Documento de Arquitetura de Software

1. Visão Geral

1.1 Objetivo

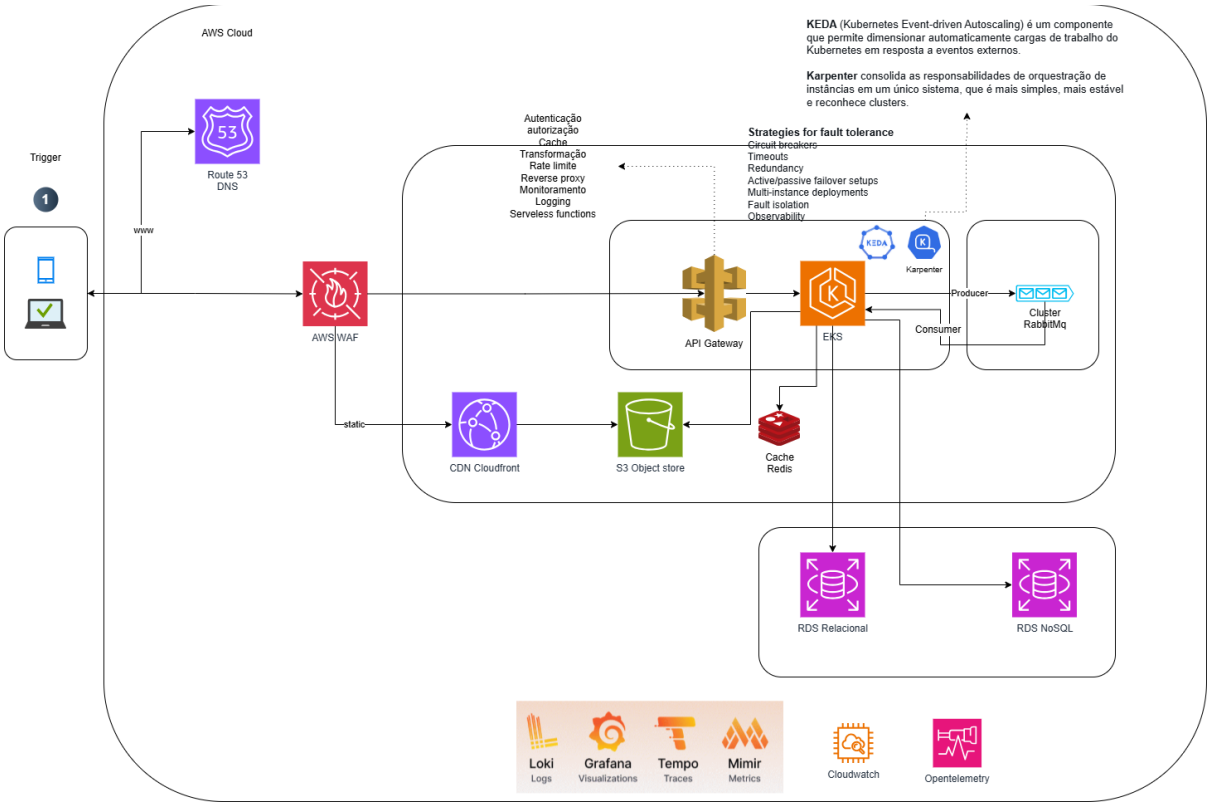
Este documento descreve a arquitetura de software de um sistema distribuído, escalável e tolerante a falhas baseado em **Clean Architecture**, princípios **SOLID**, **Clean Code**, **Event-Driven Architecture (EDA)** e **Event Sourcing**. A solução será implantada em um ambiente **Kubernetes**, utilizando **KEDA**, **Karpenter**, **Service Mesh** e **Service Discovery**.

1.2 Requisitos Arquiteturais

- **Escalabilidade**: Capacidade de escalar dinamicamente com base na demanda.
- **Tolerância a Falhas**: Garantia de resiliência contra falhas de serviço e infraestrutura.
- **Baixo Acoplamento**: Independência entre módulos para facilitar manutenção e evolução.
- **Alta Disponibilidade**: Manutenção da disponibilidade do sistema mesmo sob cargas elevadas.
- **Observabilidade**: Capacidade de rastrear e reproduzir eventos do sistema.

2. Arquitetura

2.1 Desenho da Solução



2.2.1 Detalhamento do desenho de arquitetura

Explicação por Componentes

1. Trigger (Usuário)

- O fluxo se inicia quando um usuário acessa a aplicação por meio de um navegador ou dispositivo móvel.

2. DNS e Segurança

- **Route 53**: Serviço de DNS que resolve o domínio da aplicação.
- **AWS WAF (Web Application Firewall)**: Protege contra ataques como SQL Injection e XSS.

3. Entrega de Conteúdo Estático

- **CDN CloudFront**: Distribui conteúdo estático (HTML, CSS, JS, imagens) globalmente, melhorando a performance.
- **S3 Object Store**: Armazena arquivos estáticos e permite escalabilidade.

4. API Gateway

- Encaminha requisições para o backend.
- Implementa autenticação, autorização, caching, rate limiting e transformação de payload.

