



Department of Informatics

Technical University of Munich

Bachelor's Thesis in
Informatics: Games Engineering

Autoscaling Rules Evaluation with the Autoscaling Performance Measurement Tool

Valentin Lamonos





Department of Informatics

Technical University of Munich

Bachelor's Thesis in
Informatics: Games Engineering

Autoscaling Rules Evaluation with the Autoscaling Performance Measurement Tool

Autoskalierungsregelevaluierung mit dem Autoscaling Performance Measurement Tool

Author:	Valentin Lamonos
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Vladimir Podolskiy
Submission Date:	September 24th, 2018



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

September 24th, 2018

Walentin Lamonos

Acknowledgments

I would like to extend my thanks to both Anshul Jindal and Vladimir Podolskiy for all the advice, they have given me and who helped me thoroughly when it came to the functionality of the APMT.

Abstract

One of the keys to cloud computing has been the possibility to deliver consistent performance for reduced costs compared to traditional static implementations. With more and more autoscalers becoming available from different providers, it can be difficult to determine which of the implementations best suits one's needs. This raises the question on how to reliably compare autoscalers, since "[...] it is hard to distinguish [an overall] winner [3, Chapter 6]."

This thesis will tackle this problem by providing customers with an addition to the Auto-scaling Performance Measurement Tool (APMT [4]) that includes the ability to measure various autoscalers using experimental metrics proposed by [3]. This offers customers a wide range of generalised metrics that they can use to determine which autoscaler best covers their Quality of Service (QoS) requirements.

Contents

Acknowledgements	v
Abstract	vii
1 Introduction	1
1.1 Background	1
2 APMT Extension	3
2.1 Metrics	3
2.1.1 Operation-Oriented Metrics	3
2.1.2 User-Oriented Metrics	6
2.1.3 Metric Analysis	8
2.2 Proposed Solution	9
2.2.1 APMT Architecture Extension	9
2.2.2 Metric Analysis Dashboard (Monitoring)	10
2.2.3 Daemon (Monitoring & Load Generator)	11
2.2.4 Diagnostics Cluster (Load Generator)	12
2.2.5 Assembling the module	12
3 APMT Module Implementation	15
3.1 Challenges and Limitations	15
3.2 APMT Server-Side	16
3.3 APMT Client-Side	17
3.3.1 Generic Metrics	17
3.3.2 Operation-Oriented Metrics	18
3.3.3 User-Oriented Metrics	22
3.4 Scalectl Daemon	24
3.5 Diagnostics Cluster	29
4 Future Research	34
4.1 Threshold Tuning Algorithm	35
4.2 Diagnostic Cluster Upgrades	36
4.3 Combination	37
5 Conclusion	38

Appendix	38
Bibliography	41

1 Introduction

1.1 Background

"Ability to horizontally scale IT resources up or down dynamically as per changes in workload conditions, often termed as elasticity of cloud, is one of the key features offered by clouds." [6, Chapter 1]. Picking the right autoscaling solution for your service can be a tricky task, since the type to pick relies heavily on the application being scaled. The Autoscaling Performance Measurement Tool (APMT, now named "ScaleX") introduced in [4] already offers some metrics that allow us to deduce the responsiveness and overall performance of an autoscaler. The extension of the APMT introduced by this thesis will allow customers to monitor additional experimental metrics [3] allowing them to better grasp the pros and cons of different cloud application configurations. This is particularly useful since it is usually not apparent which approach (single-layered vs. multi-layered) will perform best.

Furthermore, we will provide comparison results of a mock performance test on the multi-layered cluster autoscaler provided by Amazon Web Services. This test involves a so-called "auto scaling group" that runs a web application. This thesis extension will then calculate additional metrics that then allow a user to determine the most optimal settings for their auto scaling group. Later possible extensions to this tool would be other cloud autoscalers supplied by other providers like *Azure+Kubernetes*, *GCE+Kubernetes* & *OpenStack+Kubernetes*.

In order to cover most use cases we split the metrics into two categories:

Operation-Oriented Metrics and *User-Oriented Metrics*.

The *Operation-Oriented Metrics* are defined as metrics that can be witnessed from the point of view of a cloud application. These metrics are particularly useful for HPC applications, since they do not rely on response times, response latency and error-rate. Their main metrics are the total completion time of all tasks and the operating cost.

The *User-Oriented Metrics* are defined as metrics trackable from outside the application by, for example, a separate diagnostics cluster or an application load balancer. These metrics are oriented towards end user satisfaction. The error & successful response rates are one of the distinguishing measurements of a cloud application built for handling user requests.

As follow up research and since both the AWS cluster-autoscaler and Kubernetes' HPA algorithm [2, Algorithm 1] use an average CPU usage threshold, in the scope of this thesis we also come up with a tuning algorithm [Algorithm 1] that simulates the cluster scaling multiple times and then outputs the most ideal value for the scaling threshold.

2 APMT Extension

2.1 Metrics

2.1.1 Operation-Oriented Metrics

1. **Wrong-Provisioning Timeshare** [3, Section 3.3]

Let *demand* be the "[...] minimal amount of resources required for fulfilling a given performance-related service level objective (SLO)" [3, Section 3.1]. "Accordingly, the *supply* is the monitored number of provisioned resources that are either idle, booting or processing tasks" [3, Section 3.1]. Whilst the under- and over-provisioning accuracy metric deals with the amount of resources available, the wrong-provisioning timeshare deals with the "[...] overall timeshare spent in under- or over-provisioned states [...]" [3, Section 3.3]. Since the accuracy metric does not provide a proper per time fraction observation, and it is therefore hard to determine the actual cause of demand and supply fluctuations, the *wrong-provisioning timeshare* metric (t_U & t_O) is introduced by [3, Section 3.3].

$$t_U := \frac{1}{T} \sum_{t=1}^T (\text{sign}(d_t - s_t))^+ \Delta t, \quad (2.1)$$

$$t_O := \frac{1}{T} \sum_{t=1}^T (\text{sign}(s_t - d_t))^+ \Delta t, \quad (2.2)$$

where

t is the time

d_t is the resource demand

s_t is the resource supply

t_U is the total time spent under-provisioned [Figure 2.1, U]

t_O is the total time spent over-provisioned [Figure 2.1, O]

Δt is the time elapsed between two subsequent measurements

T is the time horizon of experiment in time steps

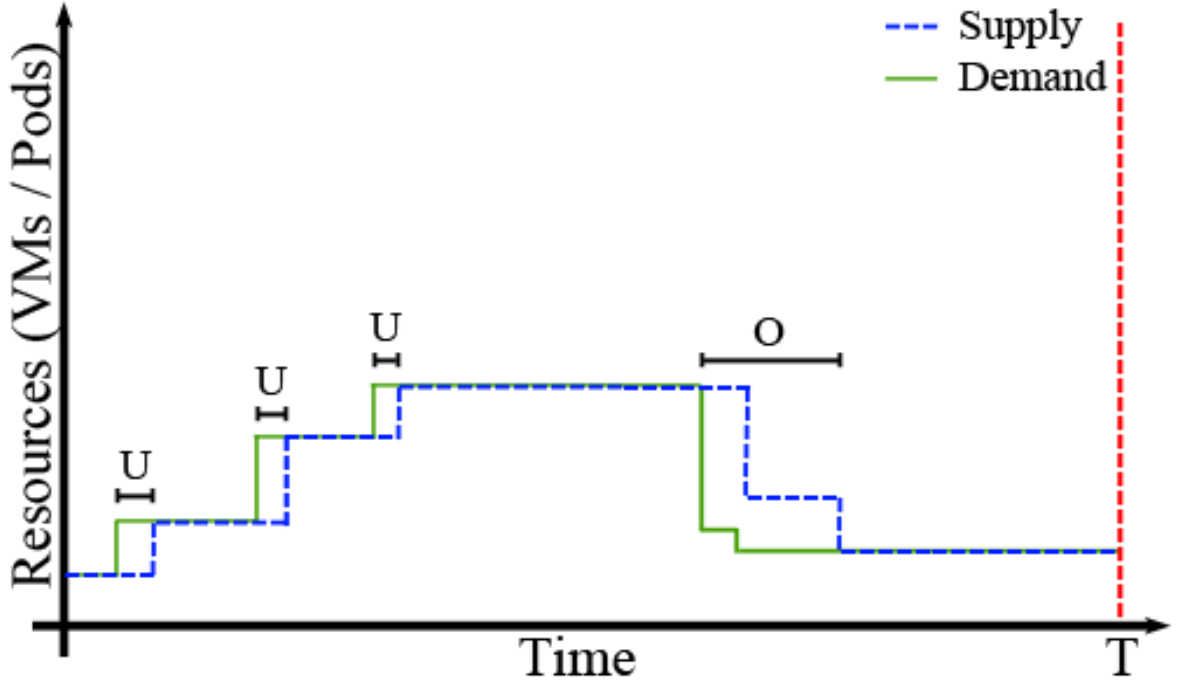


Figure 2.1: "The supply and demand curves illustrating the under- and over-provisioning periods ([U] and [O]) [...]" [3, Figure 2]

2. Resource Allocation Bounds

Since some users are limited by the maximum number of VM instances by their public cloud providers, we introduce a metric dealing with the "extremes" during the experiment's *time duration* T . The *upper bound metric* b_U provides us with the highest "peak" of resources during the experiment, whilst the *lower bound metric* b_L provides us with the lowest point of resources allocated, which is usually the initial amount at $t = 0$.

Additionally, we will use the difference of the two metrics to determine the *boundary scale metric* b_S of the resources during the time duration.

$$b_U := \max(\{s_t : t = 0, \dots, t_T\}), \quad (2.3)$$

$$b_L := \min(\{s_t : t = 0, \dots, t_T\}), \quad (2.4)$$

$$b_S = |b_U - b_L|, \quad (2.5)$$

where

t is the time

b_U is the upper bound metric [Figure 2.2]

b_L is the lower bound metric [Figure 2.2]

b_S is the boundary scale metric [Figure 2.2]

s_t is the resource supply at time t

T is the time horizon of experiment in time steps

3. Operating Expense

One of the advantages of the cloud is the on-demand usage, this is why it is imperative to take the running costs into account during each experiment. Depending on the Public Cloud Provider, the billing policies may vary from second to hourly billing. In addition some cloud providers enforce a minimum billing time of 1 minute, making it hard to determine during rapid up- & down-scaling how a user's costs will be calculated.

