Database & Container Deployment

Software Architecture

March 24, 2026

Evan Hughes, Brae Webb & Richard Thomas

Aside

Github Classroom links for this practical can be found on Edstem https://edstem.org/au/courses/21491/discussion/2429006

Warning

This practical **cannot** be completed on Windows. The Docker provider for Terraform encounters an error when creating an image on Windows. You will need to use WSL instead of the native Windows environment.

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.

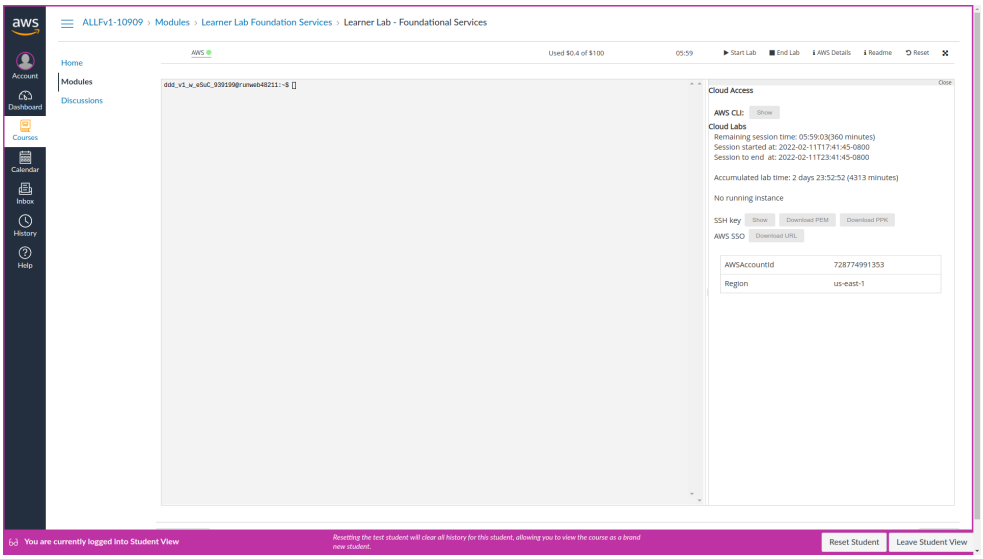
Specifically, this week you need to:

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

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 on edstem, create a repository to work within.
2. Clone the repository or open an environment in GitHub CodeSpaces[1]
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 the your repository with the following contents:

```
» cat main.tf
```

[1] If you are using CodeSpaces, you will need to reinstall Terraform using the same steps as last week.

```
terraform {
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 5.0"
        }
    }
}

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

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
```

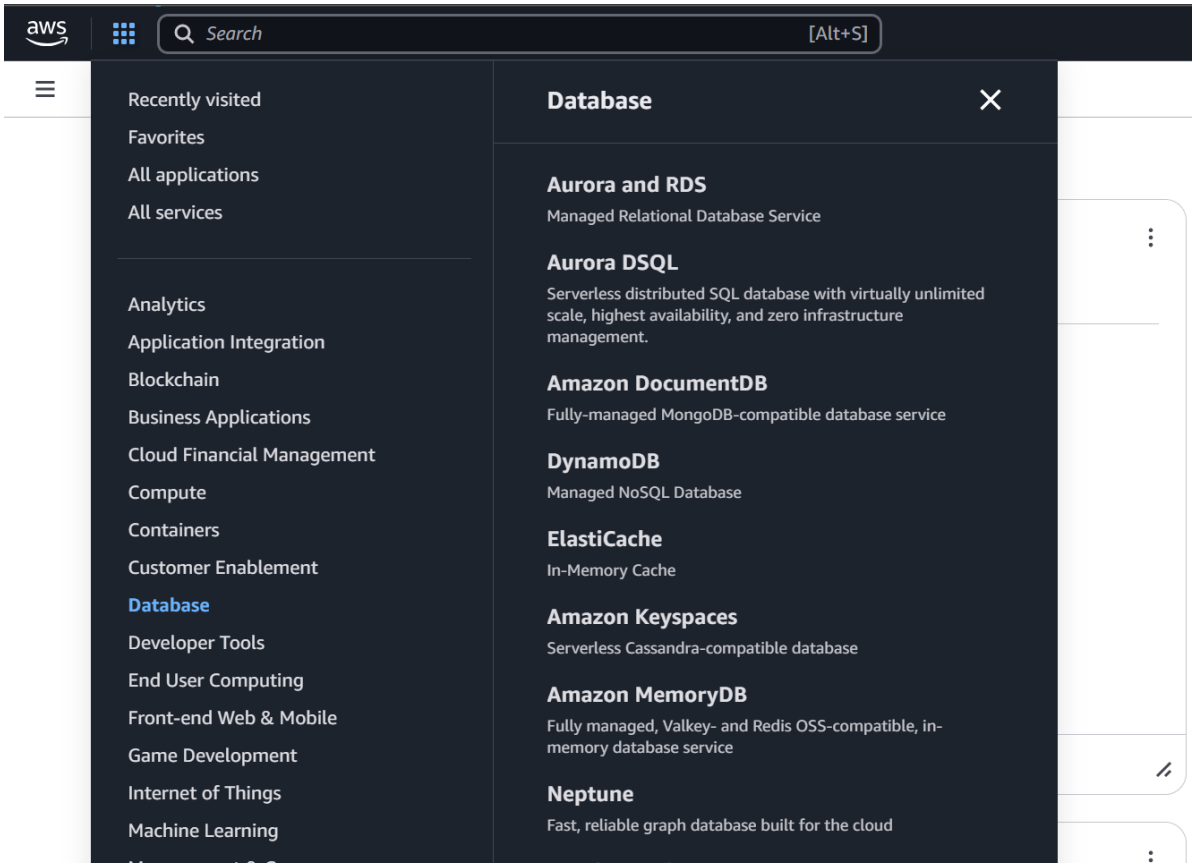
   As we currently have no resources configured, it should find that no changes are required. Note that this does not ensure our credentials are correctly configured as Terraform has no reason to try authenticating yet.

