



**UNIVERSIDADE
FEDERAL DO CEARÁ**
CAMPUS DE QUIXADÁ

Universidade Federal do Ceará - *campus* Quixadá

Arquitetura de Software
Sistema de Notificação e Mensageria

Docente:
Sidartha Azevedo Lobo De Carvalho

Discentes:
Anaildo Do Nascimento Silva, 552836
Antônio Rewelli De Oliveira, 554047
Debora Silva Viana, 557337
Douglas Eduardo Dos Santos Sousa, 554929
Miquéias Bento Da Silva, 553972

Av. José de Freitas Queiroz, 5003, Quixadá - CE, 63902-580
14.09.2024

Sumário

Proposta: Sistema de Notificação e Mensageria.....	3
(Event-Driven + Serverless).....	4
Objetivo.....	4
Principais requisitos (funcionais / não-funcionais).....	4
Decisões e descrição final do sistema.....	5
1) Visão geral do sistema.....	5
2) Descrição do domínio.....	5
Principais atores.....	5
Casos de uso principais (3–7 essenciais).....	6
3) Entidades principais (modelo conceitual).....	6
4) Fluxos principais.....	6
5) Eventos principais e campos.....	7
6) Quando enviar notificações - regras.....	8
7) Processadores serverless — como chamá-los e o papel.....	8
8) Regras operacionais e requisitos não-funcionais.....	9
9) Decisões arquiteturais formalizadas.....	9
10) Itens concretos e definidos.....	9
11) Pontos de atenção / riscos e mitigação.....	10
12) Responsáveis por cada bloco.....	10
Frontend.....	11
Producer API.....	11
Infra.....	11
Documentação.....	11

Proposta: Sistema de Notificação e Mensageria

(Event-Driven + Serverless)

Objetivo

Construir um sistema que entregue notificações e mensagens (e-mail e push) usando eventos como fonte de verdade. Deve ser escalável, baixo custo operacional (serverless) e resiliente.

Principais requisitos (funcionais / não-funcionais)

Funcionais

- Receber eventos de fontes externas (APIs, outros serviços).
- Roteamento de eventos para canais (e-mail e push).
- Gerenciar templates (da mensagem enviada para o usuário) e preferências de usuário.
- Retries, DLQ (dead-letter queue) e histórico de entregas.

Não-funcionais

- Escalabilidade automática.
- Alta disponibilidade (mesmo nível que o provider serverless).
- Garantia de *at least once* com idempotência para evitar duplicidade.
- Latência configurável (tempo máximo aceitável por canal).
- Observability: logs estruturados, métricas e traces.

Decisões e descrição final do sistema

O objetivo a seguir é descrever de forma clara e objetiva o sistema que vamos documentar e implementar. Usaremos essa descrição como base para os diagramas C4 ou 4+1 e para dividir tarefas no time.

1) Visão geral do sistema

Nome (contexto): HelpDesk — Sistema de gestão de chamados com notificação event-driven.

Objetivo: permitir abertura e gestão de tickets (chamados) via frontend/backend (producer) e entregar notificações (e-mail e push) aos usuários e agentes por meio de *processadores serverless* que consomem eventos enfileirados (SQS).

Resumo técnico:

- **Frontend** (React) + **Producer API** (Spring Boot) → publica eventos.
- **Broker:** AWS SQS (filas).
- **Processadores serverless:** **funções Lambda** (ler SQS, processar e enviar notificações).
- **Storage:** DynamoDB (ou outro) para templates, logs e controle de idempotência.
- **Observability:** CloudWatch logs/metrics, correlação por **correlationId**.

Nota terminológica: quando se referir à parte “serverless”, use termos como **processadores serverless** ou **funções Lambda (processadores de eventos)** — assim fica claro que são funções sem servidor que executam lógica ao consumir eventos.

2) Descrição do domínio

Sistema para registrar e gerenciar solicitações de suporte (tickets). Usuários finais abrem chamados, agentes atendem, e o sistema notifica automaticamente envolvidos em pontos importantes do fluxo.

Principais atores

- **Usuário cliente** — abre e recebe notificações sobre o ticket.
- **Usuário técnico** — recebe notificações de atribuição e atualizações.
- **Administrador** — gerencia templates, configura canais, visualiza logs.

-
- **Sistema externo (opcional)** — integrações que também podem publicar eventos.

