

Bachelor's Thesis

Bachelor of Engineering, Information and Communications Technology

2022

Roope Westman

Automating a Small-Scale Cloud Environment



Bachelor's Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2022 | Pages 31

Roope Westman

Automating a Small-Scale Cloud Environment

Cloud environments are a modern way to create easily scalable cost-efficient IT infrastructures. The objective of this thesis was to create an automated small scalable laboratory environment in the cloud. The environment should use cloud service providers' services and be made to replace a hardware-based laboratory.

To achieve the objective of this thesis, the most common cloud service providers were researched and compared with a particular focus on tools for automating cloud environments. The sources used were internet articles and resources offered by cloud service providers, such as guides and documents.

Based on the findings of the above mentioned comparison of cloud environment automation tools, the plan was to build this environment using a declarative tool such as AWS (Amazon Web Services) CloudFormation. However, this was deemed too difficult to achieve with limited resources, and procedural tools were used instead.

As a result, an automated environment was built using AWS Lambdas as the main component. It is supported by other services such as AWS SQS, AWS Step Functions, and AWS EventBridge.

Keywords:

Cloud computing, Automation, AWS, IaaS

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja Viestintätekniikka

2022 | 31 sivua

Roope Westman

Pienen mittakaavan pilviympäristön automatisointi

Pilviympäristöt ovat moderni tapa luoda helposti skaalautuvia ja kustannustehokkaita tietoteknisiä infrastruktuureja. Tämän opinnäytetyön tavoite oli luoda pilveen pieni automatisoitu skaalautuva laboratorioympäristö. Ympäristön on tarkoitus käyttää pilvipalvelutarjoamia palveluita ja sen tarkoitus on korvata laitteistopohjainen laboratorio.

Tämän opinnäytetyön tavoitteen saavuttamiseksi, on tutkittu ja vertailtu yleisimpiä pilvipalvelutarjoajia erityisesti keskittyen pilviympäristöjen automatisoinnissa käytettäviin työkaluihin. Käytettyjä tutkimuksen lähteitä olivat internet-artikkelit ja pilvipalvelutarjoajien tarjoamat resurssit, kuten ohjeet ja dokumentaatiot.

Tutkimus osuuden pohjalta oli tarkoitus rakentaa tämä ympäristö käyttämällä deklarativista työkalua, kuten AWS (Amazon Web Services) CloudFormation. Tämä kuitenkin katsottiin liian vaikeaksi saavuttaa rajoitetuilla resursseilla, ja sen sijaan käytettiin proseduraalisia työkaluja.

Lopputuloksena rakennettiin automatisoitu ympäristö, jossa AWS Lambdat toimii pääkomponenttina. Sen tukena muita palveluita, kuten AWS SQS, AWS Step Functions ja AWS EventBridge.

Asiasanat:

Pilvipalvelut, Automaatio, AWS, IaaS

Content

List of abbreviations (or) symbols	6
1 Introduction	7
2 Cloud Service Providers	9
2.1 Differences	9
2.2 Pricing	10
2.3 Selecting the provider	11
3 Infrastructure as Code Tools	12
3.1 Considerations for IaC Tools	12
3.1.1 Declarative and Procedural Tools and Languages	12
3.1.2 Mutable vs Immutable Tools	13
3.2 Selection of the Orchestration Tool	14
3.3 Extra Tools	15
3.3.1 AWS Cloud Development Kit	15
3.3.2 AWS Lambda	16
3.3.3 AWS Software Development Kits	16
3.3.4 AWS Step Functions	16
3.3.5 AWS EventBridge	17
4 Server Automation/Orchestration	18
4.1 Requirements for the Environment	18
4.2 Experience Creating the Environment	18
4.3 Implementation Plan	19
4.3.1 Automatic Startup and Shutdown	20
4.3.2 Termination of Expired Instances	21
4.4 Integration to a Data Source	22
4.5 Tracking costs and Billing Details	23
4.6 Testing the Automation	23
5 Conclusion	27
References	29

Appendices

Appendix 1. Instructions for Setting Up Lab Environment

Figures

Figure 1. Gartner Magic Quadrant for Cloud Service Providers [4].	9
Figure 2. Sequence Diagram of EC2 Creation.	20
Figure 3. Automatic Startup and Shutdown.	21
Figure 4. Termination of Instances.	22
Figure 5. Integration to a Service Portal.	22

Pictures

Picture 1. Example Workflow from AWS Step Functions.	17
Picture 2. Sending a Message to the SQS Queue.	23
Picture 3. CreateEC2 Logs from AWS CloudWatch.	24
Picture 4. Running Instance in AWS Console.	25
Picture 5. State Machines Execution Waiting for Termination Date.	25
Picture 6. SSH Connection to the Instance.	26
Picture 7. TerminateEC2 Lambda Logs.	26

Tables

Table 1. Cloud Provider Cost Comparison.	10
------------------------------------------	----

List of abbreviations (or) symbols

AMI	Amazon Machine Image
AWS	Amazon Web Services
CDK	Cloud Development Kit
CPU	Central Processing Unit
EC2	Elastic Compute Cloud
GB	Gigabyte
IP	Internet Protocol
IT	Information Technology
IaC	Infrastructure as Code
RAM	Random Access Memory
RDP	Remote Desktop Protocol
RSA	Rivest–Shamir–Adleman
S3	Simple Storage Service
SDK	Software Development Kit
SQS	Simple Queueing System
SSD	Solid State Drive
SSH	Secure Shell
YAML	Yet Another Markup Language

1 Introduction

This thesis aims to create a small, easily scalable laboratory environment that is to be built using the services of a public cloud service provider. The main use for this lab environment is to quickly test new technologies and software in projects that last about a month or two. The environment should be as autonomous as possible to reduce maintenance and labor costs, while still meeting certain requirements regarding uptimes and costs which are specified later.

This was ordered by a company that wants to increase their public cloud usage while at the same time solving a problem they are having. The problem that the company in question is facing is that their small-scale laboratory environment is built upon physical hardware that is hard to maintain and requires lots of labor to keep working. Because the infrastructure is not maintained by a specific person or a team, it leads to some servers staying up for far longer than originally planned. Physical hardware also requires a lot more time to set up and install for it to be usable for testing. For example, you could order some hardware and have issues with compatibility, user rights, or a myriad of other issues and in a big company small issues like these can take up to months to solve. [1], [2]

Implementation would be pretty similar to a classroom lab in the sense that a virtual machine will be created on-demand with the specified parameters and all the work is handled by automated programs. There are plenty of websites that use this kind of technology to their advantage, for example, TryHackMe.com has assignments that start up a virtual machine with the required files and vulnerabilities for people to practice their cyber security and hacking skills. [3]

Objectives for this thesis are: to familiarize and learn to work with cloud providers and tools, and successfully create an automated simple environment that launches virtual instances and also automatically takes them down after a specific period of time.

This thesis will be built of two parts: theory and implementation. In the theory segment, cloud service providers and their tools will be explored, and using found resources, a baseline will be created for the implementation part. In the implementation segment, the concrete portion of the work will be done using methods best found for success.

Outside the scope of this thesis is the actual integration to a platform that you could order these machines from. Possible implementations of this will be discussed but will not be tested. This thesis also will not discuss security-related issues or best practices and will mainly focus on automation.

2 Cloud Service Providers

There are three major cloud service providers that offer the most services, availability, and reliability, and this thesis will focus only on these providers. These three providers are Amazon Web Services, Microsoft Azure, and Google Cloud as seen in Figure 1. There are also many smaller providers in the field that may be excellent for specific use cases, but they usually don't offer as a wide selection of services as the major providers. [4]



Figure 1. Gartner Magic Quadrant for Cloud Service Providers [4].

2.1 Differences

The purpose of this comparison is not to give a full overview of all three of the major cloud providers but to instead compare them in the scope of this project. The most important points of comparison are tool availability and ease of use.

2.2 Pricing

These three major providers all offer a simple calculation tool for calculating the price of the deployment. It should be noted that these prices change regularly and better deals can most likely be made with sales representatives and/or by increasing the order size and duration. There are many resources [5]–[7] that have compared these prices but for this thesis new up-to-date calculations were made using tools provided by the cloud providers. [8]–[10]

Calculations used in this thesis are made with prices according to 4.1.2022 and all instances in the calculations have 8 vCPUs, 32GB of RAM, 174 hours of monthly usage per instance, region is set as close as possible to London. Google and AWS calculations use 30GB SSD for storage and Azure uses 64GB SSD.

Table 1. Cloud Provider Cost Comparison.

	AWS	Google	Azure
vCPU	8	8	8
RAM(GB)	32GB	32GB	32GB
Storage	30GB	30GB	64GB
hour cost per instance	0,27 €	0,30 €	0,32 €
instances 10, hours 174 per month	462,99 €	529,99 €	558,00 €
Storage	30,78 €	53,94 €	included
Total	493,77 €	583,93 €	558,00 €

As it is evident from Table 1, the prices of these providers don't vary too wildly as the hour cost per instance is almost the same on all, and major differences in the end prices are mostly due to cost differences in storage solutions and multiple different things affecting the per hour cost like the location of the data center. All of these are still much cheaper than the alternative of running your own physical infrastructure where it might take up to multiple days just to set up the hardware and it is easier to optimize costs in a cloud environment with

automatic shutdowns and with other services that cloud providers provide for saving costs like different pricing plans for example.

2.3 Selecting the provider

Our use case for the cloud provider is a common use case that any of the major cloud providers could provide the required services for, and as such it does not matter too much which provider is chosen for the project from a technical standpoint.

Selection criteria will weigh heavily on features and tools the provider provides. Since price is too hard to measure in a corporate use without the use of sales representatives, it will be left out of the selection criteria.

This project is going to use AWS and that is mostly because it offers more tools for the job compared to the other major providers. AWS offers the use of both of the biggest IaC tools, Terraform and its own proprietary AWS CloudFormation while others offer only the use of Terraform. These tools will be explored later in this thesis in Chapter 3 Infrastructure as Code Tools.

Another reason for the selection of AWS is because the company that this is being provided for and the personnel working there are more familiar with AWS than the other platforms and have already accounts and some infrastructure in place for AWS automation.

3 Infrastructure as Code Tools

3.1 Considerations for IaC Tools

In this chapter, I will go through some of the considerations to keep in mind when selecting the correct tools for the job. IaC (Infrastructure as Code) tools can be categorized in a few ways but what it boils down to is configuration orchestration and configuration management. Usually, these tools are used together by first using an orchestration management tool to build the infrastructure and then using configuration management to configure the software on the provisioned infrastructure. [11]

The most important aspect to keep in mind regarding this project is that are the tools used either declarative or procedural and mutable or immutable. Further considerations might be that should the tool have an agent like Chef and Puppet or be agentless like Ansible and SaltStack, but that does not concern cloud environments and configuration orchestration tools as they will install the agents on the clients if needed. [12]

3.1.1 Declarative and Procedural Tools and Languages

Procedural tools (ex. Chef and Ansible) use a language in which you define step-by-step how to reach a certain goal and declarative tools (ex. CloudFormation and Terraform) are tools where you state the desired end state that you want to reach and the tool itself decides the best way to reach that said end state.

The main difference that comes in using these tools is that when changing the infrastructure the procedural tools do not keep track of the infrastructure and you would always need to know it by yourself when making changes.

For example, say that you have an infrastructure with 5 instances and you want to reach a state where you have 10 instances. You would first need to know

how many instances you have and add as many as are missing to the infrastructure and write a new snippet of code or re-use and modify an old one. This may lead to a case where your infrastructure may have more or fewer instances than you wanted if they are not tracked properly.

In the declarative version of this example, you would state that you want 10 instances and the tool would add as many as are needed or remove any excess instances if you have over 10 instances. [13]

This project will focus on using declarative tools as the main component because of the infrastructure tracking nature of them and support these tools with procedural tools if needed. For example, the infrastructure would be built using a declarative tool such as CloudFormation and inside the infrastructure, the instances could be modified using Ansible scripts.

