

Automated public cloud using aws

**Eric Strong**

MSc. DevOps

Technological University of Dublin Tallaght

X00169645@mytudublin.ie

March 14, 2020

# Table of Contents

[1. Introduction 1](#_Toc35086358)

[2. Proof of Concept Solution 1](#_Toc35086359)

[3. Design Decisions 4](#_Toc35086360)

[4. Proof of Concept Solution Evaluation 7](#_Toc35086361)

[4.1 Blue-Green Deployments 7](#_Toc35086362)

[4.2 Infrastructure as code: IAC CloudFormation 7](#_Toc35086363)

[4.3 Maintenance Tools 7](#_Toc35086364)

[4.4 Security and GDPR 8](#_Toc35086365)

[4.5 Hybrid Cloud 9](#_Toc35086366)

[4.6 Licensing 9](#_Toc35086367)

[4.7 Vendor Lock In 9](#_Toc35086368)

[4.8 Fault Tolerance 9](#_Toc35086369)

[4.9 Scalability 10](#_Toc35086370)

[4.10 Cost 10](#_Toc35086371)

[4.11 Service License Agreement 10](#_Toc35086372)

[5. Conclusion 11](#_Toc35086373)

[6. References 12](#_Toc35086374)

[6.1 Bibliography 12](#_Toc35086375)

[6.2 List of Figures 12](#_Toc35086376)

[6.3 List of Tables 12](#_Toc35086377)

[7. Appendices 12](#_Toc35086378)

[7.1 Cloud Formation YAML Template Snippets used 12](#_Toc35086379)

[7.2 GitHub repository for YAML proof of concept and additional scripts 13](#_Toc35086380)

[7.3 Video link for demonstration 13](#_Toc35086381)

**Abstract**

*The following technical report dives into the world of Amazon Web Services public cloud offerings. The report outlines the technology suggestions to solve issues of inconsistent environments, infrastructure as code and configuration management. The report evaluates and justifies how each technology will solve the related issues. The report then goes on to evaluate the suggested solution against a list of cloud principals.*

***Keywords:*** *Public Cloud, AWS, Infrastructure as Code, Blue Green Deployment, Configuration management, Ansible, CloudFormation*

**Acknowledgements**

I would like to acknowledge Kevin Bayliss, David White and Martin O'Connor of Technological University of Dublin Tallaght for their outstanding work and guidance on the IT automation module.

# Introduction

The aim of this report is to offer a technical consultation on public cloud services. The proof of concept will be aimed at AWS technologies and will address the three main issues below. A customer is currently utilizing on-premise servers to which they are experiencing three main issues:

1. Due to different testing, staging, production environments application issues cause downtime resulting in poor customer experience
2. Due to the lack of available testing environments, releases are missing deadlines resulting in competitors gaining advantages
3. Due to too time spent running manual maintenance tasks of many servers, there is a lack of support for more major projects

The solution will align with the principals of infrastructure as code while offering a solution to the three problems addressed. In addition this report will discuss how the solution meets the requirements and evaluate the solution against a range of public cloud factors.

There are many public cloud vendors to choose from that offer highly available and robust services. The reason for choosing AWS over Microsoft Azure or Google cloud was based on the two main factors. AWS are evaluated as the leader in cloud infrastructure as a service according to the Gartner Magic Quadrant for Infrastructure as a service (Barr 2019). Scoring the highest in measurement, execution, ability and completeness of vision. AWS are the cloud provider of choice for some of the largest and most popular websites online today. These customers include Netflix, Coursera of course Amazon (Amazon 2020).

The public cloud allows for agility, elasticity, cost savings and can allow a company to reach customers globally in minutes. Furthermore, AWS releases a continuous growing collection of web services daily that customers can utilize to help expand their business growth and productivity.

