DDoS attacks and rate-limit brute force

1. Mitigating DDoS Attacks
    - Cloud-based WAF services such as AWS WAF, Cloudflare, or Akamai can absorb large-scale DDoS traffic.
    - Configure the WAF to filter common attack patterns, such as large HTTP requests, frequent requests, or malformed packets.
2. Rate Limiting on Requests
    - Implement rate limiting to block clients that make an excessive number of requests in a short period.
    - NGINX can be used to apply rate limiting based on IP or user-specific information.

Architecture Decisions

Node.js and Express for the Backend:

- Chosen for its asynchronous, event-driven nature which handles high-concurrency efficiently.
- Express is a lightweight, unopinionated web framework that enables rapid development of APIs.

PostgreSQL as the Database:

- Relational database chosen for managing user data (e.g., username, email, hashed passwords).
- Postgres is scalable, ACID-compliant, and provides strong support for SQL features such as complex queries, indexing, and joins.

Redis as Distributed Session Store:

- Redis is used to store session data (like session tokens) because it's highly performant and supports in-memory data storage with persistence.
- Provides scalability and low-latency access, especially in a distributed system where multiple instances of the service may be running.

JWT for Session Management:

- JWT (JSON Web Tokens) is used for secure session tokens. Tokens are signed and can optionally be encrypted, providing a secure way to validate and store session information on the client-side.
- JWTs are stateless and reduce the need for server-side session storage, although Redis is still used for maintaining active sessions.

Security and Rate-Limiting:

- bcrypt is used for password hashing to secure user credentials.
- express-rate-limit is used to apply rate limits on sensitive routes (e.g., login and registration) to mitigate brute-force attacks.
- HTTPS/TLS is assumed to ensure secure data transmission between the client and server.

API Endpoints:

- /register: Registers new users.
- /login: Authenticates a user, creates a session, and issues a JWT.

- /session/{sessionId}: Validates the session by checking the JWT token and Redis store.
- /logout: Invalidates a session by deleting the session data from Redis.

Performance Considerations

1. Redis for Session Caching:
    - Redis is used as an in-memory cache for sessions to reduce latency. Redis provides O(1) complexity for read/write operations, making it optimal for session storage.
    - Using Redis in a distributed environment allows scaling across multiple nodes with synchronized session data.
2. JWT for Reduced Server Load:
    - JWT is used for stateless session management, offloading session validation to the client by using tokens. This reduces the burden on Redis, as tokens only need to be checked for validity and expiration.
    - Token validation is computationally lightweight, involving checking the signature and expiry.
3. Database Indexing:
    - PostgreSQL uses indexes on key fields such as username and email to optimize query performance, especially for frequent operations like login and registration.
    - Implementing connection pooling to reuse database connections and minimize overhead.
4. Load Balancing and Horizontal Scaling:
    - With Redis handling session storage and JWT handling stateless authentication, the backend can be easily scaled horizontally by adding more instances behind a load balancer.
    - This ensures the system can handle increased traffic without bottlenecks.
5. Asynchronous Processing in Node.js:
    - Using asynchronous I/O in Node.js ensures that the service can handle multiple requests concurrently, without being blocked by any individual request.

