A Highly Scalable Infrastructure for Policy Optimization Based on Open Data

Isern Bennassar, Lluís

Curs 2016-2017

Director: Mario Ceresa

GRAU EN ENGINYERIA EN INFORMÀTICA

Treball de Fi de Grau
A Highly Scalable Infrastructure for Policy Optimization Based on Open Data

Lluís Isern Bennassar

TREBALL DE FI DE GRAU
GRAU EN ENGINYERIA EN INFORMÀTICA
ESCOLA SUPERIOR POLITÈCNICA UPF
2016/17

Director: Mario Ceresa
Abstract

This project consists of the research, design, and implementation of a scalable, parallelizable, system of data extraction and processing, on a cloud-based platform. This project is inspired by the trend of open data, published by institutions, often governmental, without usage restrictions, such as copyright. The software presented here tackles the problem of using large volumes of data from heterogeneous sources to run computational models.

The idea behind this project was conceived in the context of Impact Futures, an international project proposal coordinated by SIMBioSys, a research group at Universitat Pompeu Fabra. Impact Futures aims to use data, models, and simulations to explore the world’s possible futures, in order to better understand the effects of our policies.

For designing and exploring models, we turn to an emerging paradigm, known as Agent-Based Modelling (ABM), in which large numbers of autonomous agents interact with each other and the world to produce emergent behaviours found on complex, real-world systems. To achieve this, we use Pandora, a software developed at the Barcelona Supercomputing Centre.

The system is structured as a software pipeline, consisting of four stages: (a) data acquisition, (b) data processing, (c) model execution and optimization, and (d) presentation of results. We present a workflow that utilizes all of the software in the pipeline, based on open data on the hospitals of Catalonia, and an epidemiological model that we implement in ABM.
Resum

Aquest projecte consisteix en la recerca, disseny, i implementació d’un sistema escalable i paral·lelitzable per a la extracció i processament de dades, a una plataforma basada al núvol. Aquest projecte està inspirat per la tendència d’open data, publicada per institucions, sovint governamentals, sense restriccions d’ús, com copyright. Presentem un programari que tracta el problema d’utilitzar dades en grans vols de fonts heterogènies per executar models computacionals.

La idea d’aquest projecte va ser concebuda al context de Impact Futures, una proposta de projecte internacional coordinada per SIMBioSys, un grup de recerca a la Universitat Pompeu Fabra. Impact Futures pretén usar dades, models, i simulacions per a explorar els possibles futurs del món, per tal d’entendre millor els efectes de les nostres decisions.

Per dissenyar i explorar models, fem servir un paradigma emergent, conegut com a modelat basat en agents (ABM), al qual un gran nombre d’agents autònomes interaccionen entre sí i amb el món, per tal de produir comportaments emergents propis de sistemes complexes del món real. Per aconseguir això, fem servir Pandora, un programari desenvolupat al Centre de Supercomputació de Barcelona.

El nostre sistema està estructurat com una canonada, que consisteix en quatre etapes: (a) adquisició de dades, (b) processament de dades, (c) execució i optimització del model, i (d) presentació de resultats. Presentem també un pla de treball que fa servir les quatre etapes del sistema, basat en dades obertes dels hospitals de Catalunya, i un model epidemiològic que implementem fent servir ABM.
## Contents

Abstract i  
Resum ii  

1 Introduction 1  
1.1 Objectives 2  
1.2 Work plan 2  
1.3 State of the art 4  
1.3.1 Successful open data projects 4  
1.3.2 High performance scientific computing 5  
1.3.3 Apache Hadoop 6  
1.3.4 Cloud computing 7  
1.3.5 Cloud service providers 8  

2 Materials and methods 17  
2.1 Development environment and tools 17  
2.2 Data visualization with CARTO 19  
2.3 Sample dataset 21  
2.4 Dataset processing strategy 24  
2.5 Agent-based modelling with Pandora 25  
2.5.1 Baseline performance comparison 26  
2.5.2 Zombie contagion model 28  
2.6 SIR model 29  

3 Results 31  
3.1 Software architecture 31  
3.2 Infrastructure components 32  
3.2.1 Master instance 35  
3.2.2 Source downloading cluster 36  
3.2.3 Source scraping cluster 37  
3.2.4 Agent-based modelling simulation cluster 38  
3.2.5 Presentation of results 39  
3.3 Agent-based SIR model 39  
3.4 Performance analysis of the simulation cluster 44  
3.5 Example workflow 48  
3.5.1 Obtaining the source data 49  
3.5.2 Scraping the source data 49  
3.5.3 Adding geographical information 51  
3.5.4 Running the simulation 51  
3.5.5 Visualizing the simulation 52  
3.5.6 Orchestration 53  

4 Discussion and conclusions 55  
4.1 Future work 56  

References 59
Appendix A  Python worker scripts  61
Appendix B  Execution environment and setup  63
Appendix C  Orchestration with Ansible  67
Appendix D  Software licensing  69
Appendix E  Bootstrap script for the ABM tests  71
List of Tables

1. Locations of cloud providers closest to us ................................. 8
2. AWS EC2 instances matrix ....................................................... 11
3. Azure Compute instances matrix ............................................. 15
4. Indicators in the Catalonian hospitals dataset, per table .............. 24
5. Licenses of software used in this project .................................. 69

List of Figures

1. Pipeline and work packages .................................................... 3
2. Hadoop multi-node architecture .............................................. 6
3. AWS Management Console ..................................................... 9
4. Hadoop-related tools available in AWS EMR ................................. 12
5. CARTO web app, on the dataset viewer .................................... 20
6. Cassandra (GUI of Pandora) ................................................... 26
7. Software components ............................................................ 31
8. Infrastructure components ...................................................... 33
9. Integration of AWS, an HPC, and CARTO .................................. 34
10. Evolution of our agent-based SIR model, without hospitals .......... 40
11. Evolution of our agent-based SIR model, with hospitals .............. 42
12. Runtime performance of Mesa and Pandora in EC2 ..................... 45
13. Runtime impact of agents in Mesa and Pandora in EC2 ............... 45
14. Visualization of our model in CARTO ..................................... 53

List of Listings

1. Command to create the directory structure for rpmbuild, in Bash .... 18
2. URL to the Git repository with our code .................................. 18
3. config.xml file for the agent-based SIR model, in XML ............... 41
4. Source code of Human::updateState(), in C++ .......................... 44
5. Original version of OpenMPSingleNode::executeAgents(), in C++ ... 47
6. Data source descriptor, in JSON ............................................. 49
7. Data scraping descriptor, in JSON .......................................... 50
8. Definition of a data simulation descriptor, in JSON ...................... 52
9. Format of objects sent to hpopt-main-jobs-queue, in JSON .......... 53
10. Header of the hpopt-rpmbuild.spec file ................................ 63
11. %description and %prep sections of the hpopt-rpmbuild.spec file 63
12. %pre section of the hpopt-rpmbuild.spec file .......................... 64
13. %install section of the hpopt-rpmbuild.spec file ...................... 64
14. %post and %clean sections of the hpopt-rpmbuild.spec file .......... 65
15. %files section of the hpopt-rpmbuild.spec file ........................ 65
16. make-rpm.sh script, in Bash ............................................... 66
17. Example of a simple Ansible playbook for EC2 .......................... 68
18. Bootstrap script for the ABM performance tests, in Bash .............. 71
1 Introduction

In the digital era, the world is becoming increasingly more complex and interconnected. This raises several challenges for our policy makers, for whom it is not always obvious which course of action is best. Questions they face include terrorism, climate change, population growth, health epidemics, etc. We cannot fully understand the consequences of our actions, leading to suboptimal decisions in the allocation of resources.

However, the ongoing trend of publicly disclosing large datasets through open data web platforms presents new ways for government institutions to remedy this, while promoting public-private partnerships; ‘open’ is understood as specified by the Open Knowledge Foundation (OKFN)\(^1\): “Open data and content can be freely used, modified, and shared by anyone, for any purpose.”\(^2\). Unfortunately, the technical challenges are great because of the multiple non-trivial technologies and methods required in processing the large amounts of data involved, and extracting statistically valid insights. This project aims to face some of these challenges, and proposes a platform that is capable of handle large datasets, and efficiently perform computational analysis on these. The resulting platform is designed to split computational resources between a cloud-based infrastructure, and a high-performance computing (HPC) cluster.

The project presented here is a proof of concept of a larger project proposal, Impact Futures, coordinated by a research group at UPF (SIMBioSys), with collaboration from multiple organizations from around the European Union: IESE Business School, Imperial College of London, ATOS, Institut Jozef Stefan, and Electricité de France.

Impact Futures will be presented to Horizons2020, a funding program by the European Union that aims to address key societal challenges, while encouraging scientific excellence, driving economic growth, and therefore ensuring the global competitiveness of EU nations. Horizons2020 has almost 80 million € of funds available over a seven year period, from 2014 to 2020, making it the biggest EU Research and Innovation programme of the moment.

Some of the research objectives include building data driven models to analyse and visualize the world’s possible futures, building stochastic models for the description of society through the interactions of agents, and simulating these models and agents on a massive scale. We aim to fulfill that objective by designing and implementing an infrastructure capable of sourcing and processing large amounts of data, and running large simulations of agent-based models on it.

The technological objectives include making a web-based platform for interactive exploration of data, and demonstrating the achievement of the scientific objectives by implementing representative pilot scenarios. Even though we do not provide an interactive, web-based interface, our design does accommodate for such a component, and it can be trivially integrated into the infrastructure we present here.

With the work presented here, we aim to contribute to future innovation by designing a

\(^1\)https://okfn.org/
\(^2\)http://opendefinition.org/
One of the most important components of this project is the simulation of agent-based models to approximate complex dynamical models. As explained in section 2.5, there are some technical difficulties in executing large scale simulations, both at the hardware level, and at the programming level. Therefore, instead of implementing a few generic models to be used by our users, we have chosen to let them bring their own models; even though this means that more work will have to be done in the future, our infrastructure will be able to accommodate that work more easily.

1.1 Objectives

Since we intend to lay the technological foundation of the Impact Futures project, as explained in the next section, a part of their objectives constitutes a set of requirements for our own project. Additionally, we deliberately exclude the work of designing and implementing programs for analyzing concrete models, as we want to be able to produce a system that is flexible enough for usage with lots of different kinds of data and analytical models.

1. To create an infrastructure capable of using data, models, and simulations to explore a hypothetical situation of a pandemic in Catalonia.

2. To explore the modern conceptions of open data and open government, and to implement a solution that allows more transparency in the public sector, through the use of automated and semi-automated insight extraction.

3. To develop the cloud-based infrastructure in a way that adheres to modern standards for design and management. Accordingly, the infrastructure will be packaged in a way that permits handing it out to another party, preferably in the form of code.

4. To release the resulting code and infrastructure management tools under a public, free and open source license, and to develop the framework in such a way that permits collaboration from the general public, so as to allow others to improve and build upon our work. Additionally, we will utilize similarly licensed tools and frameworks, whenever feasible.

1.2 Work plan

The development effort has been divided in four work packages (WPs) that closely resemble the final software architecture, which looks like a pipeline with four consecutive stages. Additionally, some development time has been destined to the communication mechanisms that integrate the four WPs, and to the example workflow presented in section 3.5.

The first WP and stage is data acquisition. In this first stage, open data is downloaded through the Internet from one or more providers, and stored in our private cloud network
for cheaper future access, just so that we do not have to download everything every time we want to run a new simulation. In terms of runtime performance, the bottleneck is most certainly the bandwidth that we get out of the routes through the Internet.

The second WP has to do with processing the data, or scraping. The main purpose of this stage is to reformat data, so that possible inconsistencies in data format and/or semantics can be remedied. It is also be possible to enrich datasets with additional information, such as geolocation, etc. In this stage, a mixture of I/O bandwidth and processing power is desired, since we will be reading data sequentially, and performing CPU operations on every row. We use Hadoop as the computational platform, as it is optimal for processing large datasets over a distributed infrastructure. Here, we want to be able to scale linearly, so that the time required to process a specific dataset is inversely proportional to the amount of computational resources it has available; also, we want to process the entire dataset in one single pass, whenever possible, as multiple passes increase computational complexity.

Once we have all the data we want, in the desired format, we run the third WP: this stage is a simulation of an agent-based model that involves a series of sequential steps, with multiple agents that approximate human emergent behaviour by computational means. The inputs of the simulation are, on the one hand, the open data we processed in the earlier stages, and, on the other hand, model-specific parameters that can be used to tweak the response of the model in various ways. In other words, open data is what we consider to be the representation of the actual state of the world, and the parameters are what we use to model the different possible futures of that particular state. The output of this stage is another dataset, which is the state of the model after being altered by agents for a number of steps, as well as intermediate states, given a set of initial states. For this stage we integrate a high performance computational platform with our cloud-based infrastructure, send simulation jobs to it, and wait for it to finish; then, import the data back to our cloud.

Finally, for the fourth WP, we present the desired results back to the users by means of infographics, rather than raw numeric tables, due to the sheer size of the datasets that we will be working with. We have decided to integrate with CARTO for presenting results, but this is highly dependent on the nature and semantics of the datasets; therefore, any other tool, online or otherwise, should be able to be integrated. CARTO lets users upload data to a PostgreSQL database, and plot it on a map by means of various APIs.
1.3 State of the art

1.3.1 Successful open data projects

There is a substantial number of existing open government data platforms. Notable examples are UK, USA, EU, etc. In the context of the European Union, the “Directive 2003/98/EC on the re-use of public sector information” [12], otherwise known as the PSI directive, was created, which served to allow for a unified legislative framework for member states to publish and re-use public sector information; previously, this was left to the member states to regulate. Accordingly, the ePSIplus web portal was set up, and later renamed to ePSIplatform. In 2013, the directive was amended [13] to better align with the concept of open government data, and to also include information on cultural heritage, among other things; as a result, the ePSIplatform was reshaped into the European Data Portal3,4.

In the United States of America, in March 2009, the Data.gov web platform5 was set up at the Federal Government level, with the purpose of improving public access to high-value, machine-readable datasets originating from various state, local, and tribal government agencies.

According to the Open Knowledge Foundation, Spain ranks 17 in the 2015 index [10], with highly positive scores for weather and pollution datasets, as well as legislation information, election results, and some others, but with very bad results for government spending, land ownership, and company register datasets; there is room for improvement in our government.

Our project presents an infrastructure for processing large datasets, but without being restricted to specific domains, providers, or technologies. The reason for this is that there are no standards for the exchange of open data; generally, open formats are used, which do not have legal usage restrictions like proprietary formats do, which is good for this purpose, but they do not solve the problem of having an unfeasible number of similar but different formats, since no single standard is imposed (or even, a more sensible number of limited options). Given the heterogeneity in the field of open data, it makes more sense not to couple the infrastructure to the origin of data, and allow mixing different providers, as long as we are able to deal with potentially different data formats.

In section 2.3 we describe an example of an open dataset, which we use to simulate an agent-based model. Since we have used only one provider, we do not tackle the specifics of the problem described in the previous paragraph, and instead focus on the problem of building a world model that is based on open data.

3http://www.europeandataportal.eu/en
5https://www.data.gov/
1.3.2 High performance scientific computing

Technological and industrial advances in recent years have brought high performance computing (HPC) to scientists of multiple fields and disciplines, like physics, chemistry, engineering, life sciences, social sciences, etc. In Spain, there are a number of supercomputers, the most powerful of which is the MareNostrum, located in the Universitat Politècnica de Catalunya, Barcelona, Catalonia. It is maintained by the Barcelona Supercomputing Center (BSc), a publicly funded high performance computing centre for research in various scientific fields, such as computational sciences, earth sciences, life sciences, etc.

Xavier Rubio-Campillo, a postdoctoral researcher at BSc, has developed Pandora [21], a C++/Python Agent-Based Modelling (ABM) framework for HPC environments, as explained in section 2.5. ABM is a paradigm useful for simulation of sociological models, so our project can benefit considerably from a collaboration that enables our resulting infrastructure to use their software and compute resources to run simulations on the data we collect, and allows our users to optimize certain parameters of the world. The Pandora library is able to take advantage of the large compute capacity of an HPC cluster while maintaining a relatively simple programming interface, with two mechanisms: one, based in OpenMP, and another based on MPI. This can work in many kinds of different environments, but it is already being used at BSc, so it would be a very good choice for the HPC component of our infrastructure, since it already has an installation of Pandora.

The first mechanism that Pandora uses for parallel programming is OpenMP\(^6\), a specification for shared-memory multiprocessing. It is supported in most processor architectures and operating systems. It is offered as a set of compiler directives, for multiple languages. OpenMP is provided under various licenses, depending on what compilers are used. It uses parallel programming paradigms to split work among multiple processors that share memory (access to it must be synchronized by protecting critical sections of code or data).

The other mechanism Pandora uses is Message Passing Interface, or MPI, a standard communication protocol for parallel programming on multiple nodes, each with its own private memory, plus a shared distributed memory. There have been multiple implementations over the years; Pandora uses MPICH\(^7\), originally developed at the Argonne National Laboratory (Illinois, USA), and described in detail in a 1996 article [19], by William Gropp et al.; it is released as public domain software. There are three major versions of the MPI standard: MPI-1, MPI-2, and MPI-3.

At UPF, there is the SNOW HPC cluster\(^8\). In terms of hardware, there are eleven nodes with four 16-core processors, and four nodes with two dual-cores. This sums up to 736 cores, more than 3 Tb of RAM, and around 13 TB of NFS disk space. There is a software that lets users asynchronously schedule jobs: the Asynchronous Job Operator (AJO) [17]; this manages the execution of a process by enqueuing it until the specified number of compute cores become available; after that, the process is run, in the background. During development, we had access to one of these nodes, which we used, mostly, to explore the differences in performance between this cluster and a cloud-based one.

\(^6\)http://openmp.org/
\(^7\)http://www.mpich.org/
\(^8\)http://hpc.dtic.upf.edu/
1.3.3 Apache Hadoop

When it comes to processing huge amounts of data, the most widely used software system is Apache Hadoop\(^9\), developed and maintained by the Apache Software Foundation (ASF)\(^{10}\). According to them: “The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”

The Hadoop core distribution consists of four interdependent modules: Hadoop Common is the set of common utilities that supports the other Hadoop modules. Hadoop Distributed File System (HDFS) is the distributed file system that provides high-throughput access to application data. Hadoop YARN is the framework that schedules jobs manages cluster resources. Hadoop MapReduce is a YARN-based system for parallel processing of large data sets. The MapReduce algorithm itself was originally devised at Google, in 2008, by Jeffrey Dean and Sanjay Ghemawat\(^{8}\); Hadoop MapReduce is Apache’s own implementation of this algorithm.

Additionally, there are other Hadoop-related projects at Apache, such as Ambari, a tool for provisioning, managing, and monitoring Hadoop clusters. In fact, there is an ever-growing ecosystem of free and open source tools\(^{11}\), on several fields, such as filesystem alternatives to HDFS, high level distributed programming languages, integration with SQL databases and various kinds of NoSQL and NewSQL solutions, advanced machine learning, etc.

Figure 2 describes a typical multi-node Hadoop cluster. The majority of nodes act as slaves to the masters, containing the actual data of HDFS, and performing the computations of MapReduce. The masters, on the other hand, are in charge of coordinating the actions of the slaves, but do not actually own the data or computational power themselves.