We introduce a user-defined *operating expense metric* e_T , which sums up the amount of resources supplied at each time step. We also round up the result, since cloud providers tend to see fractions of a time step (1 minute of 1 hour) as a full time step and tend to bill customers accordingly. It is also noteworthy, that we consider all VMs used in the experiment as interchangeable (of the same type).

$$e_T := \left\lceil \sum_{t=1}^T s_t \Delta t \right\rceil, \quad (2.6)$$

where

e_T is the operating expense metric [Figure 2.2]

s_t is the resource supply at time t

T is the time horizon of experiment in time steps

Δt is the time elapsed between two subsequent measurements

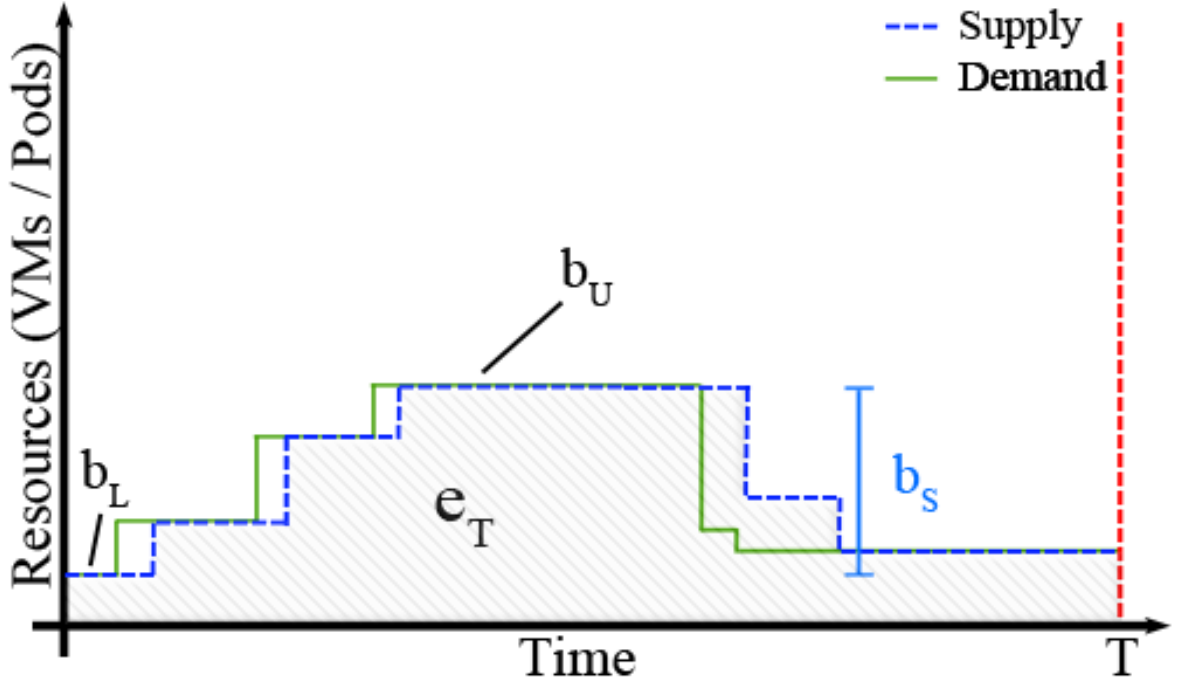


Figure 2.2: Visualization of *operating expense* e_T , *upper bound* b_U , *lower bound* b_L and *boundary scale* b_S .

2.1.2 User-Oriented Metrics

1. Average Response Time

The *average response time* r_t metric deals with the average amount of time a response takes to deliver the result to the client. It is noteworthy that we will track this value from outside of the operator's point of view, contrary to [7, Section 3.b.], using a separate static diagnostics cluster. This cluster will also provide us with machines that allow us to send customizable traffic to our scaled application.

The *response time* will be the *request round trip time* r_r plus the actual workflow processing time of the response on the server. "A *workflow* [, in our case,] is a set of tasks with precedence constraints among them" [3, Section 2.3]. Furthermore, we define "[t]he *makespan* [r_m] of a workflow [as] the time between the start of its first task until the completion of its last task" [3, Section 3.5].

$$r_t := \frac{1}{T} \sum_{t=1}^T (r_r + r_m) \Delta t, \quad (2.7)$$

where

r_t is the average response time

r_r is the round trip time of a request

r_m is the makespan of a workflow

T is the time horizon of experiment in time steps

Δt is the time elapsed between two subsequent measurements

2. Average Request Latency

The *average request latency* r_l sums up and averages the round-trip times (RTT) during our experiment duration T . Since we will not be able to send ICMP pings (OSI layer 3) through the load balancer (OSI layer 4), this metric might be of limited use. However it is still vital to monitor changes to the latency as it also indicates whenever the errors are occurring only on the load balancer (high latency) or inside the actual cluster (low latency & high error rate).

$$r_l := \frac{1}{T} \sum_{t=1}^T r_r \Delta t, \quad (2.8)$$

where

r_l is the average request latency

r_r is the round trip time of a request

T is the time horizon of experiment in time steps

Δt is the time elapsed between two subsequent measurements

3. Successful & Error Response Count

One of the most important metrics is the amount of errors produced during the scaling. Optimally, we want this value to be close to or 0. Practically, since in for example Kubernetes' Horizontal Pods Auto-scaling algorithm (KHPA) [2, Algorithm 1], the scaling speed heavily depends on the CPU utilization threshold, errors might still occur.

One of the main objectives of this thesis is to provide users with an easy way to determine or fine-tune the CPU threshold of the utilized cloud autoscaler given the parameters: VM count (and instance type) and budget. Allowing them to maximize the usage of their resources whilst keeping their budget in check.

Again, we will use the separate diagnostics cluster to simulate traffic to the target load balancer based on a provided pattern. Trivially, the *error response count* r_e is defined as the amount of server-side error responses (HTTP: 5XX) and client-side error responses (HTTP: 4XX), whilst the *successful response count* r_s is defined as the amount of successful responses (HTTP: 2XX/3XX). We declare the *total response count* r_{total} as follows:

$$r_{total} := \sum_{t=1}^T (r_e + r_s) \Delta t, \quad (2.9)$$

where

r_{total} is the total amount of responses

r_e is the amount of error responses

r_s is the amount of successful responses

T is the time horizon of experiment in time steps

Δt is the time elapsed between two subsequent measurements

2.1.3 Metric Analysis

To make comparisons between different scaling policies easier, we need to combine the metrics we have collected. The overall goal of the APMT extension is to provide the user with the ability to run and compare multiple experiments with different scaling parameters. In order to determine which experiment performed best, we need to run multiple tests and then compare the results.

We can then sort by the following metrics:

- Best *error response count* (least errors)
- Best *boundary scale* (least scaling)
- Best *operating expense* (smallest cost)
- Best *over-provisioning* (least over-provisioning)
- Best *under-provisioning* (least under-provisioning)
- Best *average response time* (fastest responses)
- Best *average request latency* (lowest latency)

After the comparison, the experiment that matches all these metrics best is selected. The user can then rerun the experiment with tighter constraints (fewer VMs, higher CPU scaling threshold) and see if their budget can be reduced even further whilst keeping the error rate below a provided threshold. This procedure can be repeated until the most cost-effective or least error-prone parameters are found.

We later discuss an automated approach to this method called CPU threshold tuning algorithm, combined with another addition to the APMT interface [Section 5].

2.2 Proposed Solution

2.2.1 APMT Architecture Extension

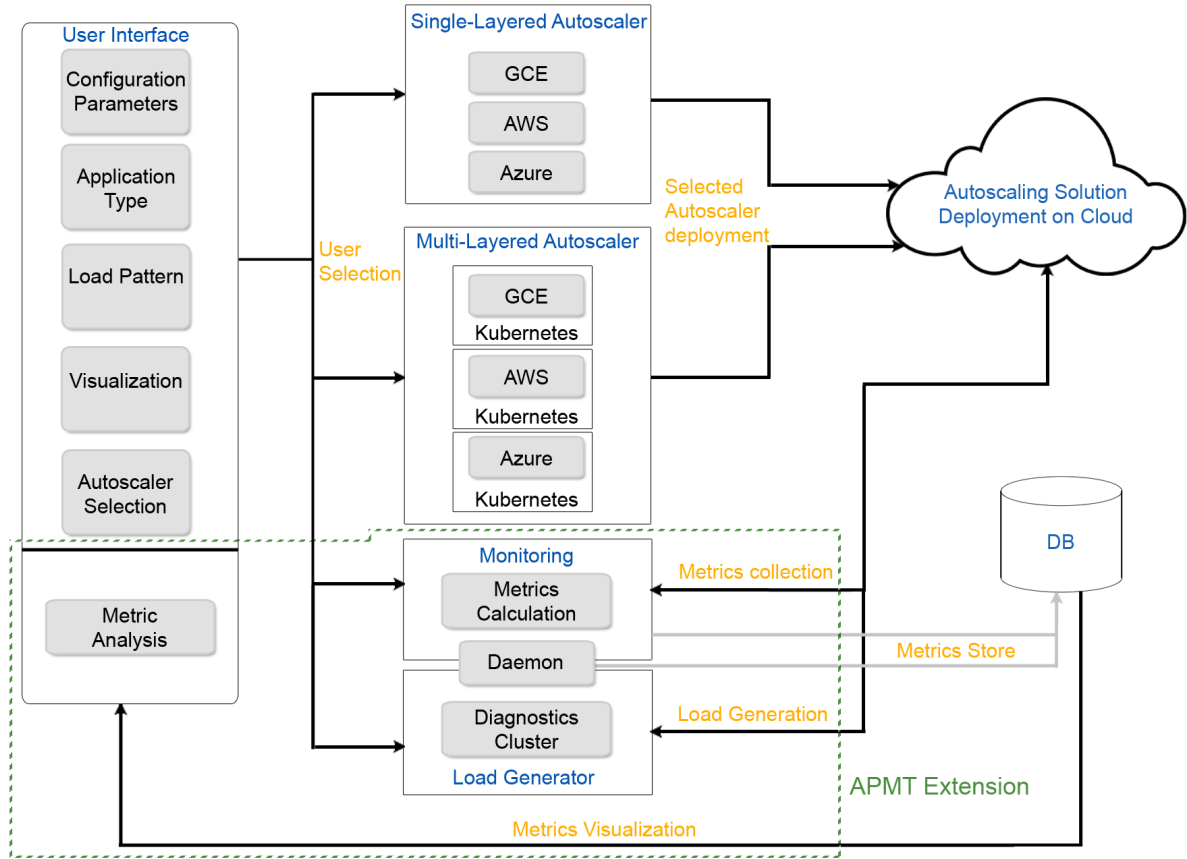


Figure 2.3: Extended APMT architecture [5, Figure 1].

