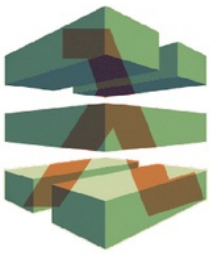
Thesis Janos Potecki

by Janos Potecki

FILE	388729_JANOS_POTECKI_THESIS_JANOS_POTECKI_2313963_1276399390.PDF		
TIME SUBMITTED	29-APR-2018 09:08PM (UTC+0100)	WORD COUNT	15780
SUBMISSION ID	87817572	CHARACTER COUNT	88160



AWS Continuous Delivery

An Open-Source CI/CD Tool for AWS Lambda Functions

`https://github.com/AwsLambdaContinuousDelivery`

Janós Potecki

BSc Computer Science
University College London

Submission Date:	**30th April 2018**
Supervisor:	**Dr. Graham Roberts**

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Dr. Graham Roberts for the continuous support during the three years of my BSc study, for his patience, motivation, and immense knowledge. His guidance helped me in all the time and I could not have imagined a better supervisor for my final year project.

In addition, I would like to thank Martí Serra Vivancos and Jaromir Latal for all those hours spent in pubs discussing why Haskell is the best programming language, spending an infinite amount of money on curries, ramen, and burgers during all those lunch breaks, for getting me through exams, and of course for reviewing this thesis.

I would also like to thank Cressida, Mark, Bruce, Reijo, Ian, Bryan, Vlastimil, and Gordon from the Amazon Lab126 Imaging Team in Cambridge for the opportunity of working in such great engineering environment even though I had just finished my first year at university. Likewise, I would like to thank the AWS Commerce Platform Team in Berlin, specifically Vittorio, Enrique, and Luigi for all the endless rejected merge requests, tough code reviews, and for showing me the power of a proper CI/CD pipeline system, which eventually was the inspiration for this thesis.

Last but not least, I want to thank my family for the continuous support throughout my life, specifically during the last six years of education after finishing my military career. Special thanks goes to my girlfriend Anne for endless support and for enduring three years of long-distance relationship while I studied in London.

Abstract

AWS Continuous Delivery `https://github.com/AwsLambdaContinuousDelivery` is an open-source Python library which helps programmers with developing and deploying continuous integration and delivery pipelines for AWS Lambda functions by creating CloudFormation templates. The library comes with pre–configured modules helping novices getting started quickly without the necessary extended knowledge of the AWS stack.

The need for such a library originated while developing a web app which was built mostly out of AWS Lambda functions. In order to quickly create and deploy pipelines for continuous delivery a tool was needed which could automate that burden. As a result this tool was developed in parallel to building the web app identifying and verifying the requirements.

The outcome is a working and usable version enabling programmers to create a continuous delivery pipeline with less than 70 lines of code, which includes automatically running unit and integration tests, building and deploying the AWS Lambda code, as well as notifications via e-mail on failures. Additionally the tool is built in a modular way, separating language dependent and independent parts making it easily extensible for further languages. Compared to alternative solutions available online this tool requires less knowledge about the AWS stack and doesn't incur any costs when no changes are committed.

Contents

List of Figures

1 Introduction

1.1 Problem Domain

Cloud computing is gaining more and more popularity in the industry and is becoming an important part in a developer's skill set. The growth predictions of the cloud computing business indicate that increasingly more companies are implementing their software via cloud services instead of running them on their own data centres. Undoubtedly one factor of the growing popularity of cloud services is the possibility to implement software on on-demand services, meaning that customers only pay when the software is actually being used. A popular on-demand service is AWS (Amazon Web Services) Lambda[1], an event-driven platform which runs code without users having to worry about provisioning, scaling, or shutting down instances to save resources and costs. But as nothing in life comes free these advantages come at a price: Complexity. Where a good old web application written in `php` could be built with two parts, the monolithic `php` code and a database, in the on-demand cloud world a simple web app easily consists of 10 or more services. This reveals another complexity: How to orchestrate continuous integration and deployment for so many services? What happens if users want to mirror the infrastructure for testing, as manually duplicating (or trebling, quadrupling, etc) quickly grows the number of services beyond a threshold which can't be handled in secure and reliable way manually?

1.2 Project Aims

There is currently no beginner friendly way to create and manage continuous integration and development (CI/CD) and automated testing for AWS Lambda (at a reasonable price). Therefore, this thesis will introduce an open-source tool which helps beginners and advanced programmers to build reliable, automatic, and secure infrastructure to deploy, maintain, and update AWS Lambda functions without having to spend an enormous amount of time learning how continuous integration and delivery is done in AWS.

The aim is to provide an easy but powerful open-source Python library, which enables programmers, specifically beginners, to create CI/CD pipeline templates including building and automated testing in less than 70 lines of code. Never-

[1] `https://aws.amazon.com/lambda`

theless, it should still be extendable, configurable, and also usable by advanced programmers. It should hide details and default configurations in separate, decoupled modules, making them easy to use for beginners but still modular enough for advanced users to add or replace them with their own modules and configurations. Likewise, it should not be a monolithic library for just one programming language, but be generic enough to add the support of other programming languages in the future by the open source community.

1.3 Project Goals

The goal of this project is to release a first working version, which supports building, testing, and deploying AWS Lambda functions with support for Python 3. Users should be able to launch pipelines which contain multiple stages, run unit tests before building the deployment package, and run custom integration tests on every stage. After having created a pipeline template and deploying the template to AWS a simple commit to the master branch should be sufficient to trigger the whole pipeline, i.e. building, deploying, and running all the tests which have been specified by the user. Furthermore, users should be able to keep tests, configuration, and source code in one repository. They should be able to extend the pipelines if necessary, without breaking the functions already deployed and it should be possible to reuse already generated templates across multiple AWS Lambda functions. Additionally, the project should be hosted in a publicly accessible place enabling and encouraging participation by other open-source developers.

1.4 Report Structure

The structure of the report will start with the definition of relevant software engineering techniques, followed by a brief introduction of relevant AWS services and alternatives to this project, an analysis of the requirements and the design of this project. The last part will point out some implementation details and design decisions before finishing with the evaluation, conclusion and a roadmap for the future work.

2 Definitions and Related Work

The purpose of this section is to give some short definitions of software engineering techniques and service descriptions in order to clarify why specific design decisions are made and implemented. It also will look into potential solutions and related work and will show, why those are not sufficient for the aims specified in 1.2.

2.1 Software Engineering Techniques

As mentioned in the first section this tool will help programmers to build more robust and reliable AWS Lambda functions by using pipelines and CI/CD. The following sections will briefly describe and clarify what these techniques mean in the context of this tool.

2.1.1 Continuous Delivery

Continuous Delivery (CD) is a software development approach in order to release new features reliably and separately in short cycles instead of the rare release of monolithic ones (Chen, 2015). Continuous delivery is usually done with the help of pipelines (defined in section 2.1.3, page 4) and is usually combined with continuous integration.

2.1.2 Continuous Integration

Continuous Integration (CI) means to build and test applications during its development on a rolling basis. As described by Humble and Farley the builds and tests of the whole system must happen whenever a commit to the masterline is done. Humble and Farley also states it is important that tests don't just contain unit tests (see section 2.1.4, page 5) but also integration tests (section 2.1.5, page 6) in a production–like environment to ensure that all parts of the system under test work together. This ensures that errors and bugs are detected early and reduces the likeliness of failures during the deployment process.(Humble and Farley, 2010)

2.1.3 Pipelines

Currently the best way to automate continuous integration and continuous deployment is to use so–called pipelines, which will be the product this tool is going to generate. A pipeline is made of multiple stages, which are usually connected linearly. If a stage succeeds, the next stage gets invoked, and if a stage fails, the whole pipeline fails. Respectively, after the last stage succeeds, the whole pipeline has succeeded. Figure 1 shows the traditional view of release candidates, but if the caption is changed and the figure is slightly modified the traditional view transforms into a modern, common structure of a continuous integration and delivery pipeline (as shown in figure 2).

A stage is a part of the pipeline, which consists of at least one action. If all actions succeeded in a stage, the stage succeeds and the next stage in the pipeline gets invoked.

Figure 1: Traditional View of a Release Candidate
(Humble and Farley, 2010, p.23)

Figure 2: Stages in a Pipeline

An Action is an operation like running tests, fetching code from a repository, or compiling some code. If an action returns without an error, it has succeeded. Figure 3 shows a stage with two actions: the first action will deploy an AWS CloudFormation template containing an AWS Lambda function followed by an

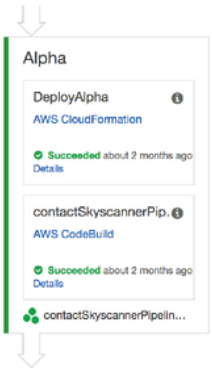


Figure 3: Alpha Stage With two Actions

action running integration tests on the deployed Lambda function. Just if both actions succeed, the stage itself succeeds.

2.1.4 Unit Tests

A Unit test is an automated function which calls other pieces of code on a predefined input and checks the results by comparing them to predefined assumptions (Osherove, 2009). Unit tests are important as they run "in isolation, preventing problem from surfacing during integration" (Spinellis, 2017). They are also very handy during the process of refactoring and can be used as a documentation for the written code (Spinellis, 2017). Unit tests should run fast and be invoked after any change in the code base, preferably locally or automatically by the continuous integration server (section 2.1.2, page 3). (Humble and Farley, 2010).

In the context of cloud computing and this tool, unit tests should still fulfil the previous mentioned definition and their main use case should be being run by the developer. Nevertheless, as Humble and Farley states, they should also run in the cloud due to the fact that developers are humans and simply can forget to run unit tests before committing their code. Likewise, the development environment might be different to the production one: A source of unexpected behaviour. Running them in the pipeline would guarantee that unit tests are always being run and wouldn't cause strange behaviour due to different environments.

2.1.5 Integration Tests

Integration Tests are similar to unit tests but instead of running in isolation they test more than one part of a software as a group (Osherove, 2009). In the specific case of Lambda, integration tests should for example test the correctness of external API URLs or whether the Lambda function can put and retrieve items from databases. Integration tests should be part of every pre-production stage. Their success or failure should decide whether to proceed or to cancel the deployment process.

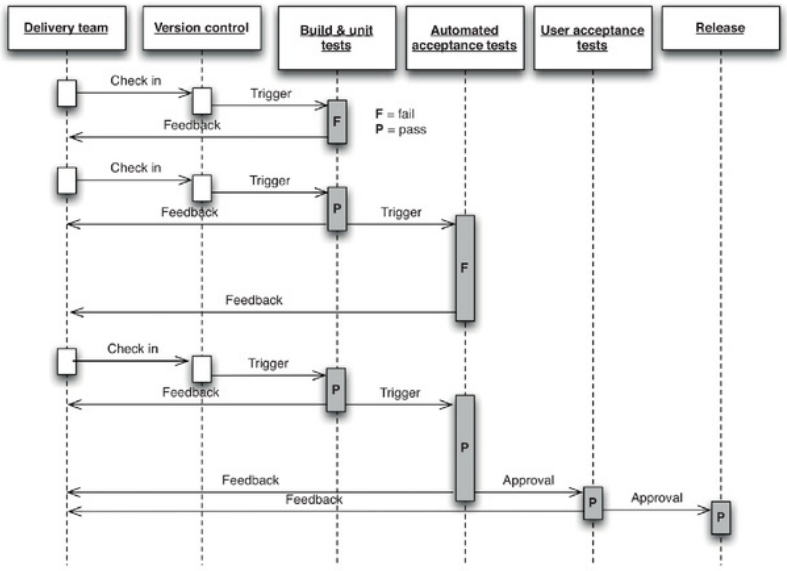


Figure 4: Changes Moving Through the Deployment Pipeline
(Humble and Farley, 2010, p.109)

Figure 4 puts the previously mentioned pieces together and shows how changes moves through a pipeline and what is need for a change to reach the last state.

2.2 Cloud Providers

Cloud computing is the delivery of services like databases, servers, and software via the internet to customers. Individuals and businesses can use these services

without building up their own infrastructure and data centres (Microsoft Azure, 2018). Corporations providing these services are called cloud providers. Figure 5 shows that currently the dominating infrastructure cloud providers are Microsoft and Amazon.

