University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Piotr Styczyński
Student no. 386038

Michał Balcerzak
Student no. 385130

Michał Ołtarzewski
Student no. 382783

Gor Safaryan
Student no. 381501

AWS Cost Optimization Tool

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Instytut Informatyki

February 2019

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Bachelor of Computer Science.

Date Supervisor's signature

Authors' statements

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Authors' signatures

Abstract

Authors describe the design and implementation process, in-depth system and code architecture as well as used communication protocols, storing methods and other internals of the AWS Cost Optimization System.

Keywords

AWS, Amazon Web Services, cloud computing, cost optimization, cost management

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

D. Software

Tytuł pracy w języku polskim

Narzędzie do Optymalizacji Kosztów AWS

Contents

Chapter 1

Introduction

1.1. Overview

Cloud computing has become recently one of the most important paradigm shifts in the area of real world software engineering. It has reshaped the whole process of how applications are developed and reduced the amount of upfront investment required to start an internet business. While commercial cloud computing services were first offered in 2006 by Amazon Inc, the original idea and preliminary implementation traces back to Multics OS developed by MIT, GE and Bell Labs. However the idea of time-sharing systems that was the ancestor of further cloud concept was widespread in 60ies [**Markus**].

The term "Cloud computing" can refer to every layer of application stack: hardware, hosting platform, software and even to a single function. Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide these services. The services themselves have long been referred to as Software as a Service (SaaS) [**Armbrust**]. Generally speaking cloud can be perceived as a shared pool of computer resources such as computing capacity, transient and persistent memory, which can be acquired or released on demand. The undisputed power of cloud computing constitutes in its elasticity and granularity: it allows users to ask for hundreds of computers for only 5 minute usage which are shipped during several minutes. Such services are usually offered over remote network connection and users are billed for the portion of the resources they have used. Depending on the cloud infrastructure type the payment models can be different [**Laatikainen**], but the common spendings are associated with data storage, data transfer, and computing timeshare.

Nowadays the industry increasingly relies on cloud technologies. More and more companies start or move their products to cloud environment. Unfortunately it comes with additional financial costs imposed by cloud providers. In order to make business profitable companies try to reduce amont of money they are supposed to pay to the minimum. Despite different solutions, like employing specific cloud cost optimization team, more and more firms decide to benefit from dedicated software, which is supposed to help manage and optimize their cloud usage. It is not easy to choose the right tool having that many choices.

There are some notable solutions of AWS cloud optimization problem – *AWS Cost Explorer* and *AWS Cost Management*, *Cloudability*, *Apptio* and others widely used nowadays. In spite of that fact there is still place for new tools targeting omitted types of clients or wrapping and bundling the greatest features from exisiting ones.

1.2. Aim of the thesis

The primary objective of the thesis is to create a tool complementing existing solutions used in cloud cost optimisation. We focus on Amazon Web Services platform maintained by Amazon as it is one of the most commonly used. In our tool, called *Omigost*, we will try to target small/middle sized companies creating simple and easy to use software with flexible configuration options.

1.3. Structure of the thesis

The thesis is structured as follows. In Chapter 2 we describe the problem of cloud cost optimization with additional description of selected existing solutions. Then, in Chapter 3 we present our solution in detail. We mention, among others, whole system architecture, API and configuration. Finally, in Chapter 4 we sum up the whole thesis. In Appendix A we describe how to introduce our solution in a company.

1.4. Contribution of each author

It is important to mention that each author worked to some extent on every part of the thesis. However authors contributed mainly to the following parts:

- Michał Ołtarzewski
  - Software development process management
  - Design of backend architecture
  - Backend part implementation
- Michał Balcerzak
  - Research of existing solutions
  - Design of backend architecture
  - Backend part implementation
- Piotr Styczyński
  - Design and prototype of visual part of tool
  - Frontend part implementation
- Gor Safaryan
  - Research of AWS APIs and SDKs
  - Design of backend architecture
  - Backend part implementation

Chapter 2

Problem statement

2.1. Motivation

As there are plenty of various billing models for cloud services [**GLaatikainen**], the effective management of them became a tough problem. The ease of resource allocation led to situation when tracking tiniest details of billings is an unaffordable challenge.

