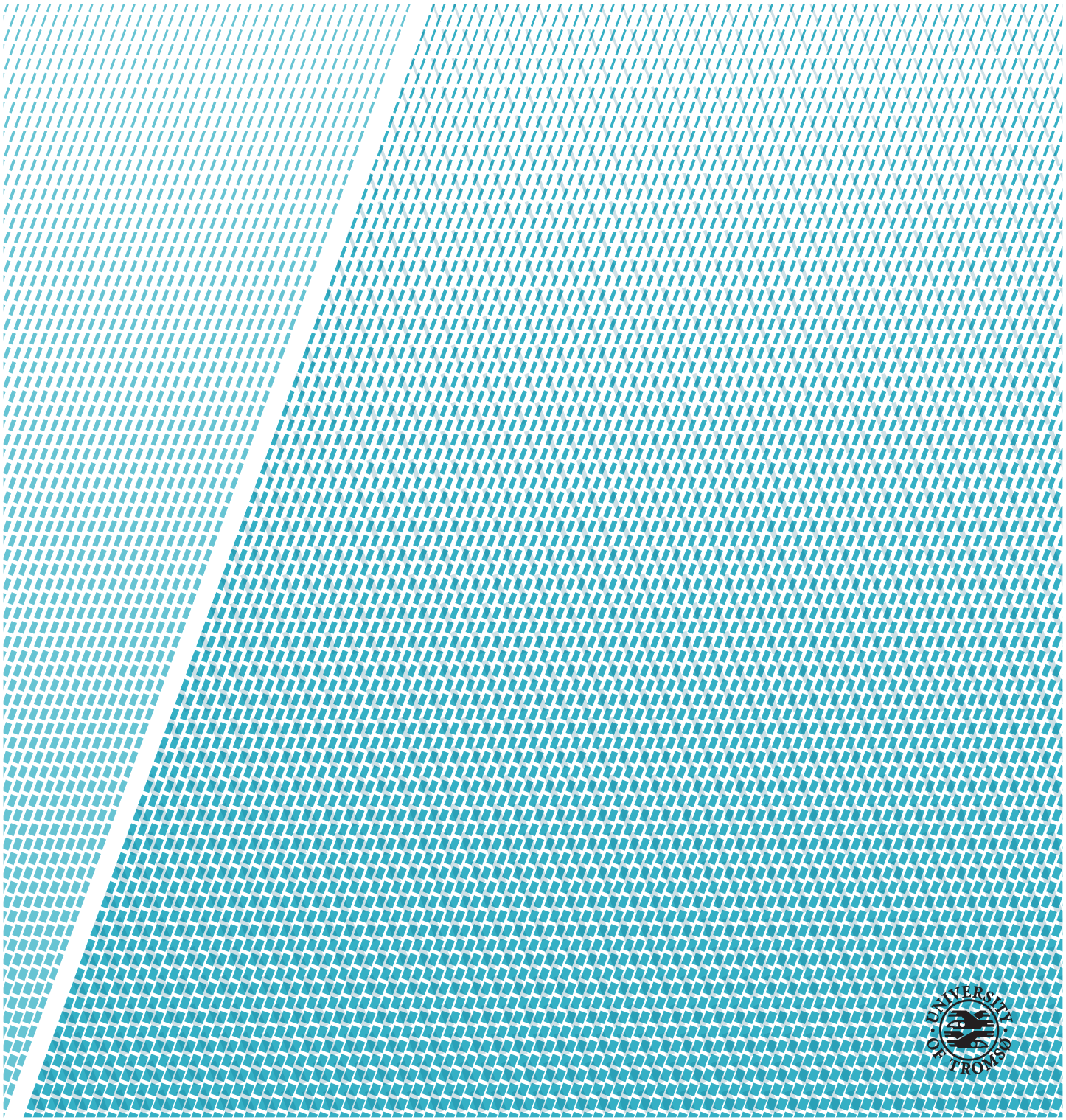
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Auto scaling framework, simulator, and algorithms for the META-pipe backend

Tim Alexander Teige

Master's thesis in Computer Science [INF-3981] June 2018



Abstract

META-pipe is a metagenomic analysis service provided in the ELIXIR distributed life science infrastructure. It provides assembly of sequence data, functional annotation, and taxonomic profiling. The analysis is computationally intensive and it consist of many jobs which have different requirements and varying complexity and execution times It therefore requires an execution environment that can provide a large set of nodes, and elasticity to scale up and down the resources depending on the current resource needs.

We propose an auto scaling framework that automatically scale clusters and schedule jobs on different execution environments. It adds auto scaling to the META-pipe architecture, and provides a simulator that enables the development and comparison of auto scaling algorithms. No earlier solutions provide the needed ability schedule jobs across multiple execution environments and scale their resources. The earlier solutions only provide scaling for a single cloud or cluster.

We designed our framework to support applications that submit jobs for processing, and to support any type of execution environment. The framework consist of of three key components, an estimator, an auto scaling algorithm and the cloud components, that interact with an auto scaling runtime and a simulator through defined interfaces. The framework relies on an external job manager to schedule jobs on the correct execution environments.By implementing these three components based on the requirements of the application, the users can both visualize the changes made by the algorithm and deploy the auto scaling algorithm to a backend system.

To evaluate the simulator we implemented an estimator, three algorithms and a simulated cloud component. The results show that the simulator can accurately simulated different algorithms and simulate the external scheduler. The results from the algorithms show that integrating and deploying an auto scaling algorithm would prove beneficial for the META-pipe application by removing the need for manual execution environment selection and reducing the total duration of the job queues.

Acknowledgements

I would like to thank the developers of META-pipe for contributing and providing insight to the META-pipe infrastructure and application. I am thankful for the time I was able to be a part of the development team, this period gave a renewed motivation for continuing the work.

I would also like to thank the BDPS team for listening to presentations of the auto scaling framework, providing suggestions on how to improve it, and expose weaknesses and errors.

Lastly I would like to thank Lars Ailo Bongo for contuously following up on the progress of the thesis and making suggestions on how to improve it. The advices I have recieved have been invaluable. I am greatful for the opportunities you have provided me with.

Contents

List of Figures

1. https://www.openstack.org/software/images/diagram/overview-diagram.svg. Accessed 31 May. 2018

List of Tables

1 Introduction

META-pipe is a metagenomic analysis service developed by Center for Bioinformatics at the Arctic University of Norway in the european wide project ELIXIR[1]. Currently, the service provides functional annotation, taxonomic classification and assembly of metagenomes. These analyses are both I/O intensive and computationally intensive. This has created the need to reduce costs and the response time of the analysis. The META-pipe analysis can currently be run on the cPouta OpenStack cloud[2], the CESNET-MetaCloud OpenNebula cloud, Amazon Web Services (AWS) Elastic MapReduce and on the Stallo supercomputer. A method for reducing cost and response time, is introducing automatic scaling, and work propagation to the most compute and cost efficient platform available. Cloud platforms such as AWS provides a method for scaling the number of compute nodes dynamically within a compute cluster, but with the current META-pipe implementation, the decision of how much resources to allocate for the job and where to run a specific META-pipe job has to be manually done as a job is submitted, either by the user or by an administrator. Inefficient job placement wastes computational power, increases job completion wait time, and increases costs. Automating this process by auto scaling both the execution environments and the META-pipe job queue would remove the need for manual allocation of compute resources, and the manual selection of execution environment. Auto scaling the META-pipe job queue is not an easy task. To scale the META-pipe application to multiple environments the cost

1. ELIXIR Europe." `https://www.elixir-europe.org/`. Accessed 12 Sep. 2017
2. CSC - cPouta. `https://research.csc.fi/cpouta`. Accessed 12 Sep. 2017

of running a job and the job execution time must be estimated to select the environment which will result in the lowest cost, the shortest execution time and, the shortest wait time. The META-pipe jobs are complex and have a large list of parameters which all can affect the execution time and which execution environment it should run on.

No existing system have been found that is able to scale across multiple execution environments and satisfy the requirements of the META-pipe application. The existing auto scaling systems rely on scaling for a single platform or within a single cluster and can therefore not be used by META-pipe.

Our solution was to design a framework, simulator and an auto scaling runtime which could be integrated in the META-pipe backend system and provide auto scaling accross different execution environments. The framework consist of multiple components that interact with an auto scaling runtime and a simulator through defined interfaces. There are three key components, an estimator, an auto scaling algorithm and the cloud components. We found that the simulator is able to simulate multiple different algorithms and provide a graphical user interface to both debug and analyze the algorithm output. We also found that the META-pipe application would benefit from integrating the auto scaling framework, in terms of reducing the total queue durations and job wait time. We make the following contributions:

- We describe the design of a framework which enable its users to implement and deploy auto scaling algorithms for cross execution environment scaling.

- We provide use cases for how user can use the framework to develop new algorithms, test the algorithms using the simulator and deploy them using the runtime.

- We discuss the results of both the algorithm and the simulator.

The thesis is separated in to 7 chapters. We first provide a background on META-pipe to give context for the META-pipe backend and the META-pipe jobs. Secondly we present the design of the auto scaling framework. In the design chapter we describe the architecture of the framework and its components. We then describe both the simulator and the auto scaling runtime. In chapter 4 we show how the framework can be used to develop and deploy new algorithms, how the user can add new compute resource providers, and how to add a new application using the existing simulator and runtime. In chapter 5 we present the results for the execution time estimator and the simulator and give an explenation as to why the runtime has not been evaluated. In the next chapter we discuss the related work and why these can not be used by META-pipe.

Finally a conclusion is drawn and we make some suggestions on what can and should be done in the future.

2

Background: META-pipe service architecture

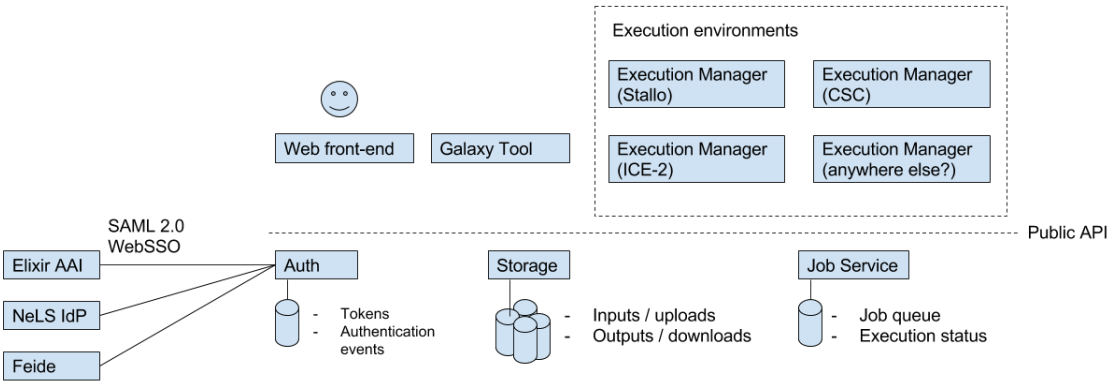


Figure 2.1: Metapipe architecture

The auto scaling algorithm and simulator presented in this paper is designed to be integrated with the META-pipe architecture (figure 2.1[1]), it can also be started independently of META-pipe as long as certain criterias are met. The META-pipe architecture consists of multiple separated services which run

1. `https://docs.google.com/document/d/16iwUrfh6_eK-_T158_zgsRuItJ-qw1YsCiF1gnNfzLY/edit` Accessed 14 Dec. 2017

independent of each other. The architecture is designed with *Separation of Concerns*[2] as a key concept, where each service has a single responsibility. To provide the background necessary to understand the constraints and integration challenges for the framework, we describe the services the auto scaling algorithm and the simulator depends on.