To allow the implementation of the aforementioned metrics into the APMT, we have to extend the infrastructure given by [5]. We will start by splitting the originally proposed Load Generator & Monitoring module into two [Figure 2.3].

In addition to retrieving the metrics from the Public Cloud Provider, *Monitoring* is now also responsible for calculating the new metrics, instead of merely displaying them.

The *Load Generator* is, depending on the need of the application, responsible for starting and monitoring the static *Diagnostics Cluster*, which is used for generating traffic if required.

Lastly, we add a *Metric Analysis* dashboard to the *User Interface*, that provides the user with the calculations done by the *Monitoring* module.

All in all, we extend the APMT tool from being a mere performance measurement tool for autoscalers, to a tool that allows users to benchmark their own custom cloud applications. In the scope of this thesis, we will show off one specific use case:

A static load-balanced web server cluster handling incoming HTTP requests. It is noteworthy, that this thesis will not be dealing with the performance of the Horizontal Pod Autoscaler (HPA), since the APMT implementation of launching a Kubernetes cluster has been delivering inconsistent results. The scope of this thesis therefore, only includes the performance measuring of the VM auto scaling.

2.2.2 Metric Analysis Dashboard (Monitoring)

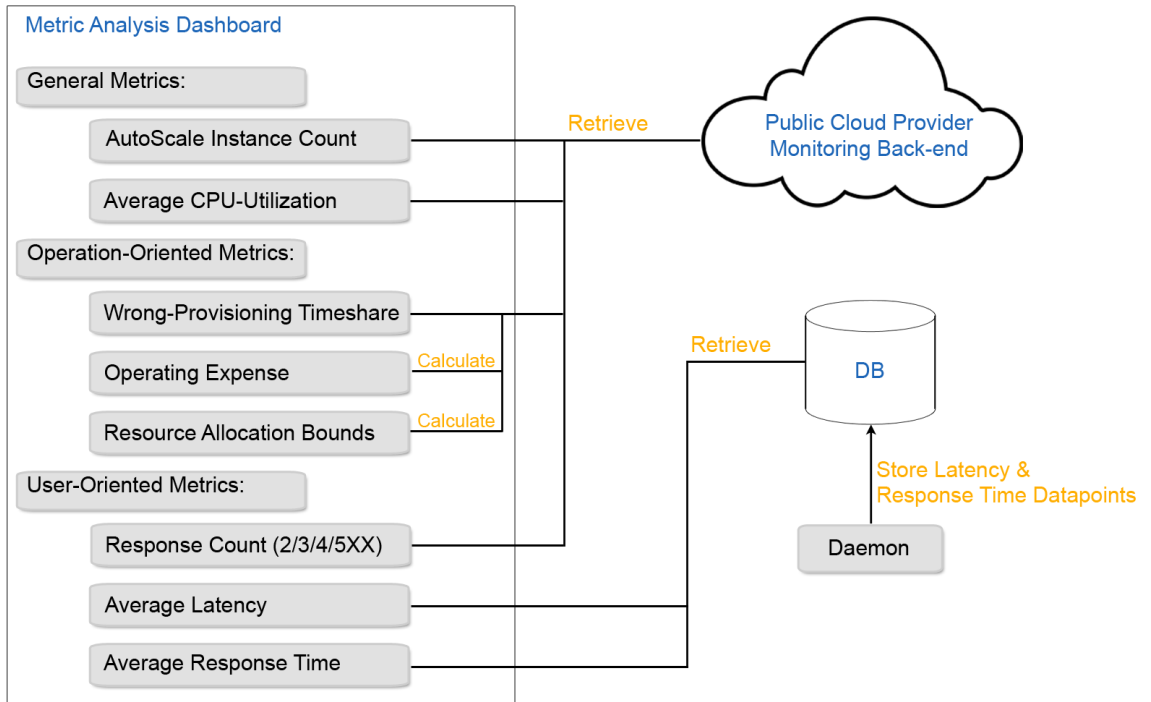


Figure 2.4: User Interface: Metric Analysis Extension Dashboard

The *Metric Analysis Dashboard* allows the user to view the results generated by an auto scale cluster over the past hour. Based on this data the user can then chose to rerun experiments with adapted parameters. This allows the user to slowly converge to the optimal parameters which best balance their requirements.

As an extension of the APMT this is written in *Javascript (Node.js)* and uses AWS's Cloudwatch to continuously query metrics from the active auto scaling cluster and then visualizing them to the user using *Chart.js*.

Figure 2.4 showcases that the dashboard has to retrieve its data from multiple sources. That data only partially yields from the Public Cloud Provider, since some, namely *Operating Expenses* and *Resource Allocation Bounds*, dependent on the results of others. Additionally, some locally generated metrics (*Average Latency & Average Response Time*)

have to be retrieved from the database, as the Public Cloud Provider Monitoring Backend does not track such metrics.

2.2.3 Daemon (Monitoring & Load Generator)

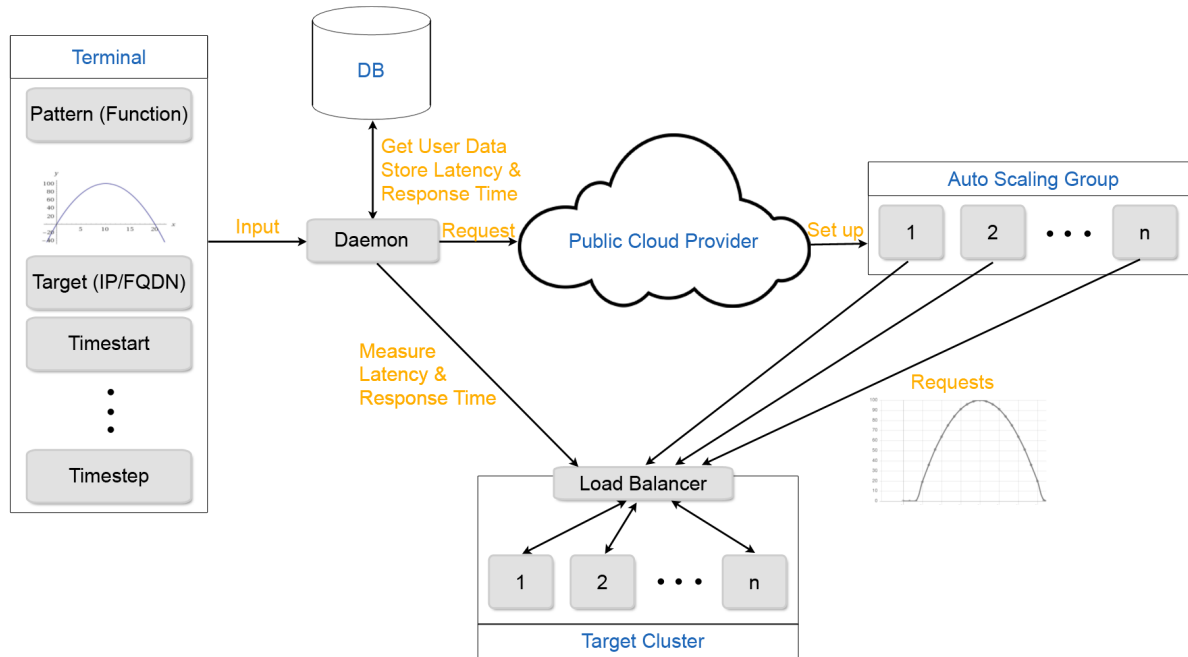


Figure 2.5: Daemon and Diagnostic Cluster Setup

The *Daemon* is a python application responsible for spawning and deleting *Diagnostics Clusters*. The *Daemon* takes the provided parameters by the user terminal and provisions an auto scaling cluster from the Public Cloud Provider. The resulting cluster then targets the provided IP or FQDN with requests following the provided pattern function over time [Figure 2.5]. Another task the *Daemon* fulfils is the collecting of two of the *User-Oriented Metrics*:

- *Average Response Time*
- *Average Request Latency*

This is due to the fact that *Node.js* does not support threads and AWS Cloudwatch does not track latency and response times from external sources. The successful (2XX, 3XX) & unsuccessful (4XX, 5XX) responses are directly read from the AWS application load balancer since it will receive all the requests to the cluster independent of any client-side connectivity issues.

The implementation of the *Daemon* only allows for one active user and one active *Diagnostics Cluster* since it's made for prototyping purposes. The *Daemon* can be set up as an

addition to the APMT without interfering with its functionality. The only requirement is that the *Daemon* is able to reach, read & write from and to the APMT database. It can not function independently as it requires the user data to be read from the APMT.

2.2.4 Diagnostics Cluster (Load Generator)

The *Diagnostics Cluster* [Figure 2.5] provides an external source of traffic for an application. Initially, the cluster size is set by the customer using the *Daemon*, additional parameters are the start & end time (as UNIX time), time step (at which rate the request should be sent) and the desired AWS instance type. The user can configure the target (IP or FQDN) and the function representing the amount of traffic that is desired at each time step. It is noteworthy, that the *Diagnostics Cluster* is static and therefore the user has to make sure to pass the required amount of machines of a sufficient size to actually generate the traffic pattern passed to the cluster.

The cluster target can be an arbitrary IP or FQDN, even outside of the Public Cloud Provider's network, it can therefore also be used to benchmark other external applications. The provided function is sampled each minute starting from the passed start time up until the end time. The result of each sampling then represents the amount of requests that will be sent during that minute. The performance and quality of the results tightly depends on the amount of requests that each of the nodes in a *Diagnostics Cluster* has to send, it is up to the user to provision VMs that are powerful enough to send the demanded amount of requests. Each additional machine added to the *Diagnostics Cluster*, trivially, acts like a multiplier on that current time steps request count.

2.2.5 Assembling the module

The combined version of the draft [Figure 2.6], showcases the various connections between the introduced parts of the implementation. Chapter 3 is dealing with the technical details of the implementation of each section. We will, however, first discuss some challenges and limitations that were encountered during the making of each module [Section 3.1]. We then look at the extension of the *Metric Analysis Dashboard* in Section 3.3, which involves revamping and reworking existing code of the APMT. Section 3.4 deals with the implementation of the *Scalectl Daemon*, which is responsible for collecting latency and response time metrics and the spawning of *Diagnostic Clusters*. Finally, section 3.5 then deals with the work conducted regarding the creation of *Diagnostic Clusters*, which involved the creation of launch configurations and the deployment to AWS using the *boto3*¹ library.