3 Deploying a Database in AWS

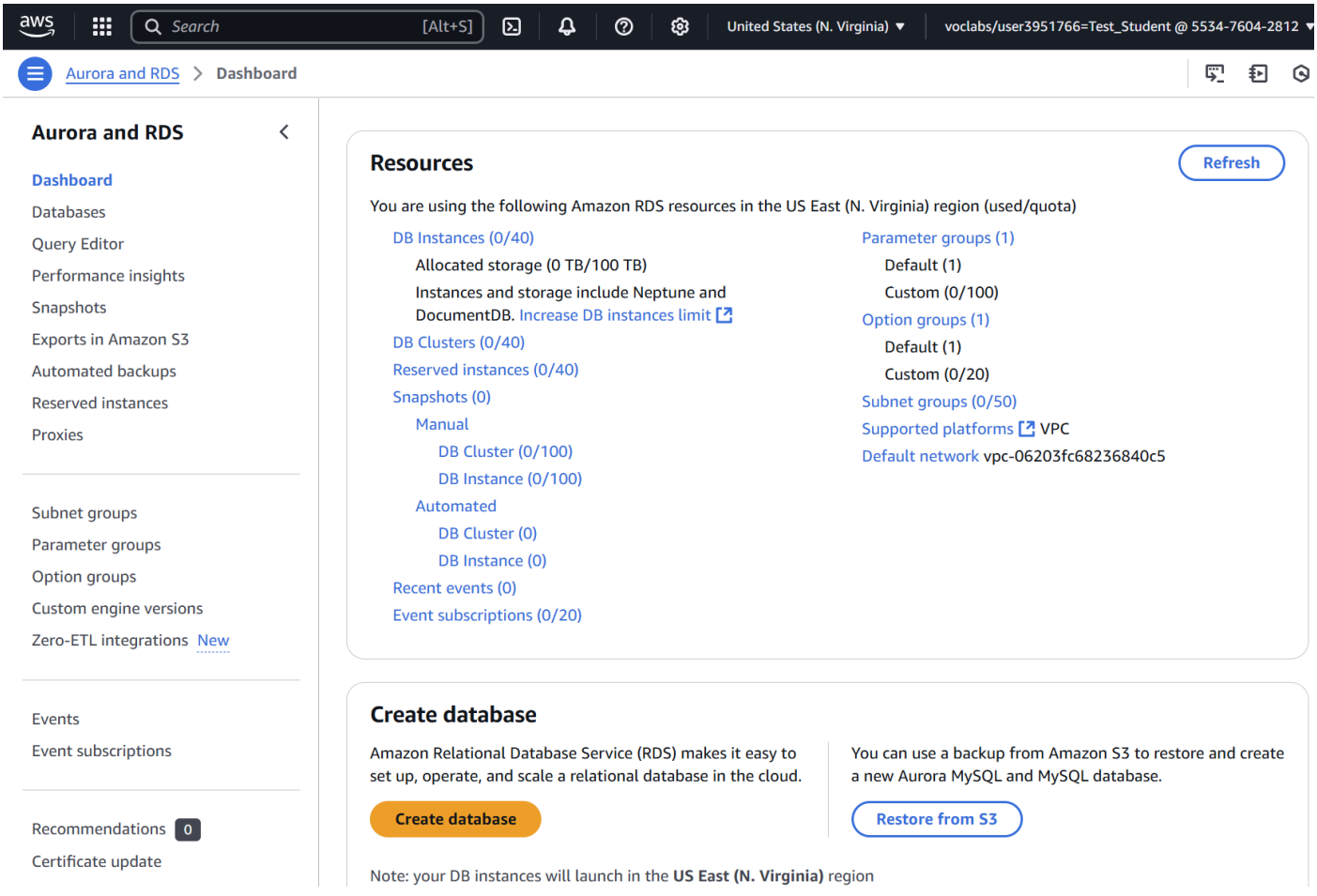
Warning

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.

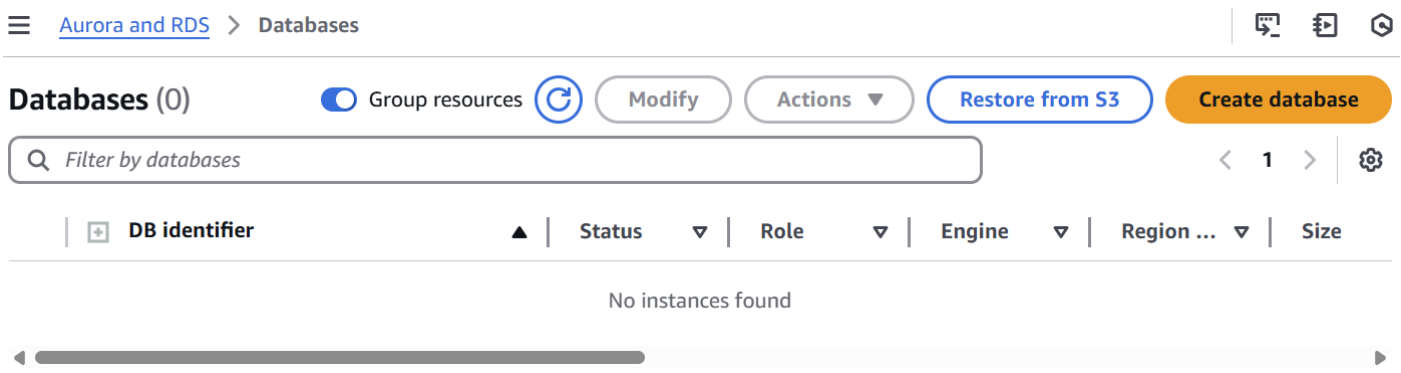
To get started let us jump into the lab environment and have a look at AWS RDS which is an AWS managed database service. To get to the RDS service either search for it or browse Services -> Database -> Aurora and RDS, as shown below.



