

# Database & Container Deployment

Software Architecture

March 22, 2023

Evan Hughes & Brae Webb



## 1 This Week

This week we are going to deploy our todo application, now called TaskOverflow, on AWS infrastructure using a hosted database and a single server website.

Specifically, this week you need to:

- Deploy an AWS Relational Database Service (RDS) using Terraform.
- Deploy the TaskOverflow container on AWS infrastructure using either:

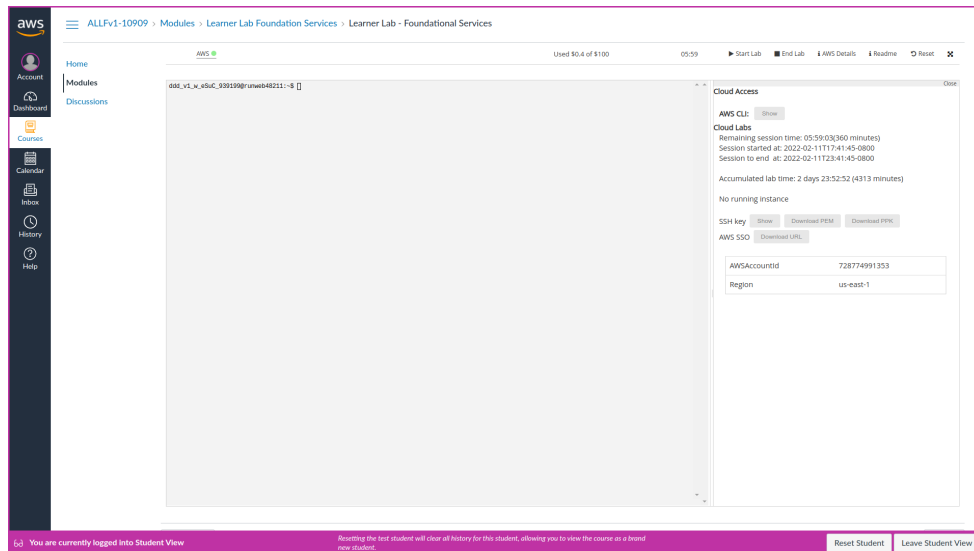
**Path A** An EC2 instance running the container.

**Path B** An ECS cluster with the container.

## 2 Terraform in AWS Learner Labs

Following the steps from the week four practical, start a Learner Lab in AWS Academy. For this practical, you do not need to create any resources using the AWS Console. The console can be used to verify that Terraform has correctly provisioned resources.

1. Using the GitHub Classroom link for this practical provided by your tutor in Slack, create a repository to work within.
2. Clone the repository or open an environment in GitHub CodeSpaces<sup>1</sup>
3. Start the Learner Lab then, once the lab has started, click on 'AWS Details' to display information about the lab.



4. Click on the first 'Show' button next to 'AWS CLI' which will display a text block starting with [default].
5. Within your repository create a `credentials` file and copy the contents of the text block into the file. **Do not share this file contents — do not commit it.**
6. Create a `main.tf` file in your repository with the following contents:

```
» cat main.tf

terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  shared_credentials_files = ["/credentials"]
}
```

<sup>1</sup>If you are using CodeSpaces, you will need to reinstall Terraform using the same steps as last week.

7. We need to initialise terraform which will fetch the required dependencies. This is done with the `terraform init` command.

```
$ terraform init
```

This command will create a `.terraform` directory which stores providers and a provider lock file, `.terraform.lock.hcl`.