3.1.2 Mutable vs Immutable Tools

Mutable infrastructure is the traditional way of handling servers and means that the updates and modifications to a server modify the existing server and apply to it. This has the benefit of infrastructure fitting the specific needs and can be usually be updated and adapted faster to changes in requirements. Downsides on the other hand include harder diagnosing and reproduction of technical issues, more difficult version tracking, and added complexity of the infrastructure. Tools that use mutable infrastructure are for example Chef, Puppet, Ansible, and Saltstack.

Immutable infrastructure is the new way of handling server infrastructure and the essence of it is that you deploy something once and cannot modify it without deleting the instance and creating a new one with the updated modifications. This has many benefits for larger infrastructures such as better version tracking and easier rollbacks, more predictability, and reduced complexity of the servers as they are never modified. Downsides include that data storage for these servers must be externalized and you cannot modify existing servers in case of

errors. Tools that use immutable infrastructure are for example Terraform, Kubernetes, and CloudFormation [14], [15]

This project will focus on using immutable infrastructure since the environment will not have a dedicated user or support group which causes the infrastructures predictability to drop drastically since there will be no one to make proper documentation and version tracking.

3.2 Selection of the Orchestration Tool

Since the main goal is to set up an ever-changing environment, orchestration management is the category that should be looked at first as they are made to handle building and managing infrastructure as their main purpose. Building a huge environment using state management tools like Ansible might be possible but would require careful planning and would most likely have many issues to overcome that IaC tools handle automatically.

The three biggest tools at the moment are Terraform, Cloudformation, and Pulumi. Out of these three Pulumi is the newest so it has a smaller community to learn from and is the only one with costs associated with it regarding the number of users and resources used, and because of that, it is not a great fit for this project. [16], [17]

Terraform and CloudFormation both have big communities, years of experience, and their own strengths. Terraform is an open-source project by Hashicorp and CloudFormation is a service offered by Amazon Web Services. Terraform supports multi-cloud and CloudFormation is only usable on AWS but since we chose AWS as the cloud provider and the project is not planning to be used in multi-cloud environments it is not a key factor.

This project is going to use CloudFormation. The main reasons for selecting CloudFormation are that it supports automatic rollbacks if the deployment fails, state management (keeping track of resources under management) is handled by AWS and it handles secrets more securely than Terraform. These features

are useful in reducing the building complexity of the project as Terraform does not support these natively and you would have to get additional third-party modules or they might not even exist. There are many features that Terraform has that CloudFormation does not have but since this is a simple environment most of those features are not important. Additional factors in the decision were that the company in question has prior experience with CloudFormation and the consensus among experts is to use CloudFormation when working with AWS. [18], [19]

3.3 Extra Tools

These are tools that might be useful while building the infrastructure or automating certain tasks and should be kept in mind

3.3.1 AWS Cloud Development Kit

The AWS CDK is a software development framework which purpose is to allow users to define cloud infrastructure using more human-readable programming languages such as JavaScript, Python, C#, and many others, and support for other languages are in the works. Since CDK is an abstraction of CloudFormation it compiles into CloudFormation templates as the final product.

Compared to writing templates with YAML in CloudFormation, CDK is easier for humans to read and design as programming languages used by CDK are concise and easier to comprehend. It also allows the use of programming idioms like loops, conditionals, and many more which allow you to design a more flexible infrastructure. Shorter programs are easier to maintain and it removes a lot of the complexity of template files in the designing phase. [20], [21]

The downsides of CDK are that since it is an abstraction of CloudFormation users can experience unexpected behavior if they do not understand how CloudFormation itself works and as such it should not be used to skip the learning curve of provisioning in the cloud. A major downside also is that since CDK is a young framework it is also still constantly changing and there might be new builds that break deployments because of deprecated methods or reworked constructs. [22]

3.3.2 AWS Lambda

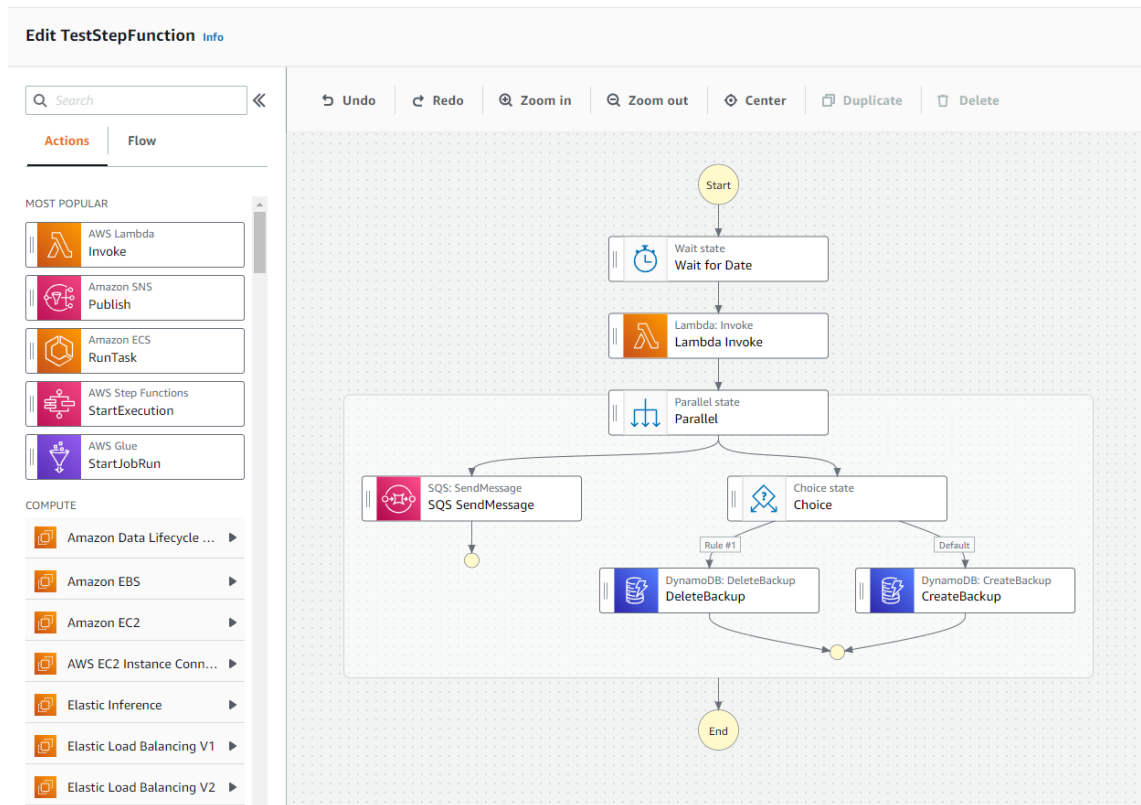
AWS Lambda is a service that allows you to run code, applications, and services without provisioning or managing servers yourself. AWS handles the management of these servers and you only pay for the time that the resources are used for. It is a powerful tool as it easily scales for high demand and is highly integrated into other services offered by AWS. [23]

3.3.3 AWS Software Development Kits

AWS also offers SDKs for many programming languages that allow you to access the Application Programming Interfaces offered by AWS and integrate your preferred programming languages with AWS services. This can be used to easily create and manage resources in your infrastructure [24]

3.3.4 AWS Step Functions

AWS Step Functions offers the creation of visual workflows that can be used to build and automate IT processes and pipelines using other AWS services. As seen in Picture 1, there is an easy-to-use visual interface for creating complex automated workflows with high integration to other AWS services. [25]



Picture 1. Example Workflow from AWS Step Functions.

3.3.5 AWS EventBridge

AWS EventBridge allows the creation of rules that trigger specified events such as Lambda functions, Step Functions, or other user-connected applications. For example, you could set a backup function to launch every day at a specific time without creating complicated scripts and event tracking software. Since there are no servers to provision or manage and scaling is handled by AWS it is easy to reduce operational costs compared to self-built systems. [26]

4 Server Automation/Orchestration

4.1 Requirements for the Environment

The major requirements for the environment are that instances in it are only running during work hours (6:00 to 18:00) and that the lifetime of each instance should be specified and be short-lived (1 to 3 mo.). The amount of users using this environment is about 10-20, but this can easily be scaled up as the cloud is easily scalable.

For now, it only needs to support basic operating systems such as different Linux distributions and Windows versions.

Permissions for SSH and RDP access are also required and login to the Linux instances will be done using RSA keys. AWS instances come with predetermined accounts depending on the image type used, for example, Windows default user is “Administrator” and Amazon’s Linux image comes with the username “ec2-user”.

4.2 Experience Creating the Environment

In this concrete portion of the thesis, it was realized that the tools and methods discussed in the theory segment of this thesis might be possible to do but would require considerably more planning, time, or personnel to implement well.

What was learned first, was that using pure CloudFormation is not suited for this type of project that well, as it is not as easy to write and there are easier tools to use that create the CloudFormation templates for you with a lot less effort. One such tool is AWS CDK. One major hurdle was that automating a constantly changing CloudFormation template is not as easy as using other tools to generate new templates based on new requirements. CloudFormation works better on a more static infrastructure that is changed by hand or an automation

pipeline where CloudFormation is the end product of another automation software, rather than being the core tool used for automation.

CDK was a great tool that had all the features that you would want when building an environment like this, but unfortunately, it is still quite young in terms of documentation and resources and that was the main reason it did not work out. It was hard to diagnose where the issues were when problems were encountered and there was not enough of a footprint of people experiencing the same issues to easily find answers. It also has the option to use many programming languages which makes it harder to find a solution for your preferred language.

After failing to build the environment using CloudFormation and CDK, Lambdas were the next step and also the place where success was found. Lambdas were easy to use and implement in AWS and they are fast and cheap. The biggest issue with using Lambdas is that we lose a lot of the benefits of using declarative tools which were mentioned earlier in chapter 3.1.1, as Lambda is a procedural tool it will create an instance even if an error is found later. This can be solved using proper error handling but requires a lot of knowledge in programming and much more testing than what a declarative tool would require. But the ease of setting up the environment, all the ready-made integrations to Lambda, make it easy to teach others and make it easily modifiable which are great benefits that in this case outweigh the negatives.

The full details of the implementation can be found in Appendix 1.

4.3 Implementation Plan

The final plan for the implementation is represented in Figure 2 and it is to have a message queueing service (AWS Simple Queue Service) wait for a package with the required parameters for execution of the environment. When the AWS SQS receives a payload, it executes a Lambda function that first creates a new subnet, key pair, AWS EC2 instance, and after that, it creates a state machine in AWS Step Functions linked to that EC2 instance and finally a state execution

for the state machine which waits for the termination date of the EC2 instance. We also set up tags in all the elements of the environment for billing and easy grouping in programming. An example of a tag could be: Name: “awsLabAutoTerminate”, Value: “True” which could be used to allow the termination of an element if set to “True” or disallow with set to “False”.

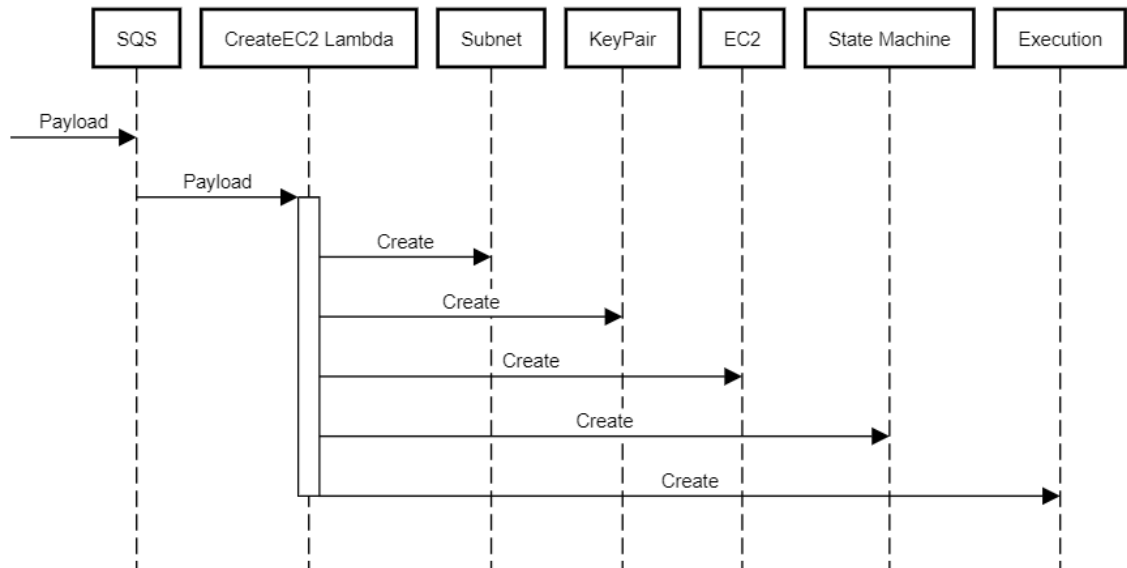


Figure 2. Sequence Diagram of EC2 Creation.

