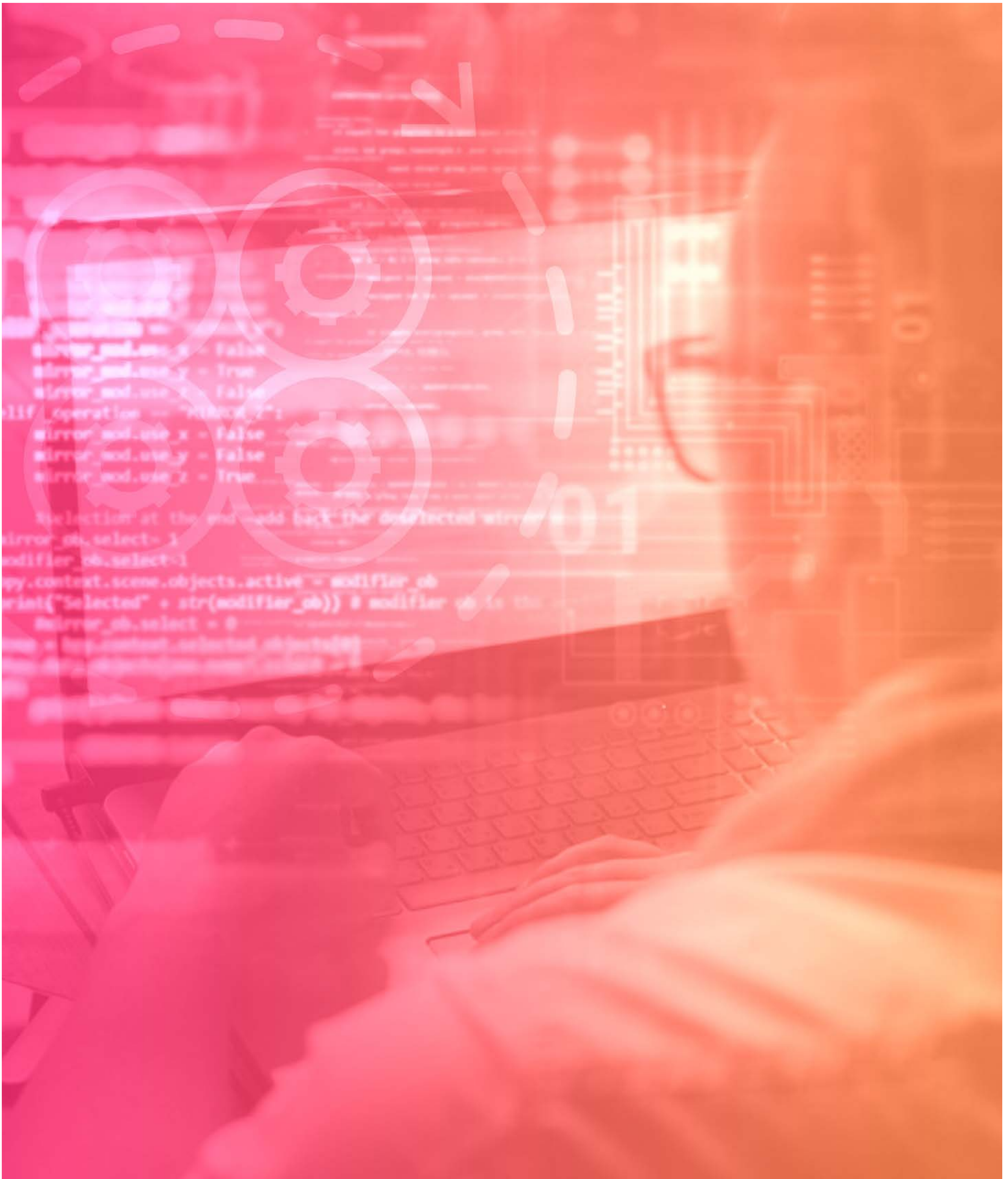
WHITE PAPER

Mphasis stelligent The Next Applied

Microservices as Containers for AWS

And How Stelligent mu Simplifies AWS Declaration and Administration of AWS Resources