8. To verify that we have setup Terraform correctly, use `terraform plan`.

```
$ terraform plan
```

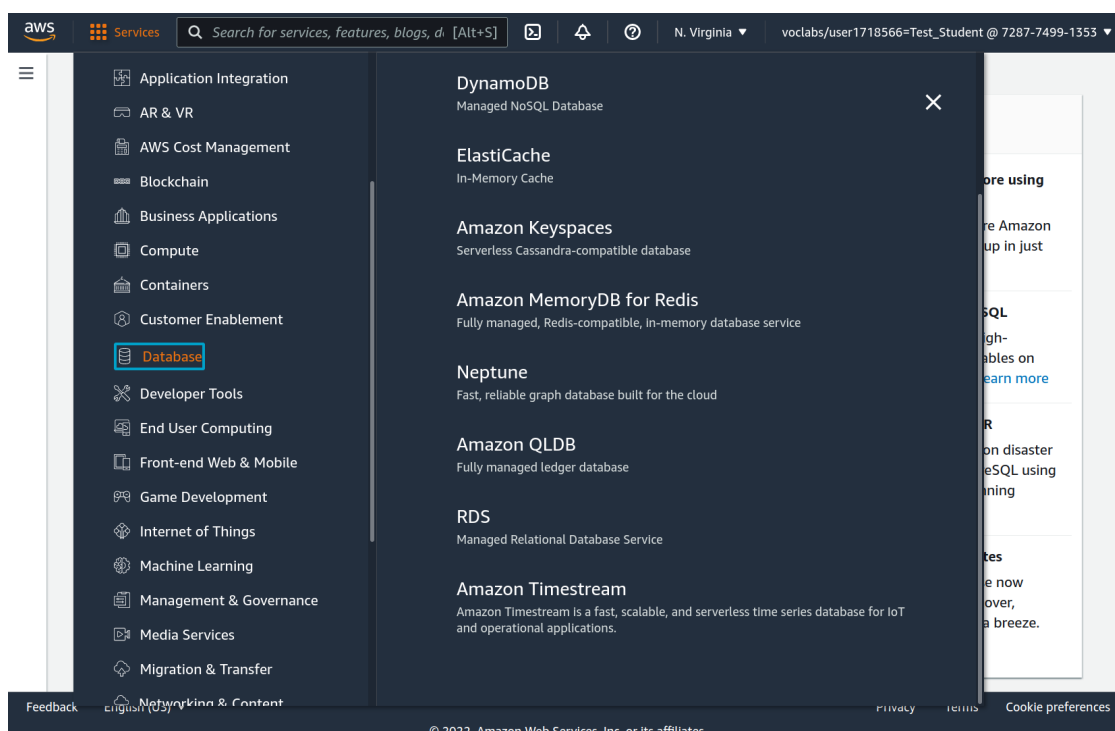
As we currently have no resources configured, it should find that no changes are required. Note that this does not ensure our credentials are correctly configured as Terraform has no reason to try authenticating yet.

### 3 Deploying a Database in AWS

#### Warning

This section manually deploys a PostgreSQL RDS instance, this is intended as a demonstration by your tutor. You should attempt to deploy your infrastructure using Terraform rather than manually.

To get started let us jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search it or browse Services -> Database -> RDS as shown below.



Now we are in the management interface for all our RDS instances. Head to “DB Instances (0/40)” or click “Databases” on the left panel.

The screenshot shows the Amazon RDS console dashboard. On the left is a navigation menu with options like Dashboard, Databases, Query Editor, Performance insights, Snapshots, Automated backups, Reserved instances, Proxies, Subnet groups, Parameter groups, Option groups, Custom Availability Zones, Custom engine versions, Events, and Event subscriptions. The main area is titled 'Resources' and shows a summary of RDS resources in the US East (N. Virginia) region. It lists DB Instances (0/40), Parameter groups (0), Option groups (0), DB Clusters (0/40), Reserved instances (0/40), Snapshots (0), Subnet groups (0/50), Supported platforms VPC, and Event subscriptions (0/20). Below this is a 'Create database' section with a description of Amazon RDS. On the right, there are 'Recommended for you' sections for Time-Series Tables in PostgreSQL, Implementing Cross-Region DR, Amazon RDS Backup and Restore using AWS Backup, and Build RDS Operational Tasks. The bottom of the page has a footer with 'Feedback', 'English (US)', 'Privacy', 'Terms', 'Cookie preferences', and a copyright notice for 2022.

This page should appear familiar as it is very similar to the AWS EC2 instance page. Let us create a new database by hitting the “Create Database” button.

The screenshot shows the 'Databases' page in the Amazon RDS console. At the top, there's a breadcrumb 'RDS > Databases'. Below it, there's a section titled 'Databases' with a 'Group resources' toggle, a 'Refresh' button, and buttons for 'Modify', 'Actions', 'Restore from S3', and 'Create database'. A search bar labeled 'Filter by databases' is present. Below the search bar is a table with columns: DB identifier, Role, Engine, Region & AZ, Size, Status, and CPU. The table is currently empty, showing 'No instances found'.

### Warning

In the next section we cannot use the Easy Create option as it tries to create a IAM account which is disabled in Learner Labs.

We will be creating a standard database so select standard and PostgreSQL. We will use version 14, which is a fairly recent release.

## Choose a database creation method [Info](#)

### ☒ Standard create

You set all of the configuration options, including ones for availability, security, backups, and maintenance.

### ☐ Easy create

Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

## Engine options

### Engine type [Info](#)

#### ☐ Amazon Aurora



#### ☒ MySQL



#### ☐ MariaDB



#### ☐ PostgreSQL



#### ☐ Oracle



#### ☐ Microsoft SQL Server



### Edition

#### ☒ MySQL Community



#### Known issues/limitations

Review the [Known issues/limitations](#) [to learn about potential compatibility issues with specific database versions.](#)

### Version

MySQL 8.0.27

**TODO: Update to use PostgreSQL.**

For today we are going to use “Free Tier” but in the future, you may wish to explore the different deployment options. Please peruse the available different options.