4.3.1 Automatic Startup and Shutdown

As shown in Figure 3 we also use AWS EventBridge to create two events, one for starting up EC2 instances and one for shutting down EC2 instances. Both of these trigger their respective Lambda functions StartEC2 and StopEC2 at specified times and will affect every EC2 instance tagged with either “awsLabAutoStart” or “awsLabAutoStop”. So if there is a need to either not shut down or start up an EC2 instance these tags can be manually removed or set to the value “False” from the EC2 clients.

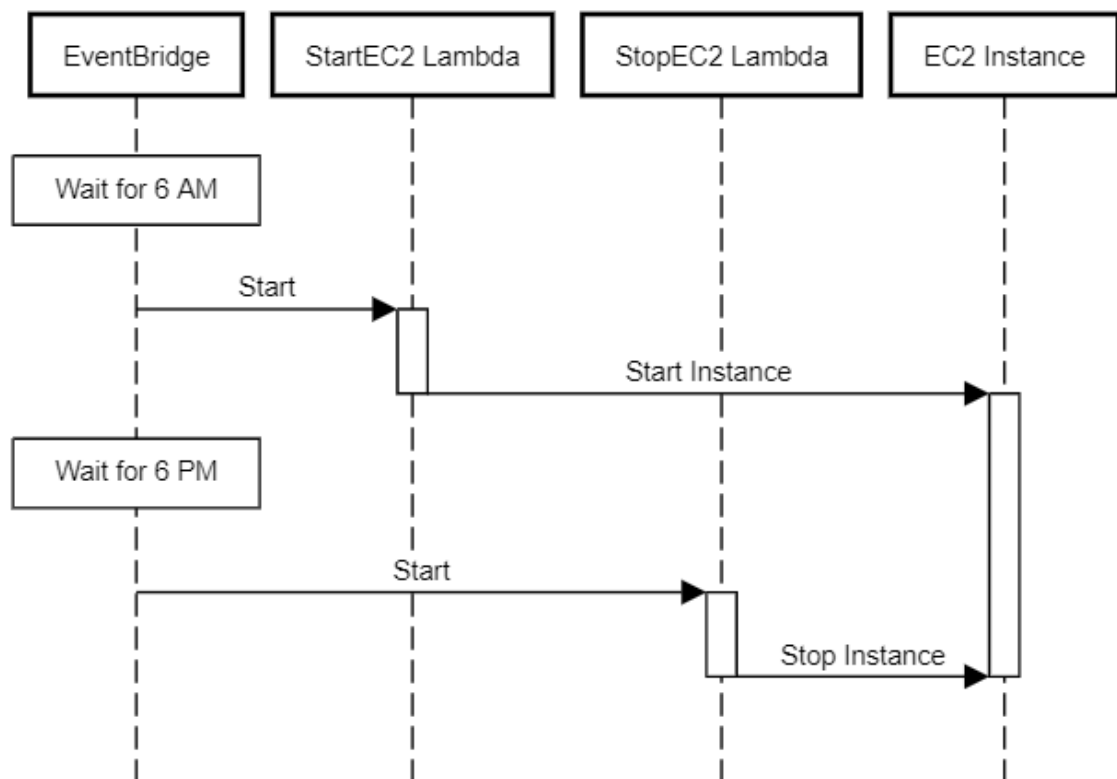


Figure 3. Automatic Startup and Shutdown.

4.3.2 Termination of Expired Instances

Termination of expired EC2 instances is handled with AWS Step Functions as shown in Figure 4. These Step Functions have a state machine that waits for a certain time, in our case the termination date, to arrive and when it does it executes a Lambda function which deletes the EC2 instance and all the components related to it and finally also deletes the state machine itself. This script is also set up to affect only instances with the tag “awsLabAutoTerminate” set to “True”, so if you want to avoid termination of an instance, set this flag to false. There is also the possibility of setting machines to hibernate first and terminate after a certain period and this would reduce the risk of accidental deletion of data that is still being used.

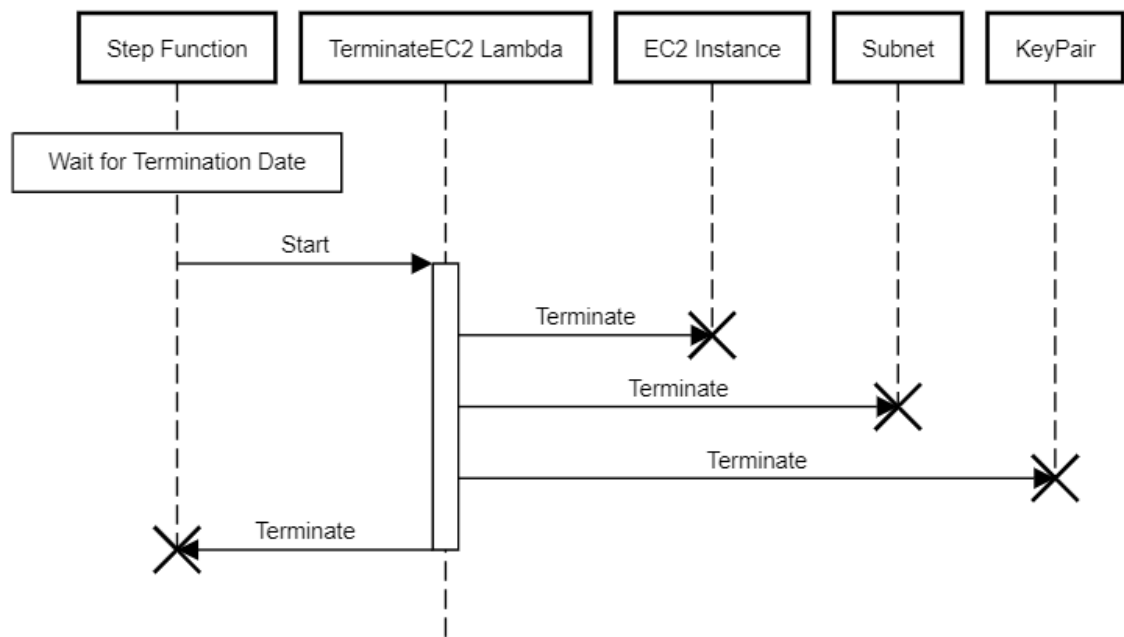


Figure 4. Termination of Instances.

4.4 Integration to a Data Source

The idea is for this AWS environment to be integrated into an existing service portal or ticketing system for ordering user rights and servers that would send data payloads to the AWS SQS with the required parameters and in return receive a payload with the required information to access the EC2 instance like for example user info and IP addresses to connect to it as shown in Figure 5.

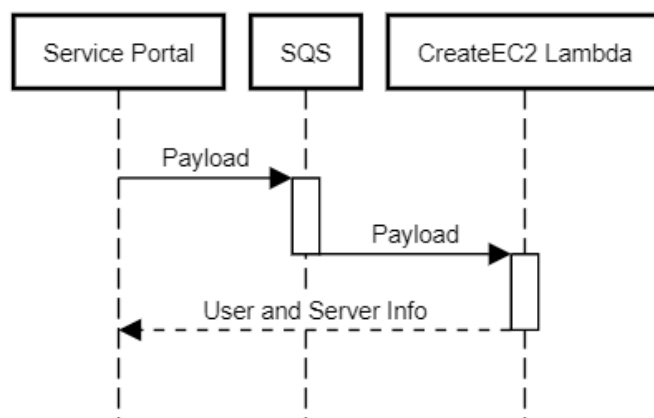


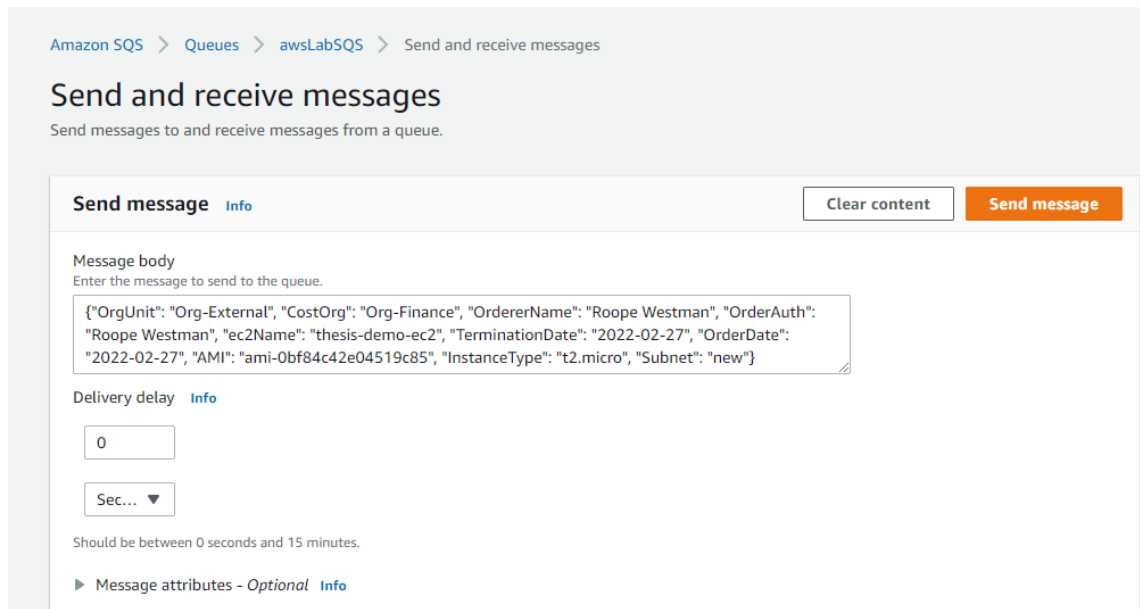
Figure 5. Integration to a Service Portal.

4.5 Tracking costs and Billing Details

Since one of the main features of this project was to reduce costs compared to a hardware-based lab environment, there should be a way to track expenses accumulated by the cloud environment. AWS allows the creation of cost reports which can be organized by user-created tags or AWS-generated tags. These cost reports then can be saved to an AWS S3 bucket as a .csv file and exported from there to where it is needed or processed further. Since this project was made using the free tier of AWS services no cost report can be made as an example.

4.6 Testing the Automation

Since there is no integration in place from an endpoint to SQS a direct message will instead be sent to the SQS queue using AWS console like shown in Picture 2. This message contains the AMI of an Amazon Linux machine and as such, that is what will be deployed. By changing the AMI in the payload other operating systems can also be deployed.



Amazon SQS > Queues > awsLabSQS > Send and receive messages

Send and receive messages

Send messages to and receive messages from a queue.

Send message [Info](#)

[Clear content](#) [Send message](#)

Message body
Enter the message to send to the queue.

```
{
  "OrgUnit": "Org-External",
  "CostOrg": "Org-Finance",
  "OrdererName": "Roope Westman",
  "OrderAuth": "Roope Westman",
  "ec2Name": "thesis-demo-ec2",
  "TerminationDate": "2022-02-27",
  "OrderDate": "2022-02-27",
  "AMI": "ami-0bf84c42e04519c85",
  "InstanceType": "t2.micro",
  "Subnet": "new"
}
```

Delivery delay [Info](#)

▼

Should be between 0 seconds and 15 minutes.