---

\(^9\)https://hadoop.apache.org/

\(^{10}\)https://www.apache.org/foundation/

\(^{11}\)https://hadoopecosystemtable.github.io/
1.3.4 Cloud computing

In recent years, several providers of cloud computing have emerged, together with new ways to deploy an organization’s infrastructure. The most widely accepted definition of cloud computing is that of the National Institute of Standards and Technology (NIST) of the United States [14]. They define cloud computing as a paradigm of distributed computing, in which processes and resources are shared over the Internet, in a way that permits very rapid deployment and/or provisioning from the customer’s side. From the point of view of the end-user, this is almost completely invisible, since cloud customers can easily expose their own services to the Internet via the cloud, which is inherently Internet-enabled, and even integrate those resources with those on-premises, allowing for almost any mixture of on-cloud and off-cloud resources.

Another such benefit is the pricing model: it is typically a pay-as-you-go kind of service, bills being paid monthly, quarterly, etc., only for those resources that are actually being consumed. Therefore, barely any upfront investment is needed, since all hardware is already being managed by the service provider. However, this means that special care must be taken not to end up spending more than what can be afforded (most cloud providers offer mechanisms to limit usage, for example, when the bill exceeds a certain monthly budget). A very attractive characteristic that cloud providers offer is the ability to scale linearly, or to reach economies-of-scale.

This is, however, not a system without its share of flaws and inconveniences, particularly those concerning privacy (both for the cloud customer and the end-user!). The reason is that data must be handed over to a set of machines over which the cloud customer has no control; this is particularly dangerous when handling sensitive user data (bank accounts, personal data, etc.). The cloud customer should also be careful with what they share (i.e. trade secrets, private data, etc.) with the cloud, as unauthorized accesses are still a real possibility on highly certified cloud providers. Another problem often found by users of the cloud is that the services offered by different providers tend to be quite heterogeneous, meaning that migrating from one provider to another can be a non-trivial task.

A modern cloud-based provider usually offers its services in a three-tier stack (be it explicitly or otherwise), dictated by the degree of virtualization and abstraction desired by the cloud customer.

**Infrastructure as a Service (IaaS)** is the most basic kind of cloud-based service model, in which the provider offers virtual machines and/or other resources to clients, while abstracting away from them the specific details of the underlying infrastructure, like physical computing resources, data partitioning/redundancy, scaling, etc. IaaS presents some characteristics that make it a very attractive option for small and medium-sized enterprises, as a way to host internal infrastructure, as well as client-facing services. This is the least abstract type of cloud-based service, and, thus, more development time must be spent in building and managing it. IaaS is suitable for infrastructures that require responsiveness to spikes in demand, small businesses that lack the investment capital for hardware, and large businesses looking to cut costs in hardware. IaaS is unsuitable for environments in which outsourcing data storage and processing is not permitted, and businesses that already own an infrastructure capable of meeting the specified requirements.
**Platform as a Service (PaaS)** further abstracts away on IaaS, on things like operating system(s), network management, etc. PaaS providers typically auto-scale up and down in a manner completely transparent to their customers. PaaS lives somewhere in between IaaS and SaaS, a middle ground that is particularly comfortable to developers looking to create and maintain web applications quickly and easily, without the burden of managing an entire infrastructure. PaaS is suitable for environments where the development team must interact with multiple external parties, and for teams that follow agile development methodologies, where frequent iterations must be quickly deployed. PaaS is unsuitable for environments with strict requirements about portability, and wherever proprietary software and/or proprietary languages are to be avoided, in order to prevent vendor lock-in.

**Software as a Service (SaaS)** means that clients need only choose and manage their desired software applications, which reside on an undefined infrastructure and platform distribution. Choice of programming language, load-balancing method(s), software deployment and updates, etc. are all left to the SaaS provider, while the client only has to store their data on the provider’s server for the application to use and serve. SaaS is suitable for generic software solutions, where duplication of efforts is largely inefficient. In terms of usage, it is suitable for software that presents significant demand spikes, where usage is periodic but infrequent, or software that need only be used for a short period of time. SaaS is unsuitable for applications where extremely fast processing or real time response is required, environments where legislation does not permit data being hosted externally, and organizations that already possess an infrastructure capable of fulfilling the software requirements.

### 1.3.5 Cloud service providers

Here, we present a survey of the current market of the three largest cloud service providers, namely, Amazon Web Services (AWS)\(^{12}\), Microsoft Azure\(^{13}\), and Google Cloud Platform\(^{14}\). We have examined and compared the services provided, based on the requirements imposed by our own project, and based on the pricing differences. Since AWS is the oldest and most widely used cloud provider, both Microsoft Azure and Google Cloud use AWS as a comparative baseline of their features\(^{15,16}\).

<table>
<thead>
<tr>
<th>Cloud Provider</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Web Services</td>
<td>eu-west-1 (Ireland)</td>
</tr>
<tr>
<td>Google Cloud Platform</td>
<td>europe-west-1 (Belgium)</td>
</tr>
<tr>
<td>Microsoft Azure</td>
<td>West Europe (Netherlands)</td>
</tr>
</tbody>
</table>

Table 1: For each cloud provider we have looked at, these are the regions that are recommended for the area of Spain.

In order to maintain high throughput, and to avoid sending sensitive data through an insecure channel (the Internet), cloud providers split the resources in different geographical

---

\(^{12}\) [https://aws.amazon.com/](https://aws.amazon.com/)

\(^{13}\) [https://azure.microsoft.com/](https://azure.microsoft.com/)

\(^{14}\) [https://cloud.google.com/](https://cloud.google.com/)


\(^{16}\) [https://cloud.google.com/docs/compare/aws/](https://cloud.google.com/docs/compare/aws/)
regions around the globe. Multi-region infrastructures must be maintained by the customer. Because this adds complexity to the development of our infrastructure, and because it would not add anything useful to the conclusions of the project, we have chosen to use a single region. See table 1 for the recommended regions for our area.

Even though we only use AWS, we have prepared a list of general guidelines on how could the infrastructure, or at least part of it, be migrated to another cloud provider. In order to be able to migrate from one provider to another, we need to define a list of requirements that the new provider must be able to fulfill. Note however that most of these requirements can be fulfilled by means of third party tools, or even with ‘homemade’ solutions; the requirements listed here are not representative of this, as we only look at the first party solutions (from the provider itself). One feature that is bound to be present in every cloud provider is a management interface, both graphical and CLI-based; we consider that we do not have any special requirements in this respect, so any provider’s interface is probably sufficient, the only difference being personal preference.

As stated previously, Amazon Web Services was the earliest provider of a general-purpose, cloud-based computation infrastructure. Amazon offers a whitepaper [1] that presents an overview of their cloud platform.

All resources and services in AWS can be managed from two different tools: the Management Console, and the Command Line Interface. The AWS Management Console is web-based, and can be accessed through the following URL, on any modern browser: https://console.aws.amazon.com. To authenticate, one must use either an Amazon account, or be granted access from another existing account through IAM.
The **AWS Command Line Interface (CLI)** is the other unified tool that developers and administrators use to access, configure, and manage all other services. Since it works from the command line, it is useful in automation tasks, such as scripts, etc. Written in Python, it is usually called from a shell, like Bash on GNU/Linux and macOS, or Powershell on Windows, but also officially supports scripting bindings for Python and Ruby.

In terms of storage, in AWS we have **Simple Storage Service (S3)**. It offers bulk data storage, accessible from any region, application, and/or machine (inside or outside an AWS-backed infrastructure). It boasts extremely high durability (99.9999999999%) and availability (99.99%). We use this service to handle persistent storage of configuration and deployment data, as well as runtime data (inputs, intermediate data, and outputs). The customer can define up to 100 object stores, called “buckets”; within each bucket, an unlimited number of objects (commonly, files) can be placed, each up to 5 TB in size. Internally, S3 is unstructured, but the higher level APIs let clients access objects through a pseudo-hierarchical directory structure, using the common / separator for directories. In terms of cost, S3 is the cheapest persistent storage service AWS offers, and, since it covers all our requirements, this is what we use.

The most fundamental compute service AWS offers is **Elastic Compute Cloud (EC2)**. It is a web service that provides scalable compute capacity in the cloud. Compute units are called EC2 instances; they are virtual machines, with Xen as hypervisor, that can be configured with lots of different hardware configurations (instance types, some can be seen in table 2), and can be loaded with many different operating systems, known as Amazon Machine Images (AMI). All EC2 instances have, at least, a private IP address that belongs to our **Virtual Private Cloud (VPC)**, which, by default, is a /16 CIDR block of logically isolated addresses; instances can also be configured to have a public address as well, so that they can expose services on an address reachable from the Internet. EC2 instances can be configured to belong to security groups with several rules, as well as network ACLs (access control lists), that allow fine-grained control of the firewall rules of everything that enters or exits our private network, enforced by the VPC.

AWS offers the possibility of running spot instances with variable prices, which fluctuate based on supply/demand. The customer decides what is the maximum price per hour they are willing to pay, and when said price is equal or less than that, AWS launches those instances. When the price goes back up, these instances are shut down without previous warning, so the infrastructure must be able to cope with that. Amazon suggests that the price of spot instances is usually between 50% and 90% cheaper than their on-demand counterparts. Spot instances are useful for running the task nodes of a Hadoop cluster.

In terms of persistent storage for EC2 instances, a few options exist. **Elastic Block Store (EBS)** is the most basic one. EBS works at the block level, and can be used for root partitions, as well as additional partitions. EBS partitions can outlive EC2 instances, which means that they can be used by multiple instances, although they can only be attached to one instance at any given time. More expensive instance types have **EC2 Instance Stores**; these are disks which are physically attached to the host that manages the instances. They are ephemeral, which means that as soon as an instance is stopped or terminated, all data is lost. Data is also lost if the underlying physical disk fails or breaks, so AWS documentation recommends using a RAID setup, if more reliability is desired. A more advanced option
<table>
<thead>
<tr>
<th>AWS Instance Type</th>
<th>vCPU (Hyper-threads)</th>
<th>Memory (GiB)</th>
<th>Storage (GB)</th>
<th>Virtualization Type (Xen)</th>
<th>Pricing for On-Demand ($/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.nano</td>
<td>1</td>
<td>0.5</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.007</td>
</tr>
<tr>
<td>t2.micro</td>
<td>1</td>
<td>1</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.0014</td>
</tr>
<tr>
<td>t2.small</td>
<td>1</td>
<td>2</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.028</td>
</tr>
<tr>
<td>t2.medium</td>
<td>2</td>
<td>4</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.056</td>
</tr>
<tr>
<td>t2.large</td>
<td>2</td>
<td>8</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.112</td>
</tr>
<tr>
<td>m4.large</td>
<td>2</td>
<td>8</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.132</td>
</tr>
<tr>
<td>m4.xlarge</td>
<td>4</td>
<td>16</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.264</td>
</tr>
<tr>
<td>m4.2xlarge</td>
<td>8</td>
<td>32</td>
<td>EBS only</td>
<td>HVM</td>
<td>0.528</td>
</tr>
<tr>
<td>m4.4xlarge</td>
<td>16</td>
<td>64</td>
<td>EBS only</td>
<td>HVM</td>
<td>1.056</td>
</tr>
</tbody>
</table>

Table 2: This table shows the EC2 instance types that are relevant to our infrastructure. Many more types exist, optimized for different kinds of workloads, but with the general-purpose instances we can already do everything we need effectively enough, so we have not considered any of them.

As seen here, t2 instances are smaller, and cheaper than m4; the reason is that they are burstable. When using t2 instances, CPU credits are slowly accumulated over time, and can be spent at any time to momentarily increase the baseline compute performance. On the other hand, m4 instances are more expensive, but their base performance is always better than the base performance of t2 instances.

Note that the prices fluctuate over time, region, and/or operating system, we have included them here to illustrate the relative difference of price between the smaller and the larger types.

is Elastic File System (EFS), which is another block-level storage service whose volumes can be automatically grown and shrunk while mounted, as files are added to or removed from it. Multiple EC2 instances can have the same EFS volume attached at the same time.

Multiple types of instances are available, with varying virtual hardware resources, backing virtualization technologies, and optimizations. See table 2 for a reference of the available types of instances, with their corresponding characteristics, that are relevant to our project. Full information is available at the AWS EC2 documentation\(^\text{17}\). Note that the prices presented here only account for the time the instance is running, and do not include bandwidth and/or any other additional services these instances may utilize.

A much simpler alternative to EC2 exists, known as Lambda. It allows developers to run code in Python 2.7 or Node.js without the burden of a EC2 instance or container. Users do not interact with the operating system; instead, they just upload code, and add one or more events that will trigger this function to run; many of the services in AWS can be configured to run user-defined Lambda functions when specific operations are performed. Since there is no mechanism to ‘bootstrap’ the environment, in case external dependencies are required, the developers must package the environment themselves. On Python, this means creating a virtualenv, installing the dependencies, and uploading it to AWS as a ZIP archive. However, there is a very important limitation to Lambda: the script can only run for a maximum of 300 seconds before it is killed. This is a kind of PaaS, and is quick and easy to use, but it is not suitable for computationally intensive tasks, nor for long running ones. This service can be used to run some or all of the infrastructure in a serverless

\(^\text{17}\)https://aws.amazon.com/ec2/instance-types/
configuration, meaning that, instead of having long running instances, we would have an isolated programming environment that can either be run manually, or automatically, in response to events.

There is another kind of compute service offered by Amazon, intended to be used as the platform for computationally intensive, long running tasks, based on Apache Hadoop. This service is **Elastic MapReduce (EMR)**, a managed Hadoop cluster provisioned and terminated by AWS; the customer needs only specify the type of the instances, how many of them to start with, and their IAM permissions. Then, the cluster can be launched in two ways: in cluster mode, or step execution mode. In cluster mode, a set of applications is installed on the specified EMR nodes, and when it finishes launching, the customer can interactively submit work to it; termination of the cluster is also the responsibility of the customer. In step execution mode, on the other hand, a list of programs are specified, and then the cluster runs them sequentially; when all steps have been executed, the cluster is automatically terminated. At each step, paths to the programs and input and output directories have to be provided.

![Figure 4: Here is a screenshot of the AWS management console, in the page for launching and provisioning an Elastic Map Reduce (EMR) cluster. As we can see, many different Hadoop-related frameworks are available, that support lots of different programming techniques and paradigms, and are optimized for diverse kinds of workloads. In this screenshot, we can see the release 5.1.0 of emr is selected; the Hadoop ecosystem is constantly evolving, so newer versions of the software listed here are almost certain to be released in the future.](image)

Regardless of the launch mode (cluster or step execution), a few types of programs are supported in EMR. The simplest way is to directly write the `map` and `reduce` functions of MapReduce, each in the form of a streaming program: this can be any executable file (binary or text) that reads from standard input and writes to standard output; any programming language that can do that, can be used to create a streaming program; choice of the language(s) to use depends on the kind of data formats used, and the kind of processing being performed. However, it may be desirable to use one of the multiple tools, native to Hadoop, that implement an abstraction layer over the MapReduce algorithm, like Apache Hive\(^{18}\), a data warehousing tool with a SQL interface that allows users to extend the language by means of ‘user defined functions’, or UDFs, or Apache Pig\(^{19}\), a high-level programming language, specifically geared towards data analysis, coupled with support for massive parallelization, and aggressive optimization; UDFs are also supported in Pig.

---

\(^{18}\)https://hive.apache.org/
\(^{19}\)https://pig.apache.org/
There are also more complex frameworks like Apache Spark\(^{20}\), a general-purpose compute engine with implicit parallelization, suitable for clusters, and programmable in many languages (through APIs); Spark can be stacked on top of Hadoop to use HDFS as storage and MapReduce for computation. Many other Hadoop-related tools and frameworks are available in EMR’s Hadoop distribution, as shown in figure 4.

EMR uses EC2 instances as Hadoop nodes. We can give our SSH key to the cluster, so that we are able to administrate the instances running our EMR cluster; we can also configure every other feature of EC2, through the EMR API. There are three kinds of EMR nodes: master nodes, core nodes, and task nodes. The master node is an instance, of which there is typically only one per cluster, that coordinates the distribution of data and tasks among the other nodes, and monitors the health of the cluster. Core nodes are slaves that store the HDFS, and run tasks. Task nodes are slaves that only run tasks; they are completely optional. The master and core nodes are run on on-demand EC2 instances, whereas task nodes are usually run as spot instances, since they do not hold any HDFS data, and therefore can be randomly terminated without a major fault or loss of data.

A key aspect of EMR is that it is highly optimized for using S3 as the backing storage for HDFS; this configuration is known as EMRFS. However, local HDFS can still be used, with the difference that, when the cluster gets terminated, all data that was not copied out of HDFS will be lost.

In AWS, the basic service for distributed message queuing is Simple Queue Service (SQS). It provides a simple, efficient way to pass messages from one EC2 instance to another, while keeping them as decoupled as possible.

All of the services in AWS are protected by Identity and Access Management (IAM), an important component for administrators, that handles privilege granting and denying to all of the users (human or machine). Administrators can specify the authentication methods for every user, including passwords, cryptographic keys, and Multi-Factor Authentication (MFA). IAM controls the granularity of authorization at three levels: users, groups, and roles. IAM users are individuals who own an authentication token (password, cryptographic key, or MFA). They can be authorized by the ‘root’ account to perform specific tasks in specific services, with multiple levels of restriction to data usage. We map IAM users to real (human) users, so that each can have their organizational requirements. These can be aggregated into IAM groups, which are collections of IAM users with similar permissions. The purpose of IAM groups is to allow easy authorization to a number of users that are allowed to perform certain common tasks; for example, the Administrators group may have a more permissive set of policies than Developers. A group may contain multiple users, and a user may belong to multiple groups. IAM Roles are similar to IAM users, but without any attached credentials (passwords, keys, etc.). This is used to configure the level of authorization of the different components of our infrastructure (i.e. EC2 instances).

The actual definitions of who can access what are IAM policies. These are JSON documents that can specify which users, groups, and/or roles can access (or not) which services, or even which API calls for services. AWS provides, by default, sane, but permissive policies for almost everything; users can also write their own, and attach these custom policies.
to existing users, groups or roles. A policy document contains an effect, which is either **Allow** or **Deny**, a list of actions, which are individual operations that the various AWS services perform (for example, `s3:ListBucket` lets a role see the contents of a bucket), and a list of resources, which are the AWS resources which are affected. Both actions and resources can be specified with wildcards, which potentially matches multiple names at once.

The services presented here only cover a tiny fraction of what one can do with AWS; there are many more services for compute, storage, and networking, that are differently optimized for several use cases, a lot of which are irrelevant to the purposes of our project. Even so, what we have implemented could have been done with different services to accomplish the same, so it must be kept in mind that this is only a preliminary implementation, or a proof of concept, that can be highly optimized to be more cost-effective, or faster, or more powerful.