## Templates

Choose a sample template to meet your use case.



### Production

Use defaults for high availability and fast, consistent performance.



### Dev/Test

This instance is intended for development use outside of a production environment.



### Free tier

Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS.

[Info](#)

## Availability and durability

### Deployment options [Info](#)

The deployment options below are limited to those supported by the engine you selected above.



#### Single DB instance (not supported for Multi-AZ DB cluster snapshot)

Creates a single DB instance with no standby DB instances.



#### Multi-AZ DB instance (not supported for Multi-AZ DB cluster snapshot)

Creates a primary DB instance and a standby DB instance in a different AZ. Provides high availability and data redundancy, but the standby DB instance doesn't support connections for read workloads.



#### Multi-AZ DB Cluster - new

Creates a DB cluster with a primary DB instance and two readable standby DB instances, with each DB instance in a different Availability Zone (AZ). Provides high availability, data redundancy and increases capacity to serve read workloads.

Now we need to name our database and create credentials to use when connecting from our application. Enter memorable credentials as these will be used later.

## Settings

### DB instance identifier [Info](#)

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

### ▼ Credentials Settings

#### Master username [Info](#)

Type a login ID for the master user of your DB instance.

1 to 16 alphanumeric characters. First character must be a letter.

☐ **Auto generate a password**

Amazon RDS can generate a password for you, or you can specify your own password.

#### Master password [Info](#)

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), ' (single quote), " (double quote) and @ (at sign).

#### Confirm password [Info](#)

For exploring the process select t2.micro, which should be adequate for our needs.

## DB instance class

### DB instance class [Info](#)

- ☐ Standard classes (includes m classes)
- ☐ Memory optimized classes (includes r and x classes)
- ☒ Burstable classes (includes t classes)

1 vCPUs   1 GiB RAM   Not EBS Optimized



☐ Include previous generation classes

For storage we will leave all the default options.

## Storage

Storage type [Info](#)

General Purpose SSD (gp2)


Baseline performance determined by volume size

Allocated storage

20

GiB

(Minimum: 20 GiB. Maximum: 16,384 GiB) Higher allocated storage **may improve** IOPS performance.

 You might see better baseline performance with your selected volume size by specifying General Purpose SSD storage. [Learn more about using Provisioned IOPS storage for consistent performance.](#)

Storage autoscaling [Info](#)

Provides dynamic scaling support for your database's storage based on your application's needs.

☒ **Enable storage autoscaling**

Enabling this feature will allow the storage to increase once the specified threshold is exceeded.

Maximum storage threshold [Info](#)

Charges will apply when your database autoscales to the specified threshold

1000

GiB

Minimum: 21 GiB. Maximum: 16,384 GiB

In connectivity we need to make sure our instance is publicly available. Usually you do not want to expose your databases publicly and, would instead, have a web server sitting in-front. For our learning purposes though we are going to expose it directly just like we did with our EC2 instances early in the course.

When selecting public access as yes we have to create a new Security Group, give this Security Group a sensible name.



## Connectivity



### Virtual private cloud (VPC) [Info](#)

VPC that defines the virtual networking environment for this DB instance.

Default VPC (vpc-07f8e8ea0408a9db9) ▼

Only VPCs with a corresponding DB subnet group are listed.

After a database is created, you can't change its VPC.

### Subnet group [Info](#)

DB subnet group that defines which subnets and IP ranges the DB instance can use in the VPC you selected.

default-vpc-07f8e8ea0408a9db9 ▼

### Public access [Info](#)

☒ Yes

Amazon EC2 instances and devices outside the VPC can connect to your database. Choose one or more VPC security groups that specify which EC2 instances and devices inside the VPC can connect to the database.

☐ No

RDS will not assign a public IP address to the database. Only Amazon EC2 instances and devices inside the VPC can connect to your database.

### VPC security group

Choose a VPC security group to allow access to your database. Ensure that the security group rules allow the appropriate incoming traffic.



Choose existing

Choose existing VPC security groups



Create new

Create new VPC security group

### New VPC security group name

todoapp-manual

### Availability Zone [Info](#)

No preference ▼

### ▼ Additional configuration

#### Database port [Info](#)

TCP/IP port that the database will use for application connections.

3306



We will leave the authentication as password based but we need to expand the “Additional configuration”. Fill in the “Initial Database Name” section to be “todo”, this will automatically create the database that our todo application expects to connect to.

## Database authentication

Database authentication options [Info](#)

- ☒ **Password authentication**  
Authenticates using database passwords.
- ☐ **Password and IAM database authentication**  
Authenticates using the database password and user credentials through AWS IAM users and roles.
- ☐ **Password and Kerberos authentication**  
Choose a directory in which you want to allow authorized users to authenticate with this DB Instance using Kerberos Authentication.

