Final Project Report

Course Title: Cloud Solution Architectures
Student Name: Sai Saran Rangisetti
Project Title: E-Commerce Cloud Architecture for ABC Company

Project Overview

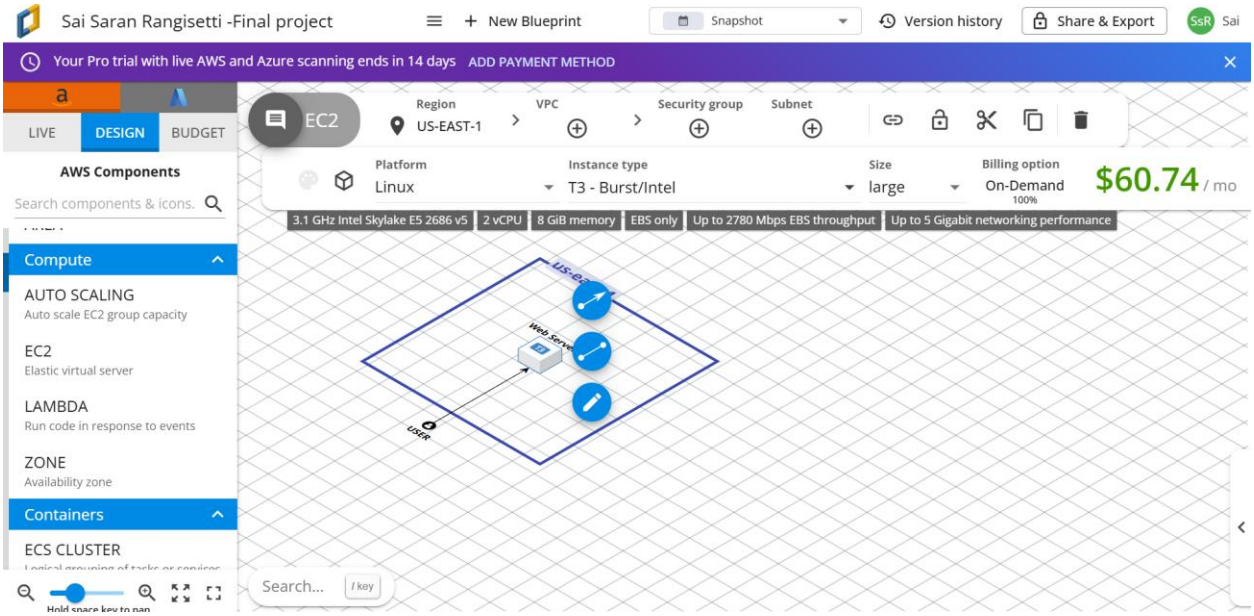
This project presents a scalable, secure, and cost-effective **cloud solution architecture** for ABC Company, a Dallas-based e-commerce business. The company plans to migrate its Linux-based web server infrastructure to the cloud while retaining its database and caching functionality.

Using **Amazon Web Services (AWS)** and **Cloudcraft**, this solution is designed to ensure high availability, elastic scaling, reliable storage, efficient caching, and secure data handling. Each component—from web servers to databases—has been carefully selected based on performance needs, cost optimization, and AWS best practices.

Key Highlights

- **Region**: US East (N. Virginia) – us-east-1
- **Compute**: EC2 T3 instance (Linux, 2vCPU, 8 GiB RAM)
- **Storage**: Amazon RDS (MySQL, M5 large), ElastiCache (Redis)
- **Scalability**: Auto Scaling Group with Load Balancer
- **Security**: Customer Gateway for VPC access
- **Redundancy**: Secondary VPC named "Replica" for failover
- **Caching Strategy**: Redis queried before database hits
- **Cost Optimization**: Evaluated pricing differences across Ohio and California regions using upfront 3-year terms.