Achieving Scalability

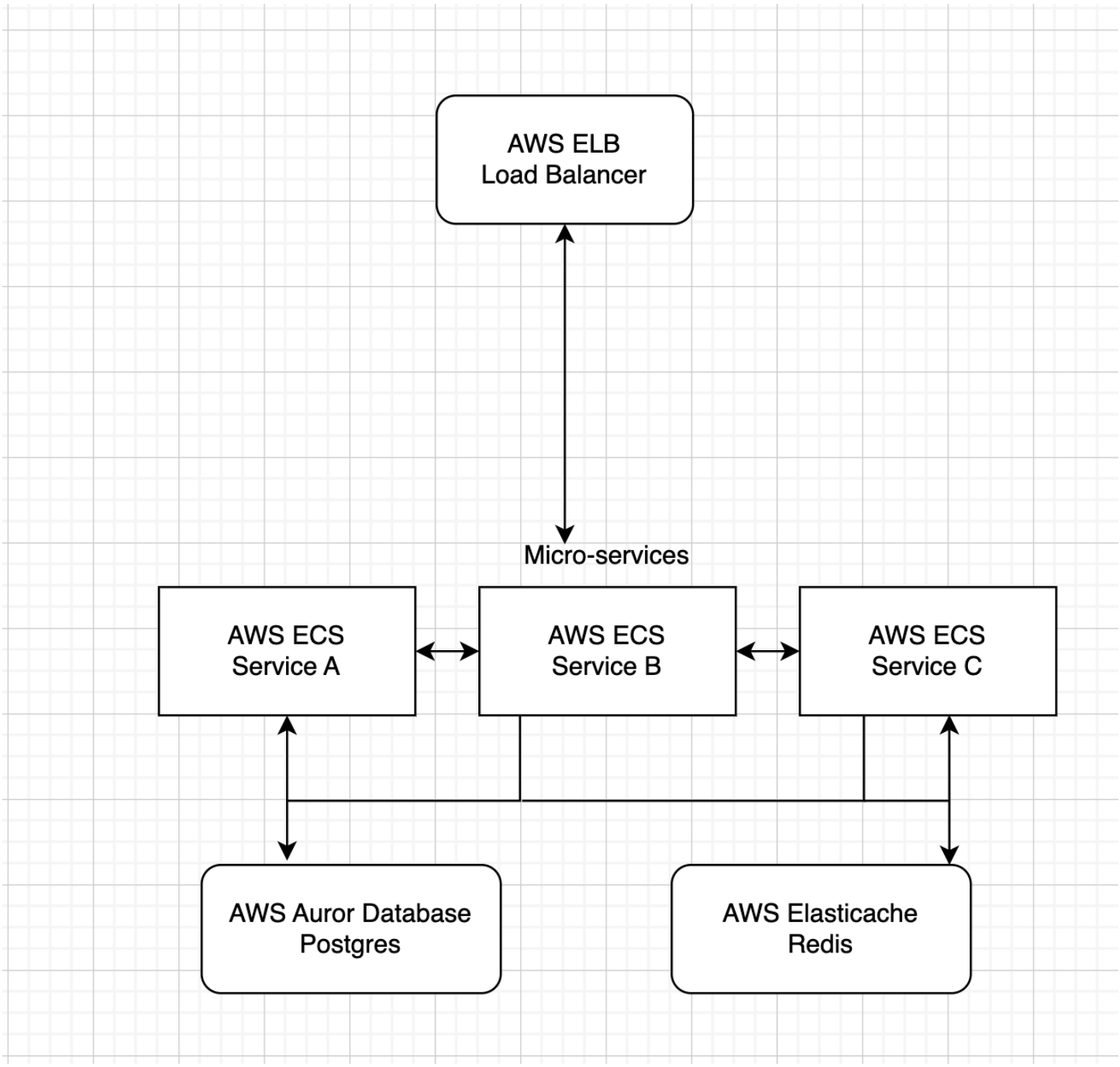
1. Distributed Session Management:
    - Redis is used for distributed session management, allowing the microservice to be horizontally scalable. Even with multiple instances of the service running behind a load balancer, session data remains consistent across all instances.
    - Redis can be clustered for further scalability, ensuring that the system can handle increasing loads as user sessions grow.
2. Stateless Authentication via JWT:
    - By using JWT, authentication becomes stateless. This enables the system to scale effortlessly, as session management is offloaded to the client, reducing the need for server-side validation.
    - JWTs are self-contained, allowing microservices to validate tokens without requiring access to a central session store, making the system more scalable.
3. Horizontal Scaling with Load Balancing:
    - The architecture supports horizontal scaling by adding more instances of the service and balancing traffic across them using a load balancer like NGINX or cloud-based solutions (AWS ELB, GCP Load Balancer).
    - Microservices can be deployed across multiple instances, improving fault tolerance and scaling capacity.
4. Database Connection Pooling:

    - Using connection pooling for PostgreSQL reduces overhead by reusing existing connections rather than establishing new ones for every request.
    - The database can be scaled vertically by adding more CPU, memory, or storage, or horizontally through read replicas to distribute the load.
5. API Gateway and Rate-Limiting:
    - An API gateway (like Kong or AWS API Gateway) can handle authentication, rate-limiting, and routing, providing a first line of defense against DDoS attacks and brute-force login attempts.
    - Rate-limiting prevents abuse and protects the backend from being overwhelmed by excessive requests.
6. Auto-scaling and Monitoring:
    - Auto-scaling policies can be configured using cloud infrastructure like AWS or Google Cloud to automatically add or remove instances based on CPU, memory, or request load.
    - Monitoring tools like Prometheus and Grafana are used to track performance metrics (latency, request rate, etc.) and trigger alerts when thresholds are breached.

Final Architecture

- Node.js/Express: Handles API requests and business logic.
- PostgreSQL: Stores user data and session history.
- Redis: Distributed in-memory session store for fast retrieval of session data.
- JWT: Provides stateless, secure authentication tokens.
- Load Balancer: Distributes traffic across multiple instances.
- Rate-Limiting: Protects against brute-force attacks and prevents abuse.
- Auto-scaling: Automatically adjusts system capacity based on traffic.

By using distributed session storage, stateless JWT authentication, and scalable infrastructure, this architecture provides both high performance and the ability to scale with demand.


AWS ELB
Load Balancer
Micro-services
AWS ECS
Service A
AWS ECS
Service B
AWS ECS
Service C
AWS Auror Database
Postgres
AWS Elasticache
Redis