24-04-2025 17:01

Status: #final

Tags: JobTracker

Job Tracker Project

Description:

- Job Tracker is an innovative solution built to help people track their job applications efficiently and have real time analytics about their job hunting journey.

Functionalities:

Job applications:

- The user can save their job applications to keep track of them.
- The user can specify for each job application the company, the position, the location, the salary, the job description (or job description link) and the status (where the user is in the job application process)(interview, applied, saved only, offer, rejected).
- The user can filter through their applications by status.
- The user can look for their application by position or company name.

Interviews:

- The user can save their scheduled interviews.
- The user can specify for each interview the company, the position, the date, the time, the interview type (phone, on-site, online) and some details about it.
- The user can access to all their scheduled interview and filter them by date or type.

Profile:

- The user can create their own profile.
- The user can specify their full name, email, phone number, location, position and skills.
- The user can upload their resume.
- The user can add links to their social media.
- The user's profile is accessible by all the other users of the app.
- The user can look for other users using their name.

- The user can have a list of profiles they are interested in.

Contacts:

- The use can save multiple contacts with their information.
- The user can specify for each one the name, the position, the company they are working at and the contact information (phone number, email, social media).
- The contacts are not necessarily users of the application but if one of them is, a link is automatically added to access their profile.
- The user can search in their contacts by name, company or position.

Analytics:

- The user can have metrics concerning their applications.
- The user can see the number of applications submitted per month.
- The user can see the status of all the applications (how many are applied, how many are rejected, etc.).
- The user can see the distribution of applications in terms of positions (frontend, DevOps, etc.).
- The user can see the distribution of applications in terms of location.

Dashboard:

- The user can access a dashboard with an overview of their job applications.
- The user can see the total number of applications that are in progress.
- The user can see the upcoming interviews.
- The user can see the recent applications.

Classes:

- User
- Interviews
- Applications
- Contacts
- Analytics
- Notifications

Architecture:

For this project, we will implement a microservices' architecture.

Services:

Service	Responsibilities
User Service	Manages user accounts, profiles.
Application Service	Handles job applications CRUD operations and filtering.
Interview Service	Manages interviews and filtering by type or date.
Contact Service	Manages personal contacts related to job hunting.
Analytics Service	Aggregates data from other services and provides statistics/insights.
Dashboard Service	Composes and aggregates data from multiple services for the dashboard view.
Notification Service	Sends reminders for interviews or updates about applications.
Search Service	Provides search functionalities for users, applications, and contacts.
File Upload Service	Handles resume uploads and media storage.
API Gateway	Entry point for client requests; routes them to the appropriate services.
Auth Service	Handles login, registration, and token issuance.

Services and Details:

1. User Service

- **Responsibilities**: Profile management.
- **Communicates with**:
  - File Upload Service (for uploading resumes).
  - Contact Service (to cross-reference if a contact is a user).
- **Framework:** Node.js
- **Database:** MongoDB
- **Storage:** Links resume uploads to S3
- **Dependencies:** File Upload, Search, Contact

2. Application Service

- **Responsibilities**: Add/edit job applications, filter/search by status, company, etc.
- **Communicates with**:
  - Analytics Service (via events).

    - Interview Service (REST/gRPC).
- **Framework:** Django
- **Database:** MongoDB
- **Event Source:** Publishes to Kafka/RabbitMQ: `application_created`, `status_updated`

3. Interview Service

- **Responsibilities**: Save and filter interviews.
- **Communicates with**:
    - Notification Service (RabbitMQ/Kafka).
    - Analytics (RabbitMQ/Kafka)
    - Application Service (REST/gRPC).
- **Framework:** Spring Boot
- **Database:** MongoDB

4. Contact Service

- **Responsibilities**: Manage user's professional contacts.
- **Cross-link**: Detect if a contact is a registered user (calls User Service).
- **Framework:** Spring Boot
- **Database:** MongoDB
- **Function:** Associates job-related contacts, links with User if possible