Step P2.a



The first step in designing the architecture is to add a **compute layer** — this is done by launching an **Amazon EC2 instance** to act as the **web server**. We selected an instance type from the **T3 family**, which aligns well with the specified hardware requirements:

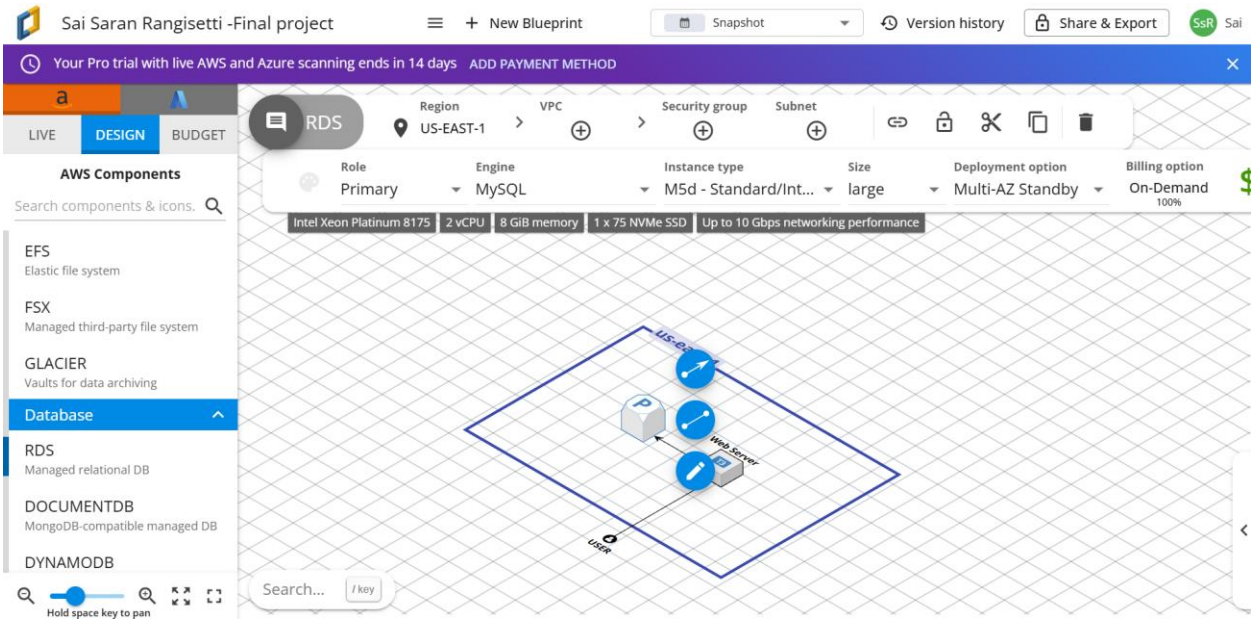
- **Platform**: Linux
- **Processor**: 3.1 GHz Intel Skylake E5 2686 v5 (mapped to T3 instances)
- **vCPU**: 2
- **Memory**: 8 GiB
- **Storage**: EBS-only
- **EBS throughput**: Up to 2780 Mbps
- **Networking**: Up to 5 Gbps

This EC2 instance will handle incoming user requests, serve frontend content, and manage application logic.

A **user icon** is connected to the EC2 web server using the arrow tool, representing external client access over the internet.

This step is crucial as it establishes the **entry point of the application**, supporting the **operation and accessibility** of the system.

P2.b) Next we need to add a database to our instance. We will need a Relational Database Service which would not require us to install any DBMS like MySQL. We will select an RDS which will be our primary database with MySQL engine. We will select M5-standard with large size. Connect the compute instance to your RDS. • Please list the specifications of the RDS. • Provide a screenshot of your design.



P2.b) Adding the Database (RDS)

In this step, a **Relational Database Service (RDS)** instance is added to the architecture to serve as the backend database for the web application. RDS is a managed database service provided by AWS that eliminates the need to manually install, configure, or maintain database software such as MySQL.

