Hand-in assignment 1 - AWS basics

Maciej Owczarek 579390, Marcin Grawiński 579370

## Meeting the Learning Objectives

## Provision a server and deploy a webapp using userdata

Our solution uses an EC2 Launch Template with a robust user-data script that, on instance boot, clones the CloudShirt repository, updates its configuration (injecting parameters like the RDS endpoint), installs .NET 6.0, restores dependencies, builds, and then runs the app on port 80. This automation is critical for ensuring that every newly-provisioned EC2 instance (launched as part of our Auto Scaling Group) deploys the CloudShirt application without any manual intervention (full details are in the Instances.yml file).

## Provision a private subnet with proper routing and NAT

To enable secure outbound connectivity from private subnets in our solution, we set up a NAT Gateway alongside appropriate routing tables, as defined in the NAT.yml file. An Elastic IP is reserved for the NAT Gateway, and a route is added in the private route table to direct outbound traffic from the private subnets through the NAT Gateway. This setup is also essential for deploying our monitoring solution, as detailed in REQ-05.

Additionally, the overall networking (including the creation of the VPC, public subnets, and private subnets) is defined in the networking.yml file. This file has been adapted from the Cloud Automation Concept lectures.

## Provision a database in the cloud

We provisioned the database for the CloudShirt solution using Amazon RDS as defined in the SSMS.yml file. Key aspects include:

* A dedicated security group (MyDBSecurityGroup) permits SQL Server access internally (port 1433 within 10.0.0.0/16).
* The RDS instance (MyDB) is set up as a non-public, managed SQL Server Express database (db.t3.small) with allocated storage and defined credentials.

This configuration was based on the hints provided in the task description, and it automates and secures the database deployment for the CloudShirt solution.

## Provision an auto scaling group with proper policies to scale out servers

Our solution utilizes an Auto Scaling Group (ASG) defined in the Instances.yml file to deploy EC2 instances across multiple Availability Zones. The default configuration now sets the minimum capacity at 1 instance and the maximum at 2 during non-peak traffic hours, along with integrating an Application Load Balancer for balanced incoming traffic.

In addition to target-tracking scaling policies (ScaleOutPolicy and ScaleInPolicy) that adjust capacity based on load, scheduled actions are configured to manage predictable peak periods (as detailed in REQ-02). This configuration ensures that the infrastructure dynamically scales during traffic spikes while optimizing performance and costs during lower demand periods.

## Provision a load balancer to divide load over multiple servers

We used an Application Load Balancer (ALB) defined in the Instances.yml file to distribute incoming traffic evenly across multiple EC2 instances. The ALB is configured as internet-facing and is associated with a dedicated security group, ensuring proper access controls. It spans multiple subnets to provide high availability and fault tolerance, and is linked to a target group that monitors the health of registered servers. This setup guarantees that user requests are efficiently balanced among the available instances, improving application performance and resilience.

## Provision a S3 bucker and deploy a static website using AWS CLI

We provisioned an S3 bucket to host a static website using two key files. First, the bucket.yml CloudFormation template creates the S3 bucket with a naming convention based on provided parameters. Then, the bucketscript.ps1 PowerShell script automates the site deployment by uploading index.html, configuring website settings (via website.json), setting policies (via policy.json), and outputting the site URL. This process ensures that the static website is reliably hosted on S3 using AWS CLI.

## Provision a file share in the cloud

We configured a file share using Amazon EFS, as defined in the efs.yml file. This template provisions an EFS file system along with its mount targets in the designated subnets, making the file system's DNS name available (exported as MyEFS-DNS). In the userdata script, this DNS name is referenced to mount the EFS on each EC2 instance during boot, providing shared and persistent storage for the application.

## Deploy a monitoring solution

For this requirement, we establish a dedicated monitoring instance that forms the core of our monitoring infrastructure. This instance hosts the Elastic Stack - Elasticsearch, Logstash, and Kibana (ELK) alongside FileBeat. It is automatically launched and deployed within a private subnet for enhanced security, keeping monitoring traffic isolated from public access. The instance is provisioned with a robust EC2 type (t2.large) and protected by a custom security group that permits only necessary traffic. To ensure secure, remote access, we utilize AWS Session Manager port forwarding, which allows authorized users to view the monitoring dashboard through a regular browser on their local machine. The necessary configuration for this secure access is detailed in our ssm.sh file.