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



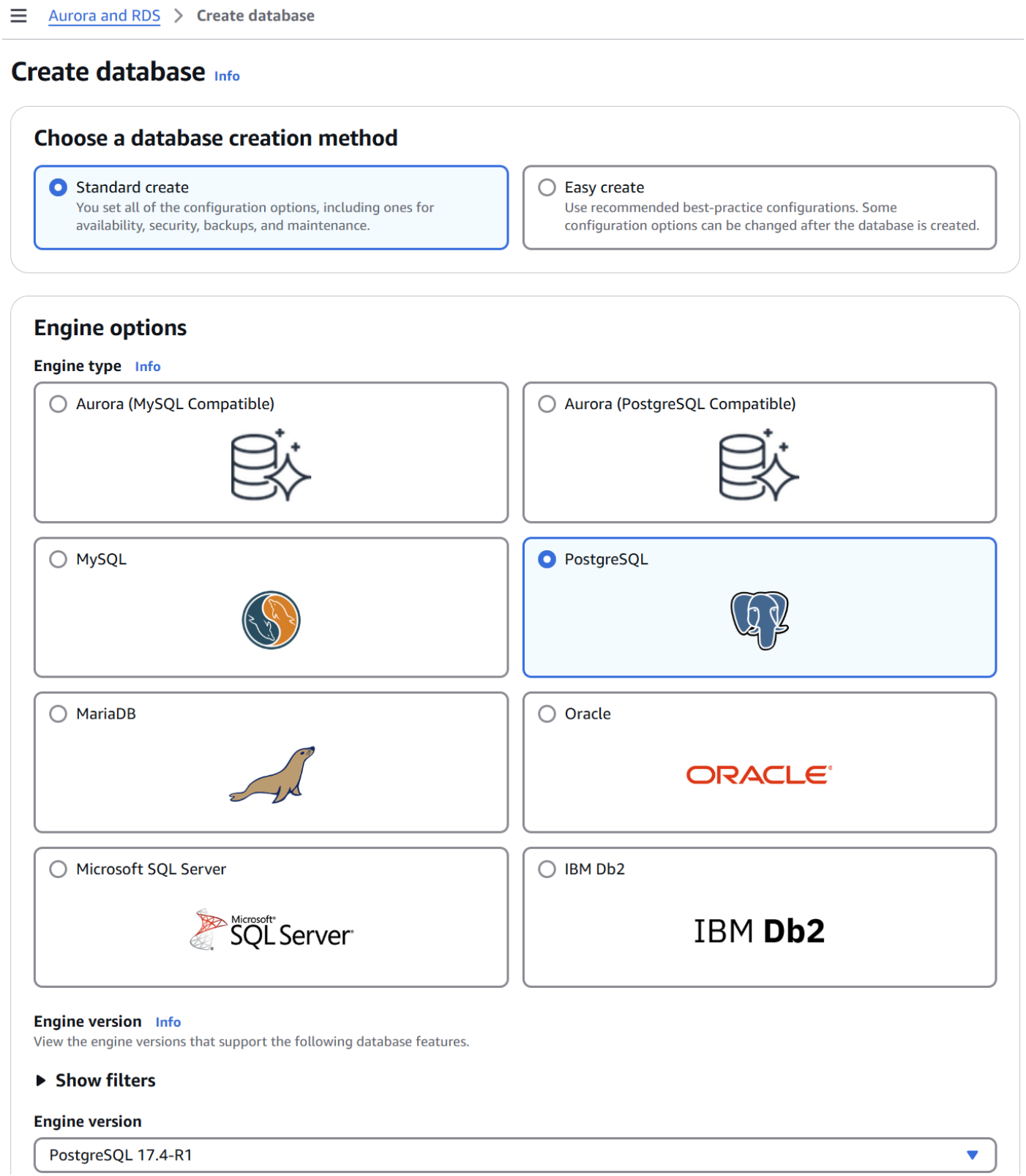
This page should appear familiar as it is very similar to the AWS EC2 instance page. Let us create a new database by clicking on the “Create Database” button.



Warning

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

We will create a standard database so select standard and PostgreSQL. We will use version 17, which is a fairly recent release.



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 options.

Templates

Choose a sample template to meet your use case.

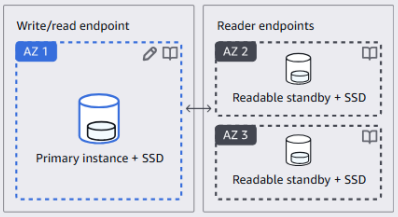
- ○ **Production**
  Use defaults for high availability and fast, consistent performance.
- ○ **Dev/Test**
  This instance is intended for development use outside of a production environment.
- ◉ **Free tier**
  Use RDS Free Tier to develop new applications, test existing applications, or gain hands-on experience with Amazon RDS. **Info**

Availability and durability

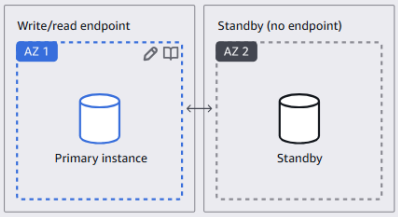
Deployment options Info

Choose the deployment option that provides the availability and durability needed for your use case. AWS is committed to a certain level of uptime depending on the deployment option you choose. Learn more in the Amazon RDS service level agreement (SLA).