¹<https://aws.amazon.com/sdk-for-python/>

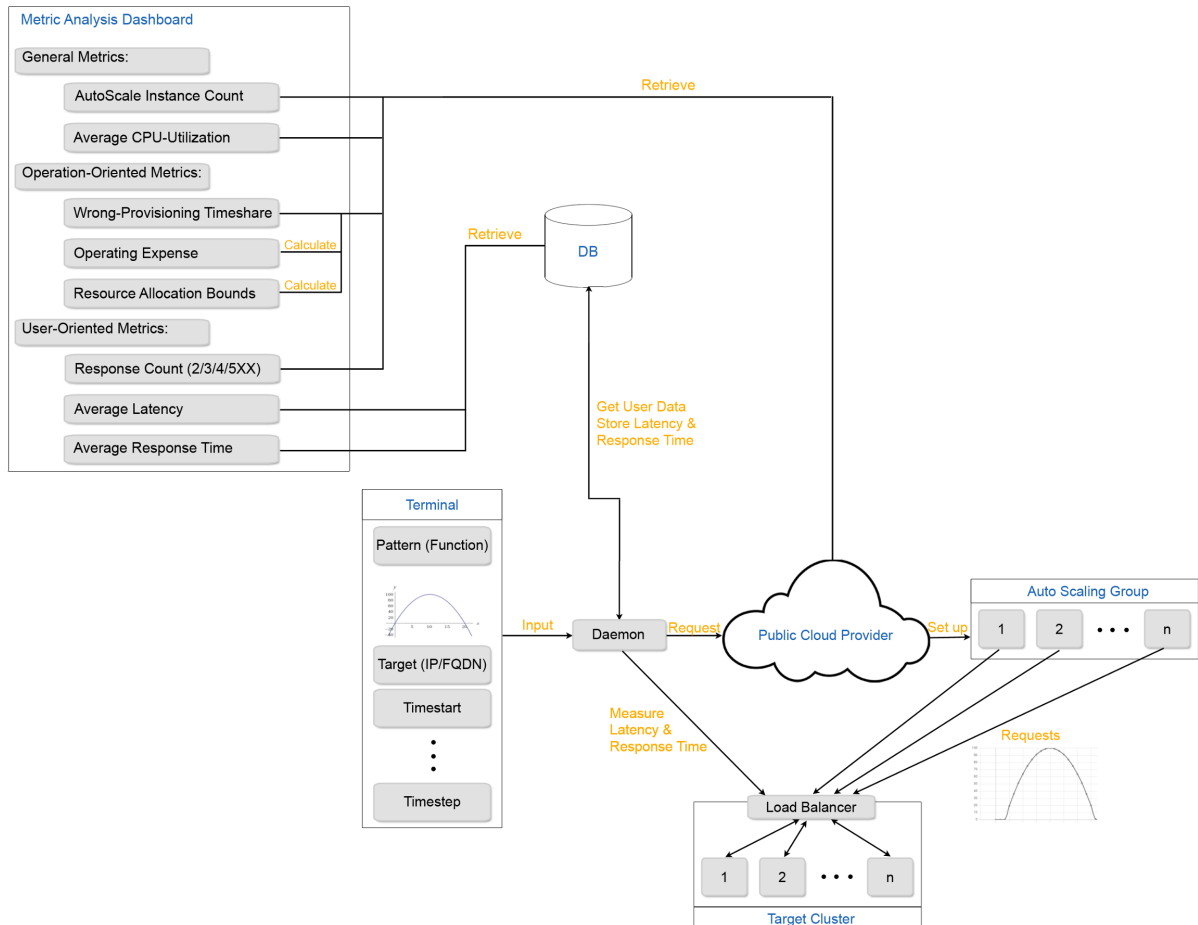


Figure 2.6: Combined Structure

3 APMT Module Implementation

3.1 Challenges and Limitations

The APMT tool was implemented using *Node.js* and is therefore only single-threaded and event-based. Whilst this is ideal for displaying remote and static data, it is a challenge to actually retrieve new metrics during runtime from a cluster. This is the reason why the *Python* based *Scalectl Daemon* is set up to record asynchronous data. However, due to the nature of proposed architecture, the *Scalectl Daemon* only allows the setup and tracking of one user and one cluster at a time.

Additional constraints arise from choice of the the Public Cloud Provider, Amazon Web Services (AWS)¹:

- The minimum Auto Scaling Group scaling interval is 300 seconds, making it hard to react to burst like traffic peaks. This also delays the automatic termination of over-provisioned machines.
- Auto Scaling Groups are fixed at an instance type, meaning you can not have machines of various performance, memory or storage levels attached to the cluster.
- The smallest metric interval of Cloudwatch, the monitoring back-end of AWS, is 1 minute. Whilst it records multiple datapoints in that duration, you can only request certain summarized fields: Minimum, Maximum, Sum or Average; for each minute, limiting our insights into the traffic received during that time frame.
- The termination time of a machine is billed to the user, this is especially important in correlation to the *Over-Provisioning Timeshare* metric, since it is the main reason why such phases occur when using AWS.
- Costs calculation and the subsequent billing is delayed by a day when using AWS, making it hard to actually determine one's budget. This is the reason why the *Operating Expense* metric is calculated off the data retrieved from Cloudwatch, it allows us an instant insight into the costs of running an application for 1 hour, making it easier to predict possible costs.

The scope of this thesis did not include research or implementation of other Public Cloud Providers, we will therefore abstain from comparing AWS's service coverage (or similar points) to those of other providers.

¹<https://aws.amazon.com/>

3.2 APMT Server-Side

The original implementation of the APMT deployed the tool using a *docker-compose* service with the application and MongoDB database container being separated. Due to the nature of the *Scalectl Daemon* implementation requiring database access and also for simplicity reasons, the service was merged into a monolithic Docker container, containing the database, the APMT and the *Scalectl Daemon*.

Retrieving Cloudwatch data:

The code for retrieving metrics from AWS's Cloudwatch was already implemented inside the APMT for the "Generic Metrics" [Section 3.3.1] and therefore only required a mere extension. It involves setting up a parameters array *params* featuring the wanted metrics and the target dimension (ELB or Auto Scaling Group), then executing the *getMetricStatistics* function. Inside the callback function we can then retrieve the data which is an array of containing "datapoints" which each have a timestamp, statistics during that time step (sum, average, min, max) and unit [Source Code 3.1].

We can now nest this function with a new parameter array inside the callback function of the previous request, slowly filling our *dataAll* (return data) array.

```
1  [...]
2  var params = {
3      EndTime: new Date,
4      MetricName: 'HTTPCode_Target_4XX_Count',
5      Namespace: 'AWS/ApplicationELB',
6      Period: 60,
7      StartTime: new Date(d.getTime() - 60 * MS_PER_MINUTE),
8      Dimensions: [{
9          Name: 'LoadBalancer',
10         Value: lbname
11     }],
12     Statistics: ["Sum"]
13 };
14 cloudwatch.getMetricStatistics(params, function (err, data) {
15     if (err) console.log(err, err.stack);
16     // an error occurred
17     else {
18         dataAll.HTTPCode4XXCount = data;
19         // Nested repeat with different params
20         [...]
21     }
22 }
```

Source Code 3.1: Retrieving metric from Cloudwatch

3.3 APMT Client-Side

Note: The full code of the APMT extension can be found on GitHub².

To increase the responsiveness of the new dashboard an automatic page refresh was introduced to the existing dashboard [Section 3.3.1]. This was done by storing the chart references outside of the function and then wrapping the chart updates and data populating tasks into a `callUpdate()` function [Source Code 3.2]. Using `setInterval([...])` we can then refresh the page every 60 seconds, which coincides with the arrival of a new AWS Cloudwatch datapoint.

```

1  $(function() {
2      var myChartLatency;
3      [...]
4      var myChartErrorResponsecount;
5      function callUpdate() {
6          [...]
7      }
8  }
9  callUpdate();
10 setInterval(callUpdate, 60000);

```

Source Code 3.2: Automatic page refresh

3.3.1 Generic Metrics

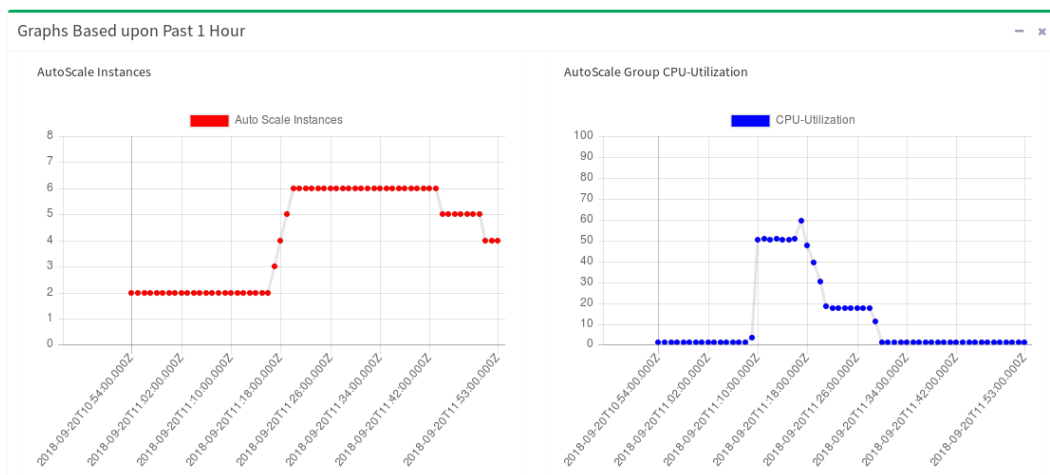


Figure 3.1: Original Metric Analysis Dashboard

²<https://github.com/CM2Walki/ScaleX>

The original metric analysis dashboard [Figure 3.1] is rendered using *Chart.js*³ and is keeping track of the amount of AWS Auto Scale instances and the average CPU Utilization of a group. The generic metrics dashboards was merely updated to feature clearer line connections and proper maximum values for aesthetic reasons.

Since the data displayed in the graphs is retrieved directly from Cloudwatch, *Diagnostic Clusters* data will not be tracked, as they have Cloudwatch disabled when launched. This also means that the the other metrics are not influenced by those clusters, meaning that additional costs might occur.