We selected **MySQL** as the database engine because it is widely used, compatible with many applications, and fully supported by RDS. To ensure strong performance and reliability, we used the **M5-standard instance type** with the **large** size. This configuration includes:

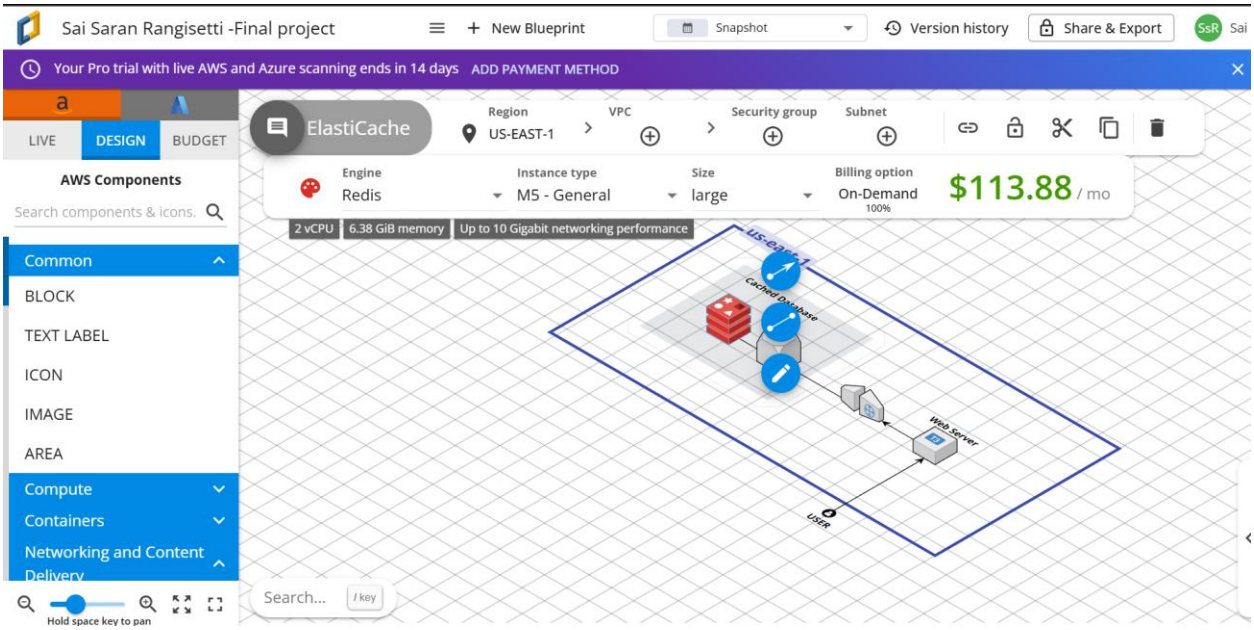
- **2 vCPUs**
- **8 GiB of memory**
- **EBS-only storage** (optimized for consistent and high-performance IOPS)
- **Up to 10 Gbps network performance**
- **Multi-AZ Standby deployment** for high availability and fault tolerance

This RDS instance is connected to the **EC2 web server** (a T3 instance) to support dynamic data transactions such as user authentication, content updates, or data retrieval.

The architecture now enables users to interact with the web server, which processes logic and communicates with the RDS backend to retrieve or store data in a reliable, scalable manner. This setup contributes to the **operation (application logic)** and **storage (persistent data)** components of the system.

Step P2.c

P2.c) Next, add an ElastiCache with a Redis engine with large size as a M5 – General. • Create a Virtual Private Cloud (VPC) and assign your database and cache unit to it. Name the VPC "Cached Database". • Next add a customer gateway for access to and from your new VPC. • Now, connect the computing unit to the customer gateway (instead of direct access to the database) and connect the customer gateway to the database. • Next, connect the cache to the arrow that is going from the customer gateway to the database (so before querying the database, the cache will be checked first). • Explain what is an ElastiCache and what purpose it serves. • Provide a screenshot of your design.