5. Backend (EKS - Kubernetes)

- **Amazon EKS (Elastic Kubernetes Service)**: Orquestra os microserviços e aplicações backend.
- **KEDA (Kubernetes Event-driven Autoscaling)**: Escala os workloads automaticamente com base em eventos.
- **Karpenter**: Gerencia a alocação de nós no cluster Kubernetes.

6. Banco de Dados e Cache

- **Redis**: Cache distribuído para reduzir a latência e melhorar o desempenho das requisições.
- **RDS Relacional**: Banco de dados relacional para armazenar dados transacionais.

- **RDS NoSQL**: Banco NoSQL para armazenar documentos, logs ou informações sem estrutura fixa.

7. Mensageria

- **RabbitMQ Cluster**: Implementa filas de mensagens, permitindo comunicação assíncrona entre os serviços.
- **Producer/Consumer**: Os serviços no EKS produzem e consomem mensagens para processamento assíncrono.

8. Monitoramento e Observabilidade

- **CloudWatch**: Monitora logs, métricas e eventos da infraestrutura.
- **OpenTelemetry**: Coleta e exporta dados de rastreamento distribuído.
- **Grafana, Loki, Tempo, Mimir**: Ferramentas para observabilidade:
  - **Grafana**: Visualizações de métricas.
  - **Loki**: Coleta e análise de logs.
  - **Tempo**: Rastreamento de transações distribuídas.
  - **Mimir**: Armazena métricas de longo prazo.

Resiliência e Estratégias de Tolerância a Falhas

A arquitetura implementa:

1. Circuit Breakers

Componentes:

- **API Gateway** → Pode implementar circuit breakers para evitar sobrecarga de serviços downstream.
- **EKS (Kubernetes)** → Implementação com Istio ou Linkerd para controle de falhas.

2. Timeouts

Componentes:

- **API Gateway** → Configuração de timeouts para chamadas de APIs.
- **EKS (Kubernetes)** → Configuração de timeouts nos microserviços com service mesh (ex.: Istio).
- **RabbitMQ** → Configuração de timeouts nas filas para evitar bloqueios indesejados.

3. Redundância

Componentes:

- **Route 53 (DNS)** → Pode configurar failover entre diferentes regiões da AWS.
- **EKS (Kubernetes)** → Execução de múltiplas réplicas de pods em diferentes nós.
- **RDS Relacional e RDS NoSQL** → Implementação de réplicas e multi-AZ (zonas de disponibilidade).
- **Redis (Cache)** → Implementação de cluster para failover automático.

4. Active/Passive Failover

Componentes:

- **Route 53 (DNS)** → Pode realizar failover entre endpoints em caso de falha.
- **EKS (Kubernetes)** → Pode escalar automaticamente para um cluster secundário.
- **RDS Relacional e NoSQL** → Suporte a failover para réplicas em outra zona de disponibilidade.
- **RabbitMQ Cluster** → Configuração de failover entre nós do cluster.

5. Multi-Instance Deployments

Componentes:

- **EKS (Kubernetes)** → Execução de múltiplas instâncias de microserviços.
- **API Gateway** → Pode rotear chamadas para diferentes instâncias de backend.
- **Redis (Cache)** → Pode ser configurado em modo cluster para distribuição de carga.

6. Fault Isolation

Componentes:

- **EKS (Kubernetes)** → Uso de namespaces, policies e network segmentation para isolar falhas.
- **RabbitMQ** → Separação de filas de mensagens para evitar falhas em cascata.

7. Retry & Backoff

Retry & Backoff define estratégias para que, em caso de falha, o sistema tente novamente após um intervalo de tempo variável, reduzindo a sobrecarga e aumentando a chance de sucesso.

Componentes que podem implementar:

- **API Gateway** → Pode implementar políticas de retry com backoff exponencial para chamadas externas.
- **EKS (Kubernetes)** → Service meshes como Istio podem gerenciar retries inteligentes.
- **RabbitMQ** → Implementação de Dead Letter Queues (DLQ) para reprocessar mensagens falhadas com delays progressivos.
- **Redis (Cache)** → Pode implementar lógica de retry para acessos concorrentes a chaves bloqueadas.

8. Dead Letter Queue (DLQ)

O **DLQ (Dead Letter Queue)** armazena mensagens que falharam repetidamente, evitando que causem bloqueios ou falhas contínuas.

Componentes que podem implementar:

- **RabbitMQ** → Configuração de DLQs para mensagens que não puderam ser processadas.
- **SQS (caso estivesse na arquitetura)** → Pode ser configurado para armazenar mensagens falhas em uma fila separada.

- **EKS (Kubernetes)** → Pode usar DLQs para armazenar requisições falhas e tentar reprocessá-las posteriormente.

9. Load Shedding

O **Load Shedding** é uma estratégia para descartar requisições quando o sistema está sobrecarregado, priorizando as mais importantes.

Componentes que podem implementar:

- **API Gateway** → Pode rejeitar requisições com base na carga do sistema.
- **EKS (Kubernetes)** → Pode usar **HPA (Horizontal Pod Autoscaler)** para aumentar a capacidade automaticamente.
- **Redis (Cache)** → Pode rejeitar requisições quando atingir limites de memória.
- **RabbitMQ** → Pode limitar o consumo de mensagens por consumidor para evitar sobrecarga.