# Proof of Concept Solution

The solution implemented uses a highly available, fault tolerant architecture that allows for scalability, configuration management and rapid deployment. The solution produces a fleet of web servers serving a basic web template that can be easily switched over to alternative deployments with zero disruption. Actions taken to combat the issues mentioned have been factored into the design and implementation of the proof of concept and will be discussed in more detail in a later section. The technologies utilized in the overall design are as follows:

* Virtual Private Network (VPC)
* VPC endpoints
* NAT gateway
* Elastic IP
* Internet gateway
* Routing tables
* Routes
* Multiple Availability Zones
* Public and private subnets
* Security groups
* Application load balancer
* EC2 Autoscaling groups
* Launch configuration
* Cloud9 IDE
* Target groups
* Nat gateway
* Service endpoint
* Ansible
* AWS CodeCommit repos
* CloudFormation template with parameters, mappings, condition

Figure 1 below demonstrates the infrastructure architecture diagram for the proof of concept. This is an overview of the architecture used and shows all of the main components and how they interact. The proof of concept exists as a CloudFormation template. This allows for versioning the overall architecture for future changes. The YAML template also uses parameters to allow the choice of instance type, instance numbers and blue or green deployment. This can be abstracted as a test or production environment. The result of the parameter will determine what target group load balancer will choose to serve. This can be depicted as the customer facing webserver group for the load balancer. While one target group is customer facing, developers can access the server groups in another deployment and make code changes to the website. Using aws CLI tools the load balancer can then be easily switched over with zero downtime for public customers. The proof of concept also caters for configuration management with multiple servers in multiple availability zones. The Cloud9 instance will have access to all servers in both zones. This allows for Ansible playbooks to keep servers in a consistent and desired state this reducing configuration drift. All playbooks are YAML files and can also be checked into version control.

The overall solution caters for flexibility with testing and production infrastructure by choosing instance type and numbers in the parameters. Depending on the parameter picked mappings to allow for better grade servers such as .t2.micro t2.nano and t2.small As this is a proof of concept more expensive instance types can be easily added without effecting the overall logic of the template. The template can be deployed multiple times as the architecture exists in its own VPC. This means the admin could create a test blue green deployment with cheaper servers and lower amount of ec2 servers. Allowing for quick testing of new features and also creation of environments in different regions thus adding to the concept of testing through implementation.

A screenshot of a social media post

Description automatically generated

*Figure 1 - Infrastructure Architecture Diagram*

Figure 2 below demonstrates a more simplistic workflow on how the proof of concept operates from the point of view of deploying the stack with CloudFormation, integrating configuration management with Ansible and managing customer facing deployments with aws cli and blue green deployments. From the diagram users access the URL. The load balancer will handle the requests routing the requests to the target group. From when the admin deploys the infrastructure as code depending on parameters used, the customer facing deployment could be blue or green. The AWS CLI commands allow the admin to switch over to other deployments. Meanwhile admins and developers can access the cloud9 IDE deploy code changes to web apps and mange servers centrally from the ansible controller. Allowing new features be tested in a safe manner. Once changes have been made, they can easily check code into version control and use playbooks to roll out changes to all servers in the fleet in both availability zones. This caters zero disruptions for customers.

A close up of a map

Description automatically generated

*Figure 2 – Solution Workflow Diagram*

# Design Decisions

The design decisions for the services and technologies chosen are outlined below (Table 1).