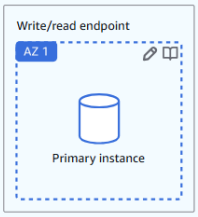
- ○ **Multi-AZ DB cluster deployment (3 instances)**
  Creates a primary DB instance with two readable standbys in separate Availability Zones. This setup provides:
  - 99.95% uptime
  - Redundancy across Availability Zones
  - Increased read capacity
  - Reduced write latency



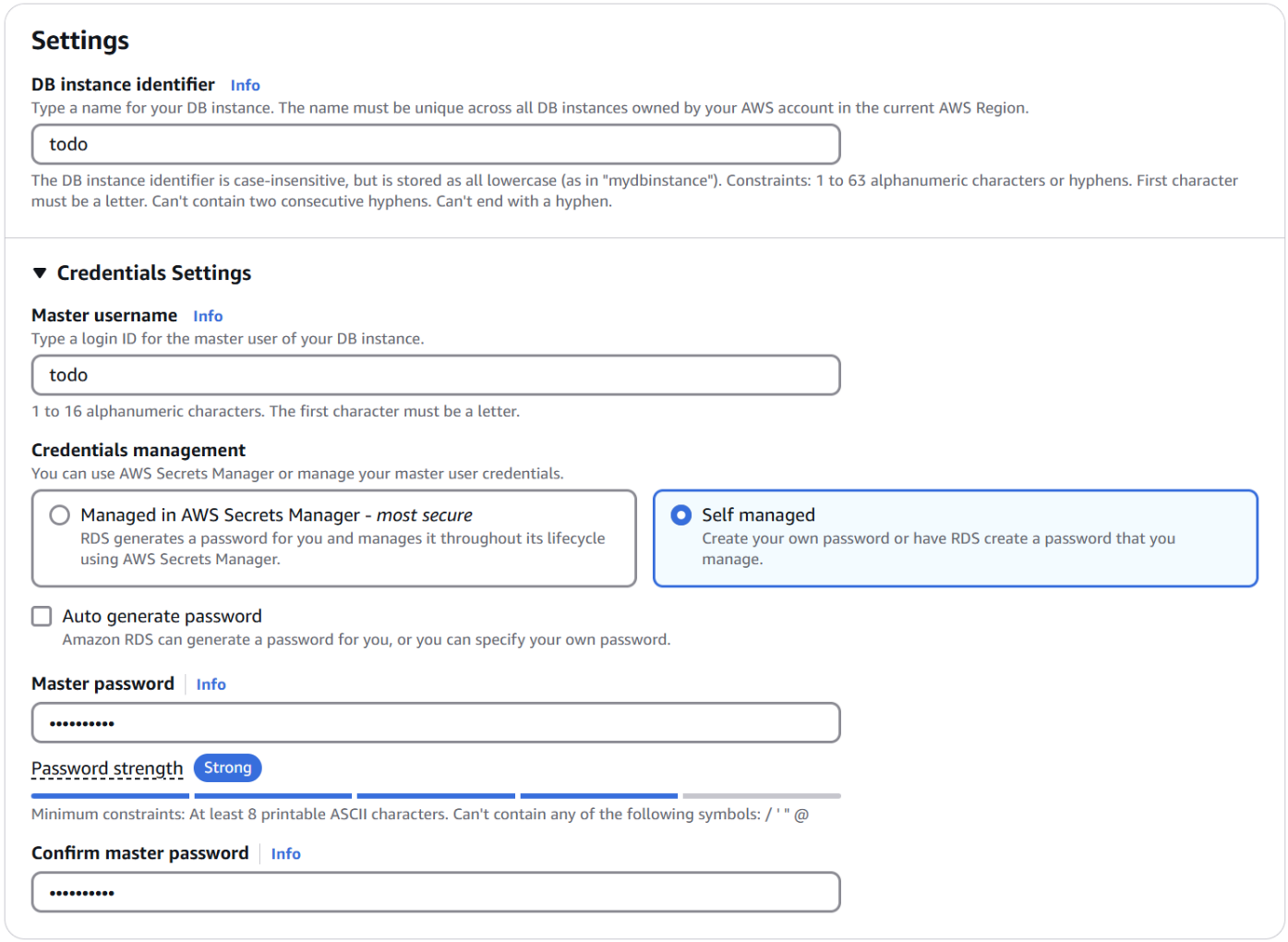
- ○ **Multi-AZ DB instance deployment (2 instances)**
  Creates a primary DB instance with a non-readable standby instance in a separate Availability Zone. This setup provides:
  - 99.95% uptime
  - Redundancy across Availability Zones



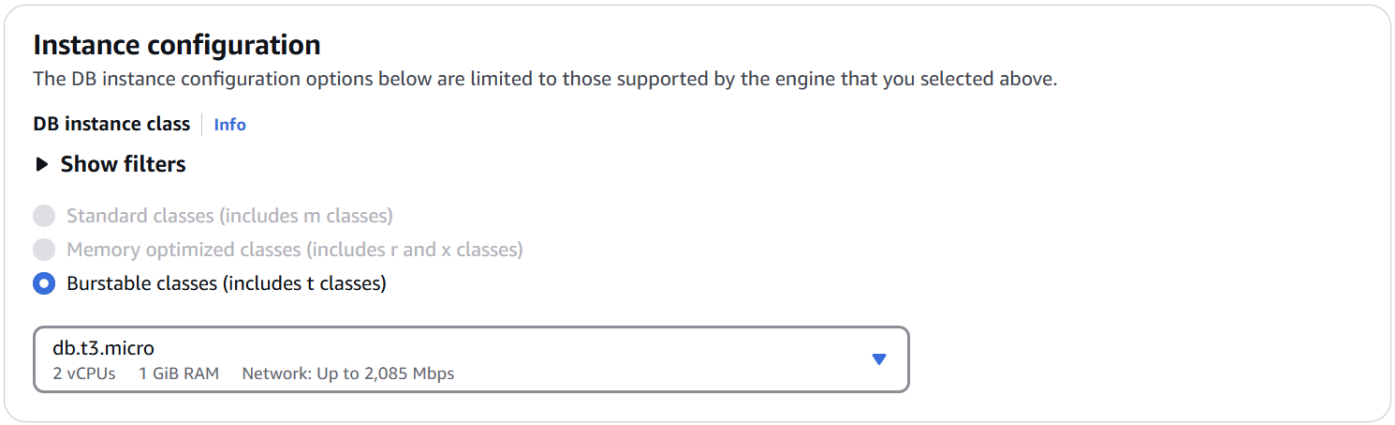
- ◉ **Single-AZ DB instance deployment (1 instance)**
  Creates a single DB instance without standby instances. This setup provides:
  - 99.5% uptime
  - No data redundancy



