Uncovering the Perfect Place

Optimising Workflow Engine Deployment in the Cloud

Michael Luckeneder

April 5, 2013
Abstract

When orchestrating highly distributed and data-intensive Web service workflows the geographical placement of the orchestration engine can greatly affect the overall performance of a workflow. Orchestration engines are typically run from within an organisation’s network; when executing data-intensive workflows that consist of highly distributed services, the orchestration engine may have to transfer data across long geographical distances which increases execution time and degrades the overall performance of a workflow.

This project presents a Web service framework and analysis tool which, given a workflow specification, computes the optimal Amazon EC2 Cloud region to automatically deploy the orchestration engine and execute the workflow. Using a simplified workflow specification model, the analysis tool uses geographical distance of the workflow, network latency and HTTP round-trip time between Amazon Cloud regions and the workflow nodes to find a ranking of Cloud regions. This overall ranking predicts where the workflow orchestration engine should be deployed in order to reduce overall execution time.

The approach is evaluated by executing randomly generated data-intensive workflows deployed on the PlanetLab platform in order to rank Amazon EC2 Cloud regions. The experimental results show that the proposed optimisation strategy can speed up execution time on average by 82% compared to local execution. The results also show that the standard deviation of execution time is reduced by an average of almost 65% using the optimisation strategy.
Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 7,385 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Michael Luckeneder
Contents

1 Introduction .................................................. 9

2 Objectives .................................................... 11
   2.1 Hypothesis .............................................. 11
   2.2 Primary Objectives ..................................... 11
   2.3 Secondary Objectives .................................... 12
   2.4 Tertiary Objectives ..................................... 12

3 Context Survey ................................................. 13
   3.1 Decentralised Orchestration ......................... 13
   3.2 Peer-to-peer Orchestration ......................... 14
   3.3 Data-Aware and Location-Aware Scheduling .... 14
   3.4 Evaluation ............................................. 14

4 Requirements Specification ................................. 15
   4.1 User Requirements ..................................... 15
      4.1.1 Workflow Specification ......................... 15
      4.1.2 Pre-Deployment Analysis Tool ............... 15
      4.1.3 Workflow Engine ................................ 15
      4.1.4 Experimentation Framework ................. 16
   4.2 Functional System Requirements ................... 16
      4.2.1 Workflow Specification ......................... 16
      4.2.2 Pre-Deployment Analysis Tool ............... 16
      4.2.3 Workflow Engine ................................ 16
      4.2.4 Experimentation Framework ................. 17
   4.3 Non-Functional System Requirements .............. 17

5 Software Engineering Process ............................ 18
   5.1 Engineering Process ................................... 18
   5.2 Development Methods and Tools .................... 19
      5.2.1 Prototyping ...................................... 19
9.4.2 Network Latency ................................................. 39
9.4.3 HTTP Round-Trip Time ........................................ 39
9.4.4 Overall Ranking .................................................. 39

9.5 Feasibility of Analysis ............................................. 40

10 Project Evaluation and Critical Appraisal .......................... 41
10.1 Objectives ................................................................ 41
10.2 Requirements .......................................................... 41
10.3 Challenges ................................................................ 41

11 Conclusion .................................................................. 43
11.1 Key Achievements ..................................................... 43
11.2 Drawbacks ............................................................... 43
11.3 Future Work ............................................................. 44

A Test Workflow Graphs .................................................. 48

B Paper ........................................................................... 52

C Documentation ............................................................. 61
List of Tables

8.1 Analysis tool result output ........................................... 33
8.2 Workflow execution results ........................................... 35
9.1 Correct predictions ................................................... 39
# List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>High-level overview of the software components</td>
<td>24</td>
</tr>
<tr>
<td>7.2</td>
<td>Workflow example 1</td>
<td>24</td>
</tr>
<tr>
<td>7.3</td>
<td>Abstract candidate data flow graph for Figure 7.2</td>
<td>25</td>
</tr>
<tr>
<td>7.4</td>
<td>Workflow example 2</td>
<td>25</td>
</tr>
<tr>
<td>7.5</td>
<td>Sample Web service node</td>
<td>26</td>
</tr>
<tr>
<td>8.1</td>
<td>AbstractWorkflow</td>
<td>28</td>
</tr>
<tr>
<td>8.2</td>
<td>Internal DAG representation</td>
<td>29</td>
</tr>
<tr>
<td>8.3</td>
<td>AbstractRegion</td>
<td>30</td>
</tr>
<tr>
<td>8.4</td>
<td>Example workflow</td>
<td>31</td>
</tr>
<tr>
<td>8.5</td>
<td>Example workflow with closest Amazon EC2 regions</td>
<td>32</td>
</tr>
<tr>
<td>8.6</td>
<td>Candidate workflow graph using EC2 region us-east-1 with network latency</td>
<td>32</td>
</tr>
<tr>
<td>9.1</td>
<td>Execution times of the sample workflows</td>
<td>38</td>
</tr>
<tr>
<td>A.1</td>
<td>Workflow A (predicted &quot;optimal&quot; EC2 region: us-east-1)</td>
<td>49</td>
</tr>
<tr>
<td>A.2</td>
<td>Workflow B (predicted &quot;optimal&quot; EC2 region: us-east-1)</td>
<td>49</td>
</tr>
<tr>
<td>A.3</td>
<td>Workflow C (predicted &quot;optimal&quot; EC2 region: eu-west-1)</td>
<td>49</td>
</tr>
<tr>
<td>A.4</td>
<td>Workflow D (predicted &quot;optimal&quot; EC2 region: us-east-1)</td>
<td>49</td>
</tr>
<tr>
<td>A.5</td>
<td>Workflow E (predicted &quot;optimal&quot; EC2 region: us-east-1)</td>
<td>49</td>
</tr>
<tr>
<td>A.6</td>
<td>Workflow F (predicted &quot;optimal&quot; EC2 region: eu-west-1)</td>
<td>50</td>
</tr>
<tr>
<td>A.7</td>
<td>Workflow G (predicted &quot;optimal&quot; EC2 region: eu-west-1)</td>
<td>50</td>
</tr>
<tr>
<td>A.8</td>
<td>Workflow H (predicted &quot;optimal&quot; EC2 region: us-west-2)</td>
<td>50</td>
</tr>
<tr>
<td>A.9</td>
<td>Workflow I (predicted &quot;optimal&quot; EC2 region: us-east-1)</td>
<td>51</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Scientific workflows [5] are typically orchestrated using a workflow engine, such as Taverna [30], running locally within an organisation’s network. However, if the Web services in the workflow are data-intensive and spread across many geographical regions, the data might have to move long distances in order to flow from the data sources to the Web services via the orchestrator. This in turn most likely slows down the execution of the workflow and degrades the overall performance.

A possible solution to this problem could be to “move” the workflow orchestrator closer to the data and the Web services. Cloud environments provide a cost-effective platform for scientists and engineers to execute their workflows in a remote data centre as demonstrated by recent research [20] [18] [28] [13] [34]. Using an Infrastructure as a Service (IaaS) Cloud, such as an Amazon EC2 instance\(^1\), it is possible to automatically deploy the orchestrator into a suitable EC2 region that is “closer” to the data source and Web service nodes; in turn data would not have to travel as far and therefore execution times could be reduced.

The interesting problem arises when workflows consist of a large number of Web service nodes - all of them in different geographical regions. In these cases it is very challenging to judge where the closest and thus best-performing Cloud region might be. Furthermore, a certain Cloud region might be geographically closer to the Web service nodes but, due to a high network latency on a certain network link, using another Cloud region would actually result in a lower execution time.

In this project I design, implement and evaluate a pre-deployment analysis tool which can dynamically compute the “optimal” Amazon EC2 Cloud region to deploy the orchestrator, given a specific workflow consisting of multiple distributed Web services. The research further discusses three factors which could potentially affect the suitability of choosing a certain Cloud region: total geographical distance of workflow, network latency and HTTP round-trip time.

\(^{1}\text{Amazon EC2 instances are essentially virtual machines in the Cloud}\)
This project successfully develops and discusses methods to:

- simultaneously manage Amazon EC2 instances in multiple Amazon AWS regions.
- automatically deploy workflows in an optimal Amazon AWS region given the topology of the workflow and a number of fixed resource locations.
- automatically evaluate and compare the performance of different workflow orchestration approaches.
- evaluate different factors which influence workflow execution time.

The remainder of this report is structured as follows: Chapter 2 outlines the objectives established at the beginning of the project. Chapter 3 discusses related work and surveys the context of the problem. Chapters 4 and 5 describe the software engineering approach taken as well as the formal requirements specification for the software. Chapter 7 outlines the abstract architecture of an analysis tool, a suitable workflow specification model and the experimental setup. Chapter 8 proceeds to explain the implementation of the testing framework using PlanetLab, a global research network, and Amazon EC2 to evaluate the approach and the implemented analysis tool. Finally, Chapter 9 discusses the results of the evaluation and how the analysis tool significantly reduced execution times. Chapters 10 and 11 critically evaluate the project progress and results and outline opportunities for future work.
Chapter 2

Objectives

This chapter outlines the objectives that were developed with the project supervisor. Since a rather broad research hypothesis was the core of the project, the objectives were altered during the course of the semester to deal with changes of research direction. Thus, I present the final iteration of the objectives here.

2.1 Hypothesis

When the computation is moved closer to the data the application can move data with lower latency, the amount of data needed to be moved can be reduced and the overall performance of the workflow can be increased.

2.2 Primary Objectives

1. Identify a suitable workflow engine and specification language, such as Taverna, or develop a custom, graph-based workflow specification language and workflow engine suitable for sequential workflows.

2. Develop a pre-deployment analysis tool which takes a Web service workflow specification, analyses it and evaluates the optimal AWS region to deploy the workflow engine.

3. Implement a monitoring tool that can automatically be deployed on instances in all AWS regions by the analysis tool, measures latency, load of the instance and can easily be queried.

