Deploying the Health Tracker API on AWS

To deploy the Health Tracker API on AWS, we will use AWS services that ensure scalability, security, and high availability.

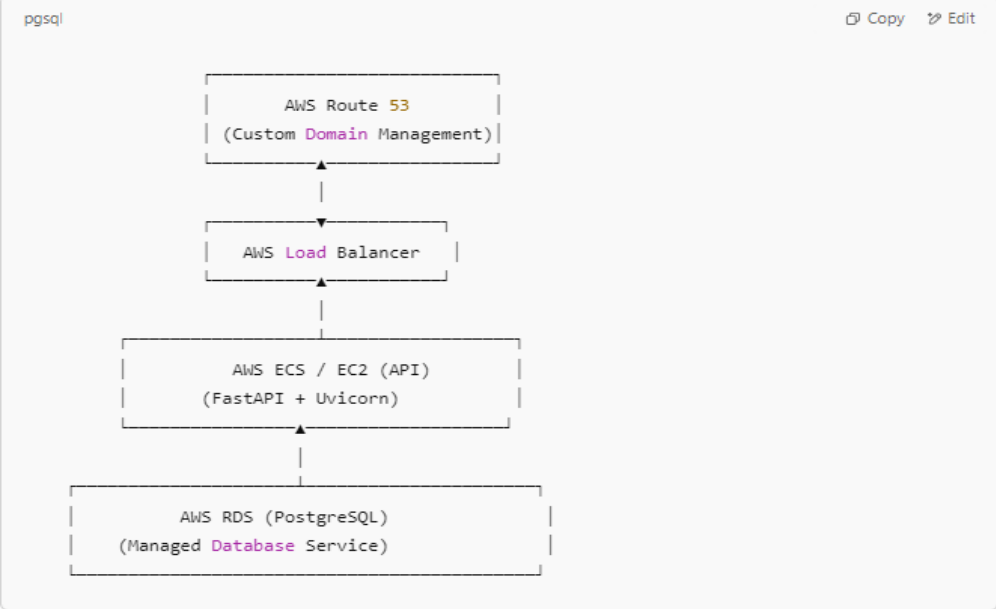
Below there are two options for deploying

Architecture Overview

We will deploy the application using AWS services, including:

- EC2 / ECS (for API Hosting)
- RDS (PostgreSQL) (for Database)
- S3 (for file storage, if needed)
- CloudWatch (for monitoring/logging)
- ALB (Application Load Balancer) (for traffic routing)
- Route 53 (for domain management)
- IAM Roles (for security)
- AWS CloudWatch: Logs API calls & performance monitoring.

Diagram: AWS Deployment Architecture



Alternative AWS Deployment Architecture: Serverless with AWS Lambda & API Gateway