► [Message attributes - Optional](#) [Info](#)

Picture 2. Sending a Message to the SQS Queue.

After the message is sent to the SQS queue it will be then sent to the Lambda function “CreateEC2”. In the logs, seen in Picture 3, it can be seen that the execution was successful and the instance, state machine, and its execution were all created properly in under 3 seconds.

Timestamp	Message
	No older events at this moment. Retry
2022-02-27T22:09:18.586+02:00	START RequestId: 98cd8ff1-d18a-55a4-a4ea-d5640d10acbf Version: \$LATEST
2022-02-27T22:09:18.587+02:00	InstanceType = t2.micro
2022-02-27T22:09:18.587+02:00	AMI = ami-0b0f84c42e04519c85
2022-02-27T22:09:18.923+02:00	Subnet is valid: 10.0.0.0/24
2022-02-27T22:09:19.377+02:00	-----BEGIN RSA PRIVATE KEY-----
2022-02-27T22:09:19.377+02:00	MIIEpAIBAAKCAQEAtoKfhGQcmj9EzYjcbHsKYNw9G2fxm5H5mz4HP0aXVKfvU2X
2022-02-27T22:09:19.377+02:00	ZSb2DNU2wTonQW9UKGaQq7x/574ITn5Kq4kncCa5jZuz39R2/zn/AfrxXwb3c9td
2022-02-27T22:09:19.377+02:00	LXC+CaxLtVgplmIg3TaRkMLB1NheUULE1bVy+ZA0cs75WLSnQaVwYGRjS2HSq0Q
2022-02-27T22:09:19.377+02:00	2m+QKOPiWIkLpDXZnq1o5cEqNVFUSabaQpQY65GvaNP1+E9a7B0HAG0/mUsIxsH
2022-02-27T22:09:19.377+02:00	ut2bEvOzYnI9FmuIDIS5QEQQV4savfZIdc8J6UsyoIBN1226tAs6MsrMk7Yj3p9s
2022-02-27T22:09:19.377+02:00	kucRMVpBYFe5am6KZHOFA2EFBc7gi38ZVFZFLQIDAQABAoIBAEX2oZaZHS9zzF6U
2022-02-27T22:09:19.377+02:00	BUI1q0se6tCU1tHC5BLFR0gEqig9HhGQbipKjXF17ryToXBzVG3gsaCBmLC9k/tW
2022-02-27T22:09:19.377+02:00	u+/r8wLyRnEG12OLhyNc5/KrcO6Q66I1eH8uKeAhmE3sojSTEK/1N68jhw8AZ5yJ2
2022-02-27T22:09:19.377+02:00	zVIX41CiatKLzbdJf46jqKA+qX13pLjZsyOQr0WKE5ST1bIDAyuraZ1Qje/oJo
2022-02-27T22:09:19.377+02:00	s4Ee6Q77et1YTeq1oQo5rwAMWdKag+sVeYkrgy5FZ8sAXvFlw8zblia3RRzDfnn
2022-02-27T22:09:19.377+02:00	BkVW/enqPaV6DARmfsNu/pr52uEekN0yYlphsv1HbioxOngjPzN2H/mxLDKjs0p6
2022-02-27T22:09:19.377+02:00	r1H3TGEcGYEA3iKgkRbrXR11ugd0FD0svf4j7HeK1HZvoRcE2mLDQvCIAR0ZIm+c
2022-02-27T22:09:19.377+02:00	E3HmdHsu2PknCsJt+uz1XD11107db4wtcc0pEk2hDrtdQI3XOHUjt07Xc1+Tunet
2022-02-27T22:09:19.377+02:00	1pQbFw1e1AMVWvNqyRsnhZUWxhUm3RI5d5qMPUL0jVAP7/viyBPPQH2UCgYEA01WI
2022-02-27T22:09:19.377+02:00	VzLjPMU1fhe7fK0BSEGDx7U08LfhGyFXdV38qE07jW3Wk+AXOTGrV4z67NeHsay
2022-02-27T22:09:19.377+02:00	5D5Q5p3WQ2cAp98KY46N2bys55c+10PjIK5q9w0+JzN3fzbdRfclUogTVpyBDAIaV
2022-02-27T22:09:19.377+02:00	yn1wmUo7cmHzaKx54TjDD6xgDEX75gnC492qzKcGYEA0CuxP+c0EmUwr7yVzf/p
2022-02-27T22:09:19.377+02:00	LPAC6HLq9sr7OALjC7RYet4B8s1ABEaG8K8EInrIXIG25eNozNm+9BuR9+2WNRgv
2022-02-27T22:09:19.377+02:00	8epzKAT5/109pbOXUw0j6UC1pVhuWmV0vO/I2nfeqYxY5QIzqmo8uXIj0V4FtuP
2022-02-27T22:09:19.377+02:00	+ccv7iEh/3UwbjTQV9Xjv7UCgYEA0utoJ3511geVdc0Hj+1eQ5HS0ucVe88puhv
2022-02-27T22:09:19.377+02:00	q7hv1ValFXtp5L6Hsc/cT7yq0jgVmeJFkxRaHCSXg5iCVnYLP+8f8nI53vWicK8
2022-02-27T22:09:19.377+02:00	nmVG77f/Dq6c0QpwyOT1f8HGG14FFM1M1qfhrKn7b4m1MHL7cARI5W+1pCP53Q8
2022-02-27T22:09:19.377+02:00	1TLdLKCgYBI+Xj+tUua6HfmoA+5pJpUSYNI7HpZ1jd+kg3py7EFn5N+1vgXabsm
2022-02-27T22:09:19.377+02:00	jA1z3CHh/oznmXybf1GDeB4UBDXHKVUn5U4dnGyVZ4005jRI1iNyz4FuqybJUGIn
2022-02-27T22:09:19.377+02:00	Tyn1YCNXpxzGKtA1rh3gCQ176bUCDUMK1fmXki
2022-02-27T22:09:19.377+02:00	-----END RSA PRIVATE KEY-----
2022-02-27T22:09:20.914+02:00	New instance created: i-021de51ad49da339a
2022-02-27T22:09:21.146+02:00	State Machine created: arn:aws:states:eu-west-1:680505452604:stateMachine:thesis-demo-ec2-state-machine
2022-02-27T22:09:21.223+02:00	Execution created
2022-02-27T22:09:21.224+02:00	END RequestId: 98cd8ff1-d18a-55a4-a4ea-d5640d10acbf
2022-02-27T22:09:21.224+02:00	REPORT RequestId: 98cd8ff1-d18a-55a4-a4ea-d5640d10acbf Duration: 2637.00 ms Billed Duration: 2637 ms Memory

Picture 3. CreateEC2 Logs from AWS CloudWatch.

In the AWS console, Picture 4, it can also be seen that the instance was created properly and received public and private IP addresses correctly and a keypair was attached to it.

Instance summary for i-021de51ad49da339a (thesis-demo-ec2) [Info](#)
Updated less than a minute ago

[Refresh](#) [Connect](#) [Instance state](#) [Actions](#)

Instance ID i-021de51ad49da339a (thesis-demo-ec2)	Public IPv4 address 63.32.106.122 open address	Private IPv4 addresses 10.0.0.107
IPv6 address -	Instance state Running	Public IPv4 DNS ec2-63-32-106-122.eu-west-1.compute.amazonaws.com open address
Hostname type IP name: ip-10-0-0-107.eu-west-1.compute.internal	Private IP DNS name (IPv4 only) ip-10-0-0-107.eu-west-1.compute.internal	Answer private resource DNS name -
Instance type t2.micro	Elastic IP addresses -	VPC ID vpc-0e18bdb91b4d1964d (awsLabVPC)
AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations. Learn more	IAM Role -	Subnet ID subnet-0acf7acf788033d43 (thesis-demo-ec2-subnet)

Picture 4. Running Instance in AWS Console.

In Picture 5, it can be seen that the state machine has also been created and is waiting for the termination date of the instance that was set in the payload of the message.

terminate-thesis-demo-ec2 [Edit state machine](#) [New execution](#) [Stop execution](#)

Details | Execution input | Execution output | Definition

Execution Status Running	Started Feb 27, 2022 10:09:21.188 PM
Execution ARN arn:aws:states:eu-west-1:680505452604:execution:thesis-demo-ec2-state-machine:terminate-thesis-demo-ec2	End Time -

Graph inspector [Data flow simulator](#) [Export](#) [Layout](#)

```

graph TD
    Start((Start)) --> Wait[Wait for termination date]
    Wait --> Lambda[Invoke TerminateEC2 lambda]
    Lambda --> End((End))
      
```

Details		Step input	Step output
Name	Wait for termination date	Type	Wait
Status	In Progress		
Resource	-		

Picture 5. State Machines Execution Waiting for Termination Date.

As shown in Picture 6, to connect to the instance the public IP and an RSA key file are used and the connection is successful. On Windows AMIs made by Amazon, it is required to use the RSA key to decrypt the Administrator password from the AWS Console and download the RDP configuration file for connection to the EC2 instance.

```

C:\Users\westm\Desktop>ssh -i thesis-demo-ec2-keypair.pem ec2-user@ec2-63-32-106-122.eu-west-1.compute.amazonaws.com
Last login: Sun Feb 27 20:21:13 2022 from dsl-tkubng12-54f95e-28.dhcp.inet.fi

  __|  __|_ )
 _| (  /  Amazon Linux 2 AMI
---\___|___|

https://aws.amazon.com/amazon-linux-2/
8 package(s) needed for security, out of 14 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-0-0-107 ~]$

```

Picture 6. SSH Connection to the Instance.

Picture 7 contains logs of a machine that was terminated but it can be seen that the function works correctly by first deleting the instance, the subnet, and keypair, and finally the state machine itself. This function takes a lot longer to run since a subnet cannot be removed until the instance termination process is completed.

▶	Timestamp	Message
		No older events at this moment. Retry
▶	2022-02-27T22:02:22.950+02:00	START RequestId: 31cdbbbe-eced-4e4c-89f5-6626731884ac Version: \$LATEST
▶	2022-02-27T22:02:23.609+02:00	[{'TerminatingInstances': [{'CurrentState': {'Code': 32, 'Name': 'shutting-down'}, 'InstanceId': 'i-06f5646b4756cfb93', 'Pr
▶	2022-02-27T22:02:39.364+02:00	Deleting subnet
▶	2022-02-27T22:02:39.599+02:00	Deleting KeyPair
▶	2022-02-27T22:02:39.774+02:00	{'ResponseMetadata': {'RequestId': '24f4c770-40a4-40da-b8f6-c4186eb17d16', 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-i
▶	2022-02-27T22:02:39.775+02:00	END RequestId: 31cdbbbe-eced-4e4c-89f5-6626731884ac
▶	2022-02-27T22:02:39.775+02:00	REPORT RequestId: 31cdbbbe-eced-4e4c-89f5-6626731884ac Duration: 16824.41 ms Billed Duration: 16825 ms Memory Size: 128 MB

Picture 7. TerminateEC2 Lambda Logs.

5 Conclusion

This thesis aimed to create an automated small-scale lab environment to reduce the maintenance and labor cost requirements of a physical lab environment and to familiarize and learn to work with cloud providers and the tools they offer. Different cloud providers and their differences were also explored, and the conclusion that was reached is that there are not that many differences between the top three most used ones. Amazon Web Services was chosen as the cloud provider used to implement the project mostly because it offered more tools than the others. Also to help select the correct tools for this project, different types of tools and their usage, strengths, and weaknesses were also discussed.

The objective of creating a small-scale automated cloud environment was met, but not with the tools originally in mind and most discussed in the theory segment. A successful deployment of the environment was created using AWS Lambdas, instead of the declarative tools like CloudFormation like was originally planned. With more time and resources it could be possible to build a more reliable and stable environment with fallbacks in case of errors, and checks to make sure that only the wanted resources are existing using declarative tools. In terms of time used, this robust environment is a good first effort that can be improved upon and learned from in future projects and implementations.