4. Create and run a set of experiments to verify the functionality of this approach compared to the traditional approach of orchestrating the workflow locally.
2.3 Secondary Objectives

1. Build a testing infrastructure for the experiments using PlanetLab and other Web servers to avoid dependency on externally provided and maintained Web services.

2. Implement a tool to automatically generate random test workflows (both sequential and non-sequential) for use in the experiments.

2.4 Tertiary Objectives

1. Extend the pre-deployment analysis tool, workflow engine and specification language to support more complicated, non-sequential workflows (e.g. multiple sources, fan-in and fan-out patterns) and then evaluate this further using more, randomly generated experiments.

2. Extend the workflow engine to automatically monitor the load/latency of each region and then dynamically migrate the workflow engine when conditions change.
Chapter 3

Context Survey

This project addresses the problem of where geographically to deploy a workflow engine, given the specification of a workflow consisting of highly distributed services and a set of fixed points - in this case Amazon EC2 Cloud regions. There is a lot of research which covers related topics such as decentralised orchestration, peer-to-peer orchestration and data-aware scheduling. However, as far as it was researched in the course of this project, there has been no prior work on evaluating a workflow specification before deployment and then using Cloud-based resources to improve the performance of the workflow execution.

3.1 Decentralised Orchestration

Pegasus [12], for example, is a framework that can map an abstract workflow description onto a Grid, automatically provision the required resources and thus reduce execution time.

A similar, related architecture for the decentralised execution of workflows are Service Invocation Triggers [8]. For this approach to work however, the input workflow specification has to be deconstructed into smaller, sequential fragments which are not permitted to contain loops. Given these fragments, they can be installed at a trigger and thus the data dependencies are not handled centrally anymore.

A further way to facilitate decentralised orchestration of workflows is the concept of pointers in service-oriented architectures [35]. These pointers are the same concepts as in a programming language context, i.e. data can be passed by reference rather than by value. When this technique is used by Web service workflows, the advantage is that the orchestrator does not handle the data passing between Web services; this responsibility now lies with every Web service node.

All the approaches in this section require the workflow specification to be altered in order to make use of decentralised orchestration - be it adding triggers, proxies or pointers.


3.2 Peer-to-peer Orchestration

Other papers also address the performance bottlenecks experienced when centralised servers are used to orchestrate workflows. *Circulate* [7] ⁹, a hybrid, proxy-based architecture is proposed to speed up data-intensive workflows. In particular, a peer-to-peer architecture is outlined, where a centralised orchestrator manages the control flow, but the data directly flows between Web service node. [27] proposes a similar decentralised workflow management system which makes heavy use of peer-to-peer data flow.

Another interesting approach [15] discusses a peer-to-peer workflow management system which is comparable to Gnutella. ¹ The system allows peers to register with the system and offer their resources to other peers. These peers can subsequently be discovered through a directory by other peers and used to deploy workflows. Thus, the system grows more powerful when more peers sign up to use it.

_Triana_ [31] is an open-source problem solving environment based on peer-to-peer architecture. It is designed to define, process, analyse, manage, execute and monitor workflows. Triana can distribute sections of a workflow to remote clients using a peer-to-peer network.

3.3 Data-Aware and Location-Aware Scheduling

Recently, Amazon AWS added *Latency-Based Routing (LBR)* [2] to Route 53, managed DNS service. LBR functionality can be used to reduce latency by serving requests from the region with the lowest network latency, an approach that is frequently used by content distribution networks (CDNs). However, LBR is fairly simplistic and cannot feasibly be used to deploy an application that is constructed from a number of highly distributed, Web services.

_Stork_ proposes an approach for *data-aware scheduling* [21]. It is a data placement scheduler for Grids which, given an application, dynamically decides where to deploy the data to minimise the execution time.

3.4 Evaluation

Overall, there are various approaches to increase the efficiency of data-intensive workflows. However, most of these require the workflow specification or the services to be altered before being able to take advantage of optimisation. In contrast, the proposed approach in this project is to statically analyse the original workflow before deployment and then to deploy the workflow orchestrator to the optimal geographical Cloud region.

---

¹ a decentralised P2P filesharing network
Chapter 4

Requirements Specification

This chapter shows the requirements specification document for the developed software. Similar to the objectives outlined in Chapter 2, the requirements specification went through a couple of iterations due to the open research nature of the project.

4.1 User Requirements

The system must provide users the following functionality:

4.1.1 Workflow Specification

1. Users can specify the topology of workflows in plain-text files
2. Users can implement the workflow in a popular programming language

4.1.2 Pre-Deployment Analysis Tool

1. Users can easily configure the AWS credentials
2. Users can easily run the analysis tool with a simple command
3. Users can view the results of the analysis
   - Scores for all factors in all AWS regions
   - A consolidated score for each region which identifies the best region for engine deployment
   - A ranking of AWS regions

4.1.3 Workflow Engine

1. Users can run the workflow in specific AWS regions with a single command
2. Users can retrieve the run time for every AWS region the workflow was executed in
4.1.4 Experimentation Framework

1. Users can easily deploy the Web services to PlanetLab slices
2. Users can easily run and terminate these Web services
3. Users can arbitrarily chain these Web services to build different workflows

4.2 Functional System Requirements

In order to fulfill the outlined user requirements, the system must do the following:

4.2.1 Workflow Specification

1. The workflow will be specified as a Python class
2. An abstract Python class will outline the workflow specification requirements
3. A comma-separated values (CSV) file will be used to specify the workflow topology
4. The workflow process will be implemented in a plain Python file

4.2.2 Pre-Deployment Analysis Tool

1. The tool will be implemented in Python
2. All API credentials will be specified in a Python module
3. Users will be able to run the system by running simple terminal commands
4. The tool will transparently launch EC2 instances, deploy and run the monitoring tools and then gather the results
5. The toll will compute the data center ranking based on the different metrics and the consolidated score
6. The tool will output the results into a plain-text file

4.2.3 Workflow Engine

1. The engine will be implemented in Python
2. Users will be able to run the system by running simple terminal commands where the EC2 regions can be specified
3. The execution tool will transparently launch EC2 instances in the specific instances, deploy the workflow engine, execute the workflows and retrieve the execution times in the specified regions
4. The tool will output the results into a plain-text file

**4.2.4 Experimentation Framework**

1. The framework will be implemented in Python and PHP (depending on the target platform)

2. The PlanetLab experimentation Web services can be deployed, started and stopped using simple terminal commands

3. The Web services will expose a simple REST API which takes an image using POST and returns it in the HTTP response

4. The Python and PHP versions will provide the same functionality

**4.3 Non-Functional System Requirements**

1. All tools can simultaneously launch and use EC2 instances in multiple regions

2. All tools will terminate any running EC2 instances after they have successfully or unsuccessfully terminated

3. All tools will display the logs written to **STDERR**

4. All tools will make use of the `boto` library to communicate with Amazon AWS

5. All tools must be able to execute on different platforms without changes to the code

6. The code will be well documented and hosted on Github
Chapter 5

Software Engineering Process

This chapter outlines the development approach taken for this project and justifications for the chosen software engineering methodologies and tools.

5.1 Engineering Process

Discovering and formalising the requirements are an essential part of building reliable software. However, due to the broad, open research nature of the project, it was hard to pin down a list of requirements at the beginning of the year. Requirements were defined and revised as the project moved along. Overall, the requirements engineering process followed a weekly pattern such as this:

1. **Talk with supervisor**: talk about new features and specify a minimal set of requirements to be implemented until the next meeting

2. **Implement features**: build software to fulfill these minimal requirements

3. **Test feasibility**: run experiment and verify if the new features or approach improve the functionality of the software

4. **Refactor code**:
   - (a) If the iteration was useful: refactor new code, integrate better with existing system and debug
   - (b) If the iteration was not useful or infeasible: discard new functionality

5. **Process restarts at stage 1**.

Thus requirements were established on a rolling basis which provided the necessary flexibility for implementing experimental software.
5.2 Development Methods and Tools

5.2.1 Prototyping

Since the software tools for this project were developed to evaluate a hypothesis rather than deliver a production-ready piece of software, prototyping \cite{29} p. 45 or rapid/throwaway prototyping \cite{22} was identified as the ideal software engineering methodology.

This methodology puts an emphasis on quickly implementing features, testing them and then evaluating their impact on the software system and the research question and hypothesis. Based on that, the features can either be identified as redundant or the feature can give rise to new necessary requirements. Throwaway prototyping further implies that the prototype will not form part of the final solution but is merely a proof of concept and a means to identify successes, short-comings and propose new features. This was especially important in the initial stages of the project when many prototypes were built to test out various aspects of the e.g. Amazon API. The actual code of these prototypes was later discarded and the concepts were incorporated in different ways into the final version.

This approach was very feasible since “perfect” reliability and software correctness were not vital and errors could be easily fixed and debugged during execution of experiments.

5.2.2 Revision Control

As with most projects with a fairly extensive codebase, revision control was used to make the development workflow smoother, manage backups of the codebase and efficiently maintain distinct versions of the project. Specifically, I decided to use the Git revision control system \cite{16} due to the decentralised architecture, the ease of use and the popularity and widespread use in industry. Git was installed locally and set up to use the St Andrews server as a remote host via SSH to keep a remote backup of the codebase.

In order to use Git effectively, a simplified version of the git flow branching model \cite{14}, which enjoys popularity in the Git community, was used. Essentially, the source repository had two branches, develop and master. For every new feature under development, a feature branch was branched from develop. Once the feature was done, the feature branch was merged back into develop. When a number of features were completed (usually every week before the supervisor meeting), the develop branch was merged into the master branch. These merges constitute a new version of the software.

The branching model made it easy to work on different features separately and to maintain a working version of the code at all times.
5.2.3 Coding Environment

The core programming language was Python 2.7.3 [25] due to the broad availability of online documentation, free packages (including boto) and the ability to write concise and powerful code. All of this made the rapid prototyping approach very efficient.