All three providers offer a **free trial** period, in order to let new developers get acquainted with the vast array of different kinds of service, free of charge. However, the restrictions imposed in this period greatly differ. AWS offers a period of one year after the initial sign-up, allowing access to a number of different services, but imposes monthly usage restrictions; for example, a single EC2 `t2.micro` instance can be run for free for the whole month, and two instances for half a month each. Any excess will be charged to the credit card provided at the time of sign-up. Google, on the other hand, offers 300$ worth of credit, to be spent during the 60 days following the date of sign-up. All services can be used, as if the user had been paying. Microsoft, similar to Google, offers 200$ worth of credit, and no usage restrictions are imposed while the initial credit remains. However, Azure, in contrast with Google, allows subscribers to continue using some of their services for free after the initial credit is spent, within certain limits, regardless of the subscription plan, for an undefined period of time. For our purposes, the one year period of AWS Free Tier was the most suitable, as it allowed the most time to both learn the general usage of AWS, and for development and testing; in this regard, what Google and Azure offered was insufficient for an unexperienced developer.

**Microsoft Azure** is a cloud computing platform created and managed by Microsoft. It runs Microsoft technologies for almost everything, such as Windows, SQL Server, IIS, etc., but it also offers its users the possibility to use Linux on virtual machines. Only a handful of Linux distributions are officially endorsed by Microsoft, which means that, in order to qualify for their Service Level Agreement for virtual machines, customers must use one of these endorsed distributions.

Azure follows the IaaS/PaaS/SaaS division explained in section 1.3.4 more closely than AWS. Their compute toolset is split into three categories: Virtual Machines, Cloud Services, and App Services. In Azure Virtual Machines, the instance types follow a naming scheme similar to that of AWS EC2. Table 3 shows the instance types that are similar to the ones on table 2, even though there are many more. Azure Virtual Machines is the option we would choose as the option most compatible to AWS EC2. For storage, Azure presents two options: Blob Storage, which is REST based, and unstructured, and File Storage, which works over SMB v3, a networked file system. Here, we can choose any of the two, as the two are similar to S3; even though S3 implements a directory hierarchy on the
## Table 3

<table>
<thead>
<tr>
<th>Azure Instance Type</th>
<th>vCPU (Hyper-threads)</th>
<th>Memory (GiB)</th>
<th>Storage (GB)</th>
<th>Virtualization Type</th>
<th>Pricing ($/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard_A0</td>
<td>1</td>
<td>0.75</td>
<td>20</td>
<td>Hyper-V</td>
<td>0.018</td>
</tr>
<tr>
<td>Standard_A1</td>
<td>1</td>
<td>1.75</td>
<td>70</td>
<td>Hyper-V</td>
<td>0.047</td>
</tr>
<tr>
<td>Standard_A2</td>
<td>2</td>
<td>3.5</td>
<td>135</td>
<td>Hyper-V</td>
<td>0.094</td>
</tr>
<tr>
<td>Standard_A3</td>
<td>4</td>
<td>7</td>
<td>285</td>
<td>Hyper-V</td>
<td>0.188</td>
</tr>
<tr>
<td>Standard_A4</td>
<td>8</td>
<td>14</td>
<td>605</td>
<td>Hyper-V</td>
<td>0.376</td>
</tr>
<tr>
<td>D1 v2</td>
<td>1</td>
<td>3.5</td>
<td>50</td>
<td>Hyper-V</td>
<td>0.073</td>
</tr>
<tr>
<td>D2 v2</td>
<td>2</td>
<td>7</td>
<td>100</td>
<td>Hyper-V</td>
<td>0.146</td>
</tr>
<tr>
<td>D3 v2</td>
<td>4</td>
<td>14</td>
<td>200</td>
<td>Hyper-V</td>
<td>0.293</td>
</tr>
<tr>
<td>D4 v2</td>
<td>8</td>
<td>28</td>
<td>400</td>
<td>Hyper-V</td>
<td>0.585</td>
</tr>
<tr>
<td>D5 v2</td>
<td>16</td>
<td>56</td>
<td>800</td>
<td>Hyper-V</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 3: In this table we have the instance types in Azure that are roughly equivalent to the AWS instance types we have considered, in table 2. Like with the AWS EC2 instance types, the prices shown here are bound to fluctuate over time, region, and operating system, and are shown here only as a comparison with the current prices in AWS.

## Discussion

Frontend, it is actually non-hierarchical, like Blob Storage. For access and identity management, Azure’s approach uses Active Directory as their system. As a replacement for EMR, Azure includes in their cloud platform a comprehensive set of ‘big data’ analysis projects, such as Apache Hadoop with Pig and Hive, Apache Spark, Apache HBase, etc. For distributed messaging, Azure offers two services: Service Bus, and Event Hub. The former can be equated to AWS’s SQS, and the latter to SNS.

Google’s compute platform is **Google Compute Engine**. While, in some aspects, this works exactly the same as the other two providers’ compute platforms, we find the biggest difference on Google’s: it lets customers programmatically define custom instance types, with an arbitrary combination of vCPUs and RAM. For storage, we find a number of solutions, somewhat similar to those seen in the other providers; Google Cloud Storage is a cheap and unstructured blob storage service, like S3 in AWS, and Blob Storage in Azure; however, buckets in Google Cloud Storage can belong to one of four storage classes, with varying levels of availability versus cost per GB stored. Their solution for a managed Hadoop cluster, like AWS’s EMR, is called Dataproc; like AWS, it supports Spark, Pig, and Hive. In Google Cloud, there is a single, unified distributed messaging service: Pub/Sub. This service is similar to AWS’s SNS, or Azure’s Event Hubs, but it can also emulate the functionality of a queue. Finally, Google’s solution for Identity and Access Management (IAM) is almost identical to AWS’s IAM.

What Azure and Google Cloud offer is conceptually very similar to AWS, so the basic design we present in sections 3.1 and 3.2 should, in principle, be completely portable to those services. Even though their official APIs are obviously different, they all offer Python bindings, and Ansible has native support for both Azure and Google Cloud; basically, if we were to port the infrastructure to one of these two services, we would have to rewrite all of the API calls to those services, even though the underlying logic is the same. A more robust software solution could have isolated the specific API calls from the abstract logic that drives the data along the pipeline.
2 Materials and methods

2.1 Development environment and tools

A few tools have been used during the development and testing of the infrastructure. Most code is written in Python, and version control is done with Git. The IDE used for programming in Python is PyCharm. For interacting with the AWS services, we use their official command line interface tools, Python bindings, and their web app. To manage the infrastructure, we use Ansible, which also lets us define it as simple text files, as will be explained in appendix C.

As for the operating system, we use and recommend GNU/Linux in the development environment. Even though most software mentioned in this section is cross-platform compatible, we find that GNU/Linux is the easiest to integrate with what we use. Other than the development software listed in this section, only a modern web browser is required, with support for Javascript, which is necessary for accessing the AWS Management Console, and CARTO.

Python\textsuperscript{21} is a scripting language specification, developed and maintained by the Python Software Foundation (PSF). In our project, Python is used to write the scripts that implement and control the behaviour in virtual machines, and to the components that are not integrable natively through AWS. It is a good choice for us, because it is a very high level programming language, with features such as automatic garbage collection, dynamic weak typing, duck typing, etc., since we are not interested at all in low-level memory management, which can often lead to dangerous memory corruptions, such as buffer overflows, irrecoverable null-pointer dereferences, etc. Also, it is highly portable (the only parts that are not fully portable are the ones that deal with the underlying operating system, but even those are quite standardized).

Python dependencies are handled with virtualenv\textsuperscript{22}. This tool permits the developer to locally maintain multiple sets of dependencies and Python interpreters, so that installing or removing packages from one environment does not affect another. Even though it is possible to use virtualenv for the production environment (i.e. in EC2 instances), we do not, and install Python dependencies directly, or system-wide. Python packages are installed with pip, the Python package manager. With this tool, we can also prepare a list of installed dependencies in the current environment, be it system-wide or virtualenv, with pip freeze, and store the result in a file called requirements.txt. This file will contain a list of package names, together with versions. This enables us to pass isolated Python environments from one machine to another, in the form of a simple and compact plaintext file.

Even though virtualenv provides the bare minimum to work with environments, it is recommended to use virtualenvwrapper\textsuperscript{23}, which includes a few command line tools for easier management. When we want to replicate the environment in another place, we start by cre-

\textsuperscript{21}https://www.python.org/
\textsuperscript{22}https://virtualenv.pypa.io/
\textsuperscript{23}https://virtualenvwrapper.readthedocs.io/
ating a new environment with `mkvirtualenv --python=/usr/bin/python env-name`. The `--python` option specifies which version of the interpreter will be used; `env-name` can be replaced by any other name. Once the environment becomes available, we can activate it with `workon env-name`. Now, any Python program, including the interpreter and packaging tools, will only have available those versions of packages that are installed inside of the isolated environment. On directories with Python source code, there is a `requirements.txt`, which contains a list of packages on which the code depends on, to indicate that a new virtualenv should be created to run the code in that directory. `pip` can be used to install a `requirements.txt` with `pip install -r requirements.txt`. To return to the regular system-wide environment, `deactivate` can be used, with no arguments.

We use the RPM package manager to package the software that we distribute to our machines, as will be explained in appendix B. In order to fully set up this program, the following directory structure has to exist in the user’s home directory:

```
mkdir -p ~/rpmbuild/{BUILD,BUILDROOT,RPMS,SOURCES,SPECS,SRPMS,tmp}
```

Listing 1: Command to create the directory structure for `rpmbuild`, in Bash

For programming, we have used PyCharm\(^24\), a cross-platform Integrated Development Environment (IDE) for Python, developed and maintained by JetBrains, a privately-held Czech company. Even though there are Professional and Educational versions available, we use the Community Edition, which is free and open source (the source code can be found at GitHub \([9]\). While it does not provide all of the features and benefits in the Professional Edition, the authors have deemed it sufficient for our purposes; therefore, this is the version that is used. It should be noted that, even though the main code of our infrastructure is Python, we have had to edit code in many different languages; for this reason, it is advisable to have available a powerful text editor with modern features like syntax highlighting support; `vim`, `emacs`, or `nano`, are all good examples. This can also be applied to Python programming; however, PyCharm has considerable benefits, like an integrated graphical debugger, that make it more convenient to work with Python.

Git\(^25\) is a free and open source distributed version control system (VCS), released under GPL-2. Some of its characteristics include support for non-linear development (i.e. branching and merging), de-centralized version history allows distributed development with several contributors, high scalability for very small and very large projects alike. For our project, this is a great choice of VCS, because the authors are already familiar with it, and because Bitbucket, our online repository hosting service, supports it. The URL to the source code repository, where all of our code is stored can be found in listing 2.

```
https://bitbucket.org/lluisisern/hpopt
```

Listing 2: URL to the Git repository with our code

We have chosen AWS as the cloud provider for this project. As explained in section 1.3.5, AWS is managed, mostly, through two unified interfaces: the management console, accessed through the web browser, and the command line interface, `awscli`, from a terminal emulator with a shell, like Bash, or from the Python scripts, using the official `boto3` library.

\(^{24}\)https://www.jetbrains.com/pycharm/

\(^{25}\)https://git-scm.com/
For orchestrating our infrastructure, we use Ansible\textsuperscript{26}, released under GPL-v3. Ansible is an infrastructure management tool that unifies provisioning, software deployment, and configuration management under one, agentless architecture. Also, by default, Ansible is installed with a static inventory, which means that we must know in advance which IP addresses will be managed from which computer. This is not feasible for cloud environments such as ours. There is another option, which involves creating a dynamic AWS inventory for Ansible to use. According to the official documentation, this is accomplished by copying an official Python script\textsuperscript{[15]} to the \texttt{/etc/ansible/hosts} path, and marking it executable with \texttt{chmod +x}. Now, when Ansible is told to run a module, it will query AWS and remember the addresses of all instances in EC2, so that they are available in subsequent runs (querying AWS in this manner can take a lot of time, and thus performance is improved by caching these requests). On appendix C there are more details on Ansible and how we use it.

\section*{2.2 Data visualization with CARTO}

For generating reports, and presenting data back to the user, we have chosen CARTO\textsuperscript{27} (previously known as CartoDB), a web-based geographical information system (GIS). This is used at the end of our pipeline, after simulating the models. CARTO can be used for free by registered users, but businesses and enterprises can purchase licenses that include larger bandwidth quotas, service-level agreements, technical support, etc. The service is aimed at two kinds of users: builders, and viewers; builders create and distribute data to viewers, which can be organizational partners, clients, or any other user (registered or not in CARTO).

There are three kinds of pricing plans available: free, personal, and enterprise. Free users can only use some of the APIs, cannot have private datasets, and have no technical support. In our case, the free plan sufficed for the development of the project, but for a production environment, one of the paid options would be a much better idea.

The most immediate way to work with CARTO is through its web platform, accessible after creating a user, and logging into their website. Each user has a profile page, or dashboard, that shows the maps and datasets that this user owns. Datasets are raw tables, with rows and columns, that can be uploaded through the web app, or with the API (as explained later in this section); if the dataset has columns with geocoding information, then CARTO can trivially replicate points and lines (or other shapes) on top of a generic map, and no other work is required to visualize the data. However, users can do more complex things by creating the maps themselves, and then include one or more datasets in multiple layers, optionally with alternative map tiles, or even user-defined graphical styles (with a CSS-like language). In our infrastructure, this is used as the GUI for displaying results.

Other than the web interface, CARTO exposes a set of APIs, with which users can interact with the service in an automated way, such as by executing scripts, or, in our case, by running jobs in our pipeline. Instead of having one big API with multiple responsibilities, there are nine different APIs, in a few languages, and with very different functionalities.

\textsuperscript{26}https://www.ansible.com/

\textsuperscript{27}https://carto.com/
Figure 5: Here we see a screenshot of the CARTO web app, on the dataset viewer. On the top row there are buttons for navigating the entire website. On the middle region, the interface is vertically split in half, with the table reader on the left, and the map viewer on the right. On the bottom row there are buttons for visiting the owning user’s profile and other datasets, and for interacting with the active dataset (downloading the source, performing API calls, or editing it).

The disadvantage of this is, obviously, that users have to learn nine APIs, instead of just one. However, the simpler implementation of the APIs, and the flexibility offered by the multiple languages used to access the API appears as a good design choice. Even though CARTO exposes these services in their own infrastructure, the APIs themselves are open source, and their source code can be accessed on GitHub [3].

Calls to the APIs are authenticated with private, per-user keys, which get automatically generated when the users are created; separate policies can be specified for each user, and public datasets have their own policies for public users. Additionally, in on-premises distributions of CARTO, the superadmin user of PostgreSQL is available. In our infrastructure, the CARTO credentials are managed in a simplistic way: the private API key is stored in a plaintext file, inside the deploy bucket; when a component of the infrastructure needs it, it simply downloads it and stores it in a private directory (inside /etc/hpopt).

The datasets are stored internally as PostgreSQL tables, but can be manipulated with the SQL API, which encapsulates SQL queries into REST HTTP calls. By sending any PostgreSQL statement to this API, using any of the standard verbs (e.g. UPDATE, SELECT, INSERT, etc.), users can manipulate their data as if they were working with a normal database. Files can be uploaded to CARTO with the Import API, through REST HTTP calls, that get interpreted and automatically converted to SQL tables; many file formats are supported. In our infrastructure, for uploading the results of simulations, any of these options is valid, although the simpler SQL API is, probably, the best option for this particular purpose; the Import API could also be useful, if the table(s) are already in a format understood by CARTO.

The main library for visualization is CARTO.js, in Javascript. This enables users to access CARTO resources, like datasets and maps, from their own website, and modify them through API calls, and also create visualizations of datasets. The graphical style of the visualizations is specified with a CSS-like language, known as CartoCSS. Alternatively, the
Maps API, also in Javascript, can be used to generate maps. Additionally, Torque.js offers the possibility of animating the data on top of existing visualizations, to create things like timelapses. Finally, there is a mobile SDK for Android (in Java), iOS (in Objective-C or Swift), and Windows Phone (in C#), that can be used to create custom map applications through an interface similar to the CARTO.js API.

Then, there are two services for data analysis: the Data Services API can be used to compute several functions on geographical data, like isolines, routing (for navigation), and so on (there are strict monthly quotas on free and personal licenses); the Data Observatory, only available to enterprise users, provides access to a large catalog of pre-analyzed data on demographics on several countries around the world.

As we just showed, there are many more services in CARTO, beyond the basic GIS rendering that we require. In the future, a closer integration between our infrastructure and this service could be beneficial, but, for now, the basic functionalities of the free license are enough.

### 2.3 Sample dataset

For development and testing purposes, a small dataset is used. It consists of health care data from all regions in Catalonia [4], and is published annually, by the Observatori del Sistema de Salut de Catalunya, Generalitat de Catalunya.

It is important to note that this dataset has only been used to test that the software being implemented is working as intended, since it is not large enough to benefit from a distributed Hadoop cluster (at least, something in the order of GBs).

This dataset consists of a number of tables with indicators, on several sectors in Catalanian healthcare: hospitals, primary care, long-term care, mental health and addictions, public health, and research. For each of the sectors, a spreadsheet file is given, in Microsoft’s Office Open XML format, or XLSX. Historical data is available since 2009. At the time of writing, the latest published edition is from 2015. All of the data is aggregated, and should therefore be completely anonymous. Finally, it must be noted that the datasets are available only in Catalan; mentions to them in this document have been translated.

The spreadsheet on the hospital sector has a single sheet, with 14 columns, and 5726 rows. The first column indicates the table or topic to which the metric belongs to. The next five columns contain the identifying information about the hospital in question: a numeric identifier (unique to the hospital), the name of the hospital, the region, the name of the organization in charge of the hospital, and a regional numeric identifier. The seventh column contains the name of the indicator, the next its value, and the next the total sum of all entries. The last five columns are used to display additional information about some of the metrics, but most of the values under this column are blank, so we have not considered these columns. In total, there are 81 metrics in 10 different tables, referencing 66 different hospitals. On table 4 we can see all of the indicators that can be found in the hospitals dataset, grouped by topic.
The first table, *General data*, contains indicators on the number of admissions through various ways, and also other generic metrics, like the case mix index, among others. On the second table, *Suitability*, we find percentages on the distribution and nature of emergency cases in general, but also specifically on pneumonia, heart strokes, and maternity. The third table, *Satisfaction*, contains percentages of the quality of the services and installations, as perceived by the patients, as well as an overall score, or satisfaction index. On the fourth table, *Effectiveness*, we find metrics on mortality, readmissions, and organ donors. The fifth table, *Safety*, contains indicators on bacteremia and other infections caused by surgery. The sixth table, *Efficiency*, has indicators on the length of patient stays at the hospital. The seventh table, *Economical data*, has indicators on liquidity, solvency, debt, productivity of human resources, and similar others. The eighth table, *ICT*, shows the amount of contributions to HC3, the shared medical history of Catalonia. The ninth and last table, *Education*, shows the average academic performance of the top three interns for each hospital.