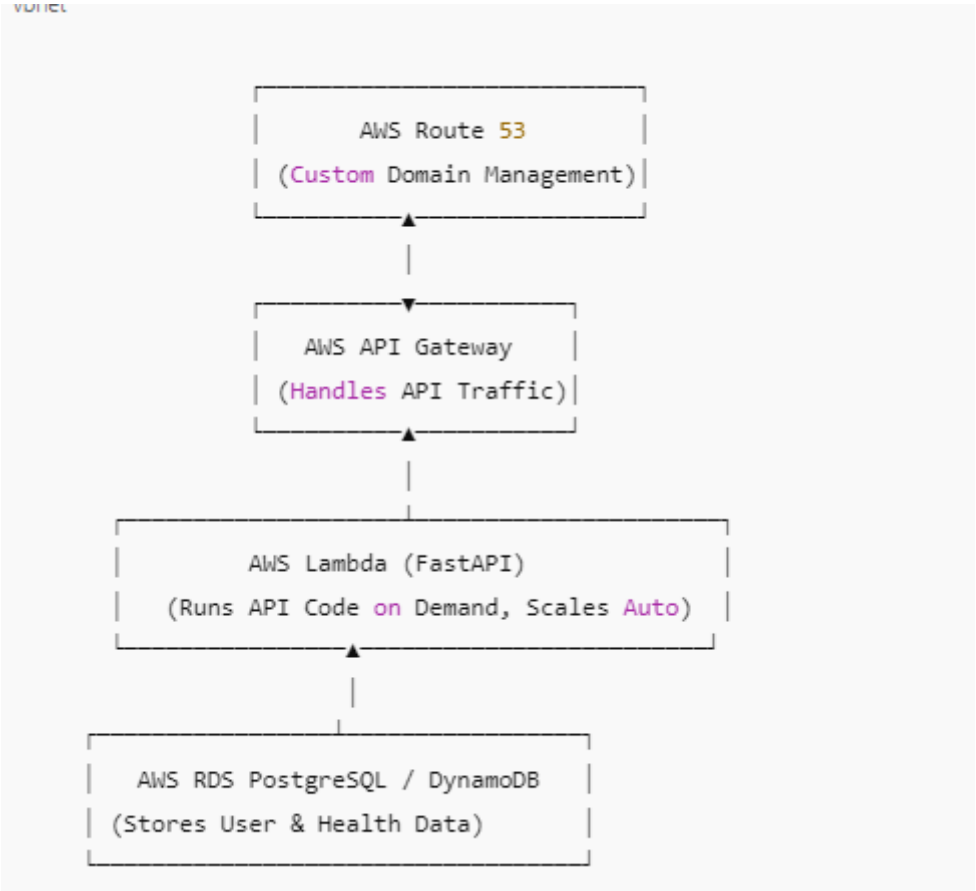
Instead of deploying our FastAPI application on EC2 or ECS, we can use a serverless approach with AWS Lambda, API Gateway, and DynamoDB/PostgreSQL on RDS.

Architecture Overview:

This fully serverless deployment includes:

- AWS Lambda: Runs FastAPI as a function, scaling automatically.
- AWS API Gateway: Routes external HTTP requests to Lambda.
- AWS RDS PostgreSQL (or DynamoDB): Stores application data.
- AWS S3: Stores user-uploaded files (if needed).
- AWS CloudWatch: Logs API calls & performance monitoring.

Diagram: AWS Serverless Deployment Architecture



Scaling & Troubleshooting

How would you approach diagnosing and solving this problem?

When an application scales rapidly, it can run into issues like inaccurate health scores, delayed API responses, and crashes under load. To systematically diagnose and resolve these problems, we will follow a structured debugging and optimization process.

Step 1: Diagnose the Issues

Health Scores Are Inaccurate

Possible Causes:

1. Data inconsistency in the database.
2. Incorrect calculations or floating-point precision errors.
3. Race conditions when updating health scores concurrently.
4. Time zone issues causing incorrect data aggregation.

How to Debug:

- Enable Logging for Score Calculation
    - Log inputs, intermediate values, and final health score computations.
- Run SQL Queries to Check Data Integrity
- Compare Calculated Scores with Expected Values
    - Run test cases to compare actual vs expected results.

We can use database transactions (BEGIN TRANSACTION; COMMIT;) to avoid race conditions.

We need to Ensure datetime consistency (use UTC timestamps in all operations).

We can implement background recalculations using AWS Lambda, Celery, or a similar worker.

API Responses Are Delayed

Possible Causes:

1. Slow database queries (inefficient indexing, full-table scans).
2. Too many API calls per request (e.g., fetching user health data using multiple queries).
3. Blocking operations (e.g., waiting for I/O).
4. High CPU or memory usage on the API server.

How to Debug:

- Enable SQL Query Logging & Analyze Slow Queries
- Check for full table scans and missing indexes.
- Use Profiling Tools (e.g., py-spy, cProfile)
- Check API Latency in AWS CloudWatch or Prometheus

We can solve that with:

Optimize SQL queries:

- Add indexes on frequently queried columns.
- Use denormalized tables for fast reads.
- Replace multiple queries with joins.

Use Caching (Redis / AWS ElastiCache):

- Cache frequent health score calculations.
- Cache recent API responses.

