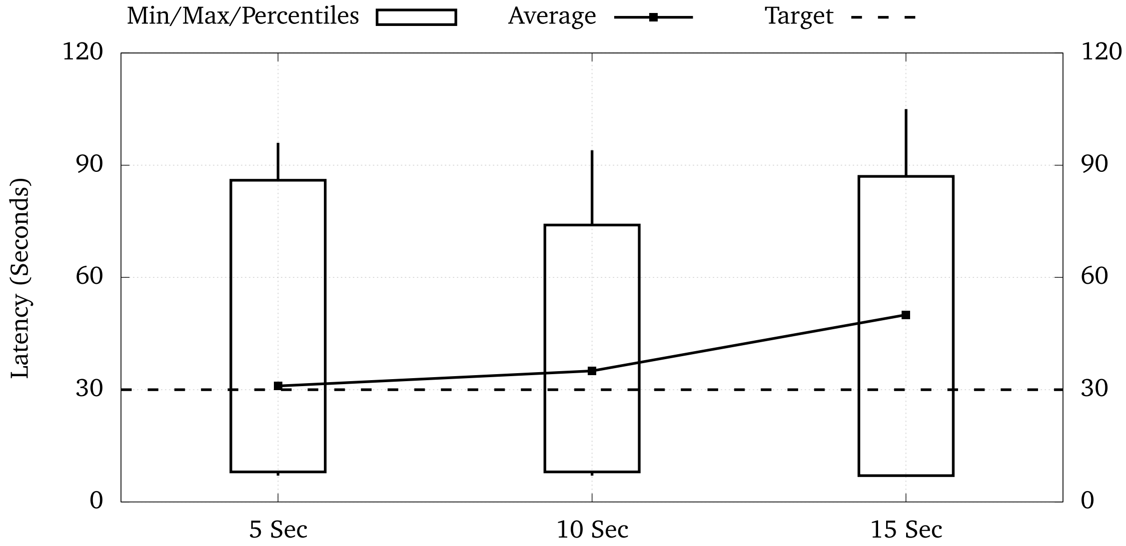


Figure 5.43: Discretization – Workload 1 – Latency

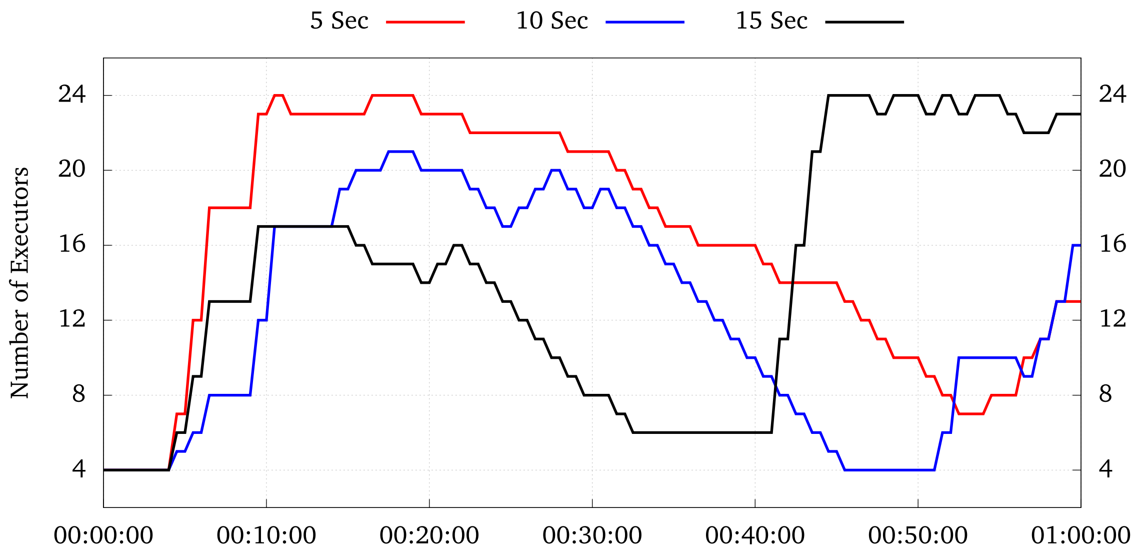


Figure 5.44: Discretization – Workload 1 – Number of Executors

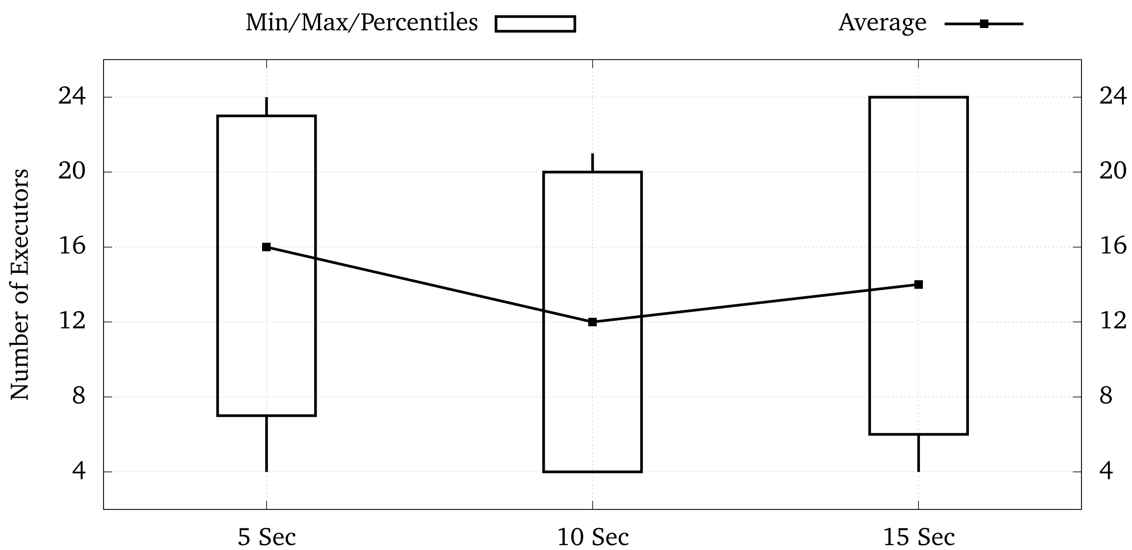


Figure 5.45: Discretization – Workload 1 – Number of Executors

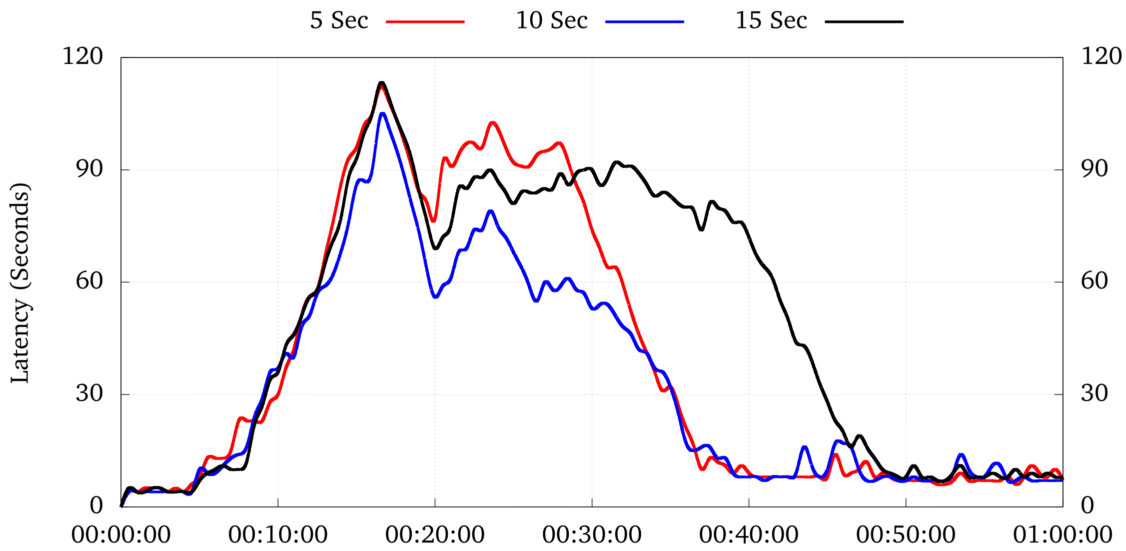


Figure 5.46: Discretization – Workload 2 – Latency

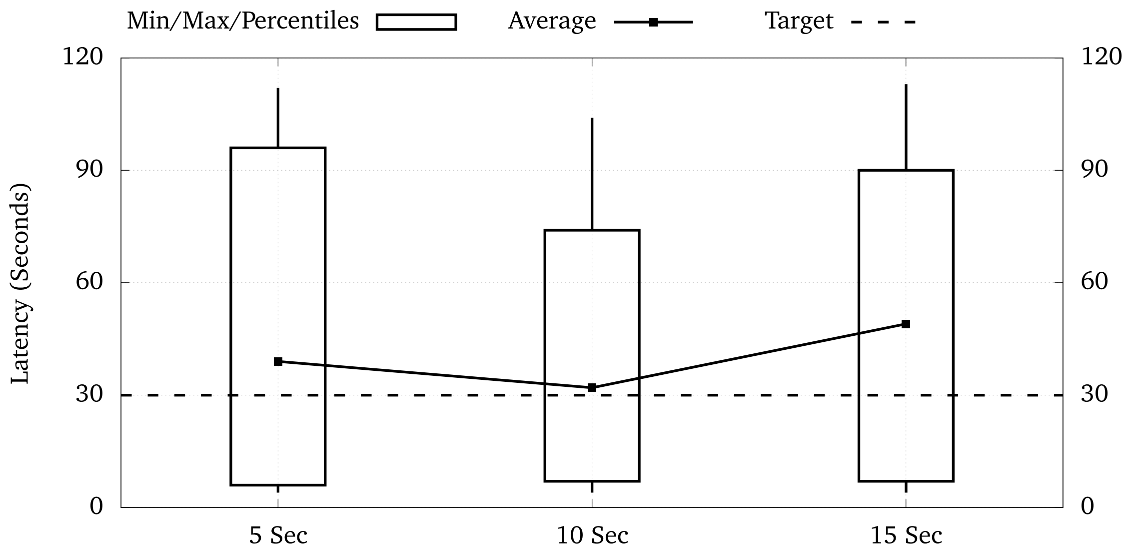


Figure 5.47: Discretization – Workload 2 – Latency

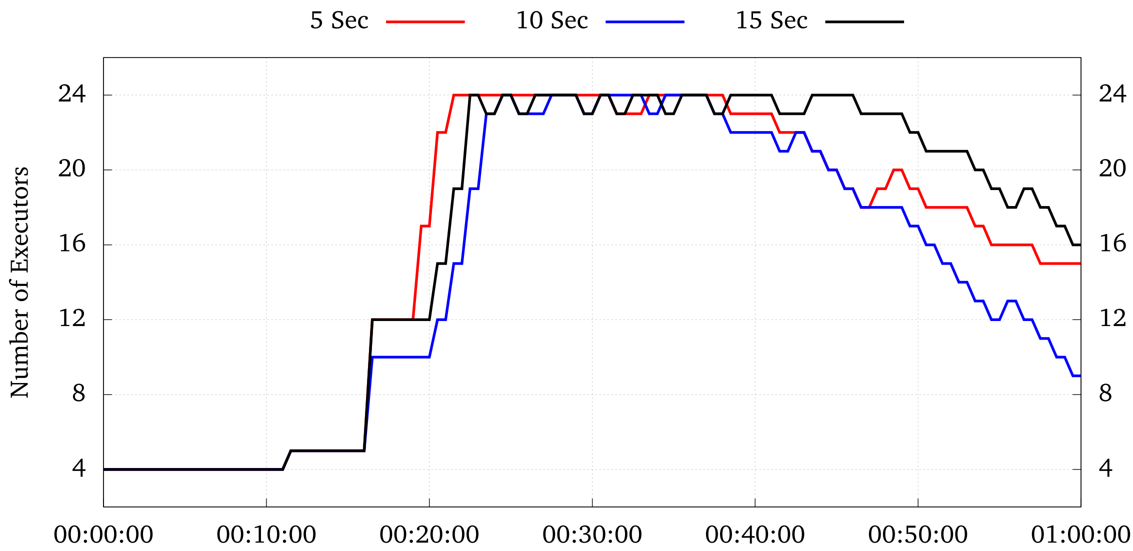


Figure 5.48: Discretization – Workload 2 – Number of Executors

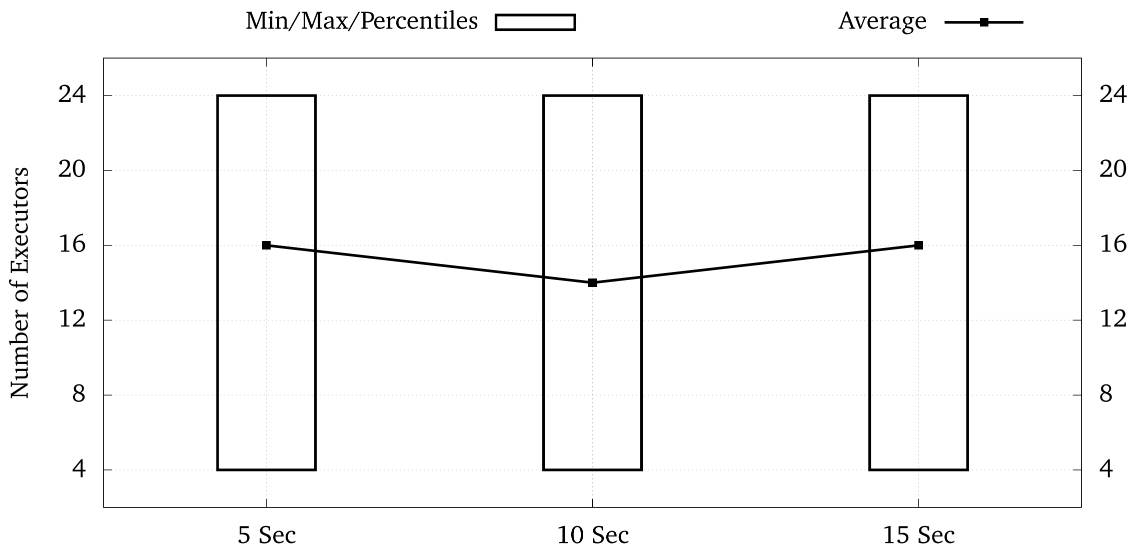


Figure 5.49: Discretization – Workload 2 – Number of Executors

5.7.1 Conclusion

For Workload 1, 5-second discretization performs better in terms of latency albeit with more executors. For Workload 2, 10-second discretization performs better both in terms of latency and number of executors.

Although 10-second discretization performs reasonably well in both workloads, we are still not sure that it is the optimal discretization. Similar to learning factor, discretization seems to be workload dependent and no generalized conclusion can be made.

5.8 Experiment 7: Value Iteration

Dutreilh et al. [22] claimed that Value Iteration algorithm helps to initialize the state table and speedup learning process. Authors conducted an experiment which was based on a sinusoidal workload. Agent is trained in first period of rotation and then standard Q-Learning is used for next rounds. However, training based on sinusoidal workload is not so challenging. In this experiment the agent is trained using a different data set than the actual workload. State space is initialized after 5, 10 and 20 minutes of training period and then Auto-Scaler's behavior is evaluated under both workloads. During training `decreasingOneMinusEpsilon` policy is used to help agent discover the environment. However, during the evaluation greedy policy is used. Table 5.8 describes the configuration parameters of this experiment.

#	Experiment	Configuration
1	Learning Period = 5 Minutes	Executor Granularity: 1 Executor Strategy: Queue Aware Table Initializer: Zero Target Latency: 30 Seconds Decision Interval: 1 Minute History Window: 2 Minutes Learning Factor = 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Learning Period =10 Minutes	
3	Learning Period = 20 Minutes	

Table 5.8: Value Iteration Configuration Parameters

Figures 5.50, 5.51, 5.52 and 5.53 illustrates latency and executor charts for Workload 1. Figures 5.54, 5.55, 5.56 and 5.57 illustrates latency and executor charts for Workload 2.