<table>
<thead>
<tr>
<th>Table</th>
<th>Indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td>General data</td>
<td>Number of SISCAT hospitalizations</td>
</tr>
<tr>
<td></td>
<td>Number of conventional hospitalizations</td>
</tr>
<tr>
<td></td>
<td>Number of medical hospitalizations</td>
</tr>
<tr>
<td></td>
<td>Number of surgical hospitalizations</td>
</tr>
<tr>
<td></td>
<td>Number of interventions in major outpatient surgery</td>
</tr>
<tr>
<td></td>
<td>Case mix index</td>
</tr>
<tr>
<td></td>
<td>Amount in the CatSalut contract (in current €)</td>
</tr>
<tr>
<td>Suitability</td>
<td>Emergency admissions (%)</td>
</tr>
<tr>
<td></td>
<td>Emergencies admitted (%)</td>
</tr>
<tr>
<td></td>
<td>Emergencies of MAT levels 1, 2 and 3 (%)</td>
</tr>
<tr>
<td></td>
<td>Pneumonia cases without complications (%)</td>
</tr>
<tr>
<td></td>
<td>Severely premature births (%)</td>
</tr>
<tr>
<td></td>
<td>Admissions on hospitalization at home (%)</td>
</tr>
<tr>
<td></td>
<td>Cessarean sections (%)</td>
</tr>
<tr>
<td></td>
<td>Time to admission of heart failure (on average)</td>
</tr>
<tr>
<td></td>
<td>Time to surgery on hip fracture (days on average)</td>
</tr>
<tr>
<td>Satisfaction</td>
<td>Satisfaction index</td>
</tr>
<tr>
<td></td>
<td>Loyalty index (%)</td>
</tr>
<tr>
<td></td>
<td>Time waiting in the queue (%)</td>
</tr>
<tr>
<td></td>
<td>Disposition to be listened to and taken care of (%)</td>
</tr>
<tr>
<td></td>
<td>Felt safe (%)</td>
</tr>
<tr>
<td></td>
<td>Slept well at night (%)</td>
</tr>
<tr>
<td></td>
<td>The food at the hospital (%)</td>
</tr>
<tr>
<td></td>
<td>Comfort at the room (%)</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Total mortality after 30 days (selected diseases)</td>
</tr>
<tr>
<td></td>
<td>Mortality after being discharged (selected diseases)</td>
</tr>
<tr>
<td></td>
<td>Mortality after 30 days (STEMI)</td>
</tr>
<tr>
<td></td>
<td>Mortality after being discharged (STEMI)</td>
</tr>
<tr>
<td></td>
<td>Mortality after 30 days (CHF)</td>
</tr>
<tr>
<td></td>
<td>Mortality after being discharged (CHF)</td>
</tr>
<tr>
<td></td>
<td>Mortality after 30 days (CVA)</td>
</tr>
<tr>
<td></td>
<td>Mortality after being discharged (CVA)</td>
</tr>
</tbody>
</table>

*Continued on next page*
### Effectiveness
- Mortality after 30 days (femoral neck fracture)
- Mortality after being discharged (femoral neck fracture)
- Percentage of pulmonary embolism
- Readmissions after 30 days (selected diseases)
- Readmissions after 30 days (complications on diabetes)
- Readmissions after 30 days (COPD)
- Readmissions after 30 days (CHF)
- Mortality on emergencies (%)
- Mortality after 3 months in cases of DVT/CVA
- Recovery after 3 supervised months in cases of DVT/CVA
- Valid bodies donated to OCATT (%)
- Familiar negatives on OCATT (%)
- Index of transplanted organs per valid OCATT donor
- Valid asystolic donors on OCATT (%)
- Live donations on live renal transplantations (%)

### Safety
- Catheter-related global bacteremia (per 1000 days)
- Catheter-related central bacteremia (per 1000 days)
- Parenteral catheter-related bacteremia (per 1000 days)
- Infection on localized elective rectal surgery
- Infection on localized elective colon surgery
- Rate of adequate prophylaxis of colo-rectal surgery
- Mortality in low-mortality DRGs
- Mortality on patients who developed complications
- Pressure ulcers

### Efficiency
- Major outpatient surgery (%)
- Rate of standard outpatient surgery
- Average stay on hospital care
- Average stay on heart failure
- Average stay on femoral neck fracture
- Average stay on CVA
- Average stay on COPD
- Rate of standard operation

### Economical
- Profitability of exploitability income (%)
- Generated cash flow (%)
- Profitability (%)
- Solvency (%)
- Liquidity (%)
- Indebtedness (%)
- Income per standard unit (%)
- Cost per standard unit (%)
- Productivity of human resources (without surrogates)
- Weight of hospital attention (%)

### ICT
- HC3 publications

### Education
- Average marks of top three interns (medicine) who chose the hospital

*Continued on next page*
### 2.4 Dataset processing strategy

In this section, we discuss the strategy that we have designed for communicating and synchronizing the different parts of our infrastructure, following the linear pipeline that we introduced in section 1.2.

On our AWS-based infrastructure, we have grouped the various resources around the pipeline stages. Each stage has an SQS queue, and an S3 bucket. The compute resources used in each stage vary: the first stage has an EC2 AutoScaling group of identical instances, the second has a single \texttt{t2.micro} EC2 instance that dynamically creates ephemeral EMR clusters, and the third and fourth stages share a single \texttt{t2.micro} EC2 instance. Each of these EC2 instances has an IAM role associated to it, which lets it receive messages from the queue of its stage, and send messages to the queue of the next stage, read/write to its bucket, and read from the previous bucket; each message in SQS represents a single job.

The frontend has a \texttt{t2.micro} instance, which has exclusive access to an SQS queue, and an S3 bucket of its own. The bucket contains JSON-formatted task descriptions for all of the stages; therefore, it has access to send messages to any of the other queues. The frontend also listens the \texttt{status} queue, to which backend EC2 instances send messages, when they finish jobs (successfully or not); this is not, strictly speaking, necessary for processing the data; nevertheless, it can be very useful, since it is the only way the frontend can know when jobs end, because it does not have read access to the S3 buckets where the intermediate results get stored.

This design has a number of advantages: first, since the compute resources of each stage are well-defined, any given stage can run in isolation, even if all of the other stages are currently down; second, because the long-running tasks do not block the communication mechanisms, each stage can scale in and out, independently of the others; and third, since each job is a logical unit, every stage can accept and run an arbitrary number of jobs, at any given time.

However, the weakness of this is that there is only one possible way for the data to flow; this is a simple design that works, but it can be improved: if we opt for less strict design, our infrastructure can support more complex communication flows, like feeding data back to previous stages, conditionally selecting stages to run, and so on. With AWS, we could accomplish this by using, instead of SQS, SNS. The difference between the two is that, with the latter, a single message can be received by multiple subscribers, that get a notification pushed to them, instead of having to continually poll the queue, like in SQS.
2.5 Agent-based modelling with Pandora

Agent-based modelling (ABM) is a paradigm for simulating the interactions of multiple autonomous entities, and studying the dynamic, emergent behaviours produced by it. For example, this can be used to model human behaviour in large-scale dynamic systems, although, in principle, almost any scientific endeavor can benefit from ABM. The model is simulated on a space, usually two-dimensional, and discrete, but it can be continuous as well; each agent is always positioned somewhere in the space. After the ‘world’ gets populated with agents, some sort of scheduling mechanism takes over, and begins executing the actions of the agents in a sequential manner; the simplest way to do this is by iterating over all agents (randomly) at each step of the simulation, but more efficient methods attempt to introduce parallel programming techniques into it. The agents can interact among themselves, gather knowledge about the world and/or other agents, set their own goals, and, in general, make autonomous decisions. On top of that, information has to be gathered about the state of the world at each step: at minimum, the position of all agents, but more complex agents that have special attributes can be included as well. Additionally, metrics can be gathered about the global state of the world, and not about the individual agents.

As the article by Xavier Rubio-Campillo [21] suggests, the user groups, and, consequently, the platforms, have bifurcated into two distinct groups: one with simpler models, smaller learning curve, and fast prototyping, and another group with more complex models, tougher to learn, but actually scalable through parallel programming techniques. Their article presents Pandora [22], an open source ABM framework that bridges the gap between these two groups, by abstracting the model definition and logic away from the execution platform. On top of this, Pandora offers a twin interface, in Python for rapid prototyping, and in C++ for an equivalent, more efficient version. It is also able to switch between a sequential environment to a parallelized one without modifying any code. This, effectively, enables developers to quickly code a prototype for testing, and then translate it to the more efficient version, where we can execute it in an HPC environment, to generate a much larger dataset.

In Pandora, agents are defined as a subclass of Engine::Agent (in C++). These classes have to explicitly declare which attributes have to be serialized, as well as implement their logic in three methods, which can modify own and others’ attributes, and so on; this can be ignored, for simplicity of programming, at the cost of possible losses in efficiency. Pandora can run models on simple, discrete 2D spaces, but it also provides full support to GIS-aware models, by using the Geospatial Data Abstraction Layer (GDAL)28 library for importing rasters in a multitude of formats. In terms of schedulers, we find two very different strategies: Engine::OpenMPSingleNode, based on OpenMP, suitable for a standalone machine (possibly with a multicore processor architecture), and Engine::SpacePartition, based on MPI, suitable for computationally expensive models, but sub-optimal for simple ones, since agent data has to be shared across multiple nodes, whose space partitions (intentionally) overlap, and this incurs a high overhead. Both schedulers are able to distribute execution of a single model, as well as running multiple models at once, in parallel; so this is certainly very interesting for performing the parameter exploration required for the last stage of the pipeline. Other than performing the actual execution of the models, Pandora comes with a set of tools that are useful, mostly, for the posterior analysis of the results.

28http://www.gdal.org/
obtained. Pandora includes a separate GUI program, named Cassandra, which can be seen in figure 6. With it, users can perform a number of different data visualizations, and then perform some analytical functions on it; it can also explore the parameter space, through a function in it known as the Laboratory.

Pandora uses the Hierarchical Data Format 5 (HDF5) [18] library for serialization; this is a high-performance, data format, with support for parallelized I/O, that is able to handle massive datasets in an efficient manner. This fits well in an HPC environment; however, this comes with a considerable overhead, and thus may not be justifiable for smaller, simpler models that run well enough on a sequential environment. The state of the model and its agents is serialized in this format, so we may need to be able to inter-operate with this format for generating the final results: this depends on whether we want the raw data obtained, or just the results of the posterior analysis; in the latter case, we do not need to copy the large HDF5 file generated by Pandora, and thus we can save a lot of bandwidth.

2.5.1 Baseline performance comparison

We wanted to see whether the overhead of Pandora is worth it for the volumes of data that we are going to be processing, so we have compared its performance against another ABM library that is simpler to program, but has no ability to parallelize: Mesa [6] [7], released under Apache v2.0. This library falls in the first category of platforms described in Pandora’s article: easy to learn and use, but unable to use HPC resources. For this reason, there are some fundamental differences between the two. Mesa collects the data in-memory only, therefore all serialization must be programmed by hand, whereas Pandora implicitly serializes all simulations in an HDF5 file. Mesa’s agents have a single method, \texttt{step()}, whereas Pandora’s have three: \texttt{updateKnowledge()}, \texttt{selectActions()}, and \texttt{updateState()}.
the reason for this is that Pandora is able to execute agents’ actions in parallel, following
an explore - decide - apply cycle, where the two first methods only evaluate their state,
and the world is changed in the third method. While Pandora uses its GUI component,
Cassandra, for all visualization (as described previously in this section), Mesa creates a
lightweight HTTP server, with a modular Python API, and an HTML5/Javascript frontend.
In Pandora, there is a special kind of ‘agent’ that serializes one value per position in space,
but does no actions, so it can be more aggressively optimized, called a raster; this has no
equivalent in Mesa, since an agent with an empty step() is essentially the same, and it
cannot be optimized like Pandora can. Finally, Mesa has no native support for GIS, but it
can run on a continuous space, so it could be implemented through an external library. In
order to determine which library to use for which kind of workload, we have done a series
of performance benchmarks, as explained in section 3.4.

Among the examples of Pandora, we find the random walkers model [24], programmed in
the C++ interface. The world of this model, in the class RandomWorld, initializes a dynamic
raster, called resources, with a random integer between zero and five, in every position of
space. The world also creates the agents, found in class RandomAgent, at random positions
throughout the space. The agents, at every step, execute the actions MoveAction, which
moves the agent one step in a random direction, and EatAction, which adds the resources
found at the current position to the agent’s, and sets the global raster value to zero (so that
no other agents can take it); the agents also lose one unit of resources at every step, and
they get removed from the world when their last unit is taken.

We have created a simplified version of this model, and we have ported it to the Python
interface of Pandora, and also to the Mesa library. We have used this model to establish
a baseline comparison of the performance of each of the libraries, as explained in sec-
tion 3.4. In this simplified version, essentially, we remove the resources raster, and the
EatAction class, since there is not a direct equivalent to rasters in Mesa. The main differ-
ence between this simplified version and the original is that agents no longer have to be
synchronized to each other, through the global resources raster; also, since the agents no
longer starve and die, the total number of agents does not change. This version is much
more suitable for benchmarking, since the performance should be very similar in every
step of the simulation, unlike the original simulation, on which all agents eventually die,
resulting in an empty world that takes almost no computational resources to simulate; the
time that it takes for the agents to deplete their resources is random, and we would see
slight, but noticeable differences in runtime performance, even when using the exact same
configuration.

The scheduler that runs this model has to compute two random integers per agent per step,
and therefore, we should expect the computational complexity of this model to increase
linearly with the number of agents, and with the number of steps. The code of the agents’
behaviour does not attempt to modify any data that other agents could also access; for this
reason, we should not, in theory, find any race conditions between agents when paralleliz-
ing this model. We have designed a few performance test cases based around this model,
which are available in section 3.4.
2.5.2  Zombie contagion model

The *epidemy* model [23] is another interesting model found among the examples of Pandora, that also uses the C++ interface. The world of this model is implemented in class Earth. It initializes two static rasters, whose values are read from two TIFF (Tagged Image File Format) files: population.tiff contains the population density of Catalonia at a one kilometer resolution, and dem_1km.tiff contains the elevation map of Catalonia. Then, it creates three dynamic rasters: humans has the number of susceptibles at every position, for all steps; zombies does the same for infected individuals; newCases tracks the number of transitions from susceptible to infected. Then, it initializes the agents, of class Human, by iterating the entire space, and placing as many agents as the population raster has for every point (it can be scaled down to a lower population in the configuration file). Finally, the infected agents are placed, at the point specified in the configuration file. In this model, the transition from infected to recovered always results in death, since the agent is removed from the simulation.

The Human agents have four attributes: float _threatLevel gets updated at every step to the number of infected divided by the number of susceptibles, bool _infected tells whether this agent is infected or not, bool _hasHumans tells whether there are susceptibles in the same cell as the infected agent’s current position, and int _remainingTime is the number of steps left until the infected agent gets removed from the world (it gets initialized to 30 steps, counting down once the agent becomes infected). Their behaviour is implemented in the three methods that we described at the beginning of this section: updateKnowledge() updates the values of _threatLevel and _hasHumans; selectActions() selects DoNothingAction() for susceptibles, and, for infected, MoveAction() if there are no susceptibles at the current position or if a random check is passed (determined by the virulence specified in the configuration file), and DoNothingAction() otherwise; updateState() decreases the _remainingTime counter for infected, removing the agent if it reaches zero, or, for susceptibles, converts them to infected if a random check is passed (that is, _infected is set to true if the random number generated is greater than _threatLevel times the virulence of the infection).

The two actions that agents perform are DoNothingAction, which has an empty execute() body that does nothing, and MoveAction, which moves the agent one position in a random direction (horizontal, vertical, or both), only if the new position is inside the boundaries of the world, and if it corresponds to land, and not sea (this is checked by reading the dem static raster).

When running this model, with an initial outbreak of fifty infected individuals in Barcelona, with a virulence of 0.1, and a 1:1000 scale of the Catalan population, the simulation usually takes between 600 and 700 steps to spread to every corner, and infect the entire population. At that point, the last agents standing trigger their _remainingTime timer, and get removed, leaving a completely barren world.
2.6 SIR model

In the previous section, we introduced an agent-based model of a zombie contagion. Here, we introduce a formal model of a generic epidemic: the Susceptible-Infected-Recovered model [20], commonly referred to as the SIR model, developed in the year 1927 by William O. Kermack and Anderson G. McKendrick. As the name suggests, there are three groups of individuals: the susceptibles are healthy individuals, the infected are the contagious vector, through which susceptibles become new infected, and the recovered are immunized individuals who survived the infection phase. Initially, all individuals are susceptible, and only a relatively small number forms the initial infected group; no one starts out as recovered.

This model can be characterized by three differential equations 1, 2, and 3, where $s(t)$ is the fraction of susceptibles, $i(t)$ is the fraction of infected, $r(t)$ is the fraction of recovered, $b$ is the rate of infection, and $k$ is the rate of recovery.

$$\begin{align*}
\frac{ds}{dt} &= -bs(t)i(t) & (1) \\
\frac{di}{dt} &= bs(t)i(t) - ki(t) & (2) \\
\frac{dr}{dt} &= ki(t) & (3)
\end{align*}$$

For all values of $t$, the sum of $s(t)$, $i(t)$, and $r(t)$ must be equal to 1. Since there are no mechanisms for adding to the susceptibles count, this model fails to account for birth rate and/or immigration. Also, the rates of infection, $b$, and recovery, $k$, are fixed, which do not take human dynamics into account, and do not include a natural death rate.

Pandora’s zombie contagion model differs from the SIR model in a few aspects. The infected agents have a fixed, decreasing timer that, when expired, causes the agents to die. In the SIR model, the population count cannot change, and therefore no agents should die; furthermore, agents are not supposed to remain infected forever, since there is a recovery factor that eventually gets to all of the infected agents. Lastly, the two transitions between the states of SIR should be probabilistic, and not deterministic after a number of steps.
3 Results

Now that we have introduced the different tools we intend to use, including a cloud provider, programming languages, and administrative and development software, we present a design of an infrastructure that fulfills our needs. On the one hand, this design is presented in a way that makes the architecture agnostic to its computational platform, but, on the other hand, we also give technical details of our infrastructure that implements this design using our chosen technologies.

We also explore the performance response of the ABM simulation library we use, Pandora, in a variety of hardware configurations, and present our implementation of an agent-based model that approximates the SIR model explained in section 2.6.

3.1 Software architecture

Figure 7: The top-left box represents the frontend of the infrastructure, which considers two kinds of users: regular, unprivileged users, and administrators. Regular users only have enough permissions to launch jobs in the pipeline through an API, but cannot directly connect to machines, change provisioning settings, configurations, etc. Administrators, on the other hand, have credentials to directly manage machines that support the infrastructure, and can make changes to the configuration and deploy buckets.

The frontend is the least developed part of the infrastructure, so only a very simple implementation has been done. There are two components in it: one is a virtual machine that has full access to the backend, intended for system administrators; and the other is an API that allows developers to make calls to the infrastructure, and, possibly, develop more complex frontends on top of it, such as a web application, or a graphical tool.

The large gray box contains all four steps of the pipeline, that coincide with the work packages described in section 1.2. Under each, there are the software components needed to support the flow of data through the pipeline. The specifics for the implementations of the stages are described in detail in section 3.2.

Figure 7 shows how the software is laid out across multiple nodes, on a cloud-based infrastructure, and on an HPC cluster.