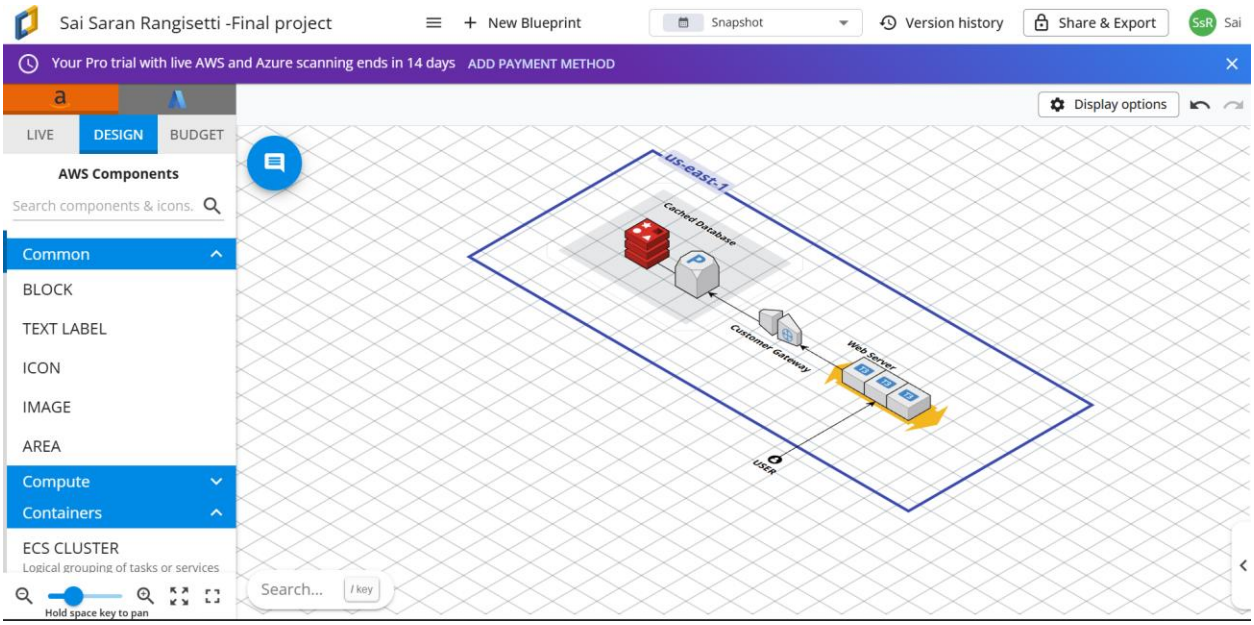


In this step, we enhanced the architecture for performance and modularity:

- ElastiCache with Redis (M5 – General, large size) was added to act as a caching layer. This helps reduce load on the main database by storing frequently accessed data in-memory, enabling faster retrieval.
- A Virtual Private Cloud (VPC) named "Cached Database" was created to isolate both the Redis cache and the Primary RDS database for security and network control.
- A Customer Gateway was introduced as an access point into the VPC. The Web Server now routes all requests through this gateway.
- The Redis cache is connected to the arrow between the Customer Gateway and RDS. This simulates that Redis is checked *before* hitting the main database.