CONTENTS

1. Introduction 3
2. The Problem with Monoliths 3
    - Team Coupling Affects Agility 3
    - Deployment Coupling Affects Availability 4
3. Why Microservices? 5
    - Microservice Principles 5
        - Team Autonomy 5
        - Independent Deployments 5
        - Public APIs 6
        - Private Implementations 7
4. From Monolith to Microservices 7
    - Step One: Capability Cohesion 7
    - Step Two: Data Cohesion 8
    - Step Three: Deployment Cohesion 8
5. How Do I Make All This Work? 8
    - Continuous Delivery 8
    - Containers 9
6. Stelligent mu Makes the AWS Developer's Life Easy! 10
    - mu Core Components 11
7. Summary 13

1. Introduction

Stelligent® has had the privilege of working with many clients helping them with their efforts to adopt a microservices architecture as part of their migration to Amazon® Web Services (AWS) cloud. These organizations are moving to a public cloud and want to architect their applications after these microservice patterns. Regardless of the industry segment — whether financial services, hospitality, health care, etc. — these organizations have used similar patterns and approaches to adopt these microservices architecture.

By reading this white paper, you will discover:

- The principles and best practices for implementing a microservices architecture.
- How to decompose a monolithic application into a microservices architecture.
- How you can improve your approach to develop a microservices architecture using other new technologies, such as containers, and how they complement the work that is being done in the microservices space.
- How Stelligent's new tool — mu™ — helps you to quickly and painlessly deploy your microservices architecture into the cloud.

2. The Problem with Monoliths

Team Coupling Affects Agility

If you have multiple IT teams that are working independently on individual components of the monolith, these teams are all coupled together into a single release train. For example, you may have a release train that requires your teams to push their code out monthly to a shared integration environment. Code might sit for a

few weeks while other individuals test it and if all goes well, they will promote the code to Quality Assurance (QA) for more testing. If this goes well, the code is promoted to a performance test environment, where another group will evaluate the code and then promote it to the security team. Unfortunately, you now have a three-month cycle time, that is, the time it takes for a developer to commit their code up until the time the code is used by a customer.

The business side of the house will tell you that three months is far too long a time, but when you have all the teams contributing into a single release cycle — with one team that wants to deploy daily and another team that wants to deploy weekly — all are ultimately constrained by the team that is delivering code the slowest. If one team can only develop and test their code on a monthly cycle, unfortunately, everybody else must too.

Deployment Coupling Affects Availability

The next level of coupling we see that is challenging is around the coupling of the deployment itself, where we take these individual pieces of software that individual teams are working on and connect and deploy them as a single artifact. For example, one organization decided to add a rewards engine to their website, so they could make suggestions to the consumer. Unfortunately, this feature caused the entire website to crash. Since this was a monolithic architecture, it is very difficult to shut off the rewards part of the application and allow the rest of the app to work because everything is running in a shared infrastructure.

3.
Why Microservices?

Decomposing an application into microservices can improve an organization's ability to deliver software faster. Microservices are independently deployable components; you can decouple them from other components to deliver code that is organized around a single business capability. As a result, your organization can use smaller, autonomous teams to deliver what customers want and at the pace the market demands.

Microservice Principles

Team Autonomy

With microservices, the teams can make their own choices and own the code that they deploy. Each team can choose what design approach to take, the timeline for delivery, what the priorities are, and what features they are implementing. For example, an organization may decide to deploy their microservices on Tomcat. But, you might see one team choose to implement their services using Spring Web model-view-controller (MVC) or another that chooses JAXRS and JAVA EE. Each team has the freedom to make its own choices, while still adopting the prescribed outer architecture.

Likewise, individual teams can choose to move at a different pace. If one team wants to adopt a newer version of one of the frameworks before the other team, they can do that. For example, if one team wants to use the latest version of Spring, they can make that change on a smaller scale and not wait for the rest of the teams to aggression test that change. So, you've accomplished team autonomy by giving each team complete ownership of the code that they write.

Independent Deployments

Each service can be deployed independent of each other in its own process space and in its own infrastructure

or container. By doing this, you get quicker scaling. For example, let's assume that an organization running Web Sphere JVMs with 30 services takes fifteen minutes to start the JVM app. In this case, your ability to scale out the application will take a long time, which means you will not be able to respond to demand. However, by decomposing your application image in microservices, you can scale faster and achieve a finer level of scaling; rather than scaling to support the entire stack of services for your application, you are only scaling the part that's running high. Additionally, you can limit the "blast radius" of a failure. For example, with the rewards program example discussed earlier, if it was written as a service and failed, the rest of the application can continue to perform.