Casos de uso principais (3–7 essenciais)

1. **Criar Ticket** — usuário submete um formulário; gera `TicketCreated`.
 2. **Atualizar Ticket** — alterar status, descrição ou prioridade; gera `TicketUpdated`.
 3. **Atribuir Ticket** — ticket alocado para agente; gera `TicketAssigned`.
 4. **Adicionar Comentário** — gera `TicketCommented`.
 5. **Fechar / Resolver Ticket** — gera `TicketResolved`.
 6. **SLA / Alerta** — gerar `TicketSLABreached` quando prazo estourar (cron job/event).
 7. **Reabrir Ticket** — gera `TicketReopened`.
-

3) Entidades principais (modelo conceitual)

- **Ticket**: `ticketId`, `title`, `description`, `status`, `priority`, `createdAt`, `updatedAt`, `ownerId`, `assignedAgentId`.
 - **User**: `userId`, `name`, `email`, `deviceTokens` (para push), `preferences` (canais).
 - **Agent**: extensão de User com `skills`, `team`.
 - **Event**: `event`, `eventId`, `occurredAt`, `payload` (campo livre com dados do domínio).
 - **Notification**: `notificationId`, `eventId`, `target`, `channel`, `status`, `attempts`, `sentAt`.
 - **Template**: id, name, body (HTML/text), channel, version.
 - **ProcessedEventIds**: tabela para idempotência (`eventId` → `processedAt`).
-

4) Fluxos principais

1. Usuário cria ticket no Frontend → chama `POST /tickets` no Producer API.
2. Producer valida e publica evento `TicketCreated` com `eventId` único.

3. Producer envia a mensagem para **SQS** (ou publica em **SNS** (Simple Notification Service) + **SQS** se precisar fan-out).
4. **SQS** entrega a mensagem para Lambda **ingest/processor** (trigger).
5. Lambda verifica idempotência; se novo:
 - o monta notificações para canais configurados (email/push) conforme **payload**/preferências;
 - o enfileira tarefas ou chama diretamente provedores (Amazon SES, FCM - Firebase Cloud Messaging) ou outra Lambda especializada;
 - o grava **Notification** e **deliveryLog** em DynamoDB;
 - o em caso de falha, permite retry automático; após N tentativas, mensagem vai para DLQ.
6. Frontend/Admin UI consulta delivery logs via API para mostrar status.

5) Eventos principais e campos

Definam e versionem estes eventos (JSON Schema):

TicketCreated - 200 ok

```
{  
  "event": "TicketCreated",  
  "eventId": "uuid",  
  "occurredAt": "ISO8601",  
  "payload": {  
    "ticketId": "tkt-001",  
    "userId": "u-10",  
    "title": "Erro X",  
    "description": "...",  
    "channels": ["email", "push"]  
  }  
}
```

TicketAssigned, **TicketUpdated**, **TicketCommented**, **TicketResolved**,
TicketSLABreached — seguir padrão: event, eventId, occurredAt, payload com campos mínimos (ticketId, userId/agentId, summary, changedFields).

Definam **channels** e **preferences** (ex.: user pode desabilitar push).

6) Quando enviar notificações - regras

Notificações serão enviadas nas situações abaixo (mínimo obrigatório):

- **TicketCreated**: notificar usuário (confirmação de criação).
- **TicketAssigned**: notificar agente e usuário (se quiser).
- **TicketUpdated**: notificar usuário / agente dependendo do campo (ex.: status).
- **TicketCommented**: notificar participante(s) do ticket.
- **TicketResolved**: notificar usuário (resolução).
- **TicketSLABreached**: notificar gestor / agente (alerta crítico).
- **Reminders/Follow-ups**: via scheduled events (EventBridge cron → publicar **Reminder**).

Cada evento deve respeitar as preferências do usuário (canal, horário, silenciamento).

7) Processadores serverless — como chamá-los e o papel

Termo recomendado: *Processadores serverless (funções Lambda)* — descrevem a função de consumir eventos da fila e executar lógica.

Papéis:

- **Ingest / Router Lambda** — valida schema, enriquece evento (ex.: resolve user data), encaminha para *processors* específicos se usar filas por canal.
- **Processor-email** — renderiza template, envia via SES (ou simula).
- **Processor-push** — monta payload e envia via FCM/APNs (ou simula).
- **DLQ handler** — notifica admins e permite reprocessamento manual via Admin UI.
- **Audit Logger** — grava eventos processados e resultados em DynamoDB para auditoria.

