Imagely: A cloud environment for online image processing

Authors: SB Ramalingam Santhanakrishnan (4740270), K Kleeberger (4748476) and B Jain (4745507)
ICT Innovation, EEMCS, TU Delft
Emails: {S.B.RamalingamSanthanakrishnan, K.Kleeberger, B.Jain}@student.tudelft.nl

Course Instructors: DHJ Epema, A Iosup and A Kuzmanovska
PDS Group, EEMCS, TU Delft
Emails: {D.H.J.Epema, A.Iosup, A.Kuzmanovska}@tudelft.nl

Lab Assistant: BI Ghit
PDS Group, EEMCS, TU Delft
Email: B.I.Ghit@tudelft.nl

Teaching Assistant: A Parthasarathy
EEMCS, TU Delft
Email: A.Parthasarathy@student.tudelft.nl

***Abstract*—*Imagely* is a cloud service for batch image processing, designed for the specific requirements of WantCloud BV. Images are submitted by the end user, which are then processed by the service and the user is notified. It runs on Amazon Web Services (AWS) IaaS barebone compute resources to provide elastic scaling capabilities and its system design adapts the standard master-worker batch processing reference architecture pattern provided by AWS. The scheduler and provisioner are written in NodeJS in reactive style. We run three workloads each on three combinations of VM and scheduling policies with a hundred images. We show that the system can stay responsive under peak load to more than 50% of the traffic at under 40 seconds while reducing the overall workload completion time.**

I. Introduction

WantCloud BV is a popular image processing company and are pioneers in converting raw satellite images to standard image formats consumable by users on the web browser. As the company is expanding into other domains and adding new customers rapidly, their existing deployment is unable to cope up with peak traffic demands and maintain service level agreements (SLAs). Thus, WantCloud BV is evaluating an IaaS-based solution that can scale with the demand.

The proposed system utilizes Amazon Web Services (AWS) [1] IaaS cloud service provider's compute instance, EC2 (Elastic Cloud Compute) for providing on-demand compute capacity to process the workloads. We design a custom provisioner and scheduler using the AWS low-level SDK, EC2 API for requesting and releasing virtual machine instances (VMs). We also utilize Amazon Machine Images (AMIs) for pre-packaging the application code, configuration information and the operating system for the instance to save time during its launch. For image processing and conversion, we use the ImageMagick [2] program. We wrap ImageMagick with a thin HTTP server written in NodeJS [3], which forks ImageMagick as a child process and performs read/write through stdin/stdout pipes, the instances which perform image processing are known as the workers. For analysis purposes, we restrict the scope of the worker to scaling down, rotating and converting the image from TIFF [4] to JPEG format [5].

The system design follows the AWS reference architecture for batch processing [6], we differ in that we have implemented our own custom scheduler and provisioner in a reactive fashion and have placed all the task queues and resource management responsibilities on the master node. The master node accepts the image processing task requests via HTTP and launches multiple worker nodes, based on the given policy. It then balances and allocates the tasks to the workers and also monitors the progress. It also keeps accumulating reports on completion of each task and launch/release of each instance during its lifetime. The policy targets the provisioner with parameters for minimum and maximum number of workers to launch and the scaling-up decision threshold.

The test workload is a collection of 100 TIFF images from NASA's publicly available Landsat satellite image repository [7], file sizes ranging from 40-60 MB. To determine the baseline (current) effort required, we run the workload on a static VM deployment and to analyse the scaling and scheduling capability of the cloud solution, we test the workload with two policies, FCFS and OD-S, explained below. The workload itself is varied with three exponential Poisson [8] mean arrival time distributions to simulate realistic traffic patterns.

In section II, additional background information on the application is provided. In section III we illustrate the design of the system in detail with focus on specific aspects pertaining to WantCloud's requirements and in section IV we present the experiments conducted along with the results, analysis and potential improvements. We conclude the report in section V with a short summary on the main findings.

II. Application

The web-based application receives tasks from end-users in JSON format [9], which point to a source image URL and defines a sequence of image manipulation operations such as scaling, rotation and finally file format conversion from TIFF to JPEG. This JSON request is forwarded to an idle worker in the resource pool by the master node. Internally, ImageMagick program is used for processing the image and it is forked by the worker's NodeJS web server. The application is I/O bound during retrieval of the source image and writing the result to AWS S3 storage (the NodeJS web server handles all I/O), CPU and memory intensive when it performs the actual image manipulation and conversion. A task is declared to be a failed if any of the above three steps encounter an error or the entire task does not complete within 30 seconds.