To be able to maintain a self-contained environment for the packages and the Python installation on the development system, virtualenv [1] was used. virtualenv installs all packages in a single directory in the codebase root. This greatly simplified running the software on different systems since dependencies could be easily moved along with the code.

5.2.4 Coding Standards

Python is an object-oriented language which made the decision easy to build the project in a very modular, object-oriented manner. This helps simplify the implementation of new workflows, new analysis techniques, etc.

In order to maintain legible and standards-compliant and easily readable code, most of the PEP8 [33] conventions were followed. The document outlines, among other things, correct indentations, naming and whitespace conventions.

Pylint [24] is a tool that can analyse Python code and find violations of the PEP8 conventions. The tool was used frequently in the development process to maintain high coding standards.

5.3 Code Maintainability

In order to maintain readable and easily (re-)usable code, the classes and algorithms were well commented. Furthermore, Python makes it easy to annotate modules, classes and methods with docstrings, similar to JavaDoc [19]. The usage conventions are proposed in PEP257 [17] are were largely followed in the software for this project.

The code is currently hosted in a public repository on Github 1 - a social coding platform which uses Git. Along with the code there is a comprehensive usage document for all components of the software system. A copy of the documentation is included in Appendix C.

5.4 Testing

The most important testing strategy used for this project was the experimental validation of the approach and software which is explained in detail in Chapter 9.

A test-driven approach to development was challenging due to the many different components of the software and the number of different external APIs. Therefore context-driven testing [3] was used for developing the software. Context-driven testing is an approach

1 https://github.com/mluckeneder/movingdata
which implies that there are no best practices for testing and that the testing approach depends on the context of the code. Most components of the software, require some sort of external dependency, such as a remote SSH server or an API, for every method call. Especially the Amazon AWS APIs did not facilitate pre-specified unit testing. There are approaches to overcome these issues, such as by using the mock \cite{23} package to mock the AWS services. This, however, was infeasible as it would have greatly increased the complexity of the software and would have made it too inflexible to be used in an open research project. In the end, the exploratory testing \cite{4} approach was used for the bulk of the code. Exploratory testing is a context-driven approach which can be defined as “simultaneous learning, test design, and test execution”. Essentially, tests are created and run on an ad-hoc basis using the context of the program and some creativity.

Pre-defined unit tests were used for classes without external API dependencies - for example the graph implementation. The unit-testing framework testify \cite{32} was used since it provides some features that Python’s built-in unit-testing framework lacks, such as the use of method decorators for more concise code.
Chapter 6

Ethics

The project focuses on designing and testing a distributed systems application. Usage and storage of any personal data is not required. Therefore, there are no ethical concerns.
Chapter 7

Design

7.1 High-level System Overview

The software system developed in this project can be seen as several distinct components, which are depicted in Figure 7.1. The workflow specification and pre-deployment analysis tool together implement a system which models the hypothesis stated earlier. The Web service nodes, random workflow generator and workflow orchestrator components are an experimental framework to experimentally evaluate the pre-deployment analysis system.

7.2 Workflow Specification

In order to correctly analyse the workflow and find the optimized Cloud region to deploy the orchestration engine to, a workflow specification model is required. This should specify data sources, intermediate processing steps as well as a data sink. Since scientific workflows are inherently graph-based, the specification language should ideally also be graph-based so that it can easily be interpreted by the analysis tool as well as a workflow orchestrator. Since most Web services make use of HTTP, I defined endpoints as being RESTful APIs. That means that the Web service input data is encoded in the URL as well as the HTTP request while the final result is returned in the HTTP response.

The workflow specification should also make it computationally simple to retrieve an ordered set of all distinct workflow nodes. This is an important feature required by the analysis engine.

Figure 7.2 shows an example of a simple, sequential workflow. Figure 7.4 shows an example of a more complex workflow with multiple data source and fan-in and fan-out patterns.
7.3 Pre-Deployment Analysis Tool

The pre-deployment analysis tool represents the core of the software. When executed, the tool goes through three stages: pre-analysis, metric gathering and analysis.

7.3.1 Pre-Analysis

In this stage the tool takes a workflow specification and builds several candidate workflow graphs which represent the data flow when the orchestrator is run in a specific Cloud region. Thus, the tool requires a list of Amazon EC2 regions that should be considered for workflow execution. Every candidate graph will be based on a different one of these Cloud regions and represent a possible optimised workflow using the Cloud instance as the workflow orchestrator. Every edge represents the data flowing from a Web service node to the orchestrator in the Cloud region or vice versa. Figure 7.3 illustrates a candidate graph built for the workflow described in Figure 7.2. The graph is in the form of a DAG (Directed Acyclic Graph).
7.3.2 Metric Gathering

In this stage, the tool launches an EC2 instance in every AWS region. Then metric commands are run on these instances in order to populate the edge weights of the candidate graphs. A metric value is required for every edge of the every candidate graph.

Potential candidates for useful metrics include geographical distance, network latency (as measured by the UNIX `ping` command) and HTTP round-trip time (as measured by the UNIX `curl` command) between every pair of adjacent nodes.

7.3.3 Analysis

Based on every candidate graph and metric used, the analysis tool computes an overall score. The final output is a separate table for every metric which ranks the Cloud regions by their predicted execution times. These tables are purely ordinal and no actual execution times are predicted. Using these metrics there are numerous possible approaches to combine them and compute an overall score. Based on the experiments run for this project (see Chapter 9), it was found that the average of the sum of pings and sum of RTTs works very
Finally, the Cloud region with the lowest overall score should be the best region to deploy the workflow orchestrator in order to minimise workflow execution time.

### 7.4 Experimental Setup

In order to verify the hypothesis and evaluate the pre-deployment analysis approach, various experiments were conducted. This section outlines the design of an experimental framework to verify the pre-deployment analysis tool.

#### 7.4.1 Web Service Nodes

As mentioned previously, in the context of this project the Web service nodes are considered to be simple RESTful APIs which should be arbitrarily chainable to form new Web service workflows.

In order to achieve this, a good approach is to implement a generic image processing Web service. The Web service is invoked sending a POST request to a URL with the image transmitted in the HTTP request body. The Web service then applies some time-intensive image transformation (e.g. rotation) and then returns the image in the HTTP response. Since the node’s input and output are a single JPEG image, the nodes can clearly be chained arbitrarily for a diverse set of experiments. Figure 7.5 graphically illustrates such a Web service node.

#### 7.4.2 Random Workflow Specification

In order to facilitate the use of many different workflows, the workflow specification model defined earlier has to be extended slightly. If many experimental workflows are run, it is infeasible to create a separate class for every single workflow. Therefore, the approach here is to define a generic experimental workflow which simply takes an image from a pre-specified data source, reads the exact workflow nodes from a plain-text file, puts the image through the specified workflow pipeline and then returns the total execution time. In order
to counteract adverse effects of small file sizes, such as connection setup overheads, the workflow executes a predetermined number of times.

Using the workflow specification model and Web service nodes outlined above, it is easy to construct a simple tool that automatically generates random workflows given a list of available Web service nodes. The generation tool takes a list of workflow nodes as an input and then, using random selection with replacement, constructs a sequential workflow of a specified length and writes the result to a plain-text file.

7.4.3 Workflow Orchestration

Finally, a tool is required to deploy and orchestrate the workflow to a specific Amazon EC2 region. Similar to the pre-deployment analysis tool, this tool is able to launch a number of specified EC2 instances, copy the workflow files via SFTP and then execute the workflow through invoking an SSH command. Finally, after the workflows have run, the tool writes a summary table of the execution times to a file. This table shows the execution time of the workflow in every EC2 region it was executed.
Chapter 8

Implementation

Since the focus of the project was to evaluate the pre-deployment analysis approach, all tools were implemented as CLI Python scripts. Developing and testing a GUI was deemed beyond the scope of the project and research question.

The implementation outlined in this Chapter represents the final version of the software. During the course of the project different implementations were made and used but later discarded due to infeasibility, errors or because the feature was not necessary anymore. Therefore, the final implementation is fairly lean, with all unnecessary features removed and only the essential components remaining.

8.1 Workflow Specification

In the implementation, the workflows are defined using concrete implementations of an abstract Python class rather than a graph-based markup language. This has the advantage that the workflow can be specified using normal Python commands and one is not restricted to using a more rigid, graph-based specification language. The abstract workflow class also ensures that workflows can automatically be deployed and remotely run via SSH in an Amazon EC2 region on instances running Ubuntu. Figure 8.1 shows a UML of the AbstractWorkflow class.

As proposed earlier, the concrete implementation of the workflow class specifies a

![AbstractWorkflow](image)

Figure 8.1: AbstractWorkflow
method that returns an ordered set of all nodes used in the workflow \( get\_endpoints() \).
The bulk of the abstract workflow class deals with providing functionality to be able to run the workflow both locally as well as remotely on an Amazon EC2 Cloud instance via SSH. Every method in the class expects a dependency variable to be injected which allows the workflow to communicate with an instance of an AbstractRegion (see Section 8.2.2). The workflow engine can then e.g. execute SSH commands and transfer files via SFTP to the Amazon EC2 instances in order to set up and run the workflow.

8.2 Pre-Deployment Analysis Tool

The pre-deployment analysis tool heavily uses the boto library to interface with the Amazon EC2 APIs.

8.2.1 Workflow Graph and Candidate Graphs

As stated in Chapter 7 the pre-deployment analysis tool uses a DAG implementation to represent and solve the workflow and candidate graphs. Since no complicated graph calculations are required, the DAG was implemented from scratch. The DAG implementation went through several iterations; the final version uses a modified version of an adjacency list representation \[11\] p. 528 as is depicted in Figure 8.2. This representation simplifies certain tasks, such as summing all the edges and is well suited for highly sequential graphs.