|  |  |  |
| --- | --- | --- |
| AWS Technology | Reason for choosing | Alternative options |
| VPC:  Security | A centralized container for all AWS resources that provides security (main concern for the customer) and to easily maintain and provide chargeback for cloud costs.  The VPC allows for a scalable infrastructure and includes security by design  The decision was made to choose a new VPC as this would allow for resources to be created and destroyed with no orphaned resources left over if the VPC is created and destroyed together. The risk is higher of missing resources that are not created using the same VPC thus resulting in unwanted billing. | A new VPC does not need to be created every time. A user can choose to deploy the infrastructure into a default VPC or existing VPC.  This can be reflected in the template by easily adding intrinsic functions to look up existing resources such as existing VPC. |
| Multi Availability Zones:  Redundancy,  High availability | This would coincide with a fault tolerant infrastructure and a highly available web server.  A major concern for the company was application downtime resulting in the loss of revenue and poor customer experience. Multi AZ’s allow for rapid disaster recovery and reduction in application downtime. | Single AZ means more susceptible to failure. This is not a recommended option. |
| Elastic Load Balancer:  High availability | Elasticity, high availability, constant monitoring and horizontal scaling without downtime. Also the application load balancer is included in the AWS free tier for 10 rules. | Network and classic load balancers could be used here also but the costs grow more with more enhanced AWS features such as network load balancers.  For the purposes of web servers an application load balancer works at layer 7 of the OSI and is a better option for this type of traffic. |
| Auto scaling groups:  Scalability and elasticity | Mainly for on demand scaling, a self-healing environment and elasticity. | Single EC2 instance can be used but does not cater for load. If the server fails it can cause disruption to the application and to customers.  Containers were also considered for the solution but found it was easier to get set up with EC2 and CloudFormation. |
| Blue-green Deployment:  Identical test and production environments | **This was a solution for issue 1** as it would allow code changes and testing in identical environments to ensure no application downtime.  A load balancer can switch between environments easily without any downtime.  Also allowing for new features to be tested without affecting end users. | An alternative to this set up would be to have two VPC labelled test and production. But this would cause configuration changes to any ansible controllers to take stock of the VPC the wanted versions exist in.  Another alternative for this technology is to use AWS Elastic Beanstalk. This is a simplistic service to get a highly available web deployment up and running.  By simply cloning the deployment and using URL swapping a blue green deployment can easily be created.  However, Elastic Beanstalk is expensive to keep running and this solution caters for cost benefit. |
| CloudFormation:  Infrastructure as code | **This was a solution for issue 2** as it can rapidly provision identical environments to allow for testing without delay.  Each environment will be identical to allow for testing. Infrastructure as code will allow the AWS infrastructure to be deployed using scripting or CLI, thus reducing waiting time on environments and allowing solutions to be proven through implementation. | An alternative to CloudFormation is Terraform. This is also an infrastructure as code automated tool for building infrastructure in the public and private cloud.  This technology does have the added benefit of cloud vendor neutrality. Meaning your template can be used on Azure, AWS or GCP. |
| Ansible:  Configuration Management | **This was a solution for issue 3.** Ansible was chosen for the configuration management tool as it is among one of the most popular tools for configuration management on the market today.  Ansible does not require any agents to be installed. The communication between controller and node happens with the SSH and public private keys.  A component that is readily available when EC2 instances are provisioned. The object is to reduce time spent on repetitive maintenance tasks for a fleet of servers.  Ansible allows operations to easily manage all servers so they conform to desired state by using playbooks to automate the package installations and perform administrative tasks on all servers at once, thus following the principal of idempotence. | Alternative options to Ansible would be AWS OpsWorks. This allows for Chef recipes to be deployed to various servers. This is its own infrastructure along with elastic load balancers also.  The solution did explore the use of AWS OpsWorks and Elastic beanstalk and found that the IAC becomes more complicated to create, using IAC thus more open bugs and issues for failure. Plus with more advanced features comes additional costs. |

Table 1 - Design Decisions

# Proof of Concept Solution Evaluation

## Blue-Green Deployments

The decision was made to use blue green a deployment as an admin can easily provision identical environments. Both deployments contain the exact same servers reducing the risk of different environments, which allows for testing to be carried out with ease. The autoscaling groups are deployed into target groups. These groups can then associated with the load balancer and can easily be alternated for testing and production deployments The advantage of this ensures one environment for production and another for testing new features such as A/B testing. This would be highly beneficial this scenario as downtime is reduced, environments are unified and the customer will never report an unsatisfactory experience.

The advantage of same environments is handled by the IAC template and ensures the environments are exact replicates of each other. Webpage differences can be controlled through version control, not effecting the underlying servers. The elastic load balancer allows environments to swap environment target groups by using CLI commands. Parameters have also been used in the CloudFormation script to allow the user to choose the main environment to be deployed. The added advantage of using autoscaling groups, multi-AZ’s and elastic load balancers also reduces the risk of any downtime.

## Infrastructure as code: IAC CloudFormation

A decision was made to use AWS CloudFormation as the infrastructure as code (IAC) tool for this solution as this is tool native to AWS. This service allows rapid deployments of infrastructure in a matter of minutes, which results in having newly deployed applications available at a moment’s notice. This also allows for rapid testing of infrastructure and quick destruction. Project developments and proof of concepts can be proven quickly instead of going over time consuming hypothesis.

The admin can simply upload the YAML file to an s3 bucket and version it, or use the AWS CLI using parameters to customise and deploy this highly scalable and highly available proof of concept. Once the concept has been proven, it can be deleted reducing further costs allowing the company to stay ahead of the competition by focusing on the product and not the resources. This results in having environments at the ready at a moment’s notice.

With CloudFormation, if a resource fails to create a rollback strategy will remove any resources created. Changes follow the principals of idempotency resulting in changes or updates taking effect on resources effectively. Tags can also be utilized to identify and easily search for any resources created for chargeback and billing estimations. The solution is written in YAML format which means it can be version controlled for testing, manipulation and improving.