6. Dashboard Service

- **Responsibilities**: Compose and aggregate data from other services.
- **Communication**: Asynchronous using Kafka/RabbitMQ (SSE) with Analytics.
- **Framework:** NodeJS
- **Listens to:** Analytics

7. Notification Service

- **Responsibilities**: Notify users about upcoming interviews.
- **Framework:** Node.js
- **Input:** Kafka/RabbitMQ events
- **Output:** Emails via AWS SES, SMS via AWS SNS, Push/In-app
- **Fat Events:** Carries user email/name to avoid fetching from User Service

8. Search Service

- **Responsibilities**: Full-text search for applications, users, and contacts.
- **Framework:** Node.js
- **Database:** Amazon Open Search or Meilisearch (Maybe Elasticsearch)
- **Input:** Kafka/RabbitMQ (User, App, Contact)
- **Output:** REST API for search

9. File Upload Service

- **Responsibilities**: Handle resume uploads.
- **Communicates with**:
  - User Service (to attach uploaded file to profile).
- **Framework:** Node.js
- **Integration:** AWS S3
- **Function:** Handles pre-signed uploads, serves resumes

10. API Gateway

- **Responsibilities**: Routes external requests, handles auth, rate limiting, and maybe GraphQL.
- **Service:** Amazon API Gateway or Kong Gateway on ECS
- **Purpose:** Expose endpoints to frontend, route to services
- **Security:** JWT token validation (public key from Auth)

11. Authentication Service

- Handles:
  - OAuth2 / JWT Token issuing
  - Login/logout
- **Framework:** Spring Boot
- **Service Location:** ECS or EKS
- **Function:** Issues JWT / OAuth2 tokens
- **Optional:** Use **Amazon Cognito** for user pool management if external auth needed

12. Analytics Service

- Handles:
  - Fetches metrics from application and interview services.
- Framework: NodeJS
- Communication: Hybrid(REST/gRPC and Kafka/RabbitMQ)
- Relations: Calls application and interview services and sends to the dashboard service

13. Service Registry

- Handles:
  - Makes the communication between services easier (Service Discovery)
  - Saves all the services in one place
- Service:
  - AWS Cloud Map
  - App Mesh
  - HashiCorp Consul
  - ECS + Cloud Map

Tech Stack:

- Backend: Node.js, Spring Boot and Django.
- Message Queue: Kafka / RabbitMQ.
- **API Gateway**: Kong / Express Gateway / GraphQL.
- **Database**: PostgreSQL / MongoDB.
- **Search**: Elasticsearch / Meilisearch.
- **Storage**: AWS S3 for resumes.
- **Auth**: JWT or OAuth2.
- **CI/CD**: GitHub Actions / GitLab CI / Docker/ Jenkins.

12. Databases

Service	DB Type	AWS Option
User, Application	Relational	Amazon RDS (PostgreSQL)
Contacts, Files	Document-based	MongoDB Atlas or DynamoDB
Search	Inverted Index	OpenSearch (AWS)

13. Messaging

Component	Technology	AWS Option
Event Broker	Kafka / RabbitMQ	Amazon MSK / MQ
Queue Consumers	Search, Notification	ECS/EKS services
Dead Letter Handling	SQS (optional)	For retries/failures

14. Storage

Purpose	AWS Service
Resume Files	AWS S3
Backups	S3 Glacier
Static Hosting	S3 + CloudFront (if needed)

15. CI/CD

Stage	Tool
Build/Test	GitHub Actions / Jenkins
Image Registry	Amazon ECR
Deployment	ECS Blue/Green Deploy or GitLab CI/CD
IaC	Terraform (for infra)

