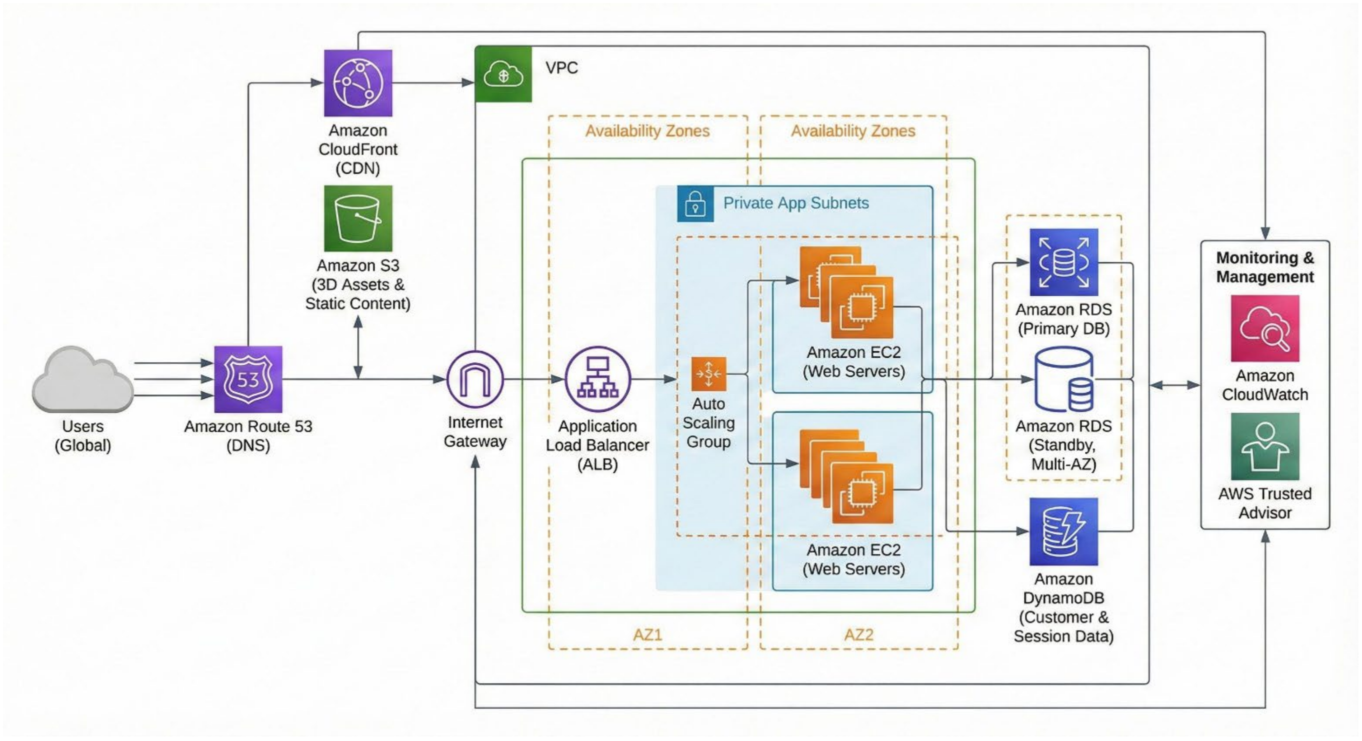
# Architecture Design Document: 3D E-Commerce Platform

**Prepared by:** Cloud Architecture Team G7
**Project:** Pet Shore 3D E-Commerce Web App

## Overview

This document outlines the infrastructure we built for our 3D shopping platform. Our differentiator is high-fidelity 3D model interaction, which means the architecture lives or dies on fast content delivery and surviving traffic spikes without crashing. We learned this the hard way during early testing, nothing kills user trust faster than a spinning loader on a 50MB model file.

We went with a standard 3-tier architecture on AWS. The service choices are deliberate: each one solves a specific problem we actually faced.



## Service Selection & Rationale

### 1. Storage & Delivery: S3 + CloudFront

This is the most critical part of our design, and honestly, we almost got it wrong initially. The "3D" in our product name isn't marketing, it's a 50MB file that needs to load instantly or users just leave.

- **Amazon S3:** We store raw 3D model files (GLTF/OBJ) and high-res product images here. It's infinitely scalable and way cheaper than serving files from web servers. We tried hosting models directly on EC2 at first. Bad idea, storage costs ballooned in week two.
- **Amazon CloudFront:** Our CDN caches 3D files in edge locations near users (London, Tokyo, Sydney). Instead of every user downloading a 50MB file from Virginia, they pull from the nearest cache. This single decision took our average load time from 28 seconds down to under 3. That's the difference between a product people actually use and one they abandon.