The frontend to the infrastructure is a very simple API, to which regular users send messages. As it is now, this is far from ideal, since these users need to have the AWS CLI installed on their computer, and send a message to an SQS queue. A much better solution would be to implement a RESTful HTTP API, so that the infrastructure can be accessed through curl commands, or even through any web browser. On top of this, a more intuitive graphical frontend could be implemented, that allows users to send jobs through it, monitor them as they progress, and then work with the results from the same place. For administrators to perform the most basic tasks, only a web browser is needed to access the AWS Management Console, but, in general, they should set up the development environment explained in section 2.1 to work with all of the infrastructure.
All four stages of the pipeline, including the frontend, require compute resources. In terms of infrastructure, the requirements are very similar between stages: only compute, storage (shared and persistent), and a distributed messaging queue are needed to support a stage. For software, every stage has its own particular requirements. The first stage only needs support for downloading files through web protocols, commonly HTTP. The second stage needs a working Hadoop distribution, as explained in section 1.3.3, and, optionally, some other Hadoop libraries, depending on the kind of processing required. On the third stage, we need a library for simulating ABMs, optimized for an HPC environment. Lastly, on the fourth stage we need an HTML5 web portal with a geographical information system (GIS) implemented on top.

The software we have developed for this project is an implementation of the architecture presented here. The emphasis is not in error resilience, or high availability, but, in simplicity, and in making it easy to extend.

In figure 7, the software pipeline depicted is connected to the underlying infrastructure in a rather abstract way; in reality, both aspects of our project are inevitably coupled together. In a future iteration, a unified interface between the backend and the frontend could improve portability, and ease of management and future development.

### 3.2 Infrastructure components

Figure 8 presents a diagram that shows how the underlying infrastructure looks like. Note that this diagram presents the infrastructure in a way that is not tied to any underlying hardware and/or network configuration in particular, and therefore abstracts away from the cloud provider we use, AWS. This should, at least in theory, allow the infrastructure to be ported to any other cloud service provider, as long as it offers equivalent services to the ones we use. The Hadoop cluster is not represented accurately, because we are not actually managing the individual nodes; we use a cloud-managed cluster, otherwise known as Hadoop as a Service, so we control the cluster from a higher level. However, different providers handle that management service differently, so we cannot be sure how this particular region will look like in a concrete environment. The HPC cluster is abstracted away, to illustrate the same point than with the cloud provider: it should be replaceable, at least in theory, with another similar environment that provides an SSH server, and a suitable compute platform for Pandora to run on.

On the other hand, figure 9 shows a diagram that is centered around specific AWS services, and presents an integration with the HPC cluster, and the CARTO web service. Even though the underlying structure is the same as the one on figure 8, the authorization and isolation mechanisms of the parts of the architecture are dependent on the backing infrastructure, and, thus, can only be accurately depicted around concrete AWS services. The big box in the middle of figure 9 is the private network in AWS where all of our EC2 instances are. This network is open to one address listening on port 22 (standard for SSH). The rest of the instances do not need have a public address, and they should not be directly accessible from the Internet.

The basic infrastructure is built on EC2 instances which run the daemon programs de-
Figure 8: In this diagram, green boxes represent computer resources (including software and hardware, virtual or otherwise), red boxes are distributed messaging queues, yellow boxes are isolated software components, and blue boxes are persistent storage.

On the leftmost column is depicted the frontend of the infrastructure. The master instance provides SSH access to administrators, and manages the frontend API. Its main task is receiving messages from the main jobs queue that are sent from the user API, or manually, from an administrator’s computer. In turn, the master instance reads the job definitions from the configuration storage, and sends a message to one of the backend queues. Each of the stages monitors itself, and sends a message to the next stage when the data becomes available for it to process.

The rightmost column shows the stages of the pipeline. The one on the top is a simple cluster of identical virtual machines. The next one, on the middle, is a multi-node Hadoop cluster, and the one on the bottom is a combination of simulation on a HPC cluster, mediated by a virtual machine, which, in turn, triggers the presentation of results.

All of the stages in the pipeline require some form of persistent, shared storage. It is depicted in the bottom-right corner as a single box, even though it is partitioned with specific isolation mechanisms explained in section 3.2.

Additionally, there is the status queue, which receives messages from the backend, and sends them to the frontend; its purpose is to notify users that the tasks they run have finished (or failed), and it is a non-critical component of the infrastructure.

scribed in appendix A. These programs communicate with each other via SQS. The persistent storage of the infrastructure is handled with a series of S3 buckets. If we did not have a cloud environment, the distributed queue, and persistent storage would have to be designed differently, because we would have to manage the hardware and software that would support those services; however, in AWS we can ignore that, and focus on the communication mechanism itself, and not on the underlying platform (the same can be said for S3).

Communication with the HPC cluster is performed via an SSH tunnel through the Internet, with which we send the simulation data we intend to run. The authentication credentials required are stored in the deploy bucket, where only the administrators have access. For presenting the results back to the user, we send it to CARTO through its HTTP API, in the form of SQL statements that contain geographical coordinates, which then get rendered on a tiled map, on HTML5.

To implement the communication model introduced in section 2.4, we need some sort of
Figure 9: On the top-left of the diagram, the two types of user are depicted: normal users, who access the system through an API, and administrators, who have SSH access to the master instance. The large box, labeled VPC, encompasses most of the AWS infrastructure. IAM roles, SQS queues, and EC2 AutoScaling policies are, technically, not part of the VPC, but appear there for simplicity. At the bottom-right, there is the HPC cluster, connected to our cloud through an SSH tunnel, and, at the bottom-left, the CARTO web service.
distributed messaging mechanism to connect the frontend of the infrastructure with the backend, and from one stage of the pipeline to the next. In addition to that, the main frontend must be able to send messages to any stage in the pipeline. In addition to this, we need to define mechanisms that react in response to messages. In the case of the master instance, we do not need anything special, as it is supposed to be already running, and continuously polling the main SQS queue. In the source downloading stage, we do not have any instances running, unless they are already working; instead, we use AWS CloudWatch, which lets us define alarms, that periodically monitor some metrics, and perform actions when certain conditions are met. In this case, the metric is the number of messages waiting in the SQS queue, and the action is to add a new instance to an EC2 AutoScaling group; the action gets triggered every time the metric is greater than zero. Unfortunately, this mechanism is not reliable enough to launch EMR clusters, which have a lengthy boot time, and therefore take very long to remove messages from the queue: this would cause the alarm to get triggered more than once per message. To solve this, we define a very small instance that continually listens for SQS messages, and creates, configures, and launches EMR clusters. For the simulation stage, we also need an EC2 instance that is continually polling the SQS queue, establishes SSH connections to a remote host, sends commands to it, and then copies the results back to the cloud storage.

3.2.1 Master instance

The main infrastructure frontend is an EC2 instance that holds two responsibilities: on the one hand, it exposes a simple programming interface for launching jobs on some or all parts of the pipeline, and, on the other hand, it accepts SSH connections from any address (open to the world).

Some parts of the infrastructure, like the instances of the source downloading and scraping clusters, are created, and therefore provisioned dynamically; this instance, master, is provisioned manually, by an administrator, using an Ansible playbook (as explained in appendix C).

As soon as this instance is created, its network adapter binds to a reserved public IP, so that it can be accessed from the outside through a known address. Therefore, a system administrator can open an SSH tunnel to this machine, and run management and deployment operations from the system shell. For that to work, we have to generate a private SSH key when we create this instance, or use an existing one, and put the public key on it. By connecting to it, administrators can update the job definitions by running the sync.py program explained in appendix A, read the logs of the status queue, and/or debug other parts of the infrastructure.

This instance is also in charge of listening to the main jobs queue, and accordingly generate messages that represent jobs to be executed by the pipeline stages. It does not directly monitor the status or health of the rest of the infrastructure, but it does listen to notifications on the status queue, that indicate that a job has finished (successfully or otherwise), and that new data is available.

As we have mentioned before, this instance also supports the frontend interface for users; in reality, there is no frontend interface, except for the fact that it listens to the main SQS
queue. In the future, this instance will support some kind of web interface, like an HTML5 app, or a REST HTTP API. Also, the frontend code could be separated from the part that sends messages to the backend SQS queues, so that it can be ported to a lighter compute service, like Lambda.

In terms of networking, this instance has a public IP (which is an elastic IP), and its security group is set to accept SSH ingress traffic from 0.0.0.0:22, which means accept connection from any address to the standard SSH port, 22. We accept all egress traffic from this instance to all addresses.

### 3.2.2 Source downloading cluster

On the first stage of the pipeline, there is an EC2 AutoScaling group of instances that listen for messages on an SQS queue, download files from the Internet, and store them on the sources S3 bucket. The download jobs are specified as lists of URLs, and filenames for these files (see section 3.5 for how data sources are modelled).

This cluster scales out according to a CloudWatch alarm, which is attached to a metric that counts the number of SQS messages waiting to be received on the source jobs queue. When the number of messages is zero, the number of instances running should also be zero. If at least one message is waiting, an instance will be spawned, it will receive the message, and it will start the download. If more messages are waiting to be processed, then additional instances will be spawned. This means that, at most, there will be one instance per SQS message. This maps to a parallelized environment, with each job running in a separate EC2 instance; if a large number of small jobs is to be run, it may be more desirable to set a maximum number of instances (for example, one instance per ten messages), and have those run jobs sequentially; this avoids a large overhead of combined boot time that may not be cost-effective.

The method for scaling in is less automated, as it requires instances to manage their own state. When an instance finishes a given job, it will query SQS to see if there are any messages waiting. If there are, then it will take the message, and start another job. If there are not any, an instance-initiated shutdown procedure will begin, in which the instance shuts down the operating system, and tells EC2 that the instance is to be terminated. A possible optimization to this is to have instances check whether their network link is running at full capacity or not, and take another message if it is not, therefore running two or more jobs in parallel (until the network link reaches full capacity).

The downloads are temporarily stored in the instances’ root partition (EBS backed). When the download ends, files get sent to an S3 bucket. This is the cheapest alternative, in economical terms; a more expensive but more network-efficient would be to write downloads directly to an Elastic File System (EFS) that is readable by the scraping stage, as long as both run in the same availability zone inside our region (eu-west-1).

Because instances in this Autoscaling Group appear and disappear dynamically, they cannot be provisioned by an administrator, with Ansible. Instead, we give these instances a Bash script that bootstraps the operating system, by installing our Python scripts, and starting the daemon.
The mechanism presented here is simple, and it works well enough for production. However, AWS presents an even more efficient method for doing this, with an important limitation: instead of launching a full-fledged EC2 instance, we can launch a much lighter Lambda instance. The limitation is that Lambda environments can only run for 300 seconds, so this is only possible for small files that we know can be downloaded from the Internet, and uploaded to S3 in less than five minutes. Since this does not add any new functionality, we have not implemented such a system.

However, if extremely large files are to be downloaded, it may be more desirable to use the multipart upload feature of S3: this would allow instances with a small storage drive to download files in chunks, cache them, and have another thread upload those to S3 in parallel. This removes the requirement of having an EBS volume as large as the file being downloaded. Since we have not had this problem for processing our example dataset, we have not implemented this optimization either.

The instances in this group only require a route to an Internet gateway, from which to download sources. These instances do not expose any service, so all ingress traffic is blocked.

3.2.3 Source scraping cluster

The scraping stage is responsible for performing various transformations to the raw data obtained in the previous stage, such as extracting tables, converting between formats, enriching the information, etc. It runs on Elastic MapReduce, or EMR, a multi-node Hadoop cluster, partly managed by AWS, partly by us. The Hadoop distribution in EMR comes bundled with many different tools for programming and managing Hadoop jobs in many different languages, with different techniques and paradigms, as seen in figure 4.

Our infrastructure does not lock the user into a particular programming language or framework for Hadoop, so we provide a mechanism for dynamically provisioning the EMR cluster with programs that are specified by the scraping job itself, as described in section 3.5. The job definition must specify which programs are to be used for mapping and reducing, and those must be accessible in the deploy bucket, so that the EMR bootstrapping process can install those in the Hadoop cluster; similarly, the job definition must specify the location of input data (i.e. what path in the sources bucket), and the location of output data (i.e. what path in the scrapes bucket, and, optionally, if the data is to be uploaded somewhere else, like an RDS instance).

Regardless of the job configuration, EMR is run in step mode, so that the cluster automatically terminates itself when the last step is run. The configuration language allows jobs to specify multiple steps to be run sequentially; these are Java JAR files that are run by the Hadoop scheduler; in EMR we can use command-runner.jar to achieve two things: we can run arbitrary programs, or we can launch Hadoop jobs of different types, as specified by the first argument: hadoop-streaming, hive-script, pig-script, spark-submit, or s3-dist-cp. Any other .jar can be chosen from the locally installed ones, or from S3.

Because of some limitations in EMR, the IAM roles used for the EC2 instances that run the cluster are not restrictive, like in the rest of the infrastructure; this means that, potentially,
the EMR instances could send messages to any SQS queue, and read/write any S3 bucket. In the future, this should be improved, so that we can have more control over the instances of EMR, and use them as we intended when we designed the pipeline.

The cluster is launched by a small EC2 instance that runs a daemon, which listens to messages in the SQS queue. The configuration contained in the message is parsed, and an EMR cluster configuration is generated. Some specific details are hardcoded, such as sending SQS messages to the status and simulation queues, and auto-terminating the cluster after the last step, but almost everything else can be configured in the scraping job description. Additionally, some other aspects are configured into the tags of the EC2 instance, such as the bid price for spot instances (hpopt_BidPrice), the instance type of master (hpopt_MasterInstType) and slave nodes (hpopt_SlaveInstType) of EMR, and the size of the EBS volumes of slave nodes (hpopt_SlaveEbsSize). Like the master instance that we described in section 3.2.1, the code that runs on this instance could be run from Lambda instead, therefore avoiding the need to pay for a long-running instance that is idle most of the time.

As we have already mentioned, EMR clusters are built on top of EC2; since Hadoop has its own requirements from the network, for communicating master and slave nodes, we cannot be as restrictive with the network configuration on EMR as we were with the other stages. Here, we leave the network security settings as they are by default.

### 3.2.4 Agent-based modelling simulation cluster

The last stage in the pipeline is in charge of running ABM simulations, and it accomplishes this by mediating on-cloud resources with an off-cloud HPC environment. This means that, from the side of the cloud, we must open a secure communication channel through the Internet, install the model, and its resources, on a remote host, and execute it there; finally, the data generated by the simulation must be sent to be processed for presentation of results. Optionally, some software setup may be needed, like installing external libraries.

Like in the scraping stage, messages in the SQS queue arrive to a small EC2 instance, which processes them (with the hpopt-optimize daemon, in this case), and dynamically provisions a larger component of the infrastructure to run the task, which in this case is an HPC environment. The communication channel is established with SSH, where the HPC environment holds the server, and our instance the client. The server address and remote user are stored as the tags of the EC2 instance, with names hpopt_EndpointAddr and hpopt_EndpointUser, respectively. On our side, we can control what software is installed, where, and how, but on the remote machine, we are given an unprivileged user, with which we cannot have that kind of control; for this reason, our infrastructure cannot automatically install and run software on the HPC cluster, like we could in the EMR cluster. In this stage, ensuring that the right software is available is the responsibility of the users. For example, Pandora, and its libraries (like HDF5), can be installed inside the home directory, where no special permissions are required, and be loaded from there (unlike in section 3.4, where we install Pandora to the system-wide location of /usr/local). Here, since we are not tearing down the execution environment as soon as the task is done, like we do in the scraping stage, it makes more sense to set up the software dependencies before any tasks are sent to the simulation stage. Since we do not have a concrete HPC environment, we do not have
any Ansible playbooks that we can launch to provision it, but it would certainly be the way to do it.

After the small EC2 instance copies the necessary files to the HPC environment, over SSH, it executes the model, and waits for the command to finish, leaving the SSH channel open. When it finishes, the generated files are copied back to the EC2 instance, and then uploaded to S3. Finally, an SQS message is sent to the status queue, from the small EC2 instance, and the presentation of results begins. While the model is being executed, we are required to hold an active connection to the HPC environment; this means that we cannot shut it down to save costs, but it also means that we do not have to install any AWS credentials or software in the HPC environment, which simplifies the provisioning of this stage.

The network configuration of the HPC cluster is completely beyond our control. The small EC2 instance does not expose any services, so all ingress traffic is blocked. An Internet connection is required to establish the SSH tunnel to the HPC cluster, so egress traffic must be allowed to reach an Internet gateway.

### 3.2.5 Presentation of results

This is not a pipeline stage, since it shares infrastructure with the simulation stage, but it performs a distinct task, and uses different services, so we consider it a separate component. Accordingly, it does not get an SQS queue, and an S3 bucket, like the other stages, since it gets executed directly by the hpopt-optimize daemon, and it can access the simulations bucket directly.

This component uses the CARTO SQL API to send the results from the simulation to a table, after interpreting them. The internal representation used by Pandora is not able to be understood by the CARTO Import API, so we have no option but to convert one format to the other as we process the resulting files. Additionally, options for visualization could be configured here, with the Javascript and/or CSS APIs of CARTO.

Calls to the CARTO APIs have to be authenticated by a private API key; in our implementation, we have our private API key stored in a file, inside the deploy bucket, where only the administrators and the EC2 instances have access (and not developers and other users), so we copy it to the instance’s private, local storage, and read it from there.

This component shares infrastructure with the simulation stage, so the network configuration is the same that was explained in the previous section.

### 3.3 Agent-based SIR model

In order to create a model that can simulate agents that interact with the hospital dataset, we have taken the source code of the zombie contagion model, described in section 2.5.2, and added our own modifications. Essentially, we have introduced a new probabilistic recovery factor, that causes infected agents to transition to the recovered phase; this replaces the old _remainingTime attribute of the agents, which is not compatible with the SIR model, as we explained at the end of section 2.6. Additionally, if a hospital is within range of the
Figure 10: Here we can see six charts with the number of agents of each group on the vertical axis (logarithmic), steps on the horizontal axis, and varying parameters for \textit{virulence} and \textit{recovery}. These models have hospitals in them, but their effect has been negated by setting \textit{recoveryMultiplier} = 1 and \textit{maxRadius} = −1.
agent, this effect is amplified (multiplied) by another given factor, as long as the hospital is not overcrowded. While ours is not a strict implementation of the SIR model, it does not contradict it either, and can therefore be considered an extension of it.

The initial state of the SIR model is defined by four parameters: the size of the susceptible population, the size of the infected population, and the two factors that determine the rates of transitions: $b$ and $k$. In our model, the initial susceptible population is given by the population.tiff file, and the infection by the value of initInfected, in the <outbreak> tag in the config.xml file. The $b$ and $k$ factors are represented by virulence and recovery, respectively. However, they are not exactly the same: in the SIR model, $b$ and $k$ are the fractions of people that become infected or recovered per unit of time (be it hours, days, etc.), and in our model, virulence and recovery are the base probabilities of transitioning for each individual. In other words, in our ABM model, agents’ probabilities are determined by how many neighbors are infected, as the actual probability is virulence times threatLevel, on a case by case basis, whereas, according to the SIR model, their probabilities would depend only on how many susceptibles and infected there are in total.

The config.xml file now has a few extra parameters: in <inputData>, hospitals points to a CSV file, with three columns (horizontal axis position, vertical axis position, and recoveryMultiplier); in <outbreak>, recovery has a decimal value, between zero and one, that represents the probability of recovery, recoveryMultiplier multiplies the base recovery factor when an infected makes contact with a hospital, and maxRadius determines the maximum distance at which an infected and a hospital can interact.

