Documento de Arquitetura de Software: Sistema de Notificação de Pagamentos

1. Introdução e Contexto

1.1 Visão Geral

O "Sistema de Notificação de Pagamentos" é uma solução de middleware projetada para o Banco Azul. Seu objetivo principal é capturar informações de pagamentos processados pelo sistema legado e notificar os clientes finais via notificação *push* no aplicativo do banco.

A iniciativa visa resolver o distanciamento do cliente final e a falta de feedback imediato sobre o pagamento de parcelas de financiamento.

1.2 Objetivos de Negócio

- **Melhoria de Relacionamento:** Fornecer feedback rápido e transparente ao cliente.
- **Redução de Custos:** Reduzir em 7% o volume de acionamentos nos canais de atendimento (clientes ligando para confirmar pagamentos) e reduzir em 1% o custo operacional.
- **Novos Negócios:** Fomentar o uso do App e criar oportunidades de *cross-selling*.

1.3 Stakeholders

- **Clientes:** Recebedores da notificação.
- **Time de Marketing:** Define a comunicação e acompanha métricas via Dashboard.
- **Time de Tecnologia:** Responsável pela implementação e manutenção da integração com o legado e CRM.

2. Restrições e Direcionadores Arquiteturais

2.1 Restrições Técnicas e de Negócio

- **Legado (Hard Constraint):** O sistema de recebimentos atual é monolítico e não emite eventos; a solução deve conviver com ele até sua futura modernização.
- **Building Blocks Existentes:** Deve-se reutilizar o sistema de CRM existente para o envio efetivo do *push* e para o dashboard de métricas.
- **Cloud Provider:** Infraestrutura obrigatória em Google Cloud Platform (GCP).
- **Stack Tecnológica:** Java Spring Boot para microsserviços e MySQL para banco de dados.
- **Privacidade:** Nenhum dado sensível deve ser exposto na notificação (apenas número da parcela ou texto genérico).

2.2 Atributos de Qualidade (Requisitos Não-Funcionais)

- **Latência:** Tempo de resposta da solução deve ser inferior a 1 segundo.
- **Disponibilidade/Resiliência:** Taxa de erros aceitável inferior a 1% e suporte a retentativas automáticas (3 tentativas).
- **Escalabilidade:** O sistema deve escalar horizontalmente (novo nó) quando a CPU atingir 75% de uso ou a vazão ultrapassar 30 notificações/segundo.
- **Volumetria:** Capacidade para processar cerca de 2,4 milhões de pagamentos mensais com picos de vazão de 180 notificações por minuto.

3. Estratégia da Solução

A solução adota uma arquitetura baseada em **Eventos e Microsserviços**, utilizando o padrão de **Orquestração**. Devido às limitações do legado, foi implementada uma estratégia de *Polling* (leitura recorrente) na origem, convertendo a entrada em eventos para o restante do fluxo.

