**1.2 AWS Services Selection**

**1. Frontend Hosting**

**Services Used:** *Amazon S3 + CloudFront*

I chose **S3** to host the frontend because it’s simple, cost-effective, and handles static files like HTML, CSS, and JavaScript really well. To make sure the site loads quickly for users everywhere, I added **CloudFront** as a CDN. It caches content globally and also helps with HTTPS setup and DDoS protection when combined with WAF.

**2. User Authentication**

**Service Used:** *Amazon Cognito*

Instead of building an authentication system from scratch, I used **Cognito**. It covers user signup, login, and even MFA (multi-factor authentication) out of the box. What I liked is how it handles user pools and roles securely, and being an AWS service, it integrates well with other components like API Gateway and Lambda.

**3. API Management**

**Service Used:** *Amazon API Gateway*

For all backend communication, I went with **API Gateway**. It acts as the front door for all client requests and connects smoothly with Lambda. I chose it because it allows rate limiting, input validation, and detailed logging — all important for managing APIs securely and at scale.

**4. Business Logic & Backend Tasks**

**Services Used:** *AWS Lambda / ECS with Fargate*

For most of the backend logic (like parsing uploaded CVs), I used **Lambda** since it's serverless and scales automatically. For more complex workloads or ones that need containers, I used **ECS with Fargate**. This setup gives me the best of both worlds — simple event-driven tasks and more control when needed.

**5. CV Extraction and Processing**

**Services Used:** *Amazon Textract, Amazon Comprehend, Custom ML Model (optional)*

To read and extract data from resumes, I integrated **Textract** because it works well with both PDF and DOCX files and gives structured output. For identifying skills or job roles, I used **Comprehend** (AWS’s NLP service). If needed later, I plan to add a **custom ML model** to improve accuracy for specific job domains.

**6. Database Layer**

**Service Used:** *Amazon RDS (PostgreSQL)*

For storing structured data like user info and extracted metadata from CVs, I selected **PostgreSQL on RDS**. It’s a solid relational database with powerful querying abilities — great for filtering candidates based on skills, experience, etc. Plus, RDS handles backups, scaling, and patching, which saves time.

**7. File Storage**

**Service Used:** *Amazon S3*

All the uploaded CV files are stored in **S3**. It’s reliable, highly durable, and supports encryption by default. It also integrates well with lifecycle rules, so I can archive older files to save on storage costs. Honestly, it’s the most straightforward and scalable option for storing large files.

**8. Search Functionality**

**Service Used:** *Amazon OpenSearch*

To let recruiters search through CV data quickly, I integrated **OpenSearch**. It allows full-text search and ranking, which is ideal for finding the right profiles fast. It’s also flexible in terms of indexing, so I can add filters like years of experience, skills, or location easily.

**9. Monitoring & Logging**

**Service Used:** *Amazon CloudWatch*

To keep track of what’s going on in the system, I used **CloudWatch**. It collects logs from Lambda and ECS, shows performance metrics, and lets me set up alerts when something goes wrong. Having a single place for monitoring really helps with debugging and maintaining the system.

**10. Security**

**Services Used:** *AWS WAF, KMS, IAM*

Security is a priority for me. I used **WAF** to block common attacks like SQL injection and cross-site scripting. **KMS** helps manage encryption keys, especially for protecting data at rest. And **IAM** roles ensure that services only access what they’re supposed to — following the least privilege principle.

**1.4 Scalability Strategy**

**1. Traffic Spikes – Be Ready for Sudden Load**

One of the first things I considered was how to manage unexpected spikes in user activity — for example, when many recruiters upload CVs at the same time.

That’s why I chose AWS Lambda for backend processing. It’s serverless and automatically scales with the number of incoming requests. No need to worry about provisioning or managing servers.

* API Gateway adds throttling and burst handling, making sure requests don’t overwhelm the system.
* For the frontend, CloudFront helps deliver content from edge locations close to the user, reducing latency and offloading traffic from the origin server.
* Amazon S3, used for file uploads, scales on its own — no need for manual setup.