16. Monitoring / Observability

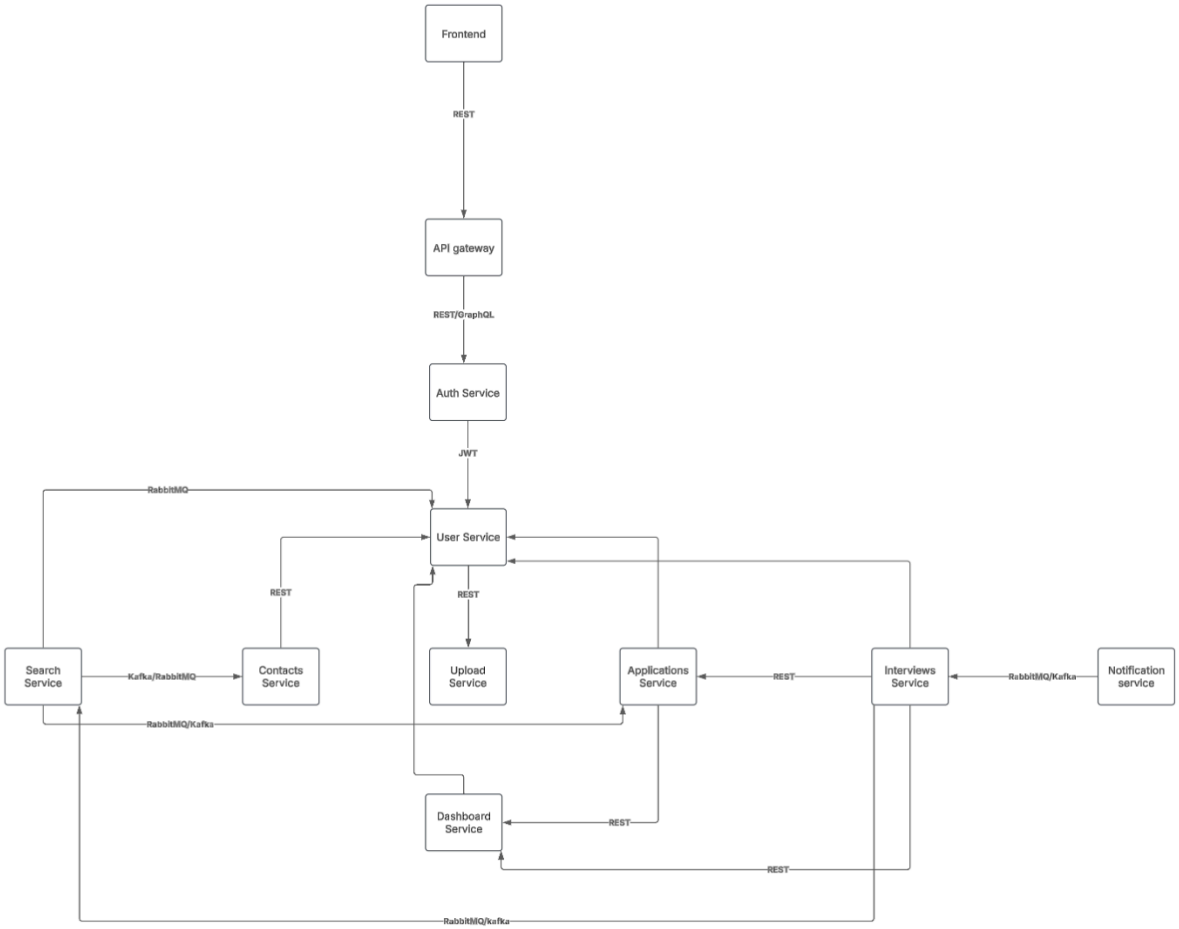
Concern	Tool
Logs	CloudWatch Logs
Metrics	CloudWatch Metrics + Alarms
Tracing	AWS X-Ray + OpenTelemetry
Dashboards	CloudWatch Dashboards / Grafana
Alerting	SNS / Slack / Email alerts