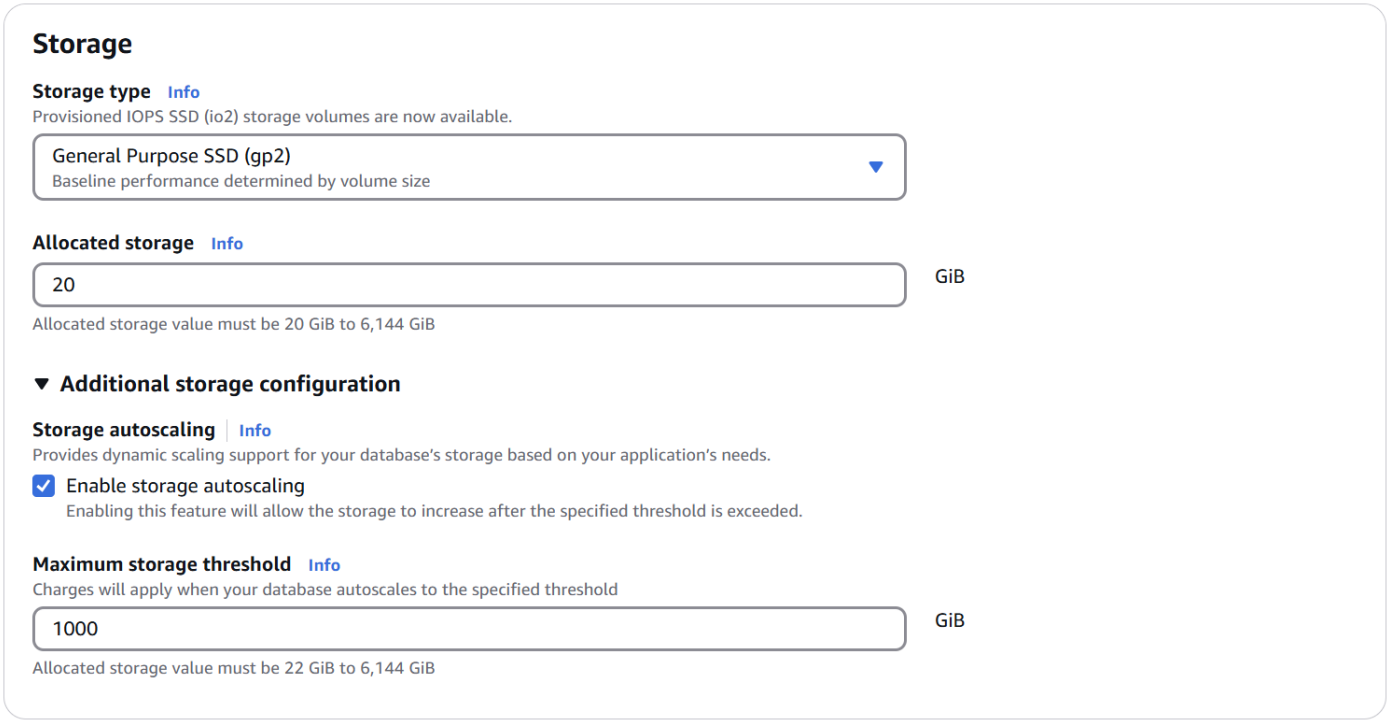
Now we need to name our database and create credentials to use when connecting from our application. Enter memorable credentials as these will be used later.



For exploring the process select t3.micro, which should be adequate for our needs.



For storage we will leave all the default options.



In connectivity we will leave the **Compute resource** and **VPC** with their default values. We need to make our instance publicly available. Usually you do ***not*** want to expose your databases publicly and, would instead, have a server sitting in-front (e.g. an API, application or web server). For our learning purposes though we are going to expose it directly just like we did with our EC2 instances earlier in the course.

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

DB subnet group Info

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

default

Public access Info

- Yes
  RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.
- No
  RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

- Choose existing
  Choose existing VPC security groups
- Create new
  Create new VPC security group

Existing VPC security groups

Choose one or more options

default

Availability Zone Info

No preference

RDS Proxy

RDS Proxy is a fully managed, highly available database proxy that improves application scalability, resiliency, and security.

- Create an RDS Proxy Info
  RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. RDS Proxy has additional costs. For more information, see Amazon RDS Proxy pricing.

Certificate authority - *optional* Info

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-rsa2048-g1 (default)
Expiry: May 26, 2061

If you don't select a certificate authority, RDS chooses one for you.

▼ Additional configuration

Database port Info

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

5432

We will leave the authentication as password based and monitoring with its default values. We need to expand the “Additional configuration” section. Fill in the “Initial Database Name” field as “todo”, this will automatically create the database to which our todo application expects to connect.

Additional configuration

Database options, encryption turned off, backup turned off, backtrack turned off, maintenance, CloudWatch Logs, delete protection turned off.

Database options

Initial database name Info

todo

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

DB parameter group Info

default.postgres17

Option group Info

default:postgres-17

Backup

☐ Enable automated backups
Creates a point-in-time snapshot of your database

Encryption

☐ Enable encryption
Choose to encrypt the given instance. Master key IDs and aliases appear in the list after they have been created using the AWS Key Management Service console. Info

Maintenance

Auto minor version upgrade Info

☑ Enable auto minor version upgrade
Enabling auto minor version upgrade will automatically upgrade to new minor versions as they are released.
The automatic upgrades occur during the maintenance window for the database.

Maintenance window Info

Select the period you want pending modifications or maintenance applied to the database by Amazon RDS.

○ Choose a window
◉ No preference

Deletion protection

☐ Enable deletion protection
Protects the database from being deleted accidentally. While this option is enabled, you can't delete the database.

Now we can click create database, 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, db.t3.micro or db.t4g.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.

