Imagine the health tracker application has become wildly popular, gaining thousands of new users every day. However, users start reporting the following issues:

- Health scores are inaccurate.
- API responses are delayed.
- The application occasionally crashes under load.

Causes:

1. Inaccuracy of Health Metrics

Data inaccuracies may arise due to:

- Errors in data collection from devices.
- Data loss during transmission.
- Incorrect processing algorithms.

2. High API Latency

API delays may be caused by:

- Slow database queries.
- Lack of caching.
- A large number of parallel requests.
- Infrastructure issues.

3. Application Freezes Under Load

Causes of freezes:

- High server load.
- Bottlenecks in data processing.
- Inefficient architecture.
- Insufficient resources.
- Blocking operations.

Diagnosis and Solution to This Problem

Health scores depend on the data entering the system (physical activity, sleep, test results, etc.). It is necessary to check whether the data is being received correctly and in the required format. Logs of incoming data should be reviewed for errors or missing information. Some data may be lost, corrupted, or not fully processed.

If health data is coming from third-party services (e.g., integrations with medical databases), it is important to ensure that these services are functioning correctly.

If the system uses algorithms to process and analyze data (e.g., evaluating physical activity or calculating health scores based on data), the **calculation logic** on the server side should be verified. It is essential to check whether incorrect models or parameters are being used. These algorithms should be tested on various datasets to ensure accuracy.

If the algorithms or rules for calculating health scores have been recently updated, this could be causing errors. It is necessary to **review change logs** and roll back recent updates if needed.

Identifying API Latency Issues

It is crucial to track which requests are causing delays and identify **performance bottlenecks**. Determining which specific request takes too much time can be done using monitoring tools such as **Prometheus, Grafana, or Datadog**.

If the API has requests that require long processing times (e.g., health score calculations or data analysis), they should be made **asynchronous** using **message queues**.

The problem may be related to the number of simultaneous requests. Frequently requested data (e.g., activity statistics) can be **cached** using **Redis or Memcached**, which will significantly speed up API responses.

Horizontal scaling should be considered by adding additional instances of the API server or transitioning to a **microservices or serverless architecture**.

Optimizing Database Performance

Delays may result from **inefficient database queries, lack of indexing, or scalability issues**.

- If database queries are slow (e.g., retrieving historical user data), they may not be optimized.
- **Indexing and database scaling issues** can cause these delays.
- SQL queries should be reviewed, and indexes should be added if necessary.

Handling Application Freezing Issues

If the application works with **user sessions or long-lived connections**, there might be **state management issues**, leading to crashes.

Freezing may occur due to **data locking** or **resource contention**.

To diagnose these issues:

- Identify which **servers and system components** are overloaded.
- Detect **memory shortages** and **CPU resource exhaustion** using **load logs and monitoring tools** (e.g., Prometheus and Grafana).

It is essential to check whether the application exceeds memory and CPU usage limits.

Possible solutions include:

- **Auto-scaling** servers or containers (e.g., with **Kubernetes**) to increase resources as the load grows.
- If the application is monolithic, **breaking it into microservices** can improve scalability and isolate issues within individual components.
- Instead of processing requests sequentially, tasks can be **parallelized** using **multithreading or task queues**.

Long-Term Plan

As the application grows, ensuring accuracy, performance, and stability under high load requires a **scalable, resilient, and optimized architecture**.
The long-term plan includes **data integrity, API performance optimization, infrastructure scalability, and proactive monitoring**.

1. Ensure Data Accuracy and Consistency

- Implement **real-time data validation** with preprocessing layers.
- Use **anomaly detection models** to filter out incorrect data.
- Store and query time-series data efficiently using **TimeSeries databases** (TimeScaleDB, InfluxDB).

2. Optimize API Performance and Reduce Latency

- **Sharding and partitioning** to distribute requests.
- Implement **read replicas** to separate read/write workloads.
- Optimize frequently used data with **materialized views and indexing**.
- Use **Redis/Memcached** to cache API responses.
- Implement **edge caching (CDN)** to reduce global latency.
- Offload computationally expensive tasks to **background processing**.
- Use **GraphQL** to allow clients to request only the necessary data, reducing payload size.
- Deploy **API gateways** (Kong, AWS API Gateway) to manage authentication, rate limiting, and throttling.
- Implement **load balancing** (NGINX, HAProxy, AWS ALB) to distribute requests across multiple instances.

3. Improve Application Resilience and Scalability

- Use **Kubernetes** for automatic service scaling based on demand.
- **Containerize** the application with **Docker** for consistency across environments.
- Implement **graceful degradation strategies** (e.g., temporarily disabling non-critical features under high load).
- **Decompose monolithic components into independent microservices** for better scalability.
- Utilize **message queues** (RabbitMQ, Kafka) for asynchronous processing.
- Implement **rate limiting and throttling** to handle traffic spikes.

4. Proactive Monitoring, Logging, and Alerts

- Use **Prometheus/Grafana** to track API response times, database queries, and system metrics.
- Implement **ELK stack** (Elasticsearch, Logstash, Kibana) for centralized log aggregation and analysis.
- Set up **real-time anomaly alerts** with **PagerDuty or AWS CloudWatch**.
- Conduct **regular stress tests** to evaluate system limits.