Example: If 100 users upload CVs at once, Lambda will spin up 100 parallel executions instantly — smooth*,* no lag*.*

**2. Data Growth – Designed to Grow with Users**

As the platform grows, so will the number of uploaded CVs and user data. We’re expecting to move from 10,000 CVs per month to 50,000+, so I’ve built in strategies to handle that growth.

* RDS partitioning by month or using UUIDs helps avoid performance issues from large tables.
* I enabled auto-scaling for RDS so it can handle growing storage and compute needs automatically.
* Older CV files (e.g., older than 1 year) will be moved to Glacier, a low-cost, long-term storage option.
* For metadata, older records can be moved to a read-only RDS instance or S3 in Parquet format for reporting and analytics.

**3. Global Access – Serving Users Worldwide**

Since the system may be accessed by recruiters from different countries, global performance is important.

* CloudFront delivers frontend assets quickly no matter where the user is.
* I can deploy the Lambda + API Gateway combo in different AWS regions like Tokyo, Singapore, or Frankfurt based on demand.
* With Route 53's latency-based routing, users are automatically directed to the region that gives them the fastest experience.
* For database reads (like search functions), RDS read replicas in different regions will help reduce cross-region delays.

**4. Cost Optimization – Scale Smart, Not Expensive**

While building for scale, I’ve also made sure the system stays cost-efficient:

* Lambda, API Gateway, and Textract are all usage-based — we only pay when they’re used.
* S3 lifecycle policies help manage storage costs by automatically moving old files to cheaper storage classes.
* I plan to use Cost Explorer and Compute Optimizer to monitor for waste and tune resources.
* Non-urgent tasks like report generation will be scheduled during off-peak hours using CloudWatch Events or Event Bridge to avoid unnecessary load.

**02. Database Architecture**  
**2.1 Entity Relationship Design**Core Entities

* **Users**  
  Represents platform users such as recruiters and admins. Contains details like name, email, role, and authentication reference.
* **Candidates**  
  Stores parsed data of job applicants extracted from uploaded CVs. Includes personal info, contact details, experience, education, and other metadata.
* **CV Files**  
  Links to actual CV documents stored in S3. Maintains file paths, formats, upload timestamps, and a reference to the related candidate.
* **Skills**  
  A separate entity to store and categorize skills for better querying and filtering. Skills are mapped to candidates via a many-to-many relationship.
* **Jobs**  
  Represents recruitment postings or job requirements. Includes job title, description, required skills, and status. Also linked to users (recruiters).
* **Audit Logs**  
  Captures user actions and events for traceability and security purposes. Useful for identifying unauthorized access or system misuse.

**Design Considerations**

* **Normalization (3NF)**:  
  All entities are normalized to at least third normal form to avoid redundancy and ensure data consistency.
* **Relationships**:
  + One-to-many: Users → Jobs
  + One-to-many: Candidates → CV Files
  + Many-to-many: Candidates ↔ Skills (handled via a junction table)
  + One-to-many: Users → Audit Logs
* **Data Types & Constraints**:  
  I carefully selected appropriate data types (e.g., UUIDs for primary keys, timestamps, ENUMs for roles/status) and applied constraints like NOT NULL, UNIQUE, and FOREIGN KEY to enforce referential integrity.
* **Scalability**:  
  The design is flexible enough to accommodate future enhancements such as:
  + Adding interview records
  + Storing job applications
  + Skill proficiency levels
  + Integration with external systems

**2.1 Entity Relationship Design – ER diagram**

Audit Logs

Users

CV Files

Jobs

Candidates

Skills

**03. Security Best Practices   
3.1 Authentication & Authorization Architecture Design** multi-factor   
 Authentication   
  
  
  
  
  
  
 + Role- Base Access Control  
 + Least Privilege   
  
 + Audit Logging

Applications and data

Identity Provider

Authentication  
Server

Users

User Roles

Permissions

Authentication  
Server