## Maintenance Tools

The decision to use Ansible over chef and puppet is due to the fact that Ansible is agentless. Ansible uses SSH to communicate with all slave nodes. The proof of concept solves this problem by utilising AWS Cloud9 IDE environment as a bastion to connect to all instances in the within the private subnets in both availability zones. A private key is all that is needed on the bastion host. This is uploaded when the infrastructure deployment is ready. As this is a cloud IDE, the admin can simple copy and paste the key into the IDE. There are more secure key stores AWS off to do this but costs are associated with AWS Secrets Manager.

A custom bash script is then used to install and set up the bastion host to connect to all EC2 instances such as public private key exchange, Ansible and Boto (AWS SDK) packages. An automated solution for generating a dynamic inventory (RedHat 2020) allows the bastion to identify all EC2 instances within the VPC. This feeds into the advantage of automation and reduction of maintenance tasks for operations teams.

A custom Ansible playbook that creates a user, sets up a webserver and updates and installs Linux packages is executed on all targeted EC2 instances. Idempotency principals are followed within the configuration management tool. Playbooks are checked in and out of version control to ensure the latest changes are pushed out to all servers. This will prevent any server creep and will always keep the servers at a desired state. This solution is completely automated so there is no need for operations to waste time performing repetitive maintenance tasks on all servers. As playbooks are written in YAML, small incremental changes can be easily tracked in version control and pushed out to all servers from the Cloud9 ansible controller.

## Security and GDPR

Security is a major concern for public cloud. However for AWS security remains the highest priority by using security by design. There are many security mechanisms already in place from the basic setup of a subscription. Features such as Network access control lists, security groups, multifactor authentication and public private keys are built in by default. A huge component to AWS services is the Identity and access management and federation. This is the central control mechanism for resources and utilises role based access control based on users and groups. AWS also offer a security advisor service to ensure the system follows security best standards and offers improvements on how to harden the AWS infrastructure.

AWS are also GDPR compliant meaning all services adhere to a high standard, setting data protection standards required for data processors. Among GDPR AWS demonstrate compliance with other rigorous international standards such as ISO 27001 for technical measures, [ISO 27017](https://aws.amazon.com/compliance/iso-27017-faqs/) for cloud security, [ISO 27018](https://aws.amazon.com/compliance/iso-27018-faqs/) for cloud privacy, SOC 1, SOC 2 and SOC 3, PCI DSS Level 1, and EU-specific certifications such as BSI’s [Common Cloud Computing Controls Catalogue](https://aws.amazon.com/compliance/bsi-c5/) (C5) (Woolf 2018).

AWS not only enforce GDPR but also gives the option for additional security measures to align with a particular company policy. Additional measures include data encryption and hardening of processing systems to ensure confidentiality, integrity, resilience and availability of data processing systems. The AWS published GDPR whitepaper (Amazon 2019) details how to ensure GDPR concepts are followed with relation to AWS services including topics discussed such as monitoring, key management and data access. AWS also offer a GDPR-compliant data processing addendum (DPA). Which enables customers to comply with GDPR SLA’s and contractual obligations.

## Hybrid Cloud

The proof of concept can be adapted for a hybrid cloud solution. AWS offer many services that allow for data migrations from private cloud to the public cloud such as AWS storage gateway. Storage gateway allows for the migration of on-premise data to AWS infrastructure. Technologies such as AWS Snowball would allow for petabytes of on premise data to be migrated with the use of specialised encryption equipment. Another alternative solution to hybrid cloud would be to use cloud bursting. This would allow existing on-premise hardware to serve web pages and when there is a spike in traffic such as black Friday or cyber Monday scale AWS can be used to expand the compute power of the existing infrastructure to meet demand. When traffic has returned to regular usage resources can be released. AWS also offer services such as Glue and Direct Connect. This is a high bandwidth VPN solution that allows AWS infrastructure to have a high speed connection directly to an on-premise data stores or databases.

## Licensing

In the context of licencing AWS also offers bring your own license (BYOL) for database and server technologies. If the customer already has a purchased license for Oracle there are options use that license in the cloud. This option also applies for compute instances. Customers can utilise BYOL for Windows and RedHat systems too. AWS has made it easy to apply licenses in the cloud. The solution can be modified to use Windows instances. This solution doesn’t include a database but AWS also offers multi AZ databases such as Oracle, Aurora, RDS and MySQL.