17. Secrets & Config

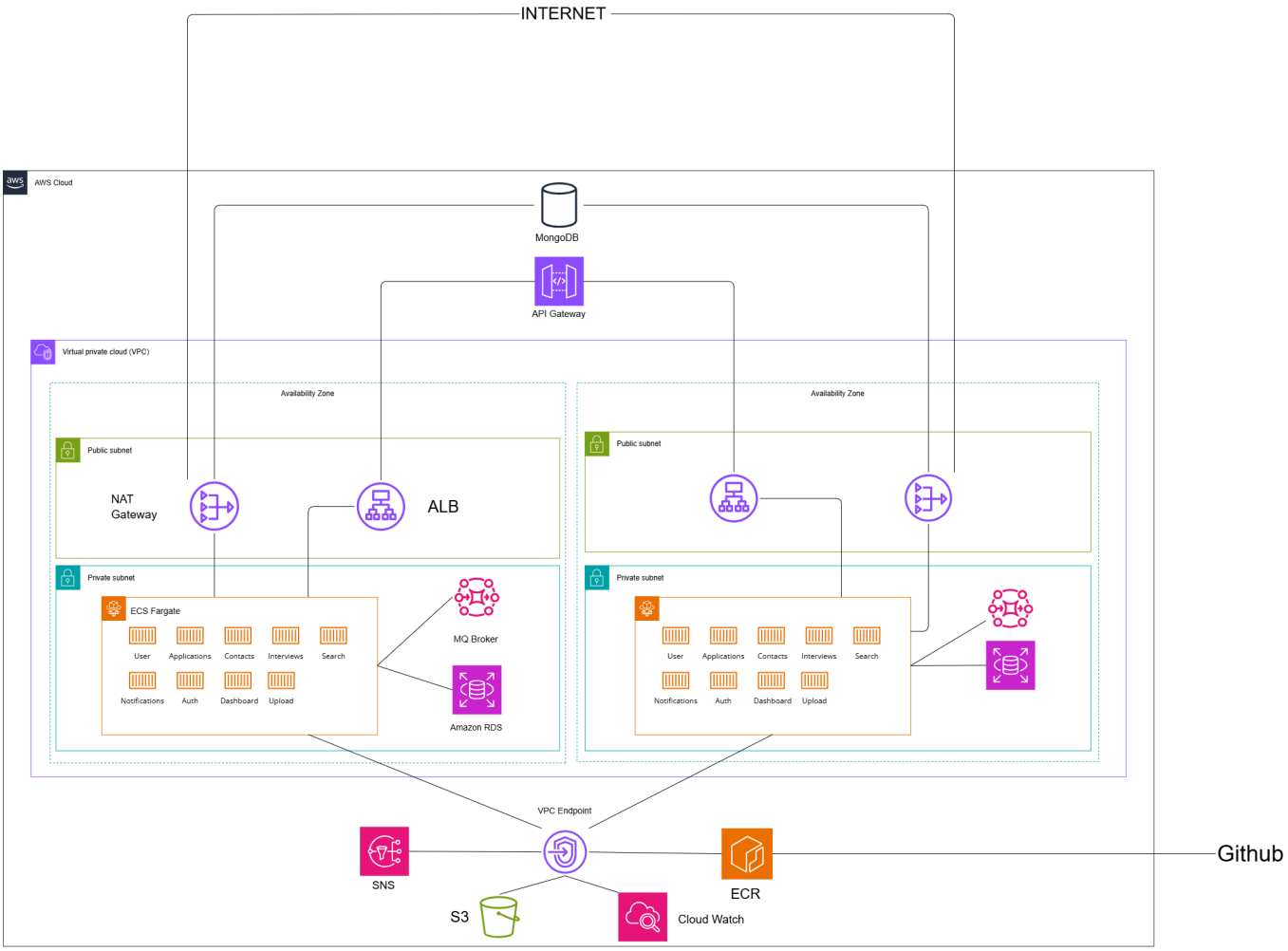
Purpose	AWS Service
DB creds, API	AWS Secrets Manager
Env variables	AWS SSM Parameter Store

Service Communication Modes

Type	Use Case	Protocol
REST/gRPC	Sync data access	HTTPS / gRPC
Event-driven	Async updates, notifications	Kafka/RabbitMQ
File Access	Resume upload & retrieval	S3 Pre-signed URLs
Search	Full-text lookup	



AWS Deployment :



1. VPC :

We need 1 VPC for the entire application.