### ▼ Additional configuration

Database options, backup enabled, backtrace disabled, Enhanced Monitoring disabled, maintenance, CloudWatch Logs, delete protection disabled.

### Database options

Initial database name [Info](#)

If you do not specify a database name, Amazon RDS does not create a database.

DB parameter group [Info](#)

Option group [Info](#)

Now we can click create which will take some time.


## Estimated monthly costs

The Amazon RDS Free Tier is available to you for 12 months. Each calendar month, the free tier will allow you to use the Amazon RDS resources listed below for free:

- 750 hrs of Amazon RDS in a Single-AZ db.t2.micro Instance.
- 20 GB of General Purpose Storage (SSD).
- 20 GB for automated backup storage and any user-initiated DB Snapshots.

[Learn more about AWS Free Tier.](#)

When your free usage expires or if your application use exceeds the free usage tiers, you simply pay standard, pay-as-you-go service rates as described in the [Amazon RDS Pricing page.](#)

 You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

Cancel


Create database


It may take 10 to 30 minutes to create. The database will also do a initial backup when its created.

RDS > Databases

**Databases**

☒ Group resources

 Modify Actions Restore from S3 Create database

 DB identifier

▲

Role ▼


Engine ▼

Region & AZ ▼

Size ▼

Status ▼

CPU


 todoapp-manual

Instance

MySQL Community

us-east-1f

db.t2.micro

 Backing-up

100%

When the database has finished being created you can select it to view the configuration and details. In this menu we also see the endpoint address which we will need to configure our TaskOverflow application to use.

Successfully created database [todoapp-manual](#)

View connection details

✕

RDS > Databases > todoapp-manual

todoapp-manual

Modify

Actions ▾

Summary

DB identifier todoapp-manual	CPU <div><div></div></div> 10.83%	Status Available	Class db.t2.micro
Role Instance	Current activity <div><div></div></div> 0 Connections	Engine MySQL Community	Region & AZ us-east-1f

Connectivity & security

Monitoring

Logs & events

Configuration

Maintenance & backups

Tags

Connectivity & security

<div>Endpoint &amp; port</div> <div>Endpoint todoapp-manual.cwf1cdgoxax.us-east-1.rds.amazonaws.com</div> <div>Port 3306</div>	<div>Networking</div> <div>Availability Zone us-east-1f</div> <div>VPC <a href="#">vpc-07f8e8ea0408a9db9</a></div> <div>Subnet group default-vpc-07f8e8ea0408a9db9</div> <div>Subnets <a href="#">subnet-0556db549e22800f1</a> <a href="#">subnet-0da793726639bd6d8</a> <a href="#">subnet-091ee1d302ae831a9</a> <a href="#">subnet-0b932c4d6a4154b2a</a> <a href="#">subnet-0416084227ea643b4</a> <a href="#">subnet-05d92ccc16a62294f</a></div>	<div>Security</div> <div>VPC security groups <a href="#">todoapp-manual2</a> <a href="#">(sg-0cc1a6ba52b85e23c)</a>  Active</div> <div>Publicly accessible Yes</div> <div>Certificate authority rds-ca-2019</div> <div>Certificate authority date August 23, 2024, 03:08 (UTC±3:08)</div>
--	---	---

## 4 RDS Database with Terraform

```
» cat main.tf

locals {
  database_username = "administrator"
  database_password = "foobarbaz" # this is bad
}

resource "aws_db_instance" "taskoverflow_database" {
  allocated_storage = 20
  max_allocated_storage = 1000
  engine = "postgres"
  engine_version = "14"
```

```

instance_class = "db.t4g.micro"
db_name = "todo"
username = local.database_username
password = local.database_password
parameter_group_name = "default.postgres14"
skip_final_snapshot = true
vpc_security_group_ids = [aws_security_group.taskoverflow_database.id]
publicly_accessible = true

tags = {
  Name = "taskoverflow_database"
}
}

```

When we created the database using the AWS Console, we needed an appropriate security group so that we could access the database. We can create the security group using Terraform as well.

```

» cat main.tf

resource "aws_security_group" "taskoverflow_database" {
  name = "taskoverflow_database"
  description = "Allow inbound Postgresql traffic"

  ingress {
    from_port = 5432
    to_port = 5432
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = [ ":::/0"]
  }

  tags = {
    Name = "taskoverflow_database"
  }
}

```

**TODO: Connect to the database and explore around.**

## 5 Container on AWS

As we mentioned in the Infrastructure as Code notes [1], in this course we will use Docker to configure machines and Terraform to configure infrastructure. AWS has the ability to deploy Docker containers using a service known as Elastic Container Service (ECS). We will cover ECS and deploying manually via EC2 so you can use the method you feel most comfortable with.