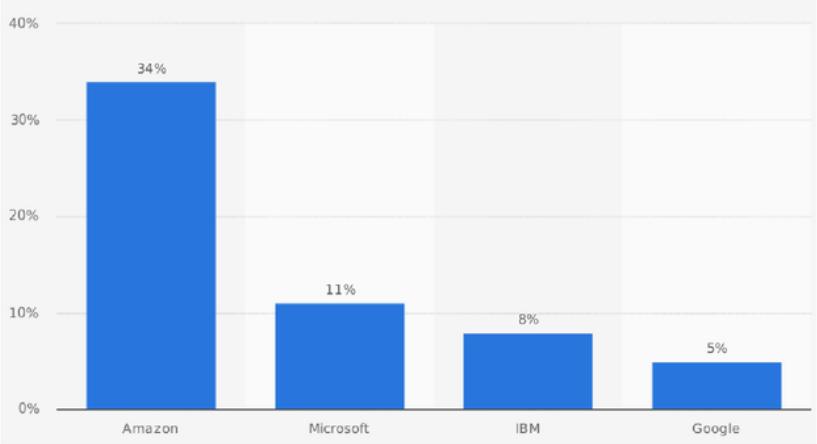


Figure 5: Market Shares of Cloud Providers (Cloud Infrastructure) by Turnover (Synergy Research Group, 2017)

2.3 Relevant AWS Services for This Project

This section will briefly introduce a small subset of relevant services for this project, as not all service–names are self-explaining.

AWS Amazon Elastic Compute Cloud (EC2) (`https://aws.amazon.com/ec2`) are nothing else than virtual machines hosted on Amazon servers. Users can rent such instances at any point and only pay for the running time (per second billing) and the processing power of an instance.

AWS Lambda (`https://aws.amazon.com/lambda`) is the service for which this tool helps building a continuous integration and delivery infrastructure. It is an event-driven, serverless computing platform which runs programmes on demand. In contrary to EC2 instances, once a Lambda is set-up, it simply waits to get invoked (e.g. by being called via a HTTPS request) without incurring

any costs. In the moment it receives an invocation signal, AWS handles all the configuration and provisioning, runs the defined code, and when it finishes, turns it off. The user just gets billed for the time the Lambda function ran, which is measured in milliseconds, where prices per second depend on the memory configuration. This means programmers can build Lambda functions for parts which are rarely invoked, and keep them "running" wasting no money on servers when there are no requests.

Deployment Package As the AWS Lambda environment just contains the standard Python libraries programmers need to provide external libraries themselves. In order to do so AWS introduced so-called deployment packages: `.zip` files which contain the function code and all necessary external libraries. These compressed files must be provided during setting up the Lambda functions along with the path and function (called handler) of the python function which will be called first during execution. This function must have the signature

```
def handler(event, context)
```

where `event` is a dictionary contain the arguments provided by the AWS Lambda function caller and `context` being a object from which developers can get run–time information about the Lambda function (e.g. memory limit of the function). For more information about the deployment package visit `https://docs.aws.amazon.com/lambda/latest/dg/lambda-python-how-to-create-deployment-package.html`.

AWS CodePipeline (`https://aws.amazon.com/codepipeline`) is the product, which will be configured and deployed after using this tool. AWS CodePipeline is a fully configurable tool, which helps to automate and visualise the deployment process. It can interact with many services from AWS, but also be used with some non AWS applications. In case of failures of a pipeline the failing stage will get a red colour and will provide links to the failing instance where users can inspect the log files in order to identify what has gone wrong.

AWS CloudFormation (`https://aws.amazon.com/cloudformation`) manages AWS cloud infrastructure as code, which is saved in a yaml or json format, and are usually called templates. Users can then use these templates to deploy, manage, and delete infrastructure on AWS without having to do it manually on the AWS website.

AWS CodeBuild (`https://aws.amazon.com/codebuild`) CodeBuild is simply a service which provisions, runs, and suspends an docker image on EC2 instances within a CodePipline. After choosing an operating system, programmers have to provide a `yaml` file with bash commands which get executed by CodeBuild after booting. The tool introduced in this thesis will make heavy use of CodeBuild to generate various interim files (called artifacts) and the AWS Lambda deployment package containing all the necessary libraries to execute the Lambda function code.

AWS CodeDeploy (`https://aws.amazon.com/codedeploy`) is like CodeBuild a tool which is used with AWS CodePipeline. Whereas CodeBuild is used in order to create CloudFormation templates, CodeDeploy is being used to actually deploy these templates from the CodePipeline.

AWS Amazon Simple Notification Service (SNS) (`https://aws.amazon.com/sns`) is a service enabling micro-services to send notifications (publish) with specific topics in order to invoke other applications, which have subscribed to these topics. This enables services to interact with each other and react to specific events. A simplified use-case: An EC2 instance is monitoring its inflow of traffic and as soon as it goes beyond a certain threshold, it publishes a notification of reaching the threshold. Another service subscribed to this topic gets notified and would take pre–configured actions, e.g. spinning up additional EC2 instances. SNS is heavily used by all services internally, but the only exposure the user will get to SNS in this tool is when adding e-mail notifications for failures in the pipeline in section 4.1.9, page 24.

AWS Simple Storage Service (S3) (`https://aws.amazon.com/s3`) is, as the name suggests, a service offering storage space. S3 organises space into so called Buckets containing objects files. S3 is the default storage solution in AWS and is being used by this tool in order to store intermediary files which are produced by the deployment pipeline.

AWS Identity and Access Management (IAM) (`https://aws.amazon.com/iam`) is being used to restrict access by services running on AWS in order to prevent specific services do malicious stuff. Configuring and managing these policies is not straightforward but unavoidable if building more complex structures on AWS and can pose a major entry barrier for beginners. In order to reduce this barrier and ensure that even beginners can use pipelines for continuous integration and delivery this tool can handle the access management. Nevertheless, it allows advanced users to manage these policies themselves if desired.

2.4 Related Work

The following section will explore some available tools which are in the same problem domain and show how they differ from the tool introduced in this paper.

2.4.1 Jenkins

Jenkins (`https://jenkins.io/`) is a widely used automation software suite, which also offers a pipeline feature[2] in order to organise CI/CD. While being a very versatile and powerful tool, Jenkins is not suitable for the use-case with Lambda functions, as it requires a server running all the time (e.g. on an AWS EC2 instance) which is waiting for a commit triggering the pipeline. This contradicts the on-demand nature of Lambda functions and would incur unnecessary costs, especially, if a Lambda function is not updated frequently (e.g. there no change for a whole month, but the Jenkins server is running all the time, incurring costs), being a major drawback compared to the tool introduced here.

[2] `https://wiki.jenkins.io/display/JENKINS/Pipeline+Plugin`

2.4.2 TravisCI

TravisCI (`https://travis-ci.org/`) comes close to the use case as the library introduced in this thesis. It is a service which only runs if triggered (e.g. by a new commit) and executes a list of pre–defined commands, so in theory there is the possibility to implement a full pipeline with bash scripts. This exposes the first problem: Users need to know how to write these scripts, which might be tricky if multiple stages are the goal. Another downside is if users want to deploy to AWS they will have to provide their credentials (being stored within TravisCI) which is always risky if not handled appropriately.

2.4.3 Drone

Drone (`https://github.com/drone`) is an open source alternative to TravisCI supporting plugins but requiring, like Jenkins, to rent out permanent server capacities. Thus it suffers from the same disadvantages as Jenkins which are high costs if not being used.

2.4.4 LambCI

LambCI (`https://github.com/lambci/lambci`) is a tool which utilises Lambda functions for builds. It also gets triggered by `git` commits and starts the building process, which users can define with simple commands. Although the name indicates that it is made for AWS Lambda functions, it is actually simply a build tool, which runs on AWS Lambda and claims to be cheaper than traditional build platforms like TravisCI (see 2.4.2) and AWS CodeBuild. So it could be rather seen as an substitute for AWS CodeBuild to reduce building costs.

2.4.5 Serverless

Serverless (`https://serverless.com`) is a framework which has the goal to achieve a universal development experience among different cloud providers. AWS was among the first supported and is quite extensive. It gives developers an easy way to deploy and update AWS Lambda functions, but unfortunately lacks a way of an automated method of running pipelines and deploying after each commit to mainline. One possibility would be to use it as a build and deployment tool, but the main problem lies in the fact that it needs access to credentials of the AWS account, which poses the risk of exposing very sensitive

data if not done correctly. Furthermore, it would need to get embedded in a pipeline for CI/CD being not really beginner friendly.

Section Summary

This section covered the basic definitions and AWS services which will be used in order to analyse the requirements and build the tool. Furthermore, this section showed why the current available tools are not suited for the intended use case justifying the need for the development of the tool introduced in this thesis.

3 Requirement Analysis

Currently if users want to create a CI/CD pipeline for AWS Lambda they need to build it from scratch themselves. They need to learn about tools like AWS CodePipeline, CodeBuild, CodeDeploy, and not to forget how to configure IAM roles and policies so everything works together. The aim of this tool is to reduce this burden and provide a system which helps programmers getting started while still staying completely configurable. This section will present the identified requirements which led the development of the library presented in this thesis.

3.1 Initial Problem Statement

The need for a tool helping users with creating deployment– and testing–pipelines arose during the development of a web app whose backend was going to be handled mostly by AWS Lambda functions. Having experienced the advantages of complete deployment and testing automation in industry I wanted to create a similar pipeline for the new web app. It should be composed of at least two stages, where one is specifically for testing, whereas the other one is the one in production used by customers. Furthermore, it should run integration tests, before deploying to production, in order to ensure the correctness of code. Unfortunately, initial research showed that there is no tool which helps with creating such pipelines and the only way to do it is by creating pipelines directly in AWS, which requires deep knowledge of an significant subset of the AWS stack. This absence of such a tool and the knowledge that further projects will also require its functionalities led to the start of the development of this tool.
This tool was mainly created during the development of the web app and as a result nearly all of requirements and design decision didn't just originate while building the web app, but also were tested and adjusted during the development of the web app. The following sections go over the requirements established during that time.

3.2 Support of Stages

As described in section 2.1.3 a CI/CD should contain a pipeline which is made up of several stages. In order to give programmers enough flexibility the stages should be modular and users should have the chance to add their own implementations of stages. Nevertheless it should provide the pre-configured stages

to build a basic pipeline.

3.2.1 Unit Tests

Usually unit tests are run by the developers on their machines before changes are pushed to the repository. But due to the fact that a developer might forget running the unit tests, the tool should support running unit tests in a stage automatically, preferably before any deployment package is being created.

3.2.2 Integration Tests

The main purpose of this tool is to support different stages for an AWS Lambda function and as already mentioned in section 2.1.2 integration tests play a pivotal part. They should be easy to configure, but still be consistent and in all stages. They should run automatically, and if one test fails the whole stage should fail. The main purpose of integration tests is to see whether they correctly interact with other services or, as in the case of the web app, with external API endpoints.

3.2.3 Notifications on Failure

As a commit running through the pipeline can take some time from start to finish, users should have the possibility to get notified if any stage of the pipeline fails, so they don't have to continuously monitor the pipeline. For the first version the notifications should be sent via e-mail.

3.3 Security

IAM, Roles, Policies: Configuration of AWS services can be very confusion for beginners, especially the management of IAM roles and policies. The CI/CD tool should take this burden from beginners and supply them with preconfigured templates out of the box. Nevertheless, it should still give advanced users the possibility to use their own roles and policies.

No Credential Storage: The tool should aim to avoid storing any credentials of the AWS user online in order to not be exposed to any risk of credential leakage.

Security by Design: Furthermore, it should aim for an architecture and enforced structures/patterns in order to prevent users from applying unwanted configuration changes. This is important especially for external API keys, as accidentally changing those might cause failures in the production environment.

3.4 Costs

As this tool will create one pipeline per AWS Lambda function, the costs should be kept at a minimum, but shouldn't sacrifice necessary features in favour of costs, as a the costs of failing services can be much higher than the costs for the pipeline. Also, due to the nature of AWS Lambda functions, once they are deployed no further costs should incur unless the function's source code is changed making new builds necessary.