8.2.2 AbstractRegion

The core component of the tool is an abstraction of multiple Amazon EC2 Cloud regions. The abstract class AbstractRegion, depicted in Figure 8.3, specifies the functionality re-
The software includes two concrete implementations of this abstract class: *EC2Region* and *EC2MultiRegionConnection* with the former representing an instance in a single EC2 region and the latter representing instances in multiple EC2 regions.

In practice, this means that when e.g. *EC2MultiRegionConnection* is instantiated, it can transparently launch an instance in every EC2 region just by invoking the `start()` method. Similarly, invoking the `invoke_ssh(cmd)` method will transparently invoke the specified shell command on every instance in every region and return a list of results. This is a very powerful library since hides the complexity of maintaining and using EC2 instances in multiple regions.

### 8.2.3 Analysis

Based on the DAG implementation and the workflow specification, the pre-deployment analysis tool create a series of graphs - one for each Amazon EC2 region and metric (24 graphs\(^1\)). These graphs are concrete implementations of the graph presented in Figure 7.3. In these graphs, the edges correspond to the metrics that have to be gathered.

The tool then makes use of the *EC2MultiRegionConnection* to launch an instance in every EC2 region. For every edge in every candidate graph, the metrics are retrieved by invoking a shell command in all EC2 regions: network latency is obtained using “ping” and HTTP RTT using “curl”. In order to obtain geographical distances between nodes, the *ipinfodb*\(^2\) API and some spherical geometry calculations are used.

The analysis tool then traverses all the generated graphs and sums their edge weights. This will generate the Cloud region ranking tables for all three metrics. Finally, the final score is calculated by averaging the network latency and HTTP RTT metric for every Cloud region (see Equation 7.1). These tables are then displayed and all previously launched EC2 instances are terminated.

---

\(^1\) 3 metrics \(\times\) 8 regions = 24 graphs

\(^2\) [http://ipinfodb.com](http://ipinfodb.com) [22/02/2013]
8.2.4 Workflow Orchestration

The pre-deployment analysis tool and the workflow orchestrator share a lot of the underlying components, such as the EC2 multi region abstraction and the processing of the workflow specification. Therefore, the orchestrator is simply a method in the PreDeployer class.

8.3 Worked Example

This section covers a worked example of how the analysis tool generates a deployment decision for a simple workflow.

Figures 8.4 show a simple, sequential workflow with a data source (wikimedia.org) and two workflow nodes hosted on PlanetLab (planetlab-03.cs.princeton.edu, cs-planetlab4.cs.surrey.sfu.ca). It takes an image from wikimedia.org and then sends it to the princeton.edu node for processing. The result from this step is then sent to the sfu.ca node. The workflow is specified in plain Python by providing a concrete implementation of the AbstractWorkflow specification class.

Figure 8.5 shows the workflow in relation to the closest Amazon EC2 Cloud regions (us-east-1, us-west-1, us-west-2). This illustrates the dilemma faced when deciding on the
Figure 8.5: Example workflow with closest Amazon EC2 regions

Figure 8.6: Candidate workflow graph using EC2 region us-east-1 with network latency metric
Table 8.1: Analysis tool result output

<table>
<thead>
<tr>
<th>EC2 endpoint</th>
<th>final score</th>
</tr>
</thead>
<tbody>
<tr>
<td>us-east-1</td>
<td>92530.42</td>
</tr>
<tr>
<td>us-west-2</td>
<td>186251.487</td>
</tr>
<tr>
<td>us-west-1</td>
<td>186374.351</td>
</tr>
<tr>
<td>sa-east-1</td>
<td>366450.152</td>
</tr>
<tr>
<td>ap-northeast-1</td>
<td>421102.237</td>
</tr>
<tr>
<td>ap-northeast-2</td>
<td>510982.726</td>
</tr>
<tr>
<td>ap-southeast-1</td>
<td>532180.129</td>
</tr>
<tr>
<td>eu-west-1</td>
<td>500178094.532</td>
</tr>
</tbody>
</table>

correct Cloud region to deploy the workflow orchestrator to: which Cloud region will result in the lowest execution time?

The analysis tool will take this workflow and build the internal candidate graphs for every metric and Cloud region. After all the metrics tools have terminated, the tool will have essentially labeled all edges in the graphs. Figure 8.6 is an example of what those internal graphs look like.

The tool then evaluates the graphs and ranks the Cloud regions in order of increasing execution time; i.e. the predicted best performing Cloud region will be at the top. Table 8.1 shows the output of the analysis tool, with the final score being the average of HTTP round-trip time and network latency (see Equation 7.1).

Based on the final score rankings, the decision is to deploy the workflow orchestrator in Cloud region us-east-1.

8.4 Experimental Setup

This section describes the concrete experimental setup in order to obtain the results used in the evaluation in Chapter 9.

8.4.1 PlanetLab (Test Web Services)

The PlanetLab setup used a total of 6 nodes, 5 of which were in North America and 1 of which was in continental Europe. These specific nodes were chosen as they offered the most reliable availability and performance to conduct the experiments. All nodes run a Python server script which listens for HTTP requests. On incoming HTTP requests with an

---

3PL nodes: Carnegie Mellon University, USA; Kansas State University, USA; Princeton University, USA; Simon Fraser University, Canada; University of Ljubljana, Slovenia; Williams College, USA
image sent in the request body, the script saves the image to the filesystem, then immediately loads the image from the filesystem and returns it in the HTTP response. In earlier versions of the experiments, images were rotated before being returned to the callee of the Web service. However, this caused significant problems with the limited memory of the PlanetLab nodes and was later abandoned.

8.4.2 Other servers (Test Web Services)

Even though PlanetLab is a collection of highly heterogenous computing clusters, I decided to include two further testing servers - one in a commercial data center in Germany and one on a server at the University of St Andrews. Similar to the PlanetLab, the servers host an image manipulation Web service which can be used interchangeably with the ones hosted on PlanetLab. However, due to technical restrictions the scripts were implemented in PHP and run on an Apache Web server.

8.4.3 Amazon EC2 (Workflow Orchestration and Analysis)

For both the analysis tool and the actual workflow orchestrator deployment, \textit{t1.micro} instances running Ubuntu 12.04.1 LTS in the 8 different regions were used. Ubuntu was chosen due to the big online community surrounding the operation system, the large availability of software and my previous experience.

The \textit{t1.micro} instances were chosen because the computation power and storage provided by them was sufficient for the experiments as the workflow analysis tool and the workflow orchestrator have very small memory footprints and small computation power requirements.

8.4.4 Random Workflows

The testing workflows were created by randomly selecting (with replacement) a sequence of nodes from a list of the available Web service nodes (PlanetLab and the other servers). These workflows are stored in plain-text files, with every line containing a single node. All workflows use \textit{wikimedia.org} as their data source. 7 workflows were generated - with 2, 3, 4, 5, 10 and 12 nodes. In order to evaluate more complex workflows, such as multiple source patterns, two additional workflows with 7 and 13 nodes were generated and included in the test suite.

Due to the lack of consistently available PlanetLab nodes in South America, Africa or Asia, the focus is on North America and Europe.

\footnote{smallest available instance type}
\footnote{EC2 Regions: North America: East & West; South America; Europe; Asia Pacific: Sydney, Tokyo, Singapore}
Table 8.2: Workflow execution results

<table>
<thead>
<tr>
<th>workflow</th>
<th>nodes</th>
<th>local execution (s)</th>
<th>2nd ranked (s)</th>
<th>2nd ranked vs local (s)</th>
<th>1st ranked (s)</th>
<th>1st ranked vs local (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mean</td>
<td>σ</td>
<td>prediction mean</td>
<td>mean</td>
<td>σ</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>124.86</td>
<td>36.05</td>
<td>us-west-1</td>
<td>110.75</td>
<td>40.29</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>66.10</td>
<td>16.45</td>
<td>us-west-1</td>
<td>87.81</td>
<td>3.77</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>118.85</td>
<td>14.82</td>
<td>us-east-1</td>
<td>178.56</td>
<td>7.70</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>339.14</td>
<td>64.73</td>
<td>eu-west-1</td>
<td>288.05</td>
<td>25.32</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>515.30</td>
<td>154.85</td>
<td>eu-west-1</td>
<td>400.45</td>
<td>39.70</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
<td>560.88</td>
<td>32.40</td>
<td>us-east-1</td>
<td>470.45</td>
<td>39.95</td>
</tr>
<tr>
<td>G</td>
<td>12</td>
<td>631.00</td>
<td>15.34</td>
<td>us-west-1</td>
<td>585.62</td>
<td>30.85</td>
</tr>
<tr>
<td>H</td>
<td>7</td>
<td>340.95</td>
<td>45.60</td>
<td>us-west-1</td>
<td>301.57</td>
<td>55.44</td>
</tr>
<tr>
<td>I</td>
<td>13</td>
<td>1604.83</td>
<td>1029.28</td>
<td>eu-west-1</td>
<td>635.04</td>
<td>39.18</td>
</tr>
<tr>
<td>MEAN</td>
<td></td>
<td>477.99</td>
<td>156.61</td>
<td>339.81</td>
<td>31.36</td>
<td>21.53%</td>
</tr>
</tbody>
</table>
Chapter 9

Evaluation of Results

This section describes how the performance and correctness of the approach were evaluated.

9.1 Experimental Procedure

In order to get the results presented in Table 8.2, the following steps were followed for every workflow:

1. Randomly generate workflow
2. Run pre-deployment analysis tool
3. Execute the workflow in the top two ranked Cloud regions multiple times and time it; execute the workflow on the St Andrews servers and time it
4. Collect results and compare execution times

9.2 Test Workflows

9 workflows were generated randomly using the tool. Workflows A - G are simple sequential workflows with a single source whereas workflows H and I are more complex workflows with multiple sources and fan-in and fan-out node patterns. Appendix A includes graphs of all test workflow.