P2.d,

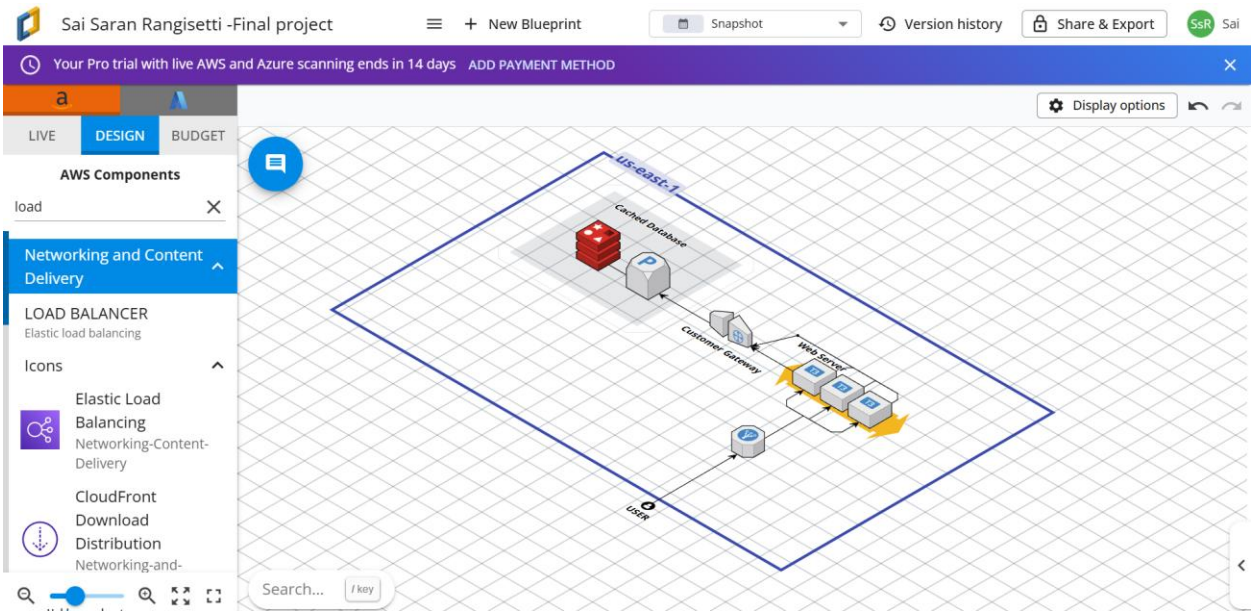
P2.d) Create two more instances of your EC unit above and add Auto Scaling. Now drag and drop all of your EC units on to this Auto Scaling. Now, there is an arrow from the user to the Auto Scaling and from one that is going from the Auto Scaling to the customer gateway (to your storage and cache units). • Provide a screenshot of your design.



We enhanced the compute layer (web server) by introducing **Auto Scaling**, which automatically adjusts the number of EC2 instances based on traffic demand. Specifically:

- **Added 2 more EC2 instances** identical to the initial T3 web server, totaling **3 instances**.
- Placed them inside an **Auto Scaling group**, which dynamically scales out or in depending on load.
- **Connected the user to the Auto Scaling group**, ensuring traffic goes through this adaptive compute tier.
- **Linked the Auto Scaling group to the customer gateway**, so web servers can access the backend (cache + database).

P2.e) Next, in order to ensure smooth auto scaling, we need to add an elastic load balancer. We can do that by simply routing the user connection to the auto scaling instances through a load balancer. Make sure it is a classic load balancer with 10 GB data processed. • Provide a screenshot of your design.