## Vendor Lock In

It is worth noting that the entire proof of concept has been built using AWS technologies. Therefore there is a strong vendor lock in with AWS. Technologies such as Terraform can be used to create IAC templates that are vendor neutral. This option was not explored in this proof of concept as AWS offered some strong advantages as the leader in public cloud offerings. Other cloud vendors would also offer similar cloud technologies and a terraform script could be created to handle the builds in multiple clouds. Moving this proof of concept to alternative public cloud providers such as Azure would not be entirely possible.

## Fault Tolerance

The proof of concept has addressed the concept of fault tolerance. By design the solution deploys ec2 instances into multiple availability zones. This means if one AZ site goes down there is another readily available. Also, the blue green deployment aspect handles the load from one AZ to another. Currently the target group points the load balancer to a specific availability zone. This is can be switched overusing AWS CLI dynamically generated by the output of the cloud formation stack. Simple copy and paste the commands into the bastion cloud9 host. The load balancer will point to a different AZ in the event of an outage. The template also allows for admin to choose what subnet the bastion host exists in to cater for fault tolerance. In the event of ec2 instances failing there are CPU alarms also set up to cater for failing instances. A min and max ec2 count have been set in the event of a single failure where multiple ec2 instances will be automatically provisioned in the event of a failure.

## Scalability

The proof of concept has addressed the concept of scalability by using an application load balancer and pointing to specific autoscaling groups in each availability zone. The current set up allows the load balancer to point to any one availability zone at once. In the event of failure, the load balancer will easily switch over to alternative availability zone. For future work a health listener and lambda function could be utilized to handle this in the event of a detected AZ failure to reduce the effect of downtime. The solution is highly scalable as autoscaling groups are set to horizontally scale in the event of CPU alerts. This can be easily customized in the CloudFormation template to allow for certain thresholds to be hit in order to act appropriately. For the demo, these alerts have been set to auto scale if CPU is greater than 90% for more that 10 minutes. The minimum ec2 scalability is set at the creation of the CloudFormation stack creation as a parameter to control costs. If the CPU drops below 70% for more than 10 minutes, the ec2 instances we be de-provisioned. This is following the principal of elasticity to allow for scalable growth by demand

## Cost

For the solution proposed a cost calculation was carried out to see how much this proof of concept would cost to run per month and yearly. The costs calculated were aimed at a production grade infrastructure in the eu-west-1region (Ireland) that would include the following details:

|  |  |  |  |
| --- | --- | --- | --- |
| Service | Monthly | First 12 months total | Config summary |
| Amazon EC2 | €87.36 | €1048.30 | Operating system (Linux), Quantity (6), Storage for each EC2 instance (General Purpose SSD (gp2)), Storage amount (30 GB) |
| Elastic Load Balancing | €17.23 | €206.79 | Number of Application Load Balancers (1) |
| Amazon Virtual Private Cloud (VPC) | €35.76 | €429.10 | Number of NAT Gateways (1) |
| Amazon Elastic IP | €0 | €0 | Number of EC2 instances (1), Number of EIPs per instance (1) |
| Subtotals | €140.35 | €1684.19 |  |

Table 2 - Estimated Costs For Solution

## Service License Agreement

The proof of concept has addressed the issue of uptime by design of the highly available architecture. It is important that the solution produces a solution that reduces downtime. AWS ec2 service license agreements therefore offer a very robust compute SLA. AWS report that less than 99.99% but equal to or greater than 99.0% (Amazon 2020). Having calculated this using uptime calculations this means EC2 service, with a catastrophic failure can only be down for up to following timeframes:

* **Daily:** 17s to **Daily:** 14m 24s
* **Weekly:** 2m 0s to **Weekly:** 1h 40m 48s
* **Monthly:** 8m 45s to **Monthly:** 7h 18m 17s
* **Yearly:** 1h 45m 11s to **Yearly:** 3d 15h 39m 29s

The following timeframes can give an idea of downtimes they can expect at the worst case scenario. This SLA can be built into customer agreements in order to guarantee a highly available service.