9.3 General Results

Table 8.2 summarises the different workflow execution times. Every workflow was executed 4 times, on different days and different times of day to avoid systematic errors. For local execution, the table presents mean and standard deviation of execution time. For the first- and
second-ranked regions the table shows the predicted regions by the pre-deployment analysis tool. For these regions, the table also shows mean and standard deviation of execution time, mean percentage speedup compared to local execution and the percentage decrease in standard deviation compared to local execution. The boxplots in Figure 9.1(a) - 9.1(i) graphically illustrate the results of Table 8.2.

By looking at the results table 8.2 and the corresponding boxplots 9.1(a) - 9.1(i), one can clearly observe that the mean execution times are greatly reduced by orchestrating them in the highest-ranked Cloud region as opposed to executing them locally. However, the data also shows that the magnitude of reduction in execution time highly depends on the workflow being analysed. Especially Figure 9.1(c) shows that for this particular workflow, local execution time is very close to the Cloud-optimised execution time where as Figure 9.1(i) shows that the first-ranked Cloud region and second-ranked Cloud region perform almost equally well.

Figure 9.1(j) illustrates the speedup in mean execution time for each sample workflow due to being run in the first-ranked Cloud region compared to local execution. The speedups range from 3% to 188% with a mean of 82.25%. In contrast, when the workflow is deployed in the second-ranked Cloud region, the mean speedup from local execution is only 21.53%. It can be concluded that the analysis correctly ranks the Cloud regions to reduce execution time.

It can further be noted from Table 8.2 that the standard deviation of execution is reduced by an average of almost 65% when run in the first-ranked region. Only workflow H (9.1(h)) shows a higher standard deviation when executed in the first-ranked Cloud region compared to local execution. This is again due to the fact that the results depend on the nature and composition of the workflow. Nevertheless, it can be concluded overall that the highest-ranked Cloud region as calculated by the analysis tool makes execution times more stable compared to local execution.

9.4 Factors

Table 8.2 shows that the first-ranked Cloud regions were consistently faster than the second-ranked Cloud regions. Here, I discuss the significance of the individual metric factors used by the analysis tool to rank the regions. Table 9.1 summarises whether a specific metric correctly predicted the best performing Cloud region.

9.4.1 Geographical Distance

Geographical distance seems to give a consistent estimate of the best Cloud region to deploy the workflow orchestrator. Based on the results of the experiments, it can be concluded that total geographical distance of a workflow on its own is already a very good indicator to rank Cloud regions.
Figure 9.1: Execution times of the sample workflows
Table 9.1: Correct predictions

<table>
<thead>
<tr>
<th>workflow</th>
<th>total distance</th>
<th>latency</th>
<th>HTTP RTT</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>F</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>G</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>H</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>I</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

However, distance analysis is static and does not take into account unexpected network latencies on specific network links. Thus, geographical distance should only serve as a crude indicator to rank Cloud regions and I chose not to include this metric in the overall score calculation.

### 9.4.2 Network Latency

Network latency, as measured by average ICMP ping times, also seems to be consistent in predicting the best performing Cloud region. However, especially when services are hosted on big server farms, the ping might only measure latency in the Internet and not in the network behind the gateway of the Web service.

### 9.4.3 HTTP Round-Trip Time

Since network latency may not take into account the private network and application layer latencies, HTTP round-trip time was included as a potential factor to rank Cloud regions. HTTP round-trip time, as measured by a timed single request to the endpoint URL using `curl`, is useful to rank Cloud regions in some instances. There are three sample workflows, however, where the RTT prediction was incorrect - the only metric where this was the case. Therefore, this metric is only partially useful on its own.

### 9.4.4 Overall Ranking

The combined score obtained by averaging ping and RTT scores (see Equation 7.1) is a consistent indicator of Cloud region performance for the specific sample workflow.
9.5 Feasibility of Analysis

Due to the implementation of the analysis tool, the metric gathering stage has to launch multiple Amazon EC2 instances and run time-intensive metric scripts. In the experiments, the analysis tool took an average of about 400s to complete the analysis. Therefore, the analysis might be infeasible for small workflows with a small data source; however, the approach is still valid for small workflows that are going to be run multiple times in the Cloud. Consequently, I suggest to use geographical distance as a crude indicator to initially rank Cloud regions and then to run the network latency and RTT analysis on the three top ranked Cloud regions from the previous ranking.
Chapter 10

Project Evaluation and Critical Appraisal

10.1 Objectives

All primary and secondary, as presented in Chapter 2, were fulfilled. Of the tertiary objectives, only 1. was fulfilled. 2. was specified at the beginning of the project and was deemed too complicated and infeasible later on in the development process.

10.2 Requirements

All requirements laid out in Chapter 4 were successfully fulfilled by the software implementation.

10.3 Challenges

1. Open research approach: As mentioned before, the project was based on a very open research question. Therefore, the design and implementation of the software went through a lot of iterations and changed substantially over the course of the year. It was hard at times to keep track of progress because, for example, the code written in an entire week would become obsolete due to a different approach taken to the problem than originally planned. Especially most of the code written in the first months is not actually included in the final software (c.f. rapid prototyping - see Chapter 5). It was mostly proofs of concepts which were eventually incorporated in different ways. Overall, it was important to keep a good outline of the past work as well as the future direction.

2. Rigidity of AWS regions: The biggest challenge was the rigidity of the Amazon EC2 platform. Every region is distinct in that they do not share the same public keys, secu-
rity groups, AMIs (VM templates), etc. This was a big challenge early on, since the pre-deployment tool needed homogenous access to instances in all regions. Cloud management products by companies such as RightScale overcome these problems but using a commercial product was not feasible in the context of this project. Therefore, a lot of the development time at the beginning of the project went into finding and developing an approach to do this. I evaluated several approaches (e.g. creating custom AMIs in all regions, preparing images locally and then uploading them using a Git repository to distribute files) - the most efficient one turned out to be the EC2MultiRegionConnection discussed in Chapter 7.

3. **Amazon EC2 bad instances**: Amazon EC2 is not an infallible system. At times, an instance can be launched on bad blocks and thus it will not be usable or even accessible via SSH. This problem unfortunately does not become apparent during launch - only later when it cannot be used. This was fairly hard to debug initially due to the lack of a unique error code.

4. **Testing with external dependencies**: This was also a major challenge - since Amazon API calls result in a different state on every invocation, it is not easy to use a unit testing approach to verify the functionality. As discussed in Chapter 5, no existing solutions to these problems were usable in the project. My solution was to rely on the boto library - since this is a frequently used AWS library in the industry, I trusted their testing and assumed that it handles all potential short-comings and errors of the AWS APIs.

5. **PlanetLab**: PlanetLab was a very unpredictable and complicated system to use. The toolkit was poorly documented and nodes appeared/disappeared frequently. At one stage there were more than 15 nodes online - towards the end of the experiments that numbers reduced to 6. This caused problems since some experiments used certain nodes which were later unavailable for the control experiment and effectively rendered a lot of experimental results unusable. And more recently, the query mechanism of PlanetLab (comon) has been offline. Luckily, the latest list of online nodes is still up to date and thus all experiments could be conducted.

6. **Lack of Web services**: In the early stages of the project it was challenging to find existing, freely available Web services to use in the experimental workflows. Thus, I decided to use PlanetLab and some other available servers to implement a custom Web service test framework. Essentially, this shifted the focus of my project work a bit since I initially believed that there would be more existing Web services to compose sufficiently complex experiments.

http://www.rightscale.com[02/04/13]
Chapter 11

Conclusion

11.1 Key Achievements

The project addressed an approach to increase the performance of highly distributed Web service workflows by dynamically deploying the workflow orchestrator on an IaaS Cloud rather than orchestrating the Web service workflow locally. An analysis tool was developed which, using the factors geographical distance, network latency and HTTP round-trip time, can analyse a given workflow and rank Amazon EC2 Cloud regions according to predicted execution time.

I ran several randomly generated workflows and found that orchestrating workflows in the Cloud significantly reduced execution time as well as the standard deviation of execution time. It can also be concluded that both total geographical distance of the workflow as well as the average of network latency and HTTP RTT scores correctly predict the best performing Cloud region to deploy the orchestrator.

The proposed approach addresses the bottlenecks associated with executing highly distributed and data-intensive applications in the Cloud. The techniques discussed are general and can be applied to any workflow specification language and set of execution resources, e.g., further IaaS nodes such as those provided by Rackspace [26] could easily be added.

11.2 Drawbacks

1. Due to the high analysis overhead of the metric gathering stage in the analysis tool, the approach might be infeasible for some workflows, especially workflows with small data sources. A solution to this could be to preliminarily rank the Cloud regions using geographical distance and then to run the network latency and RTT analysis to “fine-tune” the top three ranked regions.

2. The software produced is very targeted to running the experiments and is not fit to be used in production. Of course, the intention was to evaluate the functionality of
different approaches rather than to produce a production-ready application.

3. PlanetLab was a fairly unreliable and unpredictable Cloud service as nodes frequently appeared and disappeared which made consistent testing of different approaches very challenging. This was especially the case with nodes in the Asia-Pacific region. Initially, PlanetLab had several available servers which however disappeared in the middle of the experiments and thus could not be used for the final evaluation.

11.3 Future Work

1. Future work could potentially look at other factors than execution time. Using a Cloud cost forecasting system and different Cloud providers, the analysis could be extended to find the best Cloud region that minimises both total cost and execution time.

2. The analysis tool only makes use of the Amazon Cloud services. The tool could be extended to make use of different IaaS Cloud providers (such as Rackspace).

3. Furthermore, the pre-deployment analysis tool could be refactored to work with an existing workflow engine and specification language (e.g. Taverna) and implemented to be production-ready.
Bibliography


[19] Javadoc. [http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc](http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc) [02/04/13].


[26] Rackspace. [http://www.rackspace.com](http://www.rackspace.com) [02/04/13].