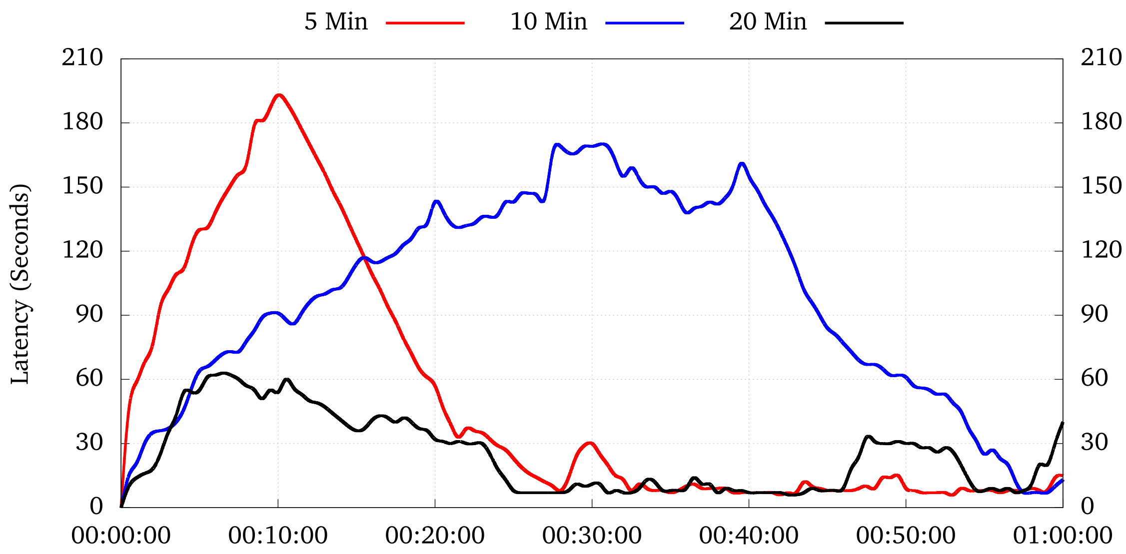


Figure 5.50: Value Iteration – Workload 1 – Latency

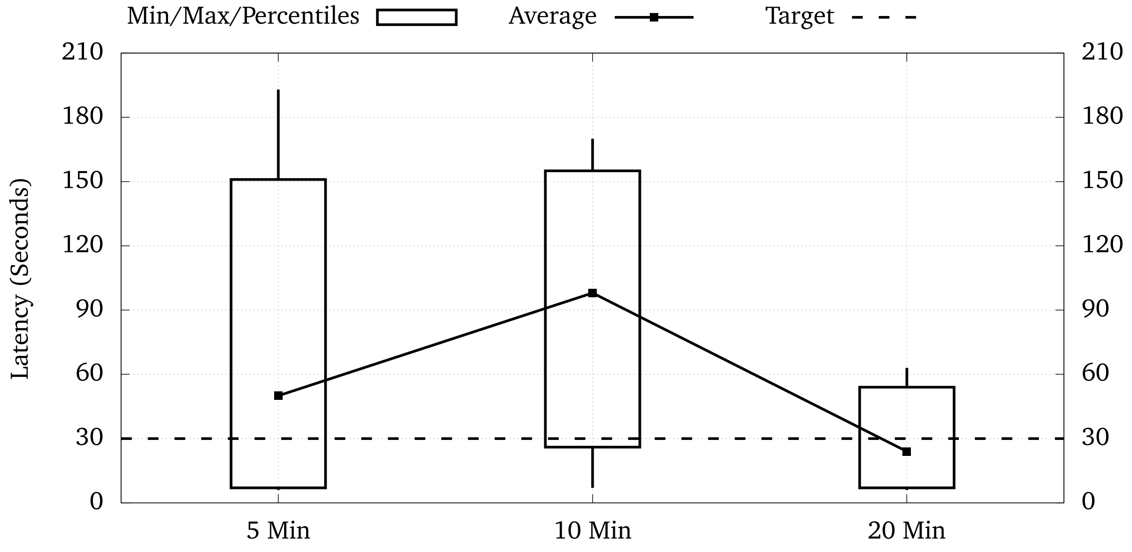


Figure 5.51: Value Iteration – Workload 1 – Latency

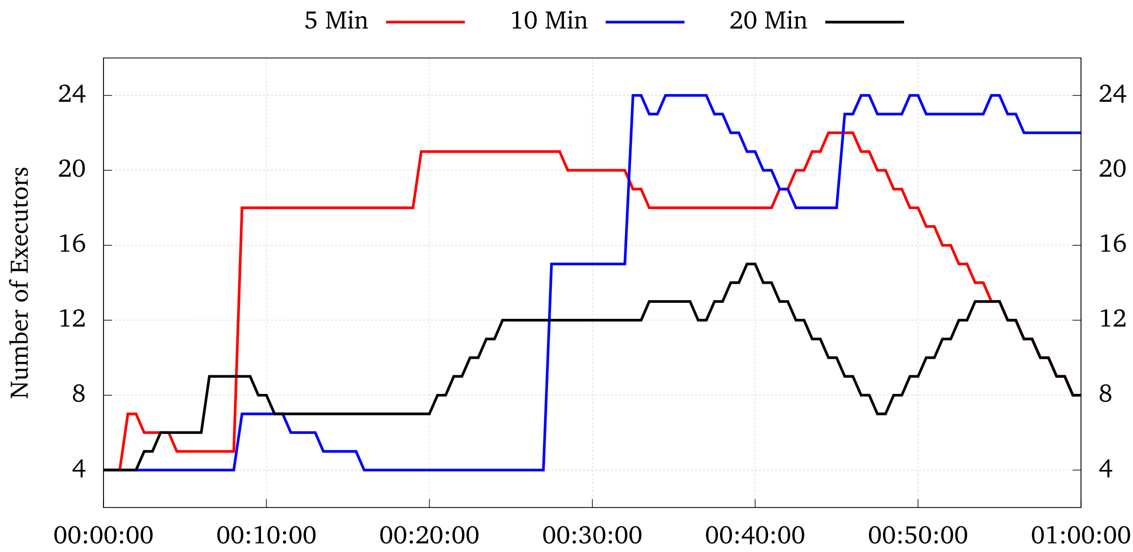


Figure 5.52: Value Iteration – Workload 1 – Number of Executors

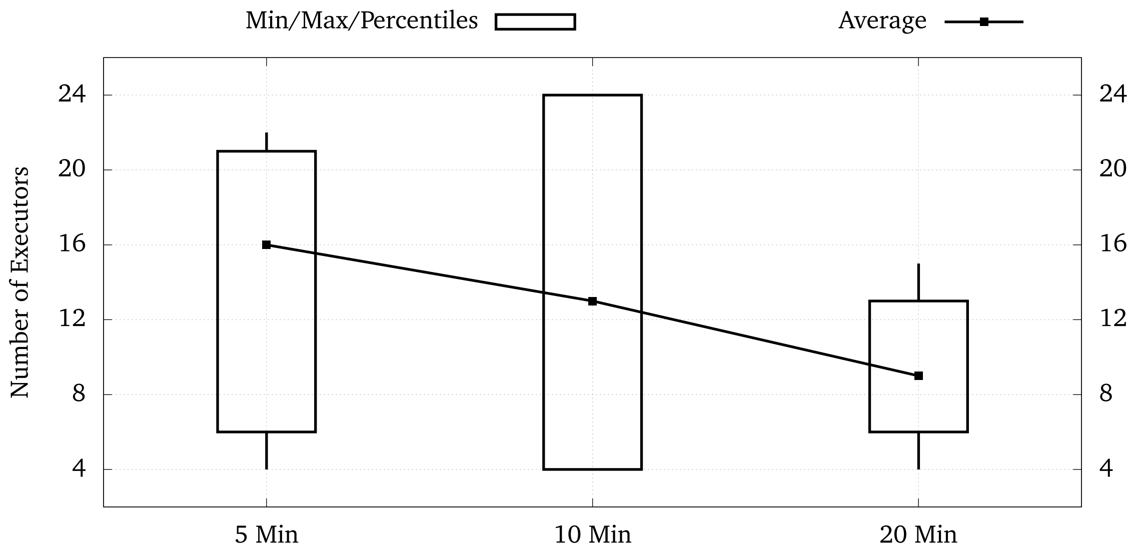


Figure 5.53: Value Iteration – Workload 1 – Number of Executors

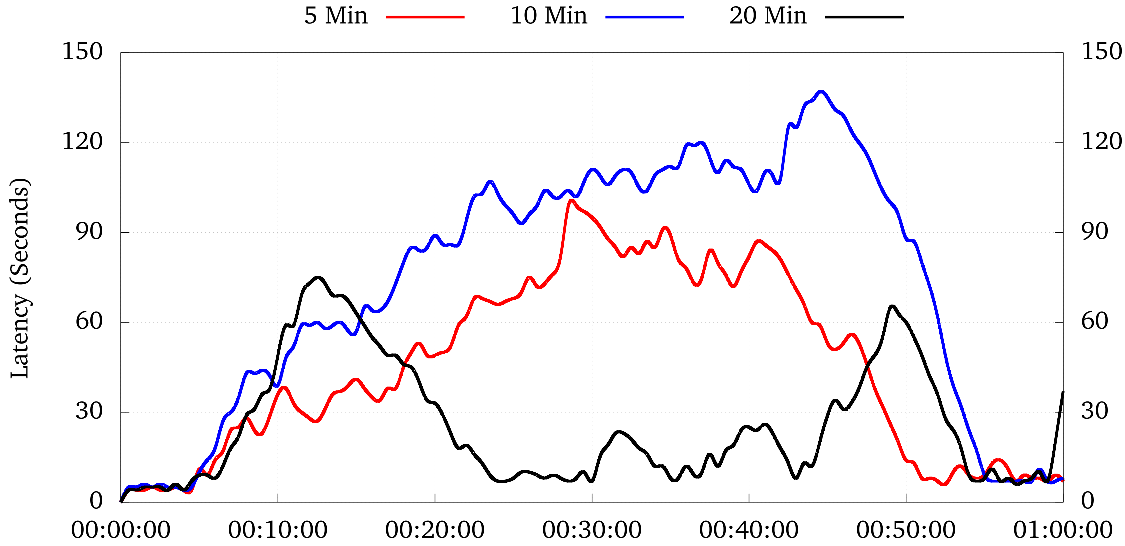


Figure 5.54: Value Iteration – Workload 2 – Latency

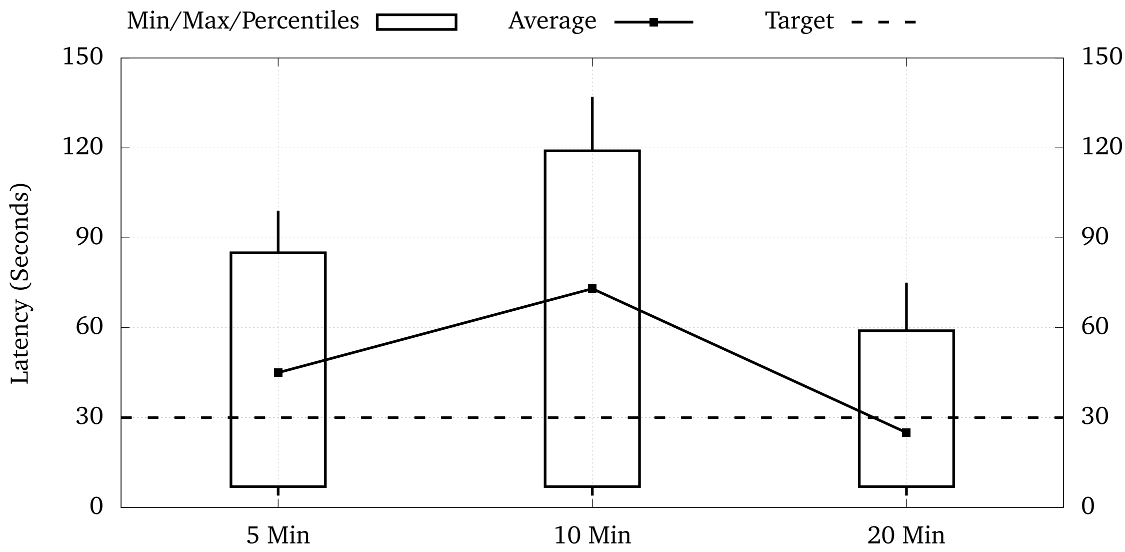


Figure 5.55: Value Iteration – Workload 2 – Latency

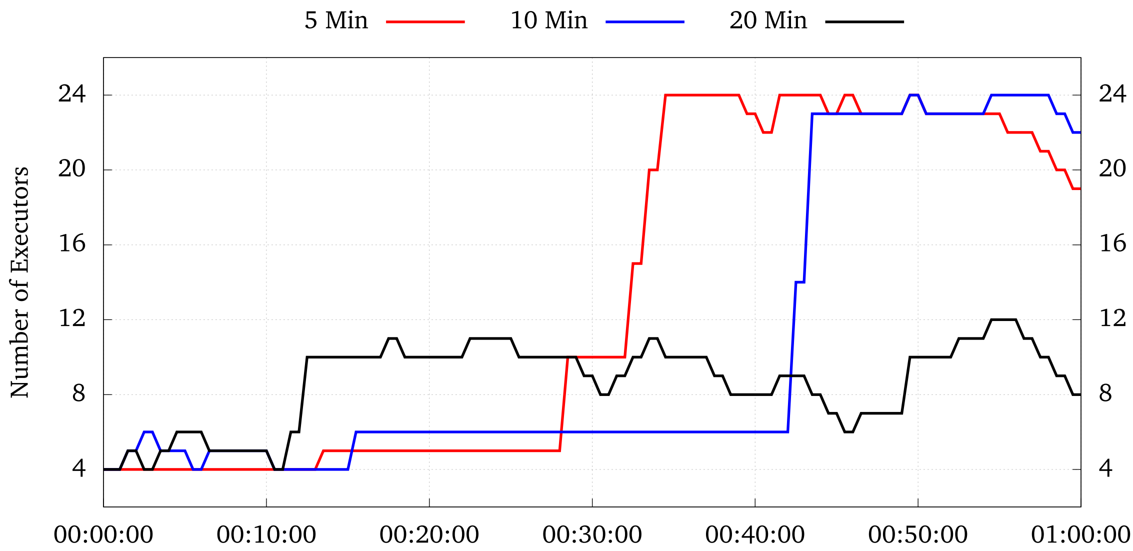


Figure 5.56: Value Iteration – Workload 2 – Number of Executors

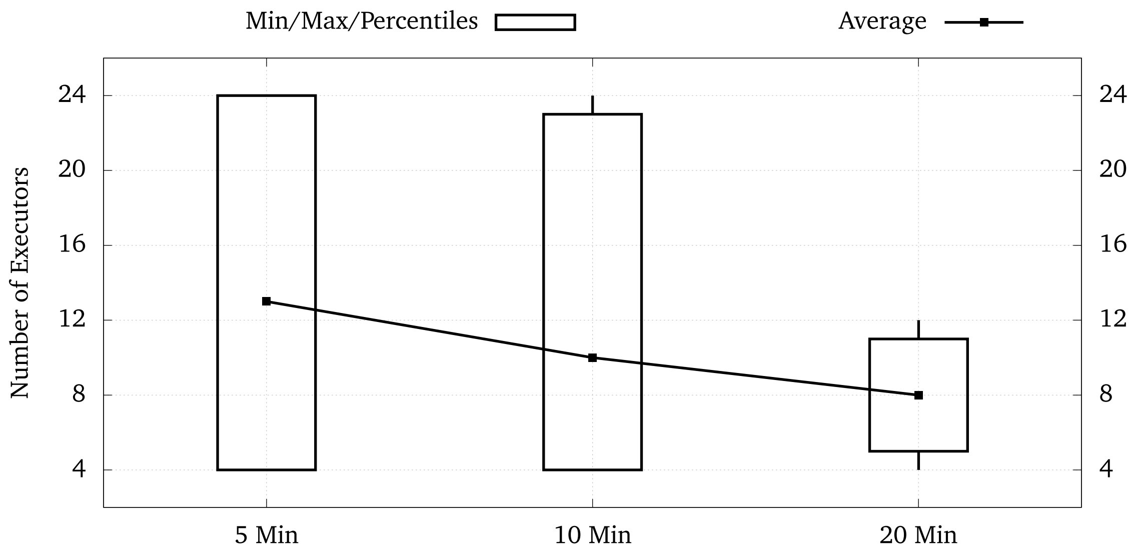


Figure 5.57: Value Iteration – Workload 2 – Number of Executors

5.8.1 Conclusion

Looking at the results and combining them with the results of Dutreilh et al. [22] reveals a couple of interesting points.

- If the sample data set is a good representative of actual workload, then Value Iteration is quite useful.
- *The more training, the better outcome* does not hold for streaming workloads. Since it is possible to introduce wrong information into state space at any time. As can be seen from this experiment, ten minute training was worse than five minute training.
- Learning or training period plays a critical role.

The major question that arises is *what is the optimal period of training?* Since the workload is changing in an unpredictable manner, finding a near optimal learning period is extremely difficult and workload dependent. No generalized conclusion can be made regarding learning period by looking at results. It may be the case that by training more, we get a worse result.

So far we have experimented the behavior of Auto-Scaler under different configurations. However, target latency has been set to 30 seconds in all experiments. In this experiment, we will set other target values to evaluate whether Q-Learning approach is able to hit multiple targets. Table 5.9 describes the configuration parameters of this experiment. Note that different sliding windows are used for Workload 1 and Workload 2. Three minute history window performs better for Workload 1, so this option is adopted. Additionally, based on previous experiments, setting target value to more than 30 seconds makes it too easy for Auto-Scaler. Thus, those cases are not tested.