The Earth initializes the static rasters like it did before, but the dynamic rasters are now: susceptibles instead of humans, infected instead of zombies, and recovered instead of newCases (which now tracks agents who transition from infected to recovered), where each raster represents the phase in the SIR model after its name; additionally, we precompute, for each position, the index to the closest hospital within range, and store the result in the hospital raster; this saves us a lot of redundant computations, since, otherwise, every agent would have to scan the entire hospitals list, at every step, and calculate its distance.
Figure 11: Here we can see six more charts, with the logarithm of the number of agents and the steps; this time, we use a fixed virulence = 0.75 and recovery = 0.125, and we vary the recoveryMultiplier and the maxRadius.
from every one of them. The creation of Humans is exactly the same as before; however, now, the Earth creates a second set of agents, of class Hospital, at the positions indicated by the hospitals.csv file.

The Hospital agents have two attributes (_served and _maxServed), and the same three methods as other Pandora agents, only that, in this case, they do nothing. Instead, Hospital agents now expose a requestAntidote() method that Human agents call that, as long as _served remains smaller than _maxServed, returns true (and increments _served), and causes the agent to multiply its recovery factor. At the end of each step, the Earth resets the counters of all hospitals to zero.

The Human agents’ behaviour has been modified as well. The updateKnowledge() method remains, mostly unchanged: susceptibles update their _threatLevel to the number of infected divided by the number of susceptibles, and infected their _hasSusceptibles to true if there are susceptibles in their position; recovered agents do nothing. The selectActions() method is also very similar to what we had before: susceptibles (and now also recovered) stay put; infected always move away from empty positions, and from positions with susceptibles with a probability equal to one minus _threatLevel.

The most representative method, however, is updateState(), which can be seen in listing 4. In it, we find three code blocks, one for each phase of the SIR model. Recovered agents do nothing. Susceptibles calculate the probability of being infected, which is virulence times _threatLevel, and change their state to infected, if a random check with said probability is passed. Infected agents, first, calculate the value of recovery, taking hospitals into account; to do this, they check the value of the hospital raster at their current position: if it points to a hospital, they call the requestAntidote() method (the Hospital checks its _served attribute, increments it if necessary, and returns true if it is below _maxServed, or false otherwise). If it returned true, the recovery factor is multiplied by recoveryMultiplier. Finally, a random check is performed, with probability recovery (whether it was multiplied due to a Hospital or not), and the agent is transitioned to recovered, if it passed.

We have run a series of tests on this model, with different parameters, but always with the same initial configuration: using the map of Catalonia, given by the population.tif and dem_1km.tif files, an initial infection at Barcelona (at position 168, 164) of 50 agents, and the hospitals given by the file hospitals.csv. First, we have run the model without the hospitals; the results can be seen on figure 10. When the virulence and recovery factor are comparable, the infection spread reaches an equilibrium, and dies out before it can get to a significant fraction of the population. However, when the virulence is high enough, the entire population eventually becomes infected, and then recovered.

When enabling the hospitals, their degree of effectiveness is determined by the recovery factor (given by config.xml), and by their maximum capacity (given by hospitals.csv). The results can be seen in figure 11. The virulence and the recovery factor are intentionally disproportionate; however, increasing the recovery multiplier and/or the maximum radius has a dramatic effect on the outcome of the epidemics, considerably reducing the time it takes for the infection to elapse.
void Human::updateState()
{
    if (!_recovered) return;

    Earth& world = (Earth&)getWorldRef();

    if (!_infected)
    {
        float probabilityRecovery = world.getRecovery();
        float recoveryMultiplier = world.getRecoveryMultiplier();

        int nearestIndex = world.getValue("hospital", _position);
        if (nearestIndex>-1)
        {
            Hospital* nearestHospital = world.getHospitalByIndex(nearestIndex);
            if (nearestHospital->requestAntidote())
                probabilityRecovery *= recoveryMultiplier;
        }

        if (Engine::GeneralState::statistics().getUniformDistValue() < probabilityRecovery)
        {
            _infected = false;
            _recovered = true;
            changeType("Recovered");
        }
    }
    else // susceptible
    {
        float probabilityAttack = _threatLevel * world.getVirulence();
        if (Engine::GeneralState::statistics().getUniformDistValue() < probabilityAttack)
        {
            _infected = true;
            changeType("Infected");
        }
    }
}

Listing 4: Source code of Human::updateState(), in C++

3.4 Performance analysis of the simulation cluster

For simulating our agent-based models, we have, at least, two software libraries to choose from, as explained in section 2.5.1: Pandora, and Mesa; we have only considered those two, but multiple others exist. We must also consider the hardware platform on which we execute our library; namely, a virtualized environment, provided by AWS EC2, or an HPC environment (spanning multiple nodes communicated by MPI). Since Pandora offers two interfaces, one in Python, and one in C++, we treat them as potentially separate libraries, in terms of performance. Other than that, we also take a look at the scalability on the number of agents of each of the libraries.

As the model, we have used the modified random walkers model described in section 2.5.1. The benefit of using such a simple model is that the computational cost of every step never changes. Also, it was easy to port from C++ to Python, and to Mesa (in Python). The drawback is that this model is too simple to fully utilize a multicore processor. Also, we must consider that the data collection mechanisms in Mesa and Pandora are radically different: while a naive Mesa implementation will hold all data in memory, an equally simple implementation in Pandora will, by default, serialize data to disk (more specifically, to an HDF5-formatted file); in order to make a fair comparison, we have disabled the implicit serialization mechanism of Pandora (instead of recording each step, we store only
Figure 12: This chart shows the results collected during the performance tests on the modified *random walkers* model. The vertical axis represents the time, in seconds, that each of the programs took to run the model. On the horizontal axis, the data points are grouped by number of compute cores available, and coloured as indicated by the legend, to indicate which of the ABM libraries it represents (Mesa, Pandora in Python, or Pandora in C++). The models have been run with 1000 agents, during 10000 steps, in a $500 \times 500$ discrete grid.

Figure 13: This chart shows the results collected during the second performance tests, on scalability on the number of agents. The vertical axis represents, again, the time, in seconds, that each of the programs took to run the model. On the horizontal axis, the data points are grouped by the number of agents in the model, with the three libraries in different colour, as indicated by the legend (Mesa, Pandora in Python, or Pandora in C++). The models have been run during 500 steps, in a $500 \times 500$ discrete grid, with varying number of agents.
one in a million, for instance). The reasoning behind this is that, even if we decide to use Mesa, we will need a similar serialization mechanism that Pandora already offers, which would incur a similar performance hit, if not worse, so we can ignore that factor.

A 2012 article by Peter Wittek and Xavier Rubio-Campillo compares the performance of a regular HPC environment, and one based on the AWS EC2 infrastructure, in two kinds of tasks: compute-intensive, and communication-intensive. Their results show that, in the former, performance is comparable on the two platforms, whereas, in the latter, EC2 showed significantly worse performance. Since the article was published, the EC2 instance types mentioned are no longer available, so the results may change on newer instance types, where faster networking hardware is available (i.e. 10 Gbps instead of 1 Gbps).

We have run this model on a series of EC2 instances with increasing numbers of virtual CPU cores. In this case, Mesa acts as the baseline, since we already know that it can only use one processor core, and thus will always take the same, regardless of the cores available. The instance types we use can be found on the second half of table 2: m4.large, with 2 cores, m4.xlarge, with 4 cores, m4.2xlarge, with 8 cores, and m4.4xlarge, with 16 cores. In this case, it is better to use m4 instances, and not t2 instances (like the master instance), since we require extended usage here (there are more details on the difference between the two families in the caption of figure 2).

The process of running the tests on EC2 has been automated through an Ansible playbook that launches the instances with a bootstrap script, which launches the test models and collects the results. The AMI we install on the instances has Ubuntu 14.04 LTS (Trusty Tahr), because that is what the Pandora developers recommend to build and run their software; since Mesa is packaged as a PyPI package (and thus can be installed in any Python distribution), we do not have any requirements for installing and running it, so we run Mesa on Ubuntu too. Because Pandora creates large HDF5 files during its simulations, the instances are created with a 50 GB EBS volume for the root partition. Also, the instances have a special role that grants them access to a bucket dedicated for distributing the code of the test models, and for storing the results; this role is not part of the infrastructure that we define, but it is needed to allow these EC2 instances to access S3, nevertheless.

The tests could run for a long time, and it would be inconvenient to use an interactive shell, so we execute them from the bootstrap code; the script installs Mesa and Pandora on the system (Pandora must be first built from sources, as it has not been packaged by its developers), then downloads the tests and runs them, and uploads the results to an S3 bucket. The specific script can be found in appendix E. It first performs a global package upgrade, and then installs the dependencies of the tests. Then, it downloads the relevant programs from the S3 bucket, and runs them; the standard outputs of the programs are piped to a text file, and when the simulation ends, these files are uploaded, together with the output of the whole bootstrap process (i.e. /var/log/cloud-init-output.log). The destination folder inside the S3 bucket is named dynamically, and indicates the instance type (e.g. m4.xlarge), and the date and time at which the test was completed. When running the simulations, we measure the time it takes for the whole execution of the program. The data generated is discarded, because it is not important for the purpose of measuring performance; also, because the model being simulated is trivial, and the data is of no interest. We also collect the cloud-init-output.log, to debug problems during runtime.
The first test was intended to measure how much the performance improves just by adding more compute cores to the model. The results can be seen in figure 12. The two Python libraries (Mesa and Pandora) show very similar results: neither of them gained any performance boost by adding more CPU cores, and both took around 160 seconds to run the test. The library that stood out was Pandora in C++, which took about 30 seconds less to run the model, due to the agent scheduler not having the overhead of a Python interpreter; even so, it did not show a performance increase that we were expecting to see.

In section 2.5.1, we established that we should see a performance improvement on parallel hardware, since there is no synchronization (and therefore, no race conditions) involved in executing the agents’ behaviour methods. However, it can be seen in figure 12 that this is not the case; after carefully studying the source code of Pandora, we found that the culprit was that the agents’ behaviour was in a non-parallel section of code: in other words, the model was correctly utilizing all of the processor cores, but the code being executed there could never be parallelized. More precisely, the issue was that the agents were, at each step, scheduling the MoveAction s in parallel, but running them sequentially. By examining the implementation of the executeAgents() method of OpenMPSingleNode, on listing 5, one can see that, while agent->selectActions() method is called in the first, parallelized for loop, agent->executeActions() is called in the second, non-parallelized loop. In this case, what determines whether a loop is parallelized or not, is the C preprocessor #pragma directive of OpenMP preceeding a for loop; the first one has it, whereas the second does not.

As was explained in section 2.5, Pandora, by design, imposes a specific strategy for implementing the agents: exploration and planning should happen in the two first methods (updateKnowledge() and selectActions()), and actuation on the third (updateState()). On the one hand, this makes a lot of sense, because it encourages developers to design agent-based models with a certain strategy that Pandora knows how to optimize well; on the other hand, we lose some control over how exactly is our code being parallelized.
In order to back this claim, we have created a slightly modified version of Pandora that also parallelizes the second half of the `executeAgents()` method, and run the same tests again, only that this time, we have used the HPC environment of the UPF, introduced in section 1.3.2: SNOW. We already know the difference in performance between EC2 and a regular HPC cluster, as we mentioned at the beginning of this section: EC2 is as good, or slightly worse; here, we are only interested in getting a performance increase with the number of compute cores. The results of this can be seen on figure 12. Finally, we were able to get a large performance gain: doubling the compute capacity from two to four cores resulted in almost a 50% decrease in runtime, and continued to split almost in half, every time we doubled the number of cores (we went up to 16, which is already a considerable amount of compute capacity).

We have also run another kind of test, to compare the impact of adding more agents to the simulation of our simple model in each of the three libraries. As seen in figure 13, the scalability of agents in Pandora in C++ is very different from that of Pandora in Python, and Mesa. Whereas, in the first two, the time that it takes increases roughly linearly, Pandora in C++ seems to take increasingly longer the more agents are added. This appears to be a bug in the C++ interface, and it has been reported\(^{29}\) to the developers and maintainers on the official Pandora forum, as a request for comments.

### 3.5 Example workflow

In this section, we show a complete workflow that uses all stages of the pipeline, together with details of the implementation, and instructions on how to properly initialize the infrastructure. We use the dataset described in section 2.3 to create a list of hospitals, processed with a MapReduce program, enriched with OpenStreetMap Nominatim, and then passed to the agent-based SIR model presented in section 3.3.

Before getting started, a development environment should be established, as explained in section 2.1, so that all of the software tools needed in this section are available. Now, `git clone` should be run with the URL of our Bitbucket repository, available in listing 2. After changing the current directory to the cloned repository, the first thing to do is to change the names of the S3 buckets in the `pipeline-names.json` file, since those have to be unique across all AWS users. Optionally, the RPM package can be generated here, although a prebuilt version is present in the repository. Running the `make-rpm.sh` script in the root of the sources will (re)create `rpm/hpopt-latest.rpm`. Every time the Python scripts are modified, this step has to be repeated, so that EC2 instances get the new version.

For this section, we have developed three configuration files, that specify the parameters of each of the stages of the pipeline. These are found in the `config_default` directory of our repository, as the sample files. In the next paragraphs, we explain what the format for each file is.

\(^{29}\)https://groups.google.com/forum/#!topic/pandora_users/_vzgVDMICX4
3.5.1 Obtaining the source data

On the data source descriptor, there are only two keys available, and both of them are mandatory. The first key is "name", which specifies the name of the directory in which the downloads will be stored; therefore, a valid file name must be given. The second key is "steps", and its value must be an array, containing one or more objects; each of those objects must specify a properly encoded URL in "url", and a valid file name in "filename". Note that both "name" and "filename" can indicate multiple subdirectories with /, but it can never be an absolute path (starting with /).

The entire dataset is contained in an XLSX file that contains a few joined tables with metrics. We are dealing with a single file, so the configuration file for it is very simple: under "files", a single object with a "url" and a "filename" is provided.

```json
{ "source": {
  "name": "name of the source",
  "steps": [
    { "url": "http://(...)="/,
      "filename": "filename1.ext"
    },
    { "url": "http://(...)="/,
      "filename": "filename2.ext"
    }
  ]}
}
```

Listing 6: Data source descriptor, in JSON

3.5.2 Scraping the source data

On data scraping descriptors, there are a number of different possible keys. The first key is "name", which is mandatory. The keys "cores" and "tasks" specify the number of instances to use as cores, and as tasks (on spot instances), respectively; "tasks" is optional, and both fields can be zero (in which case, the cluster would consist of a single master instance). The next key, "logs", is optional, and specifies a directory in which to store logs generated during the execution of the cluster; if missing, no logs are kept. The next key, "bootstrap", holds a list of actions to perform after the cluster has booted up, but before executing any steps; typically, this can be used to install extra software in the instances of the cluster. Lastly, we have the "steps" key, in which we can describe any number of programs to run sequentially on the cluster; here, a few different options are available: we can run a regular command, or we can set up distributed tasks to operate on the data, with the Hadoop framework.

Each step is represented as an object, where "name" and "type" are the only keys that must be always present. The rest of the arguments are interpreted depending on the "type": for streaming steps, "exec" expects an object with a "mapper" and a "reducer", but for all other types, a string is expected, which must point to a valid program, in a format and/or language compatible with the type of step. Some types of step can also have "args", "input", and "output" keys. The types accepted are "streaming", "hive", "pig", "spark", "custom", "command", and "s3distcp".

The "streaming" steps consist in a direct MapReduce job, in which the user passes a program for the mapper, and another for the reducer. Input/output is done through the standard
input and output file descriptors of UNIX. On the other hand, "hive", "pig", and "spark", translate to a higher level framework, in which we specify a program (text in Hive and Pig, or a JAR in Spark), and the framework implicitly runs it on MapReduce. The same can be done with "custom", but more flexibility is offered, so that the user can pass an arbitrary JAR, with arbitrary arguments. With "command", we can run regular shell commands on the master node of EMR. Finally, with "s3distcp", we can copy files between S3 and HDFS (in this case, in and out of EMRFS).

For scraping our dataset, a very simple Hadoop streaming job has been designed. Since the source file is not plaintext, we need to convert it from XLSX; this could have been avoided in native Hadoop, but a Java program would have had to be built in order to accept custom input/output formats. Since we are already using lots of different languages and frameworks, we have opted for the simpler option of creating a streaming program, which in this case has been done in Python. The resulting code is much simpler, but it has the downside of not being able to fully leverage the capabilities of Hadoop, since we have to perform part of the processing in a non-distributed way (by running regular commands on the master node of the EMR cluster). Therefore, we first convert the XLSX data to a comma-separated value CSV file on the master node with xlsx2csv.py. Then, we run our MapReduce job which is specified in two files, map.py and reduce.py: with these, we extract two tables, one with identifying information about each hospital (hospitals.json), and another with all of the metrics contained in the original document (metrics.json).
3.5.3 Adding geographical information

Unfortunately, this dataset does not contain any information about the geographical location of the hospitals described. For creating a more interesting agent-based model, we have written an additional script called geolocate.py, which we use to query an online service for reverse geocoding: OpenStreetMap’s Nominatim\(^{30}\). This is a public, free service that runs on donated hardware and bandwidth. For this reason, its maintainers ask that users conform to the acceptable use policy \([11]\), which specifies that, among other things, only one query per second is performed, results should be locally cached, and no distributed scripts should be used. We have designed our geolocate.py code so that it fits these requirements.

Finally, we have to convert the global coordinates to the local coordinates of the Pandora model. We have achieved it by matching the four corners of the boundary box of Catalonia, in global coordinates, to the local limits of the model \((260 \times 260)\), and re-scaling all the points inside of it. This is technically incorrect, since the Earth is not flat, so a better way to do this would have been to use a GDAL raster that Pandora is able to natively understand; unfortunately, due to a lack of time, we opted for the simpler, incorrect method. The local coordinates are calculated by geolocate.py, and are then rounded to the nearest integer, and converted to CSV by extract.py.

3.5.4 Running the simulation

For model simulation descriptors, we find the usual "name" key. Then, an object has to be specified as "model": this has to contain a "path" relative to the deploy bucket that contains the model, the executable name on "exec", and optionally, two lists of steps, "prep" and "post", to be executed before and after running the model, respectively. Then, under "config", a single file (or a list of them) must be specified, which contains the execution parameters that are passed to the model executable; this means that the model will be run as many times as configuration files there are. Every configuration file should specify a unique path for the output file, or otherwise it may be rewritten in a subsequent run. Additionally, the "wrap" key can be specified, which is a file that will be executed right before running the model, useful for setting environment variables (in the case of Pandora, it needs the variables \texttt{PATH} and \texttt{LD_LIBRARY_PATH} to be modified before it can find its own libraries); whereas "prep" is run once, "wrap" is run as many times as there are configurations to run. Finally, the "input" and "output" keys, which are optional, each contain a list of objects, specifying a source ("src") and a destination ("dst"); the "input" files are copied from the scrapes bucket to the model directory, before it is run, and the "output" files are copied from the model directory to the simulations bucket.