## Deploy a serverless application

In our solution, we deploy a serverless application using AWS Lambda as defined in the bucket.yml file. The Lambda function *ProcessTextLambda* is implemented in Python and is triggered by S3 events, specifically when a .txt file is uploaded. Its main job is to process the contents of the text file and generate an HTML file (index.html) that is then made available through the static website hosted on the S3 bucket. This approach uses event-driven processing, eliminating the need for manual server management.

## Meeting the Requirements

1. The CloudShirt .NET solution is high available across multiple AZ's using one url

This is accomplished by deploying an Application Load Balancer (ALB) that spans subnets in different AZs. The ALB distributes incoming traffic evenly across the healthy instances, ensuring that if one zone experiences issues, the solution remains accessible via the same endpoint. This multi-AZ deployment strategy, guarantees continuous uptime and consistent user experience with a one URL.

1. The CloudShirt .NET solution can scale out during spike traffic hours between 6pm and 8pm Eastern timezone

To meet REQ-02, we configured scheduled actions on our Auto Scaling Group to automatically update its capacity during peak traffic hours (6pm–8pm Eastern). Under normal conditions, the ASG operates with a minimum of 1 and a maximum of 2 instances. During peak periods, these actions increase the minimum, desired, and maximum instance counts (for example, to 2, 4, and 6, respectively) to handle the increased load, then revert back once the spike ends.

1. Elastic File System (EFS) is used to store public webserver logfiles on a daily basis.

To meet REQ-03, we provisioned an Elastic File System (EFS) using the efs.yml file and configured its mount targets in the appropriate subnets. In our EC2 Launch Template’s userdata script, we added commands to mount the EFS (using its DNS name exported as MyEFS-DNS) on each new instance, so that all instances have a shared directory where public webserver log files are stored. Additionally, we adjusted the log configuration to redirect log file output to this mounted directory, ensuring that the monitoring instance can later access and analyze these logs as required.

1. The data tier of the CloudShirt .NET solution is based on Amazon RDS.

The CloudShirt .NET solution's data tier relies on Amazon RDS as defined in the SSMS file. A dedicated security group restricts access to port 1433 within the VPC, and a DB Subnet Group ensures the RDS instance is deployed in the proper private subnets. Additionally, our EC2 user-data script for the CloudShirt application updates configuration files (such as appsettings.json) by injecting the RDS endpoint, ensuring the application connects with the managed SQL Server.

1. As a monitoring solution the Elastic (ELK) Stack v8.x is provisioned.

On the monitoring instance, we deploy the Elastic (ELK) Stack v8.x to manage and visualize log data. In this setup, Elasticsearch stores and indexes log data, Logstash processes and transforms incoming log events, and Kibana provides an interactive dashboard for real-time analysis. The installation and configuration of these ELK components are automated through a user-data script. Additionally, to ensure the monitoring solution works flawlessly, the instance size was set to at least t2.large to provide sufficient performance and resources for the ELK Stack.

1. Logs are made visible on Elastic Stack using FileBeat

To fulfill this requirement, FileBeat is installed and configured to collect log files from a designated directory on our EC2 instances. FileBeat then forwards these logs to Logstash for processing, with the processed data stored in Elasticsearch and made available through the Kibana dashboard. This setup ensures that logs are seamlessly integrated into the Elastic Stack and can be easily visualized for monitoring and analysis.

1. RDS export of the order table to a S3 bucket using a AWS-CLI script and the bcp utility, so external Athena partner can analyse it.

Within the Instances.yml file, the user-data script is configured to set up an export process for the order table. First, it installs mssql-tools18 (with the necessary repository and package updates) to enable the bcp utility, and then creates a dedicated directory ("athenafolder") for storing the export file. The script generates a shell script (upload\_to\_s3.sh) that uses bcp to export data from the "orders" table into a text file and then uploads this file to the specified S3 bucket using AWS CLI commands, while logging the operation. Finally, a systemd service and timer are configured to run the upload script every hour, ensuring that the exported data is updated regularly for external Athena analysis.

## Most Important Choices

One of our most important architectural choices was to deploy elastic stack (ELK) in a private subnet. As the monitoring dashboard may display sensitive information (metrics, logs), we decided that limiting the access is crucial, so that only internal users might have access to it. By keeping monitoring instance isolated from external, unwanted access, we align with WAF security pillar, ensuring that only authorized admins can view and interact with monitoring data.