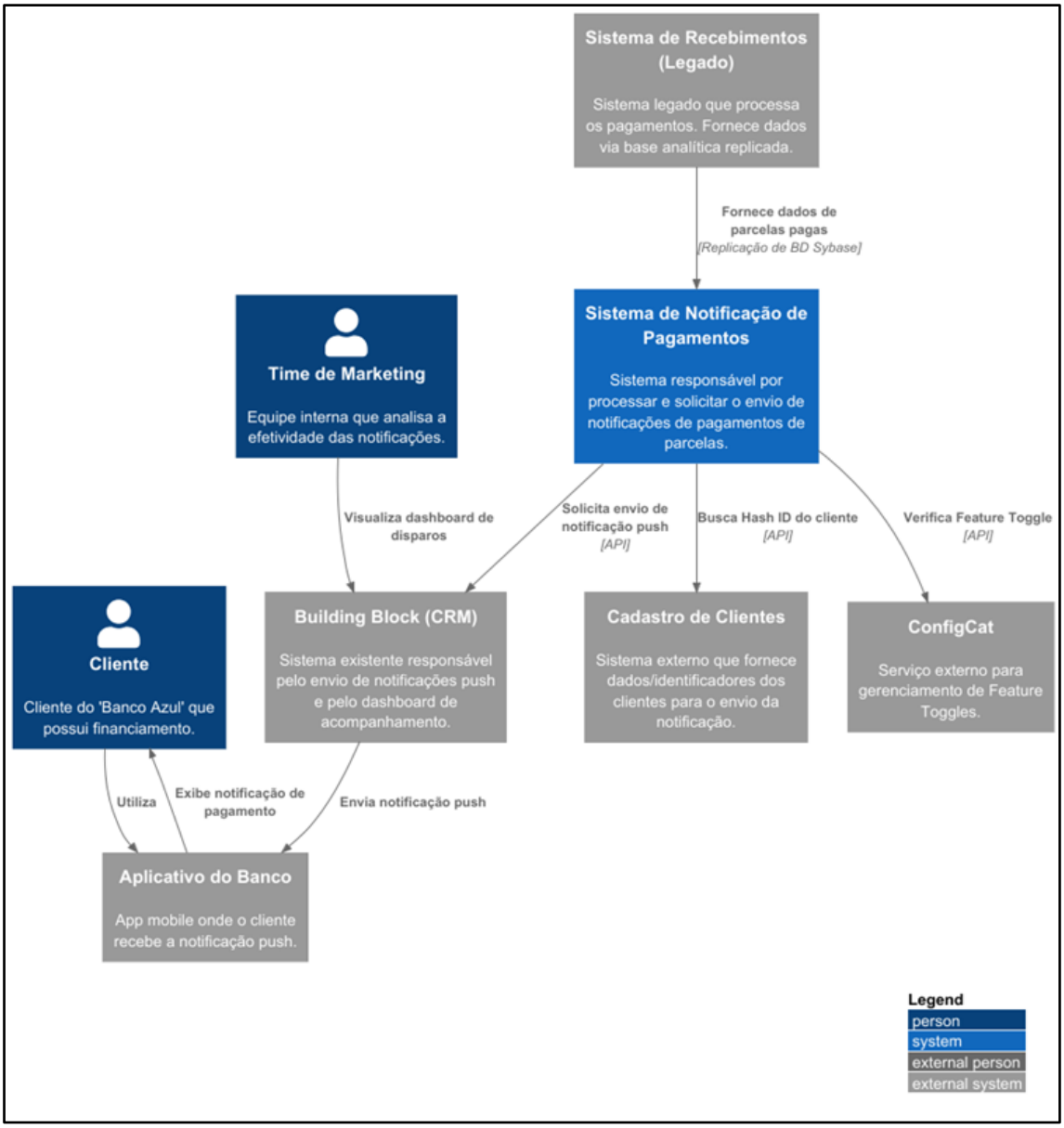
Principais Decisões (Trade-offs):

1. **Batch ETL (Polling) vs. Eventos na Origem:**
    - *Decisão:* Criar um componente de Batch ETL que lê a base analítica replicada a cada 10 minutos.
    - *Motivo:* O legado não possui *hooks* de evento e não pode ser alterado. Ler da base analítica evita impacto na performance do transacional.
2. **Orquestração vs. Coreografia:**
    - *Decisão:* Uso de um serviço central ("Orquestrador de Canais") para coordenar o fluxo.
    - *Motivo:* Facilita o monitoramento, tratamento de erros centralizado e a lógica sequencial exigida (verificar *toggle* -> buscar cliente -> enviar).
3. **Separação de Estado (Stateless vs. Stateful):**
    - *Decisão:* O Orquestrador é *stateless*. O controle de retentativas e status é delegado a um "Serviço Atômico".
    - *Motivo:* Permite escalar o orquestrador infinitamente sem complexidade de gestão de estado.