Job Manager

The main focus of the description is the job manager. This is due to its importance in integrating the auto scaling algorithm and framework. The auto scaling algorithm uses the data provided by the job manager to make predictions for the execution time, and outputs directly to it with an updated queue structure. The job manager is responsible for validating submitted jobs and enqueuing the jobs on the respectable execution environment target. The job manager handles jobs launched on different execution environments. The main environments is Stallo and cPouta. META-pipe has also been deployed on AWS.

Job State

Each job within the META-pipe infrastructure contains an overall state. The job states dictates which interactions and further transitions the job can make. The job state is maintained and stored by the job manager in the database. The state is used by the web application to filter out jobs that are no longer relevant. Internally the state of the attempts are used for calculating the next step in the scheduling and execution.

Attempts

A job contains attempts which are queued on the spark executors. These attempts hold an internal state dictates the job state. A healthy attempt has either queued locally, queued executor, running or completed state.

1. Queued locally: The attempt is queued on the job manager and is stored in the job manager database. The job manager will schedule the attempt on an executor when it becomes available.

2. Chapter 2: Key Principles of Software Architecture. `https://msdn.microsoft.com/en-us/library/ee658124.aspx`. Accessed 12 Sep. 2017.

2. Queued executor: The attempt has been queued on the executor manager and is scheduled to be submitted to a spark driver on the targeted environment. The spark executor manager contains an internal scheduling algorithm which dictates how the queued attempts are executed.

3. Running: The attempt is properly executing the job on the targeted execution environment.

4. Completed: The job and attempt has been successfully completed.

During the execution of an attempt it can fail. These failures have been structured in to different types.

1. Executor failed: The executor manager could not be schedule for the attempt at the given execution environment target.

2. Workflow failed: The attempt encountered breaking exceptions during the execution of the analysis and could not continue. This requires the intervention of an administrator in order to both debug the META-pipe pipeline and restart the attempt.

3. Timed out: The heartbeat could not reach the attempt. During this state the job manager creates a new attempt for the job and this is submitted again.

4. Cancelled: The cancelled state is only set if an admin cancelled either the attempt or the entire job.

An attempt is periodically checkpointed during execution. If the attempt transitions into an erroneous state and is restarted it will continue from the checkpoint. This reduces the computation loss when an error occurs.

Execution Manager, Spark Driver and Executors

The spark driver is a spark process [3] which manages the execution of jobs on a cluster. The driver is responsible for launching executors and managing the execution order and flow of the job. Each executor is independent from each other and is an isolated process. The execution manager is a process manages the submission of jobs. This process can be launched in two different modes depending on the execution target.

3. Cluster Mode Overview - Spark 2.2.0 Documentation - Apache Spark. `http://spark.apache.org/docs/latest/cluster-overview.html`. Accessed 24 Oct. 2017.

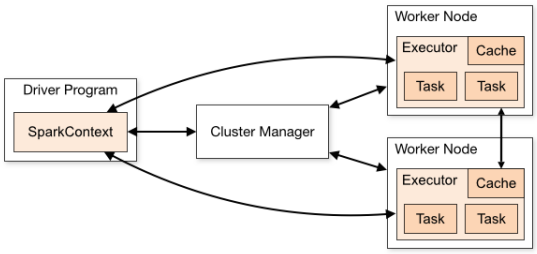


Figure 2.2: Spark process

1. The execution manager is launched on the cluster frontend and fetches jobs from the job manager queue. After a job is fetched the execution manager runs spark submit with a new spark driver for each job. The driver can be run on two different locations.

   (a) The driver is on the same node as the execution manager

   (b) The driver is launched in cluster mode. This runs the driver on a random worker node.

2. The execution manager is launched in "integrated" mode where it runs as a part of the spark driver. When launched in this mode, it is a thread that fetches jobs from the job manager and submits it to the cluster.

Which mode that is used is dependent on which cluster is the target environment. Stallo uses 1a, Amazon Web Services uses 1b, and mode 2 is used on cPouta (CSC). This is done due to the different cluster configurations and setups.

Authorization Service

The authorization service is designed as a separate service from both the job manager and the web application. The main purpose for this service is to separate users based on privileges in order to ensure correct access to the META-pipe tools, interfaces and services. For the authentication process it uses the ELIXIR-AAI as a federated id provider (IDP). The ELIXIR-AAI has federated multiple IDPs such as Feide and Google. The authorization service also provide a client and client secret authentication protocol defined in SAML 2.0. Services in the META-pipe backend use this protocol to authenticate internally.

Storage Service

The META-pipe storage service is a simple key-value store implemented by the META-pipe devlopers. The storage service provide endpoints for uploading data sets and downloading them. The storage server is used by both the META-pipe web application and the different analysis.

Execution environments

The execution environments are external compute resource provides that the META-pipe application can run on. Providers that are supported by META-pipe include cPouta OpenStack cloud, the CESNET-MetaCloud OpenNebula cloud, Amazon Web Services (AWS) Elastic MapReduce and on the Stallo supercomputer. Each execution environment has at least a single Spark driver hosted when the META-pipe application runs.

3

Design

In this chapter we describe the design of the auto scaling simulator and runtime. First, we give an overview of the auto scaling system. Then we describe the algorithm design and usage. The simulator, the auto scaling runtime and algorithms described in this section have been developed as a framework for testing, analysing and deploying the auto scaling algorithm. Parts of the framework are backend specific and have been developed for META-pipe.

Architecture

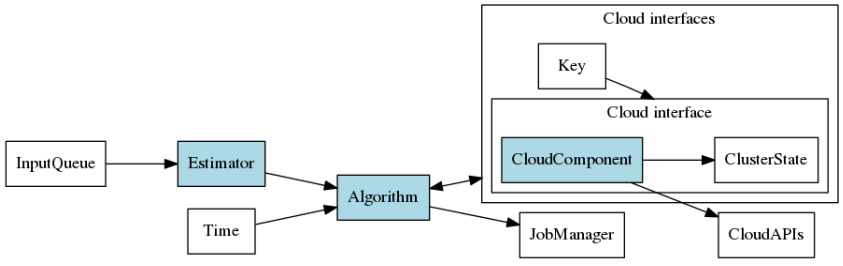


Figure 3.1: Auto scaling architecture for the auto scaling runtime and simulator

Figure 3.1 has an overview of the architecture for both the simulator and the auto scaling runtime. The blue components are provided as dependencies to both the auto scaling runtime and simulator and can be replaced by implementing

their interfaces and providing them as dependencies. This makes it easy to replace or extend these components.

Job Manager

The job manager is an external service for scheduling the execution of queued jobs on different execution environments. The job manager must be able to schedule jobs based on priority to the specified execution environment. If this requirement is not met, external resources are added, but the jobs in the queue will not be correctly scheduled.

Input queue and Estimator

The input queue is a set of external jobs that have the following values:

- Id - Created by the job manager
- Tag - Specifies where to run the job. Can be empty
- Parameters - Map of parameter name to values
- State - Defines if the job is queued, running, cancelled or delayed
- Priority - Defines the external scheduling priority on the job manager
- Execution time - Estimated execution time for each execution environment.
- Deadline - The targeted wanted finish completion time
- Created - Timestamp of when the job was submitted
- Started - Timestamp of when the job was started. , Only set if state is running
- Instance flavour - The cloud instance type for the job

These jobs are passed to the estimator that predicts the execution time of a job in the queue based on the job parameters and the input dataset size. The estimator is defined as an interface. The implementation of the estimator defines how to handle the input jobs. The estimator outputs a list of jobs that have their execution time estimated. The estimated jobs in the queue are

passed to the algorithm with a timestamp that defines the starting time of the algorithm, and a map containing the interface instances for the different cloud platforms. The estimated execution times are used by the autoscaling algorithm to decide when to add or remove compute resources, and to calculate the cost of executing the jobs.

Cloud Interface Map

The cloud interface map is a mapping between a key and the respective cloud interface instance. The key is the job tag associated with the respective cloud, for META-pipe these are

- aws - AWS
- csc - cPouta
- metapipe - Stallo

The cloud interface defines methods for interacting with cloud APIs, calculating job cost, queue cost and queue duration.

The *cloud component* implements the interface for the cloud. Each component is implemented targeting a specific execution environment, such as AWS, cPouta or Stallo. The cloud API is usually provided through a SDK or by using http requests directly against the API. The cloud components should also write events to a database when adding or deleting resources. This should be done to both debug and analyse the algorithms impact on cost and total queue durations.

The *cluster state* is an in-memory cache for the current state of the cluster or cloud. The state is a list of the current allocated resources and their states. If an instance is available for processing its state is set to “INACTIVE”. If it is running and executing, its state is set to “ACTIVE”. The cloud components has a responsibility of setting the state based on the states retrieved from the respective cloud API.

Algorithm Responsibility

The algorithm is responsible for optimizing the input queue of jobs from the estimator and making a decision of when to add or delete instances. The tag specifies the execution environment for the job, if the tag is empty it can not be scheduled by the job manager. The job should be assigned a tag according

to the algorithm scheduling policy. How the policy is defined is based on user requirements. The policy can for instance be set to reduce the execution time, or to reduce cost. The algorithm sets the job priorities according to the policy. The algorithm uses the cloud interfaces to add or remove resources during execution based on the scaling policy defined by the user. An example of a scaling policy is that it should not exceed a certain limit of instances. The priority value is used by the job manager for priority scheduling. Each execution environment is handled independently by the job manager

Auto scaling Algorithm

The auto scaling algorithm interface is designed as a single queue optimization function. The algorithm is given a queue of estimated jobs. The scaling algorithm decides where to run the queued jobs, and adds or remove resources during its execution by using the cloud interface.

Example Algorithm

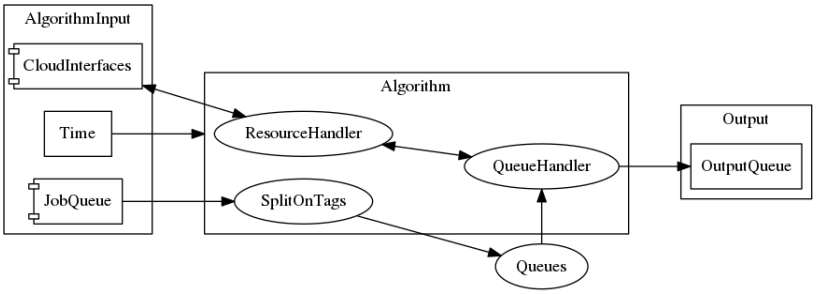


Figure 3.2: Example algorithm overview

An overview of an algorithm that scales jobs on AWS, an HPC system, and OpenStack for META-pipe jobs is in figure 3.2. The algorithm input is a map of instances implementing the cloud interface, the job queue and the timestamp for the start time of the algorithm. Each META-pipe job has a tag parameter which dictates which execution environment it should be executed on. Currently, these tags can be “aws”, “csc”, “metapipe” and an empty tag which has no specification for where to execute and is set by the algorithm to a specific execution environment. The input queue is split into separate queues based on the tag. The jobs in each queue is optimized by using any type of approach. A possible approach to optimize the queue is constraint programming [4]. A constraint programming approach yields an approximated optimal queue, based on the constraints given. For this scaling algorithm, the constraints are

the limit of allocated instances, job deadlines, priority and the cost. The limit of maximum allocated resources, the currently allocated resources and the cost for each job on a specific flavour are retrieved by the resource handler in the algorithm. Jobs that do not have tags must be assigned a tag, which can be done by running each permutation of the untagged jobs and the different queues and comparing both cost and execution time. The policy of this algorithm is: if the job has a high priority the tag which gives the shortest execution time is selected. Otherwise the tag which gives the cheapest execution is selected.