2. Subnets: 6 (at least)

We should use multiple subnets across 2+ Availability Zones for high availability.

Subnet Type	Count	Purpose
Public Subnet	2	For Load Balancer, NAT Gateway, and optional Kong Gateway
Private Subnet	2	For ECS services (Auth, User, App, etc.)
DB Subnet	2	For RDS/MongoDB (with subnet group)

Each pair is spread across 2 Availability Zones (e.g., `us-east-1a`, `us-east-1b`) for resilience.

3. Security Groups: ~5-6 recommended

SG Name	Applied To	Inbound Rules
`alb-sg`	Application Load Balancer	Allow HTTP/HTTPS from anywhere (0.0.0.0/0)
`ecs-sg`	ECS Services	Allow traffic from `alb-sg`
`db-sg`	RDS / MongoDB	Allow from `ecs-sg` only
`bastion-sg` *(optional)*	Bastion Host (for DB SSH)	Allow SSH from your IP
`search-sg`	OpenSearch / Meilisearch	Allow from `ecs-sg` or specific services
`ci-cd-sg`	GitHub Actions runners (optional EC2)	Allow outbound to Docker repos, ECS APIs

4. Route Tables and NAT Gateways

Component	Notes
Route Tables	One for public subnets, one for private.
NAT Gateway	Required to let ECS in private subnet pull images & updates from the internet (DockerHub, GitHub). Place in a public subnet.

```
VPC (10.0.0.0/16)
├── Public Subnet 1 (10.0.1.0/24) ─┬─ Load Balancer (ALB)
│                                  └─ NAT Gateway
├── Public Subnet 2 (10.0.2.0/24) ─┬─ (HA setup)
├── Private App Subnet 1 (10.0.3.0/24) — ECS Services (Task Definitions)
├── Private App Subnet 2 (10.0.4.0/24)
├── DB Subnet 1 (10.0.5.0/24) — RDS / MongoDB Atlas
└── DB Subnet 2 (10.0.6.0/24)
```

AWS Job Tracker Cost Estimate:

Service/Component	Free or Billable	Estimated Monthly Cost	Notes
VPC	Free	$0	No charge for VPC itself

Service/Component	Free or Billable	Estimated Monthly Cost	Notes
Subnets	Free	$0	No charge for subnets
Internet Gateway (IGW)	Free	$0	No charge for IGW; egress bandwidth is charged separately
NAT Gateway	Billable	~$33 + bandwidth	$0.045/hr (~$33/mo), plus $0.048/GB data processed
Route Tables	Free	$0	
Security Groups, NACLs	Free	$0	
S3 (App, logs, Terraform state)	Billable	$1–$5 (small projects)	Storage: $0.023/GB/mo; Request + bandwidth charges apply
S3 Glacier (Backups)	Billable	Varies	$0.004/GB/mo for storage, small archive retrieval charges
ECR (Image registry)	Billable	$0.10+/month	$0.10/GB/mo stored; first 500MB/month free, plus data transfer
CloudWatch Logs	Billable	$2–$30+	$0.50/GB ingested + storage + $0.10/alarm/mo (estimate: 3GB logs, 15 alarms shown)
CloudWatch Alarms	Billable	Included in above	$0.10/basic alarm/month
ECS Fargate (2 tasks, 1 vCPU, 2GB each)	Billable	~$365 (constant 2 tasks)	$0.04048/vCPU-hr, $0.004445/GB-hr; (2 tasks x 730hr x [1vCPU + 2GB]) approx ~$365/mo
Cloud Map (Service Discovery)	Billable	~$1–$5	$0.10/per discovery API/month, namespaces cost pennies
RDS db.t3.medium PostgreSQL	Billable	~$66	$0.09/hr + $0.115/GB storage (20GB = $2.30)
DocumentDB (dev, minimal)	Billable	~$200+ (or use Atlas)	Starts ~$200/mo (dev cluster, 2 r5.large on-demand); Atlas alternatives may differ
DynamoDB (for app or Terraform lock)	Billable	Low (<$1–$5 with light use)	$1.25/million writes, $0.25/million reads; locking is cheap
OpenSearch (small)	Billable	$60+	eg. t3.small.search instance; varies with nodes/storage