A. Requirements

In order to replace WantCloud's current system, the following requirements are to be met by the new system.

- *Automation*: The system should be automated for creation, provisioning and other activities with minimal human intervention.

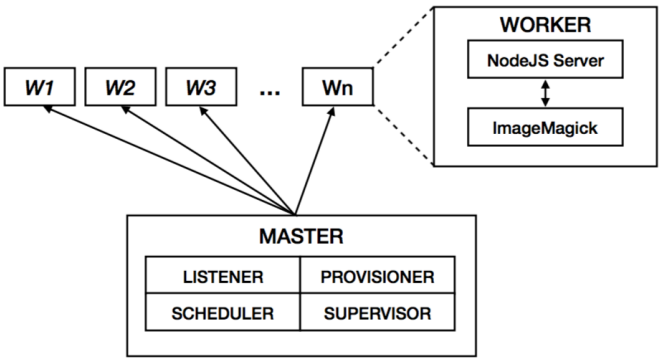

Fig. 1. Overview of system design.

- *Auto-scaling*: The system should be able to detect fluctuations in demand and scale up or down the VM resource pool automatically.
- *Load Balancing*: The system should be able to best utilize the available pool of compute resources for the workload through allocation policies.
- *Reliability*: The system should be able to retry failed tasks, ensure availability through redundancy and recover from unexpected failures.
- *Monitoring*: The system should be able to keep track of resource usage and provide information on metrics to aid future decisions.

III. System Design

As mentioned in section II, the server (master node) accepts the tasks and adds it to the pending task queue. The task is then allocated by the scheduler to an available worker instance at the next scheduling cycle. The workers finish the task and notify the end-user via E-mail about the job completion or failure, the stored location of the result and returns control to the master with usage info.

A. Overview

The system, illustrated in Figure 1 consists of the following components, placed in the master node:

- *Listener*: The NodeJS server, which accepts client requests through HTTP, parses it, creates a corresponding internal task representation and adds it to the pending task queue.
- *Scheduler*: The scheduler is invoked as soon as a task enters the pending queue (debounced by 1 second) or it is invoked by the system periodically. It matches tasks in the pending queue to free worker instances in the resource pool.
- *Provisioner*: The provisioner manages the request and release of worker instances in the resource pool according to the given policy.
- *Supervisor*: The supervisor is responsible for bootstrapping the system, monitoring the finished/failed tasks, monitoring health of the instances, maintaining an audit log of all the tasks and instances to generate a report when requested and finally perform cleanup and release of instances on unexpected shutdown (SIGINT, SIGTERM) if possible[1].

B. Resource Management Architecture

The components listed in subsection III-A each have their own life cycle and associated events. Since we follow the reactive pattern, we illustrate the resource management architecture by tracing the execution of a task through the system, handled by the various components across phases listed below and in Figure 2.

1) *Bootstrap*: This phase marks the start of the system, before the arrival of tasks. The supervisor starts the scheduler, provisioner and the listener and broadcasts a BOOTSTRAP event which the components listen to and perform initial setup activities if any. The supervisor also invokes the AWS API with a DESCRIBE call to get a list of already started workers if any, this is essential to prevent orphaned nodes in case the master restarts from a failure recovery.
2) *Scheduling and Allocation*: The listener listens for incoming tasks from end-users and adds it to the pending queue with the ADD_TASK event, after marking the arrival time. The scheduler picks tasks from the pending queue, picks free instances from the worker pool, matches the two and publishes the EXECUTE_TASK event. Another part of the scheduler asynchronously listens to EXECUTE_TASK, invokes the worker node, waits till the task is finished/failed and adds it to the pending queue again on failure. The allocation of jobs to workers is influenced by three policies – priority of tasks in the pending queue, criteria of allocation to a worker instance and the maximum number of retries for a failed task.
3) *Provisioning*: The provisioner is also invoked periodically and performs two duties – pick free instances from the resource pool for termination, request for a

[1]During bootstrap, it queries AWS for any existing workers and adds them to the resource pool.