Example queue optimization

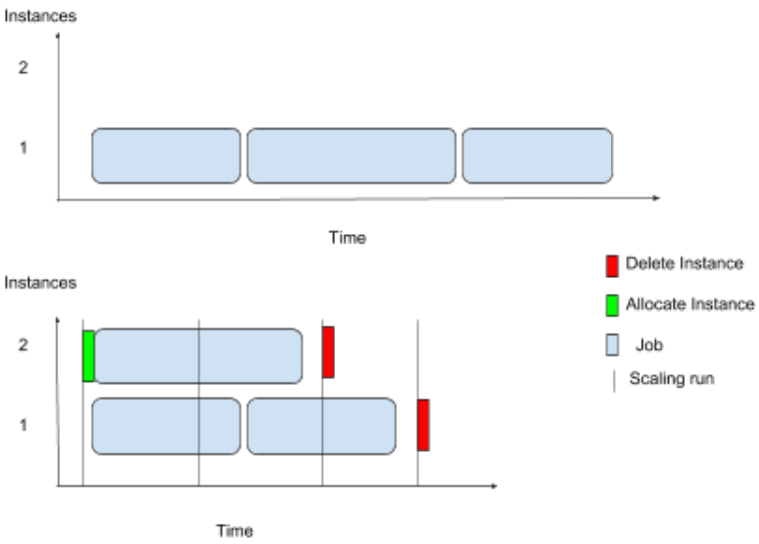


Figure 3.3: Example of a queue optimization and scaling process

An illustration of optimizing a queue is in figure 3.3. In the top half of the figure, three jobs are scheduled to run on a compute platform with only a single instance allocated. The bottom half of the figure shows an optimized queue and the allocate and delete instance events. Each line separates represents a call to the auto scaling algorithm. For each call the algorithm optimizes the queue. This results in a new instance being allocated by the algorithm in the first run and the optimized queue is sent to the META-pipe job manager in order to reschedule the jobs. This allows the longer job to run in parallel with the two shorter jobs. In the second scaling run, nothing needs to be changed. At the third scaling request the instance allocated in the first run is no longer required, and is deleted, there are no changes to the queue. In the final run there are no more jobs in the queue and the last instance can be deleted.

Clouds have instances in multiple flavors with different costs and performance characteristics. An algorithm may use different instances for cost-performance trade-off. The instance types are defined at launch time in a config file. These are loaded in the to the cloud components and can be retrieved through the cloud component interface. The META-pipe application has certain flavors that is optimal to run the application on. Changing instance flavours can therefore increase or reduce execution time and cost. The instance type is not included in the execution time estimation of a job since the information of which instance type the job used, is not available.

Three implemented algorithms

Three algorithms have been developed for testing and comparing the results of the auto scaling simulator. First, simple algorithm that takes a naive approach to optimizing the queue. Second, a worst-case algorithm that reduce the amount of instances to one, and third an algorithm that does nothing which is equivalent to no auto scaling.

Naive algorithm

The simple algorithm splits the queue into per-cloud queues based on the job tag. If there is a queue where the jobs have an empty tag it is iterated through and for each job, and a tag is assigned (Algorithm 1).

Algorithm 1 Scheduling policy

```
queueMap ← tagSplitQueue
for job in queueMap[Empty] do
    for cloud in Clouds do
        jobCost ← cloud.calculateJobCost(job)
        jobDuration ← cloud.calculateJobDuration(job)
        expectedCost ← jobCost + cloud.totalCost
        expectedDuration ← jobduration + cloud.totalDuration
        if cheapest > expectedCost then
            cheapest ← expectedCost
            cheapestCloud ← cloud
        end if
        if shortest > expectedDuration then
            shortest = expectedDuration
            shortestQueue = cloud
        end if
        if shortestCloud == cheapestCloud then
            job.Tag = shortestCloud
        else if job.Priority < THRESHOLD then
            job.Tag = shortest
        else
            job.Tag = cheapestCloud
        end if
    end for
end for
```

After the jobs with an empty tag has been assigned a tag the algorithm sorts the queue on priority and then on deadlines. The algorithm then checks if it should delete or add resources for execution environment (Algorithm 2).

Algorithm 2 Scaling policy

```
for jobinqueue do
    if job.state == RUNNING then continue
    end if
    if cloud.IdleInstances > 0 then cloud.ReuseInstance
    else if cloud.limit > instances.length then cloud.AddInstance
    else if instances.length > queue.length then cloud.DeleteInstance
    end if
end for
```

If a job has the “RUNNING” state, the algorithm skips to the next job, since it is assumed to have an instance which it is running on and can not be rescheduled.

This assumption can be made, since the META-pipe job manager continuously pings the jobs and they should instantly change state if an instance has crashed or been externally removed.

Worse-case Algorithm

The worst-case algorithm splits the queue based on the job tags into multiple queues. Jobs with an empty tag is processed with the same method as in the simple algorithm. Each queue is processed separately. The algorithms scaling policy removes instances until only a single instance for each execution environment is left. The algorithm does not remove instances that are actively processing a running job. The algorithm does nothing to the queue and returns the unchanged input queue.

Null Algorithm

The null algorithm only returns the same queue. It can be used while integrating with the META-pipe backend, while not using the auto scaling. This algorithm can also be used to compare against other algorithms to visualize the changes made. The null algorithm does not schedule unassigned jobs, so the comparison should only be done with queues that does not contain unassigned jobs.

Shared Database for the Simulator and the auto scaling Runtime

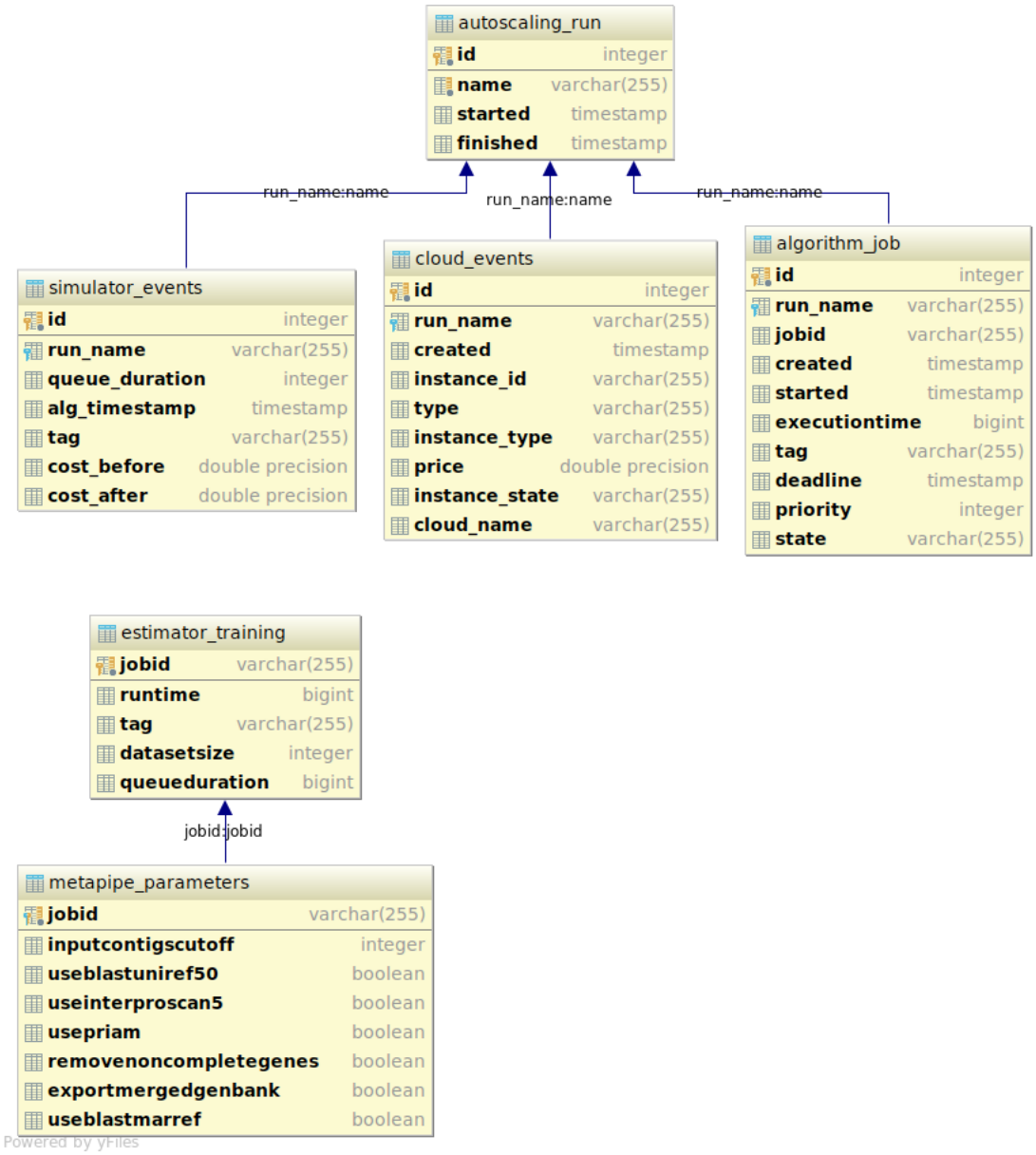


Figure 3.4: Entity Relation Diagram of the shared Database

The simulator and auto scaling runtime use a simple Postgresql database (figure 3.4) for storing the auto scaling runs and the estimator training data.

The *estimator training* table in the database contain the jobs used for training the estimator. Each row contain is a META-pipe job that have been parsed from its original JSON description to the values present in the table. The *parameters* are stored in a separate table with the job id as the foreign key to the estimator training table. This enables the future developers to easily add parameters for different type of jobs without modifying the estimator training table.

The *auto scaling run* table contain the initial start time and the finish time of a single run. The name is used as the foreign key for the simulator events, cloud events and the algorithm jobs that are stored during execution of an auto scaling run. The name must be set when creating a new auto scaling run. The simulator and the auto scaling runtime both use the database for creating runs. The *cloud events* table contain the events generated by the cloud components which implement the cloud interface. It is however up to the implementation of the cloud component to create these events and is not a requirement. The users of the simulator and the auto scaling runtime can use this table to analyse the algorithm.

The *algorithm job* table is used to store the state of the queue after the algorithm has finished executing. The jobs in the table can be used by both the simulator and the auto scaling runtime to ensure that the jobs have the expected tags and priorities.

The *simulator events* table is only used by the simulator to store total queue cost, total queue duration for each tag in the input queue. The simulator needs this because the cost calculations rely on the estimated job execution time. The events in this table is used by the simulator to create and visualize the cost and the duration of the different queues.

META-pipe execution time estimation

The auto scaling algorithms take as part of the input, a queue of jobs that have their execution times estimated and use these to calculate the cost in order to scale. The job execution times are estimated outside of the algorithm. The execution times for META-pipe jobs are measured in milliseconds. In the META-pipe job manager a job can be queued locally without being assigned an executor or queued on the executor. The time spent queued is not set in old META-pipe jobs, since it was added to the META-pipe jobs along with the time spent executing. The jobs that have the time spent executing value available use it. We calculate the execution time based on the initial creation time of the job and the last ping to the job for the jobs that do not have the time spent executing value available. For the jobs that have their values calculated

based on the creation time, can have long queue times which affects their calculated execution time. These jobs have therefore not been used in the estimator. Models were created for these jobs and are discussed in the result section.

Estimator Interface

The estimator interface is in listing 3.1.

Listing 3.1: Estimator Interface

```
type Estimator interface {
  Init() error
  ProcessQueue(jobs []AlgorithmJob)
        ([]AlgorithmJob, error)
}
```

The estimator is a dependency for both the runtime and the simulator. The init method initializes the estimator by training the models used to estimate the input jobs. The input defined in the interface is a list of jobs for the algorithm that have not been estimated. The estimator output is a list with the same jobs, with their time estimated. Converting jobs to the input format of the estimator is trivial. For META-pipe jobs a parser has been implemented.