3.5 Extendability

A very important feature for the tool is extendability, as the list of supported languages by AWS Lambda is growing (e.g. the recent introduction of go-lang[3]). The CI/CD tool should be designed in way that every part can be extended at any time.

3.6 External Libraries

During the development of the web app the need arose to use external libraries. In order to use them and as already mentioned in 2.3, external libraries must be provided by the user along with the function code in a deployment package. As it is not desired to have to store those library code in the repository the tool should support an automated way of downloading and installing of external libraries. Afterwards, it should bundle the libraries with the source-code into a deployment package.

[3] `https://aws.amazon.com/blogs/compute/announcing-go-support-for-aws-lambda/`

Section Summary

In this section the requirements which surfaced during the development of the web app have been set for the tool: It must support different, independent stages, should run unit and integration tests automatically, should automatically notify users if something fails, should take care of security while still being expandable and modular. Additionally, it should support the automatic building of deployment packages.

4 Implementation

This sections is split into two parts. The first half will take the users perspective and show how the tool works and give some details about the functions being used to to create, configure, and deploy a pipeline. After getting an idea how the tool works the second half of this section will dive into the architecture, implementation, and design decisions made.

4.1 Implementation: The Users Perspective

This section is a small walk-through how to use the tool. Figure 6 shows the pipeline this walk-through is going to build.

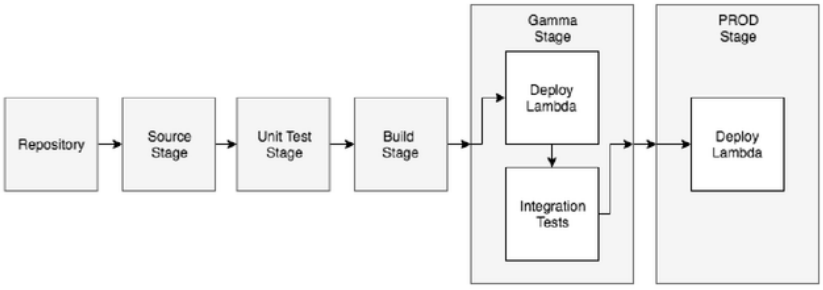


Figure 6: Pipeline to be Built

4.1.1 Creating the Pipeline

The pipeline is simply created as a Python script, which makes help of the `troposphere`[4] library (see 4.2.4 for more information) and the tool described in this thesis. This part will follow the stages of a pipeline (as shown in figure 6). The source-code for the complete example can be found in section A.1, page 40.

Template In general, all functions and stages are built similarly to keep consistency like taking a troposphere `Template` object as first argument. The `Template` is the main object which holds the whole AWS CloudFormation code, being returned and eventually converted to `json` in order to deploy the whole infrastructure.

[4] `https://github.com/cloudtools/troposphere`

4.1.2 Bootstrapping

The first step in order to use this tool is to create a troposphere `Template` object and a list with the pre-production stage names. Additionally, in order to cross-reference artifacts among stages a couple `String` objects need to be defined. This walk-through will assume the following bootstrapping:

```
template = Template()
# AWS will substitute this with the stack name during deployment
stack_name = Sub("${AWS::StackName}")
source_code = "SourceCode"
deploy_pkg_artifact = "FunctionDeployPackage"
cf_artifact = "CfOutputTemplate"
pipeline_stages = [] # list holding all stages, order matters!
stages = ["Gamma"] # we just want a Gamma stage before PROD
```

Furthermore, we want to create a storage for the artifacts and a IAM role which allows the pipeline to run everything. As IAM role creation can be confusing for beginners, a function returning a pre–configured role with the necessary permissions is provided:

```
from awslambdacontinuousdelivery.tools.iam import createCodepipelineRole

s3 = template.add_resource(
        Bucket("ArtifactStoreS3Location"
              , AccessControl = "Private"
              )
        )

pipeline_role = template.add_resource(
      createCodepipelineRole("PipelineRole"))
```

4.1.3 Source

Function The source stage is the entry point for the pipeline. It watches a source for changes, and if a change is detected it will trigger the pipeline and start pulling the data from the source. The result will be an artifact which will be stored in the designated pipeline storage.

Usage Currently just AWS CodeCommit and GitHub are supported as sources and each can be implemented by calling the functions `getCodeCommit` or `getGitHub` respectively.

```
from awslambdacontinuousdelivery.source.codecommit import getCodeCommit
from awslambdacontinuousdelivery.source.codecommit import getGitHub

# function type signatures
# def getCodeCommit(t: Template, source_code: str) -> Stages
# def getGitHub(t: Template, source_code: str) -> Stages

source = getCodeCommit(template, source_code)
#add source stage to pipeline
pipeline_stages.append(source)
```

Details Both functions add `Parameters` for the repository and branch to the template which, upon deployment, must be provided by the user allowing to reuse the same template for multiple AWS Lambda functions. If adding GitHub as a source, the user will additionally need to provide a GitHub repository owner and the corresponding OAuth token[5] .

[5]more information about integrating GitHub into CodePipeline: `https://aws.amazon.com/blogs/devops/integrating-git-with-aws-codepipeline/`

4.1.4 Running Unit Tests

Function The second stage is running all the unit tests before building the deployment package. Contrary to the source stage, unit tests are language dependant and thus users have to import the correct package in order to run the unit tests correctly.

Usage To add unit tests the user needs to import `getUnittest` and call it with the template and a reference to the source code location, essentially the same argument which is passed to create the source stage (4.1.3):

```
from awslambdacontinuousdelivery.python.test.unittest import getUnittest

# function type signature
# def getUnittest(template: Template, source_code: str) -> Stages

unit_tests = getUnittest(template, source_code)
pipeline_stages.append(unit_tests)
```

As unit tests are optional (although highly recommended) and to give programmers more flexibility in how to structure a repository (apart from the structure described in 4.1.11) the path to the unit test folder must be added to the `config.yaml` with the tag `Unittests` and subfield `Folder`. Additionally, users can add the subfield `Files` containing a list of files which should be executed. If no list of files is given, all Python files in the folder will be run. Furthermore, each Python file must be executable without any command line arguments. All non-standard libraries needed by the unit test must be added to a `requirements.txt` being located in the folder along the unit tests. It should be mentioned here that a `requirements.txt` file is the normal way of defining external dependencies in Python.

4.1.5 Build Stage

Function The job of the build stage is two-fold:
To create the deployment package for the AWS Lambda function from the source code, and secondly to generate an AWS CloudFormation file for each stage being used to deploy the AWS Lambda function with the stage-specific configurations.

Usage Import the language specific package, e.g. for Python `awslambdacontinuousdelivery.python.build` and then add the build stage to `pipeline_stages` by calling the `buildStage` function with the variables created in 4.1.2:

```
from awslambdacontinuousdelivery.python.build import getBuild

# function type signature:
# def getBuild( template: Template
#             , source_code: str
#             , deploy_pkg_artifact: str
#             , cf_artifact: str
#             , stages: List[str]
#             ) -> Stages

build_stage = getBuild(
  template, source_code, deploy_pkg_artifact, cf_artifact, stages)
pipeline_stages.append(build_stage)
```

Important Notice The `buildStage` function will **always** add a `PROD` stage. So users must not add the production stage to `stages: List[str]`.

Details This stage makes heavy use of AWS CloudBuild to generate both, the deployment package containing all the libraries needed to run the AWS Lambda function and to generate a AWS CloudFormation template for each stage. In order to save computation time a lightweight Alpine Linux with pre-installed Python libraries[6] (total 61MB) is being used. This ensures a short provision time resulting in less costs.

4.1.6 Pre-PROD Stages

Function In principle all stages following the build stage are the same just differentiating in configuration and whether they run tests or not. Each stage will use the deployment package and CloudFormation template generated in the build stage (see 4.1.5) to deploy the AWS Lambda function. After the successful deployment, integration tests are being run using AWS CloudBuild

[6] `https://hub.docker.com/r/frolvlad/alpine-python3/`

with the Alpine image. If all tests succeed the pipeline will move on to the next stage.

Usage As the deploy stage is just using CloudFormation to deploy the generated deployment package it is language independent. Nevertheless, the integration tests are **not** language independent and thus must be handed over to the deployment function. The following example will show how add integration tests for Python:

```
from awslambdacontinuousdelivery.deploy import getDeploy
# we will pass the following function to the ‘getDeploy‘ function
from awslambdacontinuousdelivery.python.test import getTest

# function type signatures
# def getTest(t: Template, source_code: str, stage: str) -> Action
# def getDeploy( t: Template
#              , cf_artifact: str         # deployment template for this stage
#              , stage: str               # stage name
#              , deploy_pkg_artifact: str
#              , source_code: str = None
#              , getTest: Callable[[Template, str, str], Action] = None
#              ) -> Stages
for s in stages:
  pipeline_stages.append(
    getDeploy( template, cf_artifact, s.capitalize()
             , deploy_pkg_artifact, source_code, getTest)
  )
```

Details As shown in the type signature in 4.1.6 the function arguments for `source_code` and `getTest` are by default set to `None`. Just if **both** are `not None` the higher order `getDeploy` function will add integration tests to the corresponding stage. One interesting detail here is that even though the `getTest` function is language dependant, it doesn't have to be the same language as the AWS Lambda function. This means, integration tests could be written also in languages like Java or Haskell (although currently just Python is supported). This is achieved by the fact that integration tests are running on an own AWS

CodeBuild image.
In order to run and prepare the integration tests, the AWS CodeBuild will use a Python script called `testRunner.py` which will invoke and run the files in the integration tests folder. In order to run the integration tests locally users can simply download the file from the repository[7] and run it with the necessary command line arguments:
`python3 testRunner.py --path folder/ --stack Example --stage Gamma`
The `--path` is the path to the folder holding the whole source-code for the AWS Lambda Functions. This is necessary, as the Python script will read the configuration file in order to extract the location of the integration tests. The `--stack` is the stackname, which was used to deploy the cloudformation template to ensure the integration tests are being run on the correct Lambda function.

4.1.7 PROD Stage

Function From a pipeline perspective the Production stage is simply the same a pre–PROD stage described in the previous section, but without running any tests. As a result the workflow is the same as described in 4.1.6 but without passing any tests to the `getDeploy` function.

Usage As mentioned, the same as in 4.1.6 is done just without adding any tests

```
PROD = getDeploy(template, cf_artifact, "PROD", deploy_pkg_artifact)
pipeline_stages.append(PROD)
```

[7] `https://github.com/AwsLambdaContinuousDelivery/pyAwsLambdaContinuousDeliveryIntegrationTests`

4.1.8 Putting Everything Together

The last step to create the pipeline is to create a troposphere `Pipeline` object and add the stages. This is very straightforward, as the following codes shows:

```
artifact_storage = ArtifactStore( Type = "S3", Location = Ref(s3))
pipeline = Pipeline( "FunctionsPipeline"
                   , Name = Sub("${AWS::StackName}-Pipeline")
                   , RoleArn = GetAtt(pipeline_role, "Arn")
                   , Stages = pipeline_stages
                   , ArtifactStore = artifact_storage
                   )

template.add_resource(pipeline)
```

4.1.9 Adding Notifications

Function After having built the pipeline with all the necessary stages the last important piece is adding an automatic notification system which will notify users. In the current state this tool contains templates for e-mail notifications on failures. If a stage fails, it sends a notification on a topic via e-mail.

Usage In order to add e-mail notifications the programmer must import `getEmailTopic` and `addFailureNotifications` to create an e-mail topic and add the notifications to the pipeline:

```
from awslambdacontinuousdelivery.notifications.sns import getEmailTopic
from awslambdacontinuousdelivery.notifications import addFailureNotifications

# type signatures:
# def getEmailTopic(topicName: str, emailAddr: str) -> Topic
# def addFailureNotifications( t: Template
#                            , pipeName: str
#                            , topic: Topic
#                            , createTopicPolicy = True
#                            ) -> Rule
topic = getEmailTopic("FailureNotification", "andy.jassy@amazon.com")
notification_role = addFailureNotifications(template, Ref(pipeline), topic)
```

Details In the `addFailureNotification` function the argument `createTopicPolicy: bool` is by default set to `True`. In this case it simply means that the necessary permissions for the pipeline to publish to the SNS topic are being automatically added to the template. Thus it should be just disabled if a user is adding these permissions manually.