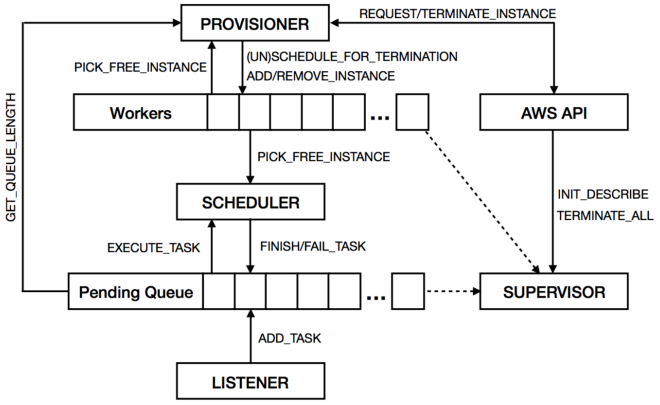

Fig. 2. Resource management architecture with message flows.

new instance based on the given policy. As soon as we request the AWS API for a new instance, we repeatedly hit the health check HTTP endpoint of the worker application to determine if it is ready[2]. If it is ready before a timeout, it is added to the worker pool else terminated immediately. Note that in regular cases we do not terminate immediately, instead we first schedule a worker for termination in the future with SCHEDULE_FOR_TERMINATION event, to prevent thrashing. A separate part of the scheduler listens to this event, waits for the scheduled time and then releases instance by calling AWS API, however during the wait, it also intercepts any calls to request for new instance, cancels the termination schedule and adds the instance back to the pool. The policies applied to provisioning are the minimum/maximum number of instances to maintain, queue threshold for scaling decisions, and the worker instance type to launch.

4) *Termination and Reliability*: During the termination phase, the supervisor shuts down each instance in the resource pool with a TERMINATE_ALL call. The reliability of the system is ensured by configuring auto-restart of the master process in the host operating system and running it as a daemon process. The supervisor also maintains an audit trail of the system, which can be retrieved via an HTTP endpoint for inspection. Since the system is single threaded we make sure that the queue and resource pool data structures are maintained in a single immutable store and updated only via invocation of events.

   To determine the health of the instances, when a task fails, the supervisor runs a health check on the instance on which the task failed. The failure of the health check leads to immediate termination of the instance.

C. System Policies

In the system, we have implemented one scheduling policy and one allocation policy illustrated below.

1) *Scheduling*: The scheduler implements the first come, first serve (FCFS) policy. It picks the task with the earliest arrival time and maximum retry count for execution on the next available worker instance.
2) *Provisioning*: We implement ON-DEMAND SINGLE VM (OD-S) policy [10], which allocates one VM per job and releases the VM after job completion plus a wait time (10 seconds in our case). The decision on provisioning of new instances is taken when the pending task queue length goes above the given threshold.

   The configurable parameters of the OD-S policy are,

   - *min-vms*: The minimum number of VMs the system must have at any point.
   - *max-vms*: The total number of VMs the system can provision at most.
   - *threshold*: The pending queue length at which the provisioner is allowed to start a new instance.
3) *Progress Condition*: The system however ensures that a minimum of 1 instance is running in case there are tasks in the pending queue irrespective of the threshold policy.
4) *Termination*: When the pending queue length falls below the threshold, the scheduler marks instances to be terminated and the provisioner executes the termination after the wait period.

[2]This is possible in AWS because EC2 allocates a private IP address to the instance as soon as the request is made. In case this is not possible with the cloud provider, we additionally wait for the IP address to be allocated before beginning health checks.

TABLE I
SYSTEM PARAMETERS AND VALUES IN IMPLEMENTATION.

MAX_RETRIES	5
MIN_VMS	*
MAX_VMS	*
THRESHOLD	*
MAX_RETRIES	5
SCHEDULER_INTERVAL	5s
TASK_TIMEOUT	25s
PROVISIONER_INTERVAL	5s
HEALTH_CHECK_INTERVAL	1s
INSTANCE_READY_TIMEOUT	60s
TERMINATION_WAIT_TIME	10s

The components of the system intercept events and lookup the central immutable store in order to enforce the policies. This flexibility enables us to implement various other policies by consuming the entire state of the system at any point in time. The system parameters are summarized in Table I, * indicates that we have varied the value according to experiments in the following section.

IV. EXPERIMENTAL RESULTS

We conduct a total of 9 experiments and analyse the results in the following sub-sections.