Public APIs

The APIs that the microservice creates must be well defined and shared with potential consumers of those APIs. All interactions with the microservice comes through a well-documented API. When developing APIs, you need to create versions. If you need to make a change and do not want to affect your current clients, you need to consider backward compatibility. When considering your strategy for versioning, follow these three steps:

- Ask yourself if you really need to make a breaking change. If there isn't a good business case to make the change, don't do it.
- If you decide to make a change, consider making the change non-breaking by making it additive if possible.
- If you must make a breaking change, make sure that you version it to ensure backward compatibility

For example, let's assume that you have a "full name" file but want to change it to two fields: "first name" and "last name." If there is a solid business reason to make this change, consider keeping the "full name" field and adding the other two fields. Making the change this way allows existing clients to continue to use this field, while the two new fields can be used by the new clients.

In addition, all APIs must be secure. You do not want to allow microservices to call each other. You also do not want to create a single point of failure or a scaling problem by having all service authentication go through a single proxy or having a single system doing authentication. Best practice is to use some sort of peer-to-peer authentication, such as HMAC or JSON Web Tokens (JWT). Do not write your own authentication but rather, select a well-known, supported third-party offering.

Private Implementations

Only the information exposed by the API contract can be shared between microservices. Everything else needs to remain private. For example, the databases that an individual microservice uses to support their service cannot be shared between services. If you need data from another database, you must go through the public API; never talk directly to a different services database. Otherwise you end up in a situation where you're back at coupling between the services.

4. From Monolith to Microservices

Step One: Capability Cohesion

Decomposing the capabilities is all about factoring the code out into appropriate packages and then factoring those packages into appropriate artifacts. Most organizations do a great job at architecting their app in a well-thought-out manner, whether it's a monolith or a microservice. Often, this is already done. Once you have those capabilities decomposed, the next step is going after the data.

Step Two: Data Cohesion

This step is more difficult to accomplish because it deals with foreign key constrains. If you have a foreign key constraint between two tables and those tables are moved into two different databases, you must solve the problem of referential integrity at the service level through that API.

Another problem occurs when you have two different services that need access to data, for example, an address table. In this case, you want to develop a third service that has its own database so that the other two services can access the address service.

Step Three: Deployment Cohesion

Now that you have decomposed your capabilities and databases, you can deploy the application as microservices instead of deploying it as a single monolithic application.

5. How Do I Make All This Work?

A frequent question that surfaces is how to manage all these moving parts. Previously, you had one big thing to deploy and now you have fifty different things to deploy. How do you get the app into production quickly? You do this by using continuous delivery and containers.

Continuous Delivery

Continuous delivery is a practice in software engineering that allows teams to produce software in short cycles and release into production. Rather than having production deployments once a month, you go to production multiple times a day. This decreases risk and improves the ability to deliver code quicker, but it also requires a high level of automation to support the number of releases that you have with all these different microservices.

Containers

When using microservices, your organization can incur increased infrastructure overhead and reengineering costs. For example, let's assume you are deploying your microservices on Amazon Elastic Cloud Compute (EC2®) instances. For each microservice, you will need a cluster of EC2 instances to ensure adequate capacity and tolerance to failures. If a single microservice requires 12 T2 small instances to meet capacity requirements and you want to survive an outage in one out of four availability zones, you need to run 16 instances in total — four per availability zone. This leaves an overhead cost of four T2 small instances. If you multiply this cost by the number of microservices for a given application, the infrastructure overhead costs can add up quickly.

Developers use containers for deploying microservices to address this challenge. Each microservice is deployed as a series of containers to a cluster of hosts that is shared by all microservices. This allows for greater density of microservices on EC2 instances and the overhead to be shared by all microservices. Amazon ECS (EC2 Container Service) provides an excellent platform for deploying microservices as containers. It leverages many AWS services to provide a robust container management solution. In addition, a developer can use tools, such as AWS CodeBuild and AWS CodePipeline, to create continuous delivery pipelines for their microservices.

This approach can also increase reengineering costs. There is a significant learning curve for developers to learn how to use all these different AWS resources to deploy their microservices in an efficient manner. If each team is using their autonomy to engineer a platform on AWS for their microservices, then you are duplicating engineering efforts. This duplication not only causes additional engineering costs, but also impedes a team's ability to deliver the differentiating business capabilities that they were commissioned to deliver in the first place.

Stelligent has seen a few clients that have attempted microservices with continuous delivery but without containers. The challenge with this approach is that these clients were able to successfully implement the application but at a higher cost. Alternatively, some clients attempted microservices using containers but did not use continuous delivery. In this case, these organizations were able to code very quickly, but it still took three months to get into production.