For this practical we have made available a Docker container running the TaskOverflow application which you can use for your AWS deployment. This container is available on GitHub under the CSSE6400 organisation:

<https://ghcr.io/csse6400/taskoverflow:latest>

This container is very similar to what you have been building in the practicals but contains a simple UI and some extra features for the future practicals.<sup>2</sup>

### 5.1 Setup

Of all the different ways that we can deploy our application, we have decided to offload the database to AWS RDS. This means that we can move all the "state" of our application away from our containerised environment.

To begin, we will reuse our Terraform from above for deploying the RDS database. Extend the existing local Terraform variables to include the address of the container, such that we have:

```
» cat main.tf

locals {
  image = "ghcr.io/csse6400/taskoverflow:latest"
  database_username = "administrator"
  database_password = "foobarbaz" # this is bad
}
```

This already sets up an RDS instance of Postgres and a security group to allow access to it. Now we can run `terraform init` and `terraform apply` to create our database.

We have also added a local variable for us to use later. Variables in Terraform can be populated via two mechanisms, they can be in a variables block which can be overridden, or they can be in a locals block which can be used to store values that are used in multiple places.

For the next step you must choose a path to follow, either EC2 or ECS, if you have trouble with one path you can always switch to the other. When switching you should destroy the resources you have created before starting the new path.

We recommend that you start with the ECS path as it is the more modern way of deploying containers and is the path that we will be suggesting for the future but all tasks are also doable using EC2.