The tooling that exists is targetting wideworld-scale companies that are able to require expensive licenses and hire cost-optimization teams. The software as it is in case of Cloudability is too complex for average user and does not provide easy way to incorporate custom business flows into the tool. Amazon, as one of the leading cloud providers, offers different tools for exploration of expenses. Most commonly used options are either their public APIs [**AWSCostManagement**] or specific SDKs supporting lots of languages. Unfortuantely previously mentioned tools are rather simple and do not satisfy all the needs of potential clients.

The common case that is unresolved is the distribution of research and development resources. We observed that there exists no tool that would support request for resources of individual worker with regards to custom management propagations as specified by client bussiness model. Cloudability [**CloudabilityAlerts**] offers simple alerts, but they lack Slack support and beforementioned propagation abilities.

There exists an obvious gap in the market, that our solution, *Omigost*, will try to cover. Having one versatile tool removes the need for using few detached pieces of software. The most important aspects of our tool are intuitive interface and different types of user notifications, which highly increases spendings clarity and helps in decisions connected to costs cutting. We hope it will allow companies to focus more on providing value to their clients having lower costs in the same time. Problem we would like to solve is lack of the tool that has simultaneously the following features:

- Free and Open Source
- Easy cloud management without complex knowledge
- Intuitive interface for individual workers to request resources
- Notifications for cost surpassing and redundant resources
- Manage notification well – only significant alerts
- Highly configurable and flexible
- Integration with communication via Email and Slack

2.2. Overview of exisiting solutions

Businesses that rely upon cloud services often reach a point where resources they are using up gradually become less and less manageable. As the problem is well known to the cloud market, both Amazon and other third-party companies made attempts to fulfill these needs by creating custom software fitting certain roles in optimizing AWS expenses that include:

1. Configuring budget limits and alerting users when they are exceeded
2. Instance alerting management
3. Cost analytics

Some of the most prominent tools currently available on the market that improve resource management experience for AWS cloud are described in the following sections. Every description is supposed to show key features crucial for cost optimization.

2.2.1. AWS Budgets

AWS Budgets[**AWSDocs**] is a part of Amazon Web Services that allows to set limits of a certain types that apply to a chosen period of time. When a limit is either exceeded, close to be exceeded or is forecasted to exceed the configured threshold before the end of that period, the administrator of that account is notified by email. Types of resources one can put this kind of a budget on include:

1. Money spent in total or on a certain type of machines.
2. Utilisation of selected services.
3. Utilisation or coverage of reserved instances.

2.2.2. AWS Cost Explorer and AWS Cost Management

AWS Cost Explorer[**CostExplorer**] enables access to all budget data. User can define and generate custom reports in a form of a data chart spanning a selected time interval with chosen time granularity of the samples. AWS Cost Explorer is also a basis for AWS Cost Management, which is basically a set of predefined reports that form an easily accessible dashboard.

2.2.3. Cloudability

Atlassian's Cloudability[**Cloudability**] delivers a budget system functionality analogous to AWS Budgets along with tools for predicting future spendings and presenting the real cost of AWS resources in utilisation. In comparison to Amazon's native tools Cloudability also allows management of multiple accounts in the same time. It saves effort of having to set up budgets separately in every owned account.

2.2.4. Apptio

Apptio[**Apptio**] provides a set of tools that mainly focuses on analysis of expenses and their forecasts, managing them collaboratively, and planning future ones. They expose features that make it easier to discover underutilized resource, compare spendings with a database of similar benchmarks, organize resources into groups to make reports even clearer, and offer other useful management utilities.

2.2.5. Stax.io

Stax.io's[**Stax.io**] main focus is to provide insight about cost, wastage, compliance and cloud quality. It can analyse how cloud resources are used, measure quality of the way cloud is utilized, set up checks for business-compliance of our cloud with several standards and give customized advice on what could be optimized to reduce wastage, while also allowing for creation of custom views of data. Basic tools for budgeting instances, accounts, tags and more, monthly or annually, and configuring overspend alerts are also available there.

2.2.6. SnowSoftware

SnowSoftware's toolset, alongside fulfilling some of the more specific usecases like optimizing usage of software from SAP Software Solutions or optimizing and managing software licenses, also has tools that are dedicated to optimizing cloud costs.

Snow for SaaS attempts to give a holistic view about application usage including, among others, how SaaS applications are used on cloud and whether there are zombie virtual machines [**SnowSaaS**].

Snow Automation Platform suggests approach based on automated and preconfigured provision of resources. By pre-giving those resources a decommission date one can avoid issue of zombie instances. It is also possible to preconfigure budgets and schedule machine starts and stops to further optimize costs [**SnowBlog**].