#	Experiment	Configuration
1	Target Latency: 10 Seconds	Executor Granularity: 1 Executor Strategy: Queue Aware Table Initializer: Optimized Decision Interval: 1 Minute History Window: 3 Minutes for Workload 1 History Window: 2 Minutes for Workload 2 Learning Factor = 0.7 Discount Factor: 0.9 Policy: Greedy Reward: Prefer Scale-In When Load Is Decreasing
2	Target Latency: 20 Seconds	
3	Target Latency: 30 Seconds	

Table 5.9: Approaching Target Configuration Parameters

Figures 5.58, 5.59, 5.60 and 5.61 illustrates latency and executor charts for Workload 1. Figures 5.62, 5.63, 5.64 and 5.65 illustrates latency and executor charts for Workload 2.

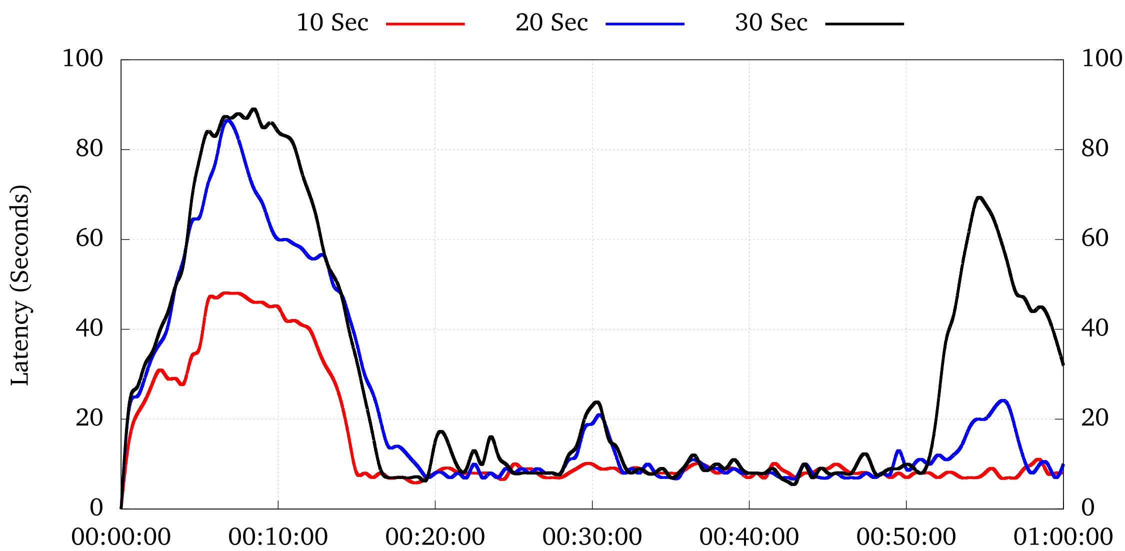


Figure 5.58: Approaching Target – Workload 1 – Latency

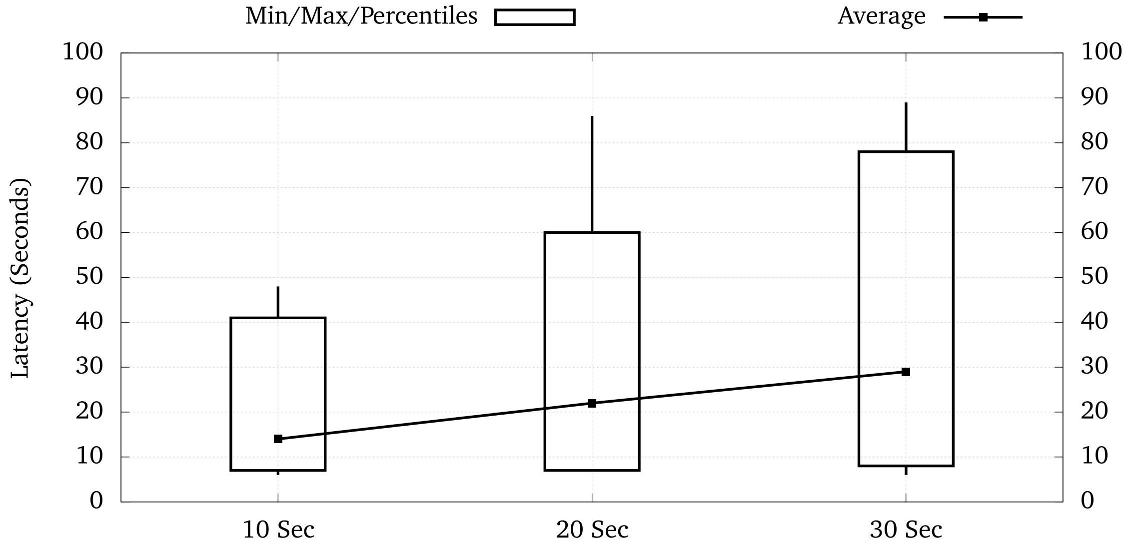


Figure 5.59: Approaching Target – Workload 1 – Latency

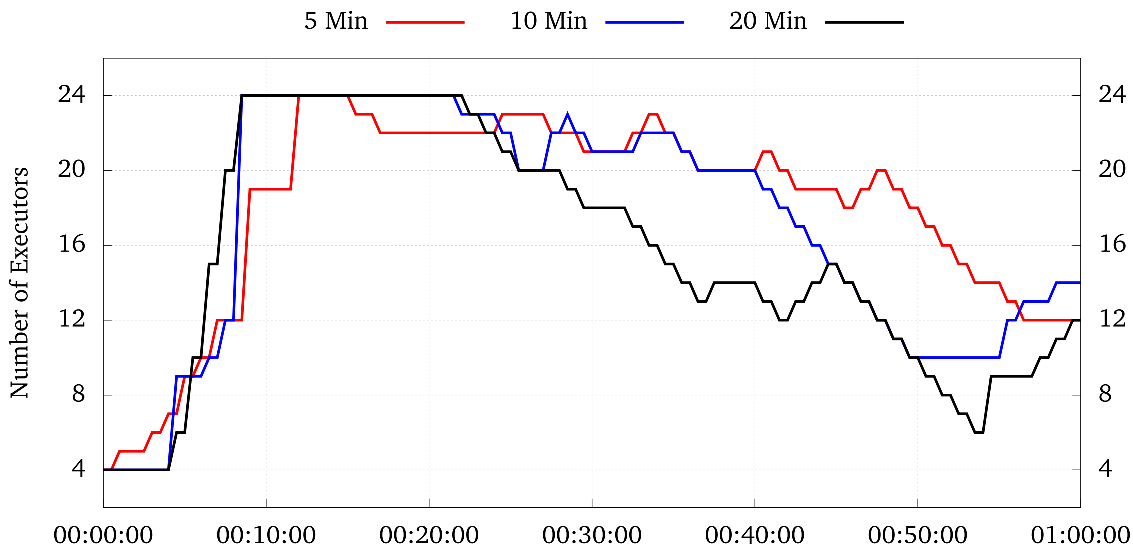


Figure 5.60: Approaching Target – Workload 1 – Number of Executors

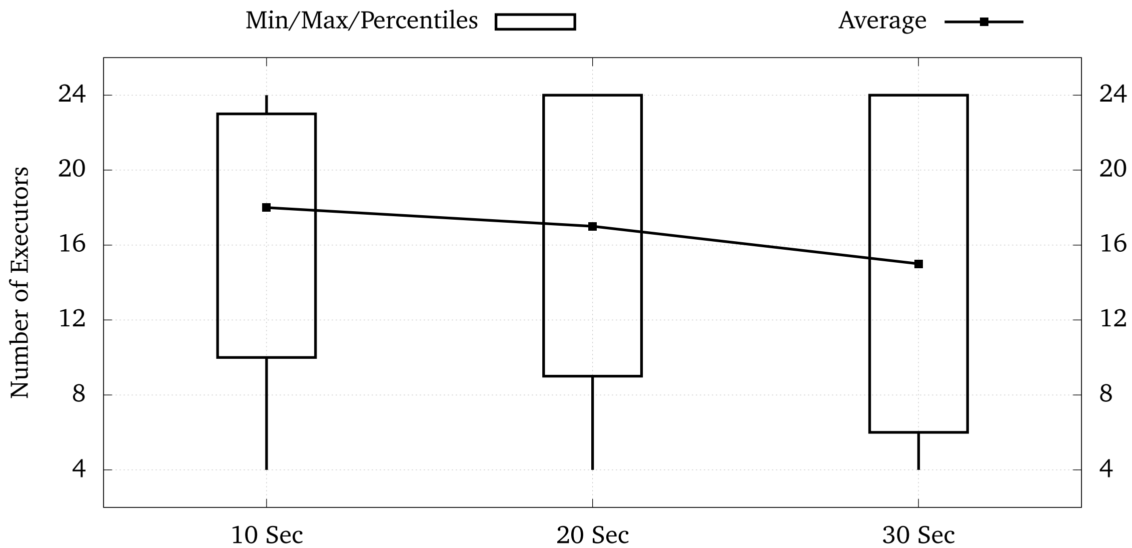


Figure 5.61: Approaching Target – Workload 1 – Number of Executors

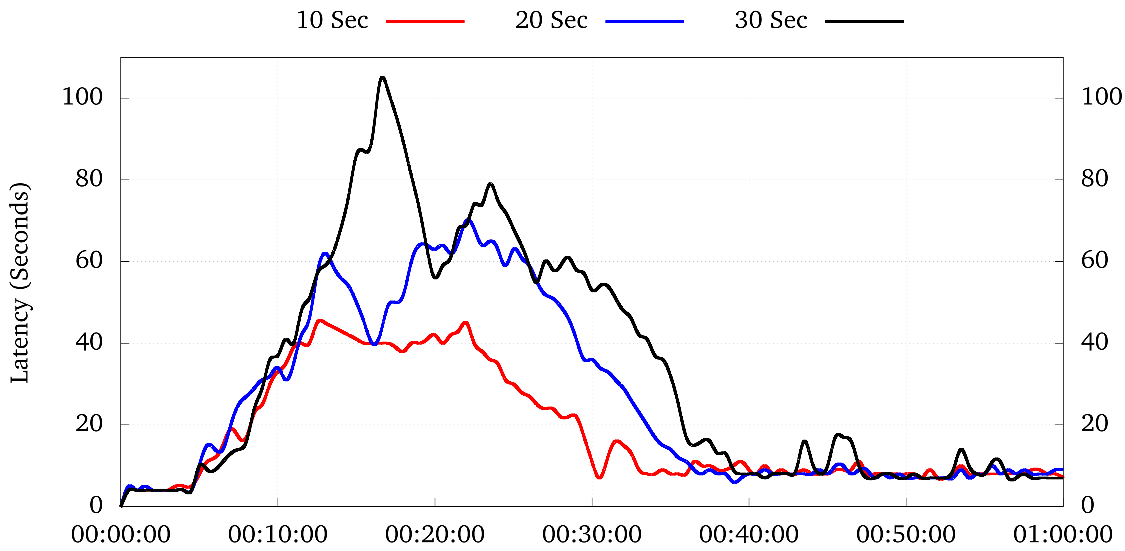


Figure 5.62: Approaching Target – Workload 2 – Latency

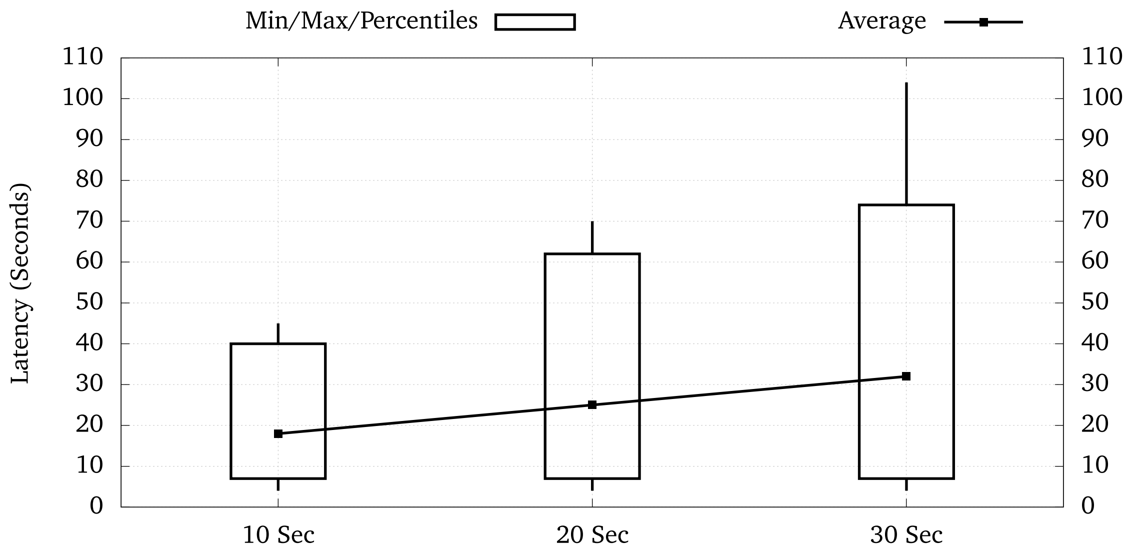


Figure 5.63: Approaching Target – Workload 2 – Latency

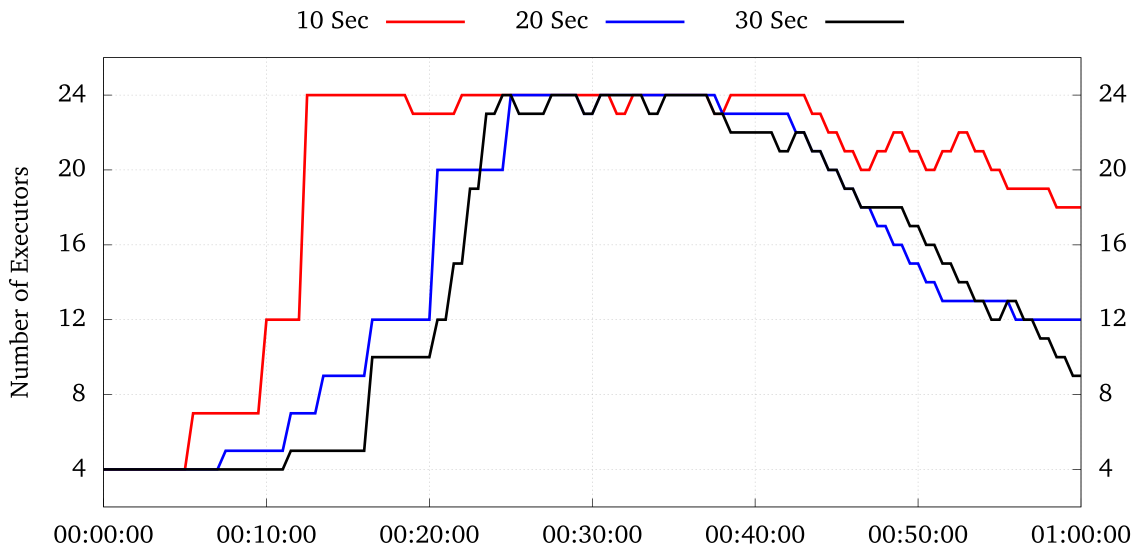


Figure 5.64: Approaching Target – Workload 2 – Number of Executors

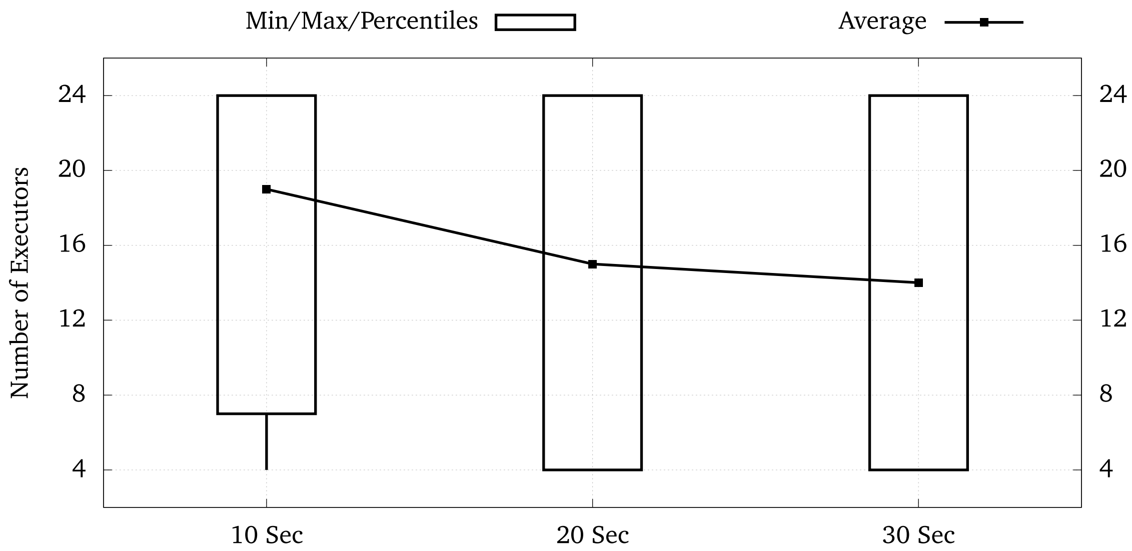


Figure 5.65: Approaching Target – Workload 2 – Number of Executors

5.9.1 Conclusion

Looking at results, the more we get close to 10-second target latency, the more it gets challenging for Auto-Scaler. However, there is one common behavior in both workloads. The key factor to reduce latency is to apply *early* Scale-Out. This can be confirmed by Figure 5.60 and Figure 5.64. However, as explained in Section 2, Reinforcement Learning agent has to visit a state to take an appropriate action which may be too late when the peak workload is observed. This is one of the cases where proactive – predictive – approaches are more useful.