3.3.2 Operation-Oriented Metrics

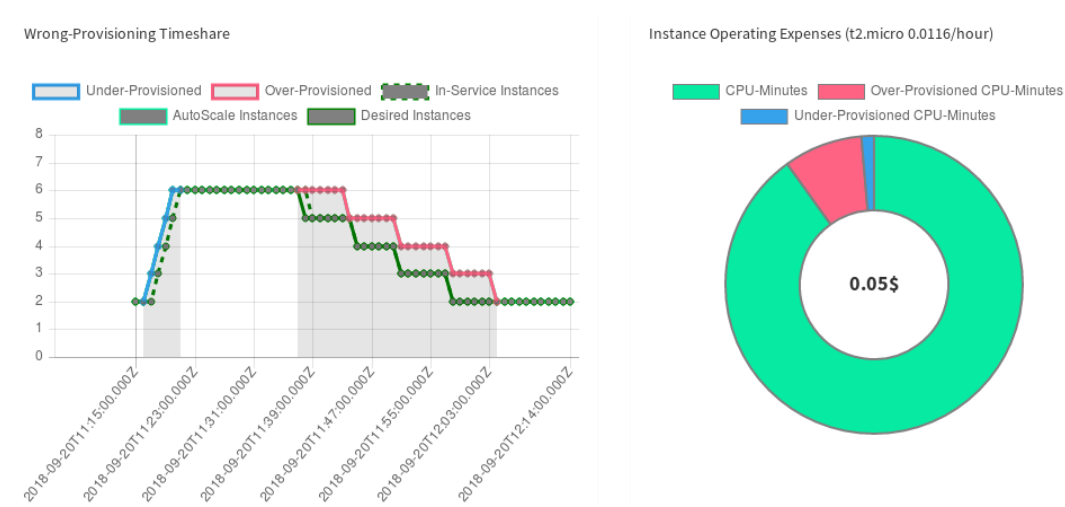


Figure 3.2: Wrong-Provisioning Timeshare & Instance Operating Expenses

As described in Section 2.1.1 the *Operation-Oriented Metrics* deal with the point of view of a cloud administrator. The *Wrong-Provisioning Timeshare* metric gives valuable intel on how long a cluster takes to scale in or out, visualized in Figure 3.2. AWS Cloudwatch offers multiple statistics that allow us to calculate the Over- and Under-provisioning phases of the cluster. As seen in the graph, we have to keep track of three separate datasets:

- *Desired Instances*: The amount of requested instances by the Auto Scaling Group.
- *In-Service Instances*: The amount of instances that are fully reachable by requests (not being started or terminated).
- *AutoScale Instances*: The amount of instances actually part of the Auto Scaling Group, regardless of their state.

³<http://www.chartjs.org/>

Trivially, we can now use the *Desired Instances* and the *In-Service Instances* at each time step and determine if we are in a under-provisioned state by subtracting the desired instances count from the active in service instance count [Source Code 3.3].

```

1  for(i=0;i<desiredInstancesDataArr.length; i++) {
2      // How many machines do we have
3      // vs how many do we need?
4      var diff = desiredInstancesDataArr[i].Average -
5                  inServiceInstancesDataArr[i].Average;
6      [...]
7      if (diff > 0) {
8          // Too few
9          valueunderprovisioning.push(
10             Math.abs(inServiceInstancesDataArr[i].Average + diff));
11             // Increment minutes spent under-provisioned
12             underprovisioningCosts = underprovisioningCosts + 1;
13             [...]
14         }
15         [...]
16     }

```

Source Code 3.3: Is cluster in under-provisioned state

```

1  for(i=0;i<desiredInstancesDataArr.length; i++) {
2      // How many machines do we have
3      // vs how many do we need?
4      [...]
5      var diff_over = desiredInstancesDataArr[i].Average -
6                      instancesDataArr[i].Average;
7      [...]
8      else if (diff_over < 0) {
9          // Too many
10         valueoverprovisioning.push(instancesDataArr[i].Average);
11         // Increment minutes spent over-provisioned
12         overprovisioningCosts = overprovisioningCosts + 1;
13         [...]
14     }

```

Source Code 3.4: Is cluster in over-provisioned state

Similarly, we can now calculate whenever the cluster currently is in an over-provisioned state by subtracting the *Desired Instances* count from the *AutoScale Instances* count [Source Code 3.4].

Since, during the process, we incremented *overprovisioningCosts* and *underprovisioningCosts* we already determined the *Wrong-Provisioning Timeshare*. Combined with the summing up of all CPU minutes [Source Code 3.5 (Line 3-7)], we can now populate our *Operating Expense* "doughnut" chart [Figure 3.2]. Since AWS uses per minute billing this graph provides a transparent feedback source regarding a user's budget usage.

```
1  var costs = 0;
2  [...]
3  for(i=0; i<instancesDataArr.length; i++) {
4      [...]
5      // Add up all instances in this time step
6      costs = costs + instancesDataArr[i].Average;
7  }
8  [...]
9  // Calculate based on per minute price (full costs)
10 var totalcosts = (costs*(instancecost/60)).toFixed(2);
11 // Subtract overprovisioningCosts for displaying purposes
12 costs = costs - overprovisioningCosts;
13 // Set centre text element
14 document.getElementById("lboperatingexpense").textContent =
15                                     `${totalcosts}\$`;
16 // Draw charts
17 [...]
```

Source Code 3.5: *Operating Expense* calculation

```
1  for(i=0; i<instancesDataArr.length; i++) {
2      if (instancesDataArr[i].Average > max) {
3          max = instancesDataArr[i].Average; // New maximum
4      }
5      if (instancesDataArr[i].Average < min) {
6          min = instancesDataArr[i].Average; // New minimum
7      }
8  }
9  var boundaryscale = Math.abs(max-min)
```

Source Code 3.6: *Resource Allocation Bounds* calculation

To calculate the boundary scale metric, we first have to find our upper & lower bounds. This is done by iterating the data points and then finding the highest and lowest amount of VMs in the retrieved data set [Source Code 3.6]. Then we can use the absolute value of the subtraction of the upper bound from the lower bound as our boundary scale. The data is then loaded into a "bar" chart [Figure 3.3] and displayed to the user. The *Resource Allocation Bounds* give cloud operators a quick overview to how much one's cluster has scaled, with lower values of the boundary scale indicating smaller overall cluster sizes.

Resource Allocation Bounds

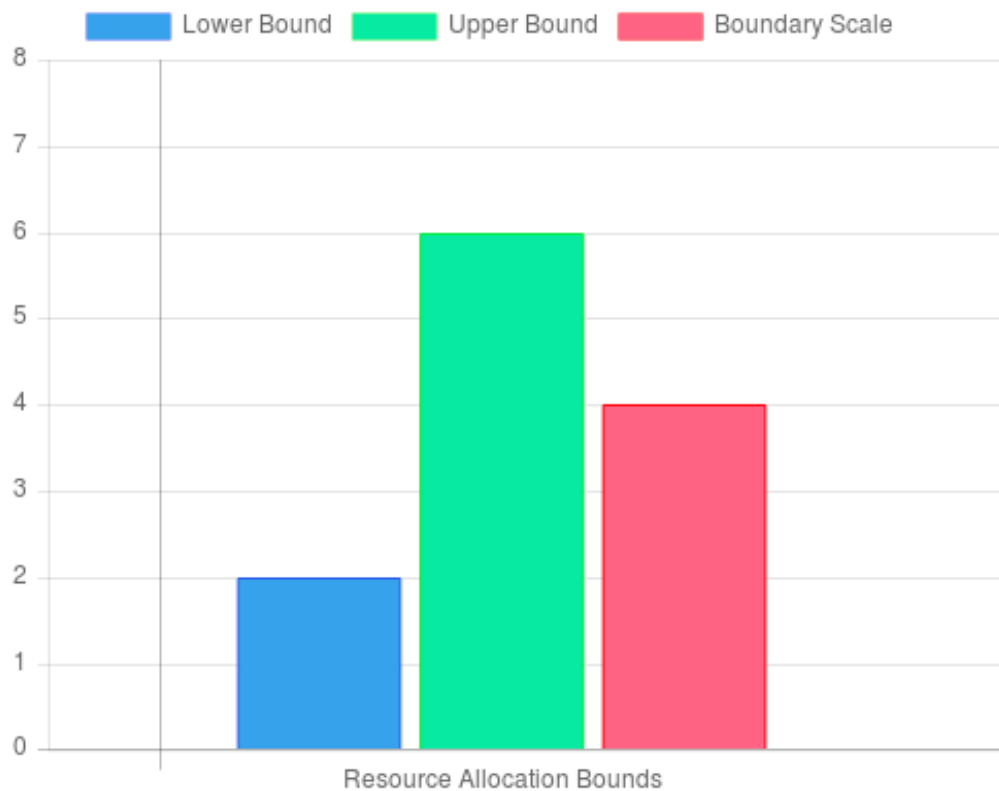


Figure 3.3: *Resource Allocation Bounds* graph

3.3.3 User-Oriented Metrics

As mentioned in Section 2.1.2, the *User-Oriented Metrics* deal with measurable data from the end-users's point of view. This gives us valuable insights into the performance of our web-application whilst it is being scaled. In order to track responsiveness, the APMT uses its *Scalectl Daemon* to continuously query the end point of an application.

Average Latency & Average Response Time

AWS Elastic Load Balancers do not reply to ICMP pings on layer 3 which is why the Daemon uses an opening of a TCP socket on layer 4 and the subsequent measured timings to simulate our *Average Latency*. It is noteworthy, that the latency measurement will not get relayed to a cluster machine, because of the *Scalectl Daemon* never sending anything on the Application Layer (layer 7).

Similar to the *Average Latency*, measuring the *Average Response Time* is done by the Daemon, however in this case, a whole request on the Application Layer (HTTP) is sent which will be routed to one of the Auto Scaling Group instances by the Elastic Load Balancer. The Daemon then reads the entire response (header and payload); the total time it takes for the response to return and to be processed is therefore the response time.

This process gets repeated up to six times per minute for each metric by the *Scalectl Daemon*, the resulting average data point and its time stamp are then added to MongoDB, having a similar structure as a AWS Cloudwatch data point (see Section 3.4).

The APMT dashboard can now read the data points and populate the appropriate graphs exactly as if the data points were delivered by AWS Cloudwatch [Figure 3.4].