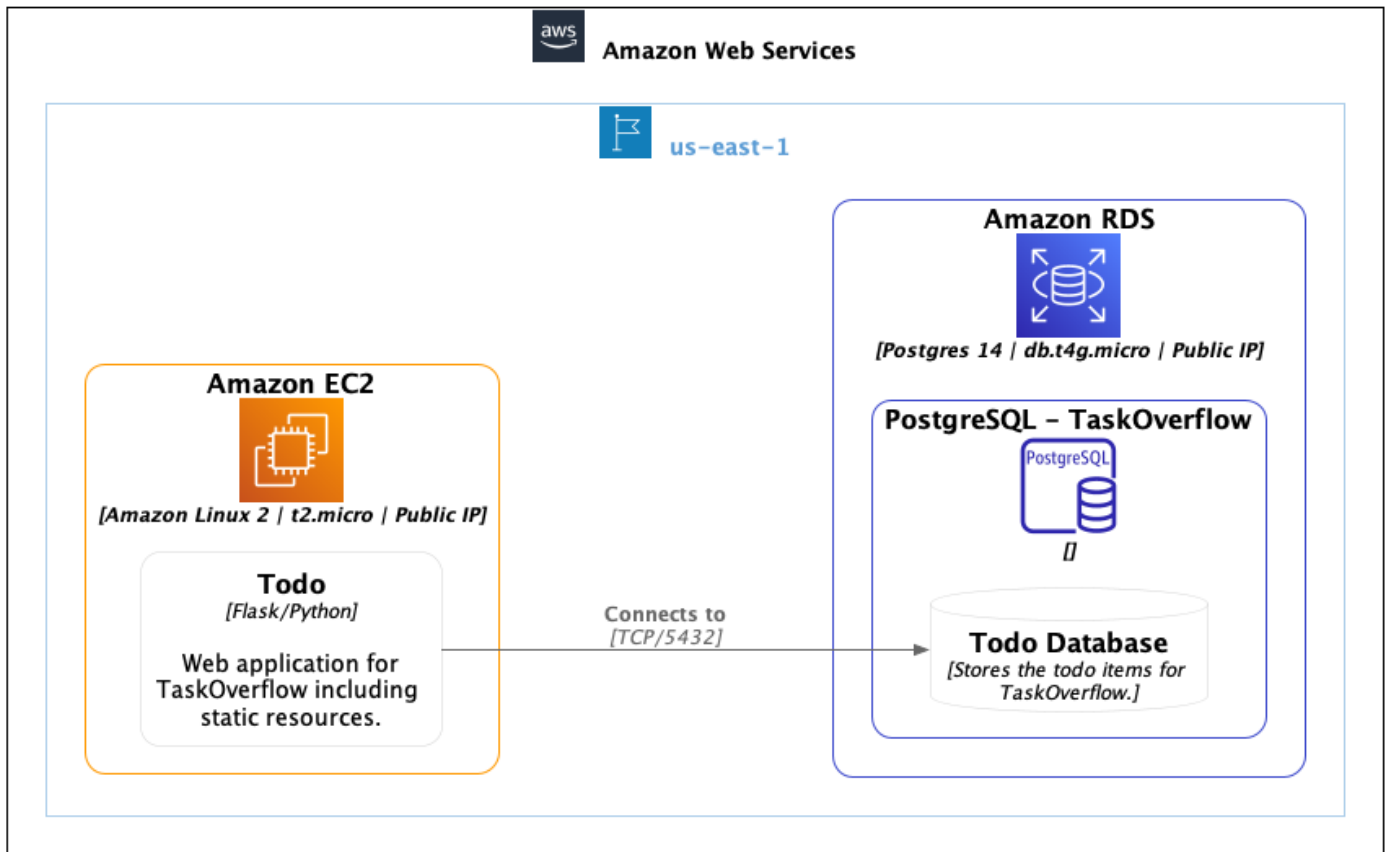
### 5.2 [Path A] EC2

#### Aside

This is not the recommended path but is provided for those who would like to use EC2 instead of ECS. Please skip to Section 5.3 for the ECS approach.

<sup>2</sup>If you are interested, the source code is available on GitHub <https://github.com/csse6400/practical>

## TaskOverflow on EC2 (Deployment Diagram)



**Legend**  
container

Congratulations! You have chosen to go down the EC2 path which builds on what we experienced in the week 4 practical. To deploy our application we first need to get the EC2 instance running and install Docker on the instance. We can do this by using the following Terraform code:

```

» cat main.tf
resource "aws_instance" "taskoverflow_instance" {
  ami = "ami-005f9685cb30f234b" # Amazon Linux 2
  instance_type = "t2.micro"
  key_name = "vockey" # allows SSH into the instance using the preconfigured key

  user_data_replace_on_change = true # changing user_data will force recreate
  user_data = <<-EOT
#!/bin/bash
yum update -y
EOT

  security_groups = [aws_security_group.taskoverflow_instance.name] # firewall for
    the instance

  tags = {
    Name = "taskoverflow_instance"
  }

```

```
}  
}
```

Compared to the last time we used EC2 instances we have moved the `user_data` from a file to being defined in-line and we have made sure to provide the `key_name` in case we want to SSH into the deployed instance.

If we ran `terraform apply` now, our instance would not do anything interesting. We need the `user_data` to install Docker and start our TaskOverflow image. For this, we will need to make the following adjustments to the `user_data` field:

#### Warning

The `<<-EOT` line cannot have a trailing space. Ensure that one has not been erroneously inserted.

```
>> cat main.tf  
  
user_data = <<-EOT  
#!/bin/bash  
yum update -y  
yum install -y docker  
service docker start  
systemctl enable docker  
usermod -a -G docker ec2-user  
docker run --restart always -e SQLALCHEMY_DATABASE_URI=postgresql://${local.  
    database_username}:${local.database_password}@${aws_db_instance.  
    taskoverflow_database.address}:${aws_db_instance.taskoverflow_database.port}/${  
    aws_db_instance.taskoverflow_database.db_name} -p 6400:6400 ${local.image}  
EOT
```

The first lines install Docker and start the Docker service and allow the `ec2-user` to be able to run Docker commands. The last line is where the magic happens by running the container via the Docker CLI. To get a refresher about the Docker CLI please see the Containers lecture [2] and the [Docker documentation](#)<sup>3</sup>.

#### Aside

The `--restart always` flag tells docker to restart the container if it crashes or is stopped. This is useful for our application as it may take a while for the database to be provisioned and we do not want to have to manually restart the container once the database is ready.

Now that our instance is ready to go we just need to make sure that it is accessible from the internet. We can do this by creating a security group that allows traffic on port 6400 and attaching it to our instance.

```
>> cat main.tf  
  
resource "aws_security_group" "taskoverflow_instance" {  
    name = "taskoverflow_instance"  
    description = "TaskOverflow Security Group"
```

---

<sup>3</sup><https://docs.docker.com/>



```

ingress {
  from_port = 6400
  to_port   = 6400
  protocol  = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

ingress {
  from_port = 22
  to_port   = 22
  protocol  = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

egress {
  from_port = 0
  to_port   = 0
  protocol  = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}
}

```

Now we can run `terraform init` and `terraform apply` to deploy our application. Once this is done we can visit the public IP of our instance on port 6400 to see our application running.

```

» cat main.tf
output "url" {
  value = "http://${aws_instance.taskoverflow_instance.public_ip}:6400/"
}

