ChatGPT

Project Overview and Architecture

We will build a **SaaS-style task tracking web app** (like Asana) using a React frontend, a Node.js/Express backend, and a SQL database. This app will be hosted entirely in the cloud (using AWS) as a public cloud service. In a public cloud, all hardware and infrastructure (servers, storage, etc.) are owned and managed by the provider [1]. Our app will use IaaS and PaaS components of AWS: for example, Amazon EC2 (IaaS) to run our application servers and Amazon RDS (a managed database service) to handle SQL data. The overall **cloud model** follows NIST's definition: "cloud computing is on-demand network access to a shared pool of configurable resources (networks, servers, storage, applications) that can be rapidly provisioned and released [2]." In practice, we rent the necessary computing resources from AWS on a pay-as-you-go, free-tier eligible basis [3] [4]. Using AWS's free tier ($100 credits), we can launch a t3.micro EC2 instance, use RDS Free Tier for MySQL, and store static assets on S3 at no cost [5] [4] [6].

Tech Stack and Components

- **Frontend:** React.js SPA (static HTML/CSS/JS) deployed on AWS. We can host it on **Amazon S3 + CloudFront CDN** for low-cost static website hosting [6]. This serves our HTML/JS/CSS, assets, and calls the API.
- **Backend:** Node.js with Express (or similar) providing RESTful APIs. This will run on EC2 (IaaS) or containerized on ECS/EKS. We develop locally, then deploy to AWS.
- **Database:** A managed SQL database. We will use **Amazon RDS (MySQL or PostgreSQL)** on the free tier [4]. RDS automates provisioning, patching, backups, etc., so we focus on app logic [7].
- **Authentication:** AWS Cognito (free tier) or a simple JWT-based login. Cognito would handle user pools/identity, another example of PaaS.
- **Containerization:** Use **Docker** to containerize our backend (and optionally frontend for more complex setup). Containers package the app code and dependencies in a portable image [8]. Unlike VMs, containers share the OS kernel and are lightweight (MBs) [9].
- **Orchestration (optional):** We can use **Kubernetes (EKS or GKE)** to orchestrate containers and demonstrate auto-scaling and rolling updates. AWS EKS has a free management tier; only EC2/EBS costs apply. Alternatively, AWS Elastic Beanstalk or ECS (Fargate) can run containers with built-in scaling.
- **Storage:** Use **Amazon S3** for static file storage (task attachments, images) and host static site content. S3 is highly durable object storage [6].
- **Networking:** Deploy inside an **AWS VPC** (Virtual Private Cloud). A VPC is a logically isolated virtual network in AWS [10] that resembles an on-premises network. We create public/private subnets, route tables, an Internet Gateway, and security groups (firewall rules). We will assign public IPs (Elastic IPs) to load balancers or NAT gateways as needed. (Example VPC architecture shown below).
  Figure: Example VPC with public/private subnets and an Internet Gateway (AWS). This illustrates a virtual network spanning multiple Availability Zones, similar to our planned network setup in AWS [10].

Development and Virtualization Setup

- **Local Environment (Dev VM):** Start by setting up a local development environment. You might install Ubuntu (or any Linux) in **VirtualBox/VMWare** to mimic cloud servers. This reinforces virtualization concepts (VMs vs containers). According to AWS, a VM is a "digital copy of a physical machine" running its own OS [8], whereas Docker containers are lightweight packages with just app code and libraries [8] [9]. Working inside a VM or Docker ensures our app runs in a controlled environment.
- **Code Management:** Use Git (e.g. GitHub) to track code. Structure the project with a React client and a Node.js server. Scaffold React (e.g. Create React App) and initialize an Express server. Define REST endpoints for tasks/projects and user auth.
- **Database Schema:** Design tables (Users, Projects, Tasks, etc.) and relationships. Test locally with a MySQL instance (Docker or local install). Practice using an ORM (like Sequelize or Prisma) or raw SQL.
- **Containerization (Docker):** Write Dockerfiles for frontend and backend. This step ties into "System virtualization" by moving from full VMs to containers. Build images and test via `docker-compose`. This aligns with syllabus topics on virtualization and containers.
- **Virtualization Concepts:** Remember hypervisors (like KVM used by AWS) manage VMs. Containers, by contrast, are managed by a container engine (like Docker) [11]. We can cite that AWS also supports Kubernetes via EKS or Docker via ECS, which run on top of underlying VMs.