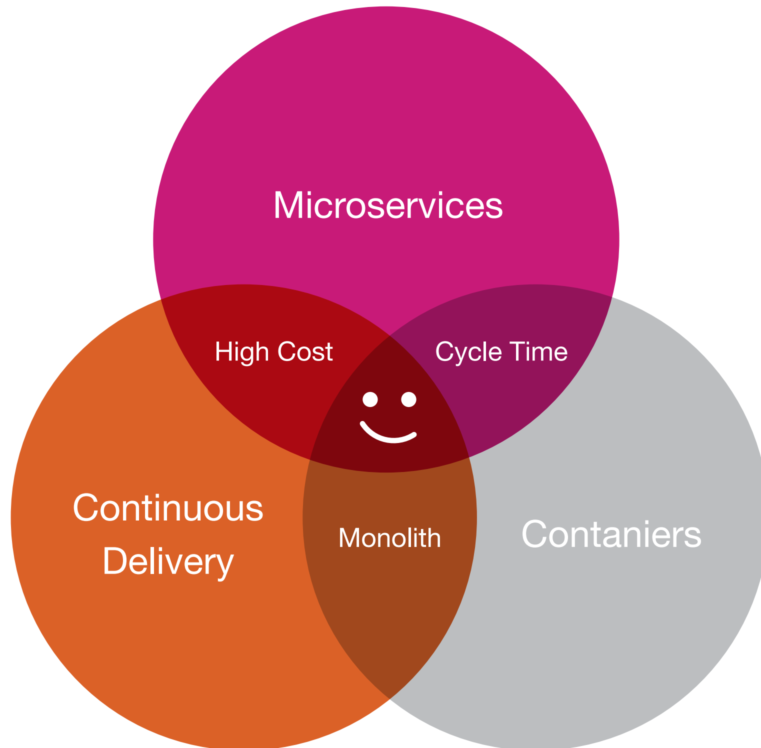


Figure 1: The World of Microservices

The “happy spot” is where you use microservices, continuous delivery, and containers (see Figure 1) but if you want to use AWS with containers and continuous delivery, Figure 2 depicts the suite of AWS Services that you will use for every single microservice plus it is ultimately up to you to wire these services together and get your microservice into production.

6. Makes the AWS Developer’s Life Easy!

To address these challenges, Stelligent created mu to simplify the declaration and administration of the AWS resources necessary to support microservices. mu is a developer’s tool. It quickly and efficiently deploys microservices to AWS as containers. It organizes best practices for microservices, containers, and continuous delivery pipelines into the AWS resources it creates on your behalf. mu uses a simple Command-Line Interface (CLI) application that can be installed on the developer’s workstation in seconds. mu makes it easier for developers to use ECS as a microservices platform, just like the way the Serverless Framework improves the developer experience with Lambda and API Gateway.

mu is completely capable and here are a few examples how:

- **Cloud Native** — mu only uses AWS resources for deploying your microservices. At any point, you can stop using mu and continue to manage the AWS resources that it created via AWS tools, such as the CLI or the console.
- **Continuous Delivery** — mu uses AWS CodePipeline and AWS CodeBuild to continuously test and deliver your microservice to production fast.
- **Polyglot** — mu doesn't have a favorite language. If you can get your microservice running with a Dockerfile, then mu can help.
- **Stateless** — mu doesn't have any servers or databases running anywhere but instead leverages AWS CloudFormation to manage the state for all AWS resources.
- **Declarative** — mu delivers what you want. You declare your configuration in a YAML file and commit with your source code. mu takes care of setting up your AWS resources to meet your needs.
- **Open Source** — mu is MIT-licensed so you can use it commercially. Stelligent is always looking to improve the mu framework so please consider contributing.

Figure 2: AWS Services

mu Core Components

The mu tool consists of three main components:

- **Environments –** includes a shared network (VPC) and cluster of hosts (ECS and EC2 instances) necessary to run microservices as clusters; environments automatically scale out or scale in based on resource

requirements across all the microservices that are deployed to it. Many environments can exist (e.g. development, staging, production).

- **Services –** a microservice that will be deployed to a given environment (or environments) as a set of containers.
- **Pipeline –** a continuous delivery pipeline that will manage the deploying of a microservice in the various environments.

mu runs in the developer's workstation. When a developer wants to deploy their service with mu, they define a YAML file, which declares the intended state of the environment and the resources. When you run the mu command locally, it generates AWSvCloudFormation templates that it sends up to the AWS CloudFormation service. AWS CloudFormation then creates your pipeline with AWS CodePipeline and AWS CodeBuild, creates the environment with an Amazon EC2 container service, and creates the ECS services for the service to run.