4.1.10 Deployment of the Pipeline

Create JSON At this point building the Python script to create the JSON template is nearly done. All left to do is to add a print statement

```
print(template.to_json())
```

to the end of the script, followed by running it and saving the output to a file via

```
python3 pipeline.py > stack.json
```

As shown in A.2 the example (for complete source–code of `pipeline.py` see section A.1, page 40) has outputted more than 900 lines of `json`. This demonstrates the complexity behind building such a pipeline without an tool.

Deploying on AWS The last step is the deployment of the pipeline. Users can choose whether to use the AWS Command Line Interface[8] (CLI) to deploy it from the terminal or doing it via the AWS web interface called AWS Console[9]. The command to deploy it from the terminal is

```
bash-3.2$ aws cloudformation create-stack
--stack-name exampleAWSFunction
--template-body file://$(pwd)/stack.json
--capabilities CAPABILITY_NAMED_IAM
--parameters
ParameterKey=CodeCommitRepo,ParameterValue=exampleRepo
ParameterKey=Branch,ParameterValue=master
```

[8] `https://aws.amazon.com/cli/`
[9] `https://console.aws.amazon.com`

Figure 7: Failed Unit Test Stage

Updating the Stack Adding or removing stages from a pipeline is very easy: Users just have to edit the pipeline Python file accordingly and output the CloudFormation template. Instead of creating a new stack on AWS, they choose "Update Stack" in CloudFormation and upload the new `.json` file. AWS will then update the new pipeline infrastructure. AWS will automatically revert to the last working version in case anything goes wrong during the update process.

Pipeline Failures As already mentioned in section 2.3 failing states in the pipeline will be marked red in the AWS CodePipeline web-interface (as shown in figure 7). Additionally, in order to investigate the failing action it will contain the word "Failed" followed by the link to services which failed. The user can then use the logs of the service to investigate what went wrong.

Re-usability Once the template file is being created it isn't restricted to one Lambda function but can be used every time if a new Lambda function must be created (simply use a new repository during the setup process). This allows to deploy AWS Lambda functions in a project coherently, as all functions will have the same pipeline structure.

4.1.11 Required Repository Structure

The following sections will show the necessary bootstrapping and conventions to deploy a function via the pipeline.
The first is the folder structure, which is enforced and must look as following:

```
RootFolder
├── config
│   ├── Gamma
│   │   ├── env.py
│   │   └── iam.py
│   ├── PROD
│   │   ├── env.py
│   │   └── iam.py
│   └── config.yaml
├── src
│   └── function.py
├── tests
│   ├── unittests
│   │   ├── utest1.py
│   │   └── utest2.py
│   └── integrationtests
│       └── itests.py
├── requirements.txt
├── .git
└── .gitignore
```

RootFolder is the root and contains everything needed to deploy the AWS Lambda function.

config/ The `config` folder contains the configurations for every stage, which are kept in subfolders which must at least contain the stages defined in the list `stages` (defined in 4.1.2) and the folder `PROD`.

config/stagename/env.py This file contains the environment variables, which will be "injected" into the Lambda function. Each stage must have a separate file in its folder, which must contain at least the following function:

```
def get_env(stage: str) -> dict
```

During the build stage (4.1.5) a Python script will open the stage specific `env.py` file, call `get_env` with the stage name, and add the returned environment variables to the Lambda function template.

config/stagename/iam.py This file is responsible for granting permissions to the Lambda function (e.g. to read data from a database) in the specific stage. It is called in the same way as the `env.py` and must at least contain the following function:

```
from troposphere.iam import Role

def get_iam(stage: str) -> Role
```

Notice The reason for having separate `env.py` and `iam.py` for each stage is the separation of concerns, more specifically to raise awareness of reviewers in code-reviews. They will be more conscious when checking changes directly impacting configurations in production than code in the pre-production stages. Additionally, having all configurations for all stages in one file increases the chance of introducing unwanted changes. Separating configurations for each stage increases the overall security during the development process and thus complying with one of the goals specified in section 3.3, page 14.

config/config.yaml – Configuration of AWS Lambda In order to configure each AWS Lambda function users have to add a `config.yaml` file to the repository with the following mandatory configurations

```
Name: String
Handler: String
```

and those optional configurations:

```
MemorySize: Integer (between 128 and 1536, and a multiple of 64)
Timeout: Integer (default: 3sec)
Unittests: String (Folder containing unit tests)
Integrationtests (Folder containing integration tests)
```

The configurations `MemorySize` and `Timeout` are directly passed to AWS during the deployment. The `Handler` field specifies which file and function should be the entry point for the AWS Lambda function (as mentioned in 2.3).

tests/ This folder is not enforced and can be named totally differently. The shown structure is just an example. The only important thing is that the path

to the folders containing the unit and integration tests (if present) can be found in the `config.yaml`.

4.2 Design, Architecture, and Implementation

4.2.1 Packages

In order to comply with 3.5 the whole project was split into several packages, which are inspired by Java conventions. Each package is responsible to do just one small part of the pipeline and is either language independent or language dependent.

4.2.2 Conventions Used

Repository Naming Convention In order to allow easy distinction between packages which are tailored towards a specific language the following naming convention was introduced for language dependent repositories:

`<fileextentions>AwsContinuousDelievery<packageName>`
`e.g. pyAwsLambdaContinuousDeliveryIntegrationTests`

Language independent packages simply drop the file extension at the beginning.

Code Naming Convention All packages are actually separate Python packages, which can be installed via `pip` using the command `pip install .` inside the package folder. Users can then import these packages into their project with the following naming convention:

`awslambdacontinuousdelivery.<language>.<category>.<name>`
`e.g. awslambdacontinuousdelivery.python.build`

The example will import all functions, which will be needed in order to create a Python deployment package in the build stage (as shown in 4.1.5). For language independent packages a similar rule as for 4.2.2 applies: Simply drop the language part.

External Libraries Handling For the Python language, dependencies are often stored in a `requirements.txt` file. If programmers want to use external packages, they will need to add them to the projects `requirements.txt` file.

Additionally, unit and integration tests have their own `requirements.txt` files. Having separated files for tests and the deployment packages reduces the size for each provision and thus reduces costs in building, deploying, and running the tests and function.

4.2.3 Architecture

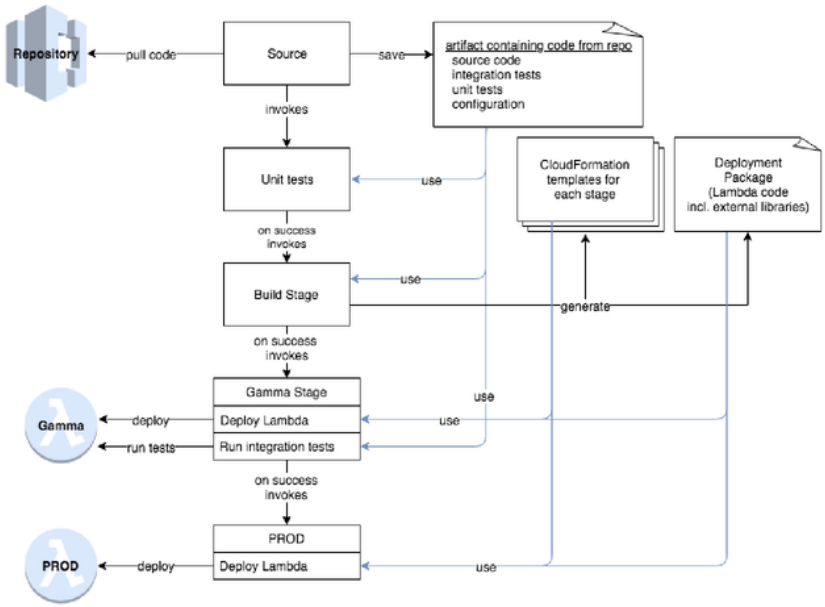


Figure 8: Diagram of the Deployment Process

The challenges of the complete automation of the build and deployment process are the many moving parts, which depend on the users input. Figure 8 shows which artifacts each stage is using and generating. When the source stage detects a new commit to the designated branch, it pulls the content from this branch in an artifact and invokes (if present) the stage running the unit tests. As seen in 4.1.11 the artifact will contain all the unit tests in a sub-folder of the root whose configuration file will contain the path to this folder. As a result the pipeline will be able to spin up a CodeBuild instance, which reads the configuration file and source folder, locates, and then executes all defined unit tests. If no error occurs in this process, the stage will succeed and the next

stage – the build stage – will be invoked.

Configuration Files or Enforced Structure The challenging part is that the pipeline must be generic enough to handle any amount of unit and integration tests to enable programmers to add additional tests at any time without touching the source code of the pipeline. In order to achieve this flexibility there are three options:

1. Enforce a folder structure with a dedicated folder for tests whose content is always being run
2. Use a configuration file to point to the specific tests to be run
3. A combination of both

The first approach has the advantage that it is easier for beginners as it removes the need of a configuration file. Although this tool is indented for novices and a first version used this approach, it turned out it is by far too limiting, especially with respect to the extendability mentioned in 3.5. Similarly, just having to define everything in a configuration file is also not optimal, as it complicates the overall implementation of the tool. As a result, this tool uses the third option and enforces a folder structure for the parts which are mandatory and uses a configuration file for optional ones (e.g. unit tests).

Functions for Configurations In order to give developers more flexibility the tool is using functions for configurations. For example, if users want to set environment variables for each stage the tool requires a Python file with a function that will be called when building the Cloudformation template. This has the advantage over .yaml or .json files, that programmers can create the variables dynamically. For example, if they are keeping these data remotely, they can make an API call during build time.

The Build stage is made of a total of two actions. First, using the source-code artifact, it builds the deployment package by installing all the necessary external libraries defined in the `requirements.txt` and deletes everything not in the `src/` folder before storing everything as a `.zip` file. The second step also uses the source-code artifact, this time reading the `config.yaml` file in order to use this information and the location of the deployment package `.zip` to generate a

CloudFormation template for each stage. The building of the templates is done by a Python script, whose source-code can be found in section A.4, page 64. After saving the CloudFormation templates in an artifact the first deployment stage is invoked

Pre-production stages containing integration tests are like the build stage made of two actions: The first one uses the deployment package and the corresponding CloudFormation template to deploy the Lambda function, followed by the second running the integration tests. In order to run the integration tests, it is using the source-code artifact, reading the `config.yaml` to locate the folder containing the tests. It then installs (if necessary) external libraries and runs all the tests.

The PROD stage is the last stage invoked in the stage. It just contains of the deployment of the CloudFormation file and the deployment package.

4.2.4 Technical Stack

troposphere (`https://github.com/cloudtools/troposphere`)
`troposphere` is a Python library which allows programmers to generate AWS Cloudformation templates. Resources, services, and configurations are being represented as Python classes and thus this library reduces the chance for mis–configurations as wrongly configured objects raise errors when being transformed into a CloudFormation template.

awacs (`https://github.com/cloudtools/awacs`)
Like troposphere, awacs is using Python classes to manage AWS Access Policies by providing classes which are checked during build time when being converted into a CloudFormation template. Troposphere and awacs combined are extremely important to minimise syntactic errors when generating CloudFormation templates.

Github Organisation (`https://github.com/AwsLambdaContinuousDelivery`)
In order to have all modules in one place, a GitHub organisation has been created. Besides having all repositories bundled, it can be also used as project management tool for handling feature requests and issue tracking.

Type Hints In order to make the code more self-explaining, the whole project is using type hints. Type hints is a Python3 feature adding type signatures to Python being checked during run–time[10]. This makes it for developers immediately clear, what to expect from functions and also makes errors clearer during run-time.

