

Task 1: Deploy the Three Tier Application from Terraform

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Outputs:

```
public_instance_public_ip = "13.56.195.118"
rds_db = "demodb.cpewg24csbd6.us-west-1.rds.amazonaws.com"
vpc-id = "vpc-0de1cf3126b70c3e4"
rohan@junie lab-database-juniemariam-main %
```

Task 2: Connect to Your Database and Create the User Table

```
CREATE TABLE
user_database=> \dt
                List of relations
 Schema | Name  | Type  | Owner
-----+-----+-----+-----
 public | users | table | replica_postgresql
(1 row)

user_database=> INSERT INTO users (name, email) VALUES ('your name', 'your email');
INSERT 0 1
user_database=> \dt
                List of relations
 Schema | Name  | Type  | Owner
-----+-----+-----+-----
 public | users | table | replica_postgresql
(1 row)


user_database=>
```


Task 3: Update the Backend App with the Values of Your Database





```
Server errorubuntu@ip-172-16-102-247:~$ curl -X POST http://172.16.101.10:8080/api/users -H "Content-Type: application/json" -d '{"name": "test_user", "email": "test_user@gmail.com"}'
{"id":2,"name":"test_user","email":"test_user@gmail.com"}ubuntu@ip-172-16-102-247:~$
```

```
ubuntu@ip-172-16-102-247:~$ curl http://172.16.101.10:8080/api/users
[{"id":1,"name":"your name","email":"your_email"}, {"id":2,"name":"test_user","email":"test_user@gmail.com"}]ubuntu@ip-172-16-102-247:~$
```

Task 4: Add a Replica to Your Database

Databases (2) ☒ Group resources  Modify Actions ▼ Restore from S3 Create database

< 1 > 

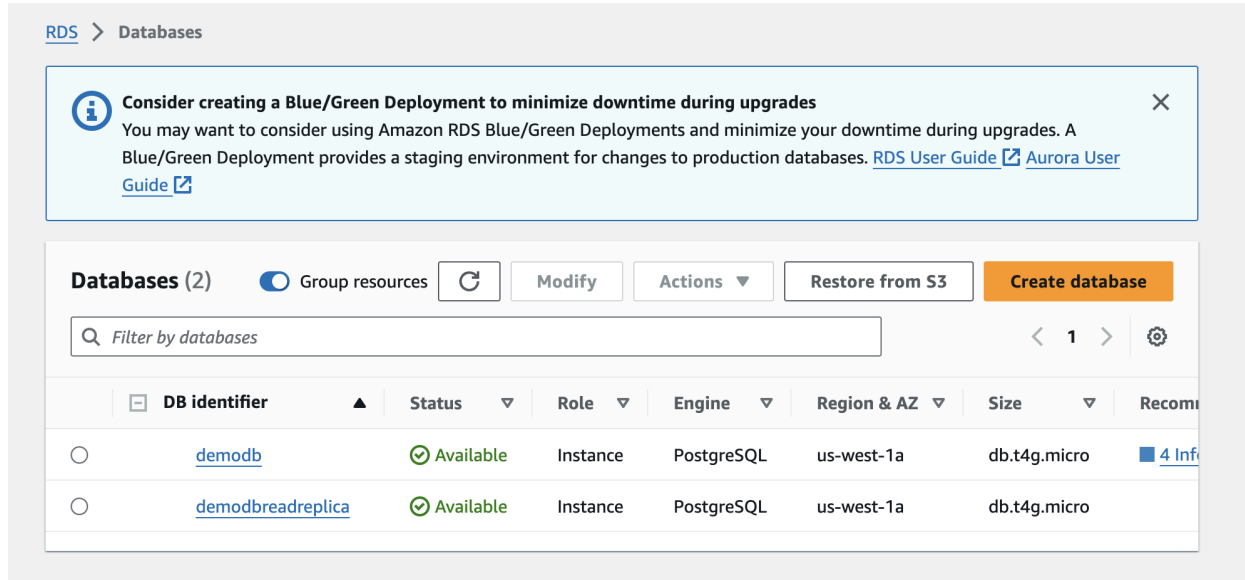
	 DB identifier ▲	Status ▼	Role ▼	Engine ▼	Region & AZ ▼	Size ▼	Recon
<input type="radio"/>	 demodb	✔ Available	Primary	PostgreSQL	us-west-1a	db.t4g.micro	 4 In
<input type="radio"/>	 demodbreadreplica	✔ Available	Replica	PostgreSQL	us-west-1a	db.t4g.micro	

Apply complete! Resources: 1 added, 1 changed, 1 destroyed.