```

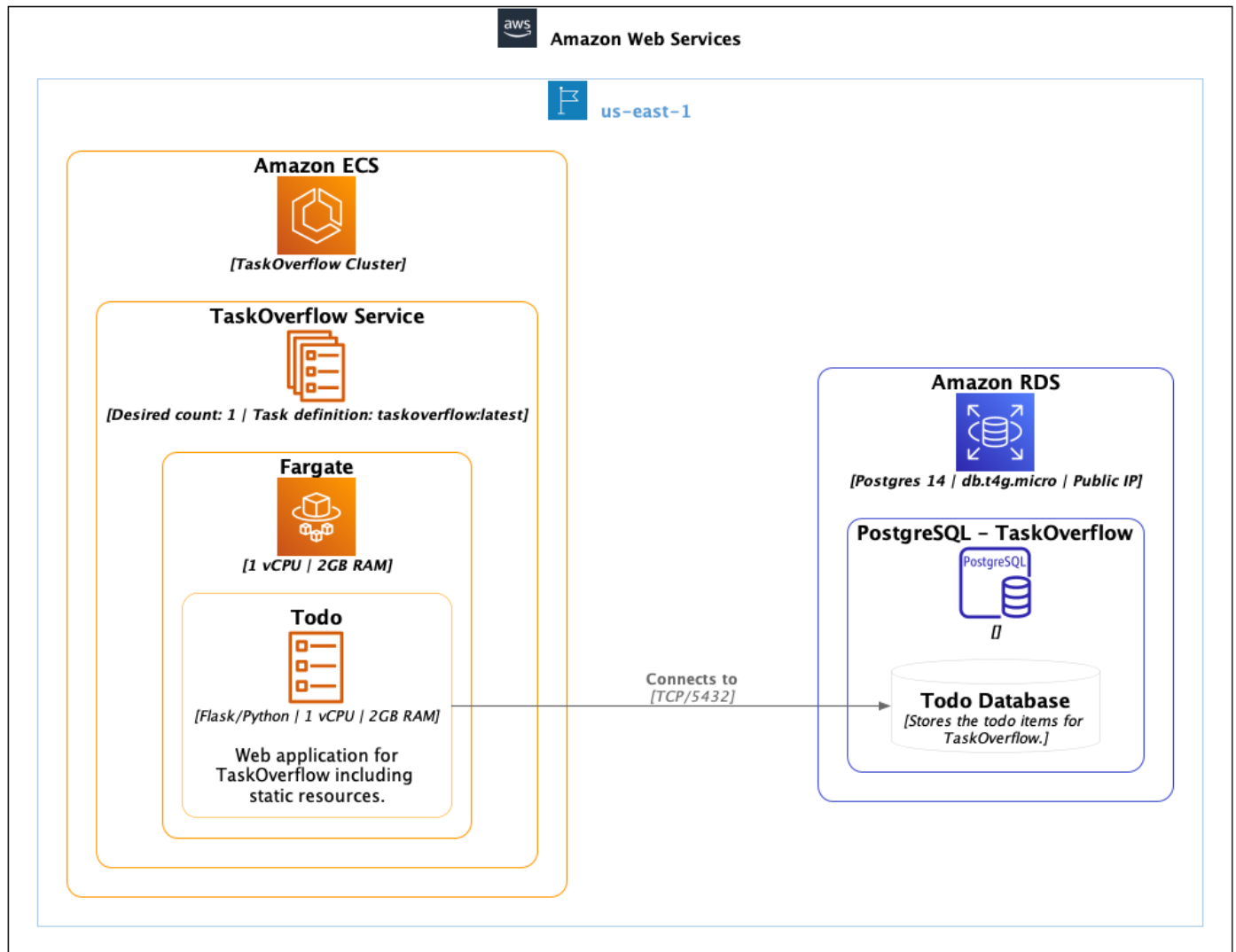
You should be presented with a very lightweight todo application called TaskOverflow.

## 5.3 [Path B] ECS

### Aside

This is the recommended path for the course and is the path that we will be suggesting for the future.

## TaskOverflow on ECS (Deployment Diagram)



Legend  
container

**TODO: Can we get higher resolution?**

Congratulations! You have chosen to go down the ECS path which mimics a similar environment as Docker Compose but as an AWS service. This path is new for the course this year so please let your tutors know of any issues you have.

To start off we need to get some information from our current AWS environment so that we can use it later. Add the below to fetch the IAM role known as LabRole which is a super user in the Learner Lab environments which can do everything you can do through the UI. We will also be fetching the default VPC and the private subnets within that VPC as they are required for the ECS network configuration.

```
data "aws_iam_role" "lab" {
  name = "LabRole"
}
```

```
data "aws_vpc" "default" {
  default = true
}

data "aws_subnets" "private" {
  filter {
    name = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}
```

In Terraform, the way to retrieve external information is data sources. These are functionally like resources but they are not created or destroyed, instead they are populated with attributes from the current state. See the below for the minor syntactic difference.

```
data "aws_iam_role" "lab" {
  ...
}

resource "aws_db_instance" "database" {
  ...
}
```

Now that we have access to the information required, we can create the ECS cluster to host our application.