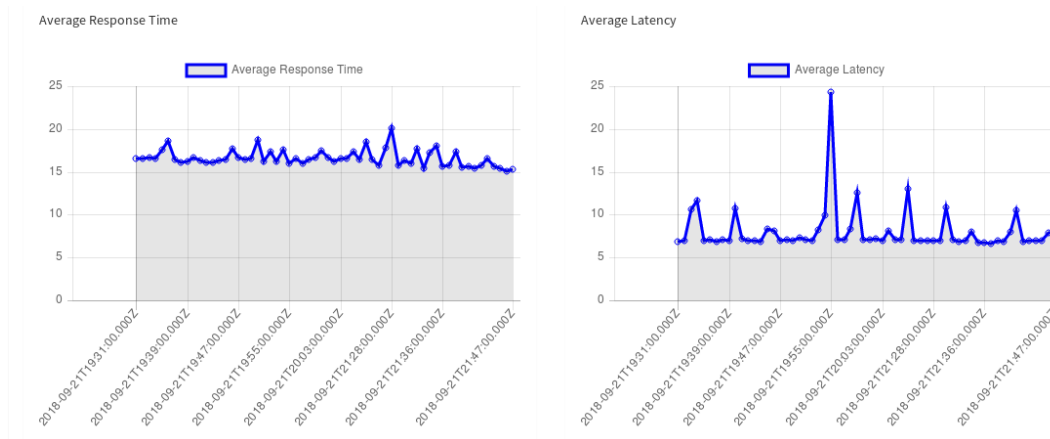


Figure 3.4: User-Oriented Metrics Dashboard
Average Response Time & Latency

Successful & Error Response Count



Figure 3.5: User-Oriented Metrics Dashboard
Successful & Error Response Count

AWS's Elastic Load Balancers differentiate between errors that occurred on the load balancer (HTTPCode5XXCountELB) and errors that happened on the target group's machines (HTTPCode5XXCount). For simplicity reasons, we sum these values up to make the graphs more readable. These metrics can be retrieved, just as with the generic metrics, from AWS's Cloudwatch.

The resulting data points can then be added up and displayed in their appropriate charts, using the "timestamp" as their x- and the "Sum" as their y-value [Figure 3.5].

3.4 Scalectl Daemon

Note: The full code of the Scalectl Daemon can be found on GitHub⁴.

The *Scalectl Daemon* is a pure Python based Daemon, it takes care of creating *Diagnostics Clusters* and monitoring the target for changes in latency and response times. It delivers its data to the APMT by storing it into a separate user-specific collection in MongoDB, that can then be read whilst accessing the dashboard. The implementation is also almost fully object-oriented. In addition, the daemon gets turned into a binary using *pyinstaller*⁵ and *Make*⁶.

The Daemon is split into a client and daemon part, with the first "scalectl" command spawning the daemon if it does not exist, yet. Subsequent clients can then communicate with the daemon using a REST API [Source Code 3.7].

```
1  if __name__ == "__main__":
2      # Setup variables
3      alias = 'scalectl'
4      api = 'api/v1/'
5      host = 'localhost'
6      port = 8085
7      # Setup scalectl client
8      scalectlclient = ScaleCtlClient(alias, port, api)
9      scalectlclient.setup_hostfile()
10     # Setup scalectl daemon (if not already running)
11     scalectldaemon = ScaleCtlDaemon('/tmp/scalectl-daemon.pid',
12                                     'ScaleAPI', host, port)
13     [...]
```

Source Code 3.7: Starting the *Scalectl Daemon* and Client

Daemon Server

The Daemon server is the heart of the *Scalectl Daemon*. The base is formed by a Python 2.7 daemon module⁷. Upon initialization of the daemon object, we overwrite the *run([...])* method, instead of running an infinite loop to keep the process alive we spawn a *gevent*⁸ web server serving a *Flask*⁹ web application [Source Code 3.8].

⁴<https://github.com/CM2Walki/ScaleX-tune>

⁵<https://www.pyinstaller.org/>

⁶ftp://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_2.html

⁷<http://web.archive.org/web/20131017130434/>

http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

⁸<http://www.gevent.org/>

⁹<http://flask.pocoo.org/>

```

1  # ORM object for Mongodb (keeps connection and queries)
2  mongodbORM = MongoDatabase('localhost', 27017)
3  userStorage = Storage() # user setup info is stored here
4  # daemon remote commands
5  commandList = DaemonCommands(mongodbORM, userStorage)
6  [...]
7  class ScaleCtlDaemon(Daemon):
8      def __init__(self, pidfile, name, host, port):
9          Daemon.__init__(self, pidfile)
10         self.app = None
11         self.host = host
12         self.port = port
13         self.name = name
14
15     def run(self):
16         self.app = Flask(self.name) # Setup Flask
17         V1View.register(self.app) # Register route
18         http_server = WSGIServer((self.host, self.port),
19                                 self.app) # Setup server
20         http_server.serve_forever() # Start server

```

Source Code 3.8: Setting up the daemon server

The *V1View* object holds all routes. It is responsible for decoding the arguments when necessary and then calling the appropriate follow-up function. The *DaemonCommands* object is “building the context” of a user that wants to log in. Since the *Scalectl Daemon* only supports one active user, there will always only be one [user] *Context* object. The *Context* object gets created by the *DaemonCommands* object, which also provides it with the user information it has previously retrieved from the APMT database [Source Code 3.9].

Since the *Context* object is storing all the AWS connections as well as the reference to the latency and response time measuring thread, we can quickly discard an active user by just replacing the object in *DaemonCommands* forcing the garbage collector to tidy up the orphaned *Context* object and all of its associated variables and objects.

```

root@4f765c6859bd:/usr/src/apmt# scalectl setup Walki
No active user detected! Setting up Walki
User setup successful! Detected 0 running scalectl auto scaling cluster(s)
Found scalectl-cluster launch configuration

```

Figure 3.6: Scalectl Daemon: Setting up a user

```
1 class DaemonCommands:
2     def setup_user(self, username):
3         # Try to find the user in the APMT database
4         result = self.mongodbORM.
5             get_user_info_from_name(username)
6         if result is not None:
7             # Check if we have the fields we need
8             if result["username"] and
9                 [...] and result["awssubnetid2"]:
10                # Setup user
11                self.userStorage.set_username(
12                    result["username"])
13                [...]
14                self.userStorage.set_aws_subnetid2(
15                    result["awssubnetid2"])
16                # Setup AWS connection
17                self.userContext = Context(self.userStorage,
18                                           self.mongodbORM)
19                # Retrieve running clusters
20                response = self.userContext.build_context(
21                    self.userStorage)
22                [...]
```

Source Code 3.9: Setting up a user

Latency & Response Time Measuring

The latency and response measuring takes place inside a separate thread. It uses an thread object python implementation¹⁰. As mentioned in Section 3.3.3, AWS Load Balancer do not respond to ICMP (layer 3) pings, this is why the *Scalectl Daemon* comes with a simple “TCPping” (layer 4) implementation. This involves initiating a TCP connection and waiting for it to be acknowledged. If we measure the blocking time of `socket.connect([...])` [Source Code 3.10 (line 25)] we get a similar latency value as with a ICMP ping. Since this is merely a TCP handshake, we are not sending any application data and are therefore not routed to a cluster instance. However, for the response time we want to access an actual instance, this is why we initiate a run-of-the-mill request aimed at the HTTP port [Source Code 3.10 (line 31)]. The thread then measures the time it took to read the headers and the payload. After measuring both metrics, the results are summed up and divided by *probes* (the amount of probes in this time step), thus yielding the average for that data point. The values are then saved into the MongoDB.

¹⁰<http://sebastiandahlgren.se/2014/06/27/running-a-method-as-a-background-thread-in-python/>

```
1 class Updater(object):
2     def __init__(self, interval, target, mongodb, username):
3         [...]
4         self.thread = threading.Thread(target=self.run, args=())
5         self.thread.daemon = True
6         self.thread.start()
7         self.stop = False
8
9     def run(self):
10        self.mongodb.create_perf_data_db(self.username)
11        counter = 1
12        probes = 0
13        total_lat = 0.0
14        total_resp = 0.0
15        while True:
16            if not self.stop:
17                now = datetime.datetime.now()
18                if now.second > 0:
19                    if counter > 10:
20                        # Latency
21                        # Open socket
22                        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23                        # Get starttime
24                        start = time.time()
25                        s.connect((str(self.target), 80))
26                        total_lat = total_lat + (1000 * (time.time() - start))
27                        s.close()
28                        # Response Time
29                        start = time.time()
30                        # Process the header and the payload!
31                        r = requests.get('http://' + (str(self.target)),
32                                       stream=True)
33                        for chunk in r.iter_content(chunk_size=1024):
34                            print ""
35                        total_resp = total_resp + (1000 * (time.time() - start))
36                        counter = 1
37                        probes = probes + 1
38                        counter = counter + 1
39                    else:
40                        if probes > 0:
41                            total_lat = total_lat / probes
42                            total_resp = total_resp / probes
43                            [...]
44                            self.mongodb.add_datapoint(self.username, total_lat,
45                                                         total_resp, int(time.time()))
46                        probes = 0
47                        total_lat = 0.0
48                        total_resp = 0.0
49                        counter = 1
50                        time.sleep(self.interval)
```

Source Code 3.10: Latency & Response Time Measuring

Daemon Client

```
root@4f765c6859bd:/usr/src/apmt# scalectl
Usage: scalectl COMMAND

Commands:
  start          Starts the scalectl-daemon
  stop           Stops the scalectl-daemon
  restart        Restarts the scalectl-daemon
  setup          Fetches AWS information from the ScaleX database
  cluster        Controls and Creates AWS autoscaling clusters
```

Figure 3.7: *Scalectl Daemon* commands

The client can now communicate with the daemon using the REST API. The Python implementation follows a similar pattern as the server-side application. The *ScaleCtlClient* object parses parameters passed through the terminal, and if required, encodes them in base64. The encoding has to be done for mathematical functions as they generally do not perform well as a plain text http parameter [Source Code 3.11].