2.2.7. Conclusions

Using resources available to us, we concluded that most of the competing tools available on the market fail to provide both instance budgeting and machine termination automation. A number of them also puts most of their emphasis on analysing usage data rather than helping with instance management. From tools listed above, Snow Automation Platform is the only one that features both budgeting and automating machine termination and stopping. However, it's approach is not to straightforwardly manage already existing machines, but rather to automate their provisioning.

2.3. Our Solution

TODO

Chapter 3

Project development

In this section we describe the way we developed the application – management and organization of work along with supporting tools.

3.1. Version control

During development as our version control system we use Git and whole code repository is hosted on GitHub. We have chosen this particular service because of a few reasons.

First of all GitHub [**GitHub**] is one of the most popular Git hosting service and also has one of the biggest open source communities. It is quite important if we consider possibility of future development of our tool even after finishing this thesis.

The second reason is integration with lots of useful applications. It is easy to use it with different services that improve development process like TravisCI [**TravisCI**] and Slac k[**Slack**].

Last but not least, GitHub has a built in issue tracker. It can be highly customised and makes project management easier. We created dashboard divided into a few sections, which separate tasks in different stages of development. Everything is automated – depending on actions performed by users like Pull Requests issues are moved between sections. In order to help prioritize tasks we introduced issues tagging system.

3.2. Continuous Integration

For Continuous Integration we use hosted service – TravisCI. It allows us to make sure new changes we introduce are compliant with the rest of our codebase. There are two reasons we have chosen this service: it is easy to configure using only one YAML file inside repository and it is free for open source projects. After every code submission, at the beginning TravisCI performs Smoke Tests trying to build the project and check whether it runs or not. Successful build is followed by both backend and frontend tests.

3.3. Communication

All of the communication happens via Slack. It is a platform allowing team members to communicate without use of email or group SMS. Various channels can be created there, with persistent messages history both public and private. There is also possibility to reach out to every member directly if needed. We use Slack due to the fact it is free and integrates easily with GitHub.

3.4. Organization of work

We work in iterations that last around one month. At the beginning we define and create new issues. During the iteration everyone chooses desirable task, completes it and makes corresponding Pull Request in GitHub. At the end we discuss our overall progress and plan next steps.

In order to let a new feature become part of the repository it has to be accepted. It means that Pull Request has to pass Continuous Integration system build and be approved by one of the reviewers. Every issue is solved in a separate branch and every PR is merged directly to the master branch, which allows us to group all commits that are part of one feature or issue.

3.5. Contact with the client company – Sumo Logic

Whole project and thesis are developed under Sumo Logic mentorship. There is one particular person designated to act as mentor – Jacek Migdał. He makes sure we are provided with every resource we need to work on the project without unnecessary breaks. Also our mentor helps revolve any ambiguity and gives technical advice.

Day to day communication with mentor takes place via Slack. Additionally approximately twice a month our team meets in Sumo Logic office to review current progress and overcome major difficulties. Every new feature is discussed beforehand with the company.

Chapter 4

Tool for AWS cost optimization

In this chapter we will describe the whole application – *Omigost*.

4.1. Use cases

In this section you can find descriptions of the most common use cases grouped by actors. Every use case assumes company in preconfigured in AWS Organizations.

4.1.1. Actor – AWS cost optimization admin

Use case	Adding employee to the system as a new user
Basic flow	1. Admin enters tab for users management 2. Admin fills username and selects accounts available for specific employee 3. Admin submits adding new user using button

Use case	Removing existing user
Basic flow	1. Admin enters tab for users management 2. Admin finds specific user in the user list 3. Admin clicks button for user removal 4. Admin confirms his intent in modal that showed after previous action
Alternative flow	In point 4. Admin notices mistake and clicks cancel button in modal. After that Admin starts again from point 1.