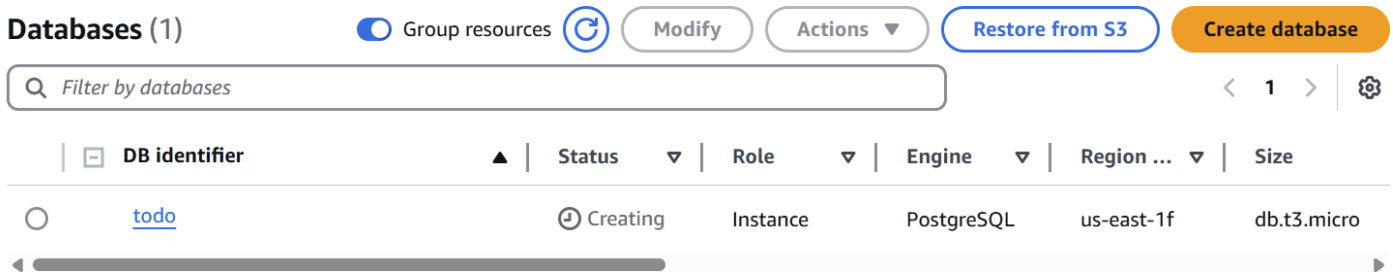
When your free usage expires or if your application use exceeds the free usage tiers, you simply pay standard, pay-as-you-go service rates as described in the Amazon RDS Pricing page.

ⓘ 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.

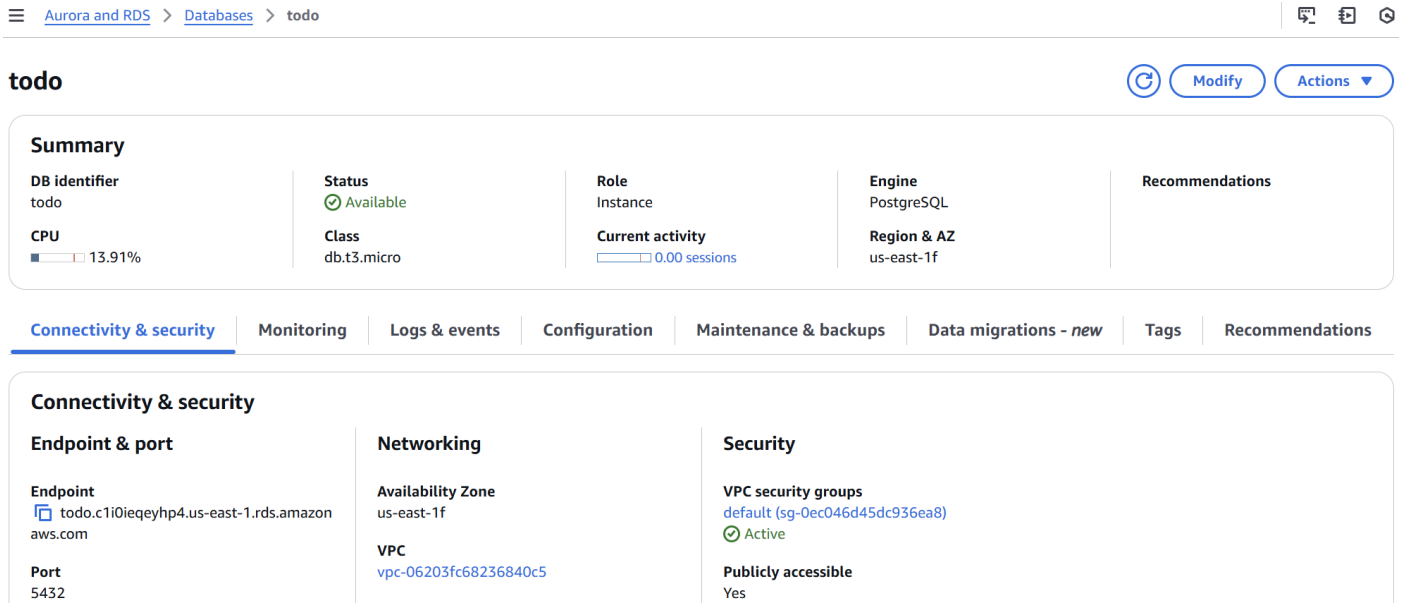
Cancel **Create database**

It may take several minutes to create. If we had selected to enable automated backups, the database would do an initial backup when it is created.

AWS will suggest add-ons for the newly created database. The suggested add-ons are useful features for a production environment. We do not need them for the purposes of this practical.



When the database has been 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.



4 RDS Database with Terraform

Now would be a good time to browse the documentation for the RDS database in Terraform. You will want to get practice at reading and understanding Terraform documentation.
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_instance

Using our manual configuration, we can come up with a resource with the appropriate parameters as below:

```
   » 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 = "17"
  instance_class = "db.t3.micro"
  db_name = "todo"
  username = local.database_username
  password = local.database_password
  parameter_group_name = "default.postgres17"
  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"
  }
}
```

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 briefly contrast it to manual deployment via EC2.

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.[2]

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 out of our containerised environment.