4. Visões Arquiteturais (C4 Model)

4.1 Visão de Contexto (Nível 1)

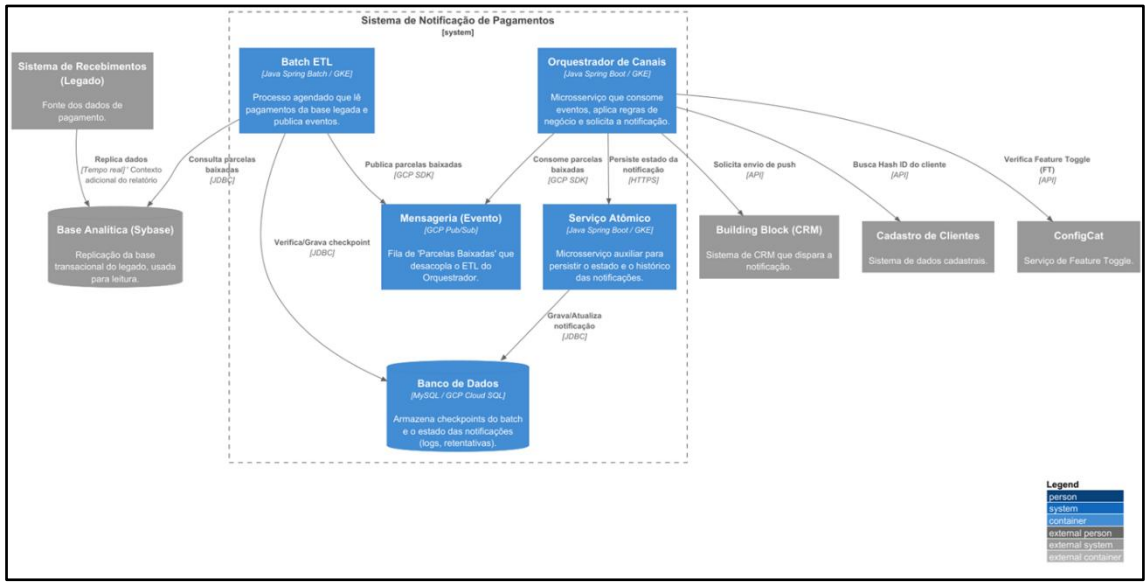
O sistema atua como um intermediário inteligente. Ele recebe dados do "Sistema de Recebimentos (Legado)", enriquece a informação consultando o "Cadastro de Clientes", verifica regras no "ConfigCat" e aciona o "Building Block (CRM)" para envio ao "Aplicativo do Banco".



4.2 Visão de Contêineres (Nível 2)

A solução é composta pelos seguintes contêineres implantados no GKE (Kubernetes):

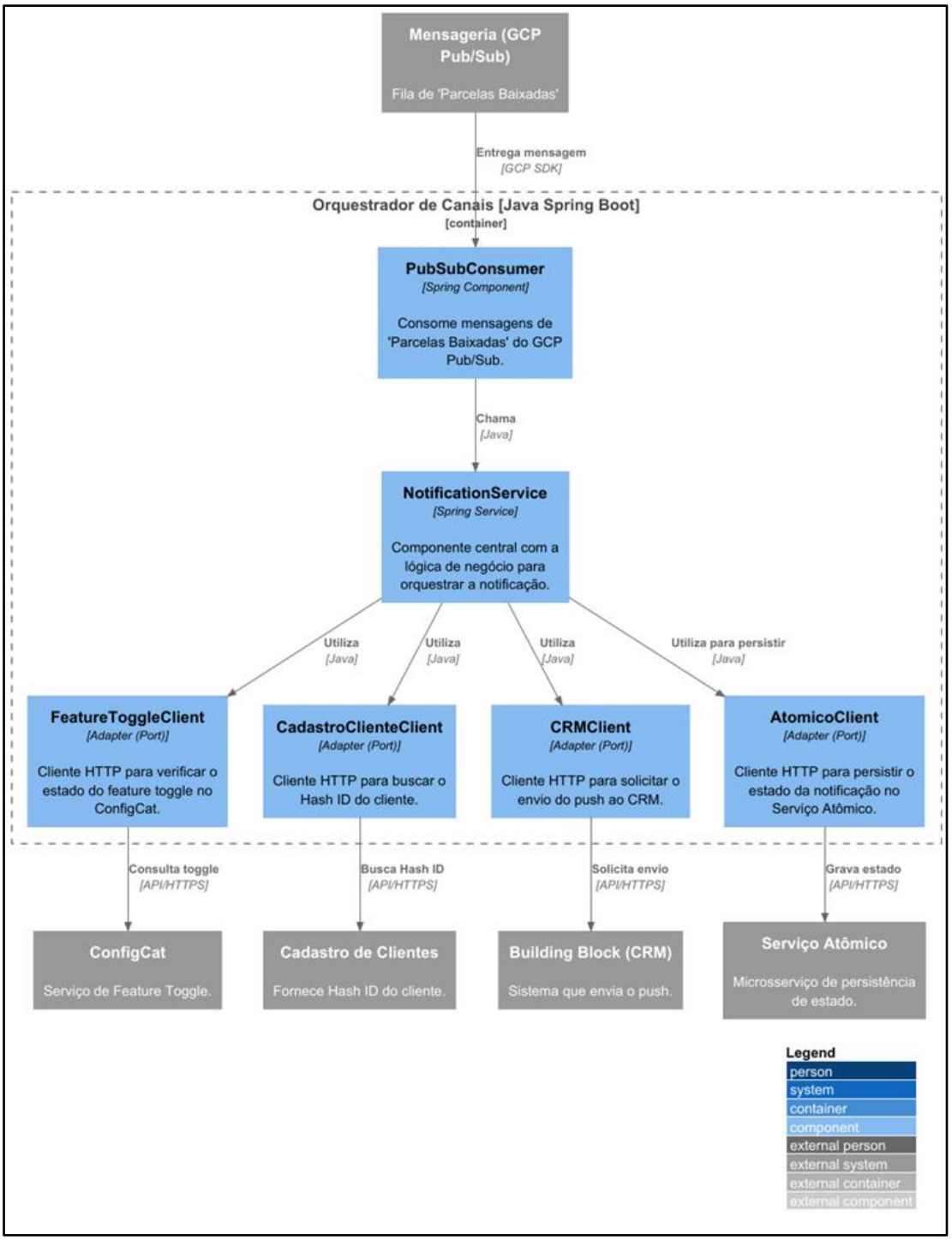
1. **Batch ETL (Java Spring Batch):** Responsável por ler as parcelas pagas na base replicada (Sybase) e publicar no tópico.
2. **Mensageria (GCP Pub/Sub):** Desacopla a leitura do processamento. Tópico de "Parcelas Baixadas" serve como evento corporativo para convivência com sistemas futuros.
3. **Orquestrador de Canais (Java Spring Boot):** Consome o evento e executa a lógica de negócio (verifica *toggle*, busca dados, solicita envio).
4. **Serviço Atômico (Java Spring Boot):** Gerencia a persistência do estado da notificação e controla as retentativas.
5. **Banco de Dados (MySQL):** Persistência dos dados de controle e auditoria.