5.10 Experiment 9: Overall Comparison

In this experiment implemented Q-Learning technique is compared to bare bone Spark with fixed number of executors and Spark Streaming's default dynamic resource manager. Furthermore, Online Parameter Optimization [30] which is implemented for Spark Streaming by Kielbowicz [42] is also included. Table 5.10 describes the configuration parameters of this experiment.

<table>
<tr><th>#</th><th>Experiment</th><th>Configuration</th></tr>
<tr><td>1</td><td>Static 4 Executors</td><td rowspan="4">–</td></tr>
<tr><td>2</td><td>Static 8 Executors</td></tr>
<tr><td>3</td><td>Static 12 Executors</td></tr>
<tr><td>4</td><td>Static 24 Executors</td></tr>
<tr><td rowspan="2">5</td><td rowspan="2">Q-Learning</td><td>Workload 1: Same as Experiment 3
Decision Interval: 2 Minutes</td></tr>
<tr><td>Workload 2: Same as Experiment 3
Decision Interval: 1 Minute</td></tr>
<tr><td>6</td><td>Spark Dynamic Resource Manager</td><td>Scaling Interval: 60 Seconds
Scale Down Ratio: 0.3
Scale Up Ratio: 0.6</td></tr>
<tr><td>7</td><td>Online Parameter Optimization</td><td>Threshold Breaches Range: 2 to 128
Threshold Breaches Step: 4</td></tr>
</table>

Table 5.10: Overall Comparison Configuration Parameters

Note that, Spark Dynamic Resource Manager uses Equation 5.2 to calculate a *Ratio* value and then compares it to Scale-Up and Scale-Down ratio values obtained from configuration. In order to influence the behavior of this resource manager, batch duration should be changed. To make it fair, in all experiments the batch size is set to 10 seconds.

$$\text{Ratio} = \frac{\text{Average Latency of History Window}}{\text{Batch Size}} \tag{5.2}$$

For Q-Learning approach, two of the best cases from previous experiments were obtained. Also note that seven cases are evaluated in this experiment which makes it difficult to comprehend the detailed latency charts. Thus, they are removed. Figures 5.66 and 5.67 illustrate latency charts for both workloads. Figures 5.68 and 5.69 illustrates executor charts for both workloads.