A. Setup

The goal of the experiments is to analyse the tradeoffs between WantCloud BV's current system (3 dedicated servers) versus the proposed IaaS cloud based solution by estimating the workload processing time, responsiveness and the cost. The key components of the system under test are the scheduler and provisioner, affected by the factors, task arrival time and threshold for scaling.

The test system is deployed on the AWS cloud platform using bare bone EC2 compute instances. We use the m3.medium type (vcpus=1, memory=3.75Gb based on Intel Xeon E5-2670 v2) for both the master and the worker. We restricted the deployment to a single region, North Virginia (US-East-1) with default availability zone placement policy and the default virtual private cloud settings provided by AWS. A custom security group is used for exposing ports 8000 (master), 3000 (worker) and 3001 (worker health check) for inbound TCP (HTTP) traffic, the worker nodes however do not have public IP address and are reachable only by the master via HTTP.

The workload consists of 100 images from the Landsat satellite in TIFF format and each task consists of scaling the image size down to a factor of 0.25, rotation by 90 degrees clockwise and finally conversion to TIFF format. We have three sets of 100 requests by varying the inter-arrival time through Poisson distribution with mean intervals of 2.5, 5 and 7.5 seconds labelled as W(2.5), W(5) and W(7.5). The system has been written in Typescript [11], targeting NodeJS version 8 using ReactiveX extensions [12]. The express [13] framework has been used for the web server and ElasticSearch [14] has been used for analysing time series results obtained from the audit log report.

B. Experiments

We run the workload on three variations of system policies by adjusting the parameters, firstly C3 – constantly 3 VMs, which represents the current setup of WantCloud BV, secondly E0 – elastic 0-15 VMs with threshold set to 0 and thirdly, E7 – elastic 0-15 VMs with threshold set to 7. The factors target the load balancing and provisioning policy of the system. In the following sections we present the analysis.

1) Overheads: The average VM allocation overhead by AWS was found to be 42 seconds and we did not find any significant network overheads. However, to prevent any disruptions due to network, we placed a workload driver instance in the same availability zone subnet as of the master node and ran the experiments.

2) Processing time: The total time the system took to process the entire workload, presented in Figure 3, shows that E0 takes the least amount of time (272 seconds) for W(2.5). However, E0 performs the worst as the mean arrival time starts to increase. This can be directly attributed to the fact that the policy is very aggressive in releasing VMs and thus a lot of thrashing is experienced.

3) Responsiveness: The metric for responsiveness is derived from the average waiting time of the tasks in the pending queue until they are scheduled for execution, presented in Figure 4. Contrary to our expectations, we see that C3 has the least average wait time of 7.5 seconds at 595.6 seconds for W(7.5). We also observe that E0 maintains a consistent wait time for both W(5) and W(7.5). However, on closer inspection, we found that the waiting time average has been heavily skewed by a few outliers affected by the provisioner. We focus on the results of W(2.5) in Figure 5, where majority of the tasks for E0 and E7 lie below 40 seconds. This large waiting time occurs for a few tasks when a free VM is not found and the provisioner is still starting a new instance, causing all the tasks in the queue to starve. This happens when the system starts the first instance (correlates to ≈40 seconds instance startup overhead) and also in one another case when all the VMs are shut down to 0.

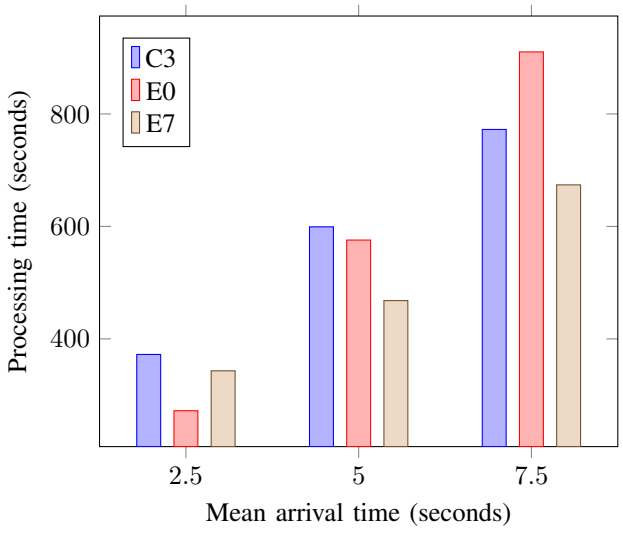

Fig. 3. Workload processing time across three mean arrival times.