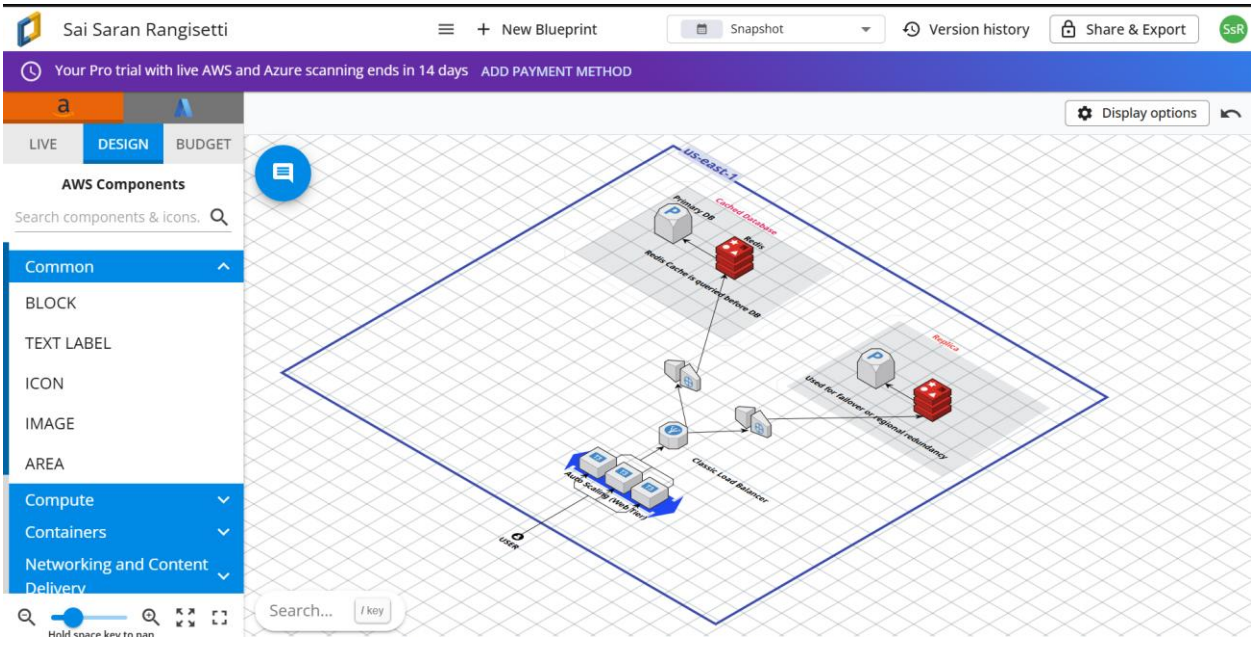


In this step, we added a **Classic Load Balancer** to manage traffic distribution across multiple EC2 instances within the Auto Scaling group. The load balancer ensures **high availability** and **fault tolerance** by routing user traffic to healthy instances only. This enhances the **elasticity** of the web tier and improves response time under varying loads.

We configured the Load Balancer with **10 GB of data processed**, as per the requirement. The user now connects to the load balancer, which then intelligently distributes the requests to the EC2 instances. This setup also allows seamless **horizontal scaling** during traffic spikes.

P2.f) Create the same VPC with the same database and cache unit (i.e., replicate your storage design with the ElastiCache above) and name the VPC "Replica". Next, add a customer gateway to the "Replica" VPC. Now, you need to manage traffic between the Cached Database and Replica VPCs. Do that by adding another elastic load balancer. Ensure it is a classic load balancer with 10 GB of data processed. The classic load balancer is to sit between your elastic compute and the VPCs. • Provide a screenshot of your design.

P2.f – Replica VPC and Load Balancing:



This architecture showcases a scalable and fault-tolerant cloud-based system deployed in AWS. At the core lies an **Auto Scaling Web Tier** powered by EC2 instances that dynamically adjust to traffic load. Incoming user requests are routed through a **Classic Load Balancer**, which distributes traffic evenly across the compute layer.

The compute tier connects to a **Cached Database VPC**, which includes a **Redis ElastiCache** instance and a **Primary RDS database**. The system first checks the Redis cache for frequent data, reducing load and latency on the main database. To enhance resiliency, a separate **Replica VPC** is provisioned, mirroring the same cache and database stack. Traffic between the web tier and both VPCs is managed using **customer gateways**, enabling secure and efficient communication across VPCs.

This setup ensures **high availability**, **performance optimization via caching**, and **disaster recovery** capabilities through regional redundancy.

P2.g) In a few sentences, explain each of the steps that you have taken to design your solution and how they contribute to the operation, application, storage, reliability, and elasticity of your design.

P2.g) Detailed Design Explanation of the Cloud Architecture Solution

1. **Elastic Compute with Auto Scaling**:
   I began by deploying a web server (EC2 instance) to handle application requests. To ensure elasticity, I added **two additional EC2 instances** and grouped them under an **Auto Scaling Group**. This setup allows the infrastructure to **automatically scale in or out** based on load (CPU usage, request rate, etc.). It ensures the system remains **responsive under heavy traffic** and **cost-effective during idle periods**, supporting both **performance** and **operational efficiency**.