The organization that this was made for was happy with the results and will improve the work further and customize it to their own needs. This thesis also offers a good base knowledge on the difference between the tools offered by cloud service providers and their possible use-cases.

The reliability of this thesis might not be that high, as the author is not a seasoned programmer nor had much knowledge of the cloud before making this thesis. With more knowledge of programming and the cloud, it could be possible to fix the issues the author found with the declarative tools in question, such as issues using the CDK.

In the future, it could be worth it to look deeper into declarative tools and how this environment could be deployed using them as they age and gain more documentation and resources to learn from.

References

- [1] J. Kirby, "Cloud Computing vs. Traditional IT Infrastructure | Micro Pro." <https://micropro.com/blog/cloud-computing-vs-traditional-it-infrastructure/> (accessed Jan. 04, 2022).
- [2] "Cloud Computing vs. Traditional IT Infrastructure | Leading Edge." <https://www.leadingedgetech.co.uk/it-services/it-consultancy-services/cloud-computing/how-is-cloud-computing-different-from-traditional-it-infrastructure/> (accessed Jan. 04, 2022).
- [3] "TryHackMe | Cyber Security Training." <https://tryhackme.com/> (accessed Feb. 17, 2022).
- [4] R. Bala, B. Gill, D. Smith, K. Ji, and D. Wright, "Gartner Reprint." <https://www.gartner.com/doc/reprints?id=1-271OE4VR&ct=210802&st=sb> (accessed Jan. 04, 2022).
- [5] S. Carey, "AWS vs Azure vs Google Cloud: What's the best cloud platform for enterprise? | Computerworld." <https://www.computerworld.com/article/3429365/aws-vs-azure-vs-google-whats-the-best-cloud-platform-for-enterprise.html> (accessed Jan. 03, 2022).
- [6] J. Solanki, "Cloud Pricing Comparison 2022: AWS vs Azure vs Google Cloud." <https://www.simform.com/blog/compute-pricing-comparison-aws-azure-googlecloud/> (accessed Jan. 04, 2022).
- [7] P. Yifat, "Azure vs AWS Pricing: Comparing Apples to Apples." <https://cloud.netapp.com/blog/azure-vs-aws-pricing-comparing-apples-to-apples-azure-aws-cvo-blg> (accessed Jan. 04, 2022).
- [8] "AWS Pricing Calculator." <https://calculator.aws/#/createCalculator/EC2> (accessed Jan. 04, 2022).

- [9] "Pricing Calculator | Microsoft Azure." <https://azure.microsoft.com/en-gb/pricing/calculator/?cdn=disable> (accessed Jan. 04, 2022).
- [10] "Google Cloud Pricing Calculator." <https://cloud.google.com/products/calculator/#id=> (accessed Jan. 04, 2022).
- [11] P. Sangode, "Understanding terms - Infrastructure As Code, Orchestration, Provisioning & Configuration Management (Ansible & Terraform, as example)." <https://www.linkedin.com/pulse/understanding-terms-infrastructure-code-management-ansible-sangode> (accessed Jan. 12, 2022).
- [12] S. G. Navdeep, "Top 10 Infrastructure as Code Tools to Boost Your Productivity." <https://www.nexastack.com/en/blog/best-iac-tools> (accessed Jan. 05, 2022).
- [13] Y. Brikman, "Why we use Terraform and not Chef, Puppet, Ansible, Saltstack, or CloudFormation." <https://lsi.vc.ehu.eus/pablogn/docencia/AS/Act7%20Admin.%20centralizada%20infrastructure-as-code,%20Configuration%20Management/Terraform%20Chef%20Puppet%20Ansible%20Salt.pdf> (accessed Jan. 04, 2022).
- [14] T. Cameron, "Mutable vs Immutable Infrastructure Comparison & Benefits | Eplexity." <https://eplexity.com/blog/a-side-by-side-comparison-of-immutable-vs-mutable-infrastructure/> (accessed Jan. 05, 2022).
- [15] A. Dadgar, "Immutable Infrastructure: Benefits, Comparisons & More." <https://www.hashicorp.com/resources/what-is-mutable-vs-immutable-infrastructure> (accessed Jan. 05, 2022).
- [16] "Pricing | Pulumi." <https://www.pulumi.com/pricing/> (accessed Jan. 12, 2022).

- [17] "Terraform vs CloudFormation vs Pulumi vs AWS CDK | Pilotcore."
<https://pilotcoresystems.com/insights/terraform-vs-cloudformation-vs-pulumi-vs-aws-cdk#aws-cdk-overview> (accessed Jan. 12, 2022).
- [18] F. Triboix, "The Definitive Guide to Terraform vs. CloudFormation | Toptal." <https://www.toptal.com/terraform/terraform-vs-cloudformation> (accessed Jan. 12, 2022).
- [19] A. Wittig, "CloudFormation vs Terraform 2021 | clouonaut." <https://clouonaut.io/cloudformation-vs-terraform/> (accessed Jan. 12, 2022).
- [20] B. Hadzhiev, "AWS CDK vs CloudFormation - Comparison | bobbyhadz." <https://bobbyhadz.com/blog/cdk-cloudformation-comparison> (accessed Feb. 22, 2022).
- [21] "What is the AWS CDK? - AWS Cloud Development Kit (CDK) v2." <https://docs.aws.amazon.com/cdk/v2/guide/home.html> (accessed Feb. 22, 2022).
- [22] "16. What are the pros and cons of CDK? - YouTube." <https://www.youtube.com/watch?v=jjyNTNQdW2s> (accessed Feb. 22, 2022).
- [23] "Serverless Computing - AWS Lambda - Amazon Web Services." <https://aws.amazon.com/lambda/> (accessed Feb. 22, 2022).
- [24] "SDKs and Programming Toolkits for AWS." <https://aws.amazon.com/tools/> (accessed Feb. 22, 2022).
- [25] "AWS Step Functions | Serverless Microservice Orchestration | Amazon Web Services." <https://aws.amazon.com/step-functions/?step-functions.sort-by=item.additionalFields.postDateTime&step-functions.sort-order=desc> (accessed Feb. 22, 2022).
- [26] "Amazon EventBridge | Event Bus | Amazon Web Services." <https://aws.amazon.com/eventbridge/> (accessed Feb. 22, 2022).

Instructions for setting up AWS lab environment

This guide assumes that you have basic knowledge in using Amazon Web Services such as IAM, Lambda, EventBridge, SQS, and VPC. This is not meant to be an in-depth guide for AWS and these components and is instead meant to offer the building blocks for setting up an automated AWS lab environment.

1 Setting up Lambdas	33
1.1 StartEC2	33
1.2 StopEC2	33
1.3 CreateEC2	34
1.4 TerminateEC2	38
2 Setting up SQS	41
3 Setting up Amazon EventBridge	43
3.1 Automatic Start Up	43
3.2 Automatic Shut Down	45
3.3 Setting up VPC for Lambda	47
4 Code explanations	51
4.1 StartEC2	51
4.2 StopEC2	53
4.3 CreateEC2	55
4.4 TerminateEC2	62

1 Setting up Lambdas

1.1 StartEC2

Create a Lambda function and set the:

- Runtime as Python 3.9
- Architecture as x86_64
- Permissions required by this lambda are:
 - ec2:DescribeInstances
 - ec2:StartInstances
 - logs:CreateLogGroup
 - logs:CreateLogStream
 - logs:PutLogEvents

Insert the following code and deploy

```
import boto3

ec2 = boto3.resource('ec2')

def lambda_handler(event, context):
    filters = [{
        'Name': 'tag:awsLabAutoStart',
        'Values': ['True'],
    },
    {
        'Name': 'instance-state-name',
        'Values': ['stopped'],
    }]

    instances = ec2.instances.filter(Filters=filters)
    stoppedInstances = [instance.id for instance in instances]
    if len(stoppedInstances) > 0:
        startingUp =
ec2.instances.filter(InstanceIds=stoppedInstances).start()
        print (startingUp)
```

1.2 StopEC2

Create a Lambda function and set the:

- Runtime as Python 3.9

- Architecture as x86_64
- Permissions required by this lambda are:
 - ec2:DescribeInstances
 - ec2:StopInstances
 - logs:CreateLogGroup
 - logs:CreateLogStream
 - logs:PutLogEvents

Insert the following code and deploy

```
import boto3

ec2 = boto3.resource('ec2')

def lambda_handler(event, context):
    filters = [{
        'Name': 'tag:awsLabAutoShut',
        'Values': ['True'],
    },
    {
        'Name': 'instance-state-name',
        'Values': ['running'],
    }]

    instances = ec2.instances.filter(Filters=filters)
    runningInstances = [instance.id for instance in instances]
    if len(runningInstances) > 0:
        shuttingDown =
ec2.instances.filter(InstanceIds=runningInstances).stop()
        print (shuttingDown)
```

1.3 CreateEC2

Create a Lambda function and set the:

- Runtime as Python 3.9
- Architecture as x86_64
- Permissions required by this lambda are:
 - states:DescribeStateMachine
 - states:DescribeExecution
 - states:UpdateStateMachine
 - states:TagResource
 - states:StartExecution
 - states:CreateStateMachine

- ec2:DeleteTags
 - ec2:CreateKeyPair
 - ec2:CreateTags
 - ec2:RunInstances
 - ec2:ModifySubnetAttribute
 - ec2:AssociateSubnetCidrBlock
 - ec2:CreateSubnet
 - ec2:DescribeSubnets
 - sqs:DeleteMessage
 - sqs:ListQueues
 - sqs:GetQueueUrl
 - sqs:ListDeadLetterSourceQueues
 - sqs:ChangeMessageVisibility
 - sqs:ReceiveMessage
 - sqs:GetQueueAttributes
 - sqs:ListQueueTags
 - iam:PassRole
 - logs:CreateLogGroup
 - logs:CreateLogStream
 - logs:PutLogEvents
- Requires a Role made for state machines that allow:
 - lambda:InvokeFunction
 - Needs a VPC to deploy subnets to

Insert the following code and deploy

```
import json
import boto3
import os

ec2 = boto3.resource('ec2')
ec2Client = boto3.client('ec2')
stepfunc = boto3.client('stepfunctions')

def create_subnet(VPC, ec2Name):
    subnetNumber = 0
    while 1:
        try:
```

```

        subnet =
ec2.create_subnet(CidrBlock='10.0.{}.0/24'.format(subnetNumber),
VpcId=VPC)
        print('Subnet is valid:
10.0.{}.0/24'.format(subnetNumber))
        break
    except Exception as e:
        print (e)
        print ('Trying next subnet')
        subnetNumber += 1

TAGS=[
    {
        'Key': 'Name',
        'Value': ec2Name+'-subnet',
    },
    {
        'Key': 'awsLab',
        'Value': 'True',
    }
]

subnet.create_tags(Tags = TAGS)
ec2Client.modify_subnet_attribute( SubnetId = subnet.id ,
MapPublicIpOnLaunch = { 'Value': True } )
return subnet.id

def create_keys(ec2Name):
    keyName = ec2Name+'-keypair'
    key_pair = ec2.create_key_pair(KeyName=keyName)
    keyPairOut = str(key_pair.key_material)
    print(keyPairOut) #DELETE THIS
    return keyPairOut, keyName

def create_tags(ec2Name, CostOrg):
    TAGS=[
        {
            'Key': 'Name',
            'Value': ec2Name,
        },
        {
            'Key': 'awsLab',
            'Value': 'True',
        },
        {
            'Key': 'CostOrg',
            'Value': CostOrg,
        },
        {
            'Key': 'awsLabAutoShut',
            'Value': 'True',
        },
        {
            'Key': 'awsLabAutoStart',
            'Value': 'True',
        },
        {
            'Key': 'awsLabAutoTerminate',
            'Value': 'True',
        },
    ],