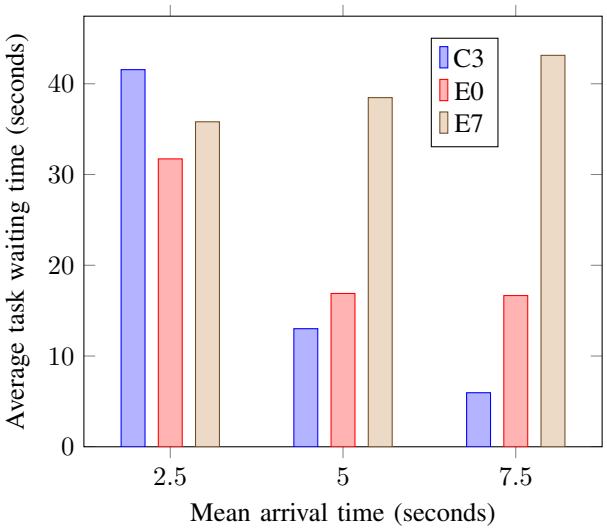

Fig. 4. Cumulative task waiting time for workload across arrival times

We also found task retries to be playing a role here. The failed task enters the pending queue with highest priority and also the instance on which the task failed is now marked for health check, effectively taking away one worker from the pool for a few seconds. This specific case occurred in E7 when there was only 1 instance running, a task failed and there were 5 other tasks (7 is threshold) in the pending queue.

4) Utilization: We analyse the utilization of the system presented in Figure 6 as the ratio of total VM time spent in task execution to its total lifetime. The maximum utilization we see even for C3 where there is no provisioning overhead is about 80%. This highlights the drawback in the periodic invocation of the scheduler and provisioner. The VMs have to wait for the next scheduling/provisioning cycle until which they are going to be idle. Another cause is the scheduled termination time where the VM waits for 10 seconds before getting terminated. If a task arrives at its 8th second of wait time, the task will miss the VM anyway.

This chart once again highlights the thrashing problem associated with E0, it is simply keeping around idle VMs due to termination wait time. However, we see a balanced utilization for E7, it gains from not letting the VM go away immediately as the allocation overhead is much larger and preventing VMs from having a short lifetime.

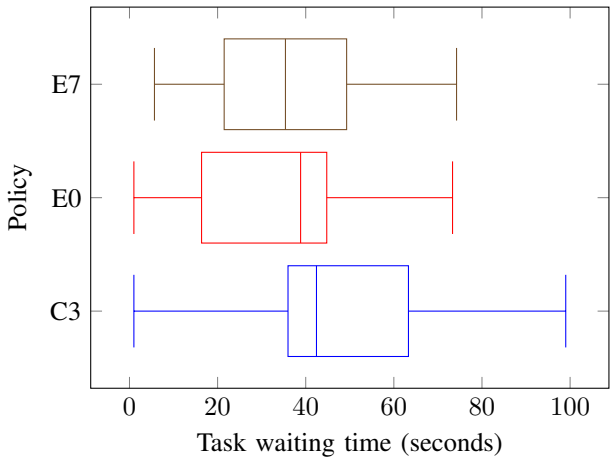

Fig. 5. Task waiting time distribution for W(2.5)

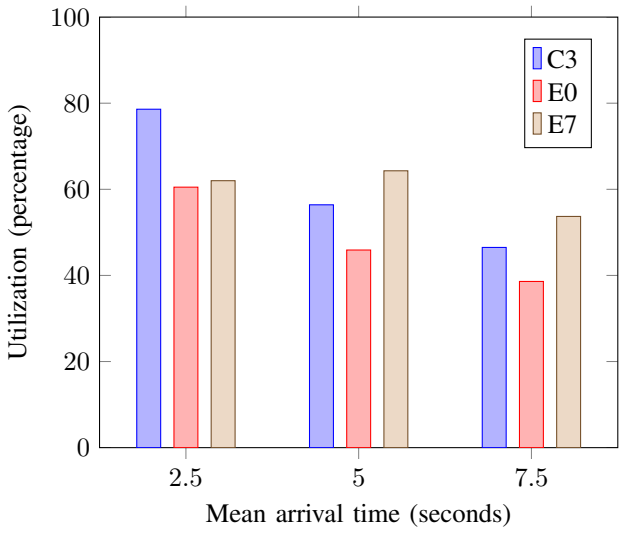

Fig. 6. Utilization of VMs for workload across arrival times

5) Costs: Each experiment (except C3 with constantly 3 workers) starts and stops several VMs over its processing time and our total instance run time is under 1 hour for each experiment. Therefore, the cost is computed in terms of hourly charges for unique instances launched, tabulated in Table II for running m3.medium instance[3]. Apart from the instance cost, there are minor acceptable costs involving the ephemeral disk storage, data transfer and AMI snapshots.