4.3 Visão de Componentes (Nível 3) - Foco: Orquestrador

O "Orquestrador de Canais" segue o padrão **Ports and Adapters (Hexagonal)** para garantir testabilidade e desacoplamento.

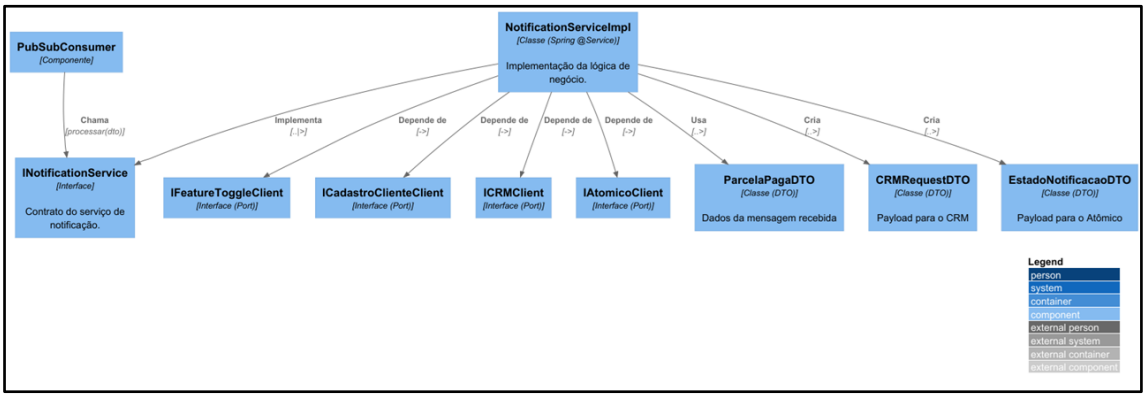
- **NotificationService:** O *core* da lógica de negócio. Não depende de implementações concretas.
- **Adapters (Clients):** Interfaces para CRMClient, FeatureToggleClient, CadastroClienteClient e AtomicoClient .
- **PubSubConsumer:** O ponto de entrada que aciona o serviço.