Outputs:

```
public_instance_public_ip = "184.169.190.203"
rds_db = "demodb.cpewg24csbd6.us-west-1.rds.amazonaws.com"
rds_read_replica = "demodbreadreplica.cpewg24csbd6.us-west-1.rds.amazonaws.com"
vpc-id = "vpc-0de1cf3126b70c3e4"
rohan@junie lab-database-juniemariam-main %
```

Task 5: Promote your database to primary



The screenshot shows the Amazon RDS Databases console. At the top, there is a notification banner that reads: "Consider creating a Blue/Green Deployment to minimize downtime during upgrades. You may want to consider using Amazon RDS Blue/Green Deployments and minimize your downtime during upgrades. A Blue/Green Deployment provides a staging environment for changes to production databases. [RDS User Guide](#) [Aurora User Guide](#)". Below the notification, the console displays a list of databases. The list has columns for DB identifier, Status, Role, Engine, Region & AZ, Size, and Recommendations. Two databases are listed: "demodb" and "demodbreadreplica", both with a status of "Available", role of "Instance", engine of "PostgreSQL", region of "us-west-1a", and size of "db.t4g.micro".

DB identifier	Status	Role	Engine	Region & AZ	Size	Recommendations
demodb	Available	Instance	PostgreSQL	us-west-1a	db.t4g.micro	4 Info
demodbreadreplica	Available	Instance	PostgreSQL	us-west-1a	db.t4g.micro	

Knowledge Check (2 pts)

1. The architecture of this three tier app is very brittle, what can be done to increase the resiliency of the data and the elasticity of the application without switching to a serverless design?

To improve the resiliency of a three-tier application without switching to a serverless architecture, there are a few practical steps worth considering. Load balancing can ensure that traffic is spread evenly, so no single server takes too much of the load, reducing the chance of downtime. Redundancy, by having multiple instances in different availability zones, adds a layer of protection—if one instance fails, others can seamlessly take over.

Auto-scaling is another way to ensure the app adjusts to demand automatically, especially during traffic spikes, which helps maintain performance without manual intervention. Caching solutions like Redis can speed up access to frequently needed data, making the overall system more efficient and taking pressure off the database.

When it comes to the database, using replication or sharding can ensure it remains responsive and reliable, even as data grows. Keeping centralized logging and monitoring in place allows for quick identification of any issues that arise. Regular backups will safeguard data, and integrating the circuit breaker pattern can help isolate failures, ensuring that one issue doesn't cause a chain reaction across the system. By applying these strategies, the app will be much better equipped to handle unexpected challenges and remain stable over time.

2. In the self study we explored multi-tiered serverless architecture. In what situations is it advisable to use serverless architecture versus traditional customer managed architecture?

To decide between serverless and traditional customer-managed architecture really depends on what the project requires. Serverless architecture works exceptionally well for applications that experience fluctuating traffic. It adjusts automatically to match demand, which can lead to cost savings during slower periods. For projects where speed is crucial, serverless allows developers to concentrate on coding rather than being bogged down by server management. This is especially useful for startups and small teams that aim to launch quickly and iterate on their concepts.

Furthermore, serverless aligns nicely with a microservices strategy, where applications are divided into smaller, independently deployable services. This approach simplifies managing and deploying specific functions without affecting the entire application. Serverless is also ideal for event-driven applications that depend on user interactions, such as responding to HTTP requests or processing file uploads. In these situations, serverless can handle everything efficiently without needing a server to be constantly active. Also, for smaller projects with low or sporadic usage, serverless can be more budget-friendly since you only pay for what you use. It is an excellent choice for rapid experimentation, enabling teams to build prototypes or test innovative ideas without needing a large upfront investment in infrastructure.

Conversely, traditional customer-managed architecture excels in different contexts. For applications with stable and predictable workloads, traditional setups are often more economical because they allow for better resource optimization to meet ongoing demand. When strict security or compliance needs exist, having direct control over your servers can offer more assurance regarding the environment and the data being managed.

Organizations with legacy systems might also find it more straightforward to continue with traditional architecture rather than refactor existing applications for a serverless approach. For complex applications with numerous interdependencies or long-running processes, traditional setups provide greater flexibility and control over resource management. If fine-tuning performance is critical—like in gaming or high-performance computing—direct control over infrastructure becomes vital. Lastly, some organizations may be concerned about getting locked into a single vendor with serverless solutions, making traditional architectures more attractive due to their flexibility in selecting or switching vendors.

In conclusion, the choice between serverless and traditional architecture should be guided by the project's specific needs and constraints. By considering factors such as workload characteristics, compliance requirements, and the level of control desired, it becomes easier to identify the most suitable approach.