Appendix A

Test Workflow Graphs

This appendix contains graphical representations of all randomly generate workflows used in the evaluation in Chapter [10]
Figure A.1: Workflow A (predicted "optimal" EC2 region: *us-east-1*)

Figure A.2: Workflow B (predicted "optimal" EC2 region: *us-east-1*)

Figure A.3: Workflow C (predicted "optimal" EC2 region: *eu-west-1*)

Figure A.4: Workflow D (predicted "optimal" EC2 region: *us-east-1*)

Figure A.5: Workflow E (predicted "optimal" EC2 region: *us-east-1*)
Figure A.6: Workflow F (predicted "optimal" EC2 region: eu-west-1)

Figure A.7: Workflow G (predicted "optimal" EC2 region: eu-west-1)

Figure A.8: Workflow H (predicted "optimal" EC2 region: us-west-2)
Figure A.9: Workflow I (predicted "optimal" EC2 region: us-east-1)
Appendix B

Paper

A paper titled "Location, Location, Location: Optimising Data-Intensive Workflows in the Cloud" is currently under review for the IEEE CLOUD 2013 conference.

[23/03/13]
Location, Location, Location: Optimising Data-Intensive Workflows in the Cloud

Michael Luckeneder and Adam Barker
School of Computer Science
University of St Andrews
St Andrews, United Kingdom
Email: {ml483, adam.barker}@st-andrews.ac.uk

Abstract—When orchestrating highly distributed and data-intensive Web service workflows the geographical placement of the orchestration engine can greatly affect the overall performance of a workflow. Orchestration engines are typically run from within an organisations’ network; when executing data-intensive workflows that consist of highly distributed services, the orchestration engine may have to transfer data across long geographical distances which increases execution time and degrades the overall performance of a workflow.

In this paper we present a Web service framework and analysis tool which, given a workflow specification, computes the optimal Amazon EC2 Cloud region to automatically deploy the orchestration engine and execute the workflow. Using a simplified, chained workflow specification model, we use geographical distance of the workflow, network latency and HTTP round-trip time between Amazon Cloud regions and the workflow nodes to find a ranking of Cloud regions. This overall ranking predicts where the workflow orchestration engine should be deployed in order to reduce overall execution time.

We evaluate our approach by executing randomly generated data-intensive workflows deployed on the PlanetLab platform in order to rank Amazon EC2 Cloud regions. Our experimental results show that our proposed optimisation strategy, depending on the particular workflow, can speed up execution time on average by 73% compared to local execution. We also show that the standard deviation of execution time is reduced by an average of almost 73% using the optimisation strategy.

Keywords-Cloud; scientific workflows; performance evaluation; topological workflow analysis

I. INTRODUCTION

Scientific workflows [2] are typically orchestrated using a workflow engine\(^1\) running locally within an organisation’s network. However, if the Web services in the workflow are data-intensive and spread across many geographical regions, the data might have to move long distances in order to flow from the data sources to the Web services via the orchestrator. This in turn most likely slows down the execution of the workflow and degrades the overall performance.

A possible solution to this problem could be to “move” the workflow orchestration closer to the data and the Web services. Cloud environments provide a cost-effective platform for scientists and engineers to execute their workflows in a remote data centre as demonstrated by recent research [11], [10], [16], [9], [18]. Using an Infrastructure as a Service (IaaS) Cloud, such as an Amazon EC2 instance, it is possible to automatically deploy the orchestrator into a suitable EC2 region that is “closer” to the data source and Web service nodes; in turn data would not have to travel as far and therefore execution times could be reduced.

The interesting problem arises when workflows consist of a large number of Web service nodes - all of them in different geographical regions. In these cases it is very challenging to judge where the closest and thus best-performing Cloud region might be. Furthermore, a certain Cloud region might be geographically closer to the Web service nodes but, due to a high network latency on a certain network link, using another Cloud region would actually result in a lower execution time.

In this paper we design, implement and evaluate a pre-deployment analysis tool which can dynamically compute the “optimal” Cloud region to deploy the orchestrator, given a specific workflow consisting of multiple distributed services. We consider different factors which could potentially affect the suitability of choosing a certain Cloud region: total geographical distance of workflow, network latency and HTTP round-trip time.

We develop methods to:

- automatically deploy workflows given the topology of the workflow and a number of fixed resource locations.
- automatically evaluate and compare the performance of different workflow orchestration approaches.
- evaluate different factors which influence workflow execution time.

The remainder of this paper is structured as follows: in Section II we will outline the theoretical architecture of an analysis tool and a suitable workflow specification model followed by a brief discussion about the actual implementation. We then explain in Section III how we built a testing framework using PlanetLab and Amazon EC2 to evaluate the approach and the implemented analysis tool. Finally, in Section IV, we discuss the results of the experiment and how the analysis tool significantly reduced execution times.

\(^1\)e.g. http://www.taverna.org.uk [22/02/2013]
II. PRE-DEPLOYMENT ANALYSIS

In this section we outline the abstract architecture of an analysis tool and workflow specification model. We then discuss the implementation and a worked example demonstrating how the optimisation is used in practice.

A. Architecture

1) Workflow Specification: In order to correctly analyse the workflow and find the optimized Cloud region to deploy the orchestration engine to, a workflow specification model is required. This should specify data sources, intermediate processing steps as well as a data sink. Since scientific workflows are inherently graph-based, the specification language should ideally also be graph-based so that it can easily be interpreted by the analysis tool as well as a workflow orchestrator. Since most Web services make use of HTTP, we defined endpoints as being RESTful APIs. That means that the Web service input data is encoded in the URL as well as the HTTP request with the final result being returned in the HTTP response.

The workflow specification should also make it computationally simple to retrieve an ordered set of all distinct workflow nodes. This is an important feature required by the analysis engine.

2) Pre-Deployment Analysis Tool: The pre-deployment analysis tool takes a workflow specification and builds several candidate workflow graphs, which represent the data flow. The tool further requires a list of Cloud regions (e.g. Amazon EC2 regions) that should be considered for workflow execution. Every candidate graph will be based on a different one of these Cloud regions and represent a possible optimised workflow using the Cloud instance as the workflow orchestrator. Every edge represents the data flowing from a Web service node to the Cloud region or vice versa. Figure 2 illustrates a candidate graph built for the abstract workflow described in Figure 1.

The analysis tool then gathers certain metrics for every graph in order to determine the optimal Cloud region. A metric value is required for every edge of the analysis tool’s graph. Potential candidates for useful metrics include geographical distance, network latency (as measured by the UNIX “ping” command) and HTTP round-trip time (as measured by the UNIX “curl” command) between every pair of adjacent nodes.

Based on every candidate graph and metric used, the analysis tool computes an overall score. The final output is a separate table for every metric which ranks the Cloud regions by their predicted execution times. These tables are purely ordinal and no actual execution times are predicted. Using these metrics, there are numerous possible approaches to combine them and compute an overall score based on these metric scores. Based on the experiments run for this paper, we found that the average of the sum of pings and sum of RTTs works very well in practice:

$$\frac{\sum \text{ping} + \sum \text{RTT}}{2}$$

Finally, the Cloud region with the lowest overall score should be the best region to deploy the workflow orchestrator in order to minimise workflow execution time.

B. Realization

1) Workflow Specification: In the actual implementation for the experiments, the workflows are defined using concrete implementations of an abstract Python class rather than a graph-based markup language. This has the advantage that the workflow can be specified using normal Python commands and one is not restricted to using a more rigid, graph-based specification language. The abstract workflow class also ensures that workflows can automatically be deployed and remotely run via SSH in an Amazon EC2 region on instances running Ubuntu.

As we proposed earlier, the concrete implementation of the workflow class specifies a method that returns an ordered set of all nodes used in the workflow. The bulk of the abstract workflow class deals with providing functionality to be able to run the workflow both locally as well as remotely on an Amazon EC2 Cloud instance via SSH. Upon initialisation of the class, a dependency variable is injected which allows the workflow to communicate with one or more Amazon EC2 instances. This can be used to transfer the workflow engine to the instances via SCP.

2) Pre-Deployment Analysis Tool: The pre-deployment analysis tool is written in Python and heavily uses the “boto” library to interface with the Amazon EC2 APIs. To build the internal representation of the workflow graph as well as the candidate graphs for different Cloud regions, we implemented a basic Directed Acyclic Graph (DAG) which can be traversed to calculate the sum of all edge weights.

The core of the tool is an abstraction of one to many Amazon EC2 instances in different Cloud regions. This means that with a class instantiation and a single method call, instances in multiple EC2 regions can be launched, 2https://github.com/boto/boto [22/02/2013]
SSH commands can be run, and the return values can be displayed simultaneously. Similarly, files can be transferred via SCP into multiple instances at once.

When the pre-deployment analysis tool is executed given a certain workflow, the tool goes through three stages: pre-analysis, metric gathering and analysis.

In the pre-analysis stage, it first uses the DAG implementation to create a series of graphs - one for each Amazon EC2 region and metric (21 graphs$^3$). These graphs are implementations of the graph presented in Figure 2. In these graphs, the edges correspond to the metrics that have to be gathered. Given this information, the tool enters the metric gathering stage.

The metric gathering stage starts with launching EC2 instances in every Cloud region and deploying the metric script. For every edge in every graph, the metrics are retrieved: network latency is obtained using “ping” and HTTP RTT using “curl”. To get geographical distances between nodes, the ipinfodb$^4$ API and some spherical geometry calculations are used.

In the analysis stage, the analysis tool traverses all the generated graphs and sums the edge weights. This will generate the Cloud region ranking tables for all three metrics. Furthermore, the final score is calculated by averaging the network latency and HTTP RTT metric for every Cloud region (see Equation 1). These tables are then displayed, all Amazon EC2 instances previously launched are terminated and the analysis tool quits.

C. Worked Example

Here we cover a worked example of how the analysis tool generates a deployment decision for a simple workflow.