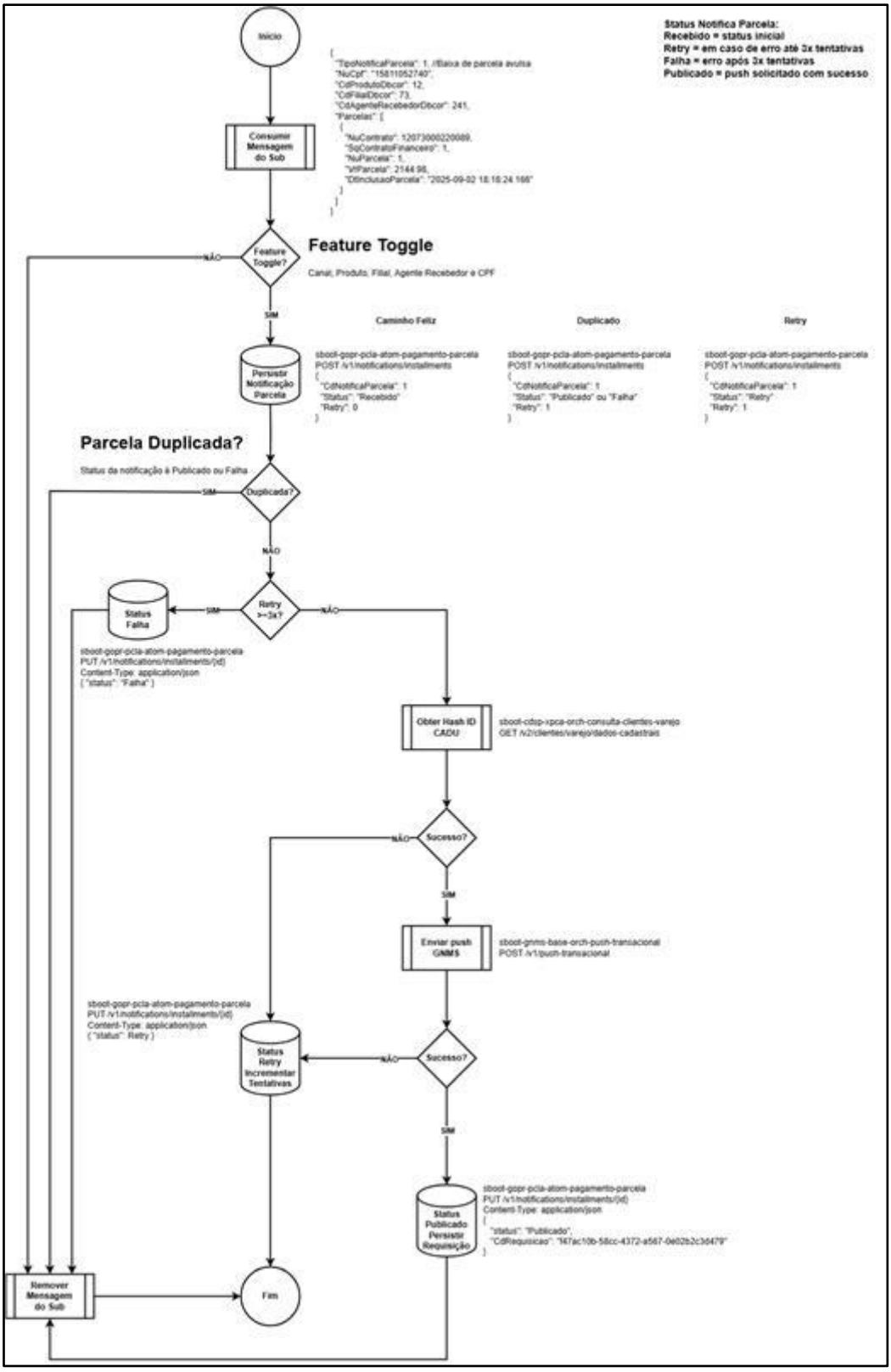
4.4 Visão de Código (Nível 4) - Foco: NotificationService

Este nível detalha o componente **NotificationService**, ilustrando como os padrões de design de código garantem robustez e manutenibilidade.