Service/Component	Free or Billable	Estimated Monthly Cost	Notes
Amazon MQ / MSK (Kafka/RabbitMQ)	Billable	$25+ (single-az dev)	MQ single instance: $0.074/hr (~$54/mo). MSK: $0.21/hr/broker (min 2 brokers: $306/mo)
SQS	Billable	Free/very low	First 1M monthly requests free, > $0.40/M for standard
Application Load Balancer (ALB)	Billable	$16.43 + BW	$0.0225/hr ($16.43/mo), + $0.008/GB data
API Gateway	Billable	~$3 (dev)/$15+ (prod)	$3.50 per million requests ($1/mo for dev, scale by usage)
X-Ray	Billable	$5–$10	$5/month per 1M traces recorded (dev), $0.50 per 1M traces scanned
SNS	Billable	Free/very low	First 1M notifications free, then $0.50/M
SES	Billable	Free/low	62,000 emails sent/mo free from EC2, then $0.10/1,000 mails
Secrets Manager	Billable	$0.40/secret/mo	
SSM Parameter Store (standard)	Free	$0	Advanced params cost $0.05 each
Terraform DynamoDB (lock)	Billable	<$1/month	Minimal read/write usage

Estimated Total (example scenario: low-mid traffic, 2 running ECS tasks)

- ECS Fargate: ~$365
- RDS: ~$66
- NAT Gateway: ~$33
- ALB: ~$16
- CloudWatch Logs/Alarms: ~$10
- S3: ~$3
- ECR: ~$0.10
- MQ/MSK: ~$54 (MQ) or higher for Kafka
- OpenSearch: ~$60

- X-Ray: ~$5
- Secrets Manager/Parameter Store/other: ~$3

Estimated Monthly Total: $550–$650+

NOTES :

- Make sure that the events published by the interview service have all the necessary information so that the notification service can contact the user, otherwise we must connect directly the user service to the notification one.
- For notification we could use Amazon SES and SNS.
- For Rabbit MQ we could use the AWS Broker.
- We can either self-host the MongoDB using a container or use MongoDB atlas.
- Add a circuit breaker between the services: Use a circuit breaker when:
  - A service calls another service (especially over the network).
  - That call is critical, and failures could cascade or cause timeouts.
  - The downstream service is known to be unstable or under heavy load.
  - You want to provide a fallback, like cached data or a default response.
- Add a service registry to the architecture.
- Since NAT gateways are expensive, we will use for the dev env only one, but it's better to use one for each AZ.
- In dev I only created one private route table, but I should create in prod one for each AZ that I have. A single public route table is enough.
- A VPC endpoint is away for resources in my private subnets to securely access AWS services without crossing the internet.
- There are two types of VPC endpoints:
  - **Gateway Endpoint:** It connects AWS storage services (S3, DynamoDB only). Adds a route to our route table. No SG needed because they are route based.
  - **Interface Endpoint:** Most other AWS services. It creates an **Elastic Network Interface (ENI)** inside our private subnet.

```
Gateway Endpoint (S3 Access):
Private Subnet ----> Private Route Table ----> S3 Gateway Endpoint ----> S3

Interface Endpoint (ECR, CloudWatch, SNS):
Private Subnet ----> ENI (PrivateLink Interface) ----> AWS Service (private
AWS IPs)
```

- When we set up our services (like ECS tasks), they will automatically use the private endpoints because AWS automatically updates the internal DNS to resolve to the VPC Endpoint! No changes needed inside the app code!
- Deleted: NAT gateway, VPC Endpoints and EIP.
- **PROBABLY SWITCH TO A LOCAL DEPLOYMENT!!!**