Figures 3 show a simple, sequential workflow with a data source (wikimedia.org) and two workflow nodes hosted on PlanetLab (planetlab-03.cs.princeton.edu, cs-planetlab4.cs.surrey.sfu.ca). It takes an image from wikimedia and then sends it to the princeton.edu node for processing. The result from this step is then sent to the sfu.ca node. The workflow is specified using plain Python by providing a concrete implementation of the abstract workflow specification class.

Figure 4 shows the workflow in relation to the closest Amazon EC2 Cloud regions (us-east-1, us-west-1, us-west-2). This illustrates the dilemma faced when deciding on the correct Cloud region to deploy the workflow orchestrator to: which Cloud region will result in the lowest execution time?

The analysis tool will take this workflow and build the internal candidate graphs for every metric and Cloud region. After all the metrics tools have terminated, the tool will have essentially labeled all edges in the graphs. Figure 5 is an example of what those internal graphs look like.

$^3$3 metrics $\times$ 8 regions $= 21$ graphs
$^4$http://ipinfodb.com [22/02/2013]

The tool then evaluates the graphs and ranks the Cloud regions in order of increasing execution time; i.e. the predicted best performing Cloud region will be at the top. Table I shows the output of the analysis tool, an average of HTTP round-trip time and network latency (see Equation 1).
Based on the final score rankings, we choose to deploy the workflow orchestrator in Cloud region us-east-1.

### III. PERFORMANCE ANALYSIS

In this section we describe how we analysed the performance and correctness of our approach.

#### A. Experimental Setup

To verify the functionality of the pre-deployment analysis tool, we randomly generated workflows, analysed them and then executed and timed them several times. In order to create a highly reusable and controlled Web service workflow, we decided to host a simple, sequential image processing workflow on the PlanetLab [7] framework (a global research network) and other servers. The idea is that these services can be invoked by a simple HTTP request with an image in the HTTP request. The Web service then executes some time-consuming image processing (rotating it) and then returns the image in the HTTP response. This allows us to arbitrarily and flexibly chain the nodes in order to form various workflows.

In a typical workflow used in the experiments (e.g. Figure 3, an image is downloaded from the wikimedia servers and then passed on to the first workflow node by the orchestrator. The processed image is then downloaded from the first node and sent to the second node where the same processing step is applied, and so on.

1) **PlanetLab (Test Web Services):** In PlanetLab we used a total of 6 nodes, 5 of which were in North America and 1 of which was in continental Europe. These specific nodes were chosen as they offered the most reliable availability and performance to conduct the experiments. All nodes run a Python server script which listens for HTTP requests. On incoming HTTP requests with an image sent in the request, the script saves the image to the filesystem, rotates the image and then returns it in the HTTP response.

2) **Other servers (Test Web Services):** Even though PlanetLab is a collection of highly heterogeneous computing clusters, we decided to include two further testing servers - one in a commercial data center in Germany and one on a server at the University of St Andrews. Similar to the PlanetLab, the servers host and image manipulation Web service which can be used interchangeably with the ones hosted on PlanetLab. However, due to technical restrictions the scripts were implemented in PHP and run on an Apache web server.

3) **Amazon EC2 (Workflow orchestration and analysis):** For both the analysis tool and the actual workflow orchestrator deployment, we used t1.micro instances running Ubuntu 12.04.1 LTS in the 8 different regions. The computation power and storage provided by the t1.micro instances was sufficient for the experiments as the workflow analysis tool and the workflow orchestrator to have very small memory footprints and small computation power requirements.

4) **Random Workflows:** The testing workflows were created by randomly selecting (with replacement) a sequence of nodes from a list of the available Web service nodes (PlanetLab and the other servers). These workflows are stored in plain-text files, with every line containing a single node. All workflows use wikimedia.org as their data source.

Due to the lack of consistently available PlanetLab nodes in South America, Africa or Asia, we focus on North America and Europe. We generated sample workflows with 2, 3, 4, 5, 10 and 12 nodes.

5) **The Verification Framework:** In order to verify the optimisation proposed by the analysis tool, we implemented a very simplified, bare-bones workflow orchestrator. It repeatedly executes the workflow a pre-defined number of times and returns the total execution time. The orchestrator can easily be executed locally as well as remotely on Amazon EC2 instances via SSH. This enables us to launch the workflow in the predicted Cloud region as well as from within the university network and verify if the Cloud optimised version is faster.

### IV. RESULTS

In this section we discuss the experimental results we obtained through evaluation the previously generated workflows using the analysis tool.

#### A. General results

Table II summarises the different workflow execution times. Every workflow was executed 5 times, on different days and different times of day to avoid systematic errors. For local execution, the table presents mean and standard deviation of execution time. For the first and second-ranked regions the table shows mean and standard deviation of execution, mean percentage speedup compared to local
Table II

WORKFLOW EXECUTION RESULTS

<table>
<thead>
<tr>
<th>workflow nodes</th>
<th>local execution (s)</th>
<th>2nd ranked (s)</th>
<th>2nd ranked vs local (s)</th>
<th>1st ranked (s)</th>
<th>1st ranked vs local (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>workflow nodes</td>
<td>mean</td>
<td>σ</td>
<td>mean</td>
<td>Δσ</td>
<td>mean</td>
</tr>
<tr>
<td>A</td>
<td>124.86</td>
<td>36.05</td>
<td>110.75</td>
<td>40.29</td>
<td>13%</td>
</tr>
<tr>
<td>B</td>
<td>66.10</td>
<td>16.45</td>
<td>87.81</td>
<td>3.77</td>
<td>-25%</td>
</tr>
<tr>
<td>C</td>
<td>118.85</td>
<td>14.82</td>
<td>178.56</td>
<td>7.70</td>
<td>-33%</td>
</tr>
<tr>
<td>D</td>
<td>339.14</td>
<td>64.73</td>
<td>288.05</td>
<td>25.32</td>
<td>18%</td>
</tr>
<tr>
<td>E</td>
<td>515.30</td>
<td>154.85</td>
<td>400.45</td>
<td>39.70</td>
<td>29%</td>
</tr>
<tr>
<td>F</td>
<td>560.88</td>
<td>32.40</td>
<td>470.45</td>
<td>39.95</td>
<td>19%</td>
</tr>
<tr>
<td>G</td>
<td>631.00</td>
<td>15.34</td>
<td>585.62</td>
<td>30.85</td>
<td>8%</td>
</tr>
<tr>
<td>MEAN</td>
<td>335.69</td>
<td>47.81</td>
<td>303.1</td>
<td>26.8</td>
<td>4.14%</td>
</tr>
</tbody>
</table>

execution and the percentage decrease in standard deviation compared to local execution.

The boxplots in Figure 6(a) - 6(g) graphically illustrate the results of Table II. We can clearly observe that the mean execution times are greatly reduced by orchestrating them in the highest-ranked Cloud region as opposed to executing them locally. However, the data also shows that the magnitude of reduction in execution time highly depends on the workflow being analysed. Especially Figure 6(c) shows that for this particular workflow, local execution time is very close to the Cloud-optimised execution time.

Figure 6(h) illustrates the speedup in mean execution time for each sample workflow due to being run in the first-ranked Cloud region compared to local execution. The speedups range from 3% to 188% with a mean of 73.71%. In contrast, when the workflow is deployed in the second-ranked Cloud region, the mean speedup from local execution is only 4.14%. We can conclude that the analysis correctly ranks the Cloud regions to reduce execution time.

We also note from Table II that the standard deviation of execution is reduced by an average of almost 73% when run in the first-ranked region. This leads to the conclusion that the highest-ranked Cloud region as calculated by the analysis tool makes execution times more stable compared to local execution.

B. Factors

Table II showed that the first-ranked Cloud regions were consistently faster than the second-ranked Cloud regions. Here, we discuss the significance of the individual metric factors used by the analysis tool to rank the data centers.

Table III summarises whether a specific metric correctly predicted the best performing Cloud region.

1) Geographical distance: Geographical distance seems to give a consistent estimate of the best Cloud region to deploy the workflow orchestrator. Based on the results of our experiments, we can conclude that total geographical distance of a workflow on its own is already a very good indicator to rank Cloud regions.

However, distance analysis is static and does not take into account unexpected network latencies on specific network links. Thus, geographical distance should only serve as a crude indicator to rank Cloud regions and we chose not to include this metric in the overall score calculation.

2) Network latency: Network latency, as measured by average ICMP ping times, also seems to be consistent in predicting the best performing Cloud region. However, especially when services are hosted on big server farms, the ping might only measure latency in the Internet and not in the network behind the gateway of the Web service.

3) HTTP round-trip time: Since network latency may not take into account the private network and application layer latencies, we included HTTP round-trip time as a potential factor to rank Cloud regions. HTTP round-trip time, as measured by a single request to the endpoint URL using “curl”, is useful to rank Cloud regions in some instances. There are two sample workflows, however, where the RTT prediction was incorrect. Therefore, this metric is only partially useful.

4) Overall ranking: The combined score obtained by averaging ping and RTT scores (see Equation 1) is also a consistent indicator of Cloud region performance for the specific sample workflow.

Table III

<table>
<thead>
<tr>
<th>workflow</th>
<th>total distance</th>
<th>latency</th>
<th>HTTP RTT</th>
<th>overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>B</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>C</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>D</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>E</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>F</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>G</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

However, distance analysis is static and does not take into account unexpected network latencies on specific network links. Thus, geographical distance should only serve as a crude indicator to rank Cloud regions and we chose not to include this metric in the overall score calculation.

2) Network latency: Network latency, as measured by average ICMP ping times, also seems to be consistent in predicting the best performing Cloud region. However, especially when services are hosted on big server farms, the ping might only measure latency in the Internet and not in the network behind the gateway of the Web service.

3) HTTP round-trip time: Since network latency may not take into account the private network and application layer latencies, we included HTTP round-trip time as a potential factor to rank Cloud regions. HTTP round-trip time, as measured by a single request to the endpoint URL using “curl”, is useful to rank Cloud regions in some instances. There are two sample workflows, however, where the RTT prediction was incorrect. Therefore, this metric is only partially useful.