The pipeline mu creates has four stages:

- A source stage pulls down your source code
- A build stage to use AWS CodeBuild to compile and prepare the application to run
- An acceptance stage where your service is deployed to an acceptance environment and then validated via automated testing
- A production stage where you deploy to the production environment

Inside the mu environment, you have a test and a production environment. An environment consists of an ECS cluster and a set of ECS container instances that are in an Auto Scaling group and have Auto Scale policies based on the number of containers currently running in the cluster. There is also an Application Load Balancer (ALB) sitting in front of the environment and handling requests for the services inside the environment.

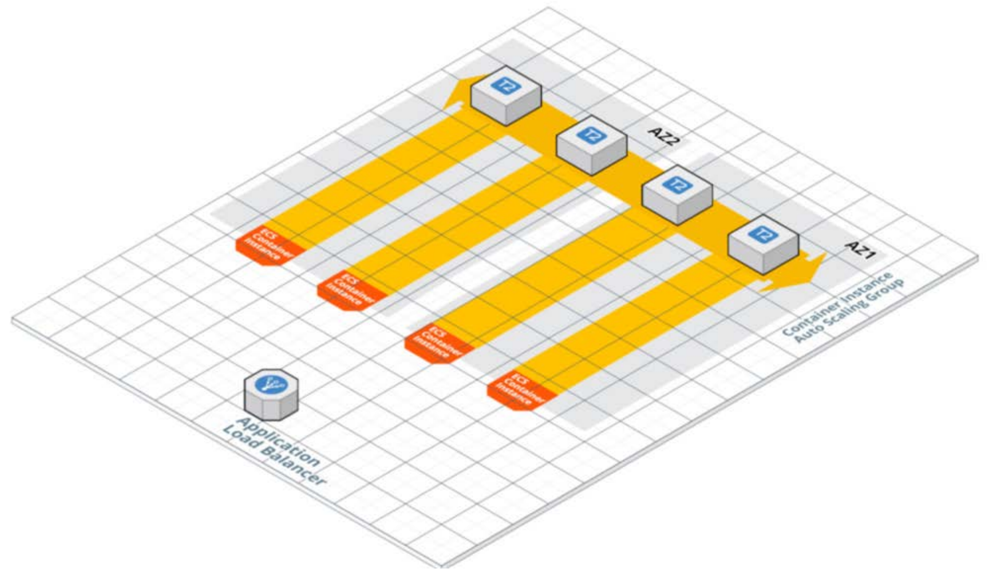


Figure 3: Environment

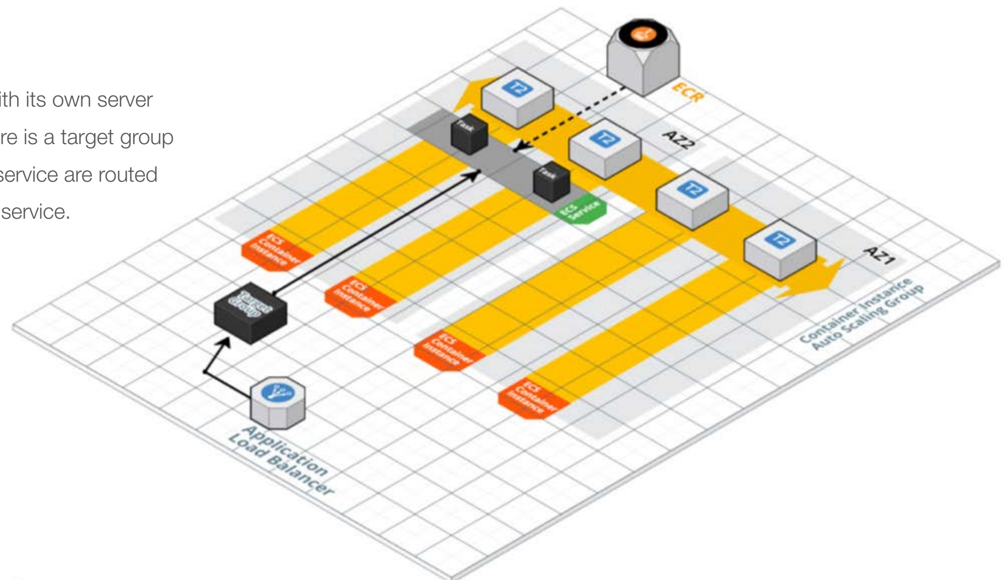


Figure 4: Service

mu deploys a pair of containers with its own server repository for the docker image. There is a target group created and URLs defined for the service are routed through the ALB into the service.