Cloud Infrastructure and Services

- **AWS Account Setup:** Use your AWS account (with $100 credits [5]). Enable AWS CloudTrail and AWS Budgets to track usage. Create IAM users/roles with least privilege for deploying resources. AWS IAM ensures only authorized access (shared responsibility for security).
- **VPC and Networking:** Launch resources in a new VPC (or default VPC). Create two public subnets (in different AZs) and two private subnets. Public subnets host load balancers or bastion; private subnets host EC2/RDS. Attach an Internet Gateway for external traffic [10]. Set up security groups: e.g. allow HTTP/HTTPS inbound to ALB, allow backend to communicate on port 3306 (MySQL) only from app servers, and restrict SSH.
- **Compute (EC2 + Auto Scaling):** Provision EC2 instances (t3.micro) in an Auto Scaling Group. Auto Scaling maintains the desired number of instances and can grow/shrink with load [12]. The tutorial notes we can use Auto Scaling on free tier (750 hours of t2/t3.micro) [13]. Place instances in private subnets.
- **Load Balancing:** Create an Application Load Balancer (ALB) in the public subnets to distribute traffic to the EC2 instances. ALB will listen on 80/443 and forward to backend port (e.g. 3000). Elastic Load Balancing "automatically distributes incoming application traffic across multiple targets" [14], improving availability.
- **Database (RDS):** Launch an Amazon RDS MySQL (or PostgreSQL) instance in a private subnet. The free tier gives 750 hours/month of db.t3.micro with 20GB storage [4]. RDS is PaaS: AWS manages the hardware, backups, and patching [7]. Connect the backend service to RDS securely (no public access).
- **Static Hosting (S3 & CloudFront):** Build the React frontend and upload it to an S3 bucket configured for static website hosting. Then front it with Amazon CloudFront (CDN) for SSL and global cache (CloudFront has an always-free tier for 1TB transfer [15]). This yields a fast, cost-free hosting for the UI.

- **Serverless (Lambda, if needed):** For certain features like sending email reminders or cron tasks, we can use AWS Lambda (free for 1M requests/month [16]). For example, trigger a Lambda via CloudWatch Events to scan for overdue tasks and send notifications. This covers "Event-driven programs with Cloud Functions" (GCP term) or Lambda.
- **Monitoring & Logging:** Use AWS CloudWatch to collect logs/metrics from EC2 and RDS. Set alarms on high CPU or errors. CloudWatch is partially free and essential for a production-ready app.
- **CI/CD Pipeline:** Set up AWS CodePipeline (free 30 days then ~$1/month) or GitHub Actions to automate builds and deployment. For example, on every push, build the Docker image and push to **Amazon ECR** (free for public images) or **Docker Hub**. Then redeploy ECS/EKS or update EC2. This embodies DevOps practices.

Integration of Cloud Concepts

- **IaaS vs PaaS vs SaaS:** Our use of EC2 and RDS illustrates IaaS/PaaS. As Azure notes, "IaaS: rent servers, storage, networks… on a pay-as-you-go basis" [3]. Here we rent EC2 and EBS volumes. RDS is PaaS: AWS provides the whole database platform, we just store data. Meanwhile, our final product is *offered as SaaS* – a fully managed web application that users access via browser. In SaaS, "cloud providers host and manage the application and underlying infrastructure" so end-users only connect to the app [17]. By contrast, in IaaS we manage more of the stack, and in PaaS AWS manages more. Google Cloud's explanation highlights that with PaaS "the provider manages all the hardware and software resources needed… You write the code and manage the data, but you do not have to manage the underlying platform" [18].
- **Virtualization:** At the infrastructure level, AWS uses virtualization (hypervisors, Xen/KVM) to run multiple VMs on physical hardware. Each EC2 is a VM "guest" running its own OS [11]. We don't deal with hypervisors directly, but it's useful to understand: VMs are heavier (full OS) and isolated by a hypervisor, whereas containers share the OS kernel and are more lightweight [9]. For this project, after building containers, we could deploy them on Kubernetes (which itself runs on VMs) to demonstrate advanced virtualization and orchestration.
- **Auto Scaling and Elasticity:** We will configure the Auto Scaling Group to scale out if CPU >70%. This shows "rapid elasticity" – automatically scaling up/down to meet load, a key cloud characteristic [2]. AWS also supports Spot instances or Spot Fleets to reduce cost when scaling.
- **Storage Options:** We use **structured storage** (RDS) for relational data and **unstructured storage** (S3) for files. AWS S3 offers "99.999999999% durability" and can host static websites [6]. If we wanted NoSQL (not required here), AWS offers DynamoDB (free tier available) or DocumentDB.
- **Security:** Follow the shared security model: AWS secures the cloud (hardware, hypervisor) while we secure in the cloud (our data and apps). Use **HTTPS (TLS)** for in-transit encryption, enable **encryption at rest** on RDS and S3. Use IAM roles for least privilege (e.g. an EC2 instance role that only allows RDS and S3 access needed).
- **Networking:** In our VPC we will manage subnets and routing. For example, use public IPv4 addresses/Elastic IPs for the load balancer, and private IPs for instances. The AWS VPC we create is "like a traditional network in your data center" [10]. Multiple AZs and subnets provide high availability. We can also illustrate hybrid concepts (e.g. our local laptop VPN to VPC) for completeness, but a simple public cloud deployment suffices.