Linear regression implementation

We assume there is a linear relation between the META-pipe job parameters, execution time and data size, and therefore linear regression[1] is used. Multiple models using linear regression were developed and implemented. Each model is explained in depth and the results are presented in the result section. The estimator used by the simulator for development and testing is split in to three different models, one for each execution environment.

META-pipe training dataset

To estimate execution time we need a representative dataset for training. The META-pipe job manager database has an entry for every job that has been run. The jobs are retrieved and every completed job is stored in the database. We use the execution time, parameters and the data size to train an estimator

1. https://en.wikipedia.org/wiki/Linear_regression Accessed 16 May. 2018

using linear regression. To use the parameters as a single number for the regression, they are OR'd together at different bit indexes for each parameter set, except for the input contig cutoff parameter which is used directly as a separate variable in the model.

Weaknesses of the linear model

A linear model is very restricted due to its inability to curve. This leads to outliers that increases the variance and reduce the fitting of the model. This could lead to heavily underestimated or overestimated execution times. The META-pipe analyses have three representative categories of data:

- Virus: kilobytes
- Genome: megabytes
- Meta-genome (Collection of genomes): gigabytes

Each category has varying sizes of input data. Viruses data size is in the kilobyte range, while genomes lies in the lower end of the megabyte scale and meta-genomes ranges from half a gigabyte to several gigabytes. It could be a possibility to separate out each category and create a model for each of them. This could potentially lead to a finer grained estimator. The input data to META-pipe jobs can vary in internal complexity. The complexity of the input data not been factored in.

Instance flavour type has also not been factored in for the estimator. The META-pipe application usually run on the same instance flavour for each compute cloud. This may change in the future and should be handled, however the instance information is not available in the job description and there is currently no possible way of retrieving it.

Cloud Interface

The cloud components each implement the cloud interface in listing 3.2

Listing 3.2: Cloud Interface

```
type Cloud interface {
  Authenticate() error

  SetScalingId(id string) error

  GetExpectedJobCost(job AlgorithmJob,
      instanceType string,
      currentTime time.Time) float64

  AddInstance(instance *Instance,
      currentTime time.Time)
      (string, error)

  DeleteInstance(id string, currentTime time.Time) error

  GetInstances() ([]Instance, error)

  GetInstanceTypes() (map[string]InstanceType, error)

  GetInstanceLimit() int

  GetTotalDuration(queue []AlgorithmJob,
      currentTime time.Time)
      (int64, error)

  GetTotalCost(queue []AlgorithmJob,
  currentTime time.Time) float64
}
```

The interface is designed to expose cloud APIs to the algorithm and the auto scaling runtime. For the auto scaling runtime each supported compute platform, including AWS, Stallo andcPouta, must be individually implemented because they require different SDKs or requests to interact with their respective APIs.

Cloud Components

A cloud component is the implementation of the cloud interface. The cloud components must implement all the methods described in the interface. A cloud component should interact with a database to keep track of adding or deleting instances. The “SetScalingId” method in the interface is used by both

the simulator and the auto scaling runtime to set the id for the current auto scaling run. The “Authenticate” method authenticates against targeted cloud and creates an open session between the component and the cloud. For META-pipe, three cloud components should be implemented to use the auto scaling algorithms.

Amazon Web Services

AWS provides auto scaling[3] for their Elastic Compute Cloud (EC2). This can scale capacity up or down based on predefined conditions. It is also possible to enable dynamic scaling which responds to resource usage and demand spikes. Using the auto scaling provided by AWS gives some benefits beyond the capacity change, including monitoring of instance health, automatic replacement of impaired and crashed instances, and capacity balance across regions. The auto scaling provided by AWS provides solutions for scaling a META-pipe job internally if it is run on AWS.

The cloud component for AWS is responsible for adding and deleting new instances on the EC2. AWS provide a SDK for interacting with their systems. To add or remove instances, a session to AWS must be initiated by using the Authenticate method in the cloud interface. The sessions should be active until, at least, the auto scaling algorithm finishes. The instances allocated on EC2 can simply be retrieved and their states can be inferred by checking for running jobs on the instances.

OpenStack

META-pipe is set up to run on an OpenStack cloud configuration. OpenStack is an open source software which cloud systems can use, and it provides APIs for accessing data, user identity, cloud management. Each underlying system of a cloud provider using OpenStack can differentiate. The system architecture is complex containing multiple systems.

- Compute - Resource pools, compute configurations, etc.
- Storage - Different types of storages
- Identity - Identity provider to get associated instances and user information
- Network - underlying network infrastructure of the cloud

- Image service - VM provider
- Cloud management - Intercloud management
- Control plane - Multipurpose controlling of the entire system, such as databases, API endpoints, scheduling policies and etc

The cloud component does not need to interact with all these systems to satisfy the cloud interface requirements. The cloud component has to interact with the identity system to authenticate and the control plane to add, remove and retrieve instances and their states.

A user of an OpenStack cloud has most of the entire system abstracted away, where the user in this case is the cloud component. The user only has to choose between compute resources and the storage solutions provided by the cloud environment.

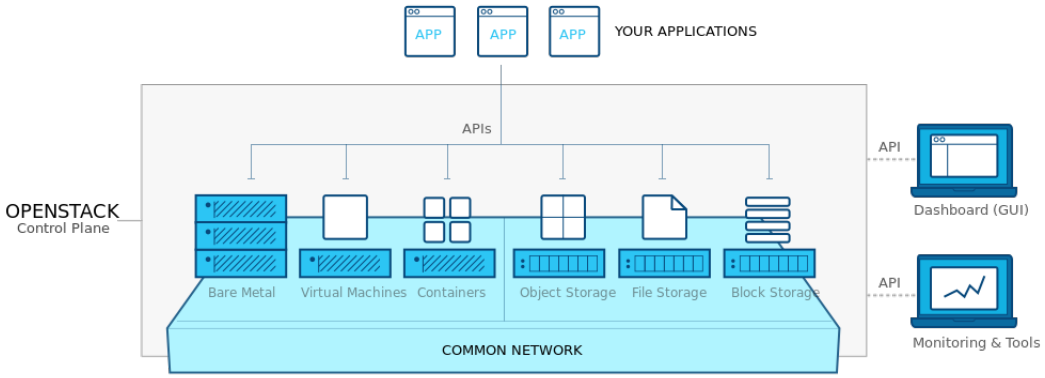


Figure 3.5: OpenStack user view[2]

The cloud component can use the APIs as seen in figure 3.5 to add and remove instances. Each instance can have different flavours which is a combination of an instance type and a storage solution. The instance states can be retrieved using the API for the monitoring tools.

Stallo

Stallo is a high performance computer, used mainly for research and is located at the Arctic University of Norway. Stallo uses SLURM to schedule jobs. The Stallo resource provision system requires users to wait in a queue for resources to be freed up and provided. META-pipe has a continuously running job that has seven compute instances that executes jobs, and an instance that host a

Spark driver. The Spark driver is used to orchestrate the jobs on Stallo. The cloud component for Stallo should interact with the Spark driver to retrieve the cluster states. It is still not clear how the cloud component should scale the cluster.

Simulated Cloud Component

The cloud component for the simulator can simulate all the different platforms, since it does not need to connect to other resources than a database. The simulated cloud component write cloud events to the database, when the "AddInstance" or "DeleteInstance" methods are used. For the simulated cloud component, the default initial states are loaded through a configuration file[3]. The configuration file is in JSON and its path is set as an environment variable on the system that runs the simulator. Authentication is not needed for the simulated cloud component since it does not interact with external resources except the database.

Simulator and auto scaling Runtime

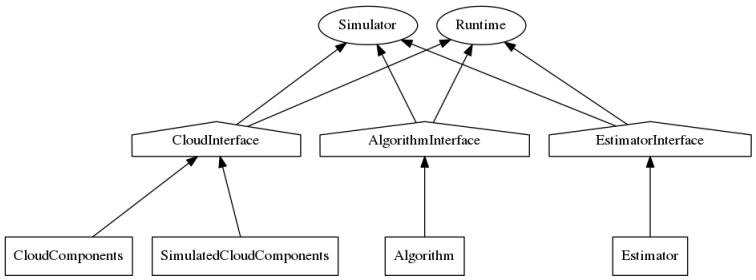


Figure 3.6: Simulator and Runtime dependencies as instances of interfaces

The simulator and the runtime are designed as two separate entities (figure 3.6). They run independent of each other. The estimator, cloud components and algorithm share the same interface across the simulator and runtime. The algorithm can therefore be easily moved from the simulator to the runtime in deployment. The simulator is designed as a testbed for the auto scaling algorithms. It can be used to both debug and compare the changes made by the algorithms. The runtime is designed as a service that is run in production side by side with a job manager that is responsible for scheduling the jobs which the algorithm outputs.

3. A config file is provided in the source repository

Simulator

The simulator is a http server which provides endpoints for running simulated job queues. Only META-pipe jobs are currently supported. It provides a graphical user interface for comparing previous simulation runs. The simulator has a Postgresql database which contain the simulations and their respective simulation events, as well as the jobs used for the estimator and the job parameters as defined in the database section. The estimator is provided as a dependency is initialized when the simulator is created. The simulator is used to analyze, evaluate, and compare auto scaling algorithms before they are deployed in production.

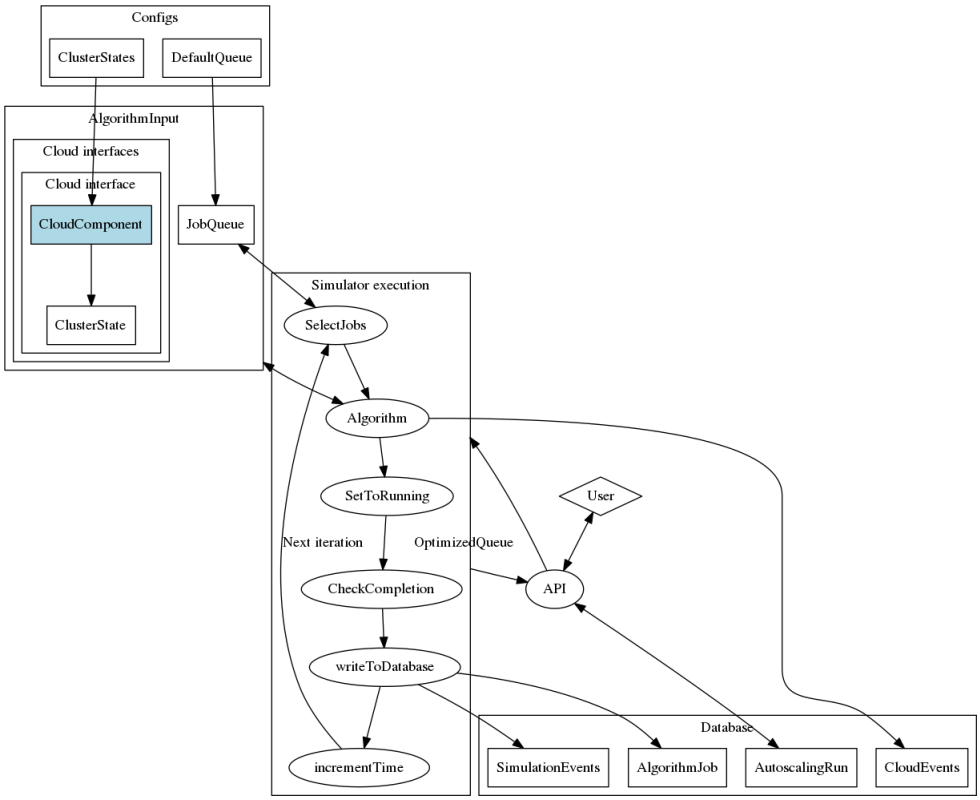


Figure 3.7: Simulator execution flow and components

Figure 3.7 is an overview of the execution of the algorithm using the default queue and cluster states. A user initiates a simulation with a request to the simulator API. The database and the algorithm have both been described in the previous sections.