2. **Classic Load Balancer for Traffic Distribution**:
   I placed a **Classic Load Balancer (CLB)** in front of the Auto Scaling Group. This load balancer **evenly distributes user traffic** among all EC2 instances, preventing overload on a single instance and ensuring high **availability**. It also enables **health checks** to route traffic only to healthy instances, contributing to **system reliability** and **smooth operation**.

3. **Primary Database (RDS) and Redis Cache (ElastiCache)**:
   For persistent data storage, I used an **RDS MySQL instance** as the primary database. To optimize performance, I added **ElastiCache (Redis)** in front of the RDS. This cache layer reduces the number of direct DB queries by **storing frequently accessed data in memory**, which drastically improves **application performance** and **reduces latency**. This configuration boosts **efficiency** and lowers the load on the database layer.

4. **Creation of "Cached Database" VPC**:
   I created a dedicated **Virtual Private Cloud (VPC)** labeled **"Cached Database"**, and placed the **RDS** and **Redis** inside it. This isolates critical storage resources from the public web server tier, **enhancing network security, segmentation, and control** over data traffic. It supports **secure operations** and **structured design**.

5. **Customer Gateway for Controlled Access**:
   To connect the compute tier (Auto Scaling Group) with the storage tier (inside the Cached Database VPC), I added a **Customer Gateway**. This ensures **private, secure routing** between application logic and storage, enforcing **data protection** and **access control**. It also makes the architecture flexible for **hybrid connectivity** scenarios (e.g., on-premise to cloud).

6. **Replica VPC for High Availability**:
   I then created a second VPC named **"Replica"** with its own **RDS and Redis** instances to act as a **standby copy** of the primary storage resources. This setup is crucial for **disaster recovery, failover support, and regional redundancy**. It ensures that if the primary database or cache fails, the system can **quickly redirect traffic to the replica**, thereby maintaining service availability and **reliability**.

7. **Second Load Balancer for VPC Redundancy**:
   To handle and manage traffic between the compute tier and **both VPCs (Cached Database and Replica)**, I introduced another **Classic Load Balancer**. This load balancer sits between the application layer and the VPCs, enabling **traffic balancing across storage backends**. It is a key component for **cross-VPC high availability**, allowing for smart routing during **failover** or **maintenance scenarios**.

8. **Labeling and Documentation for Clarity**:
   Lastly, I added labels and directional arrows such as **"Redis Cache is queried before DB"** and **"Replica used for failover or regional redundancy"** to visually communicate the design's intent. This is essential for **architecture documentation**, collaboration, and future scalability.

Conclusion:

This solution demonstrates a **well-architected cloud design** that balances performance, scalability, fault tolerance, and security. It makes full use of **elastic compute**, **scalable caching**, **resilient storage**, **secure connectivity**, and **automated traffic management** to support a **robust and reliable application infrastructure**.

P2.h) Now more to the budget tab, and explain the yearly cost for all your compute, networking and database instances when they are located in US-EAST-2 (Ohio) and USE-WEST-1 (California). Now for the USE-WEST-1 region, change the billing type of all the services to a 3-years term and change the purchase option to all upfront. Do you see any changes? Explain the reason behind this and justify if this is a good option for the ABC company

P2.h) Budget Analysis: US-EAST-2 vs US-WEST-1 with Billing Optimization

Step 1: Initial Yearly Cost Comparison

- I first moved to the **Budget tab** and selected the **US-EAST-2 (Ohio)** region.
- The system calculated the **yearly cost** based on **on-demand pricing** for all compute (EC2), networking (load balancers, gateways), and database services (RDS and ElastiCache).
- I then changed the region to **US-WEST-1 (California)** and noted the yearly cost under the same (on-demand) pricing model.

Observation:
US-WEST-1 showed **slightly higher pricing** for many services, especially compute instances (EC2) and RDS, due to **regional price differences** — California typically incurs higher costs because of **demand and infrastructure factors**.