```

```

    ]
    return TAGS

def create_state_machine(TerminationDate, ec2Name, ROLEARN):
    definition_set = {
        "StartAt": "Wait for termination date",
        "States": {
            "Wait for termination date": {
                "Type": "Wait",
                "Next": "Invoke TerminateEC2 lambda",
                "Timestamp": TerminationDate+"T00:00:00z"
            },
            "Invoke TerminateEC2 lambda": {
                "Type": "Task",
                "Resource": "arn:aws:states:::lambda:invoke",
                "OutputPath": "$",
                "Parameters": {
                    "FunctionName": "arn:aws:lambda:eu-west-
1:680505452604:function:TerminateEC2:$LATEST",
                    "Payload.$": "$"
                },
                "End": True
            }
        }
    }

    DEFINITION = json.dumps(definition_set)
    TAGS=[
        {
            'key': 'Name',
            'value': ec2Name+'-state-machine',
        },
        {
            'key': 'awsLab',
            'value': 'True',
        }
    ]

    response = stepfunc.create_state_machine(
        name=ec2Name+'-state-machine',
        roleArn=ROLEARN,
        definition=DEFINITION,
        tags=TAGS
    )

    return response['stateMachineArn']

def create_execution(instanceId, stateMachine, ec2Name):
    input_set = {
        "machineid": instanceId,
        "stateMachineArn": stateMachine,
        "ec2Name": ec2Name
    }

    INPUT = json.dumps(input_set)

    execution_response = stepfunc.start_execution(
        name='terminate-'+ec2Name,

```

```

        stateMachineArn=stateMachine,
        input=INPUT
    )

def lambda_handler(event, context):
    data = json.loads(event['Records'][0]['body'])

    AMI = data['AMI']
    INSTANCE_TYPE = data['InstanceType']
    KEY_NAME = "placeholder-key-pair"
    SUBNET = data['Subnet']

    print('InstanceType = ', INSTANCE_TYPE)
    print('AMI = ', AMI)

    if SUBNET == "new":
        SUBNET_ID = create_subnet("vpc-0e18bdb91b4d1964d",
data['ec2Name'])
        KEY_VALUE, KEY_NAME = create_keys(data['ec2Name'])
    else:
        pass

    instance = ec2.create_instances(
        ImageId=AMI,
        InstanceType=INSTANCE_TYPE,
        KeyName=KEY_NAME,
        SubnetId=SUBNET_ID,
        MaxCount=1,
        MinCount=1
    )

    TAGS = create_tags(data['ec2Name'], data['CostOrg'])
    instance[0].create_tags(Tags=TAGS)
    print("New instance created:", instance[0].id)

    roleArn = 'arn:aws:iam::680505452604:role/LabStepFunctionRole'
    stateMachine = create_state_machine(data['TerminationDate'],
data['ec2Name'], roleArn)
    print("State Machine created: " + stateMachine)

    create_execution(instance[0].id, stateMachine, data['ec2Name'])
    print("Execution created")

    print("Public IP is: " + instance[0].public_dns_name)

```

1.4 TerminateEC2

Create a Lambda function and set the:

- Runtime as Python 3.9
- Architecture as x86_64
- Permissions required by this lambda are:
 - states:DeleteStateMachine

- ec2:TerminateInstances
- ec2:DescribeInstances
- ec2:DescribeSubnets
- ec2>DeleteSubnet
- ec2>DeleteKeyPair
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Insert the following code and deploy

```
import boto3

ec2client = boto3.client('ec2')
ec2 = boto3.resource('ec2')
stepfunc = boto3.client('stepfunctions')

def delete_subnet(ec2Name):
    filters = [{
        'Name': 'tag:Name',
        'Values': [ec2Name+'-subnet'],
    }]

    subnets = list(ec2.subnets.filter(Filters=filters))
    subnets[0].delete()

def delete_key_pair(ec2Name):
    ec2client.delete_key_pair(KeyName=ec2Name+'-keypair')

def lambda_handler(event, context):
    filters = [{
        'Name': 'tag:awsLabAutoTerminate',
        'Values': ['True'],
    }]

    machineids = [event['machineid']]
    instances = ec2.instances.filter(Filters=filters)
    terminatingInstance =
ec2.instances.filter(InstanceIds=machineids).terminate()
    print(terminatingInstance)

    instance = ec2.Instance(event['machineid'])
    instance.wait_until_terminated()

    delete_subnet(event['ec2Name'])
    print('Deleting subnet')

    delete_key_pair(event['ec2Name'])
    print("Deleting KeyPair")
```

```
response = stepfunc.delete_state_machine(  
    stateMachineArn=event['stateMachineArn']  
)  
print(response)
```


2 Setting up SQS

Create an SQS with these options:

- Type: Standard
- Access Policy: Basic
 - Define which accounts have rights to access this SQS

The image shows two screenshots of the Amazon SQS console. The top screenshot is the 'Create queue' page. It shows the 'Details' section where the queue type is set to 'Standard' (selected) and 'FIFO' (unselected). The 'Name' field is filled with 'awsLabSQS'. A warning message states: 'You can't change the queue type after you create a queue.' The bottom screenshot is the 'Access policy' page. It shows the 'Choose method' section with 'Basic' (selected) and 'Advanced' (unselected). Under 'Define who can send messages to the queue', 'Only the queue owner' is selected. Under 'Define who can receive messages from the queue', 'Only the queue owner' is also selected. The 'JSON (read-only)' field contains a policy document that grants permissions to the AWS account '688585452604'.

Amazon SQS > Queues > Create queue

Create queue

Details

Type
Choose the queue type for your application or cloud infrastructure.

ⓘ You can't change the queue type after you create a queue.

☒ **Standard** Info
At-least-once delivery, message ordering isn't preserved

- At-least once delivery
- Best-effort ordering

☐ **FIFO** Info
First-in-first-out delivery, message ordering is preserved

- First-in-first-out delivery
- Exactly-once processing

Name
awsLabSQS

A queue name is case-sensitive and can have up to 80 characters. You can use alphanumeric characters, hyphens (-), and underscores (_).

Access policy
Define who can access your queue. Info

Choose method

☒ **Basic**
Use simple criteria to define a basic access policy.

☐ **Advanced**
Use a JSON object to define an advanced access policy.

Define who can send messages to the queue

☒ **Only the queue owner**
Only the owner of the queue can send messages to the queue.

☐ **Only the specified AWS accounts, IAM users and roles**
Only the specified AWS account IDs, IAM users and roles can send messages to the queue.

Define who can receive messages from the queue

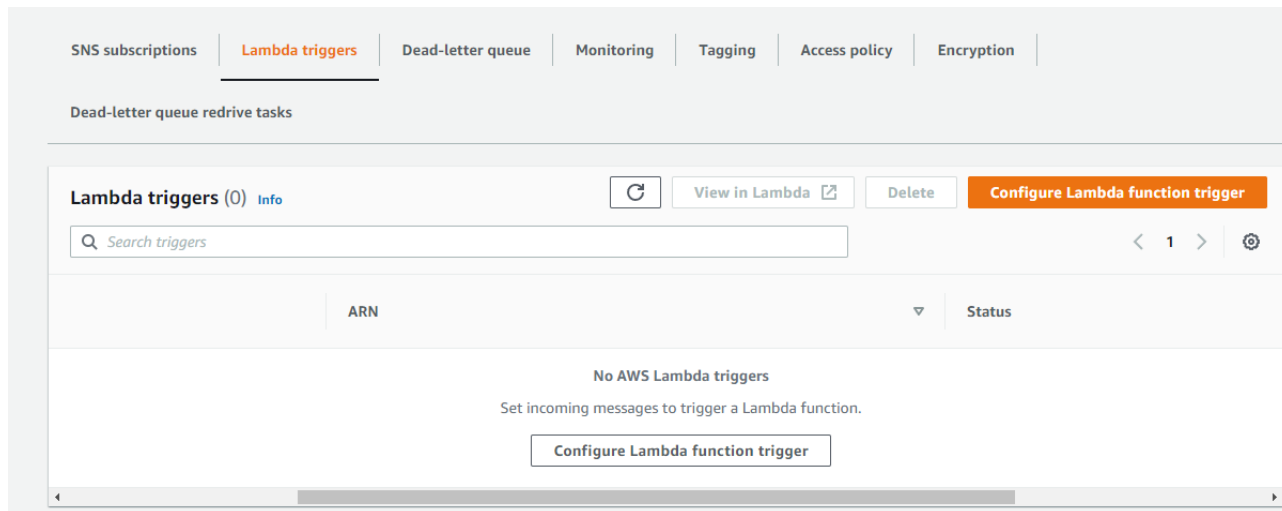
☒ **Only the queue owner**
Only the owner of the queue can receive messages from the queue.

☐ **Only the specified AWS accounts, IAM users and roles**
Only the specified AWS account IDs, IAM users and roles can receive messages from the queue.

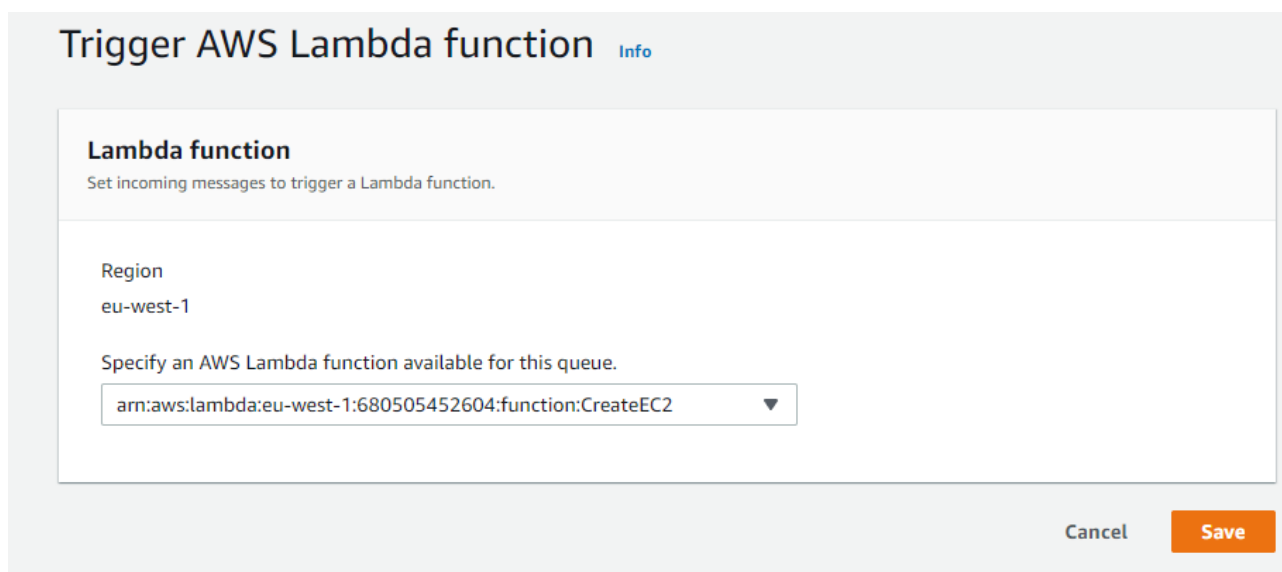
JSON (read-only)

```
{
  "Version": "2008-10-17",
  "Id": "__default_policy_ID",
  "Statement": [
    {
      "Sid": "_owner_statement",
      "Effect": "Allow",
      "Principal": {
        "AWS": "688585452604"
      },
      "Action": [
        "SQS:*"
      ],
      "Resource": "arn:aws:sqs:eu-west-1:688585452604:awsLabSQS"
    }
  ]
}
```

Go to the SQS and select 'configure a Lambda function trigger' on the SQS



Select the correct Lambda function that creates the EC2 instance when receiving a payload



3 Setting up Amazon EventBridge

3.1 Automatic Start Up

Create a new EventBridge rule with these options:

- Define a Cron expression pattern to trigger every day
 - For example: `0 4 * * ? *` triggers every day at 4 AM UTC
 - **Cron in AWS is always defined in UTC!**
- Set the rule to trigger the correct Lambda function

The screenshot shows the 'Define pattern' section of the Amazon EventBridge console. It includes two main options: 'Event pattern' (unselected) and 'Schedule' (selected). Under 'Schedule', there are two sub-options: 'Fixed rate every' (unselected) and 'Cron expression' (selected). The 'Cron expression' field contains the value `0 4 * * ? *`. Below this, a section titled 'Next 10 trigger date(s)' shows a list of dates starting from 'Tue, 22 Feb 2022 04:00:00 GMT' and ending with 'Thu, 03 Mar 2022 04:00:00 GMT'. A dropdown menu for the time zone is set to 'GMT'. At the bottom, there is a link to 'Sample event(s)'.