# Conclusion

In conclusion this report looked at solving the three main issues requested by the customer by using AWS public cloud services. The report details all the AWS technologies used, reason for choosing them, alternative options and an evaluation of each technology. The proof of concept looked at how AWS services could solve issues relating to different environments by introducing a blue green deployment. The proof of concept also suggested a solution having environments provisioned quickly by utilizing a CloudFormation template to provision environments to quickly to run tests for product releases. Finally, we looked at how the proof of concept could solve the issue of configuration management by using Ansible to effortlessly control a fleet of servers in different availability zones and keeping them at a desired state. The report then gave an overview of how this solution could be modified for a hybrid cloud and offered a monthly and yearly costing for a production environment.

For additional improvements the solution could benefit further with the following recommendations. The proof of concept currently does not house a database. AWS offer many database technologies with the option to bring your own license. A database could be deployed between multiple availability zones to ensure high availability and to evolve the web application into a more sophisticated solution. Another suggestion for improvement is to make the overall solution cloud agnostic. Currently CloudFormation templates build only in AWS. Having the solution created in a Terraform module this would allow for a more hybrid cloud approach as the Terraform module could create resources both in AWS and for other public cloud vendors such as Azure or Google cloud platform and for private cloud such as VMWare. Another suggestion is to explore the concept of AWS Elastic beanstalk to enhance the web application. This AWS service offers highly scalable solutions that can also be clone and turned into a blue green deployment. Using technologies such as AWS OpsWorks and Chef to manage configurations on servers. Elastic Beanstalk also offers logging and monitoring capabilities among many other for the instances.

# References

## Bibliography

Amazon. (2020). *Amazon Compute Service Level Agreement* [online]., *Amazon Web Services, Inc.* [online]. Available at: https://aws.amazon.com/compute/sla/ (Accessed 13 March 2020).

Amazon. (2020). *Case Studies* [online]., *Amazon Web Services, Inc.* [online]. Available at: https://aws.amazon.com/solutions/case-studies/ (Accessed 11 March 2020).

Amazon. (2019). Navigating GDPR Compliance on AWS. In: *Amazon*. p.32. Available at: https://d1.awsstatic.com/whitepapers/compliance/GDPR\_Compliance\_on\_AWS.pdf.

Barr, J. (2019). *AWS Named as a Leader in Gartner’s Infrastructure as a Service (IaaS) Magic Quadrant for the 9th Consecutive Year* [online]., *Amazon Web Services, Inc.* [online]. Available at: https://aws.amazon.com/blogs/aws/aws-named-as-a-leader-in-gartners-infrastructure-as-a-service-iaas-magic-quadrant-for-the-9th-consecutiveyear/ (Accessed 11 March 2020).

*CloudFormation template for launch configuration*. Available at: https://s3-us-west-2.amazonaws.com/CloudFormation-templates-us-west-2/VPC\_AutoScaling\_and\_ElasticLoadBalancer.template (Accessed 13 March 2020).

RedHat. (2020). *Working with dynamic inventory — Ansible Documentation* [online]., *Working with dynamic inventory* [online]. Available at: https://docs.ansible.com/ansible/latest/user\_guide/intro\_dynamic\_inventory.html#inventory-script-example-aws-ec2 (Accessed 13 March 2020).

Woolf, C. (2018). *All AWS Services GDPR ready* [online]., *Amazon Web Services* [online]. Available at: https://aws.amazon.com/blogs/security/all-aws-services-gdpr-ready/ (Accessed 12 March 2020).

## List of Figures

[*Figure 1 - Infrastructure Architecture Diagram* 3](#_Toc35085552)

[*Figure 2 – Solution Workflow Diagram* 3](#_Toc35085553)

## List of Tables

[Table 1 - Design Decisions 6](#_Toc35085557)

[Table 2 - Estimated Costs For Solution 10](#_Toc35085558)

# Appendices

## Cloud Formation YAML Template Snippets used

A snippet from the AWS CloudFormation launch configuration was used in this solution. The launch configuration metadata template was used in order to get the stack working (CloudFormation template for launch configuration n.d.).

## GitHub repository for YAML proof of concept and additional scripts

<https://github.com/ericstrongDevOps/ITAUTOCA1.git>

## Video link for demonstration

<https://tudublin-my.sharepoint.com/:v:/g/personal/x00169645_mytudublin_ie/EbWXU6cF1HFDptOxu0e-fGIBN2IAEMgUNuMAifxS488bAA?e=2sTrFp>