Section Summary

The first part of this section showed how to use the tool in order to create a CI/CD pipeline template for an AWS Lambda function with less than 70 lines (see A.1 for the whole source code). Due to the help of pre–configured modules provided by the tool beginners are able to quickly create a pipeline while still maintaining enough flexibility for advanced users. Additionally, it showed that the created pipeline template is independent from the AWS Lambda function allowing to deploy multiple Lambda functions without having to change the deployment pipeline source-code. The second and third parts shared some more details about the architecture and implementation to give some understanding about the challenges during development and how much burden is taken from the programmer when using this tool. The last part gave some additional information about the troposphere and awacs libraries in order to give an understanding why those are heavily used by this tool.

[10]for more information see `https://docs.python.org/3/library/typing.html`

5 Evaluation, Future Work, and Conclusion

5.1 Evaluation and Testing

This section will evaluate whether the requirements set in section 3 have been achieved and then explore how the tool is being tested.

5.1.1 Evaluation of Requirement Analysis

Support of Stages The main purpose of the tool is to create pipelines which are made out of stages. Users are given a pre–configured set of stages to get started, which are enough to quickly deploy and run continuous integration and delivery for AWS Lambda functions. Furthermore, as the tool just uses a simple `List[Stages]` for storing stages and users have all the freedom to add their own stages to any pipeline created with this tool. Additionally, even when a pipeline is already deployed, users can simply add or remove stages by updating the CloudFormation file and use the AWS interface to safely update the existing pipeline.

Automated Testing Another major feature requirement is the ability to run unit tests and integration tests automatically. This has also been achieved, with a pre–built stage for unit tests and by directly being able to insert integration tests (or any other tests) into deployment stages.

Notifications on Failure As explained in section 4.1.9 adding e-mail notifications is possible via a pre–configured SNS topic. As there are so many ways how to handle notifications, this pre-configuration is integrated rather as a guide on how to add notifications.

Security As mentioned in 3.3 the security aspect has three sub-aspects. The first one is to provide novices with pre–configured IAM policies could be partially fulfilled, as all roles which are necessary for creating and running the pipeline can be imported. As for the access permission for the Lambda itself there aren't any pre–configured policies available, as permission should always be just given for specific services, but not all of them (e.g. a Lambda function should be allowed to access a specific S3 bucket, but not all S3 buckets). Not providing these generic permission is a design decision following the third men-

tioned security requirement: Security by Design. Another design decision made to improve security was to separate the environmental configuration files for each stage in order to increase awareness, especially during code reviews (e.g. a code review will be done with much more care if it impacts the PROD stage than the Alpha stage). Additionally, as this tool creates a CloudFormation file which gets deployed to AWS and since the whole pipeline runs inside of AWS there is no risk of leaking any AWS credentials. The only danger for credential leakage is directly in the AWS Lambda function, if users would save those in the environment variables instead of using a secret credential storage services such as AWS Secrets Manager[11] (i.e. if credentials are hard-coded in the source code).

Costs Compared to the other services shown in 2.4 pipelines created with this tool just incur costs when actually changes are committed. Furthermore, due to AWS's free tier policy even the first 100 minutes of build time are free, meaning that by keeping the build steps efficient (e.g. by using small pre–configured docker images) costs are being saved. Nevertheless, there is still room for improvement left: Further investigations into optimising the size of the Docker images could reduce provision time and thus costs.

Extendability During the whole development process the tool was extended by multiple modules all the time, showing that it is expandable. Nevertheless, the final proof for extendability can just be done by implementing the *complete* support of another language than Python, which is on the roadmap for summer 2018 (see section 5.2).

External Libraries are supported by using Python's standard way of handling external libraries: a `requirements.txt` file[12]. In order to keep the deployment package small and not install unnecessary libraries, like libraries only used for unit and integration tests, the user can add additional libraries to a separate `requirements.txt` file for each test folder (which won't be used in the deployment package).

[11] `https://aws.amazon.com/secrets-manager/`
[12] `https://pip.pypa.io/en/stable/user_guide/#requirements-files`

5.1.2 Testing

The main testing method for the correctness of the output of this tool has been manual testing by saving the outputs in a `json` file and deploying it to AWS. Naturally it would be better to implement automated build tests but this is very difficult as the output of the program is a CloudFormation template printed in the `json` syntax, highly depending on the input of the user and how services are linked with each other. How complex and difficult it is to test the correctness of the CloudFormation files is shown by the testing done by AWS when deploying the template to AWS. AWS runs some syntax tests, but cannot predict if the linked services are correctly configured nor whether they'll interact with each other correctly. This difficulty of testing was also one of the reasons why type hints were used to have some form of validation and not experience unexpected type error. Likewise, the use of `troposphere` and `awacs` reduces the likelihood of having type errors compared to directly writing `json` templates.
One point which should be mentioned here is that this tool builds pipelines which test and deploy services, not the services itself. So there are two cases which could happen: Either the updated template won't deploy (which is not a problem) or one of the stages will fail due to a configuration error. Neither of these would kill or delete a Lambda function in production, as functions in production aren't replaced before the updated version is deployed (this is AWS internal functionality).

Manual Testing was done through the whole development process as this tool was built to be used. Besides the project where the idea originated from, it was also used for another project, which was built with multiple stages and services. As the second project used more services than the first one it didn't just reveal a few bugs, but also revealed some weaknesses which will need to be addressed (e.g. that the Alpine Docker image for the unit tests didn't contain `gcc` which is needed to install some Python packages needed for mocking AWS endpoints during unit tests). This approach showed, that the GitHub Organisation as centralised place to report bugs will always play a crucial part in the development and verification of the tool.

Automated Testing Nevertheless, automated testing will stay on the roadmap and it might be that some predefined verified working AWS infrastructure will

be built, which can verify the created template by deploying it and running specific tests on this service. This task is rather complex and would go beyond what is achievable during the time of development of this tool.

5.2 Future Work

Even though the tool is in a working and usable state there is still work to be done before making it an official release. A list of improvements and additions, which are planned, can be found on GitHub[13]. This sub-section will cover the most important ones with some more details compared to the roadmap on GitHub.

Other Language Implementations The next big milestone will be the support of another language, which will probably be Haskell. There are two reasons why Haskell: The first it is currently not trivial to create deployment packages to run Haskell on AWS Lambda and the second simply that there are already two developers who want to work on that over the summer.

Fetching from Multiple Repositories Another problem is that currently all source code must be in one repository. But developers might want to extract these into a separate repository because integration tests run in a production–like environment which can contain multiple services, where each service might own its own deployment pipeline. In such a case developers might want the same integration tests running in each pipeline in order to ensure the correctness of the system. In order to not have the test replicated in each repository there should be an option to link them to a pipeline.

SNS Notification for Release As mentioned in the former section 5.2 web applications can be made out of multiple services, which might be spread across multiple pipelines. A way to ensure that all services work properly with each other (after one has been updated) is by running all integration tests of all services in the same stage. This creates a race condition which can be solved by blocking all state transitions by default (e.g. from Alpha to Beta) and after all pipelines have finished their integration tests the transition to the next stage gets unblocked via a SNS notification (or not if one integration test fails).

[13] `https://github.com/orgs/AwsLambdaContinuousDelivery/projects/1`

Build a Script to Initialise Default Folder Structure To make it easier to use this tool there should be an easy way to initialise the default folder structure combined with the option to have a default pipeline already in place. This would mean developers could simply run one command, e.g.

```
bash-3.2$ awslambdacicd init --python
```

and the script would create all necessary folders and files in order to get started for the language Python. Furthermore, it could also contain commands to locally run unit and integration tests, e.g.

```
bash-3.2$ awslambdacicd test --unit
```

5.3 Conclusion

As mentioned in 5.1.2 the tool was used in different project than the one it originated from and proved, how much easier and faster it is to deploy AWS Lambda functions with it. In fact, it took less than 5 minutes before the first commit ran through the pipeline. That made clear how much time it saves developers which can be used for development instead of building continuous delivery pipelines. Conclusively it can be said that even though some work still remains to be done the major goals of the tool have been achieved. This tool helps improving developers productivity without incurring high costs contrary to similar solutions mentioned in section 2.4, page 10.

Bibliography

Chen, L. (2015). "Continuous Delivery: Huge Benefits, but Challenges Too". In: *IEEE Software* 32.2, pp. 50–54. ISSN: 0740-7459. DOI: 10.1109/MS.2015.27.

Humble, J. and D. Farley (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional. ISBN: 0321601912.

Microsoft Azure (2018). *What is cloud computing? A beginner's guide.*

Osherove, R. (2009). *The Art of Unit Testing: with Examples in .NET*. Manning Publications. ISBN: 1933988274.

Spinellis, D. (2017). "State-of-the-Art Software Testing". In: *IEEE Software* 34.5, pp. 4–6. ISSN: 0740-7459. DOI: 10.1109/MS.2017.3571564.

Synergy Research Group (2017). *Marktanteile der führenden Anbieter am Umsatz im Bereich Cloud Infrastructure (IaaS, PaaS, Hosted Private Cloud) weltweit im 2. Quartal 2017*. Accessed on 13. April 2018: https://de.statista.com/statistik/daten/studie/779775/umfrage/marktanteile-der-anbieter-am-umsatz-im-bereich-cloud-infrastructure-weltweit/.

A Example Source Code

A.1 pipeline.py

```
#!/usr/bin/env python3

from troposphere import Template, Sub, GetAtt, Ref
from troposphere.s3 import Bucket
from troposphere.codepipeline import Pipeline, ArtifactStore
from awslambdacontinuousdelivery.tools.iam import createCodepipelineRole
from awslambdacontinuousdelivery.source.codecommit import getCodeCommit
from awslambdacontinuousdelivery.python.test.unittest import getUnittest
from awslambdacontinuousdelivery.python.build import getBuild
from awslambdacontinuousdelivery.deploy import getDeploy
# we will pass the following function to the `getDeploy` function
from awslambdacontinuousdelivery.python.test import getTest
from awslambdacontinuousdelivery.notifications.sns import getEmailTopic
from awslambdacontinuousdelivery.notifications import addFailureNotifications

template = Template()
# AWS will substitute this with the stack name during deployment
stack_name = Sub("${AWS::StackName}")
source_code = "SourceCode"
deploy_pkg_artifact = "FunctionDeployPackage"
cf_artifact = "CfOutputTemplate"
pipeline_stages = [] # list holding all stages, order matters!
stages = ["Gamma"] # we just want a Gamma stage before PROD

s3 = template.add_resource(
  Bucket("ArtifactStoreS3Location"
        , AccessControl = "Private"
  )
)

pipeline_role = template.add_resource(
  createCodepipelineRole("PipelineRole"))
source = getCodeCommit(template, source_code)
pipeline_stages.append(source)

unit_tests = getUnittest(template, source_code)
pipeline_stages.append(unit_tests)

build_stage = getBuild(
  template, source_code, deploy_pkg_artifact, cf_artifact, stages)
pipeline_stages.append(build_stage)

for s in stages:
  pipeline_stages.append(
    getDeploy( template, cf_artifact, s.capitalize()
```

```
                , deploy_pkg_artifact, source_code, getTest)
  )

PROD = getDeploy(template, cf_artifact, "PROD", deploy_pkg_artifact)

pipeline_stages.append(PROD)

artifact_storage = ArtifactStore( Type = "S3", Location = Ref(s3))
pipeline = Pipeline( "FunctionsPipeline"
                    , Name = Sub("${AWS::StackName}-Pipeline")
                    , RoleArn = GetAtt(pipeline_role, "Arn")
                    , Stages = pipeline_stages
                    , ArtifactStore = artifact_storage
                    )

template.add_resource(pipeline)

topic = getEmailTopic("FailureNotification", "andy.jassy@amazon.com")
notification_role = addFailureNotifications(template, Ref(pipeline), topic)

print(template.to_json(indent=2))
```