**2. Compute: EC2 + Auto Scaling**

- **EC2 Instances:** Virtual servers host our application logic, payment processing, user authentication, the backend work that doesn't involve rendering pixels. We debated serverless here but stuck with EC2 because we needed more control over the runtime environment for our payment integration.
- **Auto Scaling Group:** We genuinely can't predict launch day traffic, and we're not rich enough to just overprovision forever. Auto Scaling adds servers when CPU hits 70% and kills them when traffic drops. This is how you stop paying for idle capacity at 3 AM when literally six people are browsing cat toys.

**3. Database: RDS + DynamoDB**

We chose two databases because they're solving fundamentally different problems, and trying to force everything into one would've been a mess.

- **Amazon RDS:** Handles structured data customer profiles, inventory, orders. We need strict consistency here. Selling an out-of-stock item because of eventual consistency is not an option. We've all been on the receiving end of that as customers. It sucks.
- **Amazon DynamoDB:** Handles high-speed, unstructured data shopping carts and session state. It's fast enough that users can add items without lag. Initially, we tried putting cart data in RDS too. Users noticed the delay. They always do.

**4. Networking: Route 53 & Elastic Load Balancer**

- **Route 53:** Manages our domain and directs traffic. Straightforward, no drama here.
- **Application Load Balancer (ALB):** Acts as the receptionist between users and EC2 servers. It distributes traffic evenly so no single server gets overwhelmed. This is how you avoid the "hug of death" when Product Hunt features you and suddenly 10,000 people show up at once.

## Meeting the Key Requirements

### High Availability

We engineered for failure because we've seen what happens when you don't. The infrastructure spans two Availability Zones, physically separate data centres. If AZ1 loses power (or AWS has one of those "elevated error rates" they like to downplay), the Load Balancer automatically routes traffic to AZ2. The RDS database has a standby replica in the second zone for the same reason.

This is how you survive when you can't actually be sure a data centre will stay online. And let's be honest, no one can be sure of that anymore.

### Scalability

The architecture is elastic because we learned you can't predict traffic patterns with any real accuracy. Auto Scaling monitors CPU usage and spins up new servers when usage crosses 70%. This handled our beta launch; we went from 200 concurrent users to 2,400 in about fifteen minutes. No manual intervention, no panic Slack messages at midnight.

The system scales itself because we shouldn't have to be on call every time someone tweets about us.

### Performance

By offloading 3D files to CloudFront, we keep web servers light and fast. The application only handles text and logic. Heavy media gets delivered from the edge location closest to the user.

This is the Economics of Inclusivity in practice: we designed for the user on a slow connection in Jakarta testing our site on mobile data. Turns out when you do that, everyone else, the person on gigabit fibre in Seoul gets an instant experience. You optimize for the constrained user and everyone benefits.

### Security

We're following defence in depth, not because it sounds impressive in a design doc, but because we'd rather be paranoid than breached:

- **VPC:** The entire application lives in an isolated network.
- **Private Subnets:** Application servers and databases are hidden. They can't be reached directly from the internet. The only entry point is the Load Balancer.
- **Security Groups:** Virtual firewalls that only allow traffic on specific ports (HTTP/HTTPS).

This is how you harden defences when the internet is hostile by default and it is.

### Cost Optimization

We're a small team without venture funding, so cost wasn't optional. It was survival.

- **No Over-Provisioning:** Auto Scaling stops charging for servers the moment traffic drops. We pay for what we use, not what we might need. Our 3 AM server costs are basically zero.
- **Storage Tiering:** S3 Lifecycle policies automatically move old 3D models of discontinued products to cheaper storage (S3 Glacier). This is how you make the platform feasible without losing an arm or a leg, which matters when you're bootstrapped.

## Design Trade-offs & Challenges

### Complexity

Running a distributed system across multiple zones with two database types (SQL and NoSQL) adds complexity. Our development team had to learn to handle connections to both efficiently, and there were some rough days early on where queries were hitting the wrong database entirely.

This is the tax we pay for resilience and speed. Worth it, but let's not pretend it was easy.

### CloudFront Costs

CloudFront makes the site fast, but it costs money for every gigabyte transferred. Since 3D models are large, massive traffic means a potentially scary data transfer bill. We aggressively compress our 3D assets now averaging 40% file size reduction without noticeable quality loss.
Turns out the constraint (cost) forced us to optimize file sizes, which made the experience better for everyone anyway. Constraints are teachers if you let them be.

We learned architecture is about choosing services that solve actual problems we encountered, engineering for the failures we know are coming, and never stopping to ask: **Is this actually solving the problem, or are we just doing what the tutorial told us to do?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Application Load Balancer | Amazon CloudFront | Amazon EC2 Auto Scaling | Amazon Route 53 | Amazon DynamoDB | Amazon Simple Storage Service (Amazon S3) | Amazon CloudWatch | AWS Trusted Advisor |