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

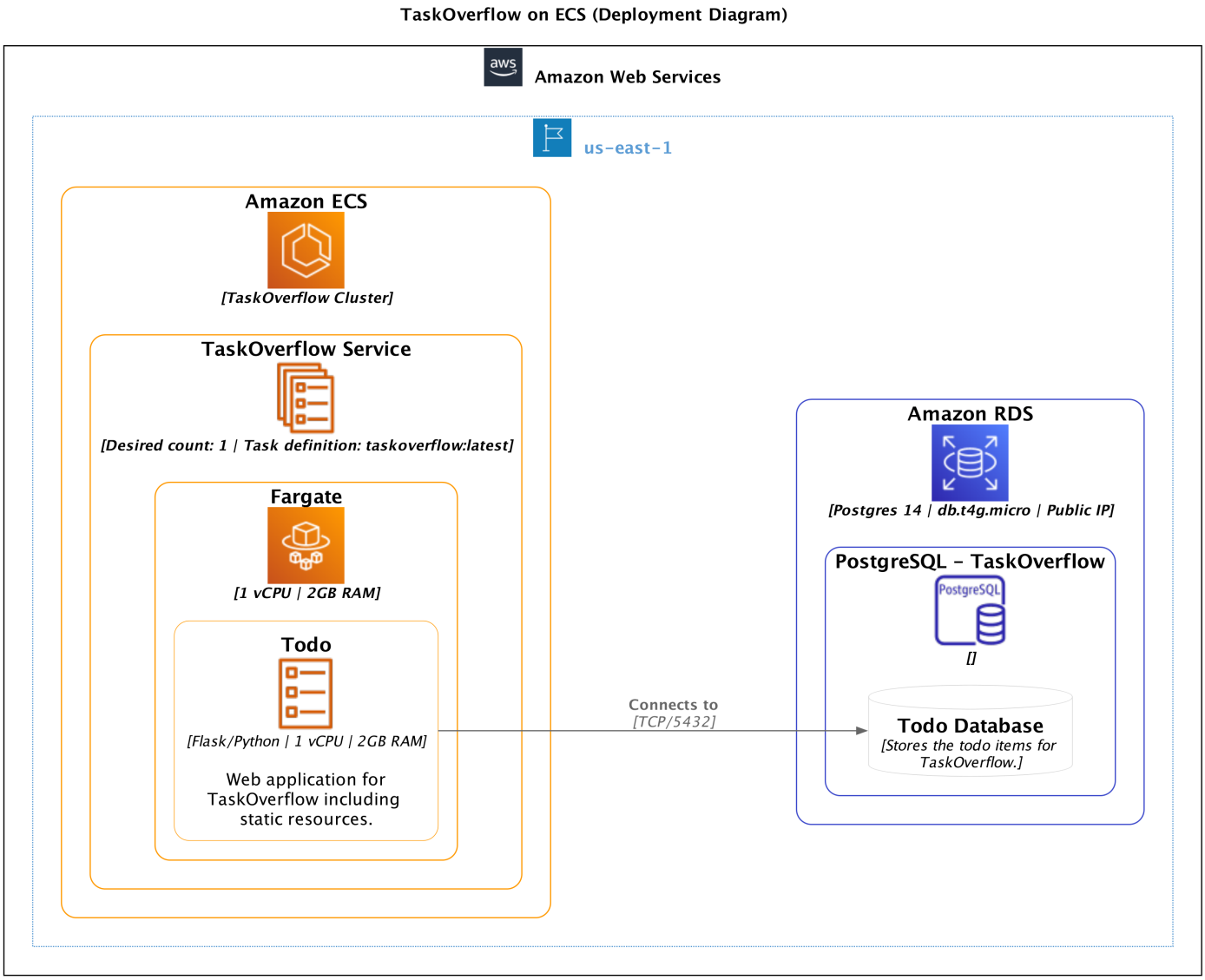
```
» cat main.tf
locals {
  image = "ghcr.io/csse6400/taskoverflow:latest"
  database_username = "administrator"
  database_password = "foobarbaz" # this is bad
}
```

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. Like when creating the database from the AWS console, this may take several minutes. Once the database has been created, go to the AWS console and check its status and details.

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.

[2] If you are interested, the source code is available on GitHub `https://github.com/csse6400/practical`

5.2 ECS Deployment



ECS mimics a similar environment as Docker Compose but as an AWS service.

To start off we need to get some information from our current AWS environment so that we can use it later. Add the code below to fetch the IAM role known as LabRole. It is a super user in the Learner Lab environments which can do everything you can do through the AWS Console. We will also fetch 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 its 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 equivalen 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 similar 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 on which to run the container. This is functionally similar to an auto-scaling group, as described in the Distributed Systems I lecture [2]. 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 `terraform apply` command, it should provision an ECS cluster with a service that will then create one ECS container based on our task description.

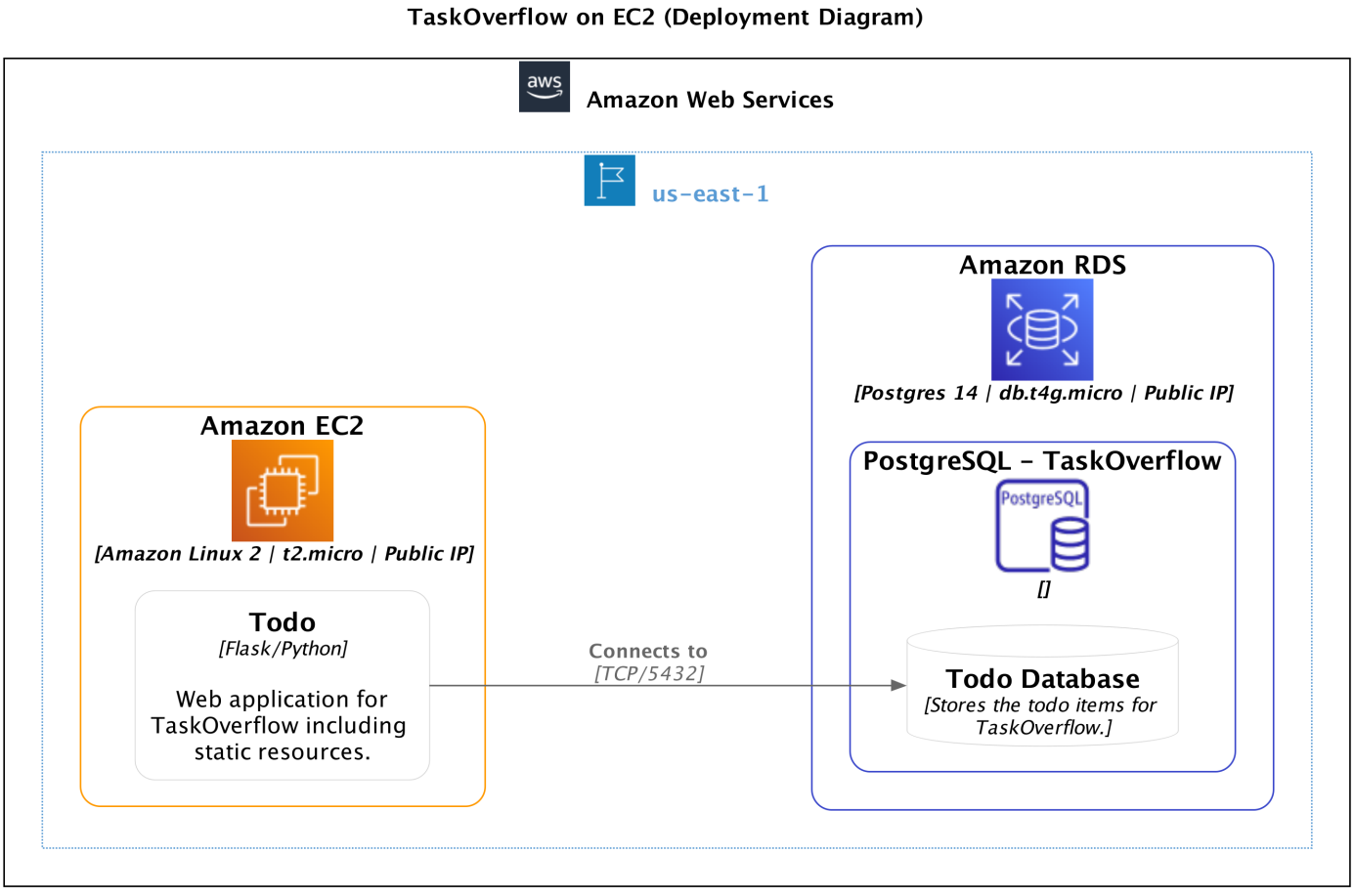
Note that we are doing something a bit weird in this deployment. Normally ECS expects multiple instances of containers, so it naturally expects a load balancer. This makes it difficult for us to discover the public IP of our single instance using Terraform. Instead, you will need to use the AWS Console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