The first command a user has to execute is the "scalectl setup [username]" command. This tells the daemon to retrieve the user information of the provided user name from the APMT database [Figure 3.8].

```
1  class ScaleCtlClient:
2      [...]
3      def build_request(self, query):
4          return '%s%s' % (self.url, query)
5      [...]
6      def setup_user(self, username):
7          try:
8              response = requests.get(self.build_request('setup_user'),
9                                     params={'username': str(username)})
10         except requests.exceptions.ConnectionError:
11             print 'Unable to contact HTTP server! Is the Daemon running?'
12             sys.exit(2)
13         [...]
```

Source Code 3.11: Setting up a user client-side

```
root@4f765c6859bd:/usr/src/apmt# scalectl setup Walki
No active user detected! Setting up Walki
User setup successful! Detected 0 running scalectl auto scaling cluster(s)
Found scalectl-cluster launch configuration
```

Figure 3.8: *Scalectl Daemon*: Setting up a user

The user can now choose to run a *Diagnostics Cluster* using the following command syntax:

```
1 scalectl cluster run STARTIME TIMEEND TIMESTEP TARGET FUNCTION SIZE VMType
```

```
root@4f765c6859bd:/usr/src/apmt# scalectl cluster run 1537699862 2147483647 60 "awsloadbal-694392381.eu-central-1.elb.amazonaws.com"
"x/8*(1-x/8)*c(3.14*x/8)*20*s(3.14*x/8)*20+100" 2 t2.micro
Setting up cluster
Timestart = 1537699862; Timeend = 2147483647; Timestep = 60; Target = awsloadbal-694392381.eu-central-1.elb.amazonaws.com; Function
= x/8*(1-x/8)*c(3.14*x/8)*20*s(3.14*x/8)*20+100; Size = 2; Instancetype: t2.micro
Deleting old launch config
Creating new launch config
Starting new auto scaling cluster
Successfully started cluster
```

Figure 3.9: *Scalectl Daemon*: Example of the cluster run command

This will trigger the following steps on the daemon server:

- Build the user context for the active user information stored in the *Storage* object
- Create a new security group if it does not exist
- Delete the old auto scaling launch configuration if it exists
- Create a new auto scaling launch configuration, with the provided instance type (vmtype) and adapted userdata (VM start up commands, including environment variables of the docker container (see Section 3.5))
- Start a new auto scaling cluster of the provided cluster size

Depending on the start time passed to the cluster the instances will start to sample the provided function each time step and output the desired amount of requests. In the next section we will look into the containerized script that is responsible for this traffic generation.

3.5 Diagnostics Cluster

Note: The Dockerfile can be found on GitHub¹¹ and the image as an automated built repository on DockerHub¹².

As we have previously established the *Diagnostics Cluster* is an Auto Scaling Group launched by the *Scalectl Daemon*. To ensure maximum performance, and again simplicity, the Linux distribution used by the cluster is CoreOS¹³, a minimalistic distribution that is made specifically for cloud usage. It deploys quickly, does not create much idle system usage and has Docker¹⁴ installed by default, only requiring us to execute a command [Source Code 3.12] to start the process of benchmarking.

¹¹<https://github.com/CM2Walki/BenchmarkContainer>

¹²<https://hub.docker.com/r/walki/benchmarkcontainer/>

¹³<https://coreos.com/>

¹⁴<https://www.docker.com/>

3 APMT Module Implementation

```
1 #!/bin/bash
2 docker run -d -e TIMESTART=1537699862 -e TIMEEND=2147483647 -e TIMESTEP=60 \
3 -e TARGET=awsloadbal-1159594470.eu-central-1.elb.amazonaws.com \
4 -e FUNCTION="(-1)*(x-10)^2+100" --name=benchmark \
5 walki/benchmarkcontainer
6 sudo systemctl stop update-engine # Stop auto updates
```

Source Code 3.12: UserData (start up script) generated by *Scalectl Daemon*

The Docker image is based on the `alpine:latest` image, making it very space efficient, however this also means that we have to use regular Bourne shell (`sh`) instead of `bash`. The script in [Source Code 3], was created as a proof of concept and should be replaced in the future, since it uses `curl` to spawn requests, which in our tests has shown that it can not handle a large amount of burst-like requests. Overtime the performance of the script can even decrease, as shown in Figure 3.5. The data in that figure shows a test conducted with 14 machines using the function “ $x*10000$ ”, whilst the test did reach 140000 requests per minute, it spawned inconsistencies over time, hinting at the fact that the $x*10000$ curl requests could not be completed in time. Contrary to the expectations the total requests per minute actually started dropping after the 140000 peak. The target machine, which was running a static nginx web page only reached 30% CPU usage at most during the test, not even triggering the Auto Scaling Group to provision more machines.

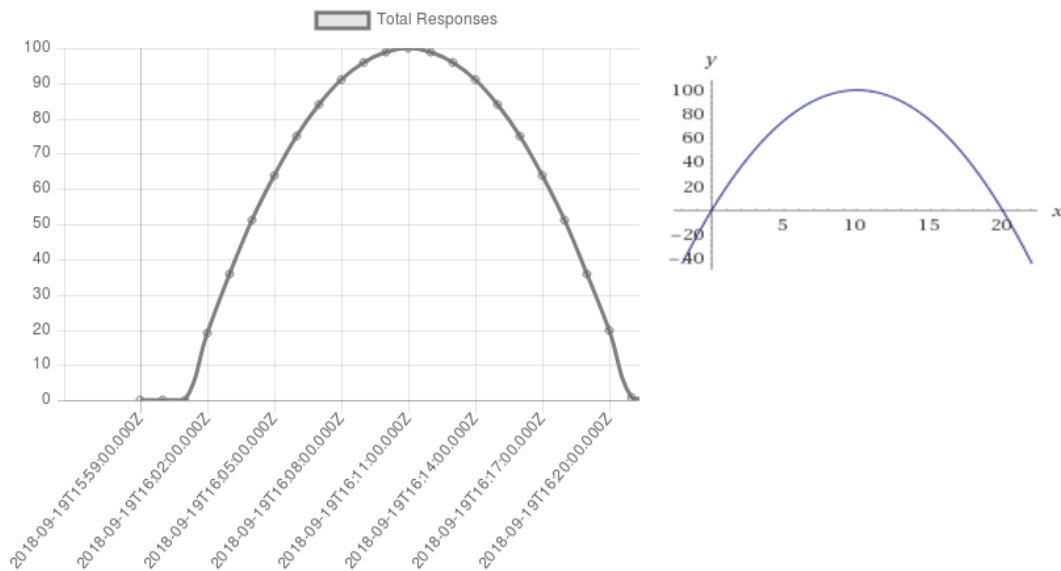


Figure 3.10: *Diagnostics Cluster*:
Resulting traffic (left)
Plot of function (right)

However the script [Source Code 3] performs up to expectations when used to replicate small amounts of requests as seen in Figure 3.10. The request count can also be influenced by adding additional machines to the *Diagnostics Cluster*, which act as a multiplier upon the total count. This, however, requires extra coordination between the machines to all act upon the same time. The variable "Timestart" was added exactly for that case. As seen in the script, when provided with a timestart located in the future, the process will wait until that time has been reached before starting to create traffic.

```
1  #!/bin/sh
2  mathfunction=$1
3  timestart=$2
4  timeend=$3
5  timestep=$4
6  target=$5
7
8  # Wait for initial time
9  while true; do
10     curtime=$(date +%s)
11     if [[ $curtime -ge $timestart ]]; then
12         break
13     fi
14     sleep 1s
15 done
16
17 COUNTER=1
18 curmathfunction=0
19 currentload=0
20 # Start hammering away
21 while true; do
22     curtime=$(date +%s)
23     if [[ $curtime -ge $timeend ]]; then
24         break
25     fi
26     curmathfunction=$(echo $mathfunction | sed 's/x/'$COUNTER'/g')
27     currentload=$(echo $curmathfunction | bc -l)
28     COUNT=0
29     while true; do
30         curl -s "$target" > /dev/null &
31         COUNT=$((COUNT+1))
32         if [[ $COUNT -ge $currentload ]]; then
33             break
34         fi
35     done
36     COUNTER=$((COUNTER+1))
37     sleep $timestep
38 done
```

Source Code 3.13: Benchmarking sh script found inside the container

Upon entering the main loop, the script now determines whenever it has already reached the "Timeend" variable. If that is not case, it retrieves the provided maths function and replaces all occurrences of x with the current sampling point (COUNTER). Then the current load is calculated using the *bc*¹⁵ command. The next while loop then spawns curl jobs aimed at the "Target" depending on the previous calculation. This process gets repeated each "Timestep"-seconds up until the "Timeend" is reached or the VM instance is terminated.

¹⁵<https://www.gnu.org/software/bc/manual/html.chapter/bc.5.html>

4 Future Research

During the making of this thesis, the suggested metrics have shown promising results when it comes to tracking the status of an Auto Scaling Group. With the help of the *Diagnostics Cluster's* capability to replicate traffic patterns based on a passed function it is possible to theorize an algorithm that tweaks CPU scaling thresholds that satisfy either *user-oriented* or *operation-oriented* metrics.

Since the *Diagnostics Cluster* accepts any function, we can replicate past traffic patterns from e.g. a sale or press release, by interpolating a function from that period's data set. This way we are able to reproduce the same environment over and over again, allowing for tweaking to take place. We can also anticipate higher event traffic by manually modifying the traffic pattern function.

In the end, this can potentially lead to budget savings, because we might have fewer over-provisioning phases. On the other hand, there is also the possibility to improve responsiveness, as we no longer only take the current moment's "snapshot" as our only reference, since we have previous similar data available. This can help us anticipate scaling up in advance but also help us by scaling down faster.

4.1 Threshold Tuning Algorithm

In order to determine the most ideal value for the CPU threshold, we assume a similar stance as the two congestion control phases of TCP: *Slow-Start* and *Congestion-Avoidance* [1, Figure 1]. Whilst TCP regards package loss as an indication for imminent congestion, we will consider an increase in errorrate over a certain threshold ($ErrorThreshold$, Algorithm 1] as our sign of congestion.

```

input : CongestionStepSize, ExposedPorts, ExperimentTimeInMinutes,
          ErrorThreshold,  $U > 0$ 
output :  $U_{Optimal}$ 
 $inSlowStartPhase \leftarrow true$ ;
1 while true do
     $Exp \leftarrow CreateExperimentWithThreshold(U)$ ;
     $wait()$ ; // Wait for experiment to launch
     $DiagnosticsClusterGenerateTraffic(ExposedPorts)$ ;
     $E_{total} \leftarrow 0$ ;
    for  $t \leftarrow 1$  to  $ExperimentTimeInMinutes$  do
         $E_{total} \leftarrow E_{total} + GetCurrentErrorRate()$ ;
         $wait(1)$ ; // Wait one minute
    end
    if  $E_{total} \leq ErrorThreshold$  then
        if  $inSlowStartPhase$  then
             $U \leftarrow U * 2$ ;
        else
             $U \leftarrow U + CongestionStepSize$ ;
        end
    else
        if  $inSlowStartPhase$  then
             $U \leftarrow U/2$ ;
             $isSlowStartPhase \leftarrow false$ ;
        else
             $U \leftarrow U - CongestionStepSize$ ;
             $return U$ ;
        end
    end
     $KillExperiment(Exp)$ ;
end

```

Algorithm 1: User-Oriented Threshold Tuning Algorithm

The algorithm above showcases the procedure to find the optimal CPU threshold for a user-oriented use case. We start off with a user-provided CPU threshold value, optimally the one that is used in production for that specific application. Next, we initialise an application experiment cluster of a predefined-size that scales if the average

CPU usage rises over the provided threshold. We now let our *Diagnostics Cluster* create traffic at the exposed ports of the application, after which we start collecting the error rate of the application each minute up until the provided *ExperimentTimeInMinutes*. When the experiment is concluded, we determine if we got over the *ErrorThreshold* if that not the case, we double the CPU utilization and run a new experiment. We keep doubling the CPU utilisation of the experiments, until we slip over the *ErrorThreshold* in which case we half the utilization again. This correlates with the *Slow-Start* of the TCP protocol congestion avoidance procedure.