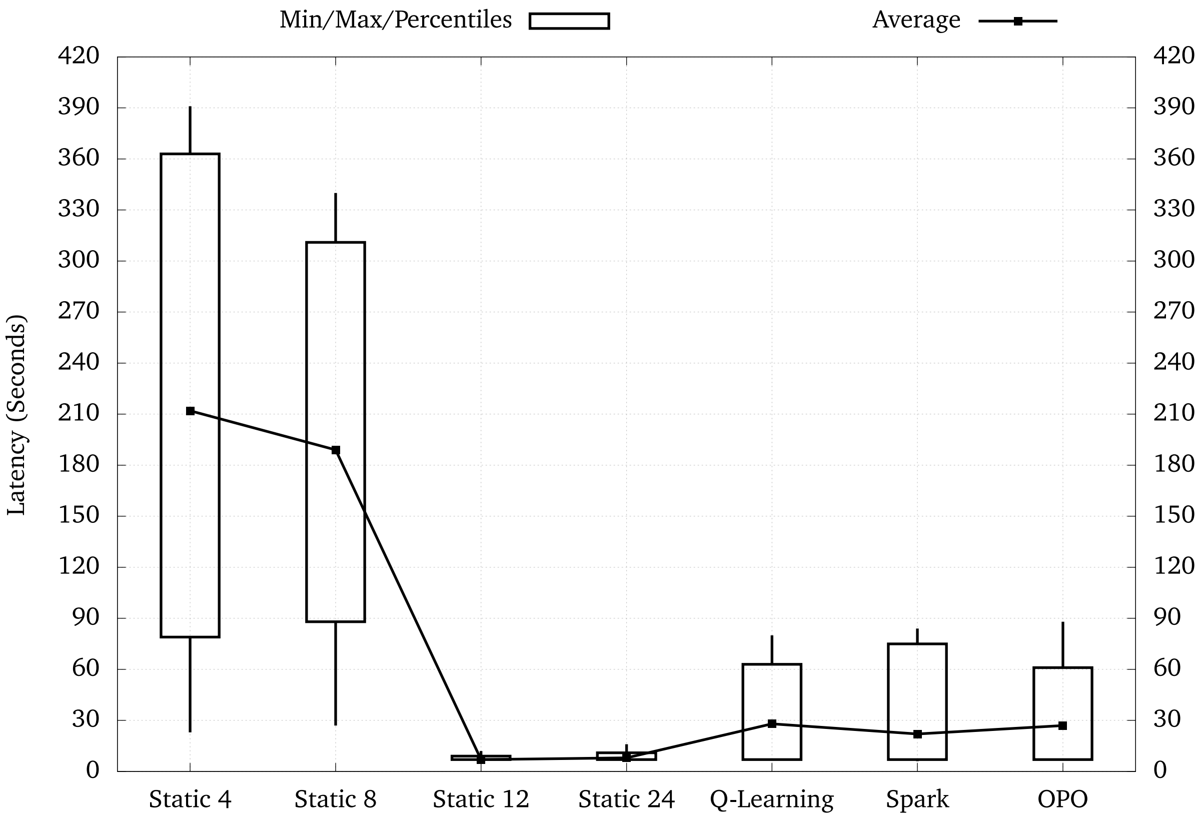


Figure 5.66: Overall Comparison – Workload 1 – Latency

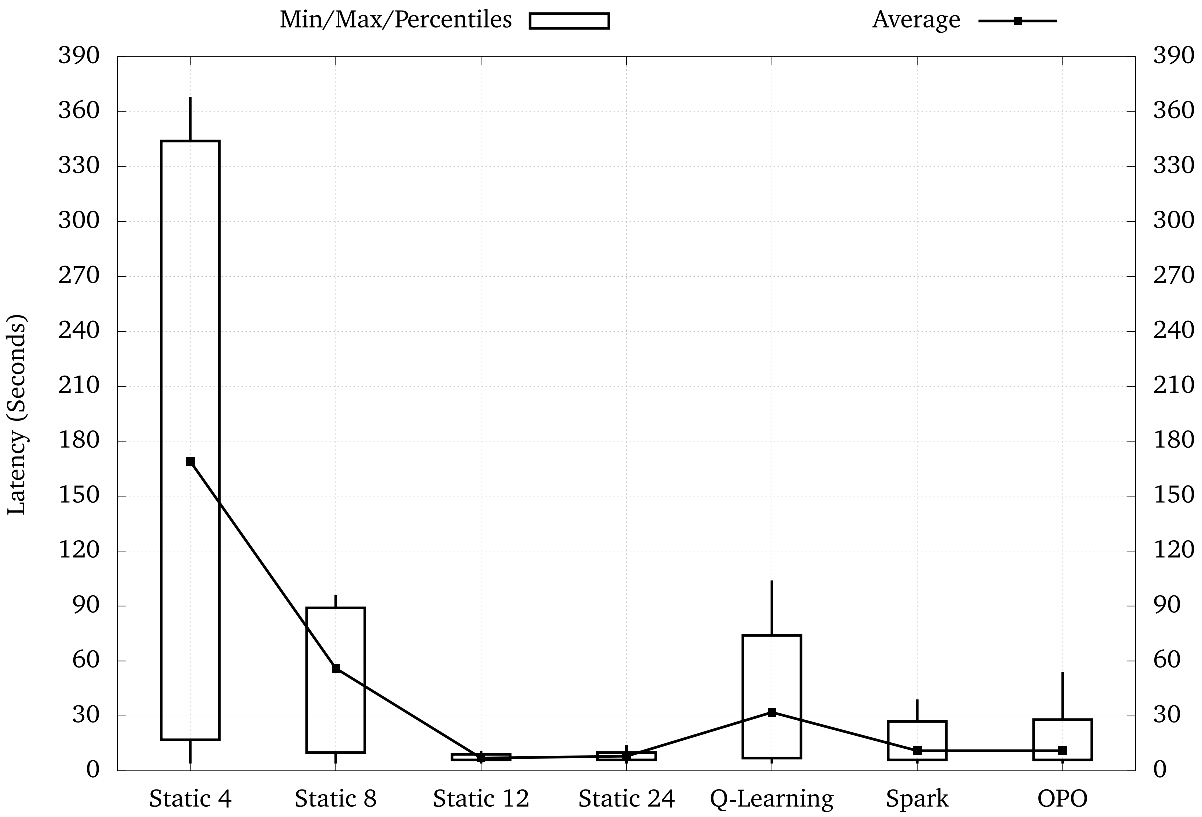


Figure 5.67: Overall Comparison – Workload 2 – Latency

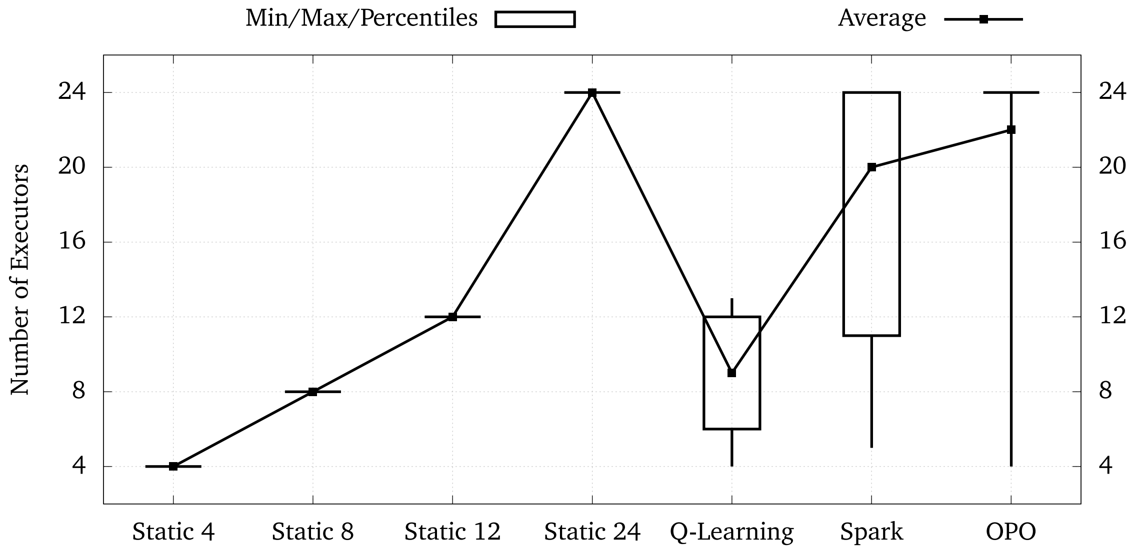


Figure 5.68: Overall Comparison – Workload 1 – Number of Executors

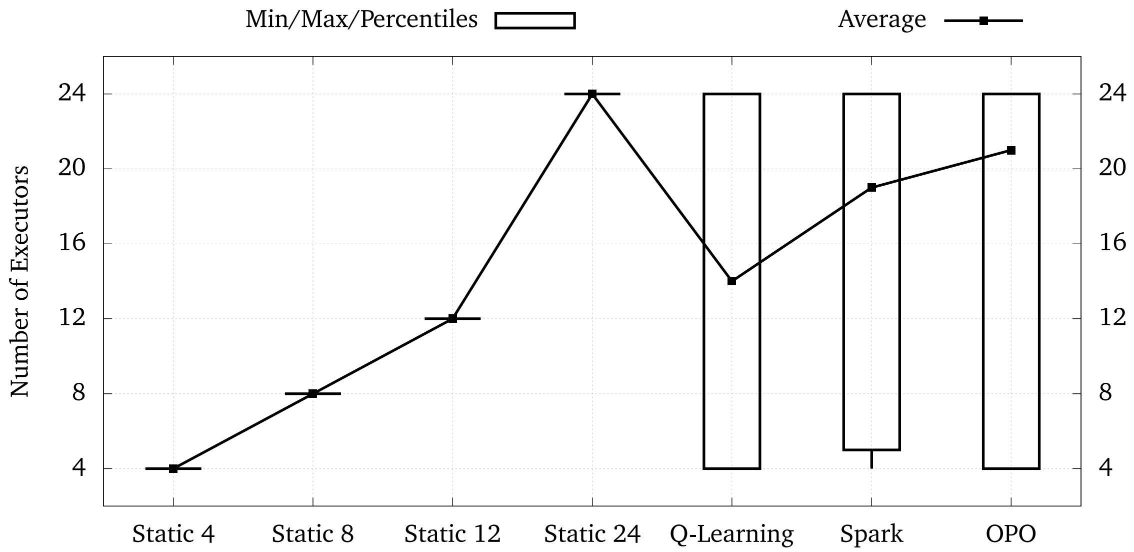


Figure 5.69: Overall Comparison – Workload 2 – Number of Executors

5.10.1 Conclusion

As can be seen, from 8 to 12 executors the latency drops significantly. It can be concluded that to hit target latency – which is 30 seconds in this case – optimal number of executors is between 8 to 12. Note that, 12 executors is slightly better than 24 executors. This could be the result of extra network communication involved in case of 24 executors. Spark and OPO achieve better latency than Q-Learning approach but with much more executors.

6 Related Work

Dynamic resource allocation in cloud environments has been studied extensively in literature. In this chapter prior work will be discussed and explored. It is organized as follows. Section 6.1 delves into threshold-based techniques. Section 6.2 investigates techniques based on time-series analysis. Section 6.3 analyses techniques based on queuing theory. Section 6.4 explores Reinforcement Learning techniques comprehensively. Finally, Section 6.5 concludes.

6.1 Threshold-Based Techniques

Hasan et al. [28] proposed four thresholds and two time periods. `ThrUpper` defines upper bound. `ThrBelowUpper` is slightly below `ThrUpper`. Similarly, `ThrLower` defines lower bound and `ThrAboveLower` is slightly above the lower bound. In case, system utilization stays between `ThrUpper` and `ThrBelowUpper` for a specific duration, then cluster controller decides to take a scale-out action, by adding resources. On the other hand, if system utilization stays between `ThrLower` and `ThrAboveLower` for a specified duration, then the central controller decides to take scale-in action. Furthermore, in order to prevent making *oscillating* decisions, *grace period* is enforced. During this period, no scaling decision is made. Defining two levels of thresholds helps to detect workload *persistency* and avoids making immature scaling decision. However, defining thresholds is a tricky and manual process, and needs to be carefully done [21]. It shall be noted that, computation overhead of this approach is very low.

RightScale [55] applies voting algorithm among nodes to make scaling decisions. In order for a specific action to be decided, majority of nodes should vote in favor of that specific action. Otherwise, no-action is elected as a default action. Afterwards, nodes apply grace period to stabilize the cluster. The complexity of the voting process in trusted environments is in the order of $O(n^2)$, which leads to heavy network traffic among participants when cluster size grows. This approach also suffers from the same issue – accurately adjusting threshold values – as other threshold-based approaches.

Heinze et al. [30] proposed a novel threshold-based solution in the context of FUGU [26] – a data stream processing framework. This techniques uses an adaptive window [9] to monitor the recent changes in workload pattern. In case a change in workload is detected, optimization component is activated and fed with recent short-term utilization history. Thereafter, the optimization component determines monetary cost of current system configuration and then simulates the cost of different scaling decisions. The *latency-aware* cost function has the responsibility to calculate monetary cost of system configuration. The search function is an implementation of *Recursive Random Search* [63] algorithm which consists of two phases. First, in *exploration* phase, the complete parameter space is explored to find a solution with minimum cost. In second phase – *exploitation phase* – only specific parts of the parameter space which has been discovered in first phase, will be investigated. Kielbowicz [42] has implemented this techniques in the context of Spark Streaming. Thus, It is considered in evaluation scenarios.