As mentioned before, we use the Pandora agent-based SIR model, which will be uploaded to the deploy bucket later in this section. As inputs, we specify the CSV file with the hospitals, in local model coordinates, extracted in the previous stage. As outputs, we specify the HDF5 file that contains the entire serialized simulation. We only have a single configuration file to run, which is enough for demonstration purposes, but for solving an optimization problem we should execute many simulations with slightly varying parameters.

\(^{30}\)https://nominatim.openstreetmap.org/
and then run an analysis phase to find out which were the most optimal outcomes. As for
the model itself, we specify the S3 path, and the name of the executable; before running
the model, it has to be compiled with `scons`, and after running it, data should be exported
to CARTO.

### 3.5.5 Visualizing the simulation

After the simulation ends, the infrastructure programs will copy the resulting simulation
files back to our S3 bucket. However, for creating visualizations, the model is responsible
for uploading any required data to CARTO (or anywhere else, for that matter); this includes
extracting the data from the HDF5 files, performing statistical analysis (e.g. gathering
averages, etc.), and converting data formats, where necessary.

Other than simulating agent-based models, Pandora also includes a few classes useful
for gathering basic statistics about past simulations, in class `Engine::SimulationRecord`,
and namespace `PostProcess`, among others. For our agent-based SIR model, we wrote a
very simple C++ program that extracts four CSV files: `susceptibles.csv`, `infected.csv`,
`recovered.csv`, and `steps.csv`. The first three contain, each, the total number of agents of
each phase of the SIR model, per step, adding up all the positions in the map. The fourth,
`steps.csv`, has a total of eight columns, and contains the minimum information necessary
to render the simulation in CARTO. Previously in this section, we converted the global
coordinates of the hospitals to the local coordinates of the Pandora model; now, we do the
opposite operation, with all of the points, and convert them back to global coordinates.
We store, for each step, the number of susceptible, infected and recovered agents in every
position. However, because we are using the free license of CARTO, we only have 250
MB of storage available, and therefore can upload only one in every ten rows; for 600
steps, this manages to stay just below that mark.

Since we just converted the data to CSV, we can use the Import API of CARTO, with a
`curl` call, directly from C++, to upload the simulation data. After the file transfer ends, it
takes a few more minutes for their servers to process the file, guess the format, and parse
the geographical information in it (in columns `lon` and `lat`). Now, we only have a dataset,
so, to generate a map from it, we can use the CARTO web interface to do it by hand (we could have used the Maps API, or even the CARTO.js API, from a small web server in the master interface).

In our case, we have taken one of the examples of Torque.js, and adapted it to take the simulation data that we uploaded. With the CARTO.js API, the tile map is initialized; then, the Torque.js API is used to create three animated layers, one with each phase of the SIR model, as seen in figure 14.

### 3.5.6 Orchestration

Now that we have defined the entire workflow, in terms of the three configuration files given above, the Ansible playbooks can be run, which will create the necessary AWS resources, and upload the necessary files to S3. Making sure that the current directory is still in the root of the sources, one should start with `create-iam.yml` and `create-sqs.yml` first, then, `sync-buckets.yml`, which will create all of the buckets, and upload the RPM, the Hadoop program, and ABM model to the `deploy` bucket, and the contents of the `config_default` directory to the `configuration` bucket. Finally, `launch-master.yml`, `launch-downloaders.yml`, `launch-scrape.py`, and `launch-optimize.py` will launch the EC2 instances that send and receive messages through SQS, and also create the scaling policies for the source downloading cluster.

```json
{
    "source": "source-job-name.json",
    "scrape": "scrape-job-name.json",
    "simulate": "simulate-job-name.json"
}
```

Listing 9: Format of objects sent to hpopt-main-jobs-queue, in JSON

At this point, everything in the infrastructure and its software is ready to run the job: to start it, we must send a message to the main SQS queue, like the one on listing 9. Once the message is sent, the hpopt-master daemon in the master instance will receive it, and
read the three possible values it contains. It will ignore the message if it does not contain a valid JSON object, or if it does not have, at least, one of the three keys. It will also ignore the message if "source" and "simulate" are specified, but not "scrape". This could be remediated by coding a pseudo-stage that does nothing (a ‘noop’), and redirects source data directly to the simulation stage, and thus avoiding the EMR cluster entirely; we have not implemented such a functionality.

Then, the script gathers the specified .json files from their respective configuration directories, and merges them in a single JSON object. If retrieval of any of those files fails, the message is ignored, and no further action is performed; a warning is printed to the logs. Then, a new JSON object of the form of listing 9 is generated by combining the JSON objects collected earlier. This object will be passed along the pipeline so that none of the stages require accessing the configuration bucket. Now, the master instance will have to find the earliest point in the pipeline in which to send the message; this means that, for a job that only includes scraping and simulating tasks, the message will be sent directly to the scraping cluster, completely skipping the sources stage. Then, it sends the message to this queue, and starts over with the next message. At this point, configuration files will not be read again for this particular job, and the resulting configuration object will be passed on from one stage to the next, untouched.

Since our example workflow includes a source downloading stage, the CloudWatch alarm that monitors its SQS queue will notice that there is a message not being processed, and therefore will launch an EC2 instance inside the AutoScaling group. After booting up, it will take the message, connect to the specified URL, save it with the correct name, and upload it to S3. At this point, it will send the original message to the next stage, scraping in this case; it will also send a message to the status queue. Now, this instance will wait for a short period of time, and then, self-terminate. It will also notify the AutoScaling group that one less instance is desired, so that it does not launch a new one.

Now, the message will arrive to the EC2 instance that monitors the scraping SQS queue. After reading and interpreting it, it will generate a corresponding EMR cluster configuration, and launch it. For this particular scraping job, a single master node is created, with no core nodes, and no task nodes on spot instances. First, this cluster runs the bootstrap script that obtains the Hadoop programs that we described earlier in this section. Then, it executes the specified steps, in a sequential manner. Finally, it uploads the resulting files to the scrapes bucket. Also, since the job description includes a simulation phase, a message is sent to the last SQS queue.

Like before, when the message arrives to the instance that monitors the simulation SQS queue, the necessary files will be gathered from S3, as explained before in this section, and sent through the SSH tunnel created with the paramiko library. This includes the executable of the model, together with its resources, the specified config.xml file, and the list of hospitals that was obtained in the previous stage. After transferring the files to the remote host over SCP, a command is sent to start executing the model, leaving the SSH channel open. When it ends, the resulting files will be transferred over SCP, and then uploaded to S3. Finally, the analysis program will be run, which will upload the results to CARTO, through its Import API.
4 Discussion and conclusions

In this project, we have tackled challenges in several different areas, and because of that, a lengthy research phase was required. I initially knew about, but had little to no experience with many concepts that this project demanded, like cloud computing, or GIS, and some even were completely new to me, like agent-based modelling. Once all of the components were understood (more or less), an initial integrative solution was sketched, and later refined, iteratively.

Along the way, there have been many unknowns, especially about the assumptions that were being made to establish the requirements from the cloud-based infrastructure: many services promise lightning-fast performance, and petabyte-level scalability, but, in the end, the simplest ones turned out to be the most effective, at least for this project, in its unique context. When studying the cloud provider, AWS, we found that there are many ways to accomplish the same results, and choosing and/or discarding services was almost never a trivial choice; for example, the NoSQL solution of AWS, DynamoDB, was initially considered as the backing storage technology, but we found that it was too complex for the little advantage that we got out of it, compared with the much simpler S3. When looking at the compute platforms of AWS, we stuck to the more traditional virtualization solution of EC2, but halfway through the development, Lambda and ECS started standing out as better alternatives; however, since both the design and the implementation were already depending on EC2, we decided against correcting that decision. When looking at orchestration solutions, we found that in AWS there are three similar but different services: Cloudformation, DevOps, and Elastic Beanstalk; the first, Cloudformation, is the most universal of the three, and in a sense, it is similar to Ansible, but with an important difference: Cloudformation does not handle software deployment and configuration, which must be handled separately. Therefore, we looked at third party software, and found three tempting solutions: Ansible, Puppet, and Salt; of these three, we kept Ansible, because of its agentless architecture described in appendix C. In summary, many of the decisions taken during this project are far from trivial, and the solutions we propose are not the most optimal.

One of the most interesting conceptual leaps crossed in this project is between open data and agent-based modelling. On the one hand, a previously unseen amount of public information has been published, following the trend of open data introduced in section 1, but it tends to lack standardization, and datasets are usually large; on the other hand, agent-based modelling presents an attractive convergence between social studies and computer programming, that can be scaled up with high performance computing hardware, like Pandora does. It is logical to assume that the ability of simulating large, dynamical human environments of agent-based modelling can only be improved by using data obtained in the real world. Together, these two concepts present a feasible way of thinking about the future, and, especially, about testing hypotheses about it.

The library we have used for ABM simulation, Pandora, was suggested to me during development. It presents a very interesting approach to model development, with its twin Python/C++ interface. However, we found that, because of the strategy of splitting the agents’ behaviour in three methods, some control over the specific synchronization mechanisms is lost; this is an acceptable sacrifice, since it simplifies the programming interface considerably.
4.1 Future work

For the design of the infrastructure that we have presented, several improvements could be made. First of all, the communication model could be extended to follow a non-linear data flow, instead of the pipeline. This would allow stages to dynamically route the data through different components, feed data back to previous stages, or even branch off into multiple concurrent stages. This would substantially increase the flexibility of the software architecture, since moving away from SQS to SNS for distributed messaging would imply that we are no longer restricted to just one recipient per message.

As mentioned in section 1.3.5, there are other options for computer resources, beyond EC2, such as Lambda. With this service, we could abandon the instances that run the Python daemons, which are inactive most of the time, and use Lambda environments that are automatically executed in response to new messages in the messaging queues. This would be easier to do after moving to SNS for messaging, since integrating Lambda with SQS is not nearly as easy as integrating Lambda with SNS.

Another interesting service that offers a lighter environment for compute resources is EC2 Container Service, or ECS, which allows users to run Docker containers on EC2 hosts. Thanks to Docker, ECS presents a much better portability over bare EC2 instances, since the Docker containers are isolated units that produce repeatable environments, no matter what operating system they are run on. Although, perhaps what is most interesting about ECS, is that only the EC2 host, the EBS volume(s), and the network bandwidth are charged: the ECS containers themselves are free of charge, which means that we could run several of the lighter-weight architectural components inside a single EC2 instance, serving as an ECS host, potentially cutting some running costs. This would be relatively easy to integrate with existing infrastructure, since Ansible offers native support for ECS (although in Extra modules, instead of the other AWS modules, which are Core Ansible modules). Even though Amazon provides support for Docker with ECS, it is possible to run other container virtualization solutions on AWS, like Kubernetes\(^{31}\), although without native support from Amazon. Kubernetes is a container orchestration framework developed by Google, released under the Apache License, Version 2.0.

The current implementation of the pipeline stages could also use some optimization. For example, in the source downloading stage, we discussed two alternative strategies for efficiently downloading files from the Internet. For very small files, Lambda environments could be used, as long as it is certain that all files can be downloaded and uploaded in less than five minutes. For avoiding the need to allocate large EBS volumes, in case we are downloading very large files, a more complex strategy involves using the multipart upload feature of S3: we could upload chunks of the (incomplete) file, as it is being downloaded, so that we can reclaim some of the storage in the EC2 instance running the task. As explained in section 3.2.2, there is not much else we can do to improve the performance of this stage, since the bottleneck will almost always be in the network route through the Internet, well beyond our control. For the source scraping stage, we have integrated a Hadoop cluster in our infrastructure, and we have provided mechanisms for automatically provisioning it, and running tasks on it. That being said, a more specific framework, better geared towards data extraction and analysis, could be researched, or even developed.

\(^{31}\text{http://kubernetes.io/}\)
The pipeline could be extended beyond what we have considered to include new stages for doing new kinds of tasks, such as web scraping and indexing, machine learning, data warehousing, etc. These stages could be communicated with the others with mechanisms very similar to what we have already presented, or with the improved implementation that was discussed at the beginning of this section.

One such stage would be a specialized optimizer component: this would wrap around the simulation stage, and it would, essentially, search a model-specific parameter space; a simple implementation would use a brute-force strategy, but a smarter one would be more akin to an artificial intelligence. However, this would require a model that is correctly optimized, so that it can be executed many times without a significant performance penalty.

Finally, a closer integration of CARTO would be wise, because it offers powerful tools for data visualization, and even some analysis. Since Pandora is able to interpret GDAL geo-rasters, plotting simulation data from Pandora in CARTO, automatically, should not be too hard.
References


Appendix A  Python worker scripts

In this appendix, we give an overview of the Python implementations of the daemons that control the pipeline stages, and process the JSON configuration files that define the tasks to be executed on demand. This is, mostly, ‘glue’ code, in the sense that its purpose is to connect large components that cannot directly talk to each other. Each instance of the stages of the pipeline runs a particular program that implements the specific behaviours, and accesses the specific services and resources required for the task.

All of the programs share a common set of dependencies on a single virtualenv (four, plus their own dependencies): the AWS API, or boto3, python-daemon-3K, a library for turning Python programs into UNIX daemon processes, paramiko, a Python-native implementation of SSH, and requests, an HTTP client library.

Daemons do not attach to an interactive session; instead, their execution is managed by the operating system, or, more specifically, by the init program, which is the first process to run during the boot process, and the last when the machine is powered off. This allows us to reboot a machine without having to tell it to run a program, while avoiding the need to hold an active session to that machine. Daemons are notoriously tricky to program, because of the many quirks involved in making it a well-behaved one. Unfortunately, even though PEP 3143 [2] addresses this problem, no Python modules that implement such a behaviour have been standardized; we use the unofficial python-daemon-3K for that. This module provides a few classes that vastly simplify the execution of a daemon, by providing simple, Python-native interfaces: DaemonContext accommodates the changes in the execution environment to run as a daemon, from inside a with block (for example, redirects standard output to a logfile); PIDLockFile wraps the creation and deletion of a lockfile, that prevents a program from being run from multiple processes at once (a common restriction for daemons).

For communicating our cloud with the HPC environment, we use SSH to dynamically run commands in it. We could use the regular OpenSSH implementation that is almost always available in GNU/Linux, from Python, but the paramiko library is better integrated into the language, so this is what we use. With this library, we can not only send commands, and receive their output, we can also send and receive files, with the Secure Copy Protocol (SCP), over SSH. The address of the remote host, and the name of the remote user, are given by the tags of the EC2 instance that the daemon runs on, as explained in section 3.2.4.

We use the requests library, which provides a human-friendly API for working with HTTP, for downloading source files in the first stage, for querying the EC2 tags of the instance in the scraping and simulation stages, and for sending data to the CARTO SQL API.

There are six main programs: master.py, download.py, scrape.py, optimize.py, sync.py, and status.py. With the exception of sync.py, which immediately synchronizes the configuration bucket with the master instance, all scripts are UNIX daemons that run in the different parts of the pipeline. The first one, master.py, is responsible for listening on the frontend SQS queue, and launching jobs in the rest of the infrastructure, with the configurations referenced in the configuration bucket, to which it has exclusive access. The second daemon, download.py, has the code that downloads files from the Internet, and uploads
them to our S3 bucket, and forwards the message to the next stage when it is done. In the
next stage, scrape.py receives messages from the SQS queue, generates a corresponding
configuration of an EMR cluster, and launches it, with hardware configuration, software
steps, and so on. An extra step is always added at the end, which runs the scrape2status.py
auxiliary script that notifies the status queue from the master node of EMR; another step
runs scrape2simulate.py, if the job description includes a simulation stage, to which it
forwards the SQS message. Then, the optimize.py daemon connects to the HPC environ-
ment, puts the agent-based model on it, runs it, and displays the results on our GIS, by
calling the script(s) provided by the job description.

When the programs want to refer to AWS resources, they need some sort of identifier. In
our case, we choose the short name, normally used because it is readable by humans. In
the case of S3, the names of buckets must be unique across all user accounts in AWS,
so the ones we have defined cannot be reused by someone else. The code we provide
contemplates the possibility that the users may want to choose other names for the in-
frastructure components. For this reason, resource identifiers are never hardcoded into
the Python programs that manage infrastructure; instead, they read them from an external,
JSON-formatted file, called pipeline-names.json, which is read and assigned to the global
variable names, inside the auxiliary script pipeline.py. Since the infrastructure is created
by the Ansible playbooks, these need to read the same file, and create the resources with
the names that the Python programs expect. With this mechanism, we let our users change
the names of the AWS resources from one single file, and not multiple ones. However, it
must be said that the configuration files given in section 3.5 do have AWS resource names
hardcoded in them; the reason is that that code does not manage infrastructure (it is, in
fact, being managed by it!), so it only needs to access the resources it knows.
Appendix B  Execution environment and setup

All code that is to be remotely deployed in a VM has been packaged in an RPM (RedHat Package Manager) file. RPM files are usually obtained from the remote repositories of the distribution in use, although, in our case, we will distribute it through S3, and install it locally. An RPM package is able to install files to directories, set UNIX permissions on them, and, in general, run arbitrary commands to wrap the installation procedure. In our case, we adhere to the Filesystem Hierarchy Standard (FHS) [5].

RPM packages are created with `rpmbuild`\(^2\), a toolkit released under GPL. After installing it in the developer’s computer, a certain directory structure has to be created at the user’s home directory (see section 2.1 for that), and a `.spec` file has to be provided. `rpmbuild` exposes certain environment variables that assist the building machine to address the special directories that will be used (for example, `/usr/bin` would refer to the executables directory on the machine used for building the package, whereas `%{_builddir}%{_bindir}` would refer to the executables directory on the installing machine, even though they refer to the same name). Then, the sources have to be compressed in a `.tar.gz` file, and copied to the specific folder of `rpmbuild`. Finally, we call the `rpmbuild` executable, and an `.rpm` file is created.

```
Name:       hpopt
Version:    %VERSION%
Release:    %RELEASE%
Summary:    Daemons to run the various nodes of hpopt
License:    GPLv3
BuildArch:  noarch
Source:     ~/rpmbuild/SOURCES/hpopt-%{version}-%{release}.tar.gz
URL:        https://bitbucket.org/lluisisern/hpopt
Requires:   python34, python34-pip, aws-cli
BuildRoot:  %(_tmppath)%{name}-%{version}-%{release}
```

Listing 10: Header of the `hpopt-rpmbuild.spec` file

The header of the `.spec` file contains a set of key-value pairs with relevant information that `yum`, the RPM package manager, needs. This includes the name of the package/program, version and release number (e.g. 1.2.3-4), a short description, the license (e.g. GPL, Apache, BSD, etc.), the target architecture (e.g. x86_64, i386, noarch, etc.), the path/URL to the source code tarball, the homepage of the package/program, and the list of packages that are required for building/running the program (in this case, `python34`, `python34-pip`, and `aws-cli`, which is already pre-installed on Amazon Linux). Note that the `Version` and `Release` variables do not have an actual value; this is because we will later replace these values with the actual ones when we execute `make-rpm.sh`.

```
%description
Daemons to run the Master, Download, Scrape, and Optimize nodes of the hpopt infrastructure. Includes a script to synchronize configuration files.

%prep
%setup -q
```

Listing 11: `%description` and `%prep` sections of the `hpopt-rpmbuild.spec` file

\(^2\)http://www.rpm.org/
After that, we find the first two proper sections: \%description, which contains a longer description with, optionally, multiple paragraphs, and \%prep, which sets up the environment and files for the imminent build (it deletes old files, and fetches the sources).