Final Conclusion

For the **Health Tracker** application to meet future demands, the system must adopt a **scalable, distributed, and fault-tolerant architecture** that ensures:

✔ **Accurate, high-quality data processing** with real-time validation.

✔ **Optimized API performance** using caching, asynchronous processing, and database tuning.

✔ **Resilient infrastructure** with auto-scaling, microservices, and load balancing.

✔ **Proactive monitoring and logging** for rapid issue detection and resolution.

Deploying the Health Tracker API on AWS (Serverless)

For deploying the **Health Tracker** backend application in a **serverless architecture** on AWS, we use:

- **AWS Lambda** to process API requests
- **Amazon API Gateway** for routing
- **Amazon RDS Serverless v2** for data storage
- **Amazon S3, CloudWatch, Secrets Manager, IAM, and VPC** for additional functionality.
- **Serverless Framework**

🔹 Deployment Architecture

- **API Gateway** – Routes requests to the appropriate AWS Lambda functions.
- **AWS Lambda** – Executes business logic and interacts with the database via SQLAlchemy.
- **Amazon RDS (Serverless v2, PostgreSQL + TimescaleDB)** – Stores all user data.
- **Amazon S3** – Stores files, reports, and other data.
- **AWS Secrets Manager** – Securely stores secret credentials (e.g., database passwords).
- **Amazon CloudWatch** – Monitors and logs AWS Lambda execution.
- **IAM Roles & Policies** – Manages access to AWS resources.
- **VPC & Security Groups** – Isolates the database for security against external access.

🔹 Deployment Steps

Database Setup (Amazon RDS Serverless v2)

Create an **Amazon RDS Serverless v2** (PostgreSQL) cluster with **TimescaleDB extension**:

- Enable **Auto Scaling**
- Deploy in a **VPC** for security
- Create a **Security Group** that allows access only from AWS Lambda
- Enable **automatic scaling**, **regular backups**, and **replication** for high availability.

Create RDS Database

```
aws rds create-db-instance \
 --db-instance-identifier health-tracker-db \
 --db-instance-class db.t3.medium \
 --engine postgresql \
 --engine-version 17 \
 --allocated-storage 50 \
 --storage-type gp2 \
 --publicly-accessible false \
 --vpc-security-group-ids sg-xxxxxxxx \
 --db-subnet-group-name my-subnet-group \
 --serverless-v2-scaling-configuration MinCapacity=0.5,MaxCapacity=4  \
 --database-name health_tracker \
 --master-username admin \
 --master-user-password mysecurepassword
```

Enable TimescaleDB

```
CREATE EXTENSION IF NOT EXISTS timescaledb;
```

Use AWS Secrets Manager for Secure Credential Storage

```
aws secretsmanager create-secret --name health-tracker-db-creds \
--secret-string '{"username":"admin", "password":"mysecurepassword",
"host":"healthtracker-db.us-east-1.rds.amazonaws.com/health_tracker"}'
```

🔹 Preparing FastAPI for AWS Lambda

Since **FastAPI** does not natively run on AWS Lambda, we use **Mangum**:

```
pip install fastapi mangum
```

Modify `main.py` to support AWS Lambda

```
from fastapi import FastAPI
from mangum import Mangum
app = FastAPI()
@app.get("/")
def home():
    return {"message": "Health Tracker API"}
handler = Mangum(app)
```

🔹 AWS Lambda Function Grouping

We will use **six AWS Lambda functions**, each handling related endpoints:

users_lambda	User operations	`/api/users/*`
blood_tests_lambda	Blood test records	`/api/blood-tests/*`
bio_metrics_lambda	Biometric indicators	`/api/bio-metrics/*`
physical_activity_lambda	Physical activity tracking	`/api/physical-activity/*`
sleep_activity_lambda	Sleep activity tracking	`/api/sleep-activity/*`
health_score_lambda	Health score evaluation	`/api/get_health_score`

1. AWS Lambda: `users_lambda` (CRUD)

File: `lambdas/users.py`

Handles:

- `POST /api/users/`
- `GET /api/users/`
- `GET /api/users/{user_uuid}`
- `PATCH /api/users/{user_uuid}`
- `DELETE /api/users/{user_uuid}`

2. AWS Lambda: `blood_tests_lambda`(CRUD)

File: `lambdas/blood_tests.py`

Handles:

- `POST /api/blood-tests/`
- `GET /api/blood-tests/{user_uuid}`
- `GET /api/blood-tests/{user_uuid}/{test_date}`
- `PATCH /api/blood-tests/{user_uuid}/{test_date}/update`
- `DELETE /api/blood-tests/{user_uuid}/delete`

3. AWS Lambda: `bio_metrics_lambda`(CRUD)

File: `lambdas/bio_metrics.py`

Handles:

- `POST /api/bio-metrics/`
- `GET /api/bio-metrics/{user_uuid}`
- `GET /api/bio-metrics/{user_uuid}/{recorded}`
- `PATCH /api/bio-metrics/{user_uuid}/{recorded}/update`
- `DELETE /api/bio-metrics/{user_uuid}/delete`