Simulator API

The simulator provides an API which can be used with http client, such as cURL. The simulator GUI retrieve simulations from the server. The simulator defines three endpoints to interact with it (table 3.1).

Table 3.1: Simulator API

Action	Endpoint	Request type
Load GUI	/	GET
Launching a simulation	/metapipe/simulate/	POST
Retrieving all simulation	/simulation/all	GET
Retrieving a simulation	/simulation/?id=	GET

The index endpoint gets the html and javascript code for the GUI. It can be used by a web browser to view the GUI.

A simulation can be launched with a request to the endpoint with the parameters in table 3.2 in JSON provided in the request body. Note that all the parameters are optional. An empty body will use the default values for all parameters.

Table 3.2: Simulation input

Parameters	Description	Default value
timestamp	Unix timestamp defines the algorithm start time	23.05.2018 - 20:40:23
job_queue	META-pipe job queue	From config file
cluster_states	State of the clusters	From config file
name	Name of the simulation run. [Unique]	auto generated
timestep	The amount of time the algorithm should increase the timestamp for each iteration	30 minutes
iterations	The number of iterations	96

A map indexed by the simulator iteration value containing the optimized queues are returned in the response body. This endpoint has been designed for the META-pipe backend services. To support different backend infrastructures, a new endpoint that can handle and parse different type of jobs to the algorithm job type must be implemented.

The request retrieves the values in table 3.3 from the simulator. The name can be used to retrieve the events and queue for a specific simulation run.

Table 3.3: Get all simulation runs return values

Return values	Description
Name	Name of the simulation
Started	Timestamp of when the simulation was started
Finished	Timestamp of when the simulation finished

Each simulation is assigned a name in the database. The name can be used to retrieve the associated events and algorithm jobs for the simulated auto scaling run with a request to the endpoint where the id is the name of the run. In the response body three lists are returned shown in table 3.4.

Table 3.4: Single simulation run return values

Return values	Description
cloud_events	All the associated cloud events in the database which have been generated by the simulated cloud component
jobs	All the jobs for each iteration of the simulator as a list
sim_events	all the associated simulation events generated by the simulator

Simulation run

When a request is received to start a simulation run, a simulation entry is created in the database. The simulation entry contain an optional user defined name, a timestamp for when the simulation was initiated and a timestamp for when the simulation finished, which is set when the simulation has completed. The algorithm input is initialized by loading in the cluster states from the configuration file and the default queue is loaded and set as the input queue. Note that the default queue does not require estimation since it is a set of predefined algorithm input jobs where the execution time has been directly set. In the default queue config a timestamp for the simulator is also provided. The simulator timestamp defines the time the simulator begins simulating from.

The user can specify a new queue and a new cluster state in the post request body. If the queue and/or cluster states are provided they will be used instead of the default values. If a user specify a new queue, it is expected to be a set

of META-pipe jobs. These are parsed to the algorithm input jobs and their execution time is estimated using the estimator. A timestamp value can be provided in the request which is used instead of the default simulator timestamp. The time increment can be specified if a longer simulation is wanted with a coarser granularity. The number of iterations the simulator executes can be increased for a finer granularity.

The simulator executes in multiple steps and for 96 iterations (default). Each iteration after the first increases the value of the simulator timestamp by 30 minutes. The simulator first select which jobs in the job queue to append to the algorithm input queue, which is initially empty in the first iteration of the simulation. The selection is done by checking the "Created" timestamp in the algorithm job. If the "Created" timestamp is before the algorithm timestamp it is added to the algorithm input queue and removed from the job queue. The input queue, the cloud interfaces and the timestamp are passed to the algorithm. The algorithm outputs an optimized queue and a set of instances that have been either added or deleted. The simulator splits the queue from the algorithm based on the job tags in to a map of queues with the tag as its key. Each queue in the map is then processed individually.

Each queue is sorted on the job priorities. The simulator counts the number of active instances for each cloud, and the number of running jobs for each queue. The simulator iterates over the queue and retrieves each job. For each job in the queue algorithm 3 is executed.

Algorithm 3 Simulating job state transitions

```
if RunningJobs < ActiveInstances then
    Job ← RUNNING
    Job.Started ← SimulatorTime
    RunningJobs + +
else if InactiveInstances > 0 then
    if Job.State ≠ RUNNING then
        Job.State ← RUNNING
        Job.Started ← SimulatorTime
        Instance.State ← ACTIVE
        InactiveInstances − −
    end if
end if
if Job.Started + Job.ExecutionTime < SimulatorTime then
    Job.State ← FINISHED
    Instance.State ← INACTIVE
    Delete Job
end if
```

When a queue has finished iterating and executing the algorithm, each job is written to the *algorithm_job* table. *Simulator events* are created for each separate queue and are written after each queue iteration. When every queue has been iterated the queues are recombined and provided as the new input queue for the algorithm.

The output to the user is a map indexed by the simulator iteration value containing the optimized queues. The map is indexed by the iteration count, from 0 - total iterations.

Cluster States

The simulator assumes that the cluster state is reflected in the job queue. If a cloud has two running jobs, it is assumed that at least two instances will have the "ACTIVE" state. The behaviour of the simulator is undefined if there are more or less running jobs than active instances and vice versa.

Graphical User Interface

The simulator provides a GUI for evaluating and comparing the auto scaling algorithms from previous runs. The GUI retrieves all the simulation names when loaded. The user can select two simulations to compare side by side. When a simulation is selected the GUI generates two timeline graphs for each simulation, a total expected cost graph for each cloud and a total queue duration for each cloud (figure 5.6). Both graphs are created by using the *simulator events* from the database. The The graphs provided in the GUI are interactive and the user can hover over each value on the x-axis to compare the values for each trace in the graph.

Auto scaling Runtime

The auto scaling runtime is a http server. The design is similar to the simulator with the exception of not simulating the external job manager. The auto scaling runtime has no graphical user interface and is designed to run as a service with an external job manager to process the output of the algorithm. The runtime is intended to be integrated with a backend infrastructure and periodically be sent auto scaling requests to continuously optimize the job queue and the clouds and cluster the jobs will run on. In figure 3.8 is an overview of the auto scaling runtime.

The database, the algorithm and the estimator can be the same as the ones

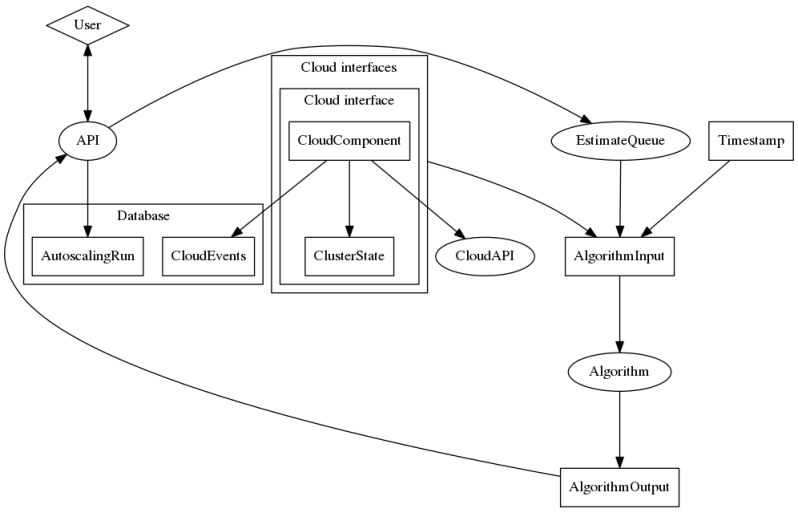


Figure 3.8: Runtime execution overview

in the simulator. The runtime uses the same interfaces and this allows the algorithm that has been tested and analysed on the simulator to be moved to the runtime without modification.

Runtime API

The auto scaling runtime has a simple API (table 3.6).

Table 3.5: Runtime API

Action	Endpoint	RequestType
Run auto scaling	/metapipe/autoscale/	POST
Get previous run	/autoscale/id?=	GET

The user creates an auto scaling run by sending a request to the endpoint with the parameters in the request body in JSON (table 3.6).

The name parameter defines the run name. If it is not set a unique name is generated. The queue is a list of META-pipe jobs that are used as the input for the estimator. The start time parameter sets the time from when the algorithm and the cloud components should do its calculations for estimating remaining execution time, queue time, and costs. The response body is populated by the output queue from the algorithm and return the http status code “201 Created”.

Table 3.6: Run auto scaling input

Parameter	Description
Name	Name of the simulation - [Optional, unique]
Queue	Input queue of META-pipe jobs - [Required]
StartTime	Timestamp for the algorithm, also used by cloud components - [Optional]

This endpoint has been implemented to handle META-pipe queues. If other type of jobs are to be supported a different endpoint must be implemented.

Getting the previous run can be done by sending a request with the name of the auto scaling run to the endpoint. The return values are in table 3.7.

Table 3.7: Get previous run return values

Name	Description
queue	The queue after scaling
cloud_events	The events generated by the cloud components
start	Start time of the run
finish	Finish time of the run

The queue returned is the optimized queue from the algorithm for the specific auto scaling run.

Auto scaling run

An auto scaling run is initiated by the user with a POST request to the endpoint. This creates an entry in the auto scaling run table in the database. The input queue is parsed to create a list of algorithm jobs that are passed to the estimator. The job execution times are estimated using the model set in the estimator. The estimated queue, the timestamp defining the start time for the algorithm and the cloud interfaces are passed to the algorithm.

The algorithm optimizes the queue and and scales the resources in the clouds by using the cloud interfaces. The cloud components add or remove resources based on the algorithm implementation and create cloud events which are stored in the database. The algorithm output is an optimized queue with the correct priorities set. The queue is then returned to the user in the response

body. The user for the runtime in production is the job manager. The job manager proceeds to schedule the jobs on the given execution environment based on the job tag set by the algorithm in the order defined by the priority value.

4

Use Cases

The users of the auto scaling simulator and the auto scaling runtime are backend developers that have an application that can be run on multiple different cloud systems and require external auto scaling and scheduling between the cloud systems. The auto scaling runtime requires an external job manager which handles launching jobs on the clouds depending on the tags and priorities set.

Develop and simulate new algorithms

An algorithm should be developed by the users to fit their specific requirements for the application and their jobs. Each type of application have different requirements. Some applications has a need for fast processing of jobs, while others require low costs. The algorithm should reflect these needs.

Implementing an algorithm

Implementing an algorithm can be done in 6 steps:

1. Split the input queue on desired tags (if more than one)

2. Define the scheduling policy (which jobs should have priority and where

to run)

3. Assign priorities and tags to the jobs based on the policy
4. Define the scaling policy (how much to scale and when)
5. Scale the compute resources
6. Return the queue with reprioritized jobs

The implementer should set the conditions for when the algorithm should assign different priorities and where to schedule the job, and assign these priorities and tags to the jobs. The implementer should also define a scaling policy to decide when to add or remove resources. Scaling the compute resources on the clouds should be done by using the cloud interfaces while executing the algorithm. The reprioritized job queue should be returned by the algorithm.

Simulating the algorithm

Once a new algorithm has been developed the user must set it as a dependency for the simulator. Currently the simulator can only have a single algorithm dependency. Once this is done the simulator can be started. The simulation can be run by making a POST request to the simulator API. Once the simulation has been finished, it will be present in the simulator GUI.