TABLE II
CHARGED COST PER HOUR ACROSS ARRIVAL TIMES

Policy	2.5s	5s	7.5s
C3	$0.27 (4)	$0.27 (4)	$0.27 (4)
E0	$1.88 (28)	$3.89 (58)	$6.16 (92)
E7	$2.01 (30)	$2.08 (31)	$3.82 (57)

[3]At the time of writing this report however, AWS offers per second billing which would greatly reduce the cost, which our policies were designed for.

C. Potential Improvements

The experiments and analysis suggest that the E7 policy is all round consistent with all arrival times. We would like to improve in the following areas.

1) Schedule more than one task in a worker, considering that 50% of the time the worker is either downloading or uploading data.
2) Scale up instances in multiples of 2 or geometrically.
3) Run a worker process in the master node itself. This can further increase the responsiveness and save cost.
4) Implement shortest job scheduling policy since the tasks can be roughly weighted with the input image size.

Clearly there are many other drawbacks in our implementation which we wish to overcome. However the experiments gave deep insights into the various bottlenecks and optimization points in the system.

V. CONCLUSION

The speedups shown in Table III indicate that a maximum speedup of upto 1.38 times was achieve by our system. The chief requirement of WantCloud BV, is to maintain responsiveness at peak traffic demands, which is clearly met by the system, illustrated in Figure 5, where a majority of the traffic is scheduled for execution within 40 seconds, W(2.5). The incidental costs of cloud when such a traffic is not present should also be noted though. However, the elastic policies can scale the system down to 0 workers, saving cost when there is no traffic. We found that in some cases it costs more than 10 times to finish the same amount of work with only a marginal speedup, indicating that policies must be chosen carefully. After analysing the tradeoffs, we think that it is cost-worthy for WantCloud BV to move their application to cloud.

TABLE III
SPEEDUP

Arr. Time	C3 Speedup	E0 Speedup	E7 Speedup
W(2.5)	1.0	1.38	1.09
W(5.0)	1.0	1.04	1.28
W(7.5)	1.0	0.84	1.14

REFERENCES

[1] Amazon Web Services (AWS). [Online]. Available: https://aws.amazon.com/
[2] ImageMagick. [Online]. Available: http://www.imagemagick.org/
[3] NodeJS. [Online]. Available: http://www.nodejs.org/
[4] G. Parsons and J. Rafferty, "Tag image file format (tiff)," Internet Requests for Comments, RFC 3302, September 2002. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3302.txt
[5] G. K. Wallace, "The jpeg still picture compression standard," *Communications of the ACM*, pp. 30–44, 1991.
[6] Aws batch processing. [Online]. Available: https://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_batch_03.pdf
[7] Landsat on AWS. [Online]. Available: https://aws.amazon.com/public-datasets/landsat/
[8] R. Jain, *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[9] T. Bray, "The javascript object notation (json) data interchange format," Internet Requests for Comments, RFC 7159, March 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7159.txt
[10] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup, "An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 612–619.
[11] Typescript programming language. [Online]. Available: https://www.typescriptlang.org/
[12] Reactive extensions. [Online]. Available: http://reactivex.io/
[13] Express framework. [Online]. Available: https://expressjs.com/
[14] Elasticsearch. [Online]. Available: https://www.elastic.co/products/elasticsearch

APPENDIX A: TIME SHEETS

TABLE IV
TIME SPENT ON THE PROJECT PER ACTIVITY.

Activity	Time (hours)
Total	170
Think	20
Dev	80
XP	15
Analysis	15
Write	20
Wasted	20

TABLE V
TIME SPENT PER EXPERIMENT.

	Time (hours)		
	Dev	Setup	Total
C3	8	4	12
E0	1	0.5	1.5
E7	1	0.5	1.5