A.2 Generated .json Template

```
{
  "Parameters": {
    "Branch": {
      "Description": "Branch triggering the deployment",
      "Type": "String"
    },
    "CodeCommitRepo": {
      "Description": "Name of the CodeCommit Repository",
      "Type": "String"
    }
  },
  "Resources": {
    "ArtifactStoreS3Location": {
      "Properties": {
        "AccessControl": "Private"
      },
      "Type": "AWS::S3::Bucket"
    },
    "CFDeployRole": {
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Statement": [
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": "cloudformation.amazonaws.com"
              }
            }
          ]
        },
        "Policies": [
          {
            "PolicyDocument": {
              "Statement": [
                {
                  "Action": [
                    "ec2:*",
                    "lambda:GetFunction",
                    "lambda:CreateFunction",
                    "lambda:GetFunctionConfiguration",
                    "lambda:DeleteFunction",
                    "lambda:UpdateFunctionCode",
                    "lambda:UpdateFunctionConfiguration",
                    "lambda:CreateAlias",
                    "lambda:DeleteAlias",
```

```
                    "s3:GetObject"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "*"
                  ]
                },
                {
                  "Action": [
                    "iam:DeleteRole",
                    "iam:DeleteRolePolicy",
                    "iam:GetRole",
                    "iam:PutRolePolicy",
                    "iam:CreateRole",
                    "iam:PassRole"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "*"
                  ]
                }
              ]
            },
            "PolicyName": "CloudFormationDeployPolicy"
          }
        ],
        "RoleName": {
          "Fn::Sub": "${AWS::StackName}-CFDeployRole"
        }
      },
      "Type": "AWS::IAM::Role"
    },
    "CodeBuildRole": {
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Statement": [
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": "codebuild.amazonaws.com"
              }
            }
          ]
        },
        "Policies": [
          {
            "PolicyDocument": {
```

```
            "Statement": [
              {
                "Action": [
                  "*"
                ],
                "Effect": "Allow",
                "Resource": [
                  "*"
                ]
              }
            ]
          },
          "PolicyName": {
            "Fn::Sub": "${AWS::StackName}-CodeBuildPolicy"
          }
        }
      ],
      "RoleName": {
        "Fn::Sub": "${AWS::StackName}-LambdaCodeBuildRole"
      }
    },
    "Type": "AWS::IAM::Role"
  },
  "DeployPkgBuilder": {
    "Properties": {
      "Artifacts": {
        "Type": "CODEPIPELINE"
      },
      "Environment": {
        "ComputeType": "BUILD_GENERAL1_SMALL",
        "Image": "frolvlad/alpine-python3",
        "PrivilegedMode": "false",
        "Type": "LINUX_CONTAINER"
      },
      "Name": {
        "Fn::Sub": "${AWS::StackName}-DeployPkgBuilder"
      },
      "ServiceRole": {
        "Ref": "CodeBuildRole"
      },
      "Source": {
        "BuildSpec": {
          "Fn::Join": [
            "\n",
            [
              "version: 0.2",
              "phases:",
              "  install:",
              "    commands:",
```

```
148                 "      - pip3 install --upgrade pip
                        setuptools",
149                 "  build:",
150                 "    commands:",
151                 "      - pip3 install -r requirements.txt -t
                        .",
152                 "      - ls -a",
153                 "      - mv -v src/* .",
154                 "      - rm -rf src",
155                 "      - rm -rf config",
156                 "      - rm -rf test",
157                 "      - ls -a",
158                 "artifacts:",
159                 "  type: zip",
160                 "  files:",
161                 "    - '**/*'"
162               ]
163             ]
164           },
165           "Type": "CODEPIPELINE"
166         }
167       },
168       "Type": "AWS::CodeBuild::Project"
169     },
170     "FailureNotificationRule": {
171       "Properties": {
172         "Description": {
173           "Fn::Sub": "Notification Rule for ${AWS::StackName
                }"
174         },
175         "EventPattern": {
176           "detail": {
177             "state": [
178               "FAILED"
179             ]
180           },
181           "detail-type": [
182             "CodePipeline Action Execution State Change"
183           ],
184           "source": [
185             "aws.codepipeline"
186           ]
187         },
188         "State": "ENABLED",
189         "Targets": [
190           {
191             "Arn": {
192               "Ref": "FailureNotificationSnsTopic"
193             },
194             "Id": {
```

```
            "Fn::Join": [
              "-",
              [
                {
                  "Ref": "FunctionsPipeline"
                },
                "FailureNotificationsTarget"
              ]
            ]
          },
          "InputTransformer": {
            "InputPathsMap": {
              "pipeline": "$.detail.pipeline"
            },
            "InputTemplate": "\"Pipeline <pipeline> failed
                .\""
          }
        }
      ]
    },
    "Type": "AWS::Events::Rule"
  },
  "FailureNotificationSnsTopic": {
    "Properties": {
      "DisplayName": {
        "Fn::Sub": "${AWS::StackName}
            FailureNotificationSnsTopic"
      },
      "Subscription": [
        {
          "Endpoint": "andy.jassy@amazon.com",
          "Protocol": "email"
        }
      ],
      "TopicName": {
        "Fn::Sub": "${AWS::StackName}
            FailureNotificationSnsTopic"
      }
    },
    "Type": "AWS::SNS::Topic"
  },
  "FunctionsPipeline": {
    "Properties": {
      "ArtifactStore": {
        "Location": {
          "Ref": "ArtifactStoreS3Location"
        },
        "Type": "S3"
      },
      "Name": {
```

```
            "Fn::Sub": "${AWS::StackName}-Pipeline"
          },
          "RoleArn": {
            "Fn::GetAtt": [
              "PipelineRole",
              "Arn"
            ]
          },
          "Stages": [
            {
              "Actions": [
                {
                  "ActionTypeId": {
                    "Category": "Source",
                    "Owner": "AWS",
                    "Provider": "CodeCommit",
                    "Version": "1"
                  },
                  "Configuration": {
                    "BranchName": {
                      "Ref": "Branch"
                    },
                    "RepositoryName": {
                      "Ref": "CodeCommitRepo"
                    }
                  },
                  "Name": {
                    "Fn::Sub": "${AWS::StackName}-LambdaSource
                       "
                  },
                  "OutputArtifacts": [
                    {
                      "Name": "SourceCode"
                    }
                  ],
                  "RunOrder": "1"
                }
              ],
              "Name": "Source"
            },
            {
              "Actions": [
                {
                  "ActionTypeId": {
                    "Category": "Build",
                    "Owner": "AWS",
                    "Provider": "CodeBuild",
                    "Version": "1"
                  },
                  "Configuration": {
```

```
                    "ProjectName": {
                      "Ref": "unittestBuilder"
                    }
                  },
                  "InputArtifacts": [
                    {
                      "Name": "SourceCode"
                    }
                  ],
                  "Name": {
                    "Fn::Sub": "${AWS::StackName}-
                       UnittestAction1"
                  },
                  "RunOrder": "1"
                }
              ],
              "Name": "Unittests"
            },
            {
              "Actions": [
                {
                  "ActionTypeId": {
                    "Category": "Build",
                    "Owner": "AWS",
                    "Provider": "CodeBuild",
                    "Version": "1"
                  },
                  "Configuration": {
                    "ProjectName": {
                      "Ref": "DeployPkgBuilder"
                    }
                  },
                  "InputArtifacts": [
                    {
                      "Name": "SourceCode"
                    }
                  ],
                  "Name": {
                    "Fn::Sub": "${AWS::StackName}-BuildAction1
                       "
                  },
                  "OutputArtifacts": [
                    {
                      "Name": "FunctionDeployPackage"
                    }
                  ],
                  "RunOrder": "1"
                },
                {
                  "ActionTypeId": {
```

```
                "Category": "Build",
                "Owner": "AWS",
                "Provider": "CodeBuild",
                "Version": "1"
              },
              "Configuration": {
                "ProjectName": {
                  "Ref": "cfBuilder"
                }
              },
              "InputArtifacts": [
                {
                  "Name": "SourceCode"
                }
              ],
              "Name": {
                "Fn::Sub": "${AWS::StackName}-BuildAction2
                    "
              },
              "OutputArtifacts": [
                {
                  "Name": "CfOutputTemplate"
                }
              ],
              "RunOrder": "2"
            }
          ],
          "Name": "Build"
        },
        {
          "Actions": [
            {
              "ActionTypeId": {
                "Category": "Deploy",
                "Owner": "AWS",
                "Provider": "CloudFormation",
                "Version": "1"
              },
              "Configuration": {
                "ActionMode": "CREATE_UPDATE",
                "Capabilities": "CAPABILITY_NAMED_IAM",
                "ParameterOverrides": "{\"S3Key\": {\"Fn::
                    GetArtifactAtt\": [\"
                    FunctionDeployPackage\", \"ObjectKey\"]
                    }, \"S3Storage\": {\"Fn::GetArtifactAtt
                    \": [\"FunctionDeployPackage\", \"
                    BucketName\"]}}",
                "RoleArn": {
                  "Fn::GetAtt": [
                   "CFDeployRole",
```

```
                    "Arn"
                  ]
                },
                "StackName": {
                  "Fn::Sub": "${AWS::StackName}
                     FunctionsGamma"
                },
                "TemplatePath": "CfOutputTemplate::
                    stackGamma.json"
              },
              "InputArtifacts": [
                {
                  "Name": "CfOutputTemplate"
                },
                {
                  "Name": "FunctionDeployPackage"
                }
              ],
              "Name": "DeployGamma",
              "RunOrder": "1"
            },
            {
              "ActionTypeId": {
                "Category": "Build",
                "Owner": "AWS",
                "Provider": "CodeBuild",
                "Version": "1"
              },
              "Configuration": {
                "ProjectName": {
                  "Ref": "TestBuildGamma"
                }
              },
              "InputArtifacts": [
                {
                  "Name": "SourceCode"
                }
              ],
              "Name": {
                "Fn::Sub": "${AWS::StackName}-
                    TestCfBuilderAction"
              },
              "RunOrder": "2"
            }
          ],
          "Name": "Gamma"
        },
        {
          "Actions": [
            {
```

```
                "ActionTypeId": {
                  "Category": "Deploy",
                  "Owner": "AWS",
                  "Provider": "CloudFormation",
                  "Version": "1"
                },
                "Configuration": {
                  "ActionMode": "CREATE_UPDATE",
                  "Capabilities": "CAPABILITY_NAMED_IAM",
                  "ParameterOverrides": "{\"S3Key\": {\"Fn::
                      GetArtifactAtt\": [\"
                      FunctionDeployPackage\", \"ObjectKey\"]
                      }, \"S3Storage\": {\"Fn::GetArtifactAtt
                      \": [\"FunctionDeployPackage\", \"
                      BucketName\"]}}",
                  "RoleArn": {
                    "Fn::GetAtt": [
                      "CFDeployRole",
                      "Arn"
                    ]
                  },
                  "StackName": {
                    "Fn::Sub": "${AWS::StackName}
                        FunctionsPROD"
                  },
                  "TemplatePath": "CfOutputTemplate::
                      stackPROD.json"
                },
                "InputArtifacts": [
                  {
                    "Name": "CfOutputTemplate"
                  },
                  {
                    "Name": "FunctionDeployPackage"
                  }
                ],
                "Name": "DeployPROD",
                "RunOrder": "1"
              }
            ],
            "Name": "PROD"
          }
        ]
      },
      "Type": "AWS::CodePipeline::Pipeline"
    },
    "PipelineRole": {
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Statement": [
```

```
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": "codepipeline.amazonaws.com"
              }
            }
          ]
        },
        "Policies": [
          {
            "PolicyDocument": {
              "Statement": [
                {
                  "Action": [
                    "s3:GetObject",
                    "s3:GetObjectVersion",
                    "s3:GetBucketVersioning"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "*"
                  ]
                },
                {
                  "Action": [
                    "s3:PutObject"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "arn:aws:s3:::codepipeline*",
                    "arn:aws:s3:::elasticbeanstalk*"
                  ]
                },
                {
                  "Action": [
                    "codecommit:CancelUploadArchive",
                    "codecommit:GetBranch",
                    "codecommit:GetCommit",
                    "codecommit:GetUploadArchiveStatus",
                    "codecommit:UploadArchive"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "*"
                  ]
                },
                {
```

```
523                     "Action": [
524                       "codedeploy:CreateDeployment",
525                       "codedeploy:GetApplicationRevision",
526                       "codedeploy:GetDeployment",
527                       "codedeploy:GetDeploymentConfig",
528                       "codedeploy:RegisterApplicationRevision"
529                     ],
530                     "Effect": "Allow",
531                     "Resource": [
532                       "*"
533                     ]
534                   },
535                   {
536                     "Action": [
537                       "elasticbeanstalk:*",
538                       "ec2:*",
539                       "elasticloadbalancing:*",
540                       "autoscaling:*",
541                       "cloudwatch:*",
542                       "s3:*",
543                       "sns:*",
544                       "cloudformation:*",
545                       "rds:*",
546                       "sqs:*",
547                       "ecs:*",
548                       "iam:PassRole"
549                     ],
550                     "Effect": "Allow",
551                     "Resource": [
552                       "*"
553                     ]
554                   },
555                   {
556                     "Action": [
557                       "lambda:InvokeFunction",
558                       "lambda:ListFunctions"
559                     ],
560                     "Effect": "Allow",
561                     "Resource": [
562                       "*"
563                     ]
564                   },
565                   {
566                     "Action": [
567                       "opsworks:CreateDeployment",
568                       "opsworks:DescribeApps",
569                       "opsworks:DescribeCommands",
570                       "opsworks:DescribeDeployments",
571                       "opsworks:DescribeInstances",
572                       "opsworks:DescribeStacks",
```

```
573                     "opsworks:UpdateApp",
574                     "opsworks:UpdateStack"
575                   ],
576                   "Effect": "Allow",
577                   "Resource": [
578                     "*"
579                   ]
580                 },
581                 {
582                   "Action": [
583                     "cloudformation:CreateStack",
584                     "cloudformation:DeleteStack",
585                     "cloudformation:DescribeStacks",
586                     "cloudformation:UpdateStack",
587                     "cloudformation:CreateChangeSet",
588                     "cloudformation:DeleteChangeSet",
589                     "cloudformation:DescribeChangeSet",
590                     "cloudformation:ExecuteChangeSet",
591                     "cloudformation:SetStackPolicy",
592                     "cloudformation:ValidateTemplate",
593                     "iam:PassRole"
594                   ],
595                   "Effect": "Allow",
596                   "Resource": [
597                     "*"
598                   ]
599                 },
600                 {
601                   "Action": [
602                     "codebuild:BatchGetBuilds",
603                     "codebuild:StartBuild"
604                   ],
605                   "Effect": "Allow",
606                   "Resource": [
607                     "*"
608                   ]
609                 }
610               ]
611             },
612             "PolicyName": {
613               "Fn::Sub": "oneClickCodePipeServicePolicy-${
                      AWS::StackName}"
614             }
615           }
616         ],
617         "RoleName": {
618           "Fn::Sub": "${AWS::StackName}PipelineRole"
619         }
620       },
621       "Type": "AWS::IAM::Role"
```

```
622       },
623       "TestBuildGamma": {
624         "Properties": {
625           "Artifacts": {
626             "Type": "CODEPIPELINE"
627           },
628           "Environment": {
629             "ComputeType": "BUILD_GENERAL1_SMALL",
630             "Image": "frolvlad/alpine-python3",
631             "PrivilegedMode": "false",
632             "Type": "LINUX_CONTAINER"
633           },
634           "Name": {
635             "Fn::Sub": "${AWS::StackName}-Gamma"
636           },
637           "ServiceRole": {
638             "Ref": "TestBuilderRoleGamma"
639           },
640           "Source": {
641             "BuildSpec": {
642               "Fn::Join": [
643                 "\n",
644                 [
645                   "version: 0.2",
646                   "\n",
647                   "phases:",
648                   "  install:",
649                   "    commands:",
650                   "      - apk add --no-cache openssl",
651                   "      - pip3 install boto3 pyyaml",
652                   "  build:",
653                   "    commands:",
654                   "      - wget https://raw.githubusercontent.
                          com/AwsLambdaContinuousDelivery/
                          AwsLambdaTesting/dev/executable/
                          testRunner.py",
655                   "\n",
656                   {
657                     "Fn::Join": [
658                       " ",
659                       [
660                         "      - python3 testRunner.py -p $(
                              pwd)/ --stage",
661                         "Gamma",
662                         "--stack",
663                         {
664                           "Fn::Sub": "${AWS::StackName}"
665                         }
666                       ]
667                     ]
```

```
                }
              ]
            ]
          },
          "Type": "CODEPIPELINE"
        }
      },
      "Type": "AWS::CodeBuild::Project"
    },
    "TestBuilderRoleGamma": {
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Statement": [
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": "codebuild.amazonaws.com"
              }
            }
          ]
        },
        "Policies": [
          {
            "PolicyDocument": {
              "Statement": [
                {
                  "Action": [
                    "*"
                  ],
                  "Effect": "Allow",
                  "Resource": [
                    "*"
                  ]
                }
              ]
            },
            "PolicyName": {
              "Fn::Sub": "${AWS::StackName}-
                  TestBuilderPolicy"
            }
          }
        ],
        "RoleName": {
          "Fn::Sub": "LambdaTestBuilderRole-${AWS::StackName
              }Gamma"
        }
      },
```

```
      "Type": "AWS::IAM::Role"
    },
    "TopicPolicy": {
      "Properties": {
        "PolicyDocument": {
          "Id": {
            "Fn::Sub": "TopicsPublicationPolicy${AWS::
                StackName}"
          },
          "Statement": [
            {
              "Action": [
                "sns:Publish"
              ],
              "Effect": "Allow",
              "Principal": {
                "AWS": "*"
              },
              "Resource": "*",
              "Sid": "Allow-SNS-SendMessage"
            }
          ],
          "Version": "2008-10-17"
        },
        "Topics": [
          {
            "Ref": "FailureNotificationSnsTopic"
          }
        ]
      },
      "Type": "AWS::SNS::TopicPolicy"
    },
    "UnittestRole": {
      "Properties": {
        "AssumeRolePolicyDocument": {
          "Statement": [
            {
              "Action": [
                "sts:AssumeRole"
              ],
              "Effect": "Allow",
              "Principal": {
                "Service": "codebuild.amazonaws.com"
              }
            }
          ]
        },
        "Policies": [
          {
            "PolicyDocument": {
```

```
765                 "Statement": [
766                   {
767                     "Action": [
768                       "*"
769                     ],
770                     "Effect": "Allow",
771                     "Resource": [
772                       "*"
773                     ]
774                   }
775                 ]
776               },
777               "PolicyName": {
778                 "Fn::Sub": "${AWS::StackName}-UnittestPolicy"
779               }
780             }
781           ],
782           "RoleName": {
783             "Fn::Sub": "${AWS::StackName}-UnittestRole"
784           }
785         },
786         "Type": "AWS::IAM::Role"
787       },
788       "cfBuilder": {
789         "Properties": {
790           "Artifacts": {
791             "Type": "CODEPIPELINE"
792           },
793           "Environment": {
794             "ComputeType": "BUILD_GENERAL1_SMALL",
795             "Image": "frolvlad/alpine-python3",
796             "PrivilegedMode": "false",
797             "Type": "LINUX_CONTAINER"
798           },
799           "Name": {
800             "Fn::Sub": "${AWS::StackName}-cfBuilder"
801           },
802           "ServiceRole": {
803             "Ref": "CodeBuildRole"
804           },
805           "Source": {
806             "BuildSpec": {
807               "Fn::Join": [
808                 "\n",
809                 [
810                   "version: 0.2",
811                   "\n",
812                   "phases:",
813                   "  install:",
814                   "    commands:",
```

```
"      - apk add --no-cache bash git openssl
    ",
"  pre_build:",
"    commands:",
"      - pip3 install troposphere",
"      - pip3 install awacs",
"      - git clone https://github.com/
    AwsLambdaContinuousDelivery/
    AwsLambdaContinuousDeliveryTools.git",
"      - cd AwsLambdaContinuousDeliveryTools
    ",
"      - pip3 install .",
"      - cd ..",
"      - rm -rf
    AwsLambdaContinuousDeliveryTools",
"      - wget https://raw.githubusercontent.
    com/AwsLambdaContinuousDelivery/
    AwsLambdaContinuousDeliveryLambdaCfGenerator
    /v2/createCF.py",
"  build:",
"    commands:",
"\n",
{
  "Fn::Join": [
    "\n",
    [
      {
        "Fn::Join": [
          " ",
          [
            "      - python3 createCF.py --
                path $(pwd)/ --stage",
            "Gamma",
            "--stack",
            {
              "Fn::Sub": "${AWS::StackName}"
            },
            ">> stackGamma.json"
          ]
        ]
      },
      {
        "Fn::Join": [
          " ",
          [
            "      - python3 createCF.py --
                path $(pwd)/ --stage",
            "PROD --stack",
            {
              "Fn::Sub": "${AWS::StackName}"
```

```
                            },
                            ">> stackPROD.json"
                          ]
                        ]
                      }
                    ]
                  ]
                },
                "\n",
                {
                  "Fn::Join": [
                    "\n",
                    [
                      "artifacts:",
                      "  files:",
                      "    - stackPROD.json",
                      "    - stackGamma.json"
                    ]
                  ]
                }
              ]
            ]
          },
          "Type": "CODEPIPELINE"
        }
      },
      "Type": "AWS::CodeBuild::Project"
    },
    "unittestBuilder": {
      "Properties": {
        "Artifacts": {
          "Type": "CODEPIPELINE"
        },
        "Environment": {
          "ComputeType": "BUILD_GENERAL1_SMALL",
          "Image": "frolvlad/alpine-python3",
          "PrivilegedMode": "false",
          "Type": "LINUX_CONTAINER"
        },
        "Name": {
          "Fn::Sub": "${AWS::StackName}-unittestBuilder"
        },
        "ServiceRole": {
          "Ref": "UnittestRole"
        },
        "Source": {
          "BuildSpec": {
            "Fn::Join": [
              "\n",
              [
```

```
                "version: 0.2",
                "\n",
                "phases:",
                "  install:",
                "    commands:",
                "      - apk add --no-cache curl python
                    pkgconfig python3-dev openssl-dev libffi-
                    dev musl-dev make gcc",
                "      - pip3 install moto",
                "      - pip3 install boto3",
                "      - pip3 install troposphere",
                "      - pip3 install awacs",
                "      - pip3 install -r requirements.txt",
                "      - pip3 install pyyaml",
                "  pre_build:",
                "    commands:",
                "      - wget https://raw.githubusercontent.
                    com/AwsLambdaContinuousDelivery/
                    pyAwsLambdaContinuousDeliveryUnittest/
                    master/executable/testRunner.py",
                "  build:",
                "    commands:",
                "      - python3 testRunner.py"
              ]
            ]
          },
          "Type": "CODEPIPELINE"
        }
      },
      "Type": "AWS::CodeBuild::Project"
    }
  }
}
```