10. Health Checks

Os **health checks** garantem que apenas instâncias saudáveis recebam tráfego, removendo automaticamente instâncias problemáticas.

Componentes que podem implementar:

- **EKS (Kubernetes)** → Usa **liveness probes** e **readiness probes** para verificar a saúde dos pods.
- **API Gateway** → Pode verificar o estado de backends antes de encaminhar requisições.
- **RabbitMQ** → Pode verificar a disponibilidade de consumidores antes de rotear mensagens.
- **RDS Relacional e NoSQL** → Failover automático baseado em health checks.

2.2 DDD, Clean Architecture, CQRS e SOLID

O sistema seguirá os princípios da **Clean Architecture**, organizando o código em camadas:

- **Domain (Domínio)**: Contém as regras de negócio essenciais para cada entidade, objetos de valor, eventos de domínio, exceções, enums e repositórios.
- **Application (Aplicação)**: Contém os casos de uso e serviços da aplicação, orquestrando a lógica do domínio. Inclui comandos, consultas e interfaces externas.
    - **Comandos (Commands)**: São responsáveis por modificar o estado do sistema e aplicar regras de negócio. *Exemplo: Criar um pedido, atualizar um perfil de usuário*.
    - **Consultas (Queries)**: Lidam com a recuperação de informações otimizadas para leitura. *Exemplo: Buscar detalhes de um pedido, listar produtos*.
- **Infrastructure (Infraestrutura)**: Lida com preocupações externas, como bancos de dados, provedores de identidade, clientes HTTP, armazenamento em nuvem, e-mails e filas de mensagens.
- **Presentation (Apresentação)**: Representa o ponto de entrada do sistema, geralmente atuando como raiz de composição para injeção de dependências. Inclui endpoints de API, serviços DI, GraphQL, gRPC, ASP.NET, middlewares e manipulação de exceções.

Os princípios **SOLID** e **Clean Code** serão aplicados para garantir modularidade, reutilização e facilidade de manutenção. Além disso, **Domain-Driven Design (DDD)** será adotado para definir **contextos delimitados (Bounded Contexts)**, permitindo uma modelagem mais precisa do domínio e facilitando a comunicação entre equipes e serviços.

O padrão **CQRS (Command Query Responsibility Segregation)** será utilizado para separar operações de leitura e escrita, melhorando o desempenho e a escalabilidade do sistema:

- **Comandos (Commands)**: Responsáveis por modificar o estado do sistema, garantindo que a lógica de negócios seja aplicada corretamente.
- **Consultas (Queries)**: Permitem a recuperação otimizada dos dados, possibilitando diferentes modelos de leitura para melhorar a eficiência.

Exemplo: Em um sistema de pedidos, a criação e atualização de pedidos (escrita) seriam gerenciadas separadamente da recuperação de detalhes dos pedidos (leitura), permitindo que cada operação utilize um modelo de banco de dados apropriado (por exemplo, MySQL para escrita e Mongodb para leitura).

2.2 Arquitetura Event-Driven & Event Sourcing

- **Event-Driven Architecture (EDA)** será usada para comunicação entre microserviços de maneira assíncrona, promovendo escalabilidade e desacoplamento.
    - **Pattern SAGA**: Para garantir a consistência distribuída, será utilizado o padrão SAGA, onde cada etapa de uma transação é um evento independente e, em caso de falha, uma ação compensatória será executada. *Exemplo: Em um fluxo de pedido de compra, se a reserva de estoque falhar, um evento de compensação pode ser disparado para cancelar o pagamento previamente aprovado*.
    - **Outbox Pattern**: Para evitar inconsistências entre banco de dados e eventos, será utilizado o padrão Outbox, garantindo que os eventos sejam armazenados de forma transacional antes de serem publicados. *Exemplo: Um serviço de pedidos pode armazenar um evento na tabela de outbox no banco de dados antes de publicá-lo no Kafka, garantindo que, se houver falha na publicação, o evento não seja perdido e possa ser reprocessado*.
- **Event Sourcing** garantirá que todas as mudanças de estado sejam armazenadas como uma sequência de eventos imutáveis.
- **Mensageria**: RabbitMQ / Apache Kafka será utilizado como event bus para garantir resiliência e entrega confiável de mensagens.

3. Considerações Finais

A arquitetura proposta garante escalabilidade, resiliência e auditabilidade, tornando o sistema preparado para cargas imprevisíveis e garantindo alta disponibilidade. A implementação segue os padrões **DDD, Clean Architecture, CQRS e Event-Driven**, aplicando **SOLID** e **Clean Code** para boas práticas de desenvolvimento.

A arquitetura implementada segue os padrões DDD, Clean architecture, CQRS e event driven, com boas praticas de desenvolvimento SOLID e clean code. Não foram abordados nessa POC a implementação de api gateway, fault tolerance, event driven (patterns SAGA ou Outbox), event sourcing, CI/CD, infraestrutura (somente de armazenamento) e escalabilidade.