Use case	Adding communication to existing user
Description	Adding new destination for user notifications
Basic flow	1. Admin enters tab for users management 2. Admin fills fields describing new mean of communication 3. Admin submits communication creation 4. Admin selects specific user from user list 5. Admin selects communication from the list and adds it to the user

Use case	Setup of one budget
Description	Admin wants to create single budget for linked accounts/tags
Basic flow	1. Admin enters tab for budgets management 2. Admin enters budget value in dollars 3. Admin selects linked accounts/tags from lists connecting them to budget 4. Admin clicks submit button

Use case	Setup of multiple separate budgets
Description	Admin wants to create single separate budget for every specified linked account
Basic flow	1. Admin enters tab for budgets management 2. Admin enters budget value in dollars 3. Admin selects linked accounts from list 4. Admin clicks checkbox to mark budget creation as separate 5. Admin clicks submit button

Use case	Setup of machines termination notifications
Description	Admin wants to create specific period of time when users are notified about running machines with possibility to terminate them
Basic flow	1. Admin enters tab for machines termination management 2. Admin enters time period users should be notified 3. Admin clicks submit button

Use case	Admin receives notification about money request
Basic flow	1. Admin receives notification via mean of communication 2. Admin reads request description and makes decision 3. Admin decides to reject request and do not react
Alternative flow	Happens just after point 2. 1. Admin decides to grant more money 2. Admin enters new budget limit 3. Admin clicks submit button

4.1.2. Actor – Employee

Use case	User receives notification about budget
Precondition	There was created budget with linked account linked to the user. Budget is surpassed or if forecasted to be.
Basic flow	1. User receives notification via mean of communication 2. User reads notification description and makes decision what to do next 3. User do not need more money and do not react to notification
Alternative flow	Happens just after point 2. 1. User needs more money and clicks button for money request 2. User gets redirected to webpage for money request 3. user describes reason for more money 4. User clicks submit button

Use case	User receives notification about machines termination
Precondition	There was created time rule for machines termination notifications. User has working machines on at least one of linked accounts.
Basic flow	1. User receives notification via mean of communication 2. User reads notification description and makes decision what to do next 3. User do not want to terminate any machines and do not react to notification
Alternative flow	Happens after point 2. 1. User decides to terminate some machines 2. User clicks terminate buttons corresponding to specific machines

4.2. Overall architecture

Our solution is based on an architecture that is visualized on Figure 4.1

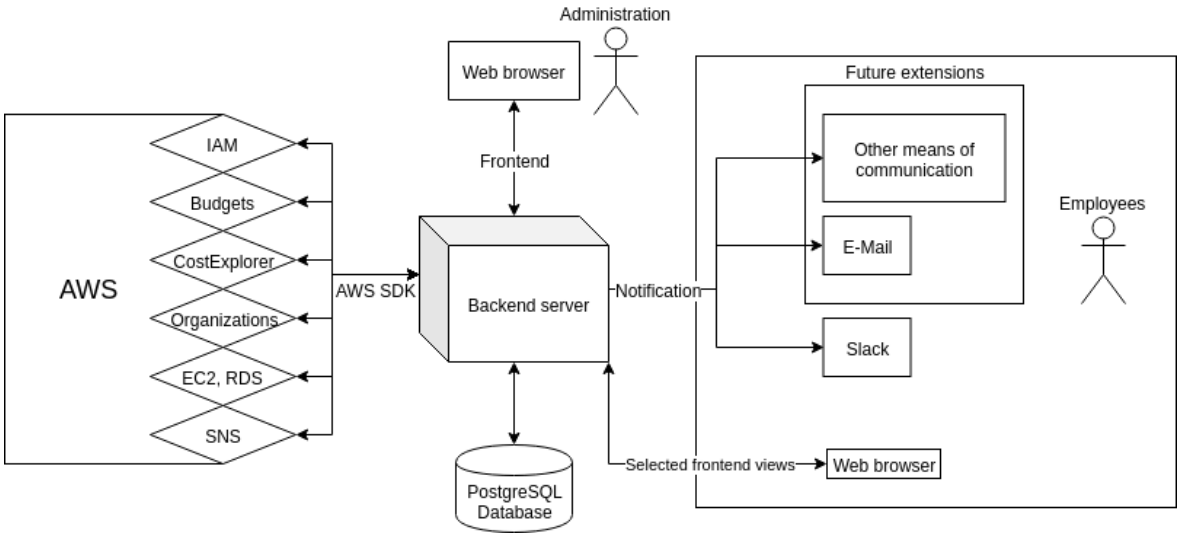

Figure 4.1: Draft diagram of the tool architecture

The architecture of our tool can be described as a traditional client-server architecture. Actors make requests over the HTTP protocol to the system with a web browser using the frontend client application served by the Omigost server or, alternatively, from any kind of a storage service like, for example, Amazon's S3. Those requests are then handled by a backend application running on a central server machine. During the handling process our application uses both a local instance of a PostgreSQL relational database and a connection with selected Amazon Web Services via a SDK library. Additionally, end users are notified about budget or instance events related to their cloud activity through Slack.