Optimize API Calls:

- Use batch queries instead of multiple small queries.
- Implement GraphQL or gRPC for efficient data retrieval.

Implement Load Balancing:

- Deploy multiple instances behind an AWS Load Balancer.
- Use AWS Lambda for event-driven processing.

Application Occasionally Crashes Under Load

Possible Causes:

1. Memory leaks in API processes.
2. Unoptimized database connections (too many open connections).
3. Insufficient server resources (CPU, RAM limits reached).
4. Unhandled exceptions crashing the server.

How to Debug:

- Check API Server Logs (/var/log/uvicorn.log or AWS CloudWatch).
- Analyze Memory Usage
- Monitor Database Connection Usage

We can solve that with:

Use Connection Pooling (SQLAlchemy & PostgreSQL)

Enable Auto-Scaling in AWS (EC2 / ECS)

Optimize Background Jobs (Celery / AWS Lambda)

- Move heavy health score computations to a background worker.

Increase Server Capacity

- Use AWS RDS Read Replicas for database scaling.
- Deploy FastAPI with Gunicorn + Uvicorn workers for concurrency.

Issue	How to Debug	Fixes
Health Scores Are Inaccurate	Enable logging, check database consistency, debug calculations	Fix race conditions, use UTC timestamps, implement async recalculations
API Responses Are Slow	Check SQL query performance, use profiling tools, analyze API logs	Optimize queries, use caching (Redis), batch API calls, enable async database queries
Application Crashes Under Load	Monitor server logs, check memory usage, monitor database connections	Use connection pooling, auto-scaling, optimize background tasks, increase server capacity

How would you design a long-term plan to make the system resilient to future scalability challenges?

Long-Term Plan for Scalability and Resilience in the Health Tracker API

As the Health Tracker API continues to grow, we must ensure scalability, resilience, and high availability. Below is a long-term plan divided into key areas, focusing on database optimization, application architecture, caching, monitoring, and infrastructure automation.

Database Scaling & Optimization:

Goal: Prevent slow queries and ensure high availability as data grows.

Short-Term (Now)

Optimize Database Queries:

- Use indexes on frequently queried fields

Optimize SQL joins and use denormalization for fast reads

Implement connection pooling

Enable Read Replicas for Load Distribution:

- Use AWS RDS Read Replicas to offload read-heavy queries

Implement Query Caching:

- Store frequently used health scores in Redis (ElastiCache)

Mid-Term (6 Months)

Introducing a NoSQL Database (DynamoDB) for High-Volume Data:

- Store time-series health data in DynamoDB for faster lookups.

Partitioning Strategy for PostgreSQL:

- Sharding user data across multiple databases.
- Partitioning tables by user ID to improve query performance.

Use CDC (Change Data Capture) for Real-Time Processing:

- Enable AWS Kinesis or Kafka to stream live health data.

Long-Term (1+ Year)

Move to a Multi-Region Setup:

- Deploy AWS RDS Global Databases for cross-region replication.

Automated Failover & Disaster Recovery:

- Use AWS Route 53 to redirect traffic in case of regional failures.

Application Architecture Improvements

Goal: Ensure high availability and scale without downtime.

Short-Term (Now)

Deploy a Load-Balanced API:

- Use AWS ALB (Application Load Balancer) to distribute traffic.
- Run multiple API instances in AWS ECS (Fargate).
-

Use Asynchronous Background Processing:

- Offload health score calculations to Celery / AWS Lambda.

Optimize API for Performance:

- Use gRPC instead of REST for low-latency communication.

Mid-Term (6 Months)

Implement Microservices:

- Split user management, activity tracking, and health scoring into independent services.

Introduce a GraphQL API:

- Fetch only required data instead of large API responses.

Move to a Serverless Architecture:

- Convert health score calculations into AWS Lambda functions.

Long-Term (1+ Year)

Multi-Cloud Deployment:

- Deploy on both AWS and Google Cloud for failover protection.

Use Kubernetes for Container Orchestration:

- Deploy API on AWS EKS (Kubernetes) for auto-scaling.