4) Overall ranking: The combined score obtained by averaging ping and RTT scores (see Equation 1) is also a consistent indicator of Cloud region performance for the specific sample workflow.
Figure 6. Execution times of the sample workflows
C. Feasibility of analysis

Due to the implementation of the analysis tool, the metric gathering stage has to launch multiple Amazon EC2 instances and run time-intensive metric scripts. In our experiments, the analysis tool takes an average of about 400s to complete the analysis. Therefore, the analysis might be infeasible for small workflows with a small data source; however, the approach is still valid for small workflows that are going to be run multiple times in the Cloud. Consequently, we suggest to use geographical distance as a crude indicator to initially rank Cloud regions and then to run the network latency and RTT analysis on the three top ranked Cloud regions from the previous ranking.

V. RELATED WORK

This paper addresses the problem of where geographically to deploy a workflow engine, given the specification of a workflow consisting of highly distributed services and a set of fixed points - in this case Amazon EC2 Cloud regions. Related work covers topics such as decentralised orchestration, third-party data transfers and data aware scheduling. However, as far as we know there has been no prior work on the topic of dynamically migrating workflow engines to Cloud-based resources in order to improve performance.

A. Decentralised Orchestration

The concept of pointers in service-oriented architectures [19] allows Web services to pass data by reference rather than by value. This has the advantage that the workflow orchestrator doesn’t need to handle all data passing between the orchestrated Web services.

The Flow-based Infrastructure for Composing Autonomous Services or FICAS [15] is a distributed data-flow architecture for composing software services. Composition of the services in the FICAS architecture is specified using the Compositional Language for Autonomous Services (CLAS), which is essentially a sequential specification of the relationships among collaborating services. This CLAS program is then translated by the build-time environment into a control sequence that can be executed by the FICAS runtime environment.

Service Invocation Triggers [5] is an architecture for decentralised execution. Before execution can begin the input workflow must be deconstructed into sequential fragments, these fragments cannot contain loops and must be installed at a trigger.

In previous work [4] [3] we proposed Circulate, a proxy-based architecture based on a centralised control flow, distributed data flow model. In [6], an architecture for decentralised orchestration of composite Web services defined in BPEL is proposed.

All the approaches discussed in this section require either the workflow specification or the services involved in the workflow to be altered prior to enactment. In FICAS the application code that is to be deployed needs to be wrapped with a FICAS interface; in the SOA pointers and Triggers approaches the workflow specification needs to be altered before enactment; the IBM approach does not deal with the problem of where to geographically deploy an orchestration engine; Circulate like the other approaches requires alteration to the workflow specification and the addition of an extra actor, a proxy. In contrast, our approach enables a workflow to be analysed and then dynamically migrated to a Cloud-based resource for execution, this avoids the costly setup cost of wrapping back-end services.

B. Third-party Data Transfers

This paper focuses primarily on optimising workflows where services are: not equipped to handle third-party transfers, owned and maintained by different organisations, and cannot be altered in anyway prior to enactment. For completeness it is important to discuss engines that support third-party transfers between nodes in task-based workflows.

Directed Acyclic Graph Manager (DAGMan) [8] submits jobs represented as a DAG to a Condor pool of resources. Intermediate data are not transferred via a workflow engine, instead they are passed directly from vertex to vertex. DAGMan removes the workflow bottleneck as data are transferred directly between vertices in a the DAG. Triana [17] is an open-source problem solving environment. It is designed to define, process, analyse, manage, execute and monitor workflows. Triana can distribute sections of a workflow to remote machines through a connected peer-to-peer network. OGSA-DAI [12] is a middleware product that supports the exposure of data resources on to Grids. This middleware facilitates data streaming between local OGSA-DAI instances. Grid Services Flow Language (GSFL) [14] provides functionality for Grid services to adopt a peer-to-peer data flow model.

C. Data-Aware and Location-Aware Scheduling

Amazon have recently added Latency-Based Routing (LBR) [1] to the Route 53 service. LBR provides functionality to reduce latency for end users by serving their requests from the region for which the network latency is lowest. LBR does not consider the complexities of where to deploy an application that is constructed from a number of highly distributed services.

Stork proposes an approach to data-aware scheduling [13]: given an application dynamically deciding where to deploy the data. In contrast our approach decides where to deploy the application assuming that the data are fixed and cannot be relocated.

VI. CONCLUSION/FUTURE WORK

This paper discussed how to increase the performance of highly distributed Web service workflows by dynamically deploying the workflow orchestrator on an IaaS Cloud rather than orchestrating remote services locally. We developed an
analysis tool which, using the factors geographical distance, network latency and HTTP round-trip time, can analyse a given workflow and rank Amazon EC2 Cloud regions according to predicted execution time.

We ran several randomly generated workflows and found that orchestrating workflows in the Cloud significantly reduced execution time as well as the standard deviation of execution time. Overall, we concluded that both total geographical distance of the workflow as well as the average of network latency and HTTP RTT scores correctly predict the best performing Cloud region to deploy the orchestrator. However, due to the high analysis overhead of the metric gathering stage in the analysis tool, we proposed to preliminarily rank the Cloud regions using geographical distance and then to run the network latency and RTT analysis to “fine-tune” the top three ranked regions.

Our proposed approach addresses the bottlenecks associated with executing highly distributed and data-intensive applications in the Cloud. The techniques discussed are general and can be applied to any workflow specification language and set of execution resources, e.g., we could easily add further IaaS nodes such as those provided by Rackspace.

Future work could potentially look at other factors than execution time. Using a Cloud cost forecasting system and different Cloud providers, the analysis could be extended to find the best Cloud region that minimises both total cost and execution time.

References
Appendix C

Documentation

The code for this project is hosted on Github under https://github.com/mluckeneder/movingdata. The documentation can be found in the repository as a Markdown document. A copy of the documentation is included below.
Uncovering the Perfect Place: Optimising Workflow Engine Deployment in the Cloud

Technology overview

Amazon AWS is used to host the workflow orchestrator and run the analysis tools.

PlanetLab runs simple RESTful web services which receive an image, write it to the disk and retransmit it again.

Required tools

AWS toolkit

To set up the Amazon AWS toolkit, follow the official guide: http://aws.amazon.com/developertools/351 and configure the toolkit with your AWS account credentials.

PlanetLab

In order to set up the PlanetLab toolkit, follow the official getting started guide: https://www.planet-lab.org/doc/guides/user

Python

Install Python 2.7.3, pip and virtualenv

Setup

Python

Set up a new virtualenv by following the getting started guide on virtualenv and then run pip install -r requirements.txt. This will install all requirements for the project.

Then take some time to edit config.py and put in your AWS credentials and an API key from ipinfodb.

AWS

Follow the Amazon AWS console instructions to create a key pair and a security group. The key pair and security group have to be copied to all AWS regions.

Configure the parameters in script/add_keypair and then, from the main directory, run sh scripts/add_keypair. This will copy the keypair to all regions.

Similarly, the command python script/copy_security_groups.py can be used to copy all security groups to all AWS regions.

PlanetLab

The PlanetLab setup files live in eg/planetlab.

The file eg/planetlab/nodes.txt contains a list of PlanetLab nodes where the test web service workflows can be executed. When the project was started, a command was used to retrieve this list of nodes from the PlanetLab comon service. However, as of March 2013, this service does not respond to requests and thus the list of live nodes cannot be queried anymore. Since PlanetLab nodes tend to go offline, it might be possible that none of the nodes defined in nodes.txt work.

The script file eg/planetlab/pl.sh is used to control the web services hosted on PlanetLab.

sh eg/planetlab/pl.sh deploy copies the file in eg/planetlab/cs4098/server.py to every node defined in nodes.txt.

sh eg/planetlab/pl.sh install installs the Python dependencies on all nodes.

sh eg/planetlab/pl.sh start starts the web service on every node (accessible via HTTP on port 31415).

sh eg/planetlab/pl.sh stop stops the web service on every node.
Workflow analysis

All the commands should be executed from the deploy directory.

Defining Workflows

A sample workflow, which was used for the IEEE Conference Paper, is included. It loads the workflow specification from plaintext files in the ieee/inputs directory. Every line of these files contains a separate node in the workflow. The data source is defined in deploy/ieee_test_workflow.py.

Generating random workflows

`python generator.py N` generates a random sequential workflow with N nodes (with replacement). It uses the nodes list in eg/planetlab/nodes.txt as a source. The workflow specification will be printed to STDOUT.

For example, if a new workflow should be generated for the sample workflow, the following command could be used:

```
python generator.py 5 > ../ieee/inputs/wf1.txt
```

Running the preanalysis tool

`python ieee_analyzer.py WF.txt` runs the workflow specified in ieee/inputs/WF.txt. All logs are displayed in STDERR. STDOUT is automatically redirected to ieee/outputs/WF.txt_pre. This file contains the output of the analysis tool.

For example, if the workflow generated above should be analysed, the following command could be used:

```
python ieee_analyzer.py wf1.txt
```

The result table can then be found in ieee/outputs/wf1.txt_pre.

Executing the workflow

`python ieee_runner.py WF.txt REGION1,REGION2,..` runs the workflow specified in ieee/inputs/WF.txt in AWS EC2 regions REGION1 and REGION2. Similarly to the analysis tool, logs are displayed in STDERR and STDOUT is redirected to ieee/outputs/WF.txt_output.

For example, if the workflow generated above should be executed in the regions us-east-1 and us-west-2 and timed, the following command could be used:

```
python ieee_runner.py wf1.txt us-east-1,us-west-2
```

The result table can then be found in ieee/outputs/wf1.txt_output.

Running unit tests

Unit tests for the DAG implementation were written using the testify framework. The tests can be invoked by simply running this command from the root directory:

```
testify tests
```