The first step is to create the ECS cluster which is just a logical grouping of any images. All that is required is a name for the new grouping.

```
>> cat main.tf

resource "aws_ecs_cluster" "taskoverflow" {
  name = "taskoverflow"
}
```

On it's own this cluster is not particularly useful. We need to create a task definition which is a description of the container that we want to run. This is where we will define the image that we want to run, the environment variables, the port mappings, etc. This is similar to a server entry in Docker Compose.

#### Warning

The «DEFINITION line cannot have a trailing space. Ensure that one has not been erroneously inserted.

```
>> cat main.tf
```

```

resource "aws_ecs_task_definition" "taskoverflow" {
  family = "taskoverflow"
  network_mode = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu = 1024
  memory = 2048
  execution_role_arn = data.aws_iam_role.lab.arn

  container_definitions = <<DEFINITION
[
  {
    "image": "${local.image}",
    "cpu": 1024,
    "memory": 2048,
    "name": "todo",
    "networkMode": "awsvpc",
    "portMappings": [
      {
        "containerPort": 6400,
        "hostPort": 6400
      }
    ],
    "environment": [
      {
        "name": "SQLALCHEMY_DATABASE_URI",
        "value": "postgresql://${local.database_username}:${local.database_password}
          @${aws_db_instance.taskoverflow_database.address}:${aws_db_instance.
            taskoverflow_database.port}/${aws_db_instance.taskoverflow_database.
              db_name}"
      }
    ],
    "logConfiguration": {
      "logDriver": "awslogs",
      "options": {
        "awslogs-group": "/taskoverflow/todo",
        "awslogs-region": "us-east-1",
        "awslogs-stream-prefix": "ecs",
        "awslogs-create-group": "true"
      }
    }
  }
]
  DEFINITION
}

```

**family** A family is similar to the name of the task but it is a name that persists through multiple revisions of the task.

**network\_mode** This is the network mode that the container will run in, we want to run on regular AWS VPC infrastructure.

**requires\_compatibilities** This is the type of container that we want to run. This can be fargate, EC2, or external.

**cpu** The amount of CPU units that the container will be allocated. 1024 is equivalent to one vCPU.

**memory** The amount of memory that the container will be allocated, here we've chosen 2GB.

**execution\_role\_arn** The IAM role that the container will run as. Importantly, we have re-used the lab role we previously retrieved. This gives the instance full admin permission for our lab environment.

**container\_definitions** This is the definition of the container, it should look very familiar to Docker Compose. The only additional feature here is the `logConfiguration`. This configures our container to write logs to AWS CloudWatch so that we can see if anything has gone wrong.

Now we have a description of our container as a task. We need a service to run the container on. This is functionally similar to an auto-scaling group from the lecture. We specify how many instances of the described container we want and it will provision them. We also specify which ECS cluster and AWS subnets to run the containers within.

```
» cat main.tf
resource "aws_ecs_service" "taskoverflow" {
  name = "taskoverflow"
  cluster = aws_ecs_cluster.taskoverflow.id
  task_definition = aws_ecs_task_definition.taskoverflow.arn
  desired_count = 1
  launch_type = "FARGATE"

  network_configuration {
    subnets = data.aws_subnets.private.ids
    security_groups = [aws_security_group.taskoverflow.id]
    assign_public_ip = true
  }
}
```

In the above we refer to a non-existent security group. As always, to be able to access our instances over the network we need to add a security group policy to enable it.

```
» cat main.tf
resource "aws_security_group" "taskoverflow" {
  name = "taskoverflow"
  description = "TaskOverflow Security Group"

  ingress {
    from_port = 6400
    to_port = 6400
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```

ingress {
  from_port = 22
  to_port   = 22
  protocol  = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}

egress {
  from_port = 0
  to_port   = 0
  protocol  = "-1"
  cidr_blocks = ["0.0.0.0/0"]
}
}

```

Finally, if we run the appropriate `terraform init` and `terraform apply` commands, it should provision an ECS cluster with a service that will then create one ECS container based on our task description.

Note that we are doing something a bit weird in this deployment. Normally ECS expects multiple instances of containers, so it naturally expects a load balancer. This makes it difficult for us to discover the public IP of our single instance using Terraform. Instead, you will need to use the AWS Console to find the public IP address.

This is an opportunity for you to explore the ECS interface and find the task, within the service, within the cluster that we have provisioned.

## 5.4 [Path C] EKS / K8S

This path is not described in the course yet, but we recommend that if you liked the course to have a look at [Kubernetes](#)<sup>4</sup> as it is widely used in industry.

## References

- [1] B. Webb, "Infrastructure as code," March 2022. <https://csse6400.uqcloud.net/handouts/iac.pdf>.
- [2] B. Webb, "Containers," March 2023. <https://csse6400.uqcloud.net/slides/containers.pdf>.

---

<sup>4</sup><https://kubernetes.io/>