Define pattern

Build or customize an Event Pattern or set a Schedule to invoke Targets.

☐ Event pattern [Info](#)
Build a pattern to match events

☒ Schedule [Info](#)
Invoke your targets on a schedule

☐ Fixed rate every

☒ Cron expression
CRON expression have six required fields, which are separated by white space. [Learn more about CRON expression.](#) [Enter CRON expression below to see the next 10 trigger date\(s\).](#)

Next 10 trigger date(s)

Tue, 22 Feb 2022 04:00:00 GMT
Wed, 23 Feb 2022 04:00:00 GMT
Thu, 24 Feb 2022 04:00:00 GMT
Fri, 25 Feb 2022 04:00:00 GMT
Sat, 26 Feb 2022 04:00:00 GMT
Sun, 27 Feb 2022 04:00:00 GMT
Mon, 28 Feb 2022 04:00:00 GMT
Tue, 01 Mar 2022 04:00:00 GMT
Wed, 02 Mar 2022 04:00:00 GMT
Thu, 03 Mar 2022 04:00:00 GMT

► **Sample event(s)**

Select targets

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule).

Target

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule).

Remove

Lambda function

▼

Function

StartEC2

▼

► Configure version/alias

► Configure input

► Retry policy and dead-letter queue

Add target

3.2 Automatic Shut Down

Create a new EventBridge rule with these options:

- Define a Cron expression pattern to trigger every day
 - For example: 0 16 * * ? * triggers every day at 4 PM UTC
 - **Cron in AWS is always defined in UTC!**
- Set the rule to trigger the correct Lambda function

The screenshot shows the 'Define pattern' interface in the AWS EventBridge console. It has a title bar 'Define pattern' and a subtitle 'Build or customize an Event Pattern or set a Schedule to invoke Targets.' Below this are two main options: 'Event pattern' (unselected) and 'Schedule' (selected). The 'Schedule' option is further divided into 'Fixed rate every' and 'Cron expression'. The 'Cron expression' option is selected, and the text 'CRON expression have six required fields, which are separated by white space. Learn more about CRON expression. Enter CRON expression below to see the next 10 trigger date(s).' is displayed. The 'Cron expression' input field contains '0 16 * * ? *'. Below this is a 'Next 10 trigger date(s)' section with a dropdown menu set to 'GMT'. The list of dates shows daily triggers at 16:00:00 GMT from February 22 to March 3, 2022. At the bottom, there is a 'Sample event(s)' section with a right-pointing triangle icon.

Define pattern

Build or customize an Event Pattern or set a Schedule to invoke Targets.

☐ Event pattern [Info](#)
Build a pattern to match events

☒ Schedule [Info](#)
Invoke your targets on a schedule

☐ Fixed rate every

☒ Cron expression
CRON expression have six required fields, which are separated by white space. [Learn more about CRON expression.](#) [Enter CRON expression below to see the next 10 trigger date\(s\).](#)

Next 10 trigger date(s)

Tue, 22 Feb 2022 16:00:00 GMT
Wed, 23 Feb 2022 16:00:00 GMT
Thu, 24 Feb 2022 16:00:00 GMT
Fri, 25 Feb 2022 16:00:00 GMT
Sat, 26 Feb 2022 16:00:00 GMT
Sun, 27 Feb 2022 16:00:00 GMT
Mon, 28 Feb 2022 16:00:00 GMT
Tue, 01 Mar 2022 16:00:00 GMT
Wed, 02 Mar 2022 16:00:00 GMT
Thu, 03 Mar 2022 16:00:00 GMT

▶ Sample event(s)

Select targets

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule).

Target

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule).

Remove

Lambda function

Function

StopEC2

► Configure version/alias

► Configure input

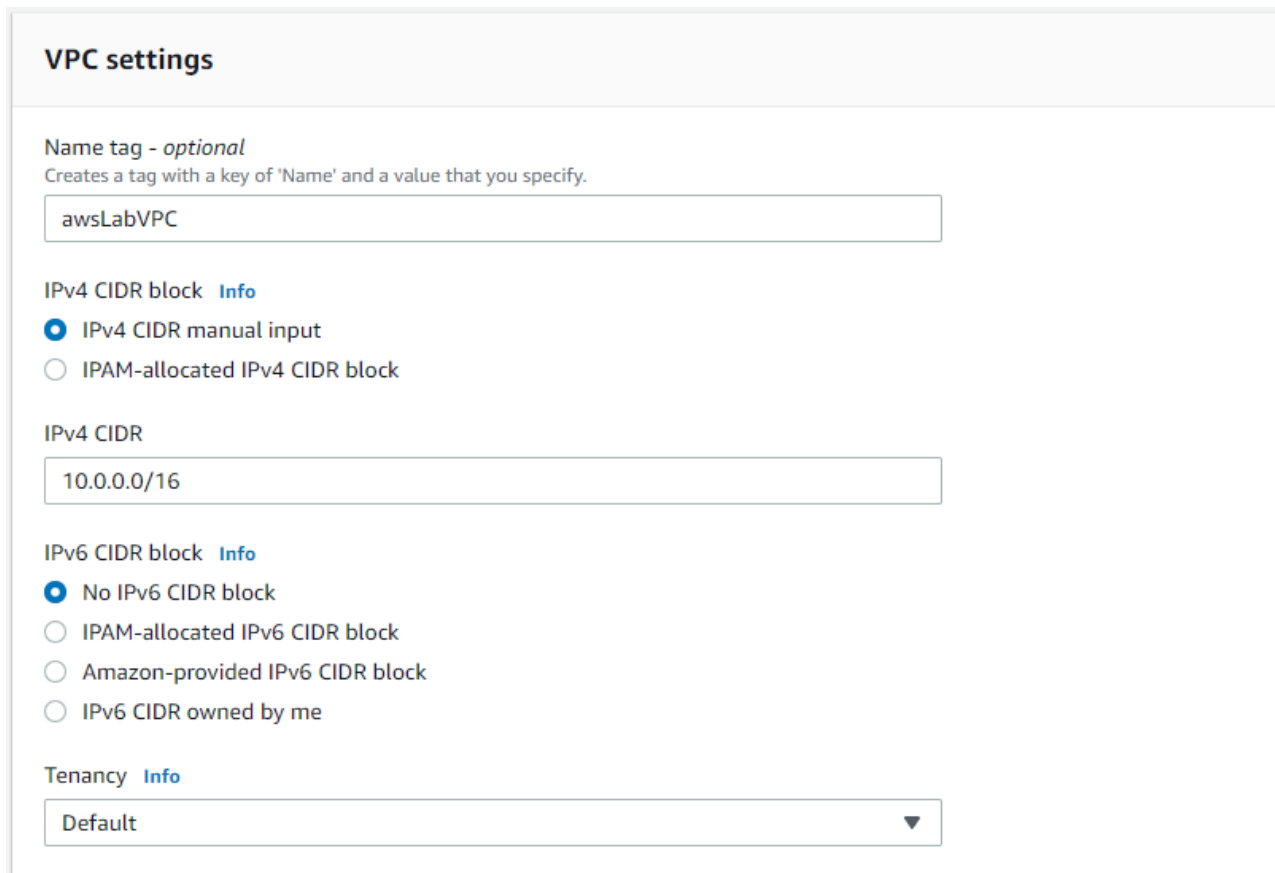
► Retry policy and dead-letter queue

Add target

3.3 Setting up VPC for Lambda

This part is for setting up a VPC that allows connection to the instances via the internet. It is optional but info from VPC used should be inserted into CreateEC2 Lambda

Go to AWS Console and create a new VPC or modify an old one

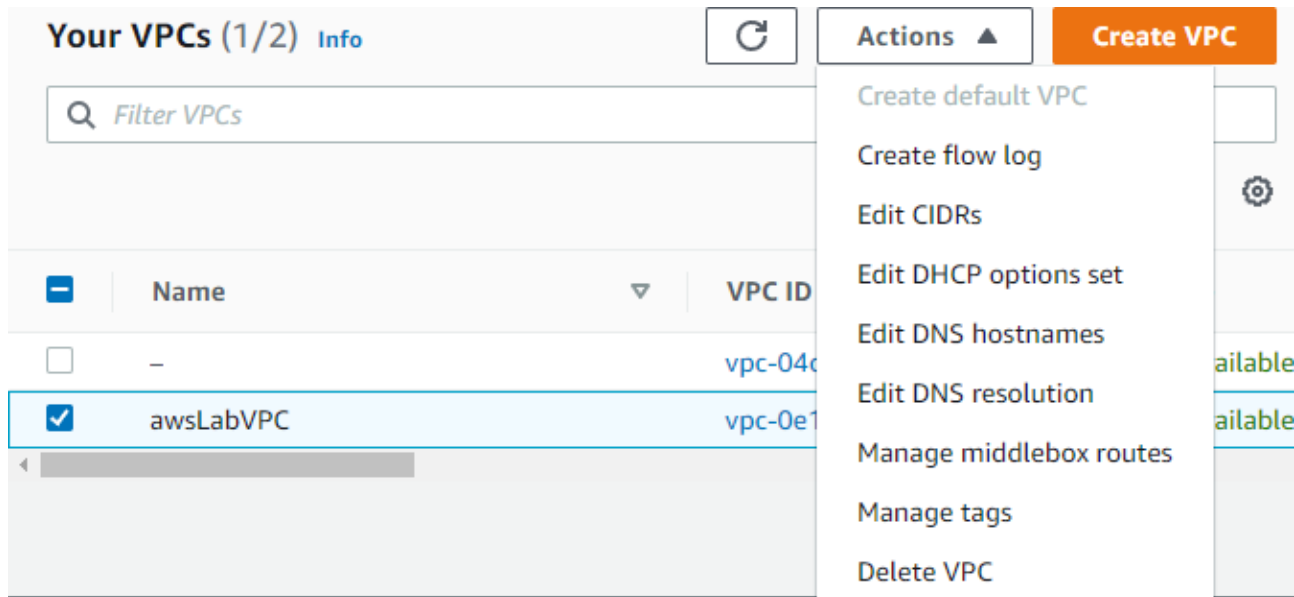


The screenshot displays the 'VPC settings' section of the AWS Management Console. It includes the following configuration options:

- Name tag - optional**: A text input field containing 'awsLabVPC'. Below it, a note states: 'Creates a tag with a key of 'Name' and a value that you specify.'
- IPv4 CIDR block**: A section with an 'Info' link. It contains two radio buttons: 'IPv4 CIDR manual input' (which is selected) and 'IPAM-allocated IPv4 CIDR block'.
- IPv4 CIDR**: A text input field containing '10.0.0.0/16'.
- IPv6 CIDR block**: A section with an 'Info' link. It contains four radio buttons: 'No IPv6 CIDR block' (selected), 'IPAM-allocated IPv6 CIDR block', 'Amazon-provided IPv6 CIDR block', and 'IPv6 CIDR owned by me'.
- Tenancy**: A dropdown menu with an 'Info' link, currently set to 'Default'.

The CIDR from here should be used to replace the one in CreateEC2 Lambda

The VPC should have both DNS hostnames and DNS resolutions set to *Enabled* from the *Actions* menu



The VPC also needs an Internet Gateway, the only required field is the name



After the gateway is created, attach it to the VPC from the *Actions* menu

igw-0b3e9879d50bf1f09 / awsLabGW

Actions ▲

- Attach to VPC
- Detach from VPC
- Manage tags
- Delete

Details [Info](#)

Internet gateway ID	State
igw-0b3e9879d50bf1f09	Detached
VPC ID	Owner
-	680505452604

Next, create a Route Table and select the VPC created

Create route table [Info](#)

A route table specifies how packets are forwarded between the subnets within your VPC, the internet, and your VPN connection.