Deploying an algorithm

Once an algorithm has been developed and is ready for production it can simply be moved by setting it as the algorithm dependency for the auto scaling runtime. The runtime must be recompiled and the new binary must be started for the changes to take effect.

Adding a new compute resource provider

Adding a new resource provider can be done in 4 steps:

1. Implement new cloud component for the provider
2. Choose a tag name (used by the cloud interface map, estimator and the

jobs)

3. Update the algorithm (if needed)

4. Update estimator

The new resource provider require an associated cloud component for the algorithm to use. The cloud component must satisfy the cloud interface and be provided as a dependency for both the simulator and the auto scaling runtime. If the algorithm refers to cloud platforms by name (keyed index into the cloud map), it has to be updated to support the new cloud component. The estimator must be updated to support the new provider, be creating a new model for the new tag. The estimator require a set of jobs that has been run on the new compute resources. If this is not available, it is recommended to run test jobs in order to generate a training set for the estimator.

Adding a new application

The users can add a new application in 6 steps:

1. Create unique tags for the jobs

2. Add an additional parameter table to the database

3. Populate the database with training data

4. Create a new estimator

5. Create a new endpoint to the simulator and the auto scaling runtime

6. Create a job parser that converts from the new job description to algorithm jobs

If the users have at least another application running on the same runtime when adding a new application, the user must ensure that the job tags that specify the execution environment is unique. The estimator_training table in the database is a generic table for storing jobs meant for training estimators. A new parameter table must be added to store the parameters for the new jobs. The user should populate the database with training data for the new estimator. A new estimator must be implemented that can predict the new jobs. The new estimator must be set as a dependency for both the simulator and the auto scaling runtime. The implementer must also create a new endpoint

in both the simulator and the auto scaling runtime. The new parser must be implemented that can convert from the job descriptions of the new application to the algorithm jobs. A method in the simulator is provided to run the simulation.

5

Results

The auto scaling runtime and simulator are both using the estimator to calculate costs, and the total duration for the jobs in each queue. The simulations use the estimated values to calculate the finish time for a job and as a consequence when the instances change state. Therefore the results from both the auto scaling runtime and the simulator are dependent on the accuracy of the estimator.

Estimator: Linear regression

The linear regression model achieved different results for the different input data. First all the jobs were used for the training set, with and without parameters set. Second only the jobs which had their total runtime and queue duration values set by the META-pipe backend were used. Finally only the jobs with their total runtime and queue duration values set were used and split to three different models based on the target execution environment.

Single model for all jobs

Figure 5.1 contain a scatter plot with all the jobs that have data sizes available, a total of 1137 jobs. . We can see that there are no clear linear solution for the META-pipe jobs. 977 of the jobs in the META-pipe database does not have their

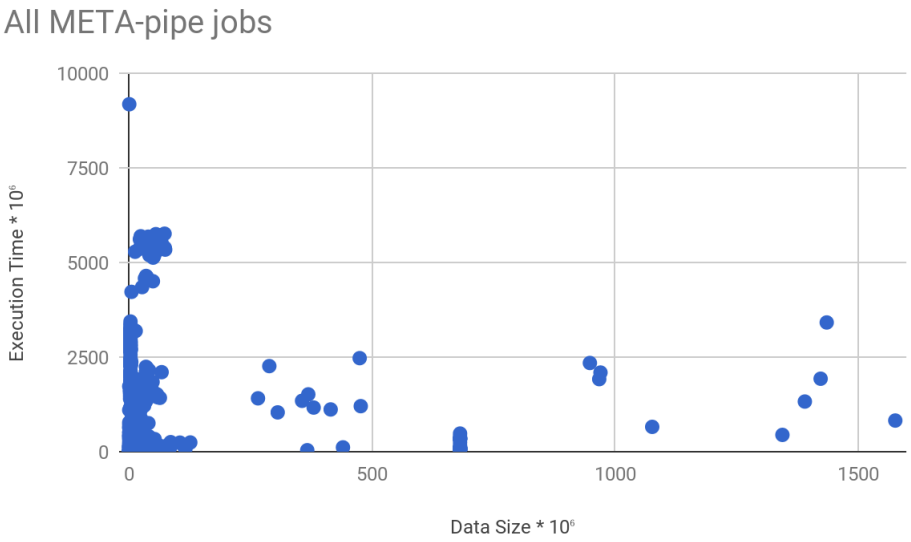


Figure 5.1: Relation between execution time and data size for all META-pipe jobs

execution time calculated by the META-pipe backend. The calculated execution time for these jobs are done when adding them to the database by using the start time and the final heartbeat time of the job. The calculated execution times contain the queue time. The queue time for the jobs vary greatly, from seconds to weeks. This can be the reason for this to show no clear correlation between data size and execution time. The input data was expected to have a more clear separation between the three data categories, virus, genome and meta-genomes. If there had been a clearer separation it could have lead to using different models for estimating the execution time based on the data category.

Figure 5.2 show the correlation between execution time and input data size for the jobs with their execution time and queue duration set by the META-pipe backend. 160 out of 176 jobs which had the queue duration value set had valid input data available on the META-pipe storage system. From this we can see that there is a linear trend. 9 jobs that have a data size above 600MB have all been executed on AWS. These jobs were run as part of testing to run the META-pipe application on AWS and have used different instance types for executing, which is the reason for the large spread in execution times for the same input data size.

Figure 5.2: Relation between execution time and data size for META-pipe jobs with queue duration and execution time set by the META-pipe backend

Table 5.1: Models

Input data	Function	Variance	R^2
All Jobs	$-994.9 * 10^6 + datasize * 0.35 + parameters * 422.6 * 10^6 + contigs * 238 * 10^3$	$1.83 * 10^{17}$	0.14
All jobs, only data size	$5410.8 * 10^5 + datasize * 0.40$	$3.35 * 10^{15}$	$0.29 * 10^{-2}$
Only jobs with queue duration	$-180.4 * 10^4 + datasize * 0.32 + parameters * 122.4 * 10^2 + contigs * 113.6 * 10^2$	$2.58 * 10^{15}$	0.57
Only jobs with queue duration, only data size	$1143.1 * 10^6 + datasize * 0.32$	$2.37 * 10^{15}$	0.57

Multiple linear regression models were trained. In table 5.1 is an overview of the models created by the different sets of input data and, with and without using the job parameters. The R2 value represent the fitting of the model to

the data. A value of 1 is a perfect fit for the model. Variance describes the fluctuation of execution times predicted by the model. The model using all the jobs without the parameters has an R2 value close to 0. This was expected when comparing it to the scatter plot of all the data. The same holds true for using all the jobs and including the parameters. The fitting of this model is higher, but the value is not high enough to be useful for estimating execution durations. The models created using only the jobs with their execution time set by the META-pipe backend resulted in a better fitting with a R2 value of 0.57. A R2 score of 0.57 shows that there is correlation between input data size and execution time, but it can be improved. Since the fitting of the models were poor, a new models were created.

Separate models for different execution environments

Each supported compute cluster have different performance capabilities. From this it is clear that each compute cluster estimation should therefore be handled separately. However an issue with this is that the number of jobs present for each tag is low and results in a poor model.

Table 5.2: Job count

Tag (execution environment)	Count
csc-* (cPouta)	40
metapipe-* (Stallo)	14
-aws- (AWS)	84

In table 5.2 the job count for each tag is listed. The job count with the tag for Stallo is underrepresented by only 14 finished jobs.

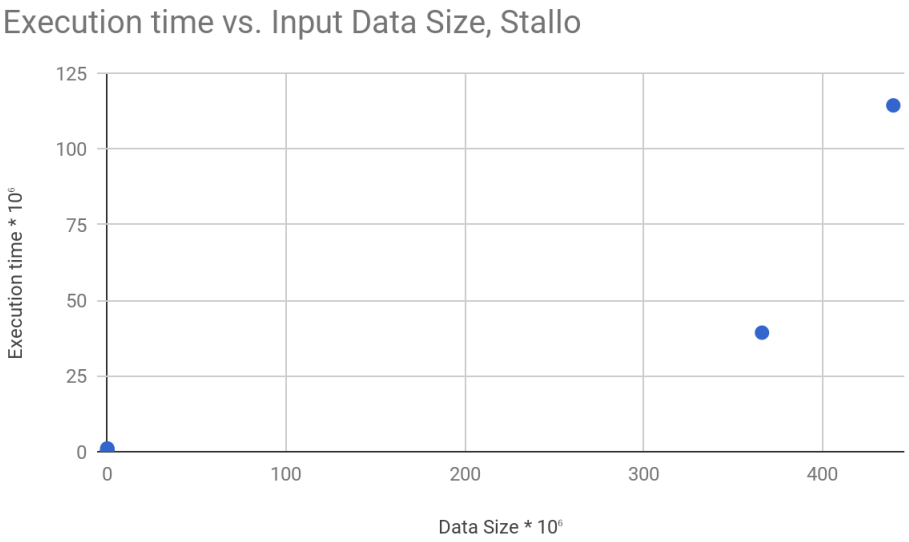


Figure 5.3: Jobs on Stallo

The jobs that have been executed on Stallo (figure 5.3) have an approximate linear trend. A possibly better model for Stallo could to use an exponential regression function for estimating the execution times.

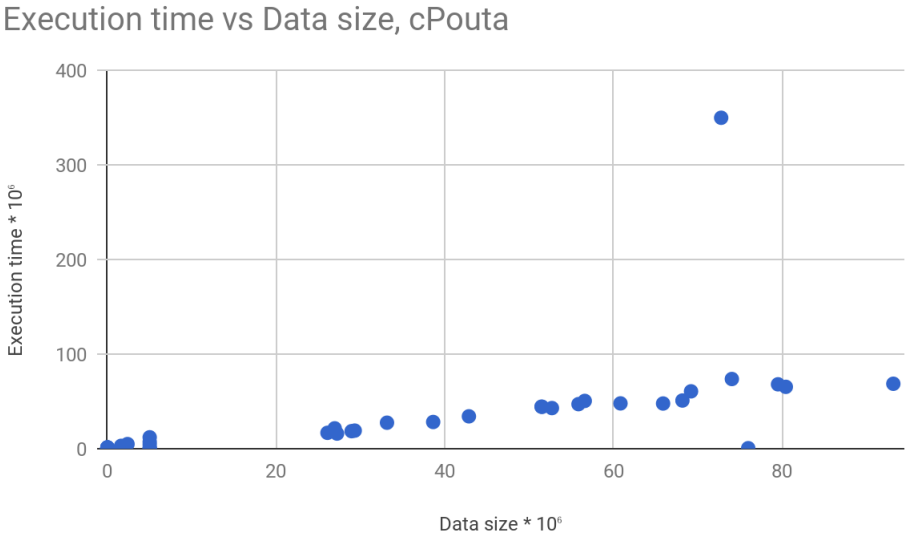


Figure 5.4: Jobs on cPouta

The figure 5.4 show the jobs that has been executed on cPouta. This again shows a linear trend. There are exceptions that do not follow the trend, however the most extremes should be filtered out. One thing to note is that the jobs ran on cPouta has not completed any large datasets. If there is a linear correlation between execution time and the input data size for META-pipe jobs on cPouta, the lack of jobs that have finished large datasets should not be a problem. However it is uncertain if the optimal regression model would be linear or exponential without any training data for jobs that have finished large sets of input data.