6.2 Time-Series Analysis Techniques

Herbst et al. [33] surveys different auto-scaling techniques based on time-series analysis in order to forecast *trends* and *seasons*. *Moving Average Method* takes the average over a sliding window and smooths out minor noise level. Its computational overhead is proportional to size of the window. *Simple Exponential Smoothing*(SES) goes further than just taking average. It gives more weight to more recent values in sliding window by an exponential factor. Although it is more computationally intensive compared to moving average, it is still negligible. SES is capable of detecting short-term trends but fails at predicting seasons. These approaches are more specific instances of *ARIMA* (Auto-Regressive Integrated Moving Average) which is a general purpose framework to calculate moving averages. However, time-series analysis is only suitable for stationary problems consist of recurring workload patterns such as web applications. Additionally, more advanced forms of time-series analysis which are capable of forecasting seasons (such as *tBATS Innovation State*

Space Modeling Framework [44], *ARIMA Stochastic Process Modeling Framework* [37]) are computationally infeasible for streaming workloads.

Taft et al. [58] applied time-series analysis in the context of OLTP databases. The authors argue that reactive approaches don't fit to database world. By the time, auto-scaler component decides to scale-out, it is already too late for a database system. This premise comes from the fact that taking scaling actions in a database doesn't take place in timely manner. The database system has to replicate some of the records which is an additional burden on a heavily loaded system. Thus, database system must take proactive approach and take scaling decisions ahead of time. While this is convincing argument, the auto-scaler module depends on a couple of parameters that are hard to calculate in heterogeneous public cloud environments. First, target throughput of a single server. Second, shortest time to move all database records with single sender-receiver thread. While this might be feasible in some scenarios, on today's cloud environments with virtual machines hosted on heterogeneous physical nodes, getting a near-precise number is unconvincing. It worth noting that author assumed an approximately uniform workload distribution for all database nodes – each database shard serves a fairly equal portion of total workload which is a questionable assumption.

6.3 Queuing Theory Techniques

Lohrmann, Janacik, and Kao [45] proposed a solution based on queuing theory. The solution is designed for *Nephele* [46] streaming engine which has a master-worker style architecture. Similar to Spark Streaming, a job is modeled as a DAG. It utilizes *adaptive output batching* [62] – which is essentially a buffer with variable size – to buffer outgoing messages emitted from one stage to the other. Each task – an executer that runs user defined function (UDF) – is modeled as a G/G/1 queue. That is, the probability distributions of message inter-arrival and service time are unknown. In order to approximate these distributions, a formula proposed by Kingman [43] is used. From a bird's eye view, this solution seems promising. However, authors made two unconvincing assumptions that led us to abandon the proposal. First, worker nodes shall be homogeneous in terms of processing power and network bandwidth. Second, there should be an effective partitioning strategy in place in order to load balance outgoing messages between stages. In reality both assumptions rarely occur. Large scale stream processing clusters are built incrementally. Depending on workload, data skew does exist and imperfect hash functions are widely used by software developers.

Zhang, Cherkasova, and Smirni [67] proposed a solution for multi-tiered enterprise applications based on regression techniques. Regression based models can absorb some level of uncertainty and noise by compacting samples. Each tier is modeled as G/G/1 queue and scaled differently compared to other tiers. The system has fixed number of users – a principle known as *closed-loop queuing network*. In order to calculate system workload – incoming message rate – and service time which is required by queuing models, the authors proposed to use Mean Value Analysis [49]. In order to simplify the queuing network, the system is modeled as a *transaction-based* system with independent requests coming from clients. However, It is widely believed that multi-tiered enterprise applications are *session-based* systems [16]. Each request from the same client depends on her previous request during a specific session.

6.4 Reinforcement Learning Techniques

Herbst et al. [31] surveys on state of the art techniques to predict future workload. It includes workload forecasting based on *Bayesian Networks* (BN) and *Neural Networks*. There are several issues with each of them that makes them unsuitable for streaming workloads. As an example, there is no universally applicable method to construct a BN. Furthermore, it requires collecting data and training the model offline. Neural networks suffer from the same issues. That is, it requires collecting samples and periodically training the model. For complex models, training phase is typically computationally infeasible which is conflicting with requirements of thesis.

Tesauro et al. [59] proposes a hybrid approach to overcome poor performance of online training. The system consists of two components: an online component based on queuing system combined with Reinforcement Learning component that is trained offline. The offline component is based on *neural networks*. The authors model the data center as multiple applications managed under a single resource manager. Modeling streaming workloads as a queuing system has two problems. First, modeling is a complicated process and determining probability distributions requires domain knowledge. Second, it requires access to each node (so it can be modeled as a queue) which is currently not possible without modifying `spark-core` package. Since, it was one the requirements to provide a solution without making any modification to `spark-core`, this work has been abandoned.

Rao et al. [53] proposed to use Reinforcement Learning to manage resources consumed by virtual machines. It employs standard model-free learning, which is known as *Temporal Difference* [57] or *Sarsa* algorithm. The state space consists of metrics collected from virtual machines (CPU, RAM, Network IO, . . .). There is no global controller and each node decides based on its own Q-Table. As mentioned in literature, standard temporal difference has a slow convergence speed. In order to speedup bootstrap phase, Q-Table is initialized by values that were obtained during separate supervised training. Since this approach also relies on offline training, it wasn't adopted by this thesis.

Enda, Enda, and Jim [23] proposed a parallel architecture to Reinforcement Learning. Standard model-free learning (Temporal Difference) is used. No global controller is involved and each node decides locally. In order to speed up learning, all nodes maintain two Q-Tables (local and global tables). Local table is learned and updated by each node. Whenever, an agent learns a new value for a specific state, it broadcasts it to other agents. The global table contains values received from other agents. Additionally, agent prioritize local and global tables by assigning weights to each table. Weights are factors that are defined by application developers. The final decision is the outcome of combining local and global tables. Although each node learns some part of the state space (which may overlap with other nodes), it is not applicable in the context of Spark Streaming. The assumption in this architecture is that, each node is operating autonomously without intervention from other nodes (such as web servers). In contrast, Spark is a centrally managed system. That is, all nodes running Spark jobs are supervised by a single master node (probably with couple of backup masters).

Heinze et al. [29] implemented Reinforcement Learning in the context of FUGU [26] and compared it to threshold-based approaches. Each node, maintains its own Q-Table and imposes local policy without coordinating other nodes. This architecture can not be applied in the context of spark streaming, since Spark abstracts away individual nodes from the perspective of application developer. In order to decrease state space, the author applied two techniques. First, only system utilization is considered. Second, system utilization is discretized using coarse grained steps. To remedy slow convergence, the controller enforces a *monotonicity constraint* [34]. That is, if the controller decides to take scale-out action for a specific utilization, it may not decide scale-in for even worse system utilization. This feature has been adopted by this thesis.

Cardellini et al. [12] proposed a two level hierarchical architecture for resource management in Apache Storm [56]. There is a local controller on each node which is cooperating with the global controller. The local controller monitors each operator using different policies (threshold-based or Reinforcement Learning using temporal difference). In case, local controller decides to scale in or out an specific operator, it contacts the global controller and informs it about its decision. Then it waits to receive confirmation from the global controller. The global controller operates using a token-bucket-based policy [13] and has global view of cluster. It ranks requests coming from local controllers and either confirms or rejects their decisions. Although, this architecture seems to be a promising approach, however it has been implemented by modifying Storm's internal components. As mentioned above, this is in conflict with thesis's requirements.

In order to mitigate the problem of large state space in Reinforcement Learning, Lolos et al. [47] proposed to start the agent from small number of coarse grained states. As more metrics are collected (and stored as historical records), agent will discover *outlier* parameters (those parameters that are affecting agent more, CPU rather than IO as an example). Then, it partitions the affected state into two states and *re-trains* newly added states using historical records. Both Temporal Difference and Value Iteration methods can be used as learning algorithm. Gradually, agent only focuses on some specific parts of the state space, since all parameters are not equally important. This approach, effectively reduces the size of state space. However, the trade-off is the storage cost in which historical metrics need to be stored. It worth noting that from the context of paper, storage cost (whether it is in-memory or on-disk and the duration of storing historical metrics) is unclear. Thus, this approach has been abandoned due to uncertainty.

Dutreilh et al. [22] proposed a model-based Reinforcement Learning approach for resource management of cloud applications. All virtual machines are supervised by a single global controller. Slow convergence is the bottleneck of model-free learning, in contrast to model-based learning. However, environment dynamics are not available at the time of modeling. Authors proposed to estimate these parameters as more metrics are collected and then switch to *Value Iteration* [57] algorithm instead of *Temporal Difference*. In short, statistical metrics are stored and updated for each visit of (old state, action, reward, new state) quadruple. As more samples are collected, statistical metrics become more accurate and can be directly used in *Bellman* equation. Until enough measurements get collected, a separate initial reward function is used which is essentially the original reward function but with penalty costs removed. Furthermore, In order to reduce the state space – tuple of [request/sec, number of VMs, average response time] – there exists a predefined upper and lower bound for state variables and average response time is measured at granularity of seconds. This approach has been partially adopted by this thesis.

Dutreilh et al. [21] proposed a model-free Reinforcement Learning approach (*Temporal difference* algorithm) with modified *exploration* policy. The standard exploration policy for Q-Learning is $1-\epsilon$. Under this policy, the agent performs a random action with probably of ϵ and with probability of $1-\epsilon$, it adheres to an action proposed by optimal policy. Although the random action is necessary to explore unknown states, but it has sever consequences under streaming workloads. In some cases, it leads to unsafe states where SLOs are severely violated. Since streaming is heavily latency sensitive, this property is undesirable. Thus, author sought toward a heuristic-based policy proposed by Bodik et al. [10]. This policy is based on couple of key observations which has been adopted by this thesis:

- It must quickly explore different states.
- It should collect accurate data as fast as possible, to speedup training.
- During exploration phase, the policy should be careful not to violate SLOs.

6.5 Summary

In this chapter prior work on auto-scaling scaling has been discussed and evaluated. First, threshold-based approaches are investigated. Simple threshold-based approaches are intuitive and simple to understand by application developers and are widely supported by cloud providers. However, adjusting thresholds is a tricky and error-prone process. Then, time-series analysis techniques are explored. As confirmed by other authors, advanced seasonal forecasting is a computationally intensive process, which makes it less suitable for streaming workloads. Queuing theory approaches are suitable for stationary networks with a known probability distribution for workload and service time. Reinforcement Learning techniques has the benefit that it requires zero knowledge about the environment which helps to gradually adapt to changes in environment.

7 Conclusion

Over-provisioning of resources is a common issue among modern cloud-based applications. The issue has been studied comprehensively in literature. It is specifically interesting in the context of streaming workloads with unpredictable workload patterns. Unpredictable workloads are getting common by increasing adoption of IOT style workloads. For such workloads, dynamic resource management is becoming a necessary component rather than a nice feature sitting around the corner. Previous research has led to a wide spectrum of dynamic resource managers with extensive list of features. Proposals vary from architectural to algorithmic aspects. In this thesis a dynamic resource manager has been implemented for Spark Streaming.

Considering thesis goals has led to the solution proposed in this thesis. Since Spark has a Master-Slave architecture, this architecture has also been adopted by this thesis. The resource manager is implemented as a library which can be included by the Spark job. It subscribes to Spark's internal events and takes actions periodically. Since modifying Spark source code was prohibited, other architectures – that require local agents on each node – have been abandoned. From algorithmic point of view, Reinforcement Learning approach has been elected. Time-series analysis approaches are suitable for repetitive workloads. Queuing approaches are suitable for stationary systems. Threshold based approaches require manually specifying threshold values. Considering previous research implies that Reinforcement Learning approach is reasonably right choice that is capable of adopting dynamic workloads.