Deployment and Scaling Strategy

- **Stage vs Prod:** Create separate environments (e.g. VPCs) for testing and production. Use CloudFormation or Terraform to codify the infrastructure, so it can be replicated.
- **CI/CD:** After initial manual setup, implement a CI/CD pipeline. For example, push code to GitHub, trigger AWS CodeBuild to test, then AWS CodeDeploy or a rolling update on EKS. This automates delivery and aligns with DevOps.
- **Load Testing:** Before deadline, perform load tests (tools like Apache JMeter or k6) to ensure the auto-scaling and load balancer work under stress. Adjust instance type or number as needed (staying within free tier limits).
- **Zero Cost Practices:** To avoid charges, ensure resources are under free-tier quotas (e.g. use one t3.micro, <20GB RDS, small S3 bucket). Shut down or terminate resources when not in use (e.g. no need for 24/7 running during development if not needed). AWS Budgets and CloudWatch billing alarms can prevent overspend.

Project Roadmap (2 Months)

1. **Week 1: Requirements & Design.** Define features (task CRUD, user auth, teams). Sketch architecture diagrams. Sign up AWS, explore Free Tier offers [5]. Set up a dev VM or container environment, and decide tech choices.
2. **Weeks 2–3: Backend Development.** Build the Node.js API (task, project, user endpoints). Design the SQL schema and integrate RDS (start with local DB, then move to RDS). Write unit tests for API.
3. **Weeks 4–5: Frontend Development.** Develop the React app (login, dashboard, task UI). Connect to the backend API. Test end-to-end flows. Style with CSS frameworks or custom design.
4. **Week 6: Containerization and Local Testing.** Create Dockerfiles for frontend/backend and run them together (using Docker Compose or Minikube). Ensure the system works with containers. Study virtualization concepts from this step (hypervisors vs container engines [11]).
5. **Week 7: Cloud Setup and Deployment.** Provision AWS resources (VPC, EC2, RDS, S3, ELB). Deploy the backend on EC2 (or EKS) and connect to RDS. Host the frontend on S3/CloudFront. Configure DNS (Route 53) and SSL (AWS Certificate Manager). This is where we apply IaaS/PaaS knowledge.
6. **Week 8: Scaling, Monitoring, Final Testing.** Configure Auto Scaling and CloudWatch alarms. Perform load testing and ensure autoscaling triggers. Implement any remaining features (email via Lambda, analytics, etc.). Polish UI/UX. Ensure the app runs reliably under cloud conditions.
7. **Delivery:** Prepare documentation and a live demo link. (While formal reports/presentation come later, ensure the site is deployed and accessible.)

Throughout the project, **cite cloud concepts** on the resume: e.g. “designed multi-AZ VPC on AWS [10],” “containerized microservices (Docker) [9],” or “implemented CI/CD pipeline with AWS CodePipeline.” By end of two months, we will have a **production-ready SaaS app**, leveraging many syllabus topics (IaaS, PaaS, virtualization, containers, auto-scaling, load balancing, VPC, IAM, etc.). All components will be on free or free-tier services, ensuring zero additional cost. The final deployed platform – a cloud-based task tracking website – will demonstrate comprehensive cloud engineering skills suitable for a resume.

Sources: AWS documentation and cloud resources were consulted to ensure best practices (e.g. AWS Free Tier offers [5] [6], RDS benefits [7], virtualization differences [8] [9], and service model definitions [3] [19] [1]).

---

1 Public Cloud vs Private Cloud vs Hybrid Cloud | Microsoft Azure
https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-are-private-public-hybrid-clouds

2 SP 800-145, The NIST Definition of Cloud Computing | CSRC
https://csrc.nist.gov/pubs/sp/800/145/final

3 What is Infrastructure as a Service (IaaS)? | Microsoft Azure
https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-iaas

4 7 Amazon RDS Free Tier | Cloud Relational Database | Amazon Web Services
https://aws.amazon.com/rds/free/

5 Free Cloud Computing Services - AWS Free Tier
https://aws.amazon.com/free/

6 14 15 16 Build Free Websites and Webapps
https://aws.amazon.com/free/webapps/

8 9 11 Containers vs VM - Difference Between Deployment Technologies - AWS
https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines/

10 What is Amazon VPC? - Amazon Virtual Private Cloud
https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html

12 13 Tutorial: Create your first Auto Scaling group - Amazon EC2 Auto Scaling
https://docs.aws.amazon.com/autoscaling/ec2/userguide/create-your-first-auto-scaling-group.html

17 18 19 What Is PaaS? | Google Cloud
https://cloud.google.com/learn/what-is-paas