A.3 TestRunner.py

```
import os, sys, boto3
import argparse
import subprocess
import yaml

from typing import List

def getTests(path: str) -> List[str]:
  files = os.listdir(path)
  files = map(lambda x: "/".join([path, x]), files)
  files = filter(lambda x: x.endswith(".py"), files)
  return list(files)

def loadConfig(path: str) -> dict:
  config = {}
  with open (path + "/config/config.yaml", "r") as c:
    config = yaml.load(c)
  if not config:
    raise Exception("Empty config")
  return config

def getArn(config: dict, stack: str, stage: str) -> str:
  funcName = config["Name"] + stack + stage
  client = boto3.client('cloudformation')
  res = client.list_exports()
  while res is not None:
    for export in res["Exports"]:
      if export["Name"] == funcName:
        return export["Value"]
    if "NextToken" not in res:
      res = None
    else:
      res = client.list_exports(NextToken = res["NextToken"])
  raise Exception("No ARN found for " + funcName)

def getTestFolder(config: dict) -> str:
  if "TestFolder" in config:
    return config["TestFolder"]
  return "test"


def exec_tests(path: str, stack: str, stage: str):
  config = loadConfig(path)
  testfolder = getTestFolder(config)
  arn = getArn(config, stack, stage)
  tests = getTests("/".join([path, testfolder]))
  for test_file in tests:
    exec_cmd = " ".join(["python3", test_file, arn])
```

```
    result = subprocess.check_output(exec_cmd, shell=True)
    print(result)
  return 0

if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("-p", "--path", help="Path of the folder with the source \
                       -code of the aws lambda functions", type = str
                      , required = True)
  parser.add_argument("--stack", help="StackName", type = str
                      , required = True)
  parser.add_argument("--stage", help="Name of the stage", type = str
                      , required = True)
  args = parser.parse_args()
  exec_tests(args.path, args.stack, args.stage)
```