Obs.: para PoC, processadores podem apenas gravar logs e simular envio; para ambiente real, usar SES/FCM.

8) Regras operacionais e requisitos não-funcionais

- **Idempotência:** registrar `eventId` em tabela `processedEventIds` antes de executar ação; ignorar duplicados.
- **At least once:** SQS garante entrega; deduplicação evita duplicidade.
- **Retry e DLQ:** configurar retries e DLQ para mensagens com falha.
- **Segurança:** Producer API autenticado (JWT/Cognito); Lambdas têm roles IAM mínimos (princípio do menor privilégio).
- **Observability:** incluir `correlationId/eventId` em logs; métricas por canal (success, fail, latency).
- **Schema Registry/Versioning:** versionar eventos (ex.: `TicketCreated.v1`) ou usar campo `schemaVersion`.
- **Privacy/retention:** definir período de retenção de logs e dados sensíveis (GDPR-like consideration).
- **Performance:** latência aceitável para notificações críticas (definir SLO simples, ex.: 95% < 5s).
- **Testes:** testes unitários, integração local (LocalStack ou mocks), e2e demo com script de publish.

9) Decisões arquiteturais formalizadas

Antes de gerar os diagramas, decidimos e registramos os seguintes ADRs:

1. **Broker:** SQS direto (evoluir para SNS+SQS se precisar fan-out).
2. **Producer stack:** Spring Boot.
3. **Linguagem das Lambdas:** [Node.js](#) ou Java (provavelmente node).
4. **Storage:** DynamoDB (por recomendações).
5. **Idempotency strategy:** tabela `processedEventIds` com TTL
6. **Retry policy & DLQ settings:** 3 número de tentativas, backoff.
7. **Event schema versioning strategy.**
8. **Auth:** Cognito or JWT + API Gateway.
9. **IaC tool:** Serverless Framework.
10. **Dev workflow:** CI/CD (GitHub Actions), segredo via GitHub Secrets.

10) Itens concretos a definidos

Definir os seguintes pontos mínimos já hoje pode gerar diagramas e documentação coerentes:

1. **Tech stack final:** Producer (Spring Boot) + Lambdas (Node.js recommended for PoC) + React frontend.
2. **Estrutura de filas:** usar 1 fila `notifications-queue` (SQS) inicialmente.
3. **Eventos a suportar v1:**
 - o `TicketCreated`,
 - o `TicketAssigned`,
 - o `TicketUpdated`,
 - o `TicketCommented`,
 - o `TicketResolved`.
4. **Campos obrigatórios do evento:**
 - o `event`,
 - o `eventId`,
 - o `occurredAt`,
 - o `payload` (com `ticketId`, `userId`, `channels`).
5. **Idempotency method:** tabela DynamoDB `processedEventIds`.
6. **Retry/DLQ policy:** 3 tentativas, DLQ após 3 falhas.
7. **Admin UI mínimos:** visualizar entregas e reprocessar DLQ.
8. **Locais de logs/monitoring:** CloudWatch + metrics custom.
9. **Nome do repositório e estrutura inicial:** conforme já definido (`producer/`, `frontend/`, `serverless/`, `processors/`, `docs/`, `schemas/`).
10. **Responsáveis por cada bloco.**
11. **Templates:** modelos de informações que são enviadas para o usuário na notificação, e que são personalizadas pelo administrador para os diferentes eventos.

11) Pontos de atenção / riscos e mitigação

- **Duplicidade de notificações** → mitigar com idempotência.
- **Perda de mensagem** → DLQ + monitoramento e alertas.
- **Queue backlog em pico** → monitorar e escalar (Lambda concurrency).
- **Erros lógicos (comprometendo ticket state)** → mantenham o producer como fonte de verdade e evitem que processadores ditem mudanças críticas no estado do ticket sem confirmação.
- **Complexidade de debug** → incluir `eventId` e `traceId` e usar dashboards CloudWatch.

12) Responsáveis por cada bloco

Frontend

- Antônio Rewelli
- Douglas Sousa
- Anaildo Nascimento

Producer API

- Miquéias Bento
- Debora Viana

Infra

- Miquéias Bento

Documentação

- Miquéias Bento