- **Injeção de Dependência**: A classe **NotificationServiceImpl** (implementação concreta) recebe suas dependências (os *clients*) como **Interfaces** (Portas).
- **Alta Testabilidade**: Esse design permite o uso de *mocks* (ex: Mockito) em testes de unidade (JUnit) para simular o comportamento dos sistemas externos (CRM, ConfigCat). Isso permite testar 100% da lógica de negócio sem depender de rede ou infraestrutura.
- **DTOs (Data Transfer Objects)**: O serviço usa DTOs distintos para traduzir contratos. Ele recebe um **ParcelaPagaDTO**, processa a lógica e cria um **CRMRequestDTO**. Isso impede que mudanças na API do CRM afetem o *core* da lógica de negócio.
- **Inversão de Controle**: O serviço (**INotificationService**) não sabe nada sobre o Pub/Sub. O **PubSubConsumer** é que chama o serviço. Isso permite que o serviço seja reutilizado por outras fontes (como uma API REST) no futuro.



5. Detalhamento Técnico e Fluxos



5.1 Fluxo Principal de Notificação

1. **Ingestão:** O *Batch ETL* identifica pagamentos na base legada e publica no *Pub/Sub*.
2. **Consumo:** O *Orquestrador* consome a mensagem.
3. **Verificação:** Consulta o *ConfigCat* (Feature Toggle) para validar se o cliente/produto está elegível.
4. **Enriquecimento:** Busca o Hash ID do cliente no serviço de *Cadastro de Clientes*.
5. **Persistência Inicial:** Registra a tentativa no *Serviço Atômico*.
6. **Envio:** Envia solicitação para o *CRM* disparar o *push*.
7. **Confirmação:** Atualiza o status no *Serviço Atômico* para "Enviado".

5.2 Estratégia de Feature Toggle

A liberação gradual da funcionalidade será controlada via **ConfigCat**, permitindo segmentação por:

- CPF do cliente.
- Tipo de produto financeiro.
- Filial do banco.

5.3 Estratégia de Dados e Expurgo

- **Retenção:** Os dados operacionais serão mantidos por 6 meses para auditoria.
- **Expurgo:** Após 6 meses, os dados devem ser extraídos para arquivo texto (armazenamento frio) e removidos do banco operacional (não incluído no MVP, mas previsto na arquitetura).

6. Infraestrutura e Operação

6.1 Infraestrutura Cloud (GCP)

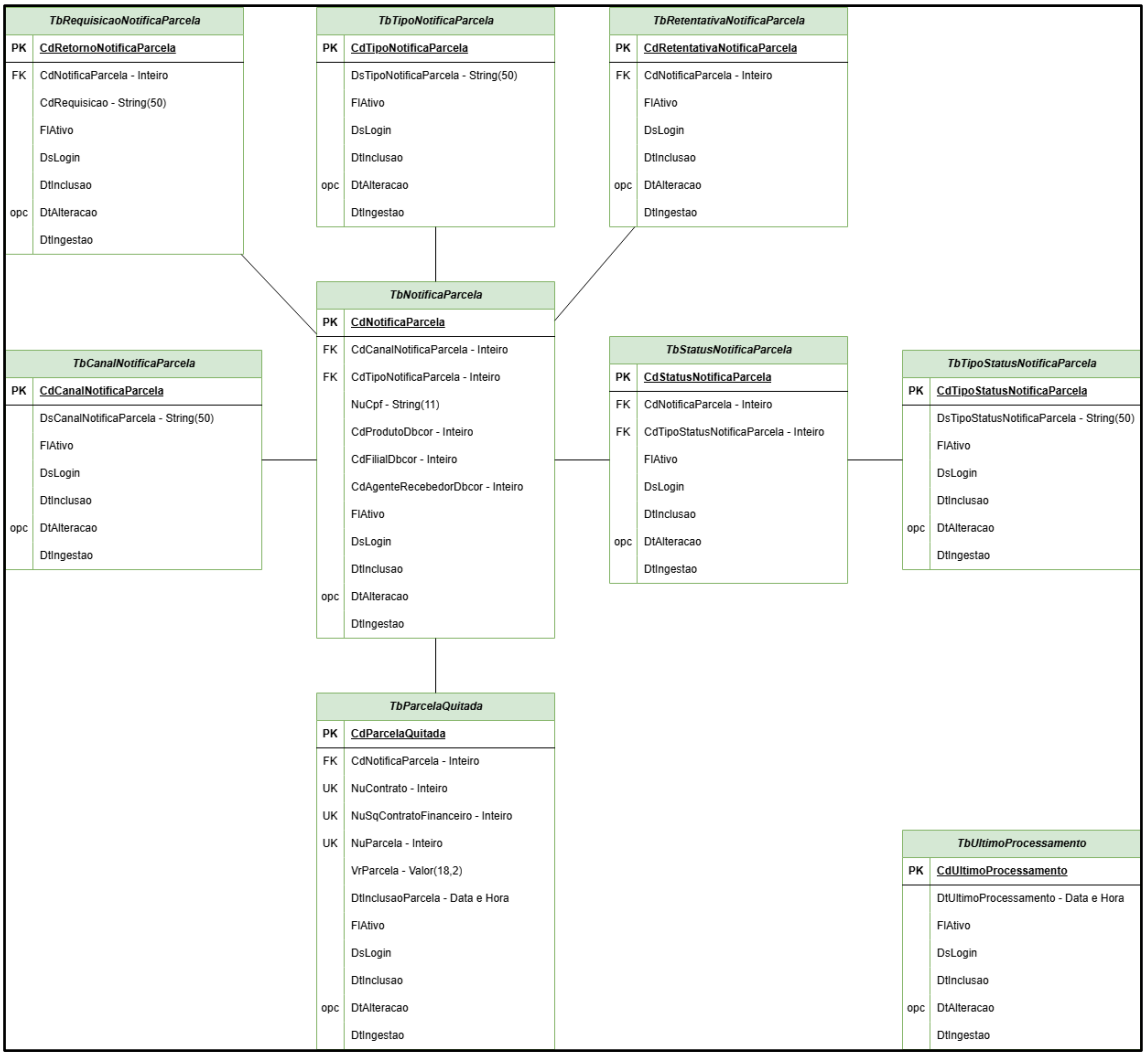
A infraestrutura será provisionada em três ambientes (DES, UAT, PRD) na região southamerica-east1 (São Paulo) para baixa latência.