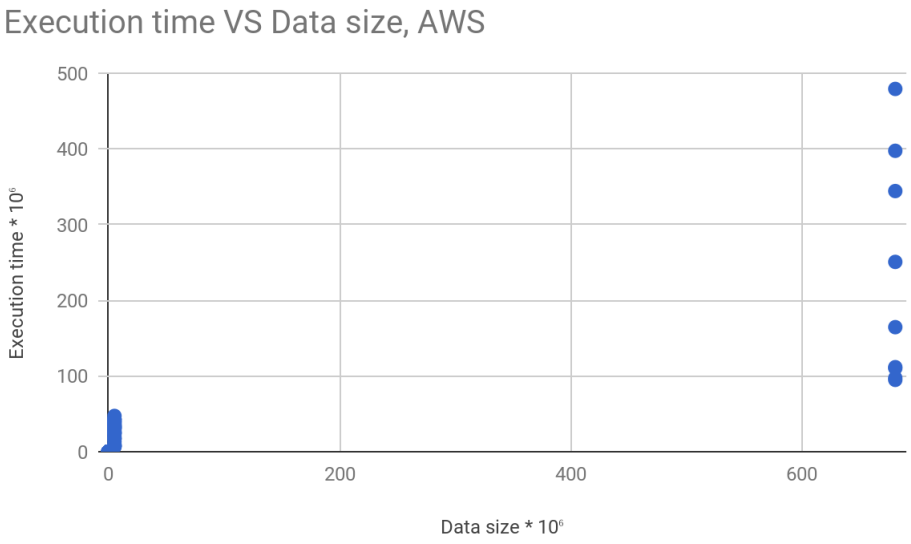


Figure 5.5: Jobs on AWS

The jobs that have executed on AWS (figure 5.5) have no clear correlation between input data size and execution time. The META-pipe jobs that have been executed on AWS were part of an experiment to evaluate different compute instance flavours and storage solutions. There is currently no method to find the instance flavour the job has executed on. However a regression model can be trained using the jobs.

Table 5.3: Models

Tags	Function	Variance	R^2
metapipe	$21.6 * 10^4 + datasize * 0.20 + parameters * -3480.54 + contigs * -498.37$	$7.42 * 10^{14}$	0.86
csc	$22.9 * 10^{19} + datasize * 1.01 + parameters * 17.0 * 10^4 contigs * -45.8 * 10^{16}$	$1.06 * 10^{15}$	0.34
Only jobs with queue duration	$22.8 * 10^{19} + datasize * 0.32 + parameters * -18.9 * 10^{18} * 10^2 + contigs * -30.2 * 10^{16}$	$4.45 * 10^{15}$	0.66

The models (table 5.3) were trained using the job parameters and data size. The trained models have a better fitting compared to the previous models that were not split based on the execution environment. The cPouta model has a R2 value of 0.34, which is due to the execution time outliers in the training data. The outliers skew the model and the outlying jobs should have been removed before training the model. The Stallo model should be able to predict META-pipe jobs which will execute on Stallo within an acceptable margin with a R2 value of 0.84. The AWS model has a better fitting than expected with a R2 value of 0.66, however the model should not be used for important decisions in production, since the instance flavour have not been decided upon by the META-pipe developers. The jobs used in the experiment should be ignored and future jobs should be used to train the model.

When more jobs are used to train the models they can shift and the fitting can become either better or worse. The models should be evaluated continuously to ensure that the estimations are acceptable. The assumption that the execution time have a linear relation to the input data size can be wrong since the number of jobs available is low and not all data categories are present for all the execution environments (figure 5.4 and 5.5). A conclusion for the relation between input data size, parameters and execution time is not possible due to the lack of data, but the models are used for the simulator.

Simulator

The simulator estimator uses the models which estimates execution time based on the target execution environment. The job queue used to compare the algorithms have been manually developed. The jobs are created as already estimated jobs that can directly be used by the algorithm. The execution times were predefined to avoid the estimator estimating different execution times for jobs between runs of the simulator, since the estimator is re-trained when the simulator is started.

The evaluation criteria set for the simulator are:

- How well it simulates the external job manager by scheduling jobs correctly
- The ability to provide insight about the algorithms and their effect on the queue and the clouds
- How well the simulator reflects the algorithms and clouds in production

Job queue

The job queue[1] used for testing the algorithms and the simulator consist of 23 jobs with different tags (table 5.4) that all have the same parameters. These parameters have not been used by any of the algorithms and does not affect the result.

Table 5.4: Job count

Tag (execution environment)	Count
csc (cPouta)	7
metapipe (Stallo)	4
aws (AWS)	7

Each job has the execution time set to either a short job, a medium job or a large job. These values might not represent the actual execution times on the execution environments, but they are used as a proof of concept for the simulator. The start time set for these jobs (see source code) are either from the beginning of the simulation or at a given point after, while still within the default time range of the simulator, 30 minutes timesteps and 96 iterations of

1. See source code under uit-go/metapipe/metapipe.go for job creation function

the simulation, a total of 2 days simulated. A single job has the state “RUNNING” in the dataset, it has the AWS tag. The jobs with the empty tag have estimated execution times for each execution environment. The test dataset is also used as the default job queue when nothing is declared in the simulation request.

Cluster states

The cluster states (table 5.5) used for testing the algorithms and the simulator reflect the job queue where only the AWS instances have a single “ACTIVE” instance.

Table 5.5: Cluster states

Tag (execution environment)	INACTIVE	ACTIVE	Limit	Cost
csc (cPouta)	2	0	5	0.78 USD / h
metapipe (Stallo)	1	0	7	0.48 USD / h
aws (AWS)	2	1	8	0.68 USD / h

The test cluster state is also used as the default cluster state for the simulator when this is not declared in the simulation request. Prices for Stallo and cPouta are in this case fictive, the AWS price is based on the c5-4xl compute instance on EC2.

Algorithm comparison

The algorithms that are used to evaluate the simulator have been defined in the design section under the implemented algorithms subsection. Each algorithm give different results for the same queue depending on what it does to the cluster states and the job queue. In the following graphs the value on the x axis is the algorithm timestamp, and on the y axis is the accumulated queue time for each simulation iteration

Naive algorithm

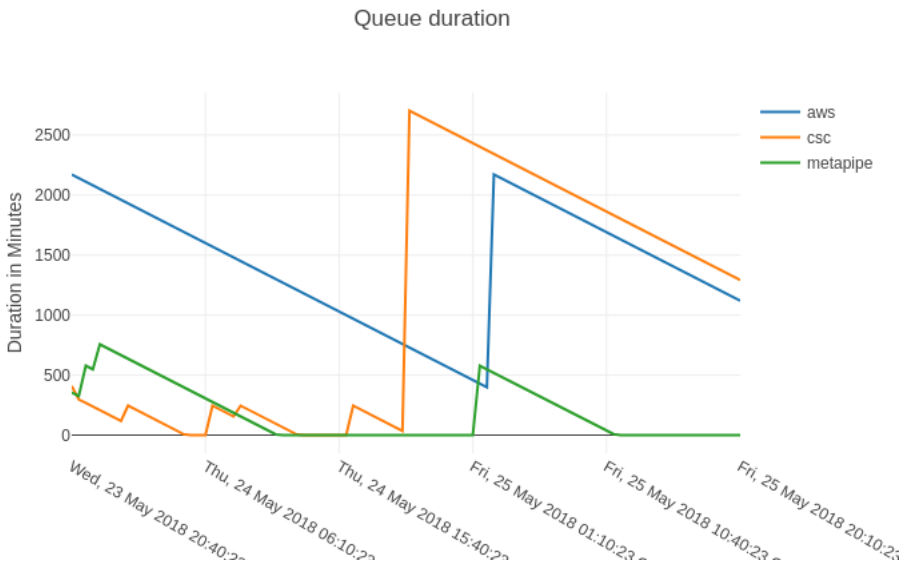


Figure 5.6: Total job durations for each cloud using the naive algorithm

The simple algorithm optimize the queue based on the cost and queue duration and assign the jobs without tags to the queue where it is most efficient to run the job. Each spike in figure **??** is a job being assigned a tag. During the simulation each cloud is scaled to the required number of instances for the queued jobs to run on, up to the limit set. Resources are deleted or reused when a job completes (see appendix A for event outputs). The simulator schedules each assigned job to the correct cloud based on the output queue of the algorithm. This has been carefully verified by manually analyzing the input queue and the behaviour of the algorithm to predict where the jobs would run. This coincides with the duration graph. A graph of the cost has not been shown since it follows the exact same trend as the duration graph. The reason for this is that the cloud components currently only use a single instance type for the different execution environments.

Worst-case algorithm

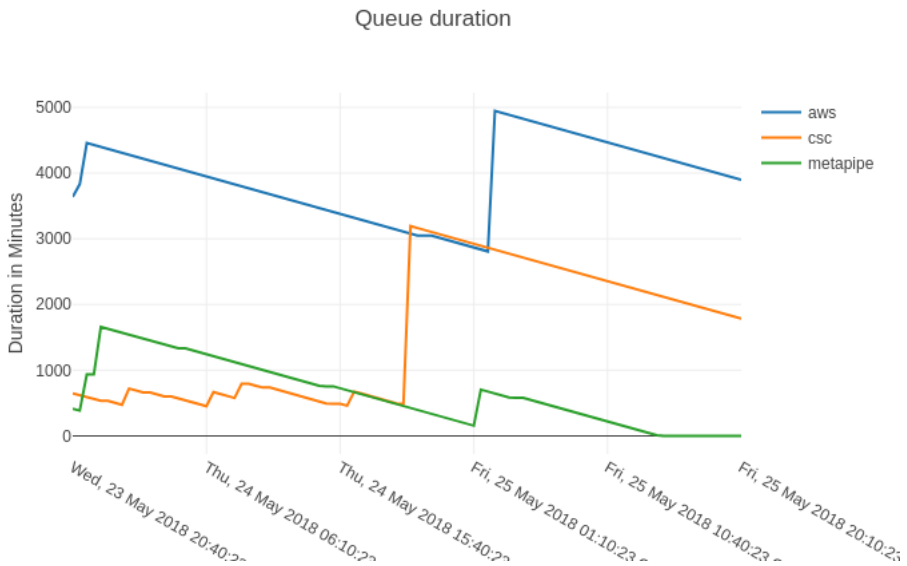


Figure 5.7: Total duration for each cloud using the worst-case algorithm

The bad algorithm reduces the amount of available instances to a single instance, as long as it satisfies the amount of running jobs. It will not remove an instance that has a job running on it. The bad algorithm behaves as expected. The queue durations (figure 5.7) are longer, however the cost will stay the same since the number of instances used are reduced to one and only a single instance type is used. When comparing the simple algorithm to the bad algorithm it is clear that the total duration for each queue is visibly shorter. The shorter duration of the queues will ensure that the users of the system where the algorithm is deployed have less waiting time for the jobs to complete. The simulator GUI enables the user to compare both the duration graphs and the cost graphs to easily make decisions on which scaling algorithm to use.

Null algorithm

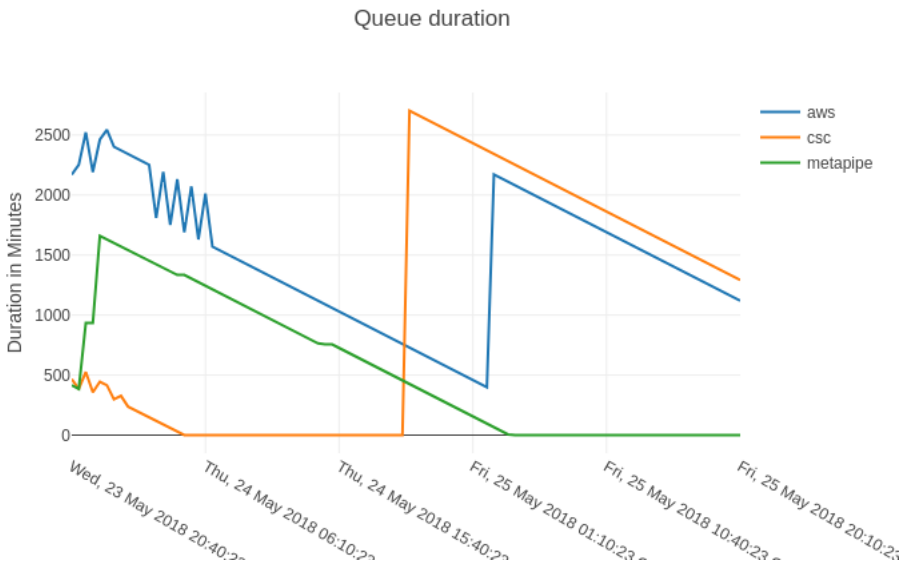


Figure 5.8: Total duration for each cloud using the null algorithm

The null algorithm does nothing to the cluster states or the input queue. As a consequence no jobs with an empty tag is scheduled on any execution environment. Hence there are no spikes from jobs being assigned a tag. The spikes present in the graph is for the jobs which have predefined tags. However it is unclear what causes the spikes in the aws trace in the beginning.