4. AWS Lambda: `physical_activity_lambda`(CRUD)

File: `lambdas/physical_activity.py`

Handles:

- `POST /api/physical-activity/`
- `GET /api/physical-activity/{user_uuid}`
- `GET /api/physical-activity/{user_uuid}/{date}`
- `PATCH /api/physical-activity/{user_uuid}/update`
- `DELETE /api/physical-activity/{user_uuid}/delete`

5. AWS Lambda: `sleep_activity_lambda`(CRUD)

File: `lambdas/sleep_activity.py`

Handles:

- `POST /api/sleep-activity/`
- `GET /api/sleep-activity/{user_uuid}`
- `GET /api/sleep-activity/{user_uuid}/{date}`
- `PATCH /api/sleep-activity/{user_uuid}/update`
- `DELETE /api/sleep-activity/{user_uuid}/delete`

6. AWS Lambda: `health_score_lambda`

File: `lambdas/health_score.py`

Handles:

- `GET /api/get_health_score`

🔹 IAM Role for AWS Lambda

Create an **IAM Role** with permissions for:

- **Amazon RDS** (via VPC)
- **Amazon S3**
- **AWS Secrets Manager**
- **CloudWatch (for logs)**

```
aws iam create-role --role-name HealthTrackerLambdaRole \

--assume-role-policy-document file://lambda-trust-policy.json
```

IAM Policy (`lambda-trust-policy.json`)

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": { "Service": "lambda.amazonaws.com" },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

🔹 Configuring API Gateway

Create an API Gateway for Lambdas

```
aws apigatewayv2 create-api \
--name HealthTrackerAPI \
--protocol-type HTTP \
--target arn:aws:lambda:us-east-1:123456789012:function:users_lambda
```

And add Routes

```
aws apigatewayv2 create-route \
--api-id API_ID \
--route-key "/api/users" \
--target "integrations/LAMBDA_INTEGRATION_ID"
```

🔹 Logging AWS Lambda Execution to CloudWatch

```
aws logs create-log-group --log-group-name /aws/lambda/health-tracker
```

🔹 Creating an S3 Bucket for Data Storage

```
aws s3api create-bucket --bucket health-tracker-data --region us-east-1
```

🔹 Deploying with Serverless Framework

Install Serverless Framework

```
npm install -g serverless
```

Create a New Project

```
serverless create --template aws-python --path health-tracker
cd health-tracker
```

Configure `serverless.yml`

```
service: health-tracker-api
provider:
  name: aws
  runtime: python3.12
  region: us-east-1
  memorySize: 256
  timeout: 10
environment:
    DATABASE_URL: ${ssm:/health-tracker/db-url}
    SECRET_KEY: ${ssm:/health-tracker/secret-key}
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - "rds-db:connect"
        - "secretsmanager:GetSecretValue"
      Resource: "*"
```

```
functions:
  app:
    handler: lambdas/main.handler
    events:
      - http:
          path: /
          method: any
          cors: true
  users_lambda:
    handler: lambdas/users.handler
    events:
      - httpApi:
          path: /api/users/{proxy+}
          method: any
          cors: true
  blood_tests_lambda:
    handler: lambdas/blood_tests.handler
    events:
      - httpApi:
          path: /api/blood-tests/{proxy+}
          method: any
          cors: true
  bio_metrics_lambda:
    handler: lambdas/bio_metrics.handler
    events:
      - httpApi:
```

```
          path: /api/bio-metrics/{proxy+}
          method: any
          cors: true
  physical_activity_lambda:
    handler: lambdas/physical_activity.handler
    events:
      - httpApi:
          path: /api/physical-activity/{proxy+}
          method: any
          cors: true
  sleep_activity_lambda:
    handler: lambdas/sleep_activity.handler
    events:
      - httpApi:
          path: /api/sleep-activity/{proxy+}
          method: any
          cors: true
  health_score_lambda:
    handler: lambdas/health_score.handler
    events:
      - httpApi:
          path: /api/get_health_score
          method: get
resources:
  Resources:
    HealthTrackerDB:
```

```
       Type: AWS::RDS::DBInstance
      Properties:
          DBInstanceIdentifier: health-tracker-db
               Engine: postgresql
               DBInstanceClass: db.serverless
               AllocatedStorage: 20
                    VPCSecurityGroups:
                    - !Ref HealthTrackerSecurityGroup
          EngineMode: serverless
        MasterUsername: !Sub
"{{resolve:secretsmanager:/health-tracker/db-user}}"
        MasterUserPassword: !Sub
"{{resolve:secretsmanager:/health-tracker/db-password}}"
HealthTrackerSecurityGroup:
      Type: AWS::EC2::SecurityGroup
      Properties:
        GroupDescription: Security Group for RDS
        VpcId: vpc-123456
        SecurityGroupIngress:
          - IpProtocol: tcp
            FromPort: 5432
            ToPort: 5432
            CidrIp: 0.0.0.0/0
plugins:
  - serverless-python-requirements
```

Deploying API to AWS

```
serverless deploy
```