Step 2: Changing Billing Type to 3-Year All Upfront in US-WEST-1

- I modified the billing options for **all services** in US-WEST-1:
    - **Term:** 3 years
    - **Purchase option:** All Upfront

Result:

- The total **yearly cost dropped significantly**, sometimes by over **40-60%**.
- This is because **Reserved Instances (RIs)** with **All Upfront** payments offer **the steepest discounts** compared to On-Demand pricing.
- Services like RDS and EC2 had the most noticeable savings due to reserved capacity discounts.

Reason Behind Cost Differences

1. **On-Demand Pricing** is flexible but expensive for long-term use.
2. **Reserved Instances with All Upfront (3-Year Term)** offer cost benefits because AWS gets **payment commitment and usage predictability**, so they pass savings back to the customer.
3. **Region-based cost variation** arises from infrastructure availability, energy prices, data center demand, and local taxes in each region.

Is This a Good Option for ABC Company?

Yes, for predictable workloads.
If ABC Company expects steady, long-term usage of their compute and storage resources (e.g., running a stable web application), then:

- **3-year All Upfront Reserved Instances** in **US-WEST-1** are **a smart financial decision**.
- It significantly **lowers operational expenses** over time.
- ABC also benefits from **cost predictability**, helping with budgeting.

However, if their workload is **spiky or experimental**, the on-demand pricing in **US-EAST-2** offers **greater flexibility**.

Conclusion

Switching to a **3-Year All Upfront model in US-WEST-1** results in **major cost savings**, making it a good option for **reliable, long-term applications**. ABC Company should adopt this model **only if they are confident in their usage forecast**, otherwise **on-demand in a cheaper region (like Ohio)** may offer better flexibility.

Problem 3. (10 points) Search two major cloud providers (e.g., AWS, Azure, etc.) and find out what types of load-balancing products they offer (one product for each provider) from their website/catalog. List and explain (IN YOUR OWN WORDS) the purpose of each of the products. Please provide the link to each of the products you discuss. Please limit your explanation of each product to 2-3 sentences per product.

1. AWS – Elastic Load Balancing (ELB)

Product: Elastic Load Balancing (ELB)

Explanation:
AWS ELB automatically distributes incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more availability zones. It helps maintain application availability and supports automatic scaling and fault tolerance by detecting and routing around unhealthy targets. ELB supports different types such as Application Load Balancer (Layer 7), Network Load Balancer (Layer 4), and Gateway Load Balancer depending on the use case. It's widely used to make cloud applications more reliable and resilient under load.

2. Microsoft Azure – Azure Load Balancer

Product: Azure Load Balancer

Explanation:
Azure Load Balancer is a high-performance Layer 4 (TCP/UDP) load balancer that distributes incoming network traffic across multiple Azure resources, like virtual machines. It ensures high availability and reliability by balancing traffic and rerouting requests in the event of service failure. Azure also supports both internal and public load balancing, making it suitable for internal application tiers as well as internet-facing services. It integrates with Azure Monitor for real-time health and traffic analytics.

Problem 4. (10 points) Search two major cloud providers (e.g., AWS, Azure, etc.) and find a blockchain technology and application they offer. List and explain the solution they offer (one solution). Please provide the link to each of the products you discuss. Please limit your explanation of each product to 2-3 sentences.

1. AWS – Amazon Managed Blockchain

Product: Amazon Managed Blockchain

Explanation:
Amazon Managed Blockchain is a fully managed service that makes it easy to create and manage scalable blockchain networks using open-source frameworks like **Hyperledger Fabric**. It enables secure, multi-party transactions without the need for a central authority and simplifies the setup and maintenance of blockchain infrastructure. It's commonly used for supply chain tracking, finance, and contract management.

2. Microsoft Azure – Azure Blockchain Service (Deprecated) / Azure Confidential Ledger

Product: Azure Confidential Ledger

Explanation:
Azure Confidential Ledger is a tamper-proof, blockchain-based ledger service designed to store sensitive data with high integrity and security. It provides **immutable** data storage using **trusted execution environments**, making it ideal for audit logs, records, and compliance. While Azure's original Blockchain Service was retired, Confidential Ledger continues to serve blockchain-related needs in a secure and scalable way.