The nil algorithm performs comparably well to the bad algorithm, with executing the jobs predefined to run on both AWS and cPouta equally to the simple algorithm. In the worst case algorithm the total queue duration is overall higher than the null algorithm. From the cluster states we know that for AWS at least three jobs can run in parallel. However the nil algorithm should not be compared against since it does not add jobs with empty tags to any queue. The simulator outputs the queue for each iteration and and this can be used to ensure that all jobs have been assigned a tag at the end of the simulation.

Summary

The simulator can accurately simulate the algorithms over time (based on the estimations from the estimator) and provides a graphical interface for comparing and analyzing the executed algorithms. The simulator schedule jobs in the correct order based on the priority and on the correct execution

environment based on the tags set in the algorithm. The simulator does not consider the start up time for either the jobs or the instances. Because of this, the simulator might not accurately reflect how the algorithms and clouds will behave in production.

6

Related Work

CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems

CloudScale [2] is a fine grained elastic resource scaling system for multi-tenant cloud computing infrastructures. It runs inside the cloud system and is connected to the VM's in the infrastructure through the Xen hypervisor. A prediction based method for resource scaling is used to achieve elasticity. It is uses a technique based on fast fourier transforms (FFT) to identify burst patterns from the resource usage time series which is retrieved from the Xen hypervisor. If such a pattern is found a resource demand model [2] is used. If no pattern was identified, a discrete time Markow chain based model is used to predict the resource usage in near future, because the FFT technique requires a pattern to calculate the expected resource demand. These techniques calculate the expected minimal resource usage. Scaling conflicts occur when multiple applications are scaled concurrently and there is not enough resources for the applications to scale. The CloudScale system resolves scaling conflicts by either locally handling the conflict or migrating processes to different virtual machines. Scaling conflicts occur when multiple applications are scaled concurrently and there is not enough resources for the applications to scale. The local conflicts are resolved either by strictly enforcing a cap on the resource usages of the processes in conflict or by allowing the cap to be raised to the maximum value for a short duration, resulting in resource waste, but alleviating the conflict until the resources are freed. Scaling conflicts that require VM migrations are

costly and only done if require to meet the expected deadlines. Migrations are triggered based on the resource usage and a predictive method is used to migrate before the conflict happens. This system relies heavily on the Xen hypervisor for the time series data and for the management of the virtual machines. The concept of using time series data for analysing the resource requirements is a relevant concept for scaling META-pipe. Real-time predictions is not in the scope of this project, but using the time series data for analyzing the resource dependencies and requirements is an approach that should be explored in future work.

Jockey: Guaranteed Job Latency in Data Parallel Clusters

Jockey[1] is a scheduling system that analyses the complex internal structure of a data analysis job, to step by step predict the resource usage of the job. Jockey was implemented to run in a cluster environment as a daemon process which continuously analyse the allocated data in the cluster. Each job has a individual tracking mechanism and is dynamically allocated resources. The Jockey system uses precomputed statistics based on a simulator which encapsulates the job complexities in order to estimate the resource usage and its complexity. The simulator uses a per stage allocation and time usage algorithm to calculate both the resource usage and time estimation for a single job. In order to retrieve a jobs statistics the internal structure is used and each stage is given both an estimate and resources needed. The jockey system is designed to work with MapReduce type of jobs (or similar data parallel processes model). Performance and synchronization issues arise with these type of jobs when an interrupting operation is required that has to synchronize all the nodes. These issues can lead to a complete halt in the entire system. Jockey requires an existing dataset to create the precomputed statistics

Jockey is relevant to scaling META-pipe. Jockey limits the implementation to a single cluster and it is not a separate entity from the cluster it operates on. Precomputed statistics are used to estimate run times for individual jobs based on the job parameters. The limitation of only scaling a single cluster makes it bad match for scaling the META-pipe job queue, it could be used to scale the cluster internaly instead of using the cloud components to scale the clusters.

Dynamically Scaling Applications in the Cloud

L. M. Vaquero et. al.[5] describes the basic structure and the necessary components to achieve cloud scaling from the view of the server. In this paper it is assumed that virtual machines are used for containing user application and new virtual machines are allocated or reused for scaling an application. A major component for cloud scaling is a load balancer that distributes the work delegated to the cloud. This is a elasticity controller, which can be implemented in two ways.

- A multi-tier implementation that synchronizes between multiple controllers at the VM abstraction level
- A single, global controller that controls all applications and VM's in the cloud. This controller operates on the logical tier in the cloud, for instance, an application is scaled when the limit for requests is reached

The methods for elasticity presented in this paper is out of date for a modern commercial cloud, as it is written in 2011. Hower for the automatic scaling algorithm for META-pipe, a single global controller for providing elasticity would be sufficient. L. M. Vaquero et. al. describes the container-level scalability where multiple user components are used in a single virtual machine. This can reduce the time and resource usage of allocating new virtual machines for each user application. The method for running, if possible, multiple applications in a VM is used in the design of the automatic scaling algorithm.

The auto scaling algorithm which have been developed for META-pipe require the ability to schedule jobs accross different cloud platform and on multiple clusters. The components described in *Dynamically Scaling Applications in the Cloud* focuses on a single cloud platform. The scaling of a single META-pipe job can be done by ensuring that the components necessary to achieve scaling on the cloud platform is present and in effect.

CloudSim, OMNeT++, SMICloud

There are multiple frameworks for simulating different structures and systems. Each of these systems contain methods for simulating an algorithms or provide data for cost and performance decisions. These have been viewed as possible solutions to the simulator needed to analyze the auto scaling algorithms.

CloudSim

CloudSim[6] is a framework for simulating and modeling of cloud computing infrastructures and services. It used to test new applications and experimenting without provisioning an actual cloud system. It can also be used to compare potential costs and energy consumption. CloudSim enables the user to identify bottlenecks before deploying. The goal for CloudSim is to provide a general and extensible simulation framework that can be used by both scientist and industry developers so that they can focus on design issues and not low level issues in specific cloud providers. Internal META-pipe runtime application could be analyzed on CloudSim, but the auto scaling algorithm is abstracted a level above the cloud and could not be simulated using CloudSim since it targets multiple execution environments.

OMNeT++

OMNeT++[8] is a network simulator that is extensible, modular and component-based. The meaning of network for OMNeT++ is in the broader sense of the word, which includes, wired and wireless communication networks, on-chip network, queueing networks and etc. Many components are provided for free. These components can be combined into a structured network. The main usages are internet protocols simulation and, performance modeling, and so on. A graphical user interface is provided for visualizing the network flow. As with CloudSim, OMNeT++ would not be a good simulator for the scaling algorithm, but it could be use to analyze the network communication of the cloud components. This could help limit bottlenecks which could potentially occur during resource provisioning on the different execution environments.

SMICloud

SMICloud [7] is a framework for comparing and ranking cloud computing services. With the multitude of cloud providers it is necessary to be able to differentiate and compare the providers to find the most suitable system for an application. The framework uses a “Service Measurement Index” (SMI) which is designed based on ISO standards. It defines a set of attributes which defines the quality of service. These attributes consist of Key Performance Indicators (KPIs). The KPIs consist of 15 different metrics. The score for each cloud provider is based on how well it meets the SMI and on previous user given scores and performance. Three different ranking systems are proposed in the framework.

- Service quality ranking using an analytical hierarchical process (AHP)

- Cost-quality based ranking
- Time complexity of AHP

A system like SMICloud could potentially be a valuable framework for the auto scaling algorithm if it could provide a proper cost analysis and comparison of different clouds for the META-pipe application, which could be used to improve the scaling algorithm. The SMICloud framework is not a finished product, and implementing such a system would go beyond the scope of this project.

7

Conclusion

We have proposed a design for an auto scaling framework which support scaling different compute clusters and schedule jobs on different execution environments. The user can use the simulator to visualize the algorithm decisions and the visualization provides a method for both debugging and comparing the algorithm results. Three algorithms have been developed for testing the simulator. The simulator simulates the external job scheduler and each algorithm behaves as expected. An estimator for the META-pipe jobs have been developed and evaluated. The estimator estimates the job execution time based on the job parameters, its input data size and the execution environments.

We have seen that the existing systems do not offer a solution to the problem of scalling accross different execution environments. The related work all focus on a single cluster or a single execution environment.

The designed simulator and auto scaling framework support simulating auto scaling algorithms and can accurately simulate META-pipe job queues when the cluster states are predefined. The simulator GUI can be used to analyze the resource allocations, cost changes and total expected queue time. The analysis can be used to further develop auto scaling algorithms and to compare the advantages and disadvantages when choosing which algorithm to use. The linear regression models used by the estimator does not fit perfectly to the training data, but for both AWS and Stallo the model fitting is within acceptable limits. The assumption that there was a linear relation between execution time and data size has held true.

The auto scaling runtime has not been evaluated. The runtime cloud components do not have an implementation and the runtime could therefore not be used to scale compute clusters on any of the execution environments.

Integrating the auto scaling runtime (once the cloud components have been implemented) and using an auto scaling algorithm would prove beneficial for an application such as META-pipe. Comparing the null algorithm to the naive algorithm show that the scaling would reduce the overall queue processing time by spreading the workload to the execution environments with the shortest expected execution time.

Future Work

The simulator only simulates a single type of external job manager. For the simulator to simulate the external job manager it must schedule jobs based on the job priority. A possible extension to the simulator would be to design an interface for the framework which could be used to simulate the external job manager.

The cloud components for the supported execution environments must be implemented before the auto scaling runtime can be deployed on the META-pipe backend. This is a task the META-pipe developers must consider before deciding to integrate the runtime. A careful approach should be taken when integrating the runtime, since it has not been fully tested.

A new estimator that support multiple instance types should be developed. From the results of the estimator implemented for META-pipe it is clear that the different instance types can be a major factor for the execution time of a job. For each instance type there is trade off for both cost and execution time.

References

[1] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In Eurosys 2012, pages 99–112

[2] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in Proceedings of the 2Nd ACM Symposium on Cloud Computing, New York, NY, USA, 2011, pp. 5:1–5:14.

[3] Amazon Web Services, Auto Scaling (`https://aws.amazon.com/autoscaling/`) Accessed 12/12/2017

[4] Constraint based programming (`https://developers.google.com/optimization/cp/`) Accessed 12/12/2017

[5] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically Scaling Applications in the Cloud," SIGCOMM Comput Commun Rev, vol. 41, no. 1, pp. 45–52, Jan. 2011.

[6] CloudSim (`http://www.cloudbus.org/cloudsim/`) Accessed 12/12/2017

[7] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," Future Gener. Comput. Syst., vol. 29, no. 4, pp. 1012–1023, Jun. 2013.

[8] OMNet++ (`https://www.omnetpp.org/`) Accessed 31/05/2018