A.4 createCF.py

```
# By  Janos Potecki
#  University   College London
# January 2018

from typing import List, Dict, Tuple
import os
import sys
from troposphere import Template, Output, Export, GetAtt, Join, Ref, Parameter
from troposphere.iam import Role
from troposphere.awslambda import Function, Alias, Environment, Code

import argparse
import re
import yaml

regex = re.compile('[^a-zA-Z0-9]')
S3 = str
Key = str
Source = Tuple[S3, Key]

def loadConfig(path: str) -> dict:
  config = {}
  with open (path + "config/config.yaml", "r") as c:
    config = yaml.load(c)
  if not config:
    raise Exception("Empty config")
  return config


def toAlphanum(s: str) -> str:
  return regex.sub('', s)

class MissingFile(Exception):
  def __init__(self, message):
    self.message = message
  def __str__(self):
    return self.message


def getIAM(path: str, prefix: str, stackname: str, stage: str) -> Role:
  ''' Returns the get_iam() function in config/stage/iam.py '''
  iam_name = "iam"
  iam_path = "".join([ path, "config/", stage])
  iam_file = "".join([ iam_path, "/", iam_name, ".py"])
  if os.path.isfile(iam_file):
    # we need to import here the file and call the get_iam() function
    sys.path.insert(0, iam_path)
    iam_module = __import__(iam_name)
```

```
    ref_name = toAlphanum("".join([prefix, stackname, stage]))
    return iam_module.get_iam(ref_name + "IAM")
  else:
    raise MissingFile("IAM file missing:" + iam_file)


def getEnvVars(path: str, prefix: str, stage: str) -> dict:
  ''' Returns the Env Vars saved in config/stage/envVars.py '''
  env_name = "env"
  env_path = "".join([path, "config/", stage])
  env_file = "".join([env_path, "/", env_name, ".py"])
  if os.path.isfile(env_file):
    sys.path.insert(0, env_path)
    env_module = __import__(env_name)
    env = env_module.get_env()
    return env_module.get_env()
  else:
    return {}


def getFunctionAlias( path: str
                    , prefix: str
                    , arn: str
                    , stackname: str
                    , stage: str
                    ) -> Alias:
  ''' Returns the Alias if present '''
  name = "".join([prefix, stackname, stage])

  return Alias( toAlphanum(name) + "Alias"
              , Description = "Automatic Generated Alias"
              , FunctionName = arn
              , FunctionVersion = "$LATEST"
              , Name = name
              )


def folders(path: str) -> List[str]:
  ''' Returns all Folders in the paths except the `lambdaCICDBuilder folder'''
  # TODO: Pretty ugly, we should find a better way
  xs = os.listdir(path)
  xs = filter(lambda x: x[0] != ".", xs)
  xs = filter(lambda x: os.path.isdir(path + x), xs)
  return list(xs)


def getLambda( name: str
             , src: Source
             , role: Role
             , stack: str
```

```
99              , stage: str
100             , env_vars: dict
101             , config: dict
102             ) -> Function:
103   ''' Takes the source code and an IAM role and creates a lambda function '''
104   code = Code( S3Bucket = src[0]
105              , S3Key = src[1]
106              )
107   func_name = "".join([name, stage])
108   env_vars = Environment( Variables = env_vars )
109   memory = 128
110   if "MemorySize" in config:
111     memory = config["MemorySize"]
112   timeout = 60
113   if "Timeout" in config:
114     timeout = config["Timeout"]
115
116   return Function( toAlphanum(name)
117                  , FunctionName = func_name
118                  , Handler = config["Handler"]
119                  , Code = code
120                  , Role = GetAtt(role, "Arn")
121                  , Runtime = "python3.6"
122                  , Environment = env_vars
123                  , MemorySize = memory
124                  , Timeout = timeout
125                  )
126
127
128 def addFunction( path: str
129                , template: Template
130                , stackname: str
131                , stage: str
132                , src: Source
133                ) -> Template:
134   ''' Takes a prefix & adds the lambda function to the template '''
135   config = loadConfig(path)
136   name = config["Name"]
137   iam_role = getIAM(path, name, stackname, stage)
138   template.add_resource(iam_role)
139   env_vars = {}
140   for key, value in getEnvVars(path, name, stage).items():
141     env_vars[key] = value
142   func = getLambda(name, src, iam_role, stackname, stage, env_vars, config)
143   func_ref = template.add_resource(func)
144   alias = getFunctionAlias(path, name, GetAtt(func_ref,"Arn"), stackname, stage)
145   if alias is not None:
146     template.add_resource(alias)
147   template.add_output([
148       Output( toAlphanum(name + stackname + stage)
```

```
            , Value = GetAtt(func_ref, "Arn")
            , Export = Export(name + stackname + stage)
            , Description = stage +": ARN for Lambda Function"
            )])
  return template


def getTemplate( path: str
               , stackname: str
               , stage: str
               ) -> Template:
  ''' Transforms a folder with Lambdas into a CF Template '''
  t = Template()
  src_key = t.add_parameter(
    Parameter( "S3Key"
             , Type = "String"
             , Description = "S3Key"
             )
    )
  src_s3 = t.add_parameter(
    Parameter( "S3Storage"
             , Type = "String"
             , Description = "S3Storage"
             )
    )
  src = (Ref(src_s3), Ref(src_key))
  t = addFunction(path, t, stackname, stage, src)
  return t

if __name__ == "__main__":
  parser = argparse.ArgumentParser()
  parser.add_argument("-p", "--path", help="Path of the folder with the \
                      source-code of the aws lambda functions"
                     , required = True)
  parser.add_argument("--stage", help="Name of the stage", type = str
                     , required = True)
  parser.add_argument("--stack", help="Name of the stack", type = str
                     , required = True)
  args = parser.parse_args()
  t = getTemplate(args.path, args.stack, args.stage)
  print(t.to_json())
```

B Interim Report and Project Plan

Interim Report
Flight Zipper

Supervisor: Dr Graham Roberts

Janos Potecki

January 2018

1 Achievements

1.1 AWS Lambda Continuous Delivery

In order to reduce costs it is cheaper to run the backend on as serverless service with AWS Lambda as the backbone. But during my research it turned out that there is no framework, which runs inside of AWS making use of AWS Pipeline which allows building, testing, and deploying AWS Lambda functions automatically (alternatives exists, but they either complicated paid solutions or saving keys somewhere on AWS which was seen as a potential security risk).

1.1.1 Change of Thesis Focus

What started as a small tool to achieve CI/CD for AWS Lambda for my web app project grew now into an project on its own, changing the nature of the project from a web app to an CI/CD tool for AWS Lambda with an web app as verification for correctness.

1.1.2 Implemented Features

The CI/CD tool, called AWS Lambda Continuous Delivery (ALCD), is already in a state, where it can be used in order to deploy Lambda Functions hosted on GitHub or AWS CodeCommit. It contains on several packages, which have a consistent naming and are separated between language dependent (in which the AWS Lambda is programmed) and language independent parts. The result is an architecture which consists of several independent building blocks which can be interchanged independently to support multiple languages or features. Language indepent packages start simply with AwsContinuousDelivery< *packagename* > where language specific packages have a prefix which corresponds to the language file extension (e.g. .py for Python).

1.1.3 Packages for AwsLambdaContinuousDelivery

The following list gives a short overview of the existing packages. The current only supported language is Python. The repository is public and can be found at `https://github.com/AwsLambdaContinuousDelivery`.

- AwsLambdaContinuousDeliverySourceCodeCommit *language independent*
  Package to use AWS CodeCommit as git repository for source code
- AwsLambdaContinuousDeliverySourceGitHub *language independent*
  Package to use Github as git repository for source code
- pyAwsLambdaContinuousDeliveryUnittest *Python specific*
  Package to run unittests on the source code. Automatically installs all dependencies which are specified in a *requirements.txt*
- pyAwsLambdaContinuousDeliveryBuild *Python specific*
  Package to generate the deployment package which will be saved in AWS S3
- pyAwsLambdaContinuousDeliveryLambdaCfGenerator *Python specific*
  Package in order to generate the AWS Cloudformation templates for each stage using environment details specified in the specific configuration files
- AwsLambdaContinuousDeliveryDeploy *language independent*
  Package generating code in order to deploy the templates generated by a build package to AWS
- pyAwsLambdaContinuousDeliveryTest *Python specific*
  Runs integration tests coded by the programmer for each stage
- AwsLambdaContinuousDeliveryTools *language independent*
  A collection of tools which might be helpful to develop more packages.
- AwsLambdaContinuousDeliveryPipeline *Python specific*
  An example repository showing how a pipeline for AWS Lambda could be built using the provided libraries.

2 Future Work

2.1 PyPI

In order to make the tool usable it should be possible to install it on local machines via *pip*. To make it available there, some more features and testing should be made. By the end of the term it shouldn't be assumed that the tool will be mature enough to be published on PyPI but on TestPyPI instead for further testing.

2.2 Further Supported Features

2.2.1 VPC Support

Sometimes users want to use AWS just for internal use and thus allowing them to just interact within an VPC. For this reason one milestone before Reading Week is to add VPC support, which can be configured for each state.

`https://github.com/AwsLambdaContinuousDelivery/pyAwsLambdaContinuousDeliveryLambdaCfGenerator/issues/5`

2.2.2 Install Custom Libraries for Integration Tests

In order to give users the freedom to choose whatever libraries they want to run their integration tests they need a way to install these python packages on their own. As this shouldn't be part of the deployment package which gets deployed a separate *requirements.txt* for integration tests is needed.

`https://github.com/AwsLambdaContinuousDelivery/pyAwsLambdaContinuousDeliveryIntegrationTests/issues/1`

2.2.3 Notifications

Currently the user doesn't get informed if something breaks in the pipeline and as a result the user has always to check the pipeline status manually. This is a bit inconvenient. It would be better to be able to publish events on topics automatically which then can notify the user. At this point it is not clear how much complexity this involves and as a result it might not be implemented before Reading Week.

`https://github.com/orgs/AwsLambdaContinuousDelivery/projects/1#card-6770831`

2.2.4 Verification App

The focus will be on the web app FlightZipper, which will consists of a few Lambda functions. This web app will be used in order to verify the tool and detected bugs, bad practises, and possible improvements.

2.2.5 Thesis

After Reading week the bulk of programming will be expected to be done and the focus will be on writing the thesis. Minor changes and improvements might still be done during this process.

FlightZipper
Project Plan

Janós Potecki
Supervisor: Dr Graham Roberts

October 2017

1 Aims

The goal is to create an minimal viable product (mvp) for the FlightZipper service, which is powered by a completely automated deployment and testing pipeline for the various micro-services powering the service.

2 Objectives

1. Research the latest features an possibilities of cloud service providers like AWS and Azure.
2. Familiarise with the tools in order to use the correct tools for the appropriate task
3. Use Infrastructure as Code (IaC) in order to orchestrate and deploy the service
4. Focus on security and scalability
5. Operate on a tight budget
6. Automate deployment of new features
7. Automate tests before deployment to production

3 Expected Deliverables

1. Fully documented, deployed, and functional mvp
2. Strategy for testing and evaluating your software or equivalent
3. Design specification for a software application or equivalent specification for relevant deliverables

4. IoC which can be easily deployed to other regions or for review/marking purposes
5. Deployment pipeline containing different stages and tests

4 Work Plan

Project Start – End October

- Initial Research
- Choose Platform
- Start building deployment platforms for frontend
- Start building deployment platforms for backend

Begin November – Mid November

- Finalise Pipelines
- Work on CI/CD architecture
- Focus on IaC

Mid November – End December

- Start building mvp of FlightZipper
- Implement tests

Begin January – Mid February

- Implement integration tests
- Iterate over the mvp
- Write interim report

Mid February – End March

- Work on final report