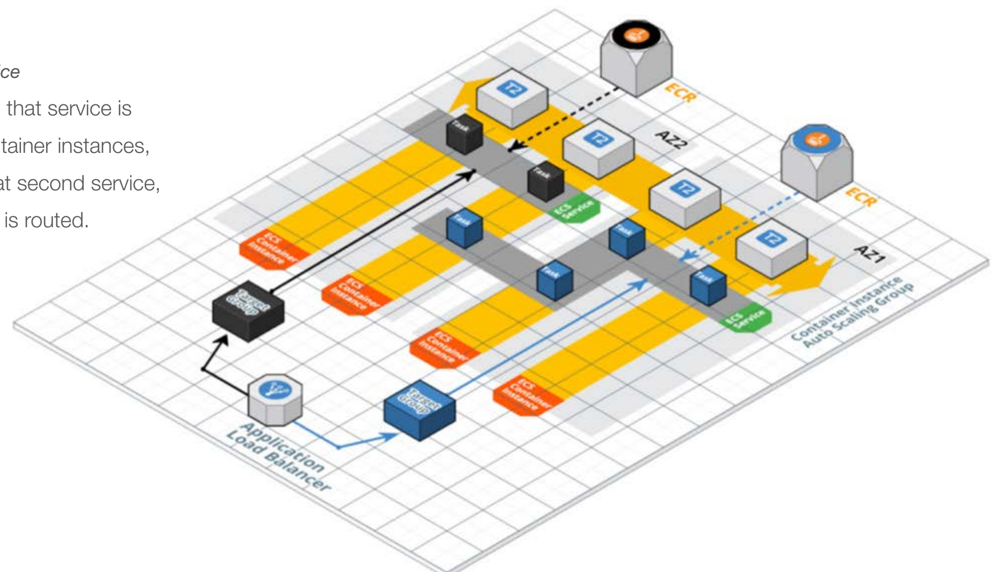


Figure 5: Second Service

If you deploy the second service, that service is deployed on the available ECS container instances, a new target group is created for that second service, and the URL for that service is routed.

6.
Summary

In this document, we discussed principles and best practices for implementing a microservices architecture, the approach that some organization have successfully used to decompose their monolith into microservices, how containers and continuous delivery can complement a microservices architecture and help achieve the benefits that microservices promises, and lastly, we introduced Stelligent's tool, mu, which helps developers simplify the declaration and administration of the AWS resources to support microservices. mu decreases the infrastructure and engineering overhead costs associated with microservices and makes it easy to deploy microservices via containers. Just as important, it ensures that deployments are repeatable by utilizing a continuous delivery pipeline to orchestrate the flow of software changes into production.

To learn more about mu, go to:

- mu - http://getmu.io/
- GitHub - https://github.com/stelligent/mu
- Blogs
  - Introduction - https://stelligent.com/2017/04/11/mu introduction-ecs-for-microservices/
  - Testing - https://stelligent.com/2017/04/27/mu-testing-continuous-delivery/
  - Databases - https://stelligent.com/2017/05/09/microservice-databases-with-mu/

ABOUT MPHASIS STELLIGENT

Mphasis Stelligent, a professional services and consulting firm with deep expertise in DevOps automation services on Amazon Web Services (AWS), enables security-conscious enterprises to focus on developing software users love by leveraging automation on AWS. Our goal is to work closely with customers to develop fundamentally secure infrastructure automation code, deployment pipelines, and feedback mechanisms for faster, more consistent software and infrastructure deployments. By embedding with our customer's engineering teams, we empower customers through education and knowledge transfer of our expertise while developing the automation to make them self-sufficient on AWS. As a Premier AWS Consulting Partner, AWS Public Sector Partner, and AWS DevOps and Financial Services Competency holder, we use our demonstrated expertise to help customers benefit from continuous AWS innovation.

For more information, contact us at: info@stelligent.com

11710 Plaza America Drive
Suite 2000
Reston, VA 20190-4743
Tel.: +1 888 924 4539

@Copyright 2019 Mphasis Stelligent. All rights reserved.