Reinforcement Learning consists of wide range of algorithms. As mentioned the focus of thesis was on streaming workloads which implies that decision making has to be done efficiently and in timely manner. As a consequence, those algorithms that require lengthy and intensive offline training were not adopted. This led us to two algorithms. Namely Temporal Difference – Q-Learning – and Value Iteration. A great aspect of Q-Learning is that it doesn't require any training at all which makes it perfectly suitable for streaming workloads. On the other hand, as confirmed by literature one of the negative aspects of Q-Learning is its slow learning process. This led us to two solutions. First, initializing the state space by sensible values which is independent of workload. Second, Value Iteration algorithm which requires not-so-intensive period of training phase.

Alongside the proposed state space initializer, a couple of optimizations have been implemented as well. First, a queue aware executor strategy which is derived from TCP Slow-Start algorithm has been implemented. It makes Scale-Out actions in large steps and Scale-In actions in small steps. Second, a monotonicity constraint has been adopted from previous research which to some degree prevents the Auto-Scaler from making undesirable decisions. Third, state space is discretized both in direction of latency and incoming messages. The implementation is extensible to a large degree. It is fairly easily to provide different implementation of state space initializer, policy function, reward function and executor strategy. Note that these components are not system wide parameters. Each job can easily provide its own implementation of these components at submission time.

Two workloads from DEBS 2014 that have high degree of unpredictability have been chosen to verify the implementation and evaluate it by changing configuration parameters. As can be seen, the implementation is able to achieve its goals. That is, it is able to optimize in two directions. First, optimizing for latency. Second, optimizing with respect to number of executors. At least based on aforementioned workloads it achieved better results than Spark's default dynamic resource manager.

In order to make the implementation suitable for vast number of workloads, many configuration parameters have been introduced. Evaluations demonstrate how sensitive Q-Learning approach is to configuration changes. However, on the negative side it makes it pretty cumbersome to find an optimal configuration. Note that, even by running many experiments in Evaluation section, it is still not possible to define optimal configuration. Although some empirical hints can be provided but it is just speculation – not formally provable – and to a large degree workload dependent. In other words, the configuration space is so large that it requires too many manual experiments to find out a fairly desirable configuration.

The issue of large configuration space remains mostly intact regardless of the underlying algorithm. Similar to sophisticated software projects, it will most probably suffer from having too many configuration parameters. As a consequence, the problem of dynamic resource management has been shifted from design space to configuration space. The author of this thesis, believes this is the right direction for future research. That is, an approach that finds the optimal configuration from a huge multi dimensional configuration space which in turn leads to optimal runtime.

Bibliography

[1] Amazon. *Amazon AWS Cloud*. Accessed July 16, 2018. 2018. URL: https://aws.amazon.com.

[2] Apache. *Apache Flink*. Accessed July 17s, 2018. 2018. URL: https://flink.apache.com.

[3] Apache. *Apache Hadoop*. Accessed July 23, 2018. 2018. URL: https://hadoop.apache.org.

[4] Apache. *Apache Hive*. Accessed July 23, 2018. 2018. URL: https://hive.apache.org.

[5] Apache. *Apache Kafka*. Accessed July 27, 2018. 2018. URL: https://kafka.apache.org.

[6] Apache. *Apache Pig*. Accessed July 23, 2018. 2018. URL: https://pig.apache.org.

[7] Apache. *Apache Spark*. Accessed July 17, 2018. 2018. URL: https://spark.apache.com.

[8] Apache. *Apache Zookeeper*. Accessed July 19, 2018. 2018. URL: https://zookeeper.apache.org.

[9] A. Bifet and R. Gavaldà. "Learning from Time-Changing Data with Adaptive Windowing". In: *Proceedings of the 7th SIAM International Conference on Data Mining*. Vol. 7. Apr. 2007.

[10] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. "Automatic Exploration of Datacenter Performance Regimes". In: *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*. ACDC '09. Barcelona, Spain: ACM, 2009, pp. 1–6.

[11] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time Series Analysis: Forecasting and Control*. 5th ed. The MIT Press, 2015.

[12] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo. "Decentralized self-adaptation for elastic Data Stream Processing". In: *Future Generation Computer Systems* 87 (2018), pp. 171 –185.

[13] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. "Towards Hierarchical Autonomous Control for Elastic Data Stream Processing in the Fog". In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer. Cham: Springer International Publishing, 2018, pp. 106–117.

[14] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 725–736.

[15] B. Chambers and M. Zaharia. *Spark: The Definitive Guide*. 1st ed. O'Reilly Media, 2018. ISBN: 1491912219.

[16] L. Cherkasova and P. Phaal. "Session-based admission control: a mechanism for peak load management of commercial Web sites". In: *IEEE Transactions on Computers* 51.6 (2002), pp. 669–685.

[17] CoreOS. *CoreOS Etcd*. Accessed July 19, 2018. 2018. URL: https://coreos.com/etcd/.

[18] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[19] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144.

[20] Docker. *Docker*. Accessed July 17, 2018. 2018. URL: https://www.docker.com/.

[21] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. "From Data Center Resource Allocation to Control Theory and Back". In: *2010 IEEE 3rd International Conference on Cloud Computing*. 2010, pp. 410–417.

[22] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. "Using Reinforcement Learning for Autonomic Resource Allocation in Clouds: towards a fully automated workflow". In: *7th International Conference on Autonomic and Autonomous Systems (ICAS'2011)*. Venice, Italy, May 2011, pp. 67–74. URL: https://hal-univ-paris8.archives-ouvertes.fr/hal-01122123.

[23] B. Enda, H. Enda, and D. Jim. "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud". In: *Concurrency and Computation: Practice and Experience* 25.12 (2012), pp. 1656–1674.

[24] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. "Exploring Alternative Approaches to Implement an Elasticity Policy". In: *2011 IEEE 4th International Conference on Cloud Computing*. 2011, pp. 716–723.

[25] Google. *Google Cloud*. Accessed July 16, 2018. 2018. URL: https://cloud.google.com.

[26] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (2014), pp. 455–466.

[27] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow. "The DEBS 2016 Grand Challenge". In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 289–292. ISBN: 978-1-4503-4021-2.

[28] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. "Integrated and autonomic cloud resource scaling". In: *2012 IEEE Network Operations and Management Symposium* (2012), pp. 1327–1334.

[29] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. "Auto-scaling techniques for elastic data stream processing". In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. 2014, pp. 296–302.

[30] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. "Online Parameter Optimization for Elastic Data Stream Processing". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 276–287.

[31] N. Herbst, A. Amin, A. Andrzejak, L. Grunske, S. Kounev, O. J. Mengshoel, and P. Sundararajan. "Online Workload Forecasting". In: *Self-Aware Computing Systems*. Ed. by S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. Cham: Springer International Publishing, 2017, pp. 529–553.

[32] N. R. Herbst, S. Kounev, and R. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not". In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pp. 23–27.

[33] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. "Self-adaptive Workload Classification and Forecasting for Proactive Resource Provisioning". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ICPE '13. Prague, Czech Republic: ACM, 2013, pp. 187–198.

[34] H. Herodotou and S. Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *Proceedings of the VLDB Endowment*. Vol. 4. Jan. 2011, pp. 1111–1122.

[35] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308.

[36] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. "High-Availability Algorithms for Distributed Stream Processing". In: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 779–790.

[37] R. Hyndman and Y. Khandakar. "Automatic Time Series Forecasting: The forecast Package for R". In: *Journal of Statistical Software, Articles* 27.3 (2008), pp. 1–22.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. “Dryad: Distributed Data-parallel Programs from Sequential Building Blocks”. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys ’07. Lisbon, Portugal: ACM, 2007, pp. 59–72.

[39] Z. Jerzak and H. Ziekow. “The DEBS 2014 Grand Challenge”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS ’14. Mumbai, India: ACM, 2014, pp. 266–269. ISBN: 978-1-4503-2737-4.

[40] Z. Jerzak and H. Ziekow. “The DEBS 2015 Grand Challenge”. In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS ’15. Oslo, Norway: ACM, 2015, pp. 266–268. ISBN: 978-1-4503-3286-6.

[41] D. G. Kendall. “Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain”. In: *Ann. Math. Statist.* 24.3 (Sept. 1953), pp. 338–354.

[42] M. Kielbowicz. “Online parameter optimization for Spark Streaming”. SAP, 2017.

[43] J. F. C. Kingman. “The Single Server Queue in Heavy Traffic”. In: *Proceedings of the Cambridge Philosophical Society* 57 (1961), p. 902.

[44] A. M. D. Livera, R. J. Hyndman, and R. D. Snyder. “Forecasting Time Series With Complex Seasonal Patterns Using Exponential Smoothing”. In: *Journal of the American Statistical Association* 106.496 (2011), pp. 1513–1527.

[45] B. Lohrmann, P. Janacik, and O. Kao. “Elastic Stream Processing with Latency Guarantees”. In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. 2015, pp. 399–410.

[46] B. Lohrmann, D. Warneke, and O. Kao. “Nephele streaming: stream processing under QoS constraints at scale”. In: *Cluster Computing* 17.1 (2014), pp. 61–78.

[47] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. “Elastic Resource Management with Adaptive State Space Partitioning of Markov Decision Processes”. In: (2017).

[48] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments”. In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592.

[49] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[50] Microsoft. *Microsoft Azure Cloud*. Accessed July 16, 2018. 2018. URL: `https://azure.microsoft.com`.

[51] T. Patikirikorala and A. Colman. “Feedback controllers in the cloud”. In: (July 2018).

[52] D. Rajarshi, G. Tesauro, and W. E. Walsh. *Model-Based and Model-Free Approaches to Autonomic Resource Allocation*. IBM Research Division, 2005.

[53] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. “VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration”. In: *Proceedings of the 6th International Conference on Autonomic Computing*. ICAC ’09. Barcelona, Spain: ACM, 2009, pp. 137–146.

[54] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. “Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. New York, NY, USA: ACM, 2012, 7:1–7:13.

[55] RightScale. *Set up Autoscaling using Voting Tags*. Accessed June 20, 2018. 2018. URL: `http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags/`.

[56] A. Storm. *Apache Storm*. Accessed June 21, 2018. 2018. URL: `http://storm.apache.org/`.

[57] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. The MIT Press, 1998. ISBN: 0262193981.

[58] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. “P-Store: An Elastic Database System with Predictive Provisioning”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: ACM, 2018, pp. 205–219. ISBN: 978-1-4503-4703-7.

[59] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. “A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation”. In: *2006 IEEE International Conference on Autonomic Computing*. 2006, pp. 65–73.

[60] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16.

[61] J. Walter, D. Okanović, and S. Kounev. “Mapping of Service Level Objectives to Performance Queries”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE ’17 Companion. L’Aquila, Italy: ACM, 2017, pp. 197–202. ISBN: 978-1-4503-4899-7.

[62] D. Warneke and O. Kao. “Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 985–997.

[63] T. Ye and S. Kalyanaraman. “A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration”. In: *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’03. San Diego, CA, USA: ACM, 2003, pp. 196–205.

[64] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. “Discretized Streams: Fault-tolerant Streaming Computation at Scale”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8.

[66] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’10. Boston, MA: USENIX Association, 2010, pp. 10–10.

[67] Q. Zhang, L. Cherkasova, and E. Smirni. “A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications”. In: *Fourth International Conference on Autonomic Computing (ICAC’07)*. 2007, pp. 27–27.