Now we start linearly increasing the CPU utilization by the *CongestionStepSize*. Until the *ErrorThreshold* is violated again, in which case we decrement by *CongestionStepSize* and output the *optimal* CPU scaling utilization threshold.

4.2 Diagnostic Cluster Upgrades

The script responsible for the *Diagnostics Cluster's* ability to replicate traffic, has a lot of potential for additions and improvements:

- Support for multiple variables, so more complex functions are possible (cubic splines).
- Support for targeting multiple routes on an endpoint.
- A programming language that can use a system's bandwidth more efficiently has to be found.
- The traffic pattern generation should be spread out across threads and across time.
- The successful and error response metric should be read from the *Diagnostic Cluster*, this way the *user-oriented* metrics can be tested independently, even on non-cloud clusters.

In total, it has to be able to generate traffic more efficiently to force Auto Scaling Groups to provision more machines, a case which rarely occurred using the current Bourne script implementation.

4.3 Combination

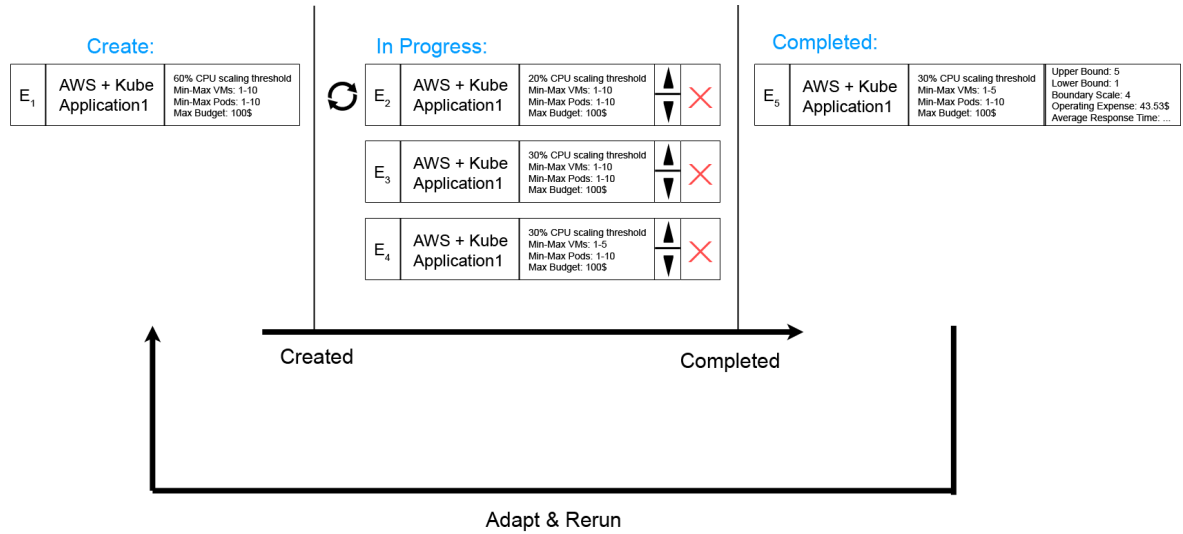


Figure 4.1: User Interface Proposal: Automatic Metric Tuning Dashboard

With the help of the *Threshold Tuning Algorithm* [Algorithm 1] and a more efficient *Diagnostic Cluster*, we can extend the APMT even further with an *Automatic Metric Tuning Dashboard* [Figure 4.1], that allows a user to run experiments/tests that converge towards certain Quality of Service requirements automatically.

Depending on the application, operators might favor responsiveness over costs or vice-versa.

5 Conclusion

In the scope of this thesis, the Autoscaling Performance Measurement Tool (APMT) was successfully extended with six additional experimental metrics, classified into two groups: *Operation-oriented* & *User-oriented*.

These metrics, combined with the addition of a *Scalectl Daemon* that records external data, now allow valuable insights into the scaling patterns of AWS Auto Scaling Groups. These patterns can also be replicated by the attached Daemon which is able to spawn *Diagnostic Clusters* that generate request patterns matching the provided function.

These additions combined with the proposed *Threshold Tuning Algorithm* have a great potential to improve long-term auto scaler performance, regardless of the Public Cloud Provider that offers it.

Appendix

Continuous Deployment during development

To ensure rapid development of both the *Scalectl Daemon* and the extended APMT, both GitHub repositories were outfitted with a “remotedeploy.sh”. These can be executed inside the IDE of the developer to automatically redeploy an application with the latest changes applied to it. In addition, the monolithic APMT image as well as the benchmarking image are both built automatically on changes to their respective git repositories.

```
1  #!/bin/bash
2  # Add the Scalectl daemon to an existing container
3  # Creates a ScaleX container if it doesn't exist already
4  if [ -z $1 ] || [ -z $2 ] || [ -z $3 ]; then
5      printf 'Not enough arguments provided!\n\n'
6      printf 'Usage: ./remotedeploy.sh REMOTE SCALEXPORT SCALEXCONTAINERNAME\n\n'
7      printf 'Example: ./remotedeploy.sh root@example.org 8080 ScaleX\n'
8      exit 1;
9  fi
10 REMOTE=$1
11 SCALEXPORT=$2
12 SCALEXCONTAINERNAME=$3
13 ssh $REMOTE << EOF
14     if ! docker start '$SCALEXCONTAINERNAME'; \
15         then docker run -d -p '$SCALEXPORT':8080 \
16             --name='$SCALEXCONTAINERNAME' walki/apmt; fi
17     docker exec '$SCALEXCONTAINERNAME' bash -c 'if cd /usr/src/ScaleX-tune/.git; \
18         then cd /usr/src/ScaleX-tune/ && git fetch --all && \
19             git reset --hard origin/master; \
20     else cd /usr/src/ && git clone https://github.com/CM2Walki/ScaleX-tune; fi'
21     docker exec '$SCALEXCONTAINERNAME' bash -c 'cd /usr/src/ScaleX-tune && \
22         make init clean-build build'
23     docker exec '$SCALEXCONTAINERNAME' bash -c 'scalectl stop'
24     docker exec '$SCALEXCONTAINERNAME' bash -c 'rm /tmp/scalectl-daemon.pid > \
25         /dev/null 2>&1'
26     docker exec '$SCALEXCONTAINERNAME' bash -c 'scalectl start'
27     printf 'Deployment script done.\n'
28 EOF
```

Source Code 1: Add the Scalectl Daemon to an existing container (or update it)

```

1 init:
2     pip install -r requirements.txt
3
4 update:
5     git fetch --all
6     git reset --hard origin/master
7
8 clean-build:
9     rm -rf tunex/build/
10    rm -rf tunex/dist/
11    rm -f /var/log/scalectl.log
12    find tunex/ -name '*.pyc' -exec rm --force {} +
13    find tunex/ -name '*.pyo' -exec rm --force {} +
14
15 build:
16     cd /usr/src/ScaleX-tune/tunex/ && pyinstaller --onefile scalectl.py
17     mv /usr/src/ScaleX-tune/tunex/dist/scalectl /usr/bin/scalectl

```

Source Code 2: Makefile that can rebuild the *Scalectl Daemon* using *pyinstaller*

```

1 #!/bin/bash
2 # Add the ScaleX module to an existing container
3 # Creates a ScaleX container if it doesn't exist already
4 if [ -z $1 ] || [ -z $2 ] || [ -z $3 ]; then
5     printf 'Not enough arguments provided!\n\n'
6     printf 'Usage: ./remotedeploy.sh REMOTE SCALEXPORT SCALEXCONTAINERNAME\n\n'
7     printf 'Example: ./remotedeploy.sh root@example.org 8080 ScaleX\n'
8     exit 1;
9 fi
10 REMOTE=$1
11 SCALEXPORT=$2
12 SCALEXCONTAINERNAME=$3
13 ssh $REMOTE << EOF
14     if ! docker start '$SCALEXCONTAINERNAME'; \
15     then docker run -d -p '$SCALEXPORT':8080 \
16         --name='$SCALEXCONTAINERNAME' walki/apmt; fi
17     docker exec '$SCALEXCONTAINERNAME' bash -c 'if cd /usr/src/apmt/.git; \
18     then cd /usr/src/apmt/ && git fetch --all && git reset --hard origin/master && \
19     cp -a /usr/src/apmt/server/. /usr/src/apmt/; else cd /usr/src/ && \
20     git clone https://github.com/CM2Walki/ScaleX && mkdir -p /data/db && \
21     mkdir -p /usr/src/apmt && cp -a /usr/src/apmt/server/. /usr/src/apmt/ && \
22     cd /usr/src/apmt && npm install; fi'
23     docker restart '$SCALEXCONTAINERNAME'
24     printf 'Deployment script done.\n'
25 EOF

```

Source Code 3: Deploy changes to APMT to remote

Bibliography

- [1] G. A. Abed, M. Ismail, and K. Jumari. A survey on performance of congestion control mechanisms for standard tcp versions. *Australian Journal of Basic and Applied Sciences*, 5(12):1345–1352, 2011. Cited By :5.
- [2] E. Casalicchio and V. Perciballi. Auto-scaling of containers: The impact of relative and absolute metrics. In *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*, pages 207–214, 2017.
- [3] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, André Bauer, Alessandro V. Papadopoulos, Dick Epema, and Alexandru Iosup. An experimental performance evaluation of autoscalers for complex workflows. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2):8:1–8:32, April 2018.
- [4] A. Jindal, V. Podolskiy, and M. Gerndt. Multilayered cloud applications autoscaling performance estimation. In *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, pages 24–31, Nov 2017.
- [5] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Autoscaling performance measurement tool. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18*, pages 91–92, New York, NY, USA, 2018. ACM.
- [6] J. Namjoshi, Y. Wadia, and R. Gaonkar. Portable autoscaler for managing multi-cloud elasticity. In *Proceedings - 2013 International Conference on Cloud and Ubiquitous Computing and Emerging Technologies, CUBE 2013*, pages 48–51, 2013.
- [7] S. Zhang, J. Lan, P. Sun, and Y. Jiang. Online load balancing for distributed control plane in software-defined data center network. *IEEE Access*, 6:18184–18191, 2018.