- **Computação:** GKE (Kubernetes Engine) para orquestração dos contêineres.
- **Banco de Dados:** Cloud SQL para MySQL (Enterprise Plus).
- **Mensageria:** Google Pub/Sub.

6.2 Monitoramento

- **Técnico:** Logs centralizados e métricas de CPU/Memória via GCP.
- **Negócio:** Utilização do Dashboard nativo do CRM para acompanhar: Disparos únicos, Taxa de entrega, Abertura e Erros.

7. Banco de Dados



7.1 Tabelas Funcionais

Tabelas utilizada para persistir os dados significativos, ou seja, com valor para o negócio e que trafegam pela aplicação.

- **TbNotificaParcela**: persiste a solicitação de notificação recebida pela aplicação a partir da leitura do batch na base de dados do sistema legado de recebimentos.
- **TbParcelaQuitada**: persiste as parcelas que foram quitadas na solicitação de notificação recebida.
- **TbStatusNotificaParcela**: persiste o status da solicitação de notificação, mantendo o histórico de mudança de status como eventos.
- **TbRetentativaNotificaParcela**: persiste as retentativas de solicitar o envio do push para o buiding block do CRM.

7.2 Tabelas de Domínio

Tabelas utilizadas no complemento dos cadastros para descrições que explicam o significado dos dados persistidos.

- **TbTipoStatusNotificaParcela**: contém a descrição dos status, são eles recebido, publicado, falha e retry.
- **TbCanalNotificaParcela**: contém os canais de notificação, no MVP será apenas APP. A implementação de novos canais deverá ocorrer no futuro.
- **TbTipoNotificaParcela**: contém os 3 tipos de cenários possíveis de notificação, são eles baixa de parcela avulsa, baixa de múltiplas parcelas e quitação de contrato.

7.3 Tabelas de Controle

Tabelas utilizadas para controle do funcionamento da aplicação em si.

- **TbUltimoProcessamento**: persiste a data e hora da última execução do batch de leitura da base de dados do sistema de pagamentos legado. Representa a fatia de tempo que o sistema está considerando no processamento de pagamentos a cada 10 minutos.

8. Próximos Passos (Roadmap Arquitetural)

1. Implementar a rotina de expurgo de dados (pós-MVP).
2. Integrar com o novo sistema de recebimentos (Cloud Native) assim que disponível, publicando diretamente no tópico do Pub/Sub existente.
3. Expandir canais para WhatsApp e SMS.