Route table settings

Name - optional
Create a tag with a key of 'Name' and a value that you specify.

VPC
The VPC to use for this route table.

Edit the routes and add a route to 0.0.0.0/0 from the gateway created earlier

Edit routes

Destination	Target	Status	Propagated
10.0.0.0/16	<input type="text" value="local"/>	Active	No
<input type="text" value="0.0.0.0/0"/>	<input type="text" value="igw-024bf0a8700347cbd"/>	-	No
<input type="button" value="Add route"/>			
<input type="button" value="Remove"/>			

Finally, edit the default security group attached to the VPC or create a new one and add rules for SSH and RDP to all the networks or specify further if needed for security reasons

Edit inbound rules [Info](#)

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info
sgr-09b63c3f3388833fd	All traffic	All	All	Custom <input type="text" value="sg-0d385d87ef8b9e2ab"/>	
sgr-0b0aefb6c9dc3e62f	RDP	TCP	3389	Custom <input type="text" value="0.0.0.0"/>	Allow RDP
sgr-0bb52d83b1d2389b7	SSH	TCP	22	Custom <input type="text" value="0.0.0.0"/>	Allow SSH
<input type="button" value="Add rule"/>					

4 Code explanations

4.1 StartEC2

```
import boto3  
  
ec2 = boto3.resource('ec2')
```

Import required libraries and set the ec2 resource of boto3 to an easier variable name

```
def lambda_handler(event, context):
```

Start the function for *lambda_handler*, this function name can be different, but then it must also be changed in the Lambda runtime settings

```
    filters = [{  
        'Name': 'tag:awsLabAutoStart',  
        'Values': ['True'],  
    },  
    {  
        'Name': 'instance-state-name',  
        'Values': ['stopped'],  
    }  
]
```

Create a filter for later use, in this case, tags are used for filtering the instances that we want to start

```
instances = ec2.instances.filter(Filters=filters)
```

Select the instances with the filter

```
stoppedInstances = [instance.id for instance in instances]
```

Create a list from instance IDs in the *instances* variable

```
if len(stoppedInstances) > 0:
    startingUp =
ec2.instances.filter(InstanceIds=stoppedInstances).start()
    print (startingUp)
```

Iterate over the instances, start them up, and print which instances were started

4.2 StopEC2

```
import boto3  
  
ec2 = boto3.resource('ec2')
```

Import required libraries and set the ec2 resource of boto3 to an easier variable name

```
def lambda_handler(event, context):
```

Start the function for *lambda_handler*, this function name can be different, but then it must also be changed in the Lambda runtime settings

```
    filters = [{  
        'Name': 'tag:awsLabAutoShut',  
        'Values': ['True'],  
    },  
    {  
        'Name': 'instance-state-name',  
        'Values': ['running'],  
    }  
    ]
```

Create a filter for later use, in this case, tags are used for filtering the instances that we want to stop

```
    instances = ec2.instances.filter(Filters=filters)
```

Select the instances with the filter

```
    runningInstances = [instance.id for instance in instances]
```

Create a list from instance IDs in the *instances* variable

```
if len(runningInstances) > 0:
    shuttingDown =
ec2.instances.filter(InstanceIds=runningInstances).stop()
print (shuttingDown)
```

Iterate over the instances, start them up, and print which instances were stopped

4.3 CreateEC2

```
import json
import boto3
import os

ec2 = boto3.resource('ec2')
ec2Client = boto3.client('ec2')
stepfunc = boto3.client('stepfunctions')
```

Import required libraries and set the ec2 and stepfunctions resources of boto3 to easier variable names

```
def create_subnet(VPC, ec2Name):
```

Function *create_subnet*, required *VPC id* and *EC2 name* parameters

```
    subnetNumber = 0
    while 1:
        try:
            subnet =
ec2.create_subnet(CidrBlock='10.0.{}.0/24'.format(subnetNumber),
VpcId=VPC)
            print('Subnet is valid:
10.0.{}.0/24'.format(subnetNumber))
            break
        except Exception as e:
            print (e)
            print ('Trying next subnet')
            subnetNumber += 1
```

This loop tries to assign a valid subnet number that is not reserved. Not optimized well but works until all subnets are taken.

```
TAGS=[
    {
        'Key': 'Name',
        'Value': ec2Name+'-subnet',
    },
    {
```

```

        'Key': 'awsLab',
        'Value': 'True',
    }
]
subnet.create_tags(Tags = TAGS)

```

Create and assign tags to subnet

```

ec2Client.modify_subnet_attribute( SubnetId = subnet.id ,
MapPublicIpOnLaunch = { 'Value': True } )

```

Set the subnet attribute *MapPublicIpOnLaunch* to “True” to map public IP addresses when new instances are created on it

```

return subnet.id

```

Return subnet ID for future use

```

def create_keys(ec2Name):
    keyName = ec2Name+'-keypair'
    key_pair = ec2.create_key_pair(KeyName=keyName)
    keyPairOut = str(key_pair.key_material)
    print(keyPairOut) #DELETE THIS WHEN IT IS NO LONGER NEEDED
    return keyPairOut, keyName

```

Function *create_keys* to create a keypair for the instance. Requires *EC2 Name* as a parameter. Prints the RSA key to logs, but this should be removed when there is a proper place to send the key. Returns RSA key and its name.

```

def create_tags(ec2Name, CostOrg):
    TAGS=[
        {
            'Key': 'Name',
            'Value': ec2Name,

```



```

    },
    {
        'Key': 'awsLab',
        'Value': 'True',
    },
    {
        'Key': 'CostOrg',
        'Value': CostOrg,
    },
    {
        'Key': 'awsLabAutoShut',
        'Value': 'True',
    },
    {
        'Key': 'awsLabAutoStart',
        'Value': 'True',
    },
    {
        'Key': 'awsLabAutoTerminate',
        'Value': 'True',
    },
]
return TAGS

```

Function `create_tags`, creates tags for the EC2 instance, required parameters are *EC2 Name* and *CostOrg*. Tags can be changed to suit needs.

```
def create_state_machine(TerminationDate, ec2Name, ROLEARN):
```

Function `create_state_machine`, required parameters are Termination Date, EC2 Name, and ARN of the role made for state machines

```

definition_set = {
    "StartAt": "Wait for termination date",
    "States": {
        "Wait for termination date": {
            "Type": "Wait",
            "Next": "Invoke TerminateEC2 lambda",
            "Timestamp": TerminationDate+"T00:00:00z"
        },
        "Invoke TerminateEC2 lambda": {
            "Type": "Task",
            "Resource": "arn:aws:states:::lambda:invoke",
            "OutputPath": "$",
            "Parameters": {

```

```

        "FunctionName": "arn:aws:lambda:eu-west-
1:680505452604:function:TerminateEC2:$LATEST",
        "Payload.$": "$"
    },
    "End": True
}
}
}
}
DEFINITION = json.dumps(definition_set)

```

Definitions for the state machine

```

TAGS=[
    {
        'key': 'Name',
        'value': ec2Name+'-state-machine',
    },
    {
        'key': 'awsLab',
        'value': 'True',
    }
]

```

Tags for the state machine

```

response = stepfunc.create_state_machine(
    name=ec2Name+'-state-machine',
    roleArn=ROLEARN,
    definition=DEFINITION,
    tags=TAGS
)
return response['stateMachineArn']

```

Creates the state machine and returns the state machines ARN

```

def create_execution(instanceId, stateMachine, ec2Name):

```

Function *create_execution*, required parameters are *instance ID*, *state machine ARN*, *EC2 Name*.

```
input_set = {  
    "machineid": instanceId,  
    "stateMachineArn": stateMachine,  
    "ec2Name": ec2Name  
}  
INPUT = json.dumps(input_set)
```

Input set for the execution

```
execution_response = stepfunc.start_execution(  
    name='terminate-'+ec2Name,  
    stateMachineArn=stateMachine,  
    input=INPUT  
)
```

Creates an execution order for the state machine

```
def lambda_handler(event, context):
```

Start the function for *lambda_handler*, this function name can be different, but then it must also be changed in the Lambda runtime settings

```
data = json.loads(event['Records'][0]['body'])
```

Assign body from the payload to *data* variable for easy access

```
AMI = data['AMI']  
INSTANCE_TYPE = data['InstanceType']  
KEY_NAME = "placeholder-key-pair"  
SUBNET = data['Subnet']
```

Create variables and assign them placeholder or final values

```
print('InstanceType = ', INSTANCE_TYPE)
print('AMI = ', AMI)
```

Print info to logs

```
if SUBNET == "new":
    SUBNET_ID = create_subnet("vpc-0e18bdb91b4d1964d",
data['ec2Name'])
    KEY_VALUE, KEY_NAME = create_keys(data['ec2Name'])
else:
    pass
```

If a subnet was not defined in the payload create a new subnet to a specific VPC

```
instance = ec2.create_instances(
    ImageId=AMI,
    InstanceType=INSTANCE_TYPE,
    KeyName=KEY_NAME,
    SubnetId=SUBNET_ID,
    MaxCount=1,
    MinCount=1
)
```

Create the EC2 instance

```
TAGS = create_tags(data['ec2Name'], data['CostOrg'])
instance[0].create_tags(Tags=TAGS)
print("New instance created:", instance[0].id)
```

Assign tags to the instance

```
roleArn = 'arn:aws:iam::680505452604:role/LabStepFunctionRole'  
stateMachine = create_state_machine(data['TerminationDate'],  
data['ec2Name'], roleArn)  
print("State Machine created: " + stateMachine)
```

Call the *create_state_machine* function

```
create_execution(instance[0].id, stateMachine, data['ec2Name'])  
print("Execution created")
```

Call the *create_execution* function

```
print("Public IP is: " + instance[0].public_dns_name)
```

Print the public DNS name to logs, should be replaced by a proper destination

4.4 TerminateEC2

```
import boto3

ec2client = boto3.client('ec2')
ec2 = boto3.resource('ec2')
stepfunc = boto3.client('stepfunctions')
```

Import required libraries and set the ec2 and stepfunctions resources of boto3 to easier variable names

```
def delete_subnet(ec2Name):
    filters = [{
        'Name': 'tag:Name',
        'Values': [ec2Name+'-subnet'],
    }]

    subnets = list(ec2.subnets.filter(Filters=filters))
    subnets[0].delete()
```

Function *delete_subnet*, deletes subnets based on a *Name* tag

```
def delete_key_pair(ec2Name):
    ec2client.delete_key_pair(KeyName=ec2Name+'-keypair')
```

Function *delete_key_pair* deletes keypairs based on *Key Pair Name*

```
def lambda_handler(event, context):
```

Start the function for *lambda_handler*, this function name can be different, but then it must also be changed in the Lambda runtime settings

```
    filters = [{
        'Name': 'tag:awsLabAutoTerminate',
```

```
'Values': ['True'],  
}]
```

Create a filter for later use, in this case, tags are used for filtering the instances that we want to terminate

```
machineids = [event['machineid']]
```

Put machine ID from the payload to *machineids* variable

```
instances = ec2.instances.filter(Filters=filters)
```

Select the instances with the filter

```
terminatingInstance =  
ec2.instances.filter(InstanceIds=machineids).terminate()  
print(terminatingInstance)
```

Terminate instances that match the *machineids* variable and print the result

```
instance = ec2.Instance(event['machineid'])  
instance.wait_until_terminated()
```

Wait until the instance is terminated, this is done to successfully delete the subnet

```
delete_subnet(event['ec2Name'])  
print('Deleting subnet')
```

Call function *delete_subnet*

```
delete_key_pair(event['ec2Name'])  
print("Deleting KeyPair")
```

Call function *delete_key_pair*

```
response = stepfunc.delete_state_machine(  
    stateMachineArn=event['stateMachineArn']  
)  
print(response)
```

Delete the state machine which is responsible for calling this function and print the result