\%
\%pre
getent group hpopt >/dev/null || groupadd -r hpopt
getent passwd hpopt >/dev/null || useradd -r -g hpopt -s /sbin/nologin \ -d /etc/hpopt hpopt
\%

Listing 12: \%pre section of the hpopt-rpmbuild.spec file

The next section is \%pre. The code here is not executed on the machine used to build the RPM, only on the targeted machine. This section ensures that an unprivileged user exists, or is created otherwise, for the application to run as. The reason for this is that we will be executing our scripts as a daemon, which implies that they will be run by \texttt{root}; what we do is to immediately drop privileges by changing the UID and GID of the process (for a production-friendly implementation, we should assign the UID and GID of \texttt{hpopt} user and group to a static value for added resiliency to errors), as soon as the program starts. Daemons run in the background, so we cannot expect them to be started from an interactive session.

\%
\%install
r
m -rf \{buildroot\}
install -d \{buildroot\}\%{datadir}/hpopt
install -d \{buildroot\}\%{_sysconfdir}/hpopt
install -d \{buildroot\}/var/log/hpopt
[ -d \{buildroot\}\%{_initddir} ] || install -d \{buildroot\}\%{_initdir}
[ -d \{buildroot\}\%{_bindir} ] || install -d \{buildroot\}\%{_bindir}
install initscripts/hpopt-download \{buildroot\}\%{_initdir}
install initscripts/hpopt-master \{buildroot\}\%{_initdir}
install initscripts/hpopt-optimize \{buildroot\}\%{_initdir}
install initscripts/hpopt-scrape \{buildroot\}\%{_initdir}
install initscripts/hpopt-status \{buildroot\}\%{_initdir}
install download.py \{buildroot\}\%{datadir}/hpopt
install master.py \{buildroot\}\%{datadir}/hpopt
install optimize.py \{buildroot\}\%{datadir}/hpopt
install scrape.py \{buildroot\}\%{datadir}/hpopt
install status.py \{buildroot\}\%{datadir}/hpopt
install sync.py \{buildroot\}\%{datadir}/hpopt
install pipeline-names.json \{buildroot\}\%{datadir}/hpopt
install requirements.txt \{buildroot\}\%{datadir}/hpopt
touch \{buildroot\}/var/log/hpopt/download.log
touch \{buildroot\}/var/log/hpopt/master.log
touch \{buildroot\}/var/log/hpopt/optimize.log
touch \{buildroot\}/var/log/hpopt/scrape.log
touch \{buildroot\}/var/log/hpopt/status.log
ln -s \{datadir\}/hpopt/download.py \{buildroot\}\%{bindir}/hpopt-download
ln -s \{datadir\}/hpopt/master.py \{buildroot\}\%{bindir}/hpopt-master
ln -s \{datadir\}/hpopt/optimize.py \{buildroot\}\%{bindir}/hpopt-optimize
ln -s \{datadir\}/hpopt/scrape.py \{buildroot\}\%{bindir}/hpopt-scrape
ln -s \{datadir\}/hpopt/status.py \{buildroot\}\%{bindir}/hpopt-status
ln -s \{datadir\}/hpopt/sync.py \{buildroot\}\%{bindir}/hpopt-sync
\%

Listing 13: \%install section of the hpopt-rpmbuild.spec file

Next section is \%install, which runs on the building machine only, and operates on a ‘fake’ directory structure that represents the target system. This section creates the necessary directory structure, installs files from sources to the system, creates symlinks, touches empty files, and so on. Later, the files and directories created during this stage will be replicated.
on the machines that install the RPM. As mentioned earlier in this section, we install our software according to the FHS. This ensures good interoperability and portability, should we want to change the operating system used to run the software. According to the FHS, we should put configuration files in `/etc`, init scripts in `/etc/init.d`, architecture-independent data in `/usr/share` (Python scripts in our case), user executables (or, alternatively, symbolic links to actual executables) in `/usr/bin`, and log files in `/var/log`. The names of the Python scripts that we install are too generic, so, in order to avoid $PATH collisions, these are installed under `/usr/bin` with the `hpopt-` prefix, and without the .py extension.

```bash
%post
  echo "hpopt ALL=(ALL) NOPASSWD: /sbin/poweroff" >> /etc/sudoers.d/hpopt

%clean
  rm -rf %{buildroot}
```

Listing 14: %post and %clean sections of the hpopt-rpmbuild.spec file

After that, there is the %post section. The code here is not executed on the building machine, only on the target machine. We use this section to create a new file on the target’s `/etc/sudoers` directory that will permit the `hpopt` unprivileged user to `poweroff` the system (needed for inscaling the `downloaders` cluster, as explained in section 3.2.2). The last section with executable code is the %clean section, which simply removes files generated by the `rpmbuild` process, in order to prepare for the next build, or to simply reclaim used space on the hard drive.

```bash
%files
  %attr(0755, hpopt, hpopt) %{_sysconfdir}/hpopt/
  %attr(0755, root, root) %{_initddir}/hpopt-download
  %attr(0755, root, root) %{_initddir}/hpopt-master
  %attr(0755, root, root) %{_initddir}/hpopt-optimize
  %attr(0755, root, root) %{_initddir}/hpopt-scrape
  %attr(0755, root, root) %{_initddir}/hpopt-status
  %attr(-, root, root) %{_bindir}/hpopt-download
  %attr(-, root, root) %{_bindir}/hpopt-master
  %attr(-, root, root) %{_bindir}/hpopt-optimize
  %attr(-, root, root) %{_bindir}/hpopt-scrape
  %attr(-, root, root) %{_bindir}/hpopt-status
  %attr(-, root, root) %{_bindir}/hpopt-sync
  %attr(0755, root, root) %{_datadir}/hpopt/download.py
  %attr(0755, root, root) %{_datadir}/hpopt/master.py
  %attr(0755, root, root) %{_datadir}/hpopt/optimize.py
  %attr(0755, root, root) %{_datadir}/hpopt/status.py
  %attr(0755, root, root) %{_datadir}/hpopt/sync.py
  %attr(0644, root, root) %{_datadir}/hpopt/pipeline-names.json
  %attr(0644, root, root) %{_datadir}/hpopt/requirements.txt
  %attr(0644, hpopt, root) /var/log/hpopt/download.log
  %attr(0644, hpopt, root) /var/log/hpopt/master.log
  %attr(0644, hpopt, root) /var/log/hpopt/optimize.log
  %attr(0644, hpopt, root) /var/log/hpopt/sync.log
  %attr(0644, hpopt, root) /var/log/hpopt/status.log
```

Listing 15: %files section of the hpopt-rpmbuild.spec file

Usually, the last section is %files. Here we find no executable code; instead, there is a list of files and directories that should have been created by the %install section (and others), together with permission masks and owning user and group information, which will be created in the target system. If, when building the RPM, the hierarchy specified here does not match the actual files and directories present, a warning will be emitted by `rpmbuild`. Additionally, on the targeted system, these files’ permissions and owner information will be checked, and, if necessary, modified, to match the values specified here.
The process of generating the RPM has been automated with a Bash script that can be seen in listing 16. This script begins by reading two files, `rpm/version`, `rpm/release`. Then, it creates a `.tar.gz` archive with the Python and Bash sources, and places it in the `rpmbuild` directory. Then, it invokes `rpmbuild`, and copies the resulting `.rpm` to the final destination: back to the current working directory. After generating the `.rpm` file, we should upload it to the `deploy` S3 bucket, so that EC2 instances can use it to bootstrap, as explained in the next paragraph.

In AWS, EC2 instances are usually provided a small Bash script at the time of creation that ‘bootstraps’ the system, or, in other words, installs our Python scripts with their dependencies, creates the needed directory structures, and registers one of the initscripts. An initscript is a regular Bash script whose environment and execution is managed by the operating system; in this case, it launches a daemon, which sleeps in the background, waits for SQS messages to arrive, and launches worker threads in response. The standard output and standard error streams are redirected to a `logfile`. The Python programs are programmed to drop privileges as soon as they are invoked; because they are started as `root`, we change the UID and GID of the processes, so that they cannot read or write where they are not supposed to.
## Appendix C  Orchestration with Ansible

The administrators can manage the infrastructure with Ansible. Orchestrating with Ansible has one main advantage: it is based on an agentless architecture, which means that no special software has to be installed in the target machines (other than the SSH server), and thus we can use already existing authentication mechanisms. When installed on the administrator’s system, a number of predefined tasks are available to automate the most common operations. In order to use it with AWS resources, the `boto3` AWS library must be installed, with credentials available to it: the AWS user credentials, and one or more SSH public-private keypairs, shared with the EC2 instances one wishes to manage.

Even though the entire infrastructure in AWS could be managed with Ansible (with the exception of EMR, but there are unofficial modules), we only intend it to be used in a non-automated way. Because it is not cost effective to have EC2 instances running for long periods of inactivity, some parts of the infrastructure are ephemeral, in the sense that they get created dynamically, and auto-terminate when they finish their work, like the EC2 instances inside the source downloading cluster, or EMR clusters. On the other hand, creating IAM roles and policies, S3 buckets, SQS queues, EC2 ASG policies, CloudWatch alarms, and synchronizing configuration and deploy files with the cloud can be done with supervision from a human, whereas adding and removing instances to AutoScaling groups, and creating and terminating EMR clusters is best left to automated mechanisms. That being said, for more complex automation, Ansible operations, or even playbooks, could be called from the daemons, and not by administrators, directly. Ansible has a Python API that could allow our code to programmatically use Ansible operations and resources; as of the latest version, 2.2.0, support for Python 3 has become available, but is still in early stages, and is offered only as a technology preview.

When an EC2 instance is created, the SSH software generates a fingerprint as part of the bootstrapping procedure, and dumps it in the system log; the administrator can download the system log of that instance through a secure HTTPS channel from AWS, and ensure that the IP of that instance, and the newly generated SSH fingerprint coincide. This fingerprint lasts as long as the instance, so if it is ever terminated and re-launched, a new one would be generated. Should this happen, the SSH client would warn the user of a possible man-in-the-middle impersonation attack, so the old fingerprint would have to be erased from the administrator’s hard drive, and the new one validated.

In the default installation of Ansible, we find a series of modules that perform common operations on most of the AWS services, like EC2, S3, IAM, and so on. These modules are either run on a local Ansible connection, or on a remote host; this enables us to use already present credentials, such as SSH keys, and AWS CLI credentials. For example, after we launch an EC2 instance from `localhost`, we can wait for it to boot up, open an SSH connection to it, and then run remote commands on it.

See listing 17 for an example on an Ansible playbook. When the command `ansible-playbook file.yaml` is called, Ansible checks if the EC2 inventory associated to the credentials being used has such an EC2 instance with the given specifications; otherwise, Ansible tries to change the settings, or deletes and re-creates the instance, if that fails. In general, commands in Ansible should be designed around the concept of idempotency, which means...
# Launch an EC2 instance

- name: launch EC2 node
  hosts: localhost
  connection: local
  gather_facts: false
  tasks:
    - name: launch EC2 instance
      ec2:
        instance_tags:
          Name: MyInstance
        instance_type: t2.micro
        image: ami-f9dd458a
        region: eu-west-1
        exact_count: 1
        count_tag:
          Name: MyInstance
        wait: yes

Listing 17: Example of a simple Ansible playbook for EC2

that tasks should be repeatable. A given command makes its change once, regardless of how many times it is called, if and only if it is idempotent. In the example of listing 17, under the Ansible key `instance_tags`, the EC2 instance is given the tag `Name`, with value `MyInstance`; a few lines after that, in the keys `exact_count` and `count_tag` we specify that there should be exactly one instance with the given tag. Accordingly, Ansible will launch a number of EC2 instances that matches the given tag and count; if `exact_count` and `count_tag` were not present, then Ansible would launch a new EC2 instance every time this playbook is called, which does not conform to the definition of idempotency.

For orchestrating our system, we provide a set of Ansible playbooks that deal with separate parts of the infrastructure. The reasoning behind this is that, even though most components are inter-dependent in some way or another, modifications to the infrastructure will usually happen separately, so that, if, for example, an IAM role is slightly modified, or, if a new SQS queue is created, we do not have to go over all AWS resources when most of them will not be affected. In general, this does not introduce management problems; however, special care must be taken by the administrator so that, when updating some parts of the infrastructure, old things do not break.

The playbooks are split in the following scheme: `create-iam.yml` contains only IAM roles and users, and creates the IAM policies they require; `create-sqs.yml` creates and configures the SQS queues; `launch-downloaders.yml` creates the EC2 AutoScaling group for the source downloading stage, with its outscaling policy, but with no initial capacity (no EC2 instances are actually launched); `launch-master.yml` launches the `master` instance; `launch-scrape.yml` launches and configures the instance that manages the SQS queue of the scraping stage; lastly, `launch-optimize.yml` launches and configures the instance that manages the SQS queue of the simulation stage, and also uploads results to CARTO; finally, `sync-buckets.yml` ensures the required S3 buckets exist, and are accessible by the required components, and synchronizes local data (like code and configurations) with S3.

Since the AWS services are referred to in the playbooks by name, we read those from an external file, `pipeline-names.json`: this is a simple JSON-formatted file which contains specific names of the AWS infrastructure that is to be managed. This is done so that names can be changed easily by editing one file (and not multiple ones).
Appendix D  Software licensing

<table>
<thead>
<tr>
<th>Software</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Web Services</td>
<td>AWS Customer Agreement</td>
</tr>
<tr>
<td>Ansible</td>
<td>GNU General Public License, v3.0</td>
</tr>
<tr>
<td>CARTO APIs</td>
<td>BSD 3-Clause</td>
</tr>
<tr>
<td>Git</td>
<td>GNU General Public License, v2.0</td>
</tr>
<tr>
<td>Hadoop</td>
<td>Apache License, v2.0</td>
</tr>
<tr>
<td>Mesa</td>
<td>Apache License, v2.0</td>
</tr>
<tr>
<td>OpenStreetMap</td>
<td>Open Data Commons Open Database License</td>
</tr>
<tr>
<td>Pandora</td>
<td>GNU General Public License, v3.0</td>
</tr>
<tr>
<td>Paramiko</td>
<td>GNU Lesser General Public License, v2.1</td>
</tr>
<tr>
<td>Paramiko SCP module</td>
<td>GNU Lesser General Public License, v2.1</td>
</tr>
<tr>
<td>PyCharm (Community)</td>
<td>Apache License, v2.0</td>
</tr>
<tr>
<td>Python</td>
<td>Python Software Foundation License</td>
</tr>
<tr>
<td>Python-daemon-3K</td>
<td>Python Software Foundation License, v2</td>
</tr>
<tr>
<td>Requests</td>
<td>Apache License, v2.0</td>
</tr>
<tr>
<td>RPM Package Manager</td>
<td>GNU General Public License, v2.0</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>Multiple (free software compatible by default)</td>
</tr>
<tr>
<td>Virtualenv</td>
<td>MIT License</td>
</tr>
<tr>
<td>Virtualenvwrapper</td>
<td>MIT License</td>
</tr>
</tbody>
</table>

Table 5: Licenses of software used in this project

In table 5 we see the licenses of all of the software we have used to compose the final infrastructure on AWS, including any software that we have used during the development and testing phases of the project. Most of these are permissive free software licenses, like the Apache License, in its 2.0 version, the Python Software Foundation (PSF) Library, the MIT License, and the BSD License. Then, there are the non-permissive free software licenses: the GNU General Public License, or GPL, in its various versions (the 2.0, the lesser 2.1, and the 3.0). Finally, there is the Open Data Commons Open Database License (ODbL), used by OpenStreetMap: this license is designed with open data in mind, in the sense that the data under this license can be freely shared, used, and modified, as long as it is shared alike. Finally, another license stands out: the AWS Customer Agreement. This governs the usage and access of the services that Amazon offers; it defines that “Our Submissions” are subjected to the Apache Software License, unless another kind was specified at the time of submission, which we did not. Additionally, they define the Amazon Software License, which governs the usage of their patented software, by the customers.

In section 1.1, we established the intention of letting others improve and build upon our work; therefore, we must make sure that we release our code as free software. The most suitable license for this purpose is the GNU General Public License, Version 3.0, or GPLv3. Also, if we had not used a GPL license, we could not have used GPL-licensed libraries. In order to do that, we provide a copy of the license’s full legal text, in the file LICENSE found at the root of the source code; then, all the source code files that belong to our software contain a comment stating that the file is part of a piece of software, licensed under GPLv3. This license ensures that our code remains a free project, that can be used and improved by anyone, for any purpose.
Appendix E  Bootstrap script for the ABM tests

The code shown here is passed to the EC2 instances as their user-data field, so that they build and install Pandora and Mesa, run the performance tests, and store the results in S3.

```
#!/bin/bash

# Update and upgrade packages
apt-get -y update
apt-get -y upgrade

# Install dependencies
apt-get -y install packaging-dev python3 python3-dev python3-pip \
    python3-matplotlib python3-pandas awscli libtinyxml-dev \
    libqt4-dev librpng1-dev mpich2 scons build-essential \
    libboost-random-dev libboost-python-dev libboost-filesystem-dev \
    libboost-system-dev libmpich2-dev libboost-test-dev libboost-timer-dev \
    libboost-chrono-dev git

# Install Mesa and untangle
pip3 install mesa untangle

# Create HDF5 directory
mkdir ~/hdf5

# Download HDF5 and install
cd ~/hdf5
wget "http://support.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.17.tar.bz2"
tar xf hdf5-1.8.17.tar.bz2
cd hdf5-1.8.17
./configure --enable-parallel --prefix="/usr/local/hdf5" --disable-shared --with-pic
make
make install

# Create Pandora build directory
mkdir ~/pandora-build

# Clone Pandora repository
cd ~/pandora-build
git clone "https://github.com/xrubio/pandora.git" .
scons
scons install

# Export Pandora and Mesa paths
export PANDORAPATH=/usr/local/pandora
export PATH=$PATH:$PANDORAPATH/bin/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PANDORAPATH/lib/

# Install tests
for i in pandora pypandora mesa; do
    aws s3 --region eu-west-1 cp ${tests_bucket}/tests/${i}.tar.xz .
tar xf ${i}.tar.xz
done

# Run tests
for i in pandora pypandora mesa; do
    cd ~/pandora
    scons
    ./test_pandora > results.txt
    cd ~/pypandora
    ./test_pandora.py > results.txt
    cd ~/mesa
    ./test_mesa.py > results.txt
    cd ~
    inst_type=$(curl http://169.254.169.254/latest/meta-data/instance-type)
    now=$(date +"%d_%m_%Y_%H%M%S")
    for i in pandora pypandora mesa; do
        aws s3 --region eu-west-1 cp "${(tests_bucket)}/${i}.results.txt" $inst_type-$(now)/$i/
done
    aws s3 --region eu-west-1 cp /var/log/cloud-init-output.log \
        $(tests_bucket)}/${inst_type}-${now}/
    poweroff
```

Listing 18: Bootstrap script for the ABM performance tests, in Bash