4.2.1. Role of AWS cloud services

Connecting with Amazon's cloud services not only allows the application to access machines and spending data that is crucial to be able to fulfill the business requirements, but many of them also solve many of the problems we would otherwise have to solve ourselves – resulting in time saved and a codebase that is more concise and easier to maintain.

Services that are used in the project and their respective roles in it are as follows:

- Identity and Access Management (IAM) – provision of the main Amazon account ID that is used in other services,
- Organizations – insight into structure of the Organization used by the company, including listing accounts,
- Budgets – service keeping eye on spendings that triggers a SNS notification whenever a budget limit is or is going to be exceeded,
- SNS – alert notifications for backend that let it know about spendings events,
- Cost Explorer – data for graph visualizing spendings in frontend,
- EC2/RDS – information about state of instances running on the account.

TODO at the end of the project – check if any other "sublibraries" were used and/or if any other use cases emerged

4.2.2. Backend

Backend is the main agent of our application's functionalities. It is responsible for fetching or receiving data from either of other connected resources, parsing it and taking appropriate actions. Some of its more important tasks include:

- receiving SNS budget alert notifications and notifying appropriate users via Slack,
  - providing single use tokens and links for non-admin users to be able to request budget limit increases,
- fetching and providing raw data for the frontend so it can be displayed to the user,
- receiving configuration or data modification requests and adjusting the application environment components to fulfill them,
- checking machine state on preconfigured times and notifying users about possible instance optimizations.

4.2.3. Frontend

Frontend is a web application that allows the user to see and alter the state of the application using a graphical user interface (GUI). Its main roles are to:

- request and parse raw data from backend into visual representations of it,
- provide an interface for the user,
- translate the interface clicks to appropriate backend requests and display the results of these requests.

4.3. Technology stack

4.3.1. Frontend

TODO

4.3.2. Backend

Core of the backend is *Java 8*. We have selected this programming language because most of us were familiar with it. Also this language has huge community and lots of useful libraries and frameworks.

For creation of the whole backend service we have chosen Spring [**Spring**] framework. It is currently one of the most common choices. Spring allows to build web applications imposing usage of design patterns like Model-View-Controller and Dependency Injection. It helps to keep code easily testable and well organized. Additionally it provides various features out of the box, which speeds up development process.

Gradle [**Gradle**] is responsible for build and dependencies management. It is easy to use with various IDEs and has flexible configuration. The main competitor of Gradle is Maven so obviously we considered both. Gradle turned out to be better in terms of performance and usage convenience.

For data persistence we have chosen PostgreSQL database. Relational database enforces having strictly defined data model with validation. Performance is not an important aspect for us so there was no need for other types of databases. PostgreSQL is a great open source database with strong data integrity and fault-tolerance guaranties. Furthermore, AWS lets set up PostgreSQL instance with just few clicks with automatically configured parameters for optimal performance.

In our application we use the following libraries:

1. Spring Core, Spring Boot, Spring Data, Spring Web,
2. Project Lombok,
3. JUnit,
4. AWS SDKs.

4.3.3. Deployment

The whole tool is bundled up by Docker. Following a straightforward instruction everyone is able to create their own Docker image of the application. Moreover, all of the configuration is present in one file making it easy to adapt it your way.

A preferable method of deployment is to use AWS Elastic Beanstalk with Docker image. In such a way AWS will be responsible for almost everything including capacity provisioning, load balancing and auto-scaling.

4.4. Data models

Waiting for final notifications implementation

4.5. Views and design

Waiting for final UI

4.6. Communication integration

TODO

4.7. Service termination architecture

While developing the application, we kept a certain model of organization of resources in mind, which the enterprise user might have in the AWS cloud. A common way of keeping an organization in AWS cloud is via service called "AWS Organizations" [**AWSOrganizations**]. The service allows to allocate AWS accounts with the predefined roles and permissions for every member of organization. Such structure also isolates resources from each other and lowers the granular control that the root account has on user allocated resources.

For our application to be able to tweak and monitor every machine created by the organization members, every employee of the organization needs to create a certain predefined role which should have the same name across all accounts. The role should have access to all resources that the user wants to be monitored. This can be done by a simple script every time a new member joins to the organization.

Secondly the account, from which the application will be deployed, needs to create an AWS IAM user [**AWSIAM**], which will be used by the application to assume the roles created by other members of the organization. To enable the functionality, the administrator needs to give "AssumeRole" permission to the newly created IAM user.