5.3 EC2 Deployment

Aside

The deployment diagram below is what it would look like, if we deployed our application to an EC2 instance. As you can see, ECS provides us with features to manage services and tasks for us.



5.4 EKS / K8S

Amazon Elastic Kubernetes Service (EKS) is a platform to run Kubernetes[3] (K8S) clusters. We recommend, when you have time, that you look at Kubernetes as it is widely used in industry.

6 Hosting TaskOverflow Images

When we last deployed a container on AWS, we used an existing hosted image. Now, we will be developing our own image, so we will need a mechanism to host the image. For this, we will use AWS ECR, Docker, and Terraform. AWS ECR is the Elastic Container Registry. It is a container registry like DockerHub or GitHub. We can use it to host our image. The steps below use Terraform to

1. create an ECR repository for our image,
2. build our Docker image, and
3. push our Docker image.

> **Info**
>
> This is a non-standard process. As you may have seen in the DevOps tutorial, we would ordinarily like our code commits to trigger a CI/CD pipeline which builds the images.
>
> If you would like, you can use GitHub actions to build and push your container to the GitHub container registry and authenticate when you pull the image. However, using ECR simplifies the process, despite the oddities introduced by having a non-persistent ECR repository.

Getting Started

1. Using the GitHub Classroom link for this practical provided on Edstem, create a repository to work within.
2. Install Terraform, if it is not already installed, as it will be required again this week and in later weeks.
3. Start your Learner Lab and copy the AWS Learner Lab credentials into a credentials file in the root of the repository.

What's New We are starting again with our todo application from roughly where we left off in the week 3 practical. We have added a new directory `todo/app` that has the static HTML files for the TaskOverflow website and added a route to serve these files. We have also created a production version of the server that uses gunicorn, the `bin` directory is used by this image. Our original Docker image is now in `Dockerfile.dev`.

We will setup our initial Terraform configuration. Note that now we introduce a new required provider. This provider is for Docker.

```
» cat main.tf
```

[3] https://kubernetes.io/

```
terraform {
   required_providers {
      aws = {
         source = "hashicorp/aws"
         version = "~> 5.0"
      }
      docker = {
         source = "kreuzwerker/docker"
         version = "3.0.2"
      }
   }
}

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

As with our AWS provider, when we initially configure the provider, we want to authenticate so that we can later push to our registry using the Docker provider. We will use the `aws_ecr_authorization_token` data block to get appropriate ECR credentials for Docker.

```
   » cat main.tf
data "aws_ecr_authorization_token" "ecr_token" {}

provider "docker" {
  registry_auth {
    address = data.aws_ecr_authorization_token.ecr_token.proxy_endpoint
    username = data.aws_ecr_authorization_token.ecr_token.user_name
    password = data.aws_ecr_authorization_token.ecr_token.password
  }
}
```

We need to use Terraform to create an ECR repository to push to.

```
   » cat main.tf
resource "aws_ecr_repository" "taskoverflow" {
  name = "taskoverflow"
}
```

The URL for containers in the ECR follow the format below:
`{ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}`
Remember — to push to a container registry we need a local container whose tag matches the remote URL. We could then create and push the container locally with:

```
docker build -t {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME} .
docker push {ACCOUNT_ID}.dkr.ecr.{REGION}.amazonaws.com/{REPOSITORY_NAME}
```

However, it would be easier if we could build and push this container from within Terraform. We can use the Docker provider for this.

```
   » cat image.tf

resource "docker_image" "taskoverflow" {
  name = "${aws_ecr_repository.taskoverflow.repository_url}:latest"
  build {
    context = "."
  }
}

resource "docker_registry_image" "taskoverflow" {
  name = docker_image.taskoverflow.name
}
```

Notice that we are able to utilise the output of the ECR repository as the URL which resolves to the correct URL for the image.

If you execute `terraform plan`, it will probably report an inconsistent dependency. This is because we have added a new provider and its dependency needs to be added to the lock file. Execute `terraform init -upgrade` to do this.

You can now `terraform apply` to push the container to the registry. Note that the Docker Engine (daemon) must be running so Terraform can talk to it.

References

[1] B. Webb, "Infrastructure as code," February 2023. https://csse6400.uqcloud.net/handouts/iac.pdf.

[2] B. Webb and R. Thomas, "Distributed systems I slides," March 2025. https://csse6400.uqcloud.net/slides/distributed1.pdf.