To access the dashboard from our private subnet, we evaluated several options including a bastion host and VPN. At the end, we decided to choose AWS Session Manager port forwarding because it provides secure access to the Kibana dashboard via a web browser on localmachine without the limitations of a bastion hosts and it is also a convenient and simple solution.

We implemented separate security groups for each and different components in our solution. Each security group permits only the necessary traffic, minimizing exposure. By doing so, we also aling with best practises for building cloud architecture.

We automated the deployment and configuration of our entire solution using CloudFormation templates via the AWS CLI, so there's no need to use the AWS Console at all. By relying entirely on Infrastructure as Code, every resource -from EC2 instances and databases to load balancers and monitoring tools - is automatically provisioned. This approach aligns with best cloud practices, particularly - operational excellence by ensuring consistency, repeatability, and ease of management.

We considered a few options for forwarding logs from our web server instances to the ELK stack. One approach was to store logs on each instance in an S3 bucket, while another was to install FileBeat on every instance to forward logs directly to Logstash. In the end, we decided that using a shared Elastic File System (EFS) was the most effective solution. Web server instances write their logs to the shared EFS, and FileBeat, set up on our dedicated monitoring instance, reads from this central location and sends the logs to Logstash. This approach simplifies setup and management, also ensuring that all logs are easily accessible by our monitoring solution.

We considered two approaches for keeping the index.html file in our S3 bucket up to date. While one option was to run a continuously operating EC2 instance, we ultimately chose a serverless solution using AWS Lambda. In our design, whenever a new object is created in the bucket, the Lambda function is triggered to generate and upload the updated index.html file. This approach minimizes operational complexity, reduces costs, and scales automatically with demand corresponding to cloud best practices.

How to Roll Out the Solution

To rollout the solution first download all files or clone GitHub repository (<https://github.com/Grawikos/CloudAutomation.git>). AWS CLI has to be installed and set up with default region and credentials. In the folder with all files execute

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *create.ps1 [-userid “<account user id>”, -bucketname “<name>”]* | *deploy.sh [-userid “<account user id>”, -bucketname “<name>”]* |

*<account user id>* - optional, in not provided, found by command  
*<name>* – optional, name used to create a bucket. It will be concatenated with userid to decrease chance of name collision, if not provided “athena-data-bucket”

After all stacks are created, the link to the website will be outputed to the terminal.

To deploy static website on the S3 Bucket go to folder with bucketscript.ps1 and website.json and execute

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *bucketscript.ps1 [-userid “<account user id>”, -bucketname “<name>”, region “<region>”]* | *bucketscript.sh [-userid “<account user id>”, -bucketname “<name>” , region “<region>”]* |

*<region>* - region of the user, if not provided, us-east-1

The link to the website will be outputed to the terminal

To connect to the ElasticStack and see logs of the application as admin, execute

|  |  |
| --- | --- |
| Windows | MacOS/Linux |
| *connect\_admins.ps1 [-port “<local Port Number>”]* | *connect\_admins.sh [-port “<local Port Number>”]* |

*<local Port Number>* - any open port on the host, if not provided – 8080.

The link to the website will be outputed to the terminal

## Recommendations

While our application aligns with best practices by leveraging Infrastructure as Code, automated scaling, and multi-AZ deployments, there are several improvements worth considering to make our solution even more reliable and cloud native.

### **Avoiding Single Points of Failure**

Instead of depending on a single RDS instance, a multi-AZ configuration with regular backups and replicas should be considered to ensure high availability and fast recovery backup plan.

### **Enhanced Monitoring & Operational Excellence**

Integration with AWS CloudWatch for real-time metrics, automated alerts, and detailed monitoring would improve operational management

### **Secure Access & Data Protection**

Implementing user authentication for accessing the S3 bucket and encrypting data in both the S3 bucket and EFS would improve security measures.

### **Optimizing Performance**

Employing a caching solution, like AWS CloudFront, to provide static content would reduce the load on backend systems and improve user experience.

### **Cloud Native Practices**

Transitioning to containerization with AWS ECS or EKS could enhance the architecture, improve scalability and overall performance.

### **Implementation of Policies & Roles**

Defining reliable IAM policies and role-based access controls is crutial. Although the current lab environment limits us from creating these policies, but their implementation would significantly improve security and operational management.