Caching & Performance Optimization:

Goal: Reduce load on the database and improve response times.

Short-Term (Now)

Enable Redis Caching for API Responses:

- Store recent health scores in Redis.

Optimize API Rate Limits:

- Use AWS API Gateway with rate limiting.

Enable Data Compression:

- Compress API responses using gzip.

Mid-Term (6 Months)

Use Edge Caching (CDN):

- Store static API responses in AWS CloudFront.

Optimize Background Tasks:

- Reduce Celery worker load by scheduling non-critical tasks.

Long-Term (1+ Year)

AI-Based Caching

- Predict frequent queries and preload them into cache.

Infrastructure Automation & CI/CD

Goal: Automate deployments and reduce operational overhead.

Short-Term (Now)

Set Up CI/CD with GitHub Actions & AWS CodeDeploy

- Automate deployments with GitHub Actions

Implement Infrastructure as Code (Terraform)

- Automate AWS setup using Terraform.

Enable Auto-Scaling

- Use AWS Auto Scaling to adjust API instances.

Mid-Term (6 Months)

Blue-Green Deployments

- Ensure zero-downtime deployments.

Implement Feature Flags

- Deploy new features gradually using AWS AppConfig.

Long-Term (1+ Year)

Self-Healing Infrastructure

- Auto-restart failed API instances.

Full Automation with AWS CDK

- Convert infrastructure setup into AWS CDK scripts.

Monitoring & Security Enhancements

Goal: Proactively detect issues and prevent security threats.

Short-Term (Now)

Enable AWS CloudWatch Logging

- Monitor API performance using

Set Up Sentry for Error Tracking

- Catch API crashes in real time.

Enable AWS WAF (Web Application Firewall)

- Protect against DDoS and SQL injection attacks.

Mid-Term (6 Months)

AI-Based Anomaly Detection

- Use AWS GuardDuty to detect security threats.

Implement API Key Management

- Secure API keys using AWS Secrets Manager.

Long-Term (1+ Year)

Automated Security Patching

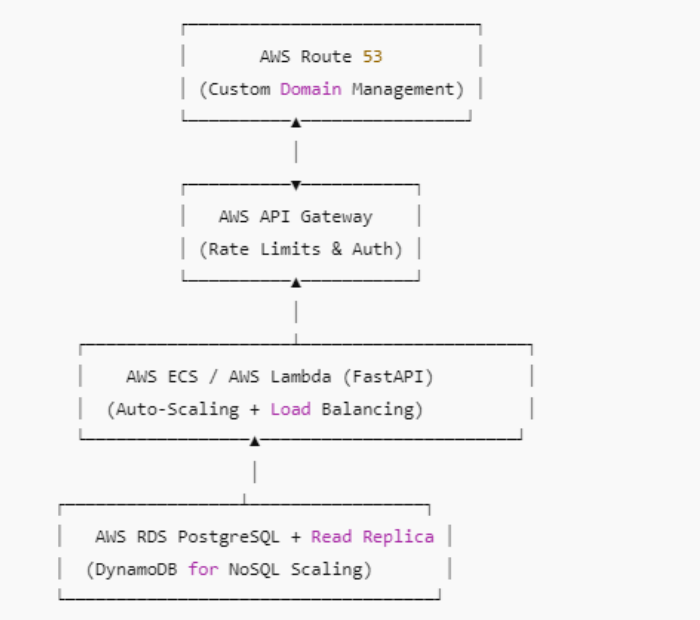
- Use AWS Systems Manager for automatic updates.

Implement Zero Trust Architecture

- Enforce strict IAM policies for microservices.

Final Scalable Architecture

Diagram: Future Scalable Architecture



Summary of the Long-Term Scalability Plan

Timeframe	Focus	Key Actions
Now	Database & Performance	Optimize queries, enable caching (Redis), add auto-scaling
6 Months	Microservices & Infrastructure	Split into microservices, CI/CD automation, feature flags
1+ Year	Global Scaling & AI-Based Optimization	Multi-cloud, AI caching, automated security