We recommended to take the following steps to grant the above mentioned permission.

1. Create an IAM group called "AssumesRolesGroup".
2. Go to the permissions section of the group.
3. Create a custom group policy
4. Add the following "JSON" to the policy document.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": "*"
    }
  ]
}
```

5. Add the IAM user to the "AssumesRoleGroup" group.

After taking all those steps, you just need to copy the AWS keys of that IAM user account to the application configuration file and you are good to go.

Bibliography

[Apptio] 2007-2019 Apptio, Inc. *Apptio Fuels Digital Transformation.* Apptio toolkit overview. *https://www.apptio.com/products.* Accessed - 01 April 2019.

[Armbrust] 2009 M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz et al. *Above the Clouds: A Berkeley View of Cloud Computing.* Technical Report No. UCB/EECS-2009-28. *http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.*

[AWSCostManagement] 2019, Amazon Web Services, Inc. or its affiliates. *AWS Cost Management – Welcome Page.* Initial description of the Cost Explorer and AWS Budgets API usage. *https://docs.aws.amazon.com/aws-cost-management/latest/APIReference/Welcome.html.* Accessed - 01 April 2019.

[AWSDocs] 2019, Amazon Web Services, Inc. or its affiliates. *AWS documentation. https://docs.aws.amazon.com.* Accessed - 01 April 2019.

[AWSIAM] 2019, Amazon Web Services, Inc. or its affiliates. *AWS Identity and Access Management (IAM) documentation. https://docs.aws.amazon.com/iam/index.html.* Accessed - 01 April 2019.

[AWSOrganizations] 2019, Amazon Web Services, Inc. or its affiliates. *AWS organizations documentation. https://docs.aws.amazon.com/organizations/index.html.* Accessed - 01 April 2019.

[Cloudability] 2019 Cloudability Inc. *Cloudability - Platform Overview.* Cloudability frontpage, description of its capabilities. *https://www.cloudability.com/product/.* Accessed - 01 April 2019.

[CloudabilityAlerts] 2015 Leah Weitz. *Creating budget alerts by tag with Cloudability. https://blog.cloudability.com/creating-budget-alerts-by-tag-with-cloudability/.*

[CostExplorer] 2019 Amazon Web Services, Inc. or its affiliates. *AWS Cost Explorer.* Service Capabilities Description. *https://aws.amazon.com/aws-cost-management/aws-cost-explorer/.* Accessed - 01 April 2019.

[Docker] 2019 Docker Inc. *Docker.* Docker tool frontpage. *https://www.docker.com/.* Accessed - 01 April 2019.

[GitHub] 2019 GitHub, Inc. *GitHub.* GitHub tool frontpage. *https://github.com/.* Accessed - 01 April 2019.

[Laatikainen] 2013 G. Laatikainen, A. Ojala, O. Mazhelis. *Cloud Services Pricing Models.*

[Markus] 2011 M. Böhm, S. Leimeisier, C. Riedl, H. Krcmar. *Cloud Computing and Computing Evolution.* Technische Universität München (TUM), Germany.

[Project Lombok] 2009-2019 The Project Lombok Authors. *Project Lombok*. Project Lombok library frontpage. *https://projectlombok.org/*. Accessed - 01 April 2019.

[Slack] 2019 Slack Technologies. *Slack*. Slack tool frontpage. *https://slack.com/*. Accessed - 01 April 2019.

[SnowBlog] 2017 David Svee. *The True Cost of AWS*. Overview of cloud cost issues and ways to optimize them. *https://www.snowsoftware.com/es/blog/2018/06/16/true-cost-aws*. Accessed - 01 April 2019.

[SnowSaaS] 2018 Snow Software. *Snow for SaaS | Snow Software - The Cloud Challenge*. Description of Snow for SaaS targets. *https://www.snowsoftware.com/int/snow-saas*. Accessed - 01 April 2019.

[Spring] 2019 Pivotal Software, Inc. *Spring by Pivotal*. Spring framework frontpage. *https://spring.io/*. Accessed - 01 April 2019.

[Stax.io] *https://www.getapp.com/it-management-software/a/stax/*. *https://www.stax.io/features*.

[TravisCI] 2019 Travis CI, GmbH. *Travis CI*. Travis CI Continuous Integration service frontpage. *https